Министерство образования Республики Беларусь
Белорусский государственный университет
Механико-математический факультет
Кафедра веб-технологий и компьютерного моделирования

СОГЛАСОВАНО
Заведующий кафедрой
_____ В.М. Волков
«29» декабря 2022 г.

СОГЛАСОВАНО
Декан факультета
_____ С.М. Босяков
«26» января 2023 г.

СОГЛАСОВАНО
Председатель учебно-методической
комиссии факультета
_____ В.Г. Кротов
«26» января 2023 г.

Технологии программирования

Электронный учебно-методический комплекс для специальности:
1-31 03 01 «Математика (по направлениям)»,
направление специальности:
1-31 03 01-02 «Математика (научно-педагогическая деятельность)»

Регистрационный № 2.4.2-24/317

Авторы:
Расолько Г.А., кандидат физ.-мат. наук, доцент;
Кремень Е.В., кандидат физ.-мат. наук, доцент;
Кремень Ю.А., кандидат физ.-мат. наук, доцент.

Рассмотрено и утверждено на заседании Научно-методического совета БГУ 15.02.2023 г., протокол № 5.

Минск 2023

УДК 004.42(075.8)
Р 242

Утверждено на заседании Научно-методического совета БГУ
Протокол № 5 от 15.02.2023 г.

Решение о депонировании вынес:
Совет Механико-математического факультета
Протокол № 5 от 26.01.2023 г.

Авторы:

Расолько Галина Алексеевна, кандидат физико-математических наук, доцент, доцент кафедры ВТиКМ ММФ БГУ;

Кремень Елена Васильевна, кандидат физико-математических наук, доцент, доцент кафедры ВТиКМ ММФ БГУ;

Кремень Юрий Алексеевич, кандидат физико-математических наук, доцент, доцент кафедры ВТиКМ ММФ БГУ.

Рецензенты:

кафедра алгебры, геометрии и математического моделирования Брестского государственного университета имени А.С. Пушкина (заведующий кафедрой Сендер А. Н., кандидат физико-математических наук, доцент);

Бровка Н. В., заведующий кафедрой теории функций ММФ БГУ, кандидат физико-математических наук, доктор педагогических наук, профессор.

Расолько, Г. А. Технологии программирования : электронный учебно-методический комплекс для специальности: 1-31 03 01 «Математика (по направлениям)», направление специальности: 1-31 03 01-02 «Математика (научно-педагогическая деятельность)» / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень ; БГУ, Механико-математический фак., Каф. веб-технологий и компьютерного моделирования. – Минск : БГУ, 2023. – 352 с. : ил., табл. – Библиогр.: с. 351–352.

Электронный учебно-методический комплекс (ЭУМК) по учебной дисциплине «Технологии программирования» предназначен для студентов специальности 1-31 03 01 «Математика (по направлениям)», направление специальности 1-31 03 01-02 «Математика (научно-педагогическая деятельность)». ЭУМК содержит тексты лекций, планы лабораторных занятий, перечень контрольных вопросов, тесты, списки рекомендованной литературы.

ОГЛАВЛЕНИЕ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Электронный учебно-методический комплекс (ЭУМК) по учебной дисциплине «Технологии программирования» создан в соответствии с типовым учебным планом № G31-1-011/пр.тип. от 31.03.2021 г., учебными планами № G31-1-016/уч. от 25.05.2021 г., № G31-1-010/уч. ин. от 31.05.2021 г., № G31-1-015/уч.з. от 31.05.2021 г.

Учебно-методический комплекс преследует **цель** по оказании посильной помощи студентам в усвоении учебного и нормативного материала, сориентировать в подборе специальной литературы для подготовки к лабораторным и практическим занятиям по курсу «Технологии программирования» в соответствии с учебной программой учреждения высшего образования по учебной дисциплине для специальности первой ступени высшего образования 1-31 03 01 Математика (по направлениям), направление 1-31 03 01-02 Математика (научно-педагогическая деятельность). № УД-10897/уч 2022 г.

Задачи учебной дисциплины:

1. Развитие математического, логико-алгоритмического и программистского стилей мышления. Выработка творческого подхода к конструированию алгоритмов с целью развития аналитических и творческих способностей студентов. Формирование практических знаний и умений использования современных методов программирования;
2. Знакомство и работа с абстрактными типами данных;
3. Изучение классических методов разработки алгоритмов;
4. Изучение базовых алгоритмов внутренней и внешней сортировок данных;
4. Знакомство и овладение объектно-ориентированной технологией программирования;
5. Решение задач высшей математики в системах компьютерной алгебры на примере MathCad.

В результате освоения учебной дисциплины студент должен:

знать:

- основы технологий создания программного обеспечения;
- типы и структуры данных, используемые в повседневной практике программирования;
- алгоритмы решения наиболее распространенных классов задач;
- современные информационные технологии;

уметь:

- решать типовые задачи математики и информатики;
- работать на современных вычислительных средствах;
- применять современные информационные технологии и методы реализации решения прикладных задач;

владеть:

- методами программирования задач в различных областях;

•современными методологиями разработки программ.

Дисциплина изучается в 3 и 4 семестрах дневной формы получения высшего образования по специальности 1-31 03 01 Математика (по направлениям), направление специальности 1-31 03 01-02 Математика (Научно-педагогическая деятельность). Всего на изучение учебной дисциплины «Технологии программирования» отведено:

– 210 часов, в том числе 140 аудиторных часов, из них: лекции – 70 часов, лабораторные занятия на персональных компьютерах – 60 часов, управляемая самостоятельная работа – 10 часов. Из них:

– в 3-м семестре: лекции – 36 часов, лабораторные занятия – 30 часов, УСР – 6 часов. Трудоемкость учебной дисциплины составляет 3 зачетные единицы. Форма текущей аттестации – зачет.

– в 4-м семестре: лекции – 34 часа, лабораторные занятия – 30 часов, УСР – 4 часа. Трудоемкость учебной дисциплины составляет 3 зачетные единицы. Форма текущей аттестации – экзамен.

В структуру ЭУМК входит:

1. Теоретический раздел (включает краткий конспект лекций по учебной дисциплине).
2. Практический раздел.
3. Контроль самостоятельной работы студентов (содержит перечень контрольных мероприятий управляемой самостоятельной работы студентов; темы рефератов и вопросы для подготовки к зачету).
4. Вспомогательный раздел (включает список литературы).

Работа студента с ЭУМК должна включать ознакомление с тематическим планом учебной дисциплины, представленным в учебной программе учреждения высшего образования, в которой можно получить информацию о тематике лекций, лабораторных занятий и рекомендуемой литературе. Для подготовки к лабораторным занятиям рекомендуется использовать материалы, представленные в теоретическом и практическом разделах ЭУМК, а также материалы для контроля самостоятельной работы студентов. В ходе подготовки к зачету целесообразно ознакомиться с требованиями к компетенциям по учебной дисциплине, изложенными в учебной программе учреждения высшего образования, а также перечнем вопросов к зачету и экзамену.

Программирование на Turbo Pascal завершается изучением основ методологии объектно-ориентированного программирования (ООП), таких как инкапсуляция, наследование, полиморфизм. Данная тема позволяет обобщить полученные в курсе знания и вывести их на новую более высокую ступень, подготовить студентов к работе с объектно-ориентированными языками программирования, которые они изучают в отдельных курсах.

На втором этапе предлагается изучать фундаментальные алгоритмы и структуры данных. Студентами отрабатываются навыки алгоритмизации задач

по обработке структур данных, в том числе связанных динамических структур: списков, стеков, очередей и деревьев. Из изучаемых фундаментальных алгоритмов отметим алгоритмы сортировки данных, поиска элемента, широкий спектр логико-комбинаторных задач. Важнейшей частью изучения этих тем является сравнение алгоритмов сортировки по нескольким критериям, проведение математического анализа этих алгоритмов с последующей проверкой на компьютере для типовых модельных задач. Анализ алгоритмов сортировки важен для развития аналитических и творческих способностей студентов.

На третьем этапе предлагается изучать математический пакет MathCad. Будущие педагоги знакомятся с возможностями символьных и численных вычислений, визуализацией результатов на примере решения как задач элементарной математики, что позволит им в дальнейшем применить накопленный опыт в школе, так и задач из линейной алгебры, дифференциальных уравнений, численного анализа. Основной упор делается не на использование стандартных возможностей пакета, хотя и этому уделяется достаточное внимание, а на написание собственных программ различных алгоритмов в MathCad, посредством встроенного языка программирования.

1. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

В теоретическом разделе изложен краткий конспект лекций в соответствии с программой дисциплины.

1 СЕМЕСТР

1.1. Технологии реализации алгоритмов

Основные понятия и подходы

Под технологией программирования будем понимать

- совокупность производственных процессов, приводящих к созданию требуемой программной системы, а также
- описание этой совокупности процессов, начиная с момента зарождения идеи этого средства до создания необходимой программной документации.

Каждый процесс этой совокупности базируется на использовании каких-либо **методов** и **средств**, например, компьютера. Тогда говориться о компьютерной технологии программирования.

В историческом аспекте в развитии технологии программирования выделяют следующие этапы:

- **Стихийное программирование** – отсутствие сформулированной технологии, когда программирование было, по сути, искусством. Стихийно использовалась разработка «сверху-вниз» – подход, при котором вначале проектировали и реализовывали сравнительно простые подпрограммы, из которых потом **пытались** построить сложную программу. Развитие программирования шло по пути замены машинных языков ассемблерами, а затем алгоритмическими языками (Fortran, Algol)
- **Структурный подход к программированию**, в основе которого лежит декомпозиция сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм. Программирование осуществлялось «сверху-вниз» и подразумевало представление задачи в виде иерархии подзадач простейшей структуры со своим тщательно проработанным интерфейсом (заголовками подпрограмм). Поддержка принципов структурного программирования была заложена в основу процедурных языков программирования (PL/1, Algol-68, Pascal, C). Появилась и начала развиваться технология модульного программирования. Модули раздельно разрабатывались и компилировались и могли появиться ошибки на этапе эксплуатации таких программных систем по причине ошибок в состыковке модулей. Модули взаимодействуют между собой, обращаясь к ресурсам друг друга.
- **Объектный подход к программированию** сложился с середины 80-х до конца 90-х годов 20-го столетия. Объектный подход предлагал новые способы организации программ, основанные на механизмах инкапсуляции, наследования, полиморфизма. В дальнейшем развитие объектного подхода в технологии программирования привело к созданию сред визуального

программирования. Появились языки визуального объектно-ориентированного программирования (Delphi, C++ Builder, Visual C++, C#).

- **Компонентный подход и Case-технологии** (с середины 90-х годов 20-го века и до наших дней).

Этот подход предполагает построение программного обеспечения из отдельных компонентов – физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через стандартизованные двоичные интерфейсы.

Важнейшая особенность современного этапа технологии программирования – широкое использование компьютерных технологий создания и сопровождения программных систем на всех этапах их жизненного цикла.

Технологии программирования развивались вместе с развитием ЭВМ и языков программирования.

Стиль программирования

Стиль программирования — это способ построения программ, основанный на определенных принципах программирования, и выбор подходящего языка, который делает понятными программы, написанные в этом стиле.

Каждый стиль программирования имеет свою концептуальную базу.

Процедурное (императивное) программирование (Basic, Pascal) является отражением архитектуры традиционных ЭВМ, которая была предложена фон Нейманом в 40-х годах.

Программа на процедурном языке программирования состоит из последовательности операторов (инструкций), задающих процедуру решения задачи. Основным является оператор присваивания, служащий для изменения содержимого областей памяти.

Концепция памяти как хранилища значений, содержимое которого может обновляться операторами программы, является фундаментальной в императивном программировании.

Таким образом, с точки зрения программиста имеются программа и память, причем первая последовательно обновляет содержимое последней.

Процедурные языки характеризуются следующими особенностями:

- необходимостью явного управления памятью, в частности, описанием переменных;
- малой пригодностью для символьных вычислений;
- отсутствием строгой математической основы;
- высокой эффективностью реализации на традиционных ЭВМ.

Функциональное программирование (Clojure, Common Lisp, F#) предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Программа представляет собой совокупность описаний функций и выражения, которые необходимо вычислить. Функциональное

программирование не использует концепцию памяти как хранилища значений переменных.

Логическое программирование (Prolog) – основано на теории и аппарате математической логики. Логические программы, в принципе, имеют небольшое быстродействие, так как вычисления осуществляются методом проб и ошибок, поиском с возвратами к предыдущим шагам. Программы пишутся не в виде последовательности инструкций, а в виде множества фактов и правил, а процесс выполнения программы сводится к выводу нужных результатов из этого множества. Логическое программирование относится к декларативному программированию, поскольку программа на нём скорее описывает свойство задачи, нежели алгоритм её решения.

Объектно-ориентированное программирование (C++ и Java) - основными концепциями являются понятия объектов и классов. Для описания объектов служат классы. Класс определяет свойства и методы объекта, принадлежащего этому классу. Соответственно, любой объект можно определить как экземпляр класса. Программирование заключается в выборе имеющихся или создании новых объектов и организации взаимодействия между ними. При создании новых объектов свойства объектов могут добавляться или наследоваться от объектов-предков. В процессе работы с объектами допускается полиморфизм — возможность использования методов с одинаковыми именами для обработки данных разных типов.

Модульное программирование предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные, в отдельно компилируемые модули (библиотеки подпрограмм), например, модуль графических ресурсов. Связи между модулями осуществляются через специальный интерфейс, в то время как доступ к реализации модуля запрещен. Эту технологию поддерживают современные версии языков Pascal и C (C++), языки Ада и Modula.

Параллелизм. Есть задачи, в которых автоматические системы должны обрабатывать много событий одновременно. В других случаях потребность в вычислительной мощности превышает ресурсы одного процессора. В каждой из таких ситуаций естественно использовать несколько компьютеров для решения задачи или задействовать многозадачность на многопроцессорном компьютере.

Наряду с указанными выше выделяют методы:

- **Декларативное программирование** – метод, предназначенный для решения задач искусственного интеллекта, когда программа описывает логическую структуру решения задачи, указывая преимущественно, что нужно сделать, не вдаваясь в детали. Используются языки программирования типа Пролог.
- **Эвристическое программирование** – метод, основанный на моделировании мыслительной деятельности человека.

О языках программирования

Под системой программирования понимают совокупность языка программирования и виртуальной машины, обеспечивающей выполнение на реальной машине программ, составленных на этом языке.

Языком программирования называют систему обозначений, служащую в целях точного описания алгоритмов для ЭВМ или по крайней мере достаточную для автоматического нахождения такого алгоритма.

Первым компьютерам приходилось программировать двоичными машинными кодами. Однако программировать таким образом – достаточно трудоемкая и сложная задача.

Для упрощения этой задачи стали появляться языки программирования низкого уровня, которые позволяли задавать машинные команды в более понятном для человека виде. Для преобразования их в двоичный код были созданы специальные программы – трансляторы.

Если проследить историю развития языков программирования, то можно выделить две основные тенденции: перемещение акцентов от программирования отдельных деталей к программированию более крупных компонент; развитие и совершенствование языков программирования высокого уровня.

Большинство современных коммерческих программных систем больше и существенно сложнее, чем были их предшественники даже несколько лет тому назад.

Этот рост сложности вызвал большое число прикладных исследований по *методологии проектирования*, особенно, по декомпозиции, абстрагированию и иерархиям.

Создание более выразительных языков программирования пополнило достижения в этой области. Возникла тенденция перехода от языков, указывающих компьютеру, что делать (*императивные* языки), к языкам, описывающим ключевые абстракции проблемной области (*декларативные* языки).

Различают *четыре поколения* в зависимости от того, какие языковые конструкции впервые в них появились:

• **Первое поколение** (1954-1958) (Математические формулы)

FORTRAN I, ALGOL-58 и др.

• **Второе поколение** (1959-1961) (Подпрограммы, раздельная компиляция, блочная структура, типы данных, описание данных, работа с файлами и т.п.)

FORTRAN II ALGOL-60 COBOL

• **Третье поколение** (1962-1970)

PL/I=(FORTRAN+ALGOL+COBOL), ALGOL-68, Pascal, Simula (Классы, абстрактные данные)

• **Потерянное поколение** (1970-1980)

Много языков созданных, но мало выживших.

Первое поколение языков высокого уровня было шагом, приближающим программирование к предметной области и удаляющим от конкретной машины.

Во втором поколении языков основной тенденцией стало развитие алгоритмических абстракций. В это время мощность компьютеров быстро росла, а компьютерная индустрия позволила расширить области их применения, особенно в бизнесе.

Особенности конкретных компьютерных архитектур в языках не учитываются, поэтому созданные программы легко переносятся с компьютера на компьютер. Разрабатывать программы на таких языках гораздо проще и ошибок допускается меньше. Значительно сокращается время разработки программы. Недостатком языков высокого уровня является больший размер программ по сравнению с программами на языке низкого уровня. Поэтому в основном языки высокого уровня используются для разработок программного обеспечения компьютеров и устройств, которые имеют большой объем памяти. А разные подвиды ассемблера применяются для программирования других устройств, где критичным является размер программы.

В конце 60-х годов с появлением транзисторов, а затем интегральных схем, стоимость компьютеров резко снизилась, а их производительность росла почти экспоненциально. Появилась возможность решать все более сложные задачи, но это требовало умения обрабатывать самые разнообразные типы данных. Такие языки как ALGOL-68 и затем Pascal стали поддерживать абстракцию данных. Программисты смогли описывать свои собственные типы данных. Это стало еще одним шагом к предметной области и от привязки к конкретной машине.

70-е годы знаменовались безумным всплеском активности: было создано около двух тысяч различных языков и их диалектов. Неадекватность более ранних языков написанию крупных программных систем стала очевидной, поэтому новые языки имели механизмы, устраняющие это ограничение. Лишь немногие из этих языков смогли выжить.

Таким образом получили языки Smalltalk (новаторски переработанное наследие Simula), Ada (наследник ALGOL-68 и Pascal с элементами Simula, Alphard и CLU), CLOS (объединивший Lisp, LOOPS и Flavors), C++ (возникший от брака C и Simula) и Eiffel (произошел от Simula и Ada).

Наибольший интерес для дальнейшего изложения представляет класс языков, называемых объектными и объектно-ориентированными, которые в наибольшей степени отвечают задаче объектно-ориентированной декомпозиции программного обеспечения.

Топология языков

Говоря "топология" мы имеем в виду основные элементы языка программирования и их взаимодействие.

Топология языков первого и начала второго поколения

Можно отметить, что для таких языков, как FORTRAN и COBOL, основным строительным блоком является **подпрограмма**. Программы, реализованные на таких языках, имеют относительно простую структуру, состоящую только из глобальных данных и подпрограмм. Ошибка в какой-либо части программы может иметь далеко идущие последствия, так как область данных открыта всем

подпрограммам. Это угрожает надежности системы и определенно снижает ясность программы.

Топология языков позднего второго и раннего третьего поколения

Начиная с середины 60-х годов стали осознавать роль подпрограмм как важного промежуточного звена между решаемой задачей и компьютером. Были разработаны языки, поддерживавшие разнообразные механизмы передачи параметров, заложены основания структурного программирования, что выразилось в языковой поддержке механизмов вложенности подпрограмм и в научном исследовании структур управления и областей видимости. Возникли **методы структурного проектирования**, стимулирующие разработчиков создавать большие системы, используя подпрограммы как готовые строительные блоки. Архитектура языков программирования этого периода представляет собой вариации на темы предыдущего поколения.

Топология языков конца третьего поколения

Разрастание программных проектов означало увеличение размеров и коллективов программистов, а, следовательно, необходимость независимой разработки отдельных частей проекта. Ответом на эту потребность стал отдельно компилируемый модуль, который сначала был просто более или менее случайным набором данных и подпрограмм. В такие модули собирали подпрограммы, которые, как казалось, скорее всего, будут изменяться совместно, и мало кто рассматривал их как новую технику абстракции. В большинстве языков этого поколения, хотя и поддерживалось **модульное программирование**, но не вводилось никаких правил, обеспечивающих согласование интерфейсов модулей, что могло привести к непоправимым аварийным ситуациям.

Топология объектных и объектно-ориентированных языков

Основным элементом конструкции в указанных языках служит Модуль, составленный из логически связанных классов и объектов, а не подпрограмма, как в языках первого поколения.

Уменьшена или отсутствует область глобальных данных. Данные и действия организуются так, что основными логическими строительными блоками наших систем становятся классы и объекты, а не алгоритмы.

Методы структурного проектирования помогают упростить процесс разработки сложных систем за счет использования алгоритмов как готовых строительных блоков. Аналогично, методы объектно-ориентированного проектирования созданы, чтобы помочь разработчикам применять мощные выразительные средства объектного и объектно-ориентированного программирования, использующего в качестве блоков классы и объекты.

Объектно-ориентированный анализ и проектирование отражают эволюционное, а не революционное развитие проектирования; новая методология не порывает с прежними методами, а строится с учетом предшествующего опыта.

О языке Free Pascal

Язык Pascal был первоначально разработан Никлаусом Виртом в 1970 году. С того дня он значительно эволюционировал, с большим количеством вкладов различными конструкциями компилятора (Особенно: Borland).

Основные элементы были сохранены на протяжении многих лет:

• Легкий синтаксис, довольно многословный, но все же легкий для чтения. Идеально подходящий для обучения.

• Строго типизированный.

• Процедурный.

• Не чувствительный к регистру.

• Позволяет вложенные процедуры.

• Встроенные процедуры для обработки ввода/вывода.

Компиляторы Turbo Pascal и Delphi ввели различные особенности в язык Pascal, наиболее заметными являются более легкая строковая обработка и объектная ориентированность. Компилятор Free Pascal первоначально эмулировал большую часть Turbo Pascal, а позже и Delphi. Он эмулирует поведение этих компиляторов в соответствующих режимах: некоторые функции доступны только в том случае если компилятор переключается на соответствующий режим. Когда для определенных функций это необходимо, нужно использовать переключатель командной строки -M или директиву {$MODE} указанную в исходном тексте. Более подробную информацию о различных режимах можно найти в руководстве пользователя и руководстве программиста.

Как выбрать первый язык программирования

Разработчики на Python хвалятся тем, как быстро пишут код. Программисты на C++ — что их код очень производительный. Те, кто используют Java, говорят, как важна кроссплатформенность. У каждого языка есть свои преимущества и недостатки. Один язык не может быть хорош для всего.

Языки программирования похожи друг на друга, поэтому чем больше вы их знаете, тем проще учить новые. Однако всегда важна цель — для чего каждый из них осваивается. Как и любым инструментом, языком нужно пользоваться на практике, иначе знания быстро забудутся. Сам процесс изучения нового порой помогает лучше понять другие технологии.

Новички ещё слишком мало знают, чтобы понять, что им нужно от языка. Поэтому выбирать нужно не язык, а то, *чем* вы хотите заниматься.

Многие языки в первую очередь затачиваются под решение определённых проблем или под определённые сферы:

- Быстро создать сайт — PHP или Python.
- Создать игру — C++ или C#.
- Веб-систему для банка — Java, C# или C++.
- Красивый интерфейс для сайта — HTML, CSS и JavaScript.
- Приложение для Android — Java или Kotlin.

- Приложение для iOS или Mac OS — Objective-C или Swift.

Парадигмы программирования

Парадигма простыми словами – это давно признанные истины, не подлежащие никакому сомнению.

Парадигма программирования – *система идей и понятий*, определяющих фундаментальный стиль программирования.

Императивное программирование — это парадигма, в которой задаётся последовательность действий, необходимых для получения результата. В нём используются переменные, операторы присваивания и составные выражения.

Первые языки программирования компьютеров (машинный, ассемблер, фортран, алгол, кобол) были императивными в силу простоты подхода.

Декларативное программирование — это парадигма программирования, в которой задаётся спецификация решения задачи: описывается, что представляет собой проблема и ожидаемый результат, но без описания способа достижения этого результата.

Чем отличается императивное программирование от декларативного?

Императивный стиль — это такой стиль программирования, при котором вы описываете, **как** добиться желаемого результата. Например, пишем:

— поставь сковородку на огонь;

— возьми два яйца (куриных);

— нанеси удар ножом по каждому;

— вылей содержимое на сковородку;

— выкинь скорлупу;…

Это что ни на есть декларативный стиль, но при этом с примесью императивного. Ну хотя бы потому, что получатель должен знать, что такое сковорода и яйца.

Декларативный стиль — это такой стиль, в котором вы описываете, **какой именно результат** вам нужен. Тут просто пишем:

— приготовь яичницу

И получатель такого сообщения уже сам разбирается, какие шаги для этого надо предпринять.

Это пример императивного интерпретатора декларативного языка. Каким образом будет получена не важно. Вы показываете пальцем и говорите: здесь должна быть яичница. Интерпретатор, допустим официант, может заказать ее в другом ресторане, на кухне, сделать сам, собрать из атомов, разогреть вчерашнюю, главное, чтобы она соответствовала декларативному описанию.

Практически все современные языки программирования общего назначения высокого уровня, за исключением некоторых функциональных, относятся к императивным языкам.

Императивные языки, такие, как Java, Python, JavaScript, C, C++ занимают доминирующее положение в индустрии ПО, соответственно императивное программирование — самое распространённое. Смысл его в том, что императивная программа содержит прямые указания, что должен сделать

компьютер и в каком порядке должны выполняться инструкции. Этот подход легко понять программисту, а компилятору — легко породить достаточно эффективный код.

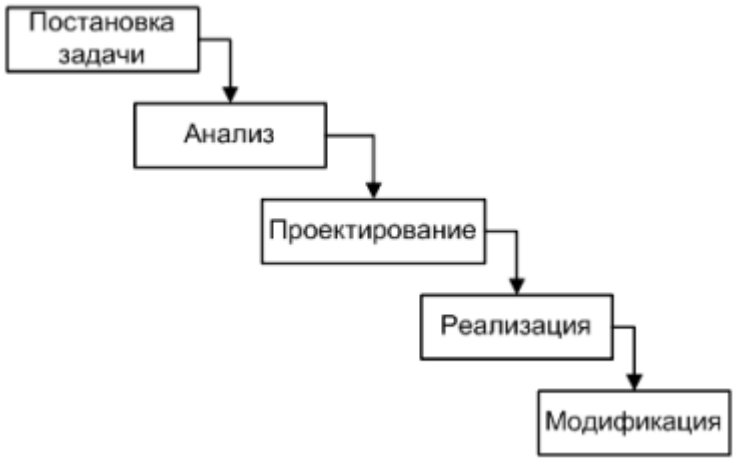
Самый популярный язык РСУБД SQL так же является декларативным. На нём описывается конечный результат, а способ его получения генерируется сервером СУБД исходя из множества факторов.

Жизненный цикл программного продукта

Под жизненным циклом программного продукта (software life cycle) понимают весь период его разработки и эксплуатации, начиная от момента возникновения замысла программного продукта, определения его целевого назначения и заканчивая выводом его из эксплуатации. Понятие «жизненный цикл» охватывает весь процесс создания и использования программного продукта. Этот процесс может быть организован по-разному в зависимости от особенностей коллектива разработчиков и разрабатываемого программного продукта.

В настоящее время можно выделить следующие основные стратегии конструирования программного продукта.

• **Каскадный (или водопадный) подход**. При таком подходе процесс разработки состоит из цепочки этапов, выполняемых последовательно друг за другом. На каждом этапе создается документация, используемая на последующем этапе. Это классический подход к созданию программного продукта.



• **Итеративные (или инкрементальные) модели**. Процесс разработки заключается в разбиение создаваемой программной системы на набор частей. Каждая часть реализуется путем нескольких последовательных проходов последовательности всех работ. На первой итерации разрабатывается независимая часть системы. При этом чаще всего проводится полный цикл работ. Затем оцениваются полученные результаты. Следующая итерация заключается либо в исправление недочетов первой части, либо в проектировании следующего фрагмента, зависящего от первого, либо добавлением новых функций. В результате на каждой итерации можно анализировать промежуточные результаты работ, учитывать мнение конечных пользователей и вносить изменения на следующих итерациях. Каждая итерация в процессе своего

выполнения может содержать полный цикл видов деятельности, начиная от анализа требований.

• Продолжением идеи итераций является **спиральная модель жизненного цикла** программного обеспечения. В данной модели разработки каждая итерация начинается с анализа целей, разработки основных альтернатив, определения необходимых ограничений, оценки рисков и т. д. Результатом каждой итерации будет оценка проведенных в ее рамках работ. Следует отметить общую структуру действий на каждой итерации — планирование, определение задач, ограничений и вариантов решений, оценка предложенных решений и рисков, выполнение основных работ итерации и оценка их результатов. Название спиральной эта модель получила из-за изображения хода работ в «полярных координатах», в которых угол соответствует выполняемому этапу в рамках общей структуры итераций, а удаление от начала координат — затраченным ресурсам.

• **Исследовательское программирование**. Этот подход предполагает быструю (насколько это возможно) реализацию рабочих версий программ, выполняющих основные функции. После опытного применения созданных программ переходят к их модификации с целью устранения недочетов и приближении их к реальным требованиям пользователей. Такой подход

соответствовал ранним этапам развития программирования. В настоящее время этот подход применяется для разработки систем искусственного интеллекта.

• **Прототипирование**. Этот подход моделирует начальную фазу работы вплоть до создания рабочих версий программ с целью установить (уточнить) требования заказчика к программному продукту.

Основная цель создания прототипа — снять неопределенности в требованиях заказчика. В дальнейшем обычно следует разработка программы по установленным требованиям в рамках какого-либо другого подхода (например, классического).

Достоинством данного подхода является определение полных требований к программному продукту.

Недостатки: когда заказчик видит уже работающую версию программного продукта, возникает соблазн принять его за готовую версию и оставить нерешенными вопросы качества и удобства сопровождения. При этом заказчик не может адекватно оценить объем будущих работ и требует получить рабочий продукт в нереально короткие сроки.

Следует также отметить, что для быстрого получения работающего макета программист может идти на определенные компромиссы, например, использование не самых подходящих языков программирования и операционных систем, или для демонстрации функциональных возможностей может применяться неэффективный алгоритм.

• **Сборочное программирование**. Этот подход заключается в конструировании программ из стандартных компонент, которые уже существуют в разработанных библиотеках.

Эти библиотеки компонент достаточно хорошо разработаны и многократно используются при создании сложных программных систем, имеющие типовые фрагменты обработки данных.

Такой процесс разработки состоит скорее из сборки программ из компонент, чем из их программирования.

Весь жизненный цикл программного продукта можно разбить на три крупные фазы:

1) разработка;

2) эксплуатация;

3) сопровождение и разработка новых версий.

В фазе разработки программный продукт разрабатывается, документируется, тестируется и выпускается.

В фазе эксплуатации разработанный программный продукт используется на практике конкретными потребителями, при этом могут выявляться ошибки, недоработки, изменяться требования к пользовательскому интерфейсу.

В фазе сопровождения программный продукт изменяется в соответствии с поступившими требованиями, и появляются его принципиально новые версии.

Фазу разработки обычно разделяют на следующие логические этапы:

1) анализ требований;

2) проектирование;

3) программирование (кодирование);

4) отладка и тестирование;

5) документирование;

6) выпуск.

На этапе анализа требований для заказчика обосновывается необходимость нового программного продукта, выявляются наиболее общие требование к нему. Результатом системного анализа является выработка спецификации требований на программный продукт, содержащая указанные требования в формальном виде.

Этот документ является описанием поведения программы с точки зрения ее будущего пользователя и с фиксацией требований относительно его качества. По единой системе программной документации такая спецификация называется техническим заданием.

На этапе проектирования сформулированные общие требования к будущему программному продукту последовательно реализуются в рабочий проект. Он в деталях описывает структуру программной системы, применяемые форматы данных и алгоритмы, внешний вид интерфейса пользователя. Результатом проектирования является технический проект.

Этап кодирования при наличии детального технического проекта является достаточно рутинным. По сути, кодирование является автоматическим процессом реализации предложенных алгоритмов обработки на конкретном языке программирования с использованием конкретной методологии и инструментария.

Результатом данного этапа являются программы в исходном тексте и выполняемые файлы.

Этап кодирования сопровождается процессом документирования.

Этапы проектирования и кодирования часто перекрываются, иногда довольно сильно.

Этап отладки и тестирования предназначен для выявления и устранения ошибок и недочетов в программах. Результатом данного этапа являются отлаженные программы, для которых тестированием установлено (насколько это возможно) соответствие запрашиваемым требованиям.

На этапе документирования к созданной программной системе подготавливается пакет документации, описывающей создаваемый программный продукт. Каждый документ ориентируется на конкретный тип пользователей: конечных пользователей, системных администраторов, прикладных программистов и т. д.

На этапе выпуска программный продукт подвергается итоговым испытаниям по утвержденной методике. После этого программный комплекс продается или внедряется на фирме, заказавшей его.

Тестирование и отладка программ

Тестирование программ осуществлялось еще при их написании в машинных кодах, но к концу 60-х годов 20-го века эта работа стала упорядочиваться, и к середине 70-х были сформулированы основные приемы организации тестирования отдельных программ и программных комплексов. Эти методы используются и в настоящее время.

Методы тестирования включают в себя модульное тестирование, внешнее тестирование, нисходящее и восходящее тестирование, анализ передачи управления в программе при тестировании, принципы проектирования и разработки надежных программных систем.

В это же время появилась потребность в теории доказательства правильности программ. Был предложен следующий подход: если есть формальное описание семантики всех конструкций языка программирования, то возможно на основе анализа текста строго математически вывести заключение о правильности или неправильности программы.

Отсюда следует, что сначала необходимо создать строгое описание синтаксиса языка и его семантики (смысла его конструкций).

Для описания синтаксиса языка проблема была решена с помощью использования формальной теории грамматики языка Хомского и способа записи грамматики Бэкуса — Каура. Для описания семантики языка было предложено несколько методов описания: W-грамматики, аксиоматический подход, денотационный метод, атрибутные грамматики, Венский метод описания и ряд других. С помощью этих методов было описано несколько реальных языков программирования: PL/1, Algol-68, Pascal.

Методы доказательства правильности программы продолжали развиваться, и они позволяли доказать корректность небольших программ, сгенерированных для тестирования данной теории. Но прежде всего необходимо было четко сформулировать понятие «корректная» программа, и, с другой стороны, как оценить сложность реальных программ. Для большинства реальных программ строгое описание того, что программа должна делать, существенно больше по объему самой программы и требует очень высокой математической квалификации программиста. Для большого количества программ такое подробное описание в принципе невозможно. При разработке и использовании программного продукта мы имеем дело с многократно преобразованием (переводом) информации из одной формы представления в другую.

Как показывает опыт программирования, несмотря на тщательное проведение этапов проектирования и использование современных технологий программирования, не удается разработать полностью безошибочную программу. Основными активными методами поиска и устранения ошибок являются тестирование и отладка.

Тестирование — это процесс выявления имеющихся в программе ошибок, а отладка — это процесс их устранения.

При тестировании проверяется, работает ли программа и все ее ветви в соответствии с заявленными требованиями.

Чтобы убедиться в правильности функционирования программы и обеспечить полный и эффективный контроль выполнения всех ее ветвей, сначала выбирается стратегия тестирования.

У процесса тестирования программы есть свои особенности:

- не существует эталона программы, с которым можно сравнить результаты тестирования;
- не существует теста для полностью исчерпывающей проверки;
- отсутствуют формализованные критерии процесса тестирования;
- обычно пишутся тесты для поиска ошибок в программе, а не для доказательства корректности ее работы;
- обычно программу тестирует программист, который ее написал.

Наиболее характерными объектами тестирования являются: требования и спецификации, отдельные программные модули, группы программ, законченные функциональные задачи.

Существуют три основных подхода к тестированию программ: нисходящее, восходящее и раздельное. Они отличаются очередностью написания отдельных модулей и процедурой передачи данных тестируемым компонентам. Как и при проектировании программ, наибольший эффект достигается при совместном использовании этих методов.

При стратегии *восходящего тестирования* сначала обычно пишутся и тестируются физические модули нижнего уровня, затем переходят к вызывающим модулям и так вплоть до главного модуля.

Основные трудности такого подхода состоят в необходимости постоянного обновления тестов при подсоединении каждого нового модуля более высокого уровня. При этом достоинством можно считать, что все модули нижнего уровня тестируются детально и независимо, это отлавливает значительное количество ошибок в них. Главный модуль (модуль верхнего уровня иерархии) программируется и тестируется последним.

Передача данных и имитация вызывающих модулей выполняется при использовании специальных программ-тестировщиков. Эта программа имитирует работу вызывающих модулей и обеспечивает передачу данных в тестируемый физический модуль и из него. Все исходные, промежуточные и конечные данные выводятся на печать. Возможна реализация автоматической проверки полученных и ожидаемых результатов.

Такой подход ориентирован на испытание программ модульной структуры и требует работы целой команды квалифицированных программистов.

Алгоритм *нисходящего тестирования* предписывает начинать проверку с главного модуля, к которому последовательно подключаются модули более низких иерархических уровней, при этом активно используются программы-заглушки. При использовании этого подхода есть возможность сохранения исходных тестов и их дополнение по мере подключения новых модулей, т. е. с самого начала тестирования могут использоваться реальные исходные данные, которые расширяются по мере работы. Недостаток: более сложный, чем в предыдущем подходе, поиск ошибок в подключаемых модулях.

Раздельное тестирование — этот метод применяется только для раздельно компилируемых физических модулей. Относится к приемам восходящего тестирования. Он применяется в случаях, когда программу невозможно полностью описать или когда существует критический модуль, на функционирование которого накладываются определенные ограничения, и он должен быть испытан в самом начале тестирования. В этом случае процедура тестирования разбивается на два этапа:

- независимая разработка каждого физического модуля с имитацией вызывающего модуля и использованием модулей-заглушек;
- совместное редактирование связей уже проверенных модулей и их комплексное тестирование.

Проектирование тестов следует начинать сразу же после создания документации на программный продукт. Существуют два основных подхода к выбору стратегии проектирования тестов.

Метод «черного ящика» заключается в том, что тесты проектируются только на основании изучения внешнего описания программного продукта, описания его архитектуры и отдельных модулей. Внутренняя структура модулей при этом никак не учитывается. Фактически такой подход заключается в полном переборе входных данных, так как в противном случае некоторые ветви алгоритма могут не работать, и это не определится при пропуске теста. Эта ситуация ведет к тому, что не все ошибки будут выявлены. Однако тестирование полным набором входных данных практически неосуществимо и требует огромного времени.

Метод «белого ящика» заключается в том, что тесты проектируются на основании изучения логической структуры программы с целью протестировать все возможные пути выполнения. Следует учесть, что большинство программ содержит циклы с переменным числом повторений и сложную логику. Следовательно, различных ветвей выполнения программы может оказаться достаточно много, и их тестирование также будет практически неосуществимо.

Для выбора оптимальной стратегии проектирования тестов можно использовать следующий принцип: для каждого программного модуля, входящего в состав программного продукта, необходимо проектировать собственные тесты для обнаружения в нем ошибок. Это будет соответствовать требованиям разработки надежных программ.

Кроме этого, необходим хотя бы один тест:

- на каждую область и на каждую границу изменения какой-либо входной величины;
- на каждую особую (исключительную) ситуацию, указанную в спецификациях.

Вообще, тестирование программ не дает гарантий их качества, так как невозможно однозначно гарантировать отсутствие ошибок в реальной системе.

Математическое обоснование надежности программ основано на формальной *верификации,* которая посредством формального доказательства позволяет устанавливать присутствие требуемых свойств программы для всех допустимых этой программой выполнений.

Основой верификации является логика, формальный язык логики, а также формальные модели и методы. Верификация программ обычно сводит анализ их свойств к доказательству истинности условий корректности в виде логических формул. При этом исследуется (верифицируется) не сама программа, а ее спецификация (формальная модель).

Примером распространенных формальных моделей являются сети Петри. Несмотря на результаты применения верификации, практическое значение для обоснования надежности программного продукта имеет тестирование. Это связано с трудоемкостью и высокими затратами процесса верификации.

Отладка — это процесс исправления ошибок, обнаруженных на этапе тестировании программы. Обычно отладку представляют в виде циклического повторения трех процессов: тестирования, которое позволяет выявить ошибку, локализация ошибки и исправление программы (и документации).

Отладка = Тестирование + Поиск ошибок + Редактирование.

Если в результате тестирования полученные результаты отличаются от эталонных (ожидаемых), то необходимо определить местоположение ошибки. Для этого можно использовать контрольные точки и слежение за значениями переменных.

Во многих системах программирования на языках высокого уровня перечисленные выше возможности включены в отладчики, позволяющие вести отладку в интерактивном режиме.

Известен феномен — по мере роста числа обнаруженных и исправленных ошибок в программном продукте растет также относительная вероятность существования в нем необнаруженных ошибок. Поэтому используются следующие принципы тестирования:

Принцип 1. Тестирование является основной задачей разработки программного продукта, поэтому ее необходимо поручать квалифицированным программистам; нежелательно тестировать свою собственную программу.

Принцип 2. Необходимо разрабатывать тесты, для которых высока вероятность обнаружить ошибку.

Принцип 3. Необходимо сгенерировать тестовые наборы как для правильных, так и для неправильных исходных данных.

Принцип 4. Рекомендуется документировать результат каждого теста для дальнейшего детального анализа.

Принцип 5. Каждый модуль подключается к программе только один раз.

Принцип 6. Если в программу были внесены изменения (например, в результате устранения ошибки), проводите заново все тесты, связанные с проверкой ее работы или ее взаимодействия с другими программами.

Классификация ошибок

Для каждого языка программирования существует свое характерное подмножество ошибок. Рассмотрим классификацию ошибок с точки зрения причины их появления:

1. **Синтаксические ошибки**. В основном это ошибки в ключевых словах и конструкциях языка программирования. Легко выявляются транслятором, их устранение не вызывает особых трудностей.

2. **Опечатки**. Как правило, они вызваны невнимательностью программиста при механическом наборе или редактировании исходного текста (ошибка в операции, имя не той переменной и т. д.). В результате появляется синтаксически правильный, но абсолютно неверный логически участок программы.

3. **Ошибки реализации алгоритма**. Это ошибки, вызванные неверным программированием при верном алгоритме. Этот тип ошибок, по мнению многих авторов, является самым распространенным и наиболее трудно классифицируемым.

4. **Ошибки алгоритма**. Это логические ошибки в применяемом методе, которые приводят к ошибочной работе программы при правильной программной реализации. Такие ошибки очень трудно найти, так как предположение о их наличии делается обычно после проверки всех остальных альтернатив. К этому типу ошибок относят также отсутствие в алгоритме учета ограничений реализуемого им метода.

Ограничения любого математического метода хорошо описаны в специализированной литературе, но часто существуют попытки применить его без учета его особенностей.

5. **Ошибки метода**. Каждый используемый в программе метод должен сопровождаться оценкой вычислительных погрешностей и перечнем ограничений, снижающих его универсальность.

Документирование и стандартизация

При разработке программного продукта создается большой объем документации. Она используется как средство передачи информации между разработчиками и как средство описания информации, которая необходима пользователям для применения программы. Также программная документация может быть использована для тестирования программы.

Документирование должно начинаться одновременно с разработкой продукта.

Существуют следующие основные программные документы:

Текст программы — запись программы с необходимыми комментариями.

Описание программы — сведения о логической структуре и функционировании программы.

Программа и методика испытаний — требования, подлежащие проверке при испытании программы, а также порядок и методы их контроля.

Техническое задание — описанием поведения программы с точки зрения ее будущего пользователя и с фиксацией требований относительно его качества.

Пояснительная записка — содержит общее описание алгоритма, часто в виде граф-схем, схему функционирование и взаимосвязи программных модулей, а также обоснование принятых технических и экономических решений.

Раздел эксплуатационных документов включает в себя:

Руководство пользователя — сведения об области назначения программы, области ее применения, используемых при реализации методах, ограничениях алгоритма, конфигурации требуемых технических средств; описание интерфейса пользователя и допустимых к использованию функций.

Руководство системного администратора — сведения для обеспечения установки, функционирования и настройки программ на условиях конкретного применения.

Интерфейсом пользователя называют совокупность способов и правил взаимодействия программы с пользователем.

1.2. Версии реализации и среды разработки Pascal

Содержание

- Отличительные особенности Free Pascal.
- Данные языка Free Pascal.
- Символьные данные Free Pascal.
- Строки символов.
- Преобразование строк.
- Новые функции преобразования числовых данных.
- Массивы в языке Free Pascal.
- Базовые операторы языка.

Отличительные особенности Free Pascal

Существенные отличия диалекта Turbo Pascal и Free Pascal наблюдаются в основном при работе с динамическими массивами и, в некоторой мере, при работе с графикой, потому что Free Pascal поддерживает большее количество видеоадаптеров и может работать с гораздо лучшим разрешением экрана.

К значительным отличиям следует также отнести поддержку Free Pascal перегрузки арифметических операторов (+, -, *, **, /, **div**, **mod**), операторов сравнения (<, >, =, >=, <=) и оператора присваивания :=, поддержку операторов присваивания с выполнением арифметической операции в стиле Си (+=, -=, *=, /=).

Комментарии

В Free Pascal можно использовать два разделителя // для комментария последующего за ними однострочного текста.

Данные языка Free Pascal

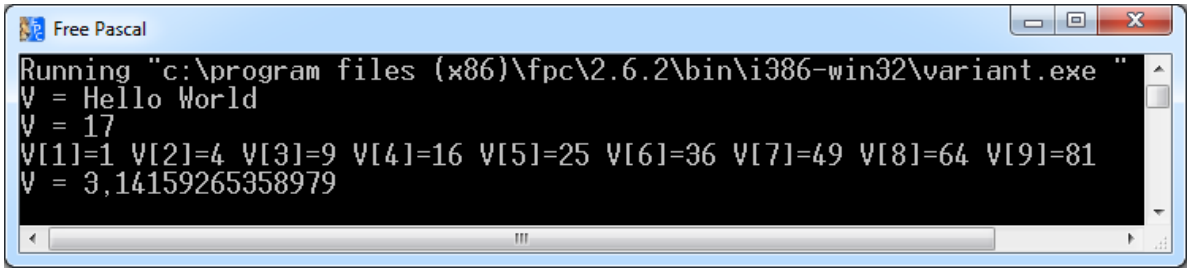
Простые типы данных Free Pascal такие же, как и в Turbo Pascal. В составных типах данных дополнительно включены объединения и тип данных

Variant. Тип Variant определяется только во время выполнения программы и зависит от того, какое значение было присвоено такой переменной. Для работы с переменными типа Variant необходимо в разделе **Uses** подключить модуль Variants.

Приведем пример использования переменной типа Variant в программе:

```
Uses Variants;
var  V:Variant; i:Integer;
begin
   //Используем V для хранения строки
   V:='Hello World';
   Writeln('V = ',V);
   //Используем V для хранения целого числа
   V:=16;
   V:=V+1;
   Writeln('V = ',V);
   //Используем V для хранения одномерного массива
   V:=VarArrayCreate([1,9],varInteger);
   for i:=1 to 9 do V[i]:=Sqr(i);
   for i:=1 to 9 do Write('V[',i,']=',V[i],' ');
   Writeln;
   //Используем V для хранения вещественного числа
   V:=PI;
   Writeln('V = ',V);
 end.
```

Результат работы программы представлен на рис. ниже.



Числовые данные Free Pascal

Во Free Pascal существует десять целых типов, которые отличаются допустимым диапазоном значений и размером оперативной памяти. Их характеристики и названия приведены в таблице (1).

Таблица 1 – Характеристики целочисленных данных.

Целый тип	Диапазон	Размер памяти, байт	Особенность
Shortint	-128 .. 127	1	Со знаком
SmallInt	-32768 .. 32767	2	

Integer	SmallInt или Longint	2 или 4	
Longint	-2147483648 .. 2147483647	4	
Int64	-9223372036854775808 .. 9223372036854775807	8	
Byte	0 .. 255	1	Без знака
Word	0 .. 65535	2	
LongWord	0..4 294 967 295	4	
Cardinal	LongWord	4	
QWord	0..18 446 744 073 709 551 615	8	

Следующие таблицы показывают, какие характеристики имеют вещественные данные.

Тип	Диапазон модуля	Количество цифр мантиссы	Память в байтах
Real	Зависит от операционной системы		4 или 8
Single	$1.5\cdot10^{-45}..3.4\cdot10^{38}$	7-8	4
Double	$5.0\cdot10^{-324}..1.7\cdot10^{308}$	15-16	8
Extended	$1.9\cdot10^{-4932}..1.1\cdot10^{4932}$	19-20	10
Тип	Диапазон	Количество цифр мантиссы	Память в байтах
Comp	$-2^{63}..2^{63}-1$	19-20	8
Currency	От –922337203685477.5808 до 922337203685477.5807	19-20	8

Модуль Math

Дополнительный набор подпрограмм вычисления элементарных и специальных функций сосредоточен в модуле Math Free Pascal. Здесь описаны обращения к тригонометрическим функциям, функции возведения в степень, поиска наибольшего и наименьшего элементов и многие другие. Приведем некоторые из них (таблица 2).

Таблица 2 – Некоторые функции модуля Math Free Pascal.

Функция	Значение		Функция	Значение
arccos(x)	$\arccos(x)$		log10(x)	$\log_{10}(x)$
arcsin(x)	$\arcsin(x)$		log2(x)	$\log_2(x)$
cotan(x); cot(x)	$\frac{\cos x}{\sin x}$		logn(x,y)	$\log_y(x)$
tan(x)	$\frac{\sin x}{\cos x}$		max(x,y); min(x,y)	Тип x, y – integer, int64, extended
sign(x)	$\mathrm{sign}(x)$		power(x,y)	x^y

Процедура sincos(x,s,c) в переменной s возвращает значение синуса x, а в переменную c – косинуса x.

Из вспомогательных функций рассмотрим следующие (таблица 3).

Таблица 3 – Вспомогательные функции.

Формат вызова	Выполняемое действие
`i:=ceil(x)`	Округление к большему ближайшему целому

`i:=CompareValue(x,y);`
где x,y – integer, int64, extended
	`i:= -1, если x<y;`
	`0, если x=y;`
	`1, если x>y`

`DivMod(k1, k2, d, r)`	`d:= k1 div k2;  r:= k1 mod k2`

`j:=EnsureRange(i,min,max);`	`j:= i, если min<= i <=max;`
	`min, если i<min;`
	`max, если i>max`

`i:=floor(x)`	Округление к ближайшему меньшему целому
`FrExp(x,fr,exp)`	Выделение дробной части (`fr`) и порядка (`exp`) вещественного значения *x*
`v:=IfThen(be,et,ef)` где оба типа `et и ef – integer, int64, double` и **`string`**	**`if`** `be` **`then`** `v:=et` **`else`** `v:=ef;`
`bv:=InRange(i,min,max)` где тип `i` – `integer` или `int64`	`bv:=`**`true`**, если `min<= i <=max`, иначе `bv:=`**`false`**.
`bv:=IsInfinity(x)`	`bv:=`**`true`**, если `x=1/0`
`bv:=IsNaN(x)`	`bv:=`**`true`**, если `x=0/0`
`bv:=IsZero(x)`	`bv:=`**`true`**, если `x=0`
`y:=RoundTo(x,k)`	Округление `x` до k–ой десятичной цифры (k>=0 – в целой части, k <0 – в дробной части)
`bv:=SameValue(x,y)`	`bv:=`**`true`**, если $x \equiv y$ (оба - extended)
`bv:=SameValue(x,y,eps)`	`bv:=`**`true`**, если $\lvert x - y \rvert \le$ eps
`bv:=SimpleRoundTo(x,k)`	То же, что и `RoundTo`

Кроме элементарных функций в состав модуля `Math` включены и другие подпрограммы, например, подпрограммы обработки целочисленных и вещественных векторов. Их аргументом является либо открытый массив соответствующего типа (например, `A`), либо указатель на массив (например, `pA`) в сочетании с количеством `N` обрабатываемых элементов (таблица 4).

Таблица 4 – Функции работы с массивами.

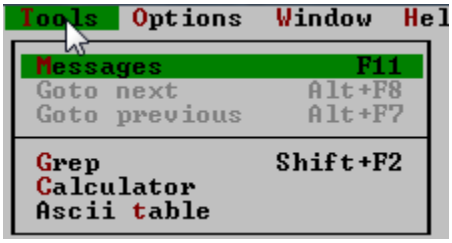
Формат вызова	Выполняемое действие
`k:= MaxIntValue(A)`	Поиск максимального значения в целочисленном массиве
`v:= MaxValue(A);` `v:= MaxValue(pA,N)`	Поиск максимального значения в целочисленном или вещественном массиве
`k:= MinIntValue(A)`	Поиск минимального значения в целочисленном массиве
`v:= MinValue(A);` `v:= MinValue(pA,N)`	Поиск минимального значения в целочисленном или вещественном массиве

`v:= norm(A);` `v:= norm(pA,N)`	Вычисления евклидовой нормы вектора $\sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}$

В состав модуля `Math` включены функции преобразования угловых величин, например, радианы в градусы и наоборот.

Символьные данные Free Pascal

В среде Free Pascal в меню Tools подменю Ascii table позволяет открыть таблицу кодов символов, в которой можно извлечь нужный символ или узнать его код:



В языке Free Pascal имеется функция `LowerCase`, которая переводит символы с верхнего регистра в нижний регистр для букв латинского алфавита: `LowerCase('A')⇒'a'`.

Строки символов

`Free Pascal` поддерживает работу со строковыми константами и переменными пяти типов:

- `String` (короткие строки, появились в самой первой версии `Pascal`. Длина строки до 255 байт);
- `PChar` (совместимые с языком C++. Длина строки до 2 Гбайт);
- `AnsiString` (строки неограниченной длины, состоящие из символов в 1 байт. Длина строки до 2 Гбайт);
- `WideString` (широкие строки, состоящие из символов в 2 байта. Длина строки до 2 Гбайт).
- `UnicodeString` (широкие строки, состоящие из символов Unicode. Длина строки до 2 Гбайт).

Короткие строки

Во Free Pascal объявление строки приводит к объявлению короткой строка в следующих случаях:

– если используется директива компилятора {$H-} и при объявлении строки с использованием ключевого слова **String** не указана ее максимальная длинна;

– если при объявлении строки с использованием ключевого слова **String** указана максимальная длина строки (длина строки не должна превышать 255), то независимо от директив компилятора {$H-}, {$H+};

– если при объявлении строки используется ключевое слово ShortString (независимо от директив компилятора {$H-}, {$H+}).

Например, после выполнения следующего объявления

```
{$H-}
var    s1 : String [10] = 'Hello, world!';
       s2 : String = 'Hello, world!';
       s3 : ShortString = 'Hello, world!';
```

строка s1 может содержать максимум 10 символов, поэтому ей присвоится значение 'Hello, wor', строки s2 и s3 могут содержать до 255 символов и им присвоится значение 'Hello, world!'.

В строковых значениях разрешено употреблять русские буквы, и с их выводом в системе Free Pascal никаких проблем не возникает (в отличие от консольных приложений Delphi). Это обусловлено тем, что режим набора текста программы в среде FP IDE выполняется в той же кодовой странице, с какой работает и консольное приложение.

Строки PChar

Строка типа PChar является указателем на массив типа Char, который заканчивается признаком конца строки #0 (нуль-символом). Длина строк типа PChar не ограничена.

В момент объявления переменной типа PChar компилятор выделяет 4 байта под указатель и заносит туда **Nil**, что эквивалентно созданию пустой строки. Обращение к такому указателю по имени строки типа PChar соответствует выборке или изменению значения всей строки. К символам строки типа PChar также можно обращаться по индексу, который отсчитывается от 0.

Строки AnsiString

Строки типа AnsiString являются строковым типом данных неограниченной длины, где каждый символ представлен однобайтовым ANSI кодом. Объявляются при использовании **String** без указания длины строки при включенной директиве компилятора {$H+} или при помощи предопределенного типа AnsiString. Например, в приведенной ниже программе объявляются две переменные типа AnsiString:

```
{$H+}
var
   A1 : Ansistring = 'Hello';
   A2 : String =', world';
begin
   Writeln(A1,A2);
end.
```

Имя переменной типа AnsiString является указателем на значение, находящееся в куче. В момент объявления переменной данного типа компилятор

выделяет 4 байта под указатель и заносит туда **Nil**, что эквивалентно созданию пустой строки.

В отличие от PChar этот указатель *типизирован,* т. е. ему известен не только адрес, но и длина значения и количество ссылок на значение.

Память, выделяемая для хранения переменной St типа AnsiString, содержащей *n* символов, отражена на следующей схеме:

В куче хранится *дескриптор*, содержащий *текущую длину строки* и *счетчик ссылок* на данное значение строки. После дескриптора последовательно размещаются символы строки, завершающиеся символом с кодом 0.

Если строка типа AnsiString имеет нулевую длину, то память под нее и под дескриптор не выделяется, а соответствующая переменная имеет значение **NIL**.

Переменным типа AnsiString при создании автоматически присваивается пустое начальное значение **NIL**. И только после присваивания переменной нового значения, происходит выделение памяти под дескриптор строки, символы строки и завершающий символ #0.

Если со значением строки связана единственная активная переменная S1 типа AnsiString, то в счетчике ссылок находится 1.

Присваивание значения одной переменной типа AnsiString другой переменной типа AnsiString не приводит к копированию содержимого в новое место. Например, для строк типа AnsiString присваивание S2:=S1 приводит к выполнению трёх действий:

1) счётчик ссылок на строку S2 уменьшается на единицу (если S2 не равно **NIL**);

2) счётчик ссылок на строку S1 увеличивается на единицу;

3) S1 (как указатель) копируется в S2.

Такой механизм присваивания строк типа AnsiString существенно уменьшает время выполнения присваивания строк.

Если в результате выполнения оператора S2:=S1 счетчик ссылок станет равным нулю, то это приведет к освобождению области памяти, на которую ранее указывал указатель S2.

Особенностью реализации AnsiString-строк является автоматическое добавление нуль-символа (символа с кодом 0) в конец строки. Этот символ не

является необходимым для реализации строк типа AnsiString. Он добавляется, чтобы обеспечить возможность преобразования строк типа AnsiString к строкам типа PChar. При таком преобразования (перезаписывания в новую область памяти) как такового не происходит, а строковый указатель типа PChar устанавливается на начало значащих символов строки типа AnsiString.

Для обработки строк типа `AnsiString` существует более 90 функций и процедур, описанных в модуле `StrUtils`.

Строки WideString

Строки `WideString` во многом похожи на строки `PChar`, но для хранения символов используется не один, а два байта, что позволяет хранить в таких строках Юникод-символы. Завершается строка `WideString` двухбайтовым нулём. Строки `WideString`, в отличии от `AnsiString`-строк, не подсчитывают и не хранят количество ссылок на строку. Поэтому каждое присваивание означает создание новой уникальной строки с полным копированием текстовой информации. Тип `WideString` не отличается особой производительностью – вот почему был введён новый тип `UnicodeString`.

Строки UnicodeString

Тип `UnicodeString`, объединяющий возможности типов `AnsiString` и `WideString`, используется для хранения строковых Юникод-данных во Free Pascal. По способу хранения значений тип `UnicodeString` похож на `AnsiString`. Строка `UnicodeString` состоит из дескриптора, символов строки и завершающего строку символа #0. Дескриптор содержит *кодовую страницу, размер элемента*, текущую длину строки, счетчик ссылок на строку.

Преобразование строк

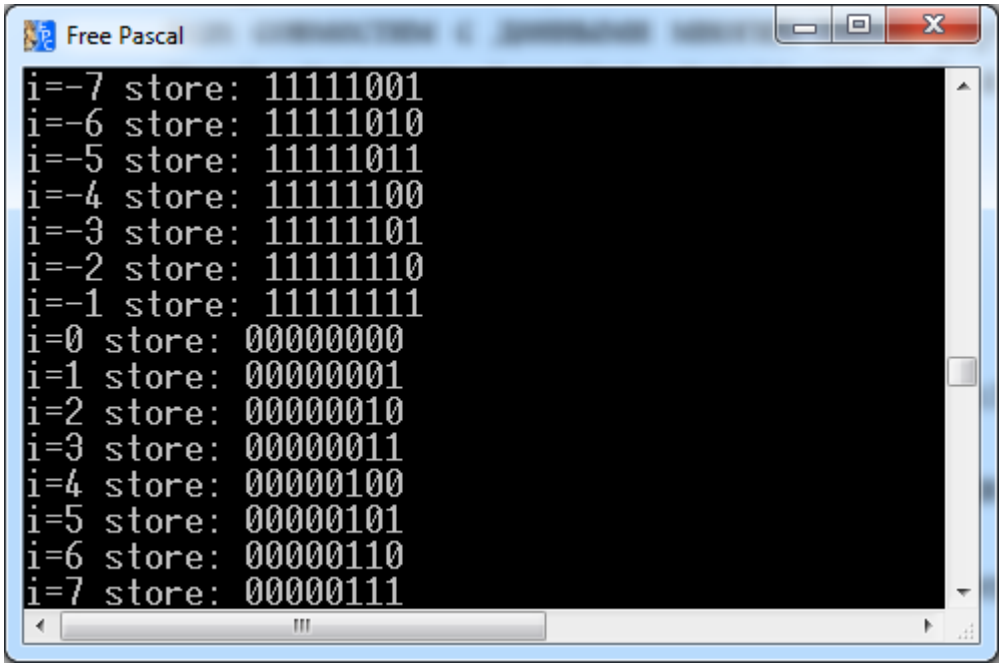
Функция `BinStr(num, k)` преобразовывает целочисленное значение `num` из машинного формата в символьное представление, содержащее `k` (типа `byte`) двоичных разрядов. Здесь и далее тип параметра `num` совместим с данными многих типов (`Byte`, `SmallInt`, `ShortInt`, `Word`, `Integer`, `LongInt`, `Int64`, `QWord`), а результат, возвращаемый функцией, имеет тип `ShortSTRING`.

Например, программа

```
var
  i:ShortInt;
begin
   for i:=-128 to 127 do
   Writeln('i=', i, ' store: ', BinStr(i,8));
end.
```

позволит вывести машинное представление для всех возможных значений переменной типа `ShortInt`.

Приведём фрагмент результата работы этой программы:



Функция `OctStr(num, k)` преобразовывает целочисленное значение `num` из машинного формата в символьное представление, содержащее `k` (типа `byte`) восьмеричных разрядов.

Функция `HexStr(num, k)` преобразовывает целочисленное значение `num` из машинного формата в символьное представление, содержащее `k` (типа `byte`) шестнадцатеричных разрядов.

Новые функции преобразования числовых данных

В системе `Free Pascal` довольно много функций по прямому и обратному преобразованию числовых данных, представленных в машинном и символьном форматах. Приведем их (таблица 5).

Таблица 5 – Прямое и обратное преобразование числовых данных.

Формат обращения	Описание
`IntToBin(num,k[,n])`	Преобразование целочисленного значения `num` из машинного формата в символьное представление, содержащее `k` двоичных разрядов. Необязательный параметр `n` задает количество двоичных цифр, после которых надо вставить дополнительный пробел
`IntToHex(num, k)`	Преобразование целочисленного значения `num` из машинного формата в символьное представление, содержащее `k` шестнадцатеричных разрядов
`IntToStr(num)`	Преобразование целочисленного значения `num` из машинного формата в символьное представление десятичного числа
`IntToRoman(num)`	Преобразование целочисленного значения `num` из машинного формата в символьное представление в римской системе счисления
`RomanToInt(s)`	Преобразование символьного представления целого числа из римской системы счисления в машинный формат

StrToInt(s)	Преобразование целочисленного значения из символьного представления в машинный формат типа Integer
StrToInt64(s)	Преобразование целочисленного значения из символьного представления в машинный формат типа Int64
StrToQWord(s)	Преобразование целочисленного значения из символьного представления в машинный формат типа QWord
FloatToStr(num)	Преобразование вещественного значения num в символьное представление
StrToFloat(s)	Преобразование символьного представления вещественного числа в машинный формат типа Extended
FloatToCurr(num)	Преобразование вещественного значения num в машинный формат представления денежных единиц
CurrToFloat(s)	Преобразование символьного представления денежных единиц в машинный формат типа Currency

Функция преобразования вещественного значения в символьный формат допускает задание дополнительных аргументов, управляющих форматом результата:

```
s:=FloatToStr(num, Format [, Precision, m] );
```

Параметр Format может принимать одно из следующих значений:

- ffCurrency – перевод в символьный формат денежных единиц;
- ffExponent – перевод в символьный формат с плавающей запятой;
- ffFixed – перевод в символьный формат с фиксированной запятой;
- ffGeneral – перевод в формат с плавающей или фиксированной запятой;
- ffNumber – перевод в формат десятичного числа со вставкой символа разделителя "тысяч".

Параметр Precision с последующим числовым аргументом m управляет количеством значащих цифр. Выбор представления в формате ffGeneral определяется системой по величине преобразуемого значения.

Массивы в языке Free Pascal

Free Pascal так же, как и Turbo Pascal, поддерживает массивы двух категорий: традиционные (*статические*) массивы, при объявлении которых в явном виде указываются конкретные границы изменения каждого индекса и динамические массивы, объявление которых не содержит указания о границах изменения индексов.

Работа с динамическими массивами Free Pascal

Так как при объявлении динамического массива не указывается его длина, то компилятор выделяет для каждого динамического массива (глобального или локального) по 4 байта для хранения указателя на первый элемент динамического массива.

```
var
   A1_d : array of Integer;    // одномерный массив
   A2_d : array of array of Integer;
```

```
                                        // двумерный массив
```

При объявлении переменной типа динамический массив значение указателя первоначально равно **Nil**. Фактическое выделение памяти под динамические массивы производится *только во время работы программы* путем вызова стандартной процедуры `SetLength`.

Например, в программе ниже объявляется динамический массив ADin, затем процедура `SetLength` выделяет место для хранения ста элементов

```
var
   VDin : array of Integer;    // одномерный массив
begin
   SetLength(VDin, 100);
   ...
end.
```

После вызова `SetLength` доступными станут индексы массива от 0 до 99. Всем ста элементам массива присвоится начальное значение, равное нулю.

Индексы динамических массивов *всегда отсчитываются от нуля*, поэтому при вызове процедуры `SetLength` кроме имени динамического массива задается количество элементов. Память, *впервые выделяемая* динамическому массиву, *всегда чистится.*

Во время работы программы можно неоднократно обращаться к процедуре `SetLength`. Если при очередном обращении новая длина массива больше предыдущей, то значения ранее вычисленных элементов сохраняются, а всем добавляемым элементам присваиваются нулевые значения. Если новая длина динамического массива меньше текущей, то сохраняются значения оставшихся первых элементов, отбрасываемые элементы будут безвозвратно потеряны.

Для выделения памяти под двумерный динамический массив к процедуре `SetLength` обращаются с тремя параметрами, задавая количество строк и количество столбцов. Например, следующий код:

```
var
   MDin : array of array of Integer // двумерный массив
begin
   SetLength(MDin, 10, 3);
   ...
end.
```

эквивалентен «статическому» описанию вида:

```
var
    MDin : array[0..9, 0..2] of Integer;
```

Если динамический массив был объявлен в процедуре или функции, то он является локальным и после выхода из программной единицы память, занимаемая значениями элементов массива, освобождается.

Освободить память, выделенную, например, для динамического массива `MDin` можно следующими способами:

```
MDin := Nil;
```

или

```
SetLength(MDin,0);
```

или

```
Finalize(MDin};
```

Определение длины и размеров массивов

Объем оперативной памяти в байтах, занятых элементами одномерного статического массива, называется *длиной одномерного статического массива*. Функции `SizeOf` возвращает длину одномерного статического массива.

```
var
    Al_s: array[10..15] of Double;
begin
    Writeln(SizeOf(Al_s));
```

Так как каждый элемент массива `Al_s` имеет тип `double`, т. е. занимает 8 байт, и в массиве объявлено 6 элементов, то для хранения всех элементов массива потребуется $8 \times 6 = 48$ байт. Именно эта величина и будет выведена оператором `Writeln(SizeOf(Al_s))`.

Количество элементов одномерного статического массива называется *размером одномерного статического массива*. Для определения размера одномерного статистического массива можно использовать функцию `Length`. Например, для объявленного выше массива `Al_s` оператор `Writeln(Length(Al_s))` выведет значение 6.

Для определения *минимального* и *максимального* значения индекса одномерного статического массива в `Free Pascal` существуют функции `Low` и `High`. Так для объявленного выше массива `Al_s` оператор `Writeln(Low(Al_s))` выведет значение 10, а оператор `Writeln(High(Al_s))` выведет значение `15`.

Для одномерного динамического массива `Al_d`, объявленного, например, в виде

```
var   Al_d:   array of Double;
```

результат функции `SizeOf (Al_d)` всегда равен 4, независимо от того, выделена ли оперативная память под значения элементов динамического массива оператором `SetLength` или еще не выделена.

Фактически `SizeOf (Al_d)` – объем памяти, занимаемый указателем `Al_d` на соответствующие данные.

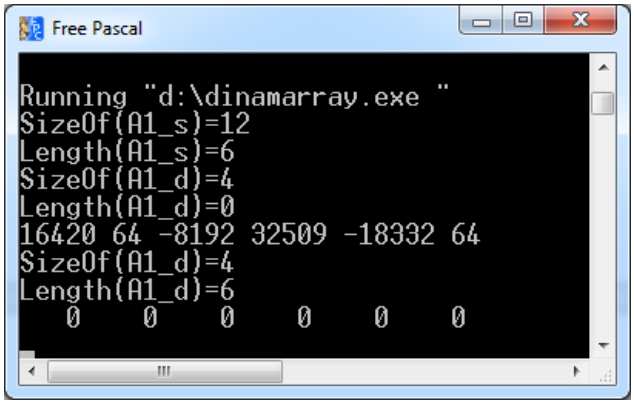
Функция `Length` выдает фактическое значение, равное текущему количеству элементов динамического массива. Если ещё не выполнялась процедура `SetLength`, то значение `Length(Al_d)` равно 0.

Не имеет смысла применять функцию `Low` к динамическим массивам, т. к. минимальное значение индекса любого одномерного динамического массива всегда равно 0. Максимальное значение индекса можно узнать вызвав процедуру

High(Al_d) или вычислить по формуле: Length(Al_d) – 1. Например, до обращения к процедуре SetLength максимальное значение индекса массива Al_d равно –1.

Следующая программа демонстрирует тактику выделения памяти для локальных массивов. Локальному статическому массиву память выделяется в момент входа в подпрограмму, но эта память не чистится. Локальному динамическому массиву поначалу выделяется 4 байта под указатель, но после обращения к процедуре SetLength выделяется чистая память.

```
procedure WorkMatr;
var
     A1_s:array[10..15] of integer;
     A1_d:array of integer;
     J   :integer;
begin
  Writeln('SizeOf(A1_s)=',SizeOf(A1_s));
  Writeln('Length(A1_s)=',Length(A1_s));
  Writeln('SizeOf(A1_d)=',SizeOf(A1_d));
  Writeln('Length(A1_d)=',Length(A1_d));
  for j:=Low(A1_s) to High(A1_s) do
      Write(A1_s[j],' ');
  Writeln;
  SetLength(A1_d,6);
  Writeln('SizeOf(A1_d)=',SizeOf(A1_d));
  Writeln('Length(A1_d)=',Length(A1_d));
  for j:=0 to High(A1_d) do
      Write(A1_d[j]:4,' ');
  Writeln;
end;
begin
  WorkMatr;
end.
```



Для двумерного массива q, как статического, так и динамического, размером nxm компилятор заводит n указателей. Первый из них имеет имя q, совпадающее с именем массива. Он является указателем *на начало массива* и одновременно *на первую строку массива.* Если минимальное значение первого

индекса равно k, то указатель q[k] тоже адресует первую строку массива (т.е. q и q[k] — это два имени одного и того же указателя). Следующий указатель с именем q[k+1] ссылается на вторую строку массива и т. д.

Базовые операторы языка

Оператор множественного присваивания

```
V1:= V2:=… :=e;
```

Значение выражения e присваивается нескольким переменным.

Оператор присваивания – краткая запись

```
V⊗=e;
```

Краткая запись оператора V:=V⊗e;, заимствованная из языка C. В качестве знака операции могут выступать операции сложения (+), вычитания (–), умножения (*) и деления (/). Этот формат может использоваться только при включении в текст программы директивы {$COPERATORS **ON**}.

Процедуры и функции

Описание процедуры:

```
procedure name_proc[(list_arguments)];[directives;]
  Блок процедуры
end;
```

Описание функции:

```
Function name_func[(list_arguments)]:тип;[directives;]
  Блок вычисления значения функции
  Возврат значения функции
end;
```

Здесь list_arguments – список формальных параметров, Directives – уточняющие директивы, тип – это тип возвращаемого значения, который может быть именем простого типа или **String**.

Многие директивы пришли из Turbo Pascal, но большинство из них связано с вызовами подпрограмм, написанных в других системах программирования.

Нормальным завершением работы процедуры считается выход на завершающий **end**. Достаточный возврат из процедуры реализуется с помощью оператора **exit** (выход).

Система Free Pascal допускает четыре варианта возврата вычисленного значения функции. Традиционный способ заключается в присвоении возвращаемое значения имени функции:

```
name_func := e;
```

Второй вариант, появившийся в языке **Object** Pascal, заключается в использовании системной переменной Result:

```
Result   :=  e;
```

Третий способ использует модифицированную системную функцию exit.

```
exit(e);     //с одновременным выходом из функции
```

Параметры

Список формальных параметров (`list_arguments`) состоит из элементов, которые могут содержать до трех полей:

```
[характеристика] имя [: тип параметра]
```

или

```
[характеристика] имя_1,…, имя_n [: тип параметров]
```

Характеристика является признаком передаваемых данных.

`Имя` – идентификатор параметра.

`Тип параметра` – это имя типа или конструкция **`Array of`** `тип_элементов`.

Если характеристика отсутствует, то параметр передается по значению.

Если указана характеристика **`Var`**, `Out`, **`Const`**, то реализуется некоторая вариация передачи параметра по ссылке. В случае использования **`Var`** – можно читать и изменять значение, в случае использования **`Const`** – только читать, случае `Out` – только записывать.

В текущей версии Free Pascal характеристики **`Var`** и `Out` эквивалентны.

Параметры подпрограмм по умолчанию

В языке `Free Pascal` имеется возможность объявить процедуру или функцию, у которой значения нескольких последних формальных аргументов списка `list_arguments` заданы со значениями по умолчанию. Например, рассмотрим функцию, вычисляющее среднее арифметическое трёх действительных чисел, из которых два последних числа имеют значение по умолчанию:

```
Function avg(x1:Double; x2:Double=6; x3:Double=9):Double;
 begin
  avg:=(x1+x2+x3)/3;
 end;
```

К функции `avg` можно обращаться:

- с одним первым (обязательным) аргументом, например вызов функции `avg(8)` эквивалентен обращению `avg(8, 6, 9)` в котором используются значения по умолчанию второго и третьего аргументов;
- с двумя первыми аргументами; в этом случае, например, вызов функции `avg(8, 2)` эквивалентен обращению `avg(6, 2, 9)` в котором используется значение по умолчанию третьего аргумента;
- с тремя аргументами, например, `avg(8, 2, 1)`; в этом случае значения аргументов по умолчанию не используются

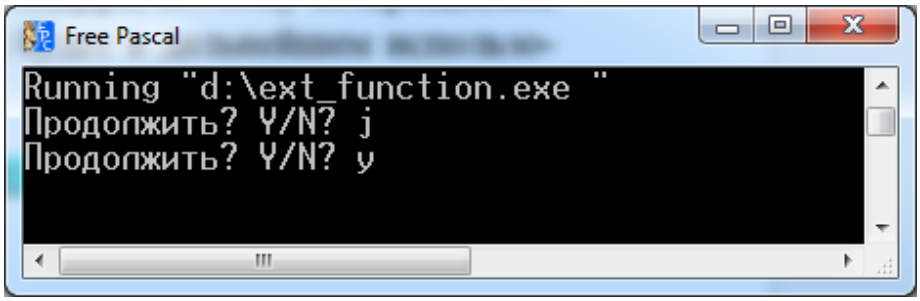
Расширенный вызов функций

В программах на языке `Free Pascal` допускается вызов функций с игнорированием возвращаемого значения. По сути, происходит вызов функции как процедуры.

Например, в приведенной ниже программе функция ContinueOrExit вызывается как процедура, поэтому возвращаемое ей значение будет потеряно и его нельзя будет в дальнейшем использовать в программе.

```
program ext_function;
 function ContinueOrExit:Char;
 var
     Ch:Char;
 begin
 repeat
   Write('Продолжить? Y/N? ');
   Readln(Ch);
 until UpCase(Ch) in ['Y','N'];
 ContinueOrExit:=UpCase(Ch);
 end;
begin
 ContinueOrExit;
end.
```

Например, на рис. ниже в процессе вызова функции ContinueOrExit как процедуры в качестве реакции на поставленный вопрос сперва ввели с клавиатуры символ «j», а на повторный вопрос ввели символ «y». Однако поскольку вызов функции был осуществлён как вызов процедуры, то в программе нельзя определить какую клавишу «y» или «n» нажал пользователь при ответе на вопрос.



Перегрузка функций и процедур

Перегрузка (англ. overloading) функций (процедур) означает, что в одной области видимости определено несколько функций (процедур) с одинаковым именем, но различным списком формальных параметров. Различие могут заключаться в количестве параметров и/или типе параметров. Нужная функция выбирается в момент компиляции исходя из количества и типа аргументов.

Приведем пример программы, в которой будет использоваться перегружаемая функция Print, которая в случае вызова с одним параметром типа одномерный целочисленный динамический массив будет выводить его элементы последовательно в строку; в случае вызова с одним параметром типа двумерный динамический массив будет выводить его в виде матрицы построчно; в случае вызова с двумя параметрами: записью типа TCompNum, содержащая вещественную и мнимую части комплексного числа, и целого числа n, будет

выводить содержимое этой записи в виде комплексного числа с n дробными знаками в вещественной и мнимой части. При вызове функции Print с одним параметром типа TCompNum, будет выводить содержимое этой записи в виде комплексного числа с двумя дробными знаками в вещественной и мнимой части.

```
program OverLoadProc;
uses strutils;
type
     TVec=array of ShortInt;
     TMat=array of array of ShortInt;
     TCompNum=record
               Re,Im:Double;
              end;
procedure Print(A:TVec);
 var
     i: byte;
 begin
 for i:=0 to High(A) do Write(A[i]:5);
 Writeln;
 end;
procedure Print(A:TMat);
 var
     i,j: byte;
 begin
 for i:=0 to High(A) do
     begin
      for j:=0 to High(A[0]) do
         Write(A[i,j]:5);
      Writeln;
     end;
 end;
procedure Print(A:TCompNum;n:Byte=2);
 begin
 if (A.Re<>0) And (A.Im<>0) then
   Write(A.Re:0:n,IfThen(A.Im>0,'+','-'),
         Abs(A.Im):0:n,'*i')
 else
    if (A.Im=0) And (A.Re=0) then Write (0)
    else
        if A.Re=0 then Write (A.Im:0:n,'*i')
        else Write(A.Re:0:n);
 end;

var
    Vec     : TVec;
    Mat     : TMat;
    CompNum : TCompNum;
```

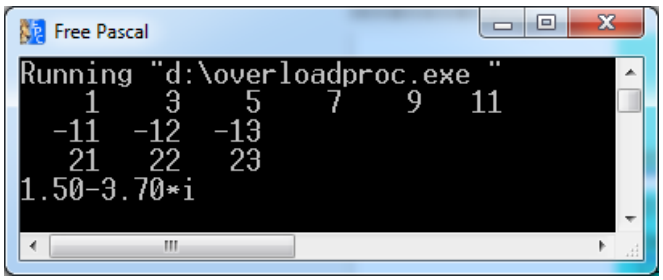
```
begin
 SetLength(Vec,6);
 Vec[0]:=1;
 Vec[1]:=3;
 Vec[2]:=5;
 Vec[3]:=7;
 Vec[4]:=9;
 Vec[5]:=11;
 SetLength(Mat,2,3);
 Mat[0,0]:=-11;
 Mat[0,1]:=-12;
 Mat[0,2]:=-13;
 Mat[1,0]:=21;
 Mat[1,1]:=22;
 Mat[1,2]:=23;
 CompNum.Re:=1.5;
 CompNum.Im:=-3.7;
 Print(Vec);
 Print(Mat);
 Print(CompNum);
end.
```

В этой программе использовалась функция `IfThen`, описанная в модуле `strutils`. Функция `IfThen` возвращает одну из двух строк в зависимости от проверяемого логического выражения. Синтаксис функции `IfThen`:

```
IfThen(LogicalExp, StrTrue, StrFalse)
```

где `LogicalExp` – проверяемое логическое выражение, `StrTrue` – строка, возвращаемая функцией `IfThen` в том случае, когда значение логического выражения `LogicalExp` есть **`True`**, `StrFalse` – строка, возвращаемая функцией `IfThen` в том случае, когда значение логического выражения `LogicalExp` есть **`False`**.

Результат работы программы представлен на рис. ниже.



Управление файлами в стиле Windows

В модуле . Free Pascal содержится много процедур и функций по управлению каталогами и файлами. В большинстве своем эти процедуры используют числовые атрибуты – дескрипторы (*англ. handle*), которые операционная система присваивает файлам при их создании или открытии. Приведем некоторые полезные процедуры и функции из модуля `sysutils`.

Функции для работы с дисками

Функция DiskSize возвращает размер в байтах диска, указанного параметром Disk:

```
DiskSize(Disk)
```

где Disk – номер диска: 0 – для текущего диска, 1 – для первого флоппи-дисковода, 2 – для второго флоппи-дисковода, 3 – для первого жесткого диска, 4-26 – для последующих дисков.

Функция DiskFree возвращает размер свободного пространства в байтах диска, указанного параметром Disk:

```
DiskFree(Disk)
```

где Disk – номер диска.

Функции для работы с директориями

Функция DirectoryExists проверяет существует ли указанная в качестве строкового аргумента Directory директория и в случае существования такой директории возвращает значение **True**, иначе – **False**. Синтаксис функции:

```
DirectoryExists(Directory);
```

Функция CreateDir создает новую директорию:

```
CreateDir(NewDir)
```

где параметр NewDir имеет тип **String** и содержит имя создаваемой директории. Если в NewDir не указан абсолютный путь, то новая директория создается в текущей рабочей директории. Функция CreateDir возвращает значение **True**, если директория была успешна создана, иначе – значение **False**.

Функция RemoveDir удаляет директорию, задаваемую параметром Dir, с диска:

```
RemoveDir(Dir)
```

Функция RemoveDir возвращает значение **True**, если директория была успешна удалена, иначе – значение **False** (например директория не была пуста или не существовала).

Функции для работы с файлами

Функция FileExists проверяет существует ли указанная в качестве строкового аргумента FileName директория и в случае существования такого файла возвращает значение **True**, иначе – **False**. Если в качестве FileName указано имя директории, то функция вернёт значение **False**, если программа запущена в операционной системе Windows и **True** – если в Unix. Синтаксис функции:

```
FileExists(FileName);
```

Функция FileCreate создает новый файл и возвращает дескриптор этого файла, который может быть использован для чтения из файла или для записи в файл при помощи функций FileRead и FileWrite соответственно. Синтаксис:

```
FileCreate(FileName)
```

Если файл с именем FileName уже существует на диске, то он будет перезаписан. Если возникнет ошибка создания файла (например, из-за некорректного имени или нехватки места на диске), то функция вернет значение –1 (минус один).

Функция FileOpen открывает существующий файл и возвращает дескриптор файла. Если файл не существует, то он будет создан при попытке открыть несуществующий файл. Синтаксис:

```
FileOpen(FileName, Mode)
```

где FileName имеет тип **String** и содержит имя файла, Mode имеет тип Integer и указывает режим доступа к файлу. Mode может быть одной из следующих констант (таблица 6):

Таблица 6 – Режимы доступа к файлу.

Mode	Значение	Описание
fmOpenRead	$0000	Открывает файл в режиме только для чтения
fmOpenWrite	$0001	Открывает файл в режиме только для записи
fmOpenReadWrite	$0002	Открывает файл в режиме для чтения и записи

С вышеуказанными режимами доступа могут использоваться флаги совместного доступа и блокировки (таблица 7).

Таблица 7 – Флаги совместного доступа и блокировки.

Mode	Значение	Описание
fmShareCompat	$0000	Совместный доступ, посредством FCBs (**File** Control Blocks)
fmShareExclusive	$0010	Никакие другие приложения не могут открывать файл ни в каком режиме
fmShareDenyWrite	$0020	Другие приложения могут открывать файл только для записи
fmShareDenyRead	$0030	Другие приложения могут открывать файл только для чтения
fmShareDenyNone	$0040	Полный доступ для других приложений

Значение параметра Mode формируется как логическая сумма из нужных наборов признаков при помощи побитового оператора **Or**.

В случае возникновения ошибки при открытии файла, функция FileOpen возвращает значение – 1 (минус один).

Функция FileWrite записывает данные из буфера в открытый файл с указанным дескриптором. Синтаксис:

```
FileWrite(Handle, Buffer, Count)
```

где Handle – дескриптор файла, в который осуществляется запись, Buffer – буфер (Buffer фактически содержит записываемые данные, а не является указателем, и должен содержать не менее Count байт, иначе может произойти ошибка), Count – число байт информации, которая записывается из буфера Buffer)

В случае успешного выполнения операции записи в файл функция возвращает количество фактически записанных байт, которое может быть меньше значения Count (например, во время записи закончилось место на диске). В случае возникновения ошибки функция возвращает -1.

Функция FileRead читает Count байт из открытого файла с дескриптором Handle в буфер Buffer. Синтаксис

```
FileRead(Handle, Buffer, Count)
```

Если значение Count превышает размер файла, то фактическое количество прочитанных байт будет меньше значения Count.

Функция возвращает число фактически прочитанных байт. В случае возникновения ошибки функция возвращает -1.

Функция FileSeek устанавливает указатель в заданную позицию открытого файла. Синтаксис:

```
FileSeek(Handle, Offset, Origin)
```

где Handle – дескриптор файла, полученный при создании или открытии файла с помощью функции FileCreate и FileOpen соответственно, Offset –смещение указателя в байтах относительно позиции, указанной в параметре Origin. Параметр Origin может принимать следующие значения (таблица 8):

Таблица 8 – Значения параметра Origin.

Origin	Значение	Описание
fsFromBeginning	0	Смещение указателя относительно начала файла (нумеруется начиная с нуля)
fsFromCurrent	1	Смещение указателя относительно текущей позиции
fsFromEnd	2	Смещение указателя относительно конца файла. В этом случае параметр Offset может принимать только отрицательные и нулевые значения.

При успешном выполнении функция возвращает новую позицию указателя файла относительно начала файла, а если возникает ошибка, то возвращается –1.

Функция FileTruncate усекает открытый для записи файл до указанного количества байт и возвращает значение **True**, в случае успешного выполнения и значение **False** в случае возникновения ошибки. Синтаксис:

```
FileTruncate(Handle, Size)
```

где Handle – дескриптор файла, Size – количество байт, до которого будет усекаться файл.

Функция FileClose закрывает файл, определенный дескриптором Handle. Синтаксис:

```
FileClose(Handle)
```

Функция DeleteFile удаляет директорию, задаваемую параметром Dir, с диска:

```
DeleteFile(FileName)
```

Функция DeleteFile возвращает значение **True**, если файл был успешно удален, иначе – значение **False**.

Функция RenameFile переименовывает файл, заменяя старое имя OldName на новое NewName:

```
RenameFile(OldName, NewName)
```

Функция возвращает значение **True** в случае успешного выполнения операции переименования файла и **False** в противном случае.

Кроме рассмотренных выше функций в модуле **sysutils** размещено много процедур и функций для работы с атрибутами файлов (FileAge, FileDateToDateTime, FileGetAttr, FileGetDate, FileSetAttr, FileSetDate, GetCurrentDir), по извлечению из полного имени файла отдельных компонент (диска, пути, имени файла, расширения файла и т. д.) и объединению компонентов имени (ExpandFileName, ExpandFileNameCase, ExpandUNCFileName, ExtractFileDir, ExtractFileDrive, ExtractFileExt, ExtractFileName, ExtractFilePath, ExtractRelativePath), по поиску файлов, чьи имена удовлетворяют заданному шаблону поиска (FindFirst, FindNext, FindClose) и др.

Дополнительную информацию по форматам вызова этих подпрограмм вы можете найти в онлайн справке по Free Pascal:

http://www.freepascal.org/docs-html/rtl/sysutils/filenameroutines.html

Стандартные модули Free Pascal

Free Pascal включает более 40 модулей, однако лишь половина из них ориентирована на эксплуатацию под управлением Windows и только порядка десятка могут стать повседневным инструментом большинства программистов. Список таких модулей приведен ниже (таблица 9).

Таблица 9 – Стандартные модули Free Pascal.

Имя модуля	Назначение
Crt	Работа с дисплеем, клавиатурой и звуком
DateUtils	Подпрограммы по работе с датой и временем
Dos	Опрос и установка системной даты и таймера, работа с прерываниями
KeyBoard	Доступ к низкоуровневым функциям по работе с клавиатурой
Graph	Работа с библиотекой графических программ Borland Graphics **Interface** (BGI)
Math	Вычисление элементарных и специальных функций
Mouse	Доступ к низкоуровневым функциям по работе с мышью
Strings	Работа со строковыми данными типа PChar

Имя модуля	Назначение
StrUtils	Работа со строковыми данными типа AnsiString
System	Набор наиболее используемых подпрограмм разного назначения
SysUtils	Расширенный аналог такого же модуля Delphi
Windows	Поддержка вызова системных функций Windows (Win32 API)

Для ознакомления с другими модулями нужно обращаться к соответствующим документам по описанию системы Free Pascal.

Программирование с объектами

В этом разделе демонстрируются некоторые особенности объектно-ориентированного программирования в Free Pascal, реализованные в режиме **Object** Pascal extension **on**, который устанавливается с помощью команды Options —> Compiler.

Создадим модуль для работы с комплексными числами. Для этого определим объект TCompNum. Приведем поля и методы этого объекта в таблице 10.

Таблица 10 – Поля и методы этого объекта TCompNum.

Поля, методы	**Назначение**
Re	Поле для хранения действительной части комплексного числа
Im	Поле для хранения мнимой части комплексного числа
constructor Init	Конструктор без параметров (конструктор по умолчанию) создает комплексное число с нулевой вещественной и мнимой частями
constructor Init(x:TCompNum)	Конструктор копирования, копирует значение ранее определенного экземпляра объекта
constructor Init(ReNew,ImNew:double)	Конструктор, инициализирующий поля экземпляра объекта из двух введенных чисел
function Conjugate:TCompNum;	Метод объекта, возвращающий комплексно-сопряженное число
function Modulus:double;	Метод объекта, возвращающий модуль комплексного числа

Для типа TCompNum перегрузим операторы сложения, вычитания, умножения и деления. Будем использовать процедуру Print для вывода на печать комплексного числа.

```
Unit ComplexNumbers;
// Компилировать в режиме Object Pascal Extension on
interface

uses
     strutils;
```

```
type
 TCompNum=object

  private
   Re,Im:double;

  public
   constructor Init;
   constructor Init(x:TCompNum);
   constructor Init(ReNew,ImNew:double);
   function Conjugate:TCompNum;
   function Modulus:double;
  end;

 operator +(x,y:TCompNum) z:TCompNum;
 operator -(x,y:TCompNum) z:TCompNum;
 operator *(x,y:TCompNum) z:TCompNum;
 operator /(x,y:TCompNum) z:TCompNum;
 procedure Print(x:TCompNum; n:byte=2);
implementation

constructor TCompNum.Init();
 begin
 Re:=0; Im:=0;
 end;

constructor TCompNum.Init(x:TCompNum);
 begin
 Re:=x.Re; Im:=x.Im;
 end;

constructor TCompNum.Init(ReNew,ImNew:double);
 begin
 Re:=ReNew; Im:=ImNew;
 end;

function TCompNum.Conjugate:TCompNum;
 begin
 Result.Re:=Re;
 Result.Im:=-Im;
 end;

function TCompNum.Modulus:double;
begin
 Result:=Sqrt(Sqr(Re)+Sqr(Im))
 end;

operator + (x,y:TCompNum) z:TCompNum;
 begin
```

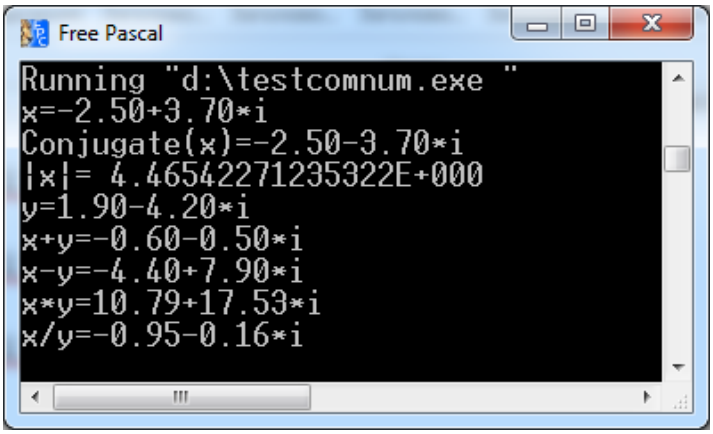
```
 z.Re:=x.Re+y.Re;
 z.Im:=x.Im+y.Im;
 end;

operator - (x,y:TCompNum) z:TCompNum;
 begin
 z.Re:=x.Re-y.Re;
 z.Im:=x.Im-y.Im;
 end;

operator * (x,y:TCompNum) z:TCompNum;
 begin
 z.Re:=x.Re*y.Re-x.Im*y.Im;
 z.Im:=x.Re*y.Im+x.Im*y.Re;
end;

operator / (x,y:TCompNum) z:TCompNum;
 begin
 if (y.Re=0) And (y.Im=0) then
     begin
     Writeln('Деление на 0');
     halt;
     end;
 z.Re:=(x.Re*y.Re+x.Im*y.Im)/(Sqr(y.Re)+Sqr(y.Im));
 z.Im:=(-x.Re*y.Im+x.Im*y.Re)/(Sqr(y.Re)+Sqr(y.Im));
 end;

procedure Print(x:TCompNum; n:byte=2);
 begin
 if (x.Re<>0) And (x.Im<>0) then
   Write(x.Re:0:n,IfThen(x.Im>0,'+','-'),
         Abs(x.Im):0:n,'*i')
   else
    if (x.Im=0) And (x.Re=0) then Write (0)
       else
        if x.Re=0 then Write (x.Im:0:n,'*i')
           else Write(x.Re:0:n);
 end;
end.
```

Переопределение арифметических операций производится следующим образом. Для каждой операции используются два операнда – аргументы переопределяемой операции. Переопределяемая операция должна вернуть результат своей работы в виде значения `TCompNum`. Для этой цели в программе используется переменная `z`, тип которой определяется типом возвращаемого значения.

Приведем текст программы, использующей описанный выше модуль `ComplexNumbers`.

```
Program TestCompNum;
uses ComplexNumbers;
var
   x,y,z : TCompNum;
begin
  x.Init(-2.5,3.7);
  Write('x='); Print(x); Writeln;
  Write('Conjugate(x)='); Print(x.Conjugate); Writeln;
  Writeln('|x|=',x.Modulus);
  y.Init(1.9,-4.2);
  Write('y='); Print(y); Writeln;
  z:=x+y;
  Write('x+y='); Print(z); Writeln;
  z:=x-y;
  Write('x-y='); Print(z); Writeln;
  z:=x*y;
  Write('x*y='); Print(z); Writeln;
  z:=x/y;
  Write('x/y='); Print(z); Writeln;
  Readln;
end.
```

Результат выполнения программы представлен на рис. ниже.



Особенности работы со звуком в Free Pascal

В Free Pascal процедура `Sound` выполняются не так, как это происходило в Turbo Pascal. В Turbo Pascal вызов процедура `Sound(f)` приводил к воспроизведению непрерывному звуку частоты `f` (в герцах), остановить который можно было спустя некоторое время, регулируемое с помощью задержки `Delay`, путем вызова процедуры `NoSound`.

В Free Pascal процедура существует две процедуры `Sound`. Одна описана в модуле Crt, а вторая – в WinCrt. Процедура `Sound` из модуля Crt игнорирует указанную при вызове частоту, и вместо непрерывного звука указанной частоты генерирует непродолжительный звук, напоминающий системный звук, воспроизводимый в Windows при возникновении ошибки. Поэтому обращение к процедуре `NoSound` лишено смысла. Процедура `Sound` из модуля `WinCrt`

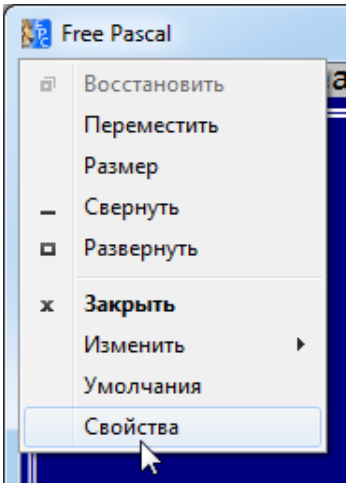
воспроизводит звук указанной при вызове частоты, но этот звук воспроизводится не непрерывно, а разово на непродолжительное время.

Особенности работы с экраном в текстовом режиме в Free Pascal

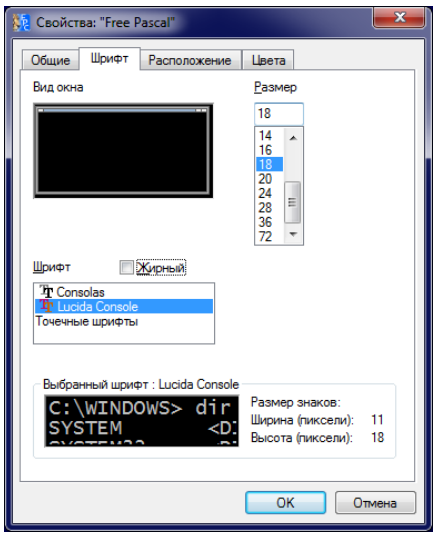
При работе на компьютере под управлением операционной системы MS DOS текстовый режим поддерживался аппаратными средствами. В этом режиме экран по умолчанию вмещал 25 строк по 80 символов в строке. При выводе информации на экран в текстовом режиме в MS DOS последовательно заполнялись строки с первой по двадцать пятую. Заполнение 25-й строки приводило к подъему содержимого окна вывода и выталкиванию верхних строк за пределы зоны видимости.

В настоящее время при работе в Windows текстовый режим поддерживается программными средствами. Поскольку при работе в Windows можно выбирать гарнитуру и размер шрифта в текстовом режиме, то можно изменять и количество строк, отображаемых на экране, и количество символов в строке.

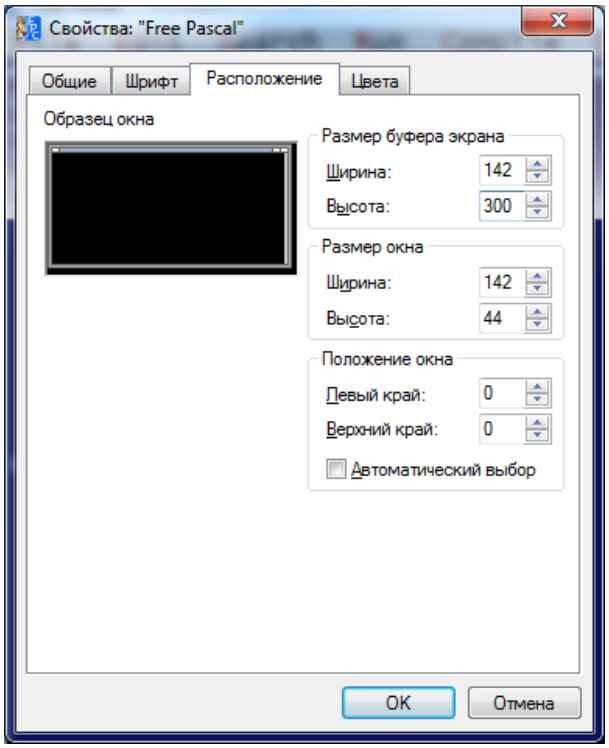
Чтобы изменить размер экрана в текстовом режиме, необходимо запустить приложение Free Pascal, затем щелкнуть кнопку системного меню этого приложения и выбрать команду свойства.



В открывшемся диалоговом окне на вкладке **Шрифт** можно выбрать одну из трех гарнитур шрифта (Consolas, Lucida Console, Точечные шрифты) и установить размер шрифта.



На вкладке **Расположение** можно выбрать ширину и высоту окна в текстовом режиме и начальное положение окна при запуске. На рисунке ниже выбран размер экрана, состоящий из 44 строк по 142 символа в строке. Положение окна выбрано так, чтобы левый верхний угол окна находился в левом верхнем углу экрана. Также целесообразно увеличить высоту буфера экрана, например, как на рис. ниже, до 300 строк. Это позволит по мере вывода строк в количестве, превышающем высоту экрана (44 строки на рис. ниже), не терять первые строки вывода программы. Эти строки не пропадут, их можно будет увидеть, выполнив прокрутку.



Особенности работы с графикой в Free Pascal

В Free Pascal для работы с графикой в операционной системе Windows существует три модуля: `Graph`, `ptcGraph`, `sdlGraph`. Из них первые два рекомендуется использовать для совместимости с графической библиотекой BGI (Borland Graphics **Interface**) Turbo Pascal.

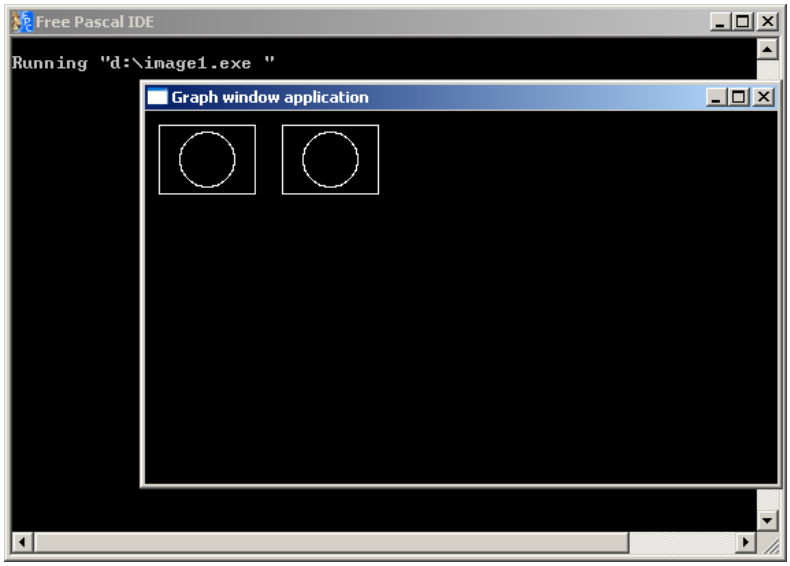
Модуль `sdlGraph` ориентирован на работу с кроссплатформенной библиотекой SDL (Simple DirectMedia Layer), которая обеспечивает низкоуровневый доступ к аудио, клавиатуре, мыши и графическому аппаратному обеспечению через OpenGL и Direct3D. Рассмотрение работы с `sdlGraph` выходит за рамки данного пособия.

Модуль ptcGraph ориентирован на работу с кроссплатформенной библиотекой OpenPTC, поддержка которой прекращена в 2005 году. При использовании модуля `ptcGraph` рекомендуется, при необходимости, вместо модуля `Crt`, использовать модуль `ptcCrt`.

Рассмотрим два основных отличия в работе с модулями `Graph`, `ptcGraph` в Free Pascal от работы с модулем `Graph` в Turbo Pascal.

Во-первых, модули `Graph`, `ptcGraph` в Free Pascal поддерживают работу с более высоким разрешением экрана и большим количеством цветов.

Во-вторых, консольному приложению, работающему с графикой, выделяется два окна.



В главном (текстовом) окне реализуются обычные интерфейсные взаимодействия между пользователем и приложением (ввод/вывод по операторам Read/Readln, Write/Writeln, ReadKey, KeyPressed), в дополнительном окне выполняются построения графических фигур и отображение пояснительных подписей с помощью процедур BGI. Из-за такого механизма работы с графикой в Free Pascal осложняется работа с интерактивными графическими программами. Пусть, например, вы выводите на экран некоторый рисунок и перемещаете его по экрану при помощи клавиш со стрелочками. В этом случае рисунок отображается в графическом окне, а программа ждет нажатия на клавиши в другом (текстовом) окне, которое размещается позади графического окна. Поэтому придётся все время переключаться между этими окнами, что очень неудобно и, фактически, делает такого рода программы бесполезными. Однако, если использовать модуль `ptcGraph` совместно с модулем `ptcCrt`, то хотя по-прежнему консольное графическое приложение будет использовать два окна, появится возможность, чтобы программа отслеживала нажатие клавиш не в текстовом окне, а в графическом.

Особенности инициализации графического режима в Free Pascal

Нецелесообразно при инициализации графического режима в Free Pascal использовать для задания значений `Driver` и `Mode` оператор вида:

```
Driver:=DETECT
```

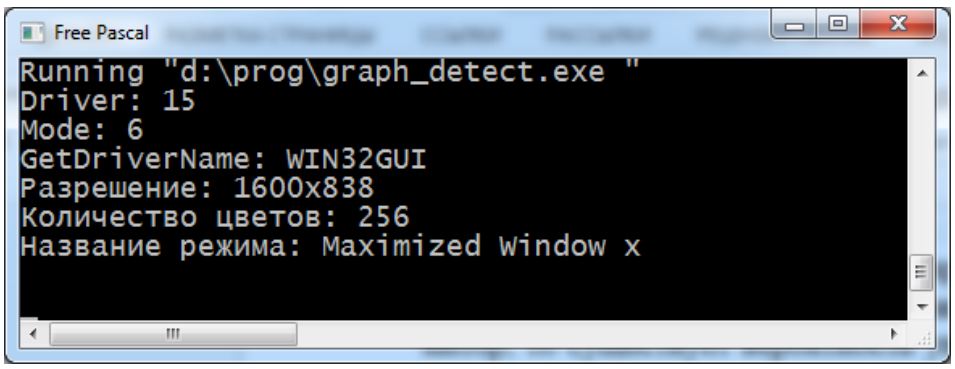
Это обусловлено тем, что в таком случае выбирается максимально возможное разрешение для видеокарты компьютера, а не для монитора. А так как часто видеокарта поддерживает большее разрешение, чем монитор, то существует вероятность увидеть пустой экран вместо выводимого графического изображения.

Для автоматического выбора `Driver` и `Mode`, обеспечивающих максимальное разрешение экрана и максимальное количество цветов при данном разрешении можно использовать процедуру DetectGraph.

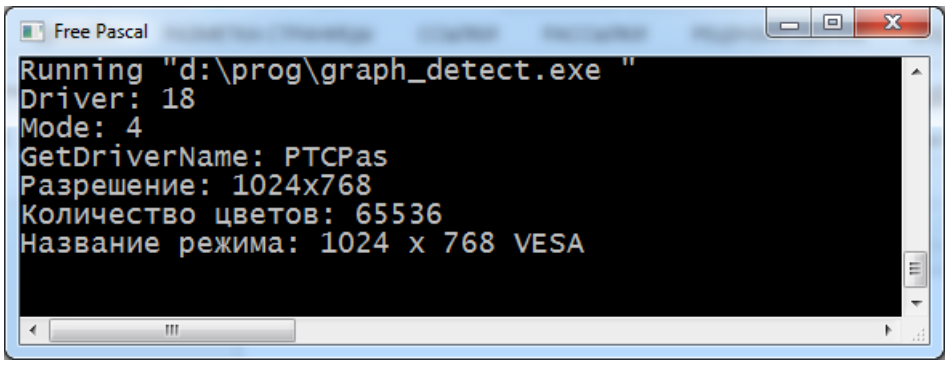
Приведем программу, которая позволит получить информацию об инициализированном режиме, `Driver` и `Mode` которого выбирается автоматически.

```
program Graph_detect;
uses Graph;
var
    Driver: SmallInt;
    Mode: SmallInt;
begin
  DetectGraph(Driver, Mode);
  InitGraph(Driver, Mode, '');
  Writeln('Driver: ',Driver);
  Writeln('Mode: ', Mode);
  Writeln('GetDriverName: ', GetDriverName);
  Writeln('Разрешение: ', GetMaxX+1,'x',GetMaxY+1);
  Writeln('Количество цветов: ', GetMaxColor+1);
  Writeln('Название режима: ', GetModeName(GetGraphMode));
  ReadLn;
  CloseGraph;
end.
```

Результат работы программы представлен на рис. ниже:



Если в приведенной выше программе вместо модуля `Graph` подключить модуль `ptcGraph`, то результат работы программы изменится:



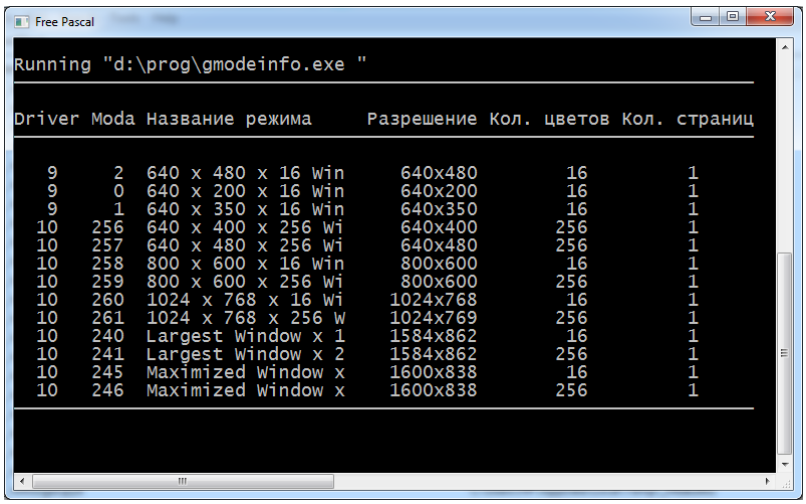
Чтобы узнать все графические режимы, которые поддерживает видеоадаптер конкретного компьютера можно воспользоваться функцией QueryAdapterInfo, возвращающей связанный список всех поддерживаемых видеоадаптером графических режимов с их подробным описанием. Элементы этого списка имеют тип запись TModeInfo, которая состоит из 36 полей. Описание типа TModeInfo можно посмотреть в справке Free Pascal: http://www.freepascal.org/docs-html/rtl/graph/tmodeinfo.html.

Приведем текст программы, которая выведет все доступные графические режимы.

```
program GmodeInfo;
uses Graph, SysUtils;
//uses ptcGraph, SysUtils;
//uses sdlGraph, SysUtils;
var
   ModeInfo: PModeInfo;
   St: String;
   Line:String;
begin
  FillChar(Line,80,'-');
  Line[0]:=#80;
  ModeInfo:=QueryAdapterInfo;
  if ModeInfo=NIL Then
     Writeln('Не удалось получить информацию у видеоадаптера')
  else
    begin
    Writeln(Line);
    Writeln ('Driver ', 'Mode ', 'Название режима ':28,
          'Разрешение ',  'Кол. цветов ', 'Кол. страниц');
    Writeln (Line);
    repeat
       Write(ModeInfo^.DriverNumber:4,' ':3);
       Write (ModeInfo^.ModeNumber:3,' ':2);
       Write (ModeInfo^.ModeName:28);
       St:=IntToStr(ModeInfo^.MaxX+1)+'x'+
```

```
        IntToStr(ModeInfo^.MaxY+1);
        Write (St:12);
        Write (ModeInfo^.MaxColor:10);
        Writeln(ModeInfo^.Hardwarepages+1:10);
        ModeInfo:=ModeInfo^.Next;
      until ModeInfo=NIL;
  Writeln(Line);
  Readln;
  end;
end.
```

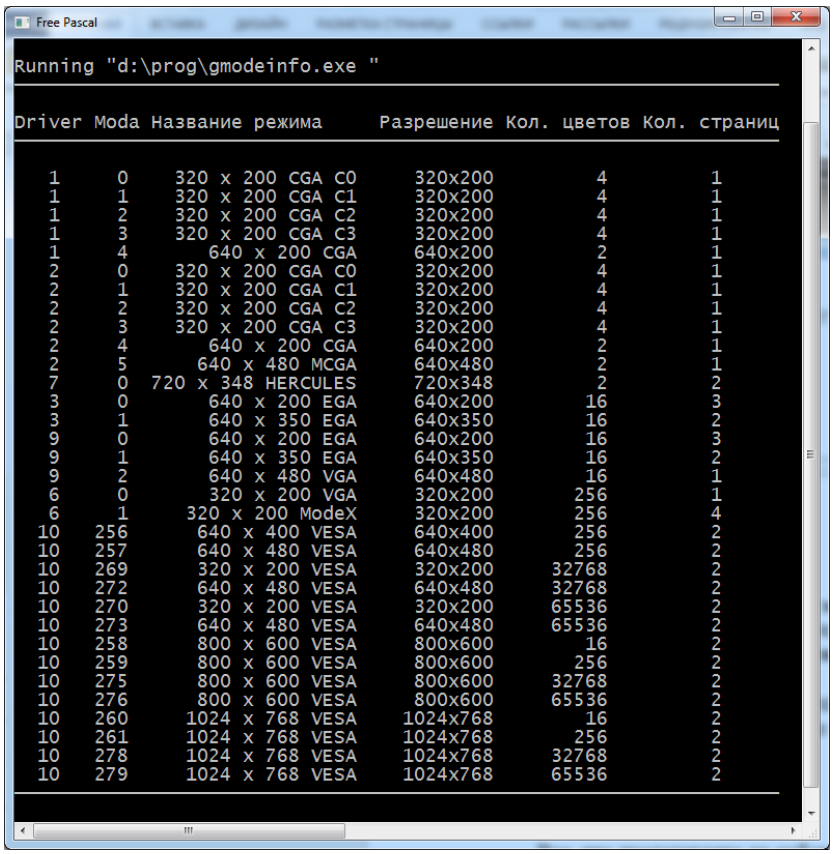
Результат работы зависит от компьютера.



Как видно, при подключении модуля `Graph` количество цветов не превышает 256 и доступна только одна видеостраница. Однако максимальное разрешение графического окна может достигать 1600×838, для монитора с разрешением 1600×900.

Если вместо модуля `Graph` подключить модуль `ptcGraph`, то результат работы программы на том же компьютере будет другим.

При использовании модуля `ptcGraph` максимальное разрешение графического окна получилось меньше, всего 1024×768, но увеличилось количество цветов до 65536 и количество видеостраниц до 2.



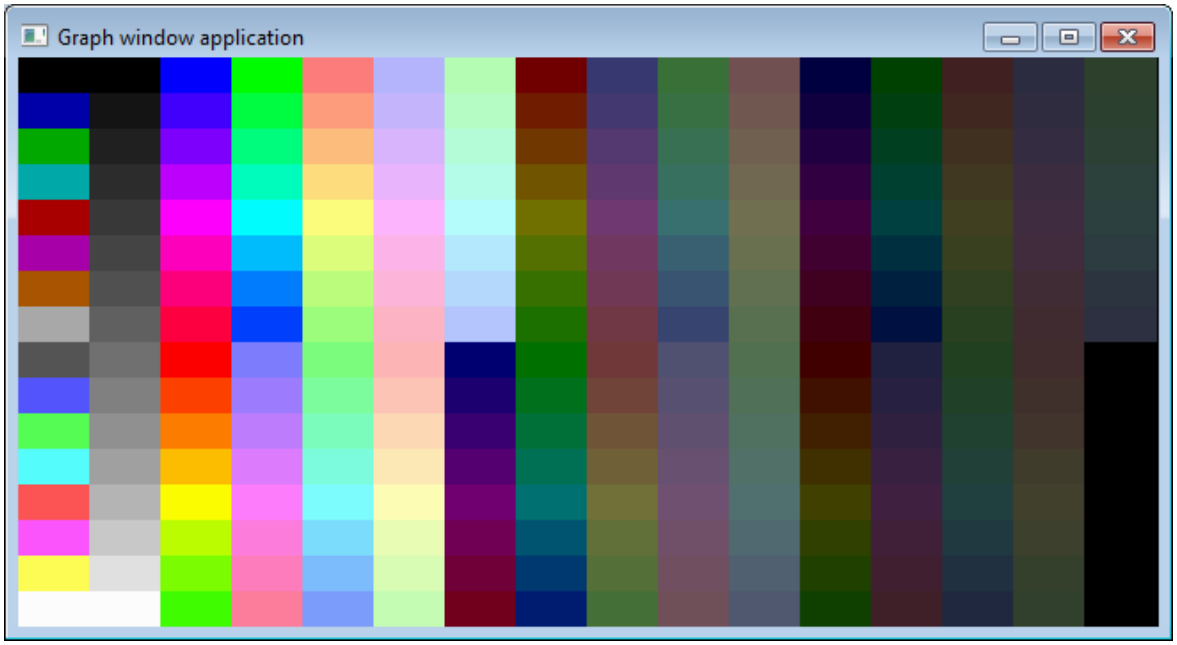
Особенности управления цветом в Free Pascal

При использовании модуля `Graph` становятся доступными 256 цветов, а при использовании `ptcGraph` – 65536. Возникает вопрос как отображается цвет, соответствующий некоторому числу. Для ответа на этот вопрос можно воспользоваться программой, в которой последовательно слева направо и сверху вниз выводятся на экран закрашенные прямоугольники цвета, соответствующего порядковому номеру прямоугольника:

```
program ShowColor;
uses Graph, SysUtils;
//uses ptcGraph, SysUtils;
var
    Driver: SmallInt;
    Mode: SmallInt;
   x,y,i,j,N,dX,dY : LongInt;
begin
DetectGraph(Driver, Mode);
InitGraph(Driver, Mode, '');
N:= Round(Sqrt(GetMaxColor+1));
dX:=( GetMaxX+1) div N;
dY:=( GetMaxY+1) div N; //dX:=dY;
Writeln('dX=',dX,' dY=',dY);
for i:=0 to N-1 do
    for j:=0 to N-1 do
        begin
```

```
            x:=i*dX;
            y:=j*dY;
            SetFillStyle(SolidFill,i*N+j);
            Bar(x,y,x+dX,y+dY);
            end;
Readln;
CloseGraph;
end.
```

Результат работы программы представлен на рис. ниже.



Если вместо модуля `Graph` подключить модуль `ptcGraph`, то можно будет увидеть палитру из 65536 цветов, но размер каждого прямоугольника будет очень маленьким, на тестируемом компьютере всего 4×3 пиксела.

1.3. Динамические структуры данных

Абстрактные типы данных

Абстрактный тип данных (АТД) – это математическая модель с совокупностью данных и операторов, определенных в рамках этой модели.

В модели АТД операндами операторов могут быть не только данные из данного АТД, но и других типов: стандартных типов языка программирования или из других АТД. Результат действия оператора также может иметь тип, отличающийся от типов данной модели АТД. Будем думать, что существует хотя бы один операнд или результат любого оператора, который имеет тип данных из модели АТД.

Заметим, что процедуры можно рассматривать как обобщение понятия оператора. В отличие от ограниченных по своим возможностям встроенных операторов языка программирования (сложения, умножения и т д.), при помощи процедур программист может создавать собственные операторы и применять их к операндам разных типов, не только базовых.

Примером такой процедуры-оператора может служить процедура перемножения матриц.

Если поместить (инкапсулировать) в отдельный модуль все операторы, отвечающие за определенный аспект функционирования программы, то их можно не только использовать в других программах, но и заменять в случае возникновения проблем при эксплуатации программы.

АТД можно рассматривать как обобщение простых типов данных (целых, действительных и т д.), также как процедура является обобщением простых операторов.

АТД инкапсулирует типы данных в том смысле, что определение типа и все операторы, которые выполняются над данными этого типа, содержатся в одном разделе программы.

Для АТД используются структуры данных, которые представляют собой набор переменных, объединенных определенным образом.

В качестве примера АТД можно взять файловый тип и множество операций, связанных с ним.

К основным АТД обычно относят наиболее общие абстрактные типы данных: списки (последовательности элементов); два специальных случая списков – стеки, где элементы добавляются и уничтожаются только на одном конце списка, и очереди, когда элементы добавляются на одном конце, а уничтожаются на втором; деревья, графы.

Общие сведения о динамических структурах данных

Между объектами материального мира, поведение которых моделируют программы, существуют разнообразные, постоянно изменяющиеся, связи. Одни объекты могут исчезать, другие – появляться. Понятно, что, когда в программе мы хотим моделировать группы с переменным числом объектов, между которыми связи могут быть изменчивыми, тогда нужны языковые средства для налаживания, изменения и расторжения связей между объектами, которые моделируется, а также для *рождения* и *уничтожения* объектов. Для этой цели в языке Pascal служат динамические переменные и указатели.

Самый простой способ связать множество элементов – это соединить их линейно в список, очередь или стек. А если элементы связаны рекурсивно, то их представляют деревом (примером является генеалогическое древо человека). Более сложные связи между элементами дают графы.

Динамические структуры данных появились тогда, когда использование регулярных структур (массивы, строки) начало тормозить написание программ несовершенным аппаратом.

Работа с массивами имеет свои преимущества и недостатки. Отметим их.

Преимущества:

- массивы помогают объединять совокупности сведений в осмысленные группы;
- элементы массива отличаются друг от друга только индексами;

- использование индексов обеспечивает непосредственный доступ к любому элементу массива;
- индексация позволяет проводить автоматическую, быструю, эффективную обработку элементов массива. Обработка – это инициализация, модификация, поиск и др.

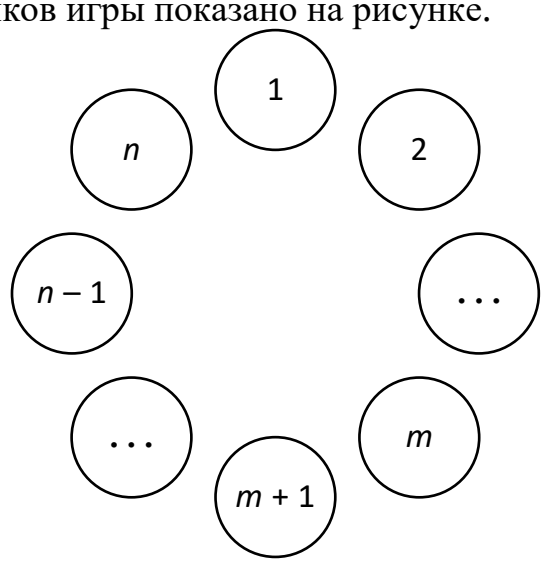
К *недостаткам* можно отнести операцию изменения чередования элементов массива, удаления или добавления элементов массива.

Значит, если какие-либо элементы нужно добавлять или удалять в совокупности, то такие абстрактные структуры данных плохо ложатся на массивы.

Задача о считалочке

Пусть в круг становятся n человек и получают номера *1*, *2*, ..., *n*, считая по часовой стрелке. Поскольку люди стоят по кругу, то после последнего сразу идет первый. Затем, начиная с первого, тоже по часовой стрелке отсчитывается m-ый человек и выводится из круга. После этого, начиная со следующего, снова отсчитывается m-ый человек и выводится из круга. Это повторяется $n-1$ раз. Победитель – тот, кто останется последним. Найти его номер.

Расстановка участников игры показано на рисунке.



Обсудим возможные варианты решения задачи.

Вариант 1. Используем массив на n человек и присвоим каждому элементу массива значение, равное значению его индекса.

Запрограммируем проход по элементам массива на $n-1$ раз так, что, сумев попасть на последний элемент, мы переходим на первый.

Получив при просмотре очередной элемент массива, который нужно вывести из игры (m-й по счету), можно поступить двояко:

1) заменить значение элемента, который выводится, на некоторую оценку (например, отрицательное число или логическое значение), чтобы при дальнейшем просмотре этот элемент не участвовал в игре;
2) вычеркнуть элемент из массива, подтягивая следующие элементы на один шаг вперед.

Элемент, который останется последним, и даст решение этой задачи.

С какими препятствиями нам придется встретиться при программировании этих подходов?

В первом случае нужно отметить выбывшие элементы, при очередном просмотре пропускать отмеченные элементы и после последнего элемента переходить на первый.

Рассмотрим при $n = 7$, $m = 5$ подробнее работу первого варианта.

В первой строке таблицы разместим индексы элементов, во втором – признак наличия элемента, в третьем – признак удаления элемента и то, на каком проходе он удаляется.

1	2	3	4	5	6	7
true	true	true	true	true	true	true
false$_{(6)}$	false$_{(3)}$	false$_{(2)}$	false$_{(4)}$	false$_{(1)}$	true	false$_{(5)}$

С элементом массива связан индекс. Нас интересует индекс неотмеченного элемента. Ответом будет 6. Программу при желании напишите самостоятельно.

Во втором случае нужно своевременно делать перестановку элементов и изменять количество имеющихся элементов.

Очевидно, что второй вариант можно запрограммировать рекурсией, так как количество элементов в массиве уменьшается, но вычеркивать нужно, как и ранее.

Однако, чтобы свести задачу к первоначальной (предыдущей), требуется перестановка элементов массива, чтобы отсчет начинать с первого.

Тогда получится следующая программа.

```
Program Recursia_Schitalochka;
const
       n = 7;
       m = 5;
type
       TItem = Integer;
       TMas  = array[1..n] of TItem;
var    a      : TMas;
       kol, i : Integer;
 function  Last(n,m : Integer; a : TMas) : Integer;

   function  Rec(n,m:Integer) : Integer;

     procedure Zryx;
     var i,j : Integer;
         m1  : Integer;
         z   : TItem;
     begin           {procedure Zryx}
      m1 := m;
      if m > n then  m1 := (m - 1) mod n + 1;
        for i := 1 to n - m1 do
           begin
            z := a[n];
```

```
            for j := n - 1 downto 1 do
                a[j + 1] := a[j];
            a[1] := z;
          end;
    end; {procedure Zryx}

 begin             {function Rec}
  if n = 1 then Rec := a[n]
  else
     begin
      if n <> m then Zryx;
      Rec := Rec(n - 1, m);
     end;
 end; {function Rec}

begin {function Last}
 Last := Rec(n, m);
end;{function Last}

begin
 for i := 1 to n do a[i] := i;
 Writeln('зачеркивая ',m, '-го из ', n,
        ' участников, останется: ', Last(n, m, a));
 Readln;
end.
```

Мы знаем, что рекурсия требует много дополнительных ресурсов, и тут пришлось довольно часто выполнять перестановки элементов вспомогательного массива, поэтому рассмотрим другие подходы.

Эти препятствия отнимают время и создают определенные трудности при программировании.

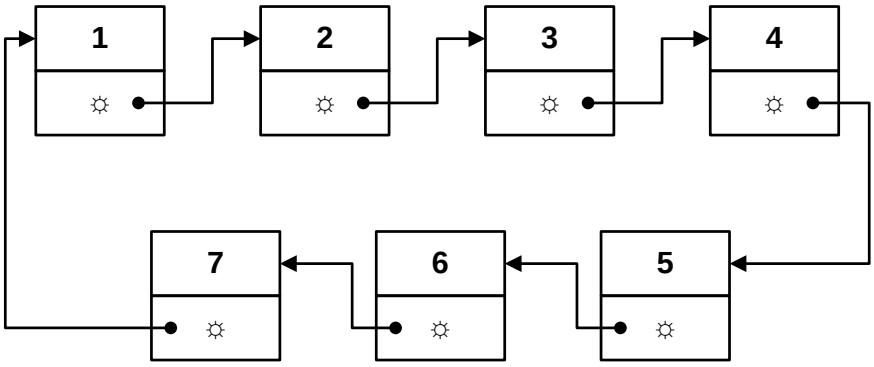
Вариант 2. Уже на этой задаче видно, что нужно иметь какие-то новые структуры данных, которые позволяли бы проще устанавливать и разрывать связи между элементами во время выполнения программы. А это уже динамические структуры данных и, следовательно, нужно использовать указатели, значениями которых являются адреса динамических переменных.

Введем элемент, объединяющий два поля: номер элемента и ссылку на следующий. Разместим элементы в круг и отобразим стрелками их связи.

Сейчас промоделируем ход считалочки, используя указатели. Начиная с первого, пропустим $m-1$ элемент и изменим связь: $(m-1)-$ый элемент соединим с $(m+1)$-ым. Продолжим $n-1$ раз процесс поиска и вычеркивания. В круге останется один элемент. Это и будет победитель.

Такое представление более точно отражает действительную ситуацию: люди стоят в кругу и между ними существуют соседские связи, а потом связи постепенно меняются, если кто-то выбывает из круга.

Упражнение. Переосмыслите ход считалочки на примере, явно разрывая связи с выбывшими кандидатами.



Чтобы запрограммировать последний вариант, определим тип Kandidat, характеризующий элемент-запись из считалочки и сохраним номер кандидата и адрес (указатель на) человека, который стоит следующим. Определим также тип – указатель на тип Kandidat. В первой части процедуры построим цепочку с участниками. Во второй части организуем цикл на удаление.

Получим следующую программу.

```
Program Schitalochka;

var
   m, n, l : Integer;
  procedure Last(n,m:Integer; var l:Integer);
  type
      Ptr_kand = ^Kandidat;
      Kandidat = record
                  Num  : Word;
                  Next : Ptr_kand;
                 end;

  var
     First, Current, Newitem : Ptr_kand;
     i, j                    : Integer;
  begin
   New(First);
   First^.Num := 1;
   Current    := First;
   for i := 2 to n do
               { создание ряда кандидатов}
               { и налаживание между ними связи}
     begin
      New(Newitem);
      Newitem^.Num  := i;
      Current^.Next := Newitem;
      Current       := Newitem;
     end;
  {после выхода из цикла указатели Current и Newitem
            сохраняют ссылку на последнего кандидата }
   Current^.Next := First;         {завершаем цепочку}
   for i := 2 to n do
```

```
    begin         {образуем отсчет (m-1)-го человека}
     for j := 1 to m - 1 do
           Current := Current^.Next;
     Current^.Next := Current^.Next^.Next;
           { этим вывели человека, который был m-м}
    end;
  l := Current^.Num;
 end;

begin
 n := 7;   m := 5;
 Last(n, m, l);
 Writeln(l);
end.
```

Задание 1. Запрограммируйте вариант считалочки, если у какого-то из участников существует на 1 раз «индульгенция» на не выбывание.

Задание 2. Запрограммируйте вариант считалочки, если у некоторых из участников существует на несколько раз «индульгенция» на не выбывание.

Задание 3. Получите номера выбывающих кандидатов в порядке их выбывания.

Списки и их классификация

Основные положения

Списком называют такую структуру данных, каждый элемент которой содержит ссылку, связывающую его с последующим элементом – *узлом* (*звеном*) списка.

Различают списки:

- линейные;
  - линейные с одной связью;
  - линейные с несколькими связями;
- кольцевые (циклические);
  - кольцевые с одной связью;
  - кольцевые с несколькими связями;
- ассоциативные;
- иерархические.

Существуют два метода хранения списков – последовательный и связный.

При *последовательном* хранении элементы списка (или узлы списка) располагаются в массиве зафиксированного размера. Этот прием используется, когда нельзя работать с динамической памятью.

При *связном* хранении узлы списка располагаются динамически в Heap при помощи указателей. Для организации связного списка, как мы видели выше при решении задачи на считалочку, используются узлы списка – записи, состоящие

из двух частей. Одна часть – тело узла списка – имеет информацию, подлежащую обработке, вторая часть – ссылочная – содержит указатель на следующий узел списка.

Связные списки

Связные списки обладают одним очень важным преимуществом: операции вставки и удаления принадлежат по времени обработки классу $O(1)$ ($O(1)$ – константа, не зависящая от количества внешних данных), так как для таких действий всегда требуется одно и то же время.

Основным недостатком связных списков является то, что получение доступа к их узлам принадлежит классу $O(n)$, так как при поиске n-го узла мы начинаем с некоторой позиции в списке и переходим по ссылке к искомому узлу. Чем больше узлов в списке, тем больше переходов нужно совершить.

Для увеличения скорости доступа к узлам в некоторых частных случаях используются стеки и очереди.

По сравнению с массивами списки требуют большего размера памяти, так как узел списка содержит указатель или указатели на следующий узел.

Как и массив, связный список является универсальной структурой данных, которая широко используется программистами. Однако в отличие от массива связный список не входит в состав стандартного языка Pascal.

Тем не менее, в Pascal создать связный список довольно просто. Все, что для этого нужно, – иметь в составе языка указатель.

По своей сути связный список – это цепочка узлов или звеньев с некоторыми описаниями, образующими информационную часть узла списка. При этом каждый узел содержит указатель, который указывает на следующий узел в списке или равен `nil`, если он последний. В случае кольцевого списка последний должен ссылаться на первый.

Сам список начинается с первого узла, от которого путем последовательных переходов по ссылке можно обойти все остальные узлы. В связном списке узлы могут быть разбросаны по разным местам памяти, а их чередование определяется ссылками. Память под каждый узел в `Heap` выделяется отдельно.

Чем же связный список отличается от массива?

Первое, что нужно отметить, – размер связного списка не устанавливается.

Для массива всегда надо знать заранее, сколько элементов будет в нем храниться (чтобы можно было статически выделить непрерывный участок памяти для его хранения), или разработать некоторую схему расширения массива (или его сокращения), чтобы массив смог поместить большее (или меньшее) количество элементов.

В массиве каждый следующий элемент находится рядом с предыдущим.

Выделение памяти под n элементов массива фактически представляет собой операцию класса $O(1)$: все элементы должны находиться в одном непрерывном блоке памяти, поэтому одновременно выделяется сразу весь блок. В массиве доступ к n-ному элементу требует проведения простых арифметических подсчетов адреса памяти. Эта операция класса $O(1)$.

Прежде чем начать описание операций со связным списком, рассмотрим, как каждый узел списка будет представляться в памяти. Знание структуры узла позволит нам более детально рассмотреть основные операции со связными списками.

Структура узла списка выглядит следующим образом: тип узла – это запись, в котором хранятся данные и указатель на следующий узел списка.

Действия со списками

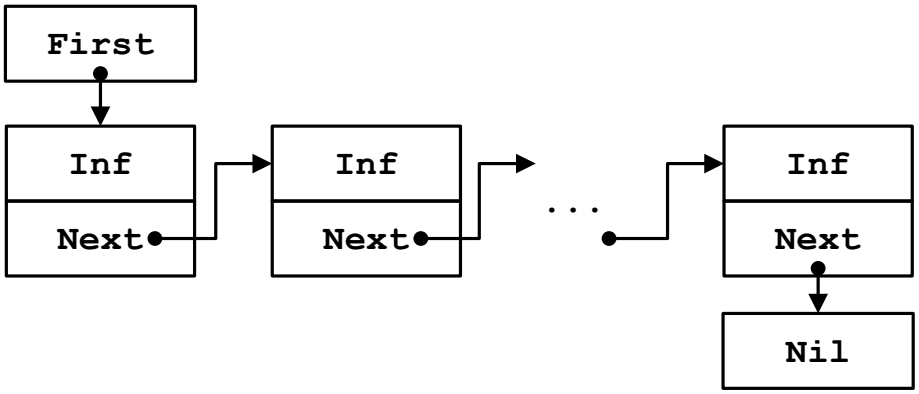
При работе со списками чаще всего приходится выполнять следующие действия:

- найти узел с заданным свойством;
- удалить узел;
- определить по номеру нужный узел;
- добавить узел;
- распечатать узлы списка;
- упорядочить узлы списка по некоторому ключу и прочее.

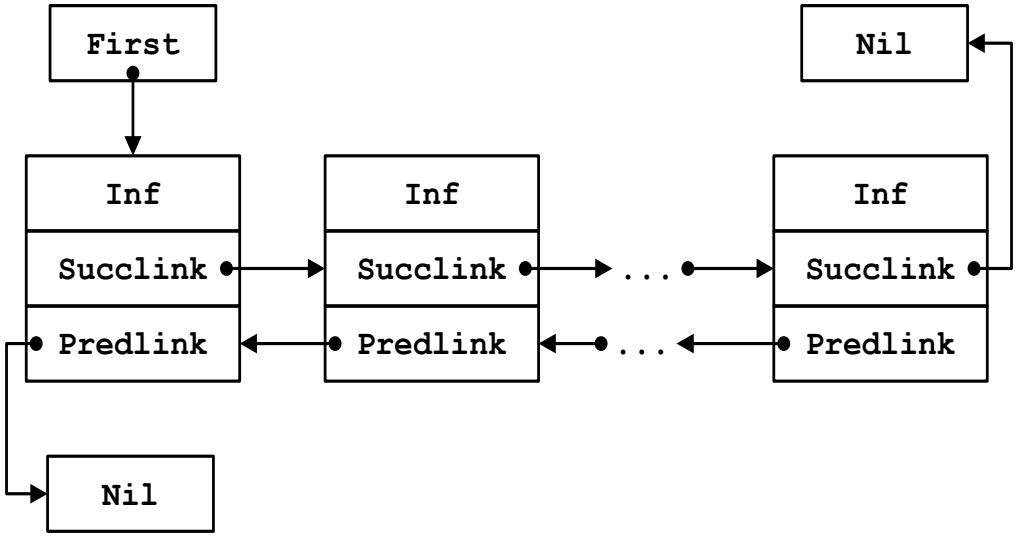
Типы списков по методам доступа к узлам

Приведем графическую интерпретацию методов связного хранения некоторых простых типов списков.

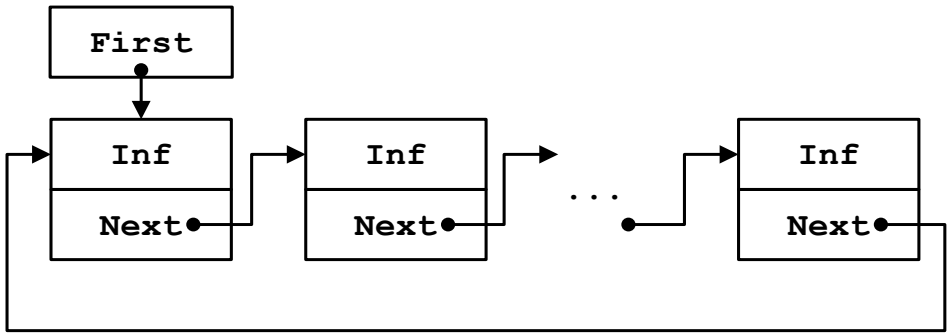
1. Обычный линейный список с одной связью:



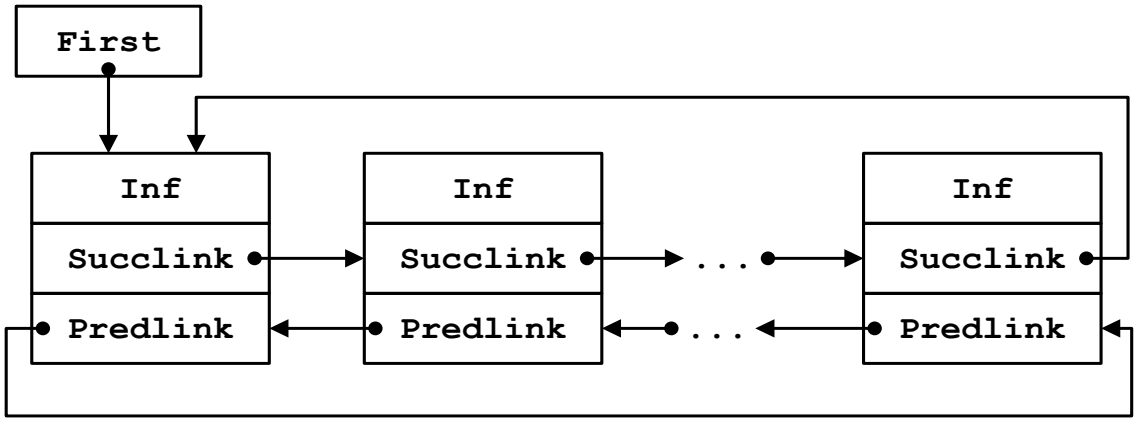
2. Обычный линейный список с двумя связями:



3. Обычный линейный циклический список с одной связью:



4. Обычный линейный циклический список с двумя связями:

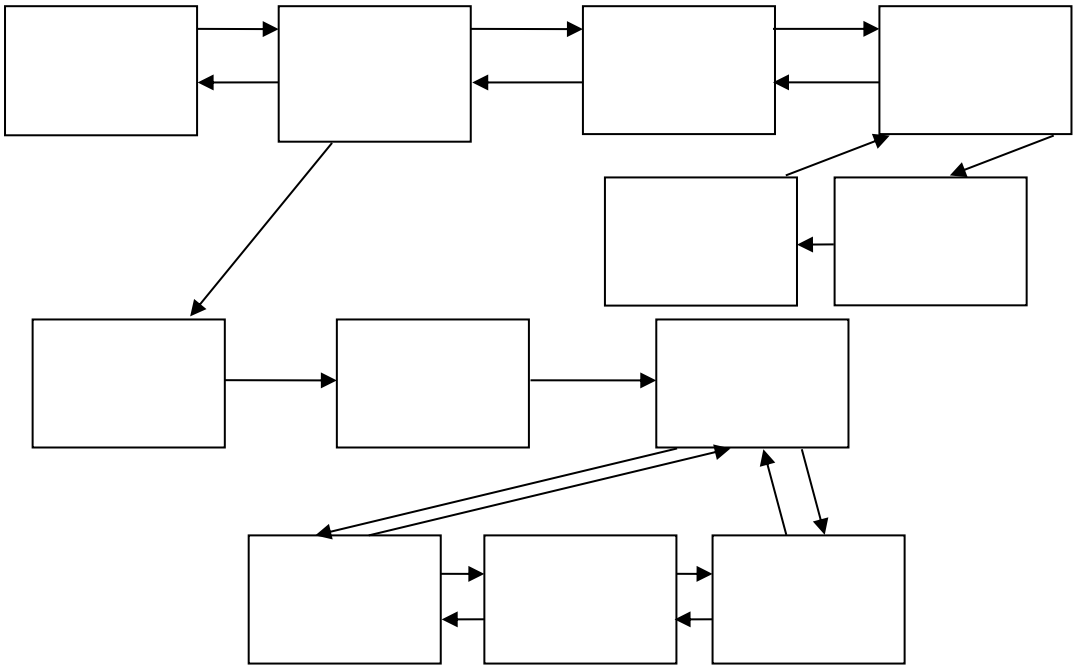


Во всех этих списках доступ возможен к любому узлу списка. Если в линейном списке добавление узла происходит в конце списка, а уничтожение – с начала списка, тогда такой список называется *очередью*. Если же в линейном списке добавление узла происходит в начало списка и уничтожение идет с начала списка, тогда такой список называется *стеком*.

Ассоциативные списки организуются на одном общем наборе узлов, причем каждый из подсписков объединяет в определенном порядке только те узлы из этого набора, которые обладают заданным им характеристическим свойством. Значит,

узел такого списка должен содержать столько ссылочных частей, сколько подсписков предполагается создать на базе общего набора записей.

Иерархические списки – более сложная структура. Их можно представить, например, следующей схемой:



Информационная часть в иерархических списках – тело узла списка – может образовывать группу записей с определенным внутренним порядком, например списком. Иерархическим списком можно представить список высших учебных заведений, каждое из которых имеет свои списки факультетов, а факультеты – списки курсов и т.д. Такая структура данных наиболее близко отражает связи в реальных задачах.

Ресурсы по работе со списками лучше объединить в отдельном модуле и этим образовать АТД «линейный список» и т. п.

Для работы со списками желательно предусмотреть следующие действия:

- размещение узла списка в динамической памяти;
- инициализация информационной части узла;
- включение узла в нужное место списка;
- поиск определенного узла списка;
- распечатка информационной части узла списка;
- распечатка информационных частей всех узлов списка – распечатка списка;
- нахождение и удаление конечного числа узлов списка;
- сортировка узлов списка по некоторому ключу из информационной части.

Однонаправленные связные списки

Пусть надо работать со списком, который содержит сведения о каком-то автомобиле. Информационная часть узла такого списка будет содержать сведения о марке автомобиля, номере двигателя, шасси, государственном номере регистрации, владельце и т.д. Но если мы рассматриваем собственно работу с

узлами списка, то эта информация для нас несущественна. Поэтому сделаем в модуле только такие описания:

```
unit  Spis_New;
interface
type
      TInf   = record ... end;
      TLink  = ^TZveno;
      TZveno = record
                 Inf  : TInf;
                 Next : TLink
               end;
procedure  Print_inf (a : TZveno);
procedure  Init_inf  (a : TZveno);
procedure  Sozd_spis (var   First : TLink);
procedure  Show_spis (const First : TLink);
           .....
implementation
.....
end.
```

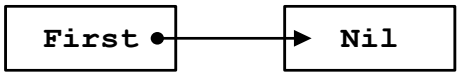
Здесь одно из полей записи предназначено для связывания узлов списка в единое целое – однонаправленный связный список. Это поле – `Next : TLink`. Структура связного списка предполагает, что очередной узел в списке через этот указатель связывается с последующим узлом. Последний узел списка имеет в поле `Next` указатель на `nil`.

Указатель на первый узел списка (и тем самым на весь список целиком) содержится в переменной `First`.

Алгоритм создания списка

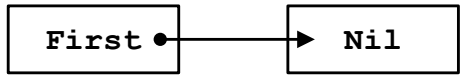
Это тривиальная задача. Добавлять узлы в создаваемый список можно перед первым или после последнего. Рассмотрим следующий алгоритм, основанный на добавлении нового узла первым.

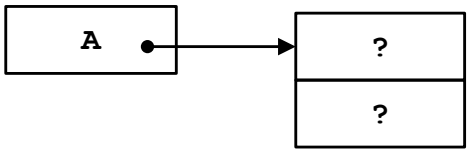
1. Изначально, когда список пуст, указатель `First` устанавливается в `nil`.



2. Затем в динамической области под указатель `A` выделяется память.

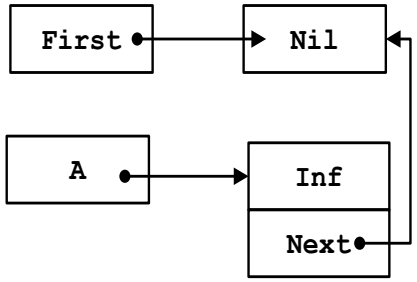
```
A := New(TLink);
```





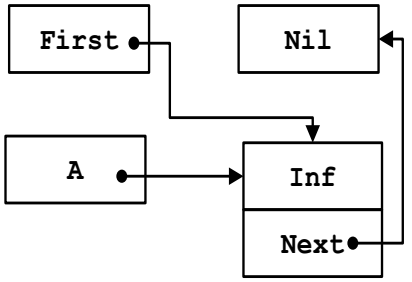
3. Информационная часть `A^`, например, при помощи процедуры `Init_inf`, наполняется информацией.

4. Для выполнения операции вставки узла в список требуется ссылочному полю `Next` присвоить ссылку на `First` (первоначально фактически на `nil`).



5. Меняя значение указателя `First`, узел вставляется в список.

```
First := A;
```



Далее указатель `A` можно использовать для других целей или освободить память, выделенную под него.

Пункты 2 - 5 повторяются.

Рассмотрите самостоятельно алгоритм, основанный на добавлении последним.

Выделим далее операции по работе с одиночным узлом. На основании предыдущих алгоритмов становится очевидным, что для осуществления таких действий требуется выполнить некоторую работу с указателями. Оформим нужные действия в виде процедур, которые далее можно использовать как операторы при работе с АТД «линейный список».

Вставка узла в список

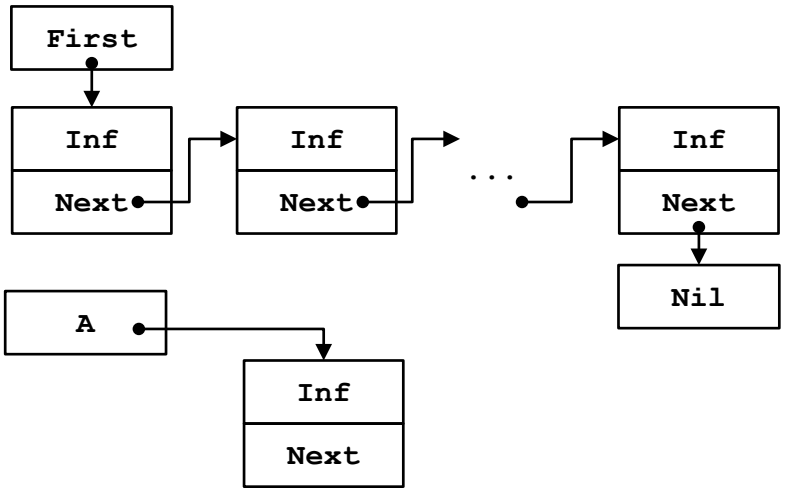
При вставке очередного узла в список могут возникнуть следующие ситуации:

- включить первым;
- включить в середину;
- поместить в конец списка последним, но не единственным.

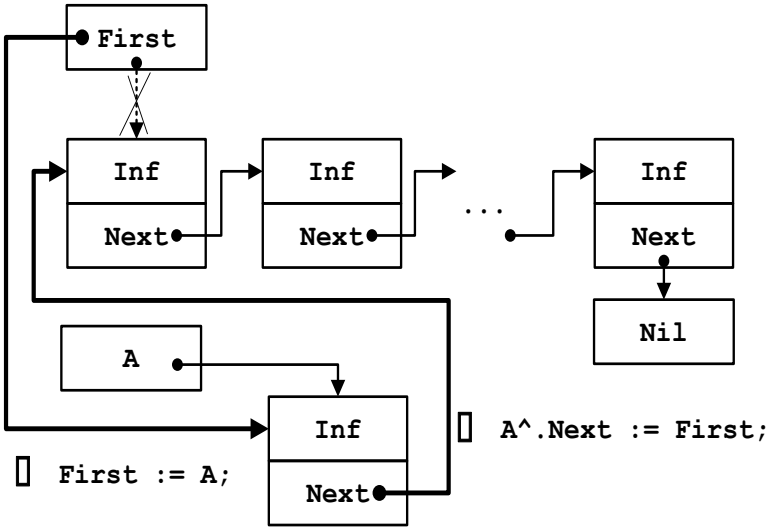
Этим случаям соответствуют следующие процедуры.

Включить узел в список первым

Рассмотрим включение очередного узла в начало списка. Исходное состояние будет следующим:



Этапы включения очередного узла A в начало списка приведены ниже:



Соответственно напишем процедуру AddFirst для включения очередного узла A в начало списка.

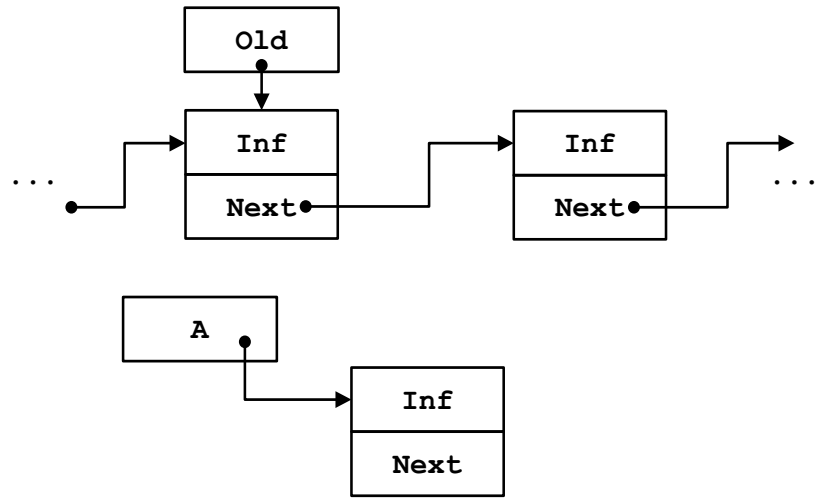
```
procedure AddFirst (var  A, First : TLink);
begin
     A^.Next := First;              {(1)}
     First   := A;                  {(2)}
end;
```

Включить узел в середину

Пусть Old – указатель, который указывает на место вставки. Рассмотрим две процедуры включения, так как здесь могут возникнуть такие ситуации, когда нужно включить:

- после узла, на который указывает указатель Old (процедура AddAfter);
- перед узлом, на который указывает указатель Old (процедура AddBefore).

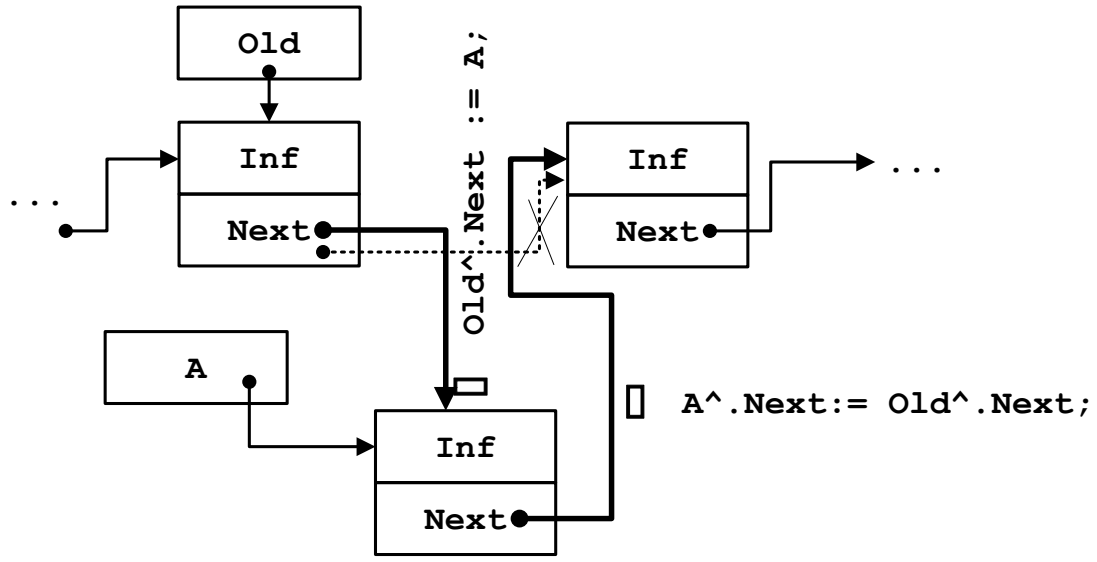
Исходное состояние для включения узла в середину списка:



Для включения после узла, на который указывает указатель `Old`, получим процедуру `AddAfter`.

```
procedure  AddAfter(var  A, Old : TLink);
begin
   A^.Next   := Old^.Next;          {(1)}
   Old^.Next := A;                  {(2)}
end;
```

Результат выполнения операции включения в середину списка после узла, на который указывает указатель `Old`, будет следующий:

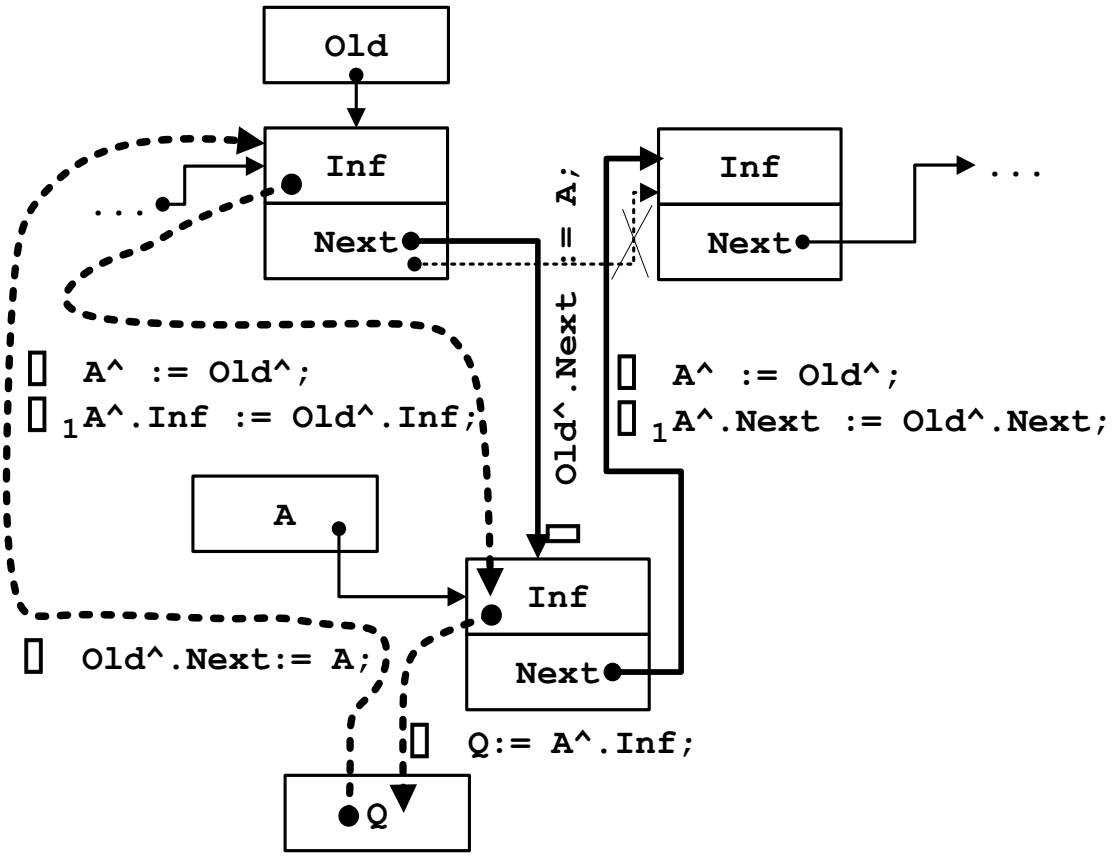


Если нужно включить очередной узел перед узлом, на который указывает `Old`, то однонаправленная цепочка связей создает трудность, поскольку нет доступа к узлу, который предшествует данному (`Old`). Тогда можно предложить такой прием:

1) содержимое `A^.Inf` переслать во вспомогательный элемент `Q`;

2) на место содержимого `A^` послать содержимое `Old^`;

3) на место содержимого `Old^.Inf` послать вспомогательный элемент `Q`;
4) установить ссылку `Old^.Next = A`.
Покажем это:



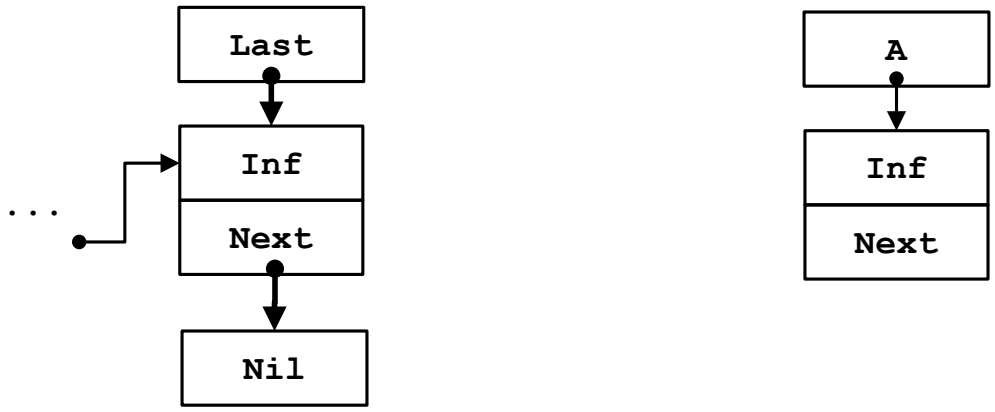
В результате выполнения операции включения очередного узла `A` в середину списка перед узлом, на который указывает `Old`, получим следующую процедуру.

```
procedure AddBefore(var  A, Old : TLink);
var   Q : TInf;
begin
   Q         := A^.Inf;              {(1)}
   A^        := Old^;                {(2)}
   Old^.Inf  := Q;                   {(3)}
   Old^.Next := A;                   {(4)}
end;
```

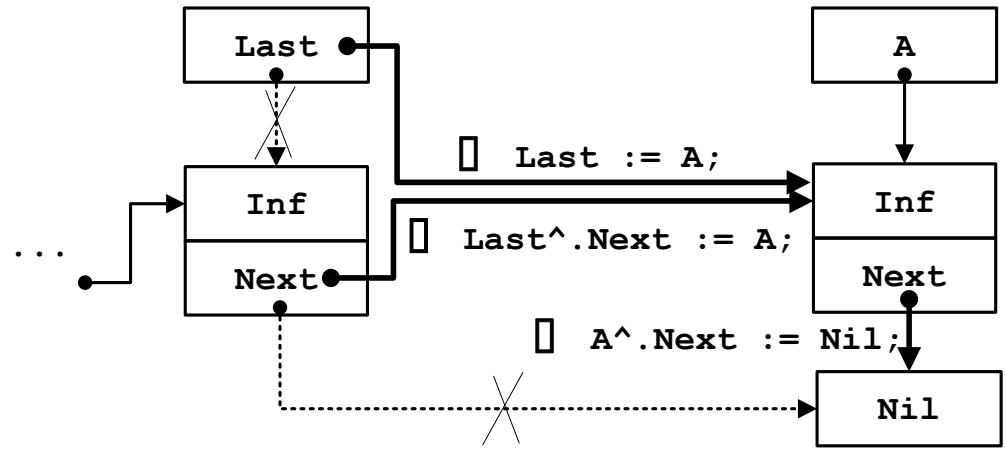
Поместить узел в конец списка последним, но не единственным

Рассмотрим включение очередного узла в конец списка последним, но не единственным. Ссылка на последний узел находится в указателе `Last`.

Исходное состояние:



Этапы включения очередного узла A в конец списка последним, но не единственным:



Соответственно получим следующую процедуру включения очередного узла A в конец списка последним, но не единственным.

```
procedure AddLast (var  A, Last : TLink);
begin
     Last^.Next := A;                {(1)}
     A^.Next    := nil;              {(2)}
     Last       := A;                {(3)}
end;
```

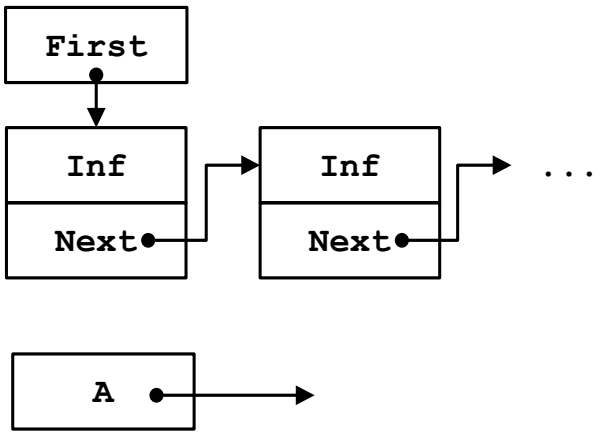
Удаление узла списка

При удалении узла списка также могут возникнуть различные ситуации:

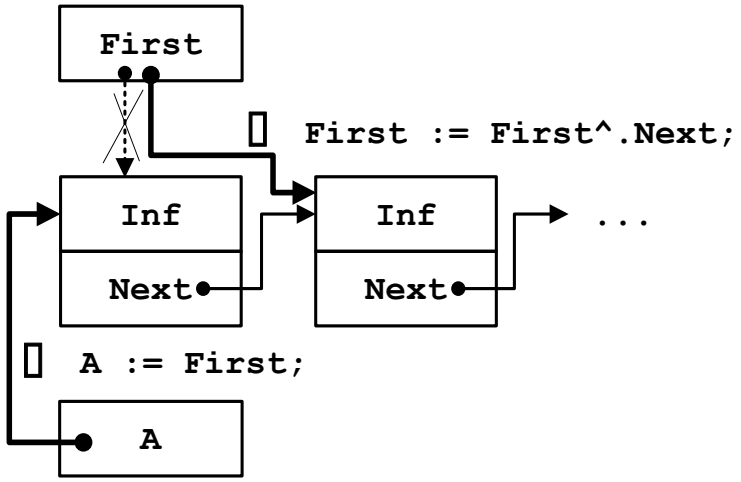
- удаление первого узла списка (DelFirst);
- удаление узла в середине списка:
    - узел, который удаляется, стоит после узла Old (DelAfter);
    - удаление самого Old (DelCurrent);
- удаление последнего узла (DelLast).

Удаление первого узла списка

Рассмотрим удаление первого узла списка. Исходное состояние:



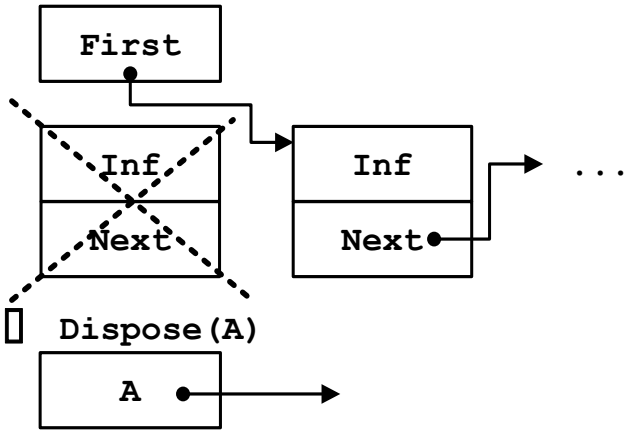
Этапы удаления первого узла списка:



Соответственно получим следующую процедуру удаления первого узла списка.

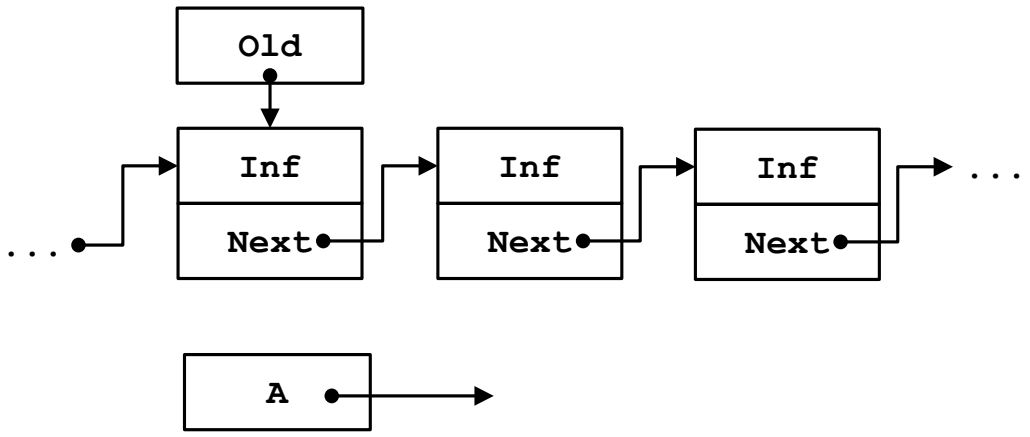
```
procedure  DelFirst(var First, A : TLink);
begin
      A     := First;                    {(1)}
      First := First^.Next;              {(2)}
end;
```

Далее в вызывающей программе можно уничтожить (`Dispose(A)`) объект, на который ссылается указатель `A` или использовать его для других целей:

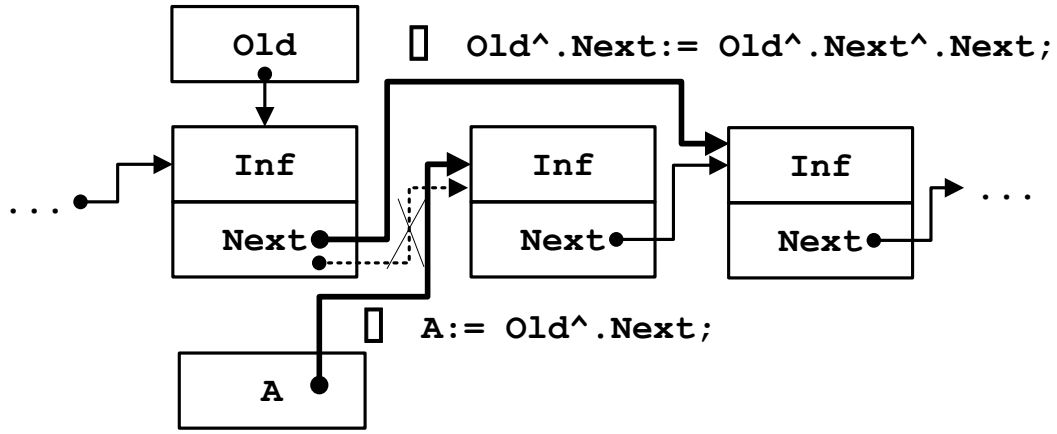


Удаление узла в середине списка после узла Old

Рассмотрим удаление узла в середине списка после узла `Old`. Исходное положение:



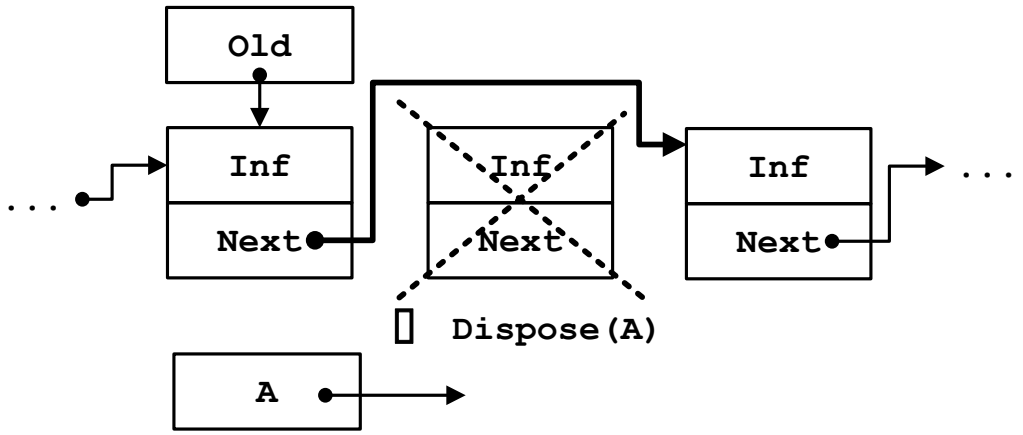
Этапы удаления узла в середине списка после узла `Old`:



Предложенную схему удаления узла в середине списка после узла `Old` опишем в процедуре `DelAfter`.

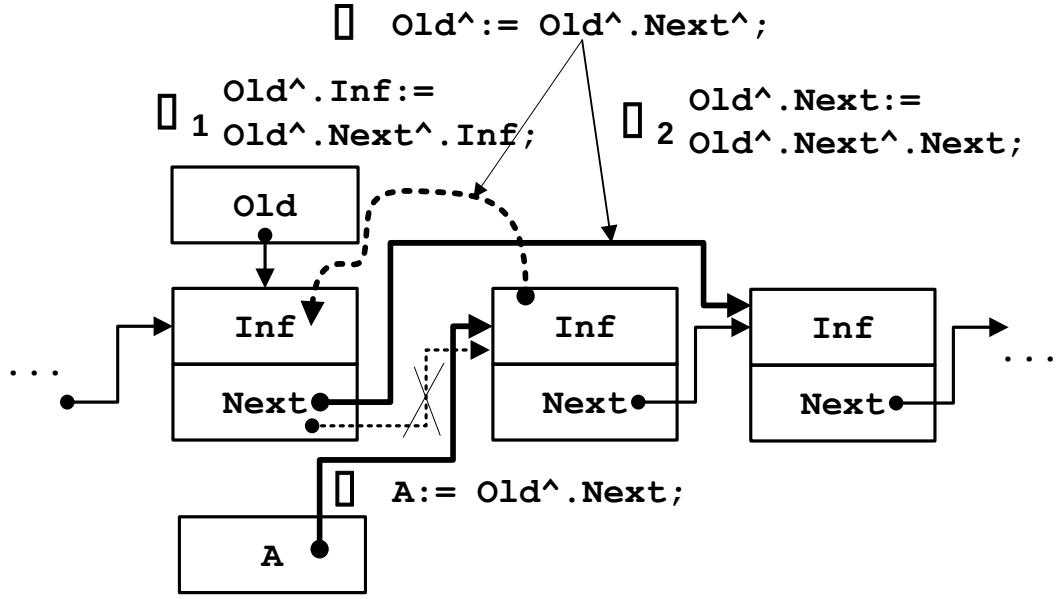
```
procedure  DelAfter (var  Old, A : TLink);
begin
     A         := Old^.Next;             {(1)}
     Old^.Next := Old^.Next^.Next;       {(2)}
end;
```

Далее в вызывающей программе можно уничтожить (`Dispose(A)`) объект, на который ссылается указатель `A`:



Удаление текущего узла в середине списка

Труднее удалить сам элемент Old, а не следующий за ним, так как обращение к узлу, который предшествует Old, невозможно. Если узел, на который указывает Old, последний в списке, то удалить его тоже невозможно, потому что нужно знать ссылку на предпоследний узел. Если же узел, на который указывает Old, не последний в списке, то удалить его можно: запомним в A местоположение следующего узла, перешлем содержимое следующего узла вместо содержимого узла, который удаляется, и получим решение задачи.



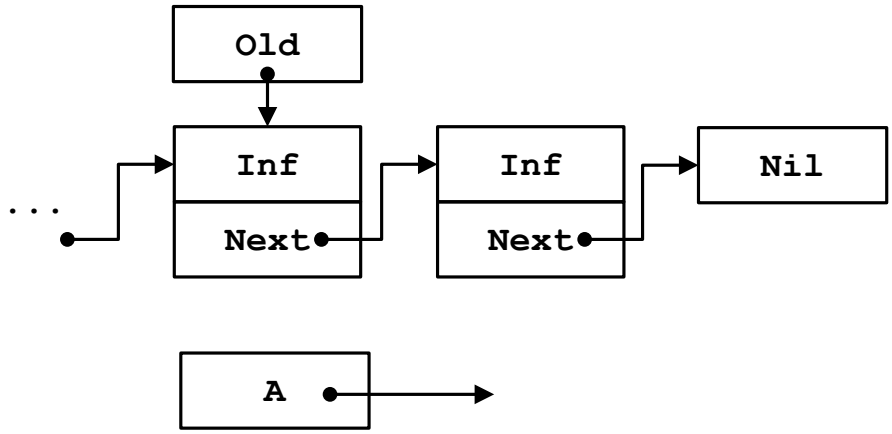
Получим такую процедуру.

```
procedure  DelCurrent (var  Old, A : TLink);
begin
      A    := Old^.Next;              {(1)}
      Old^ := Old^.Next^;             {(2)}
end;
```

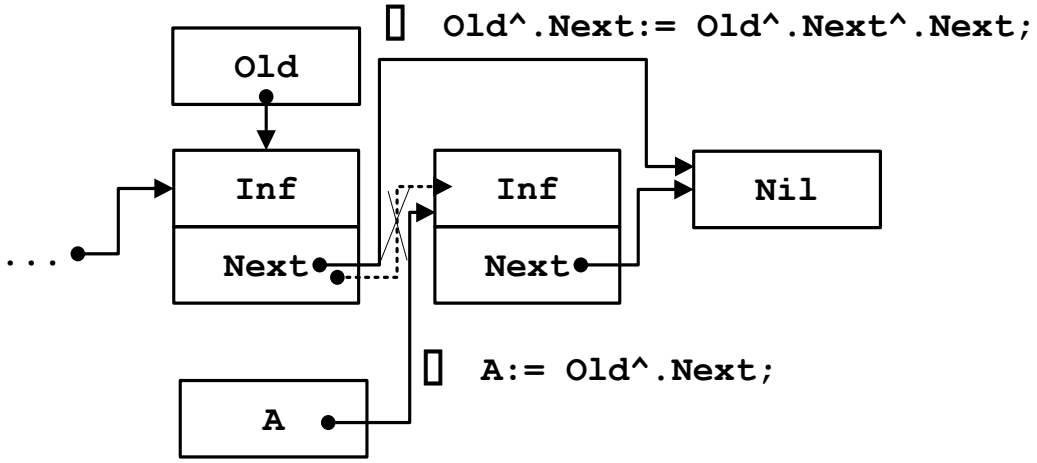
Далее в вызывающей программе можно уничтожить (Dispose) объект, на который ссылается указатель A.

Удаление последнего узла списка

Рассмотрим удаление последнего узла из однонаправленного списка. Исходное состояние:



Этапы удаления последнего узла списка:



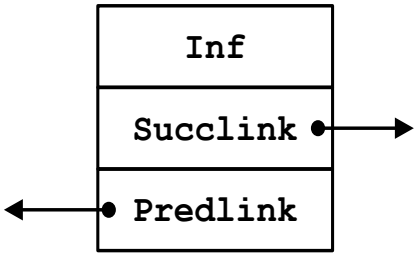
Соответственно получим следующую процедуру удаления последнего узла списка.

```
procedure DelLast(var Old, Last, A : TLink);
begin
   A         := Old^.Next;
   Old^.Next := Old^.Next^.Next;
   Last      := Old;
end;
```

Далее в вызывающей программе можно уничтожить (Dispose) объект, на который ссылается указатель A.

Двунаправленные связные списки

В двунаправленном списке каждый узел имеет два указателя: `Succlink` описывает связь узла со следующим, `Predlink` – с предыдущим.



Для работы с двунаправленным списком введем следующие типы данных:

```
type
        TInf   = record ...  end;
        TLink  = ^TZveno;
        TZveno = record
                   Inf      : TInf;
                   Succlink : Tlink;
                   Predlink : Tlink

                 end;
```

Ресурсы по работе с двунаправленным списком лучше объединить в отдельном модуле и этим образовать АТД «двунаправленный линейный список». Желательно предусмотреть следующие действия:

- размещение узла списка в динамической памяти;
- инициализация информационной части узла;
- включение узла в нужное место списка;
- поиск определенного узла списка;
- распечатка информационной части узла списка;
- распечатка информационных частей всех узлов списка – распечатка списка;
- нахождение и удаление конечного числа узлов списка;
- сортировка узлов списка по некоторому ключу из информационной части.

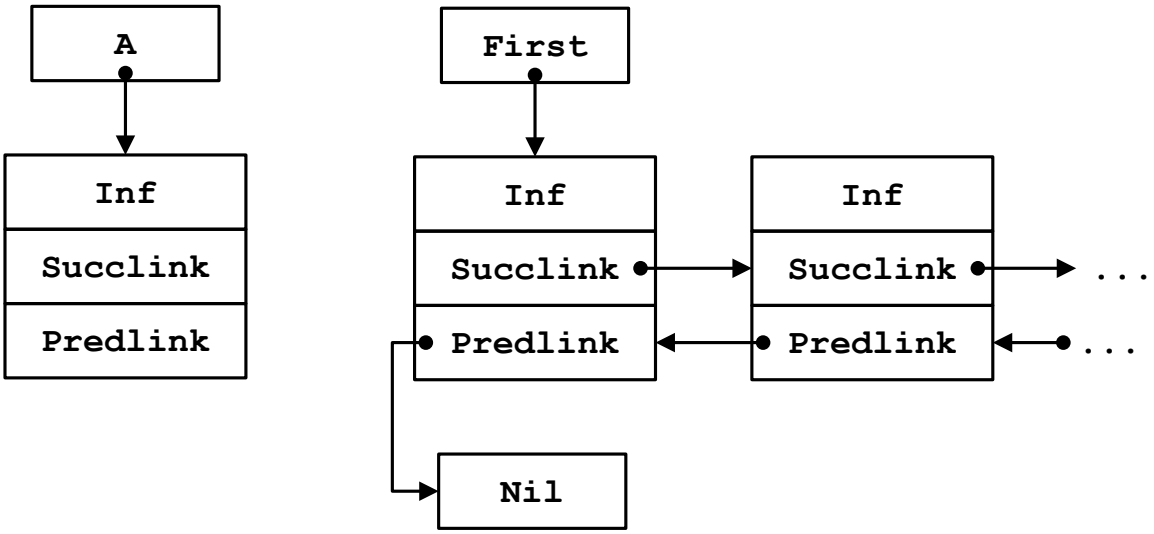
Приведём основные операции для работы с двунаправленным списком.

Процедуры включения узла в двунаправленный список

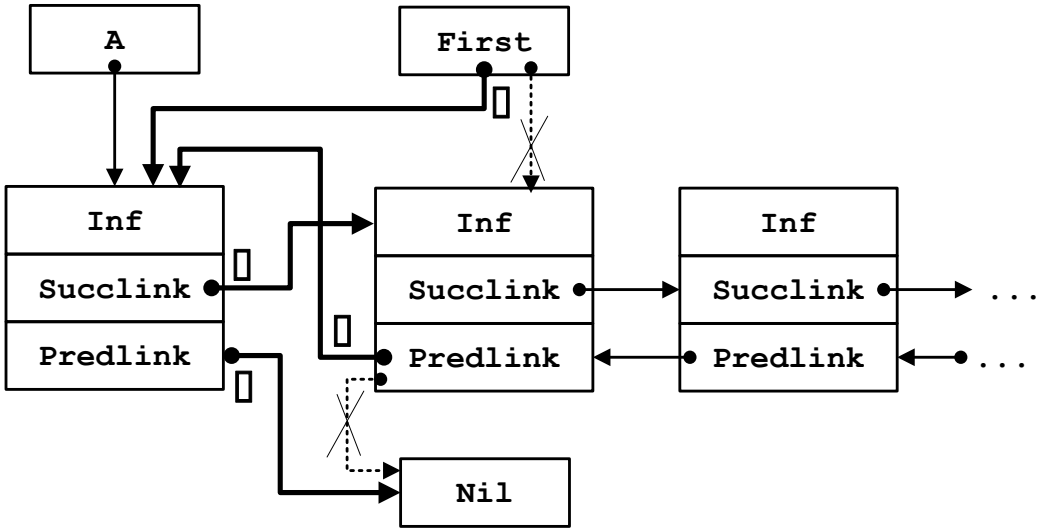
Рассмотрим процедуры *включения* узла в двунаправленный список.

Включение узла первым

Исходное состояние:



Этапы включения очередного узла A в начало двунаправленного списка:

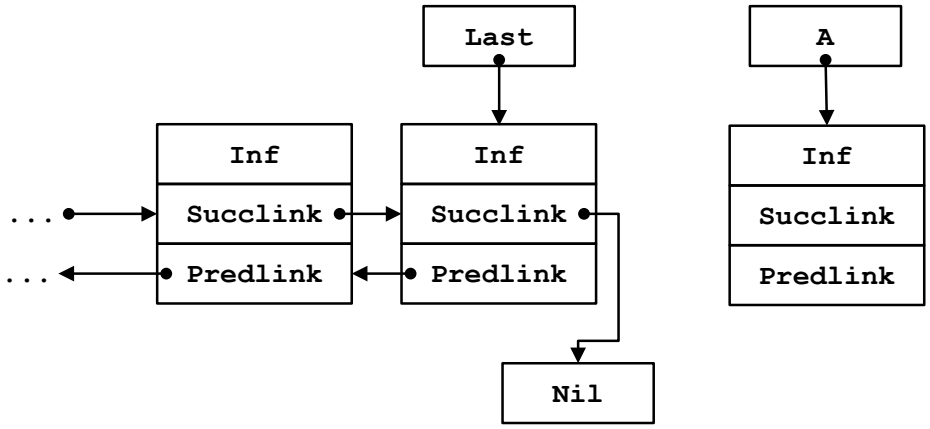


Соответственно напишем процедуру AddFirst_2 для включения очередного узла A в начало двунаправленного списка.

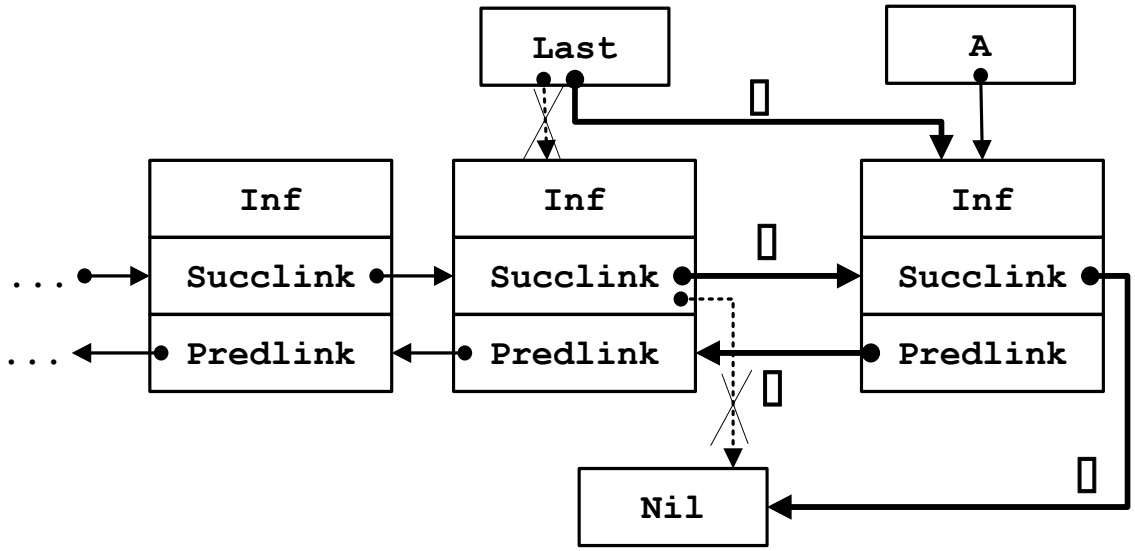
```
procedure AddFirst_2(First, A : TLink);
begin
      A^.Succlink     := First;       {(1)}
      First^.Predlink := A;           {(2)}
      A^.Predlink     := nil;         {(3)}
      First           := A;           {(4)}
end;
```

Включение узла в список последним

Рассмотрим включение очередного узла в конец двунаправленного списка последним, но не единственным. Ссылка на последний узел находится в указателе Last. Исходное состояние:



Этапы включения очередного узла A в конец списка последним, но не единственным:



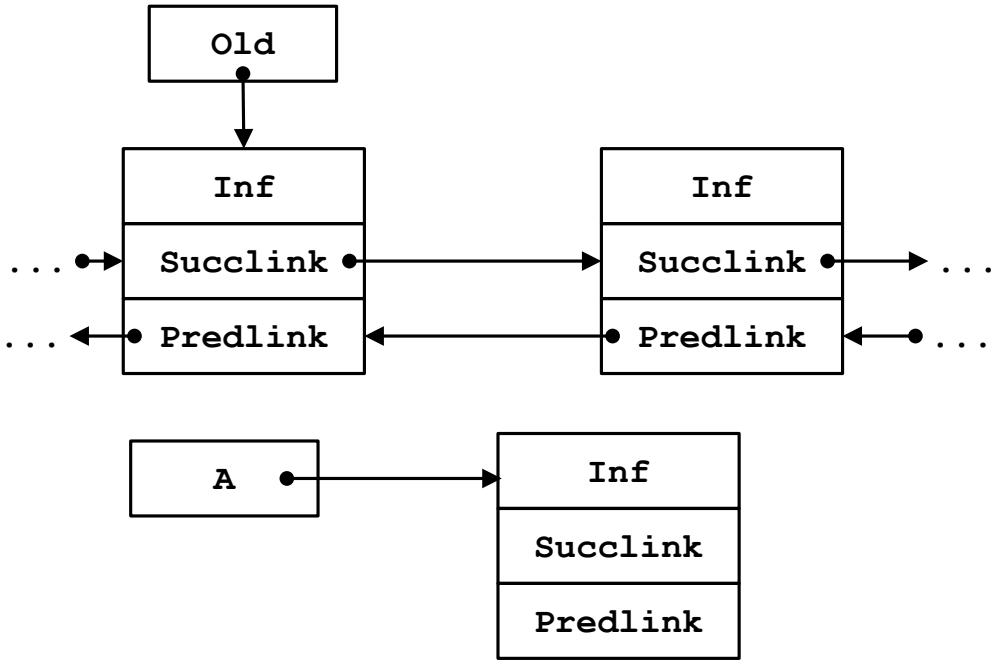
Соответственно получим следующую процедуру включения очередного узла A в конец двунаправленного списка последним, но не единственным.

```
procedure AddLast_2(var Last, A: TLink);
begin
      Last^.Succlink := A;                {(1)}
      A^.Predlink    := Last;             {(2)}
      A^.Succlink    := nil;              {(3)}
      Last           := A;                {(4)}
end;
```

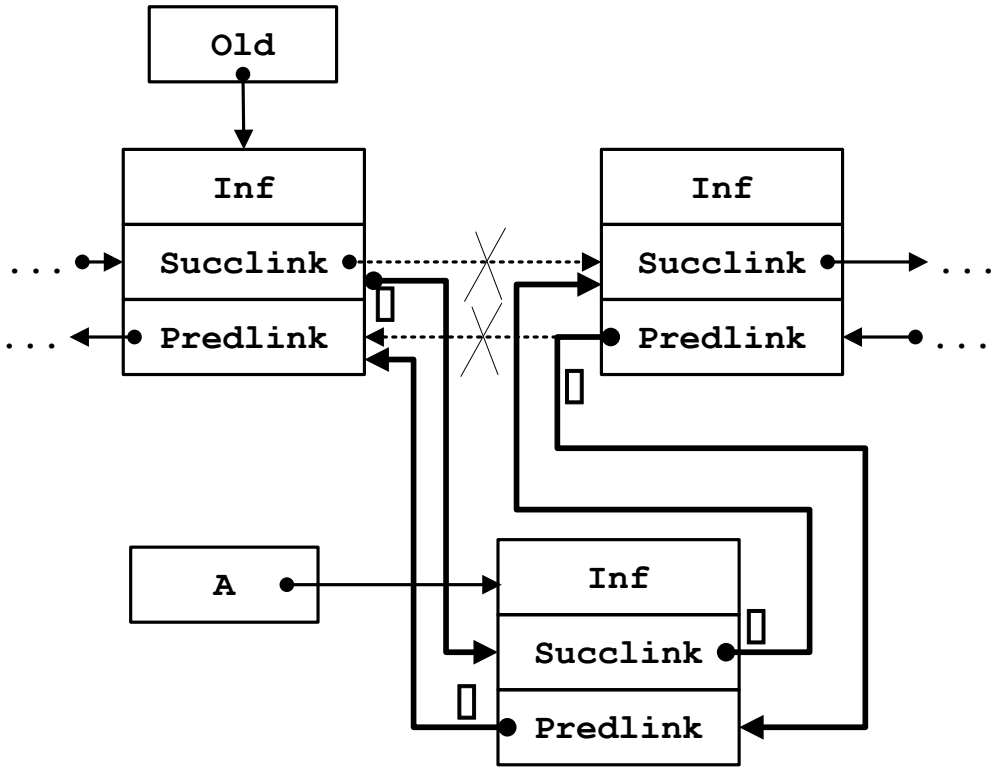
Включение узла A после узла Old

Пусть Old – указатель, который указывает на место вставки. Рассмотрим процедуру включения нового узла A после узла, на который указывает Old (процедура AddAfter_2).

Рассмотрим исходную ситуацию:



Этапы включения очередного узла A в двунаправленный список после узла, на который указывает Old:

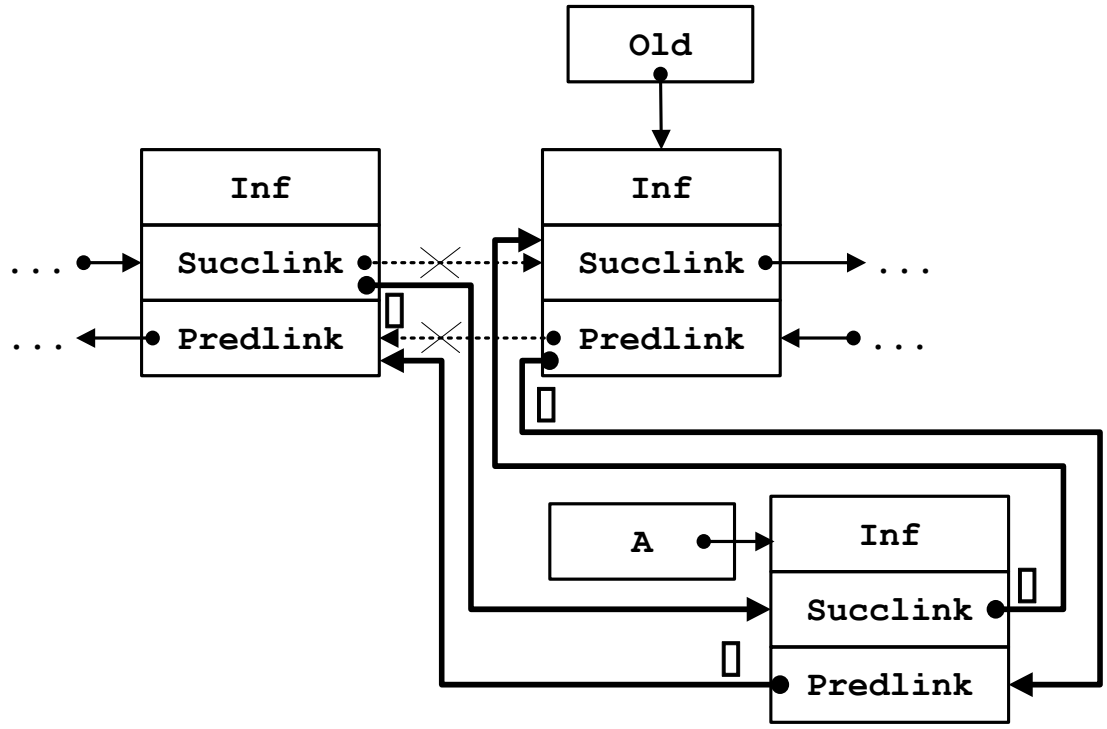


Для включения нового узла в двунаправленный список после узла, на который указывает Old, получим такую процедуру.

```
procedure AddAfter_2(var A, Old : TLink);
begin
   A^.Succlink           := Old^.Succlink; {(1)}
   A^.Predlink           := Old;           {(2)}
   Old^.Succlink         := A;             {(3)}
   A^.Succlink^.Predlink := A;             {(4)}
end;
```

Включение узла A перед узлом Old

Если нужно включить очередной узел перед узлом, на который указывает Old, то необходимо выполнить следующие действия.



Соответственно получим такую процедуру включения очередного узла A в середину списка перед узлом, на который указывает Old.

```
procedure AddBefore_2(var A, Old : TLink);
begin
     A^.Succlink           := Old;             {(1)}
     A^.Predlink           := Old^.Predlink;   {(2)}
     Old^.Predlink         := A;               {(3)}
     A^.Predlink^.Succlink := A;               {(4)}
end;
```

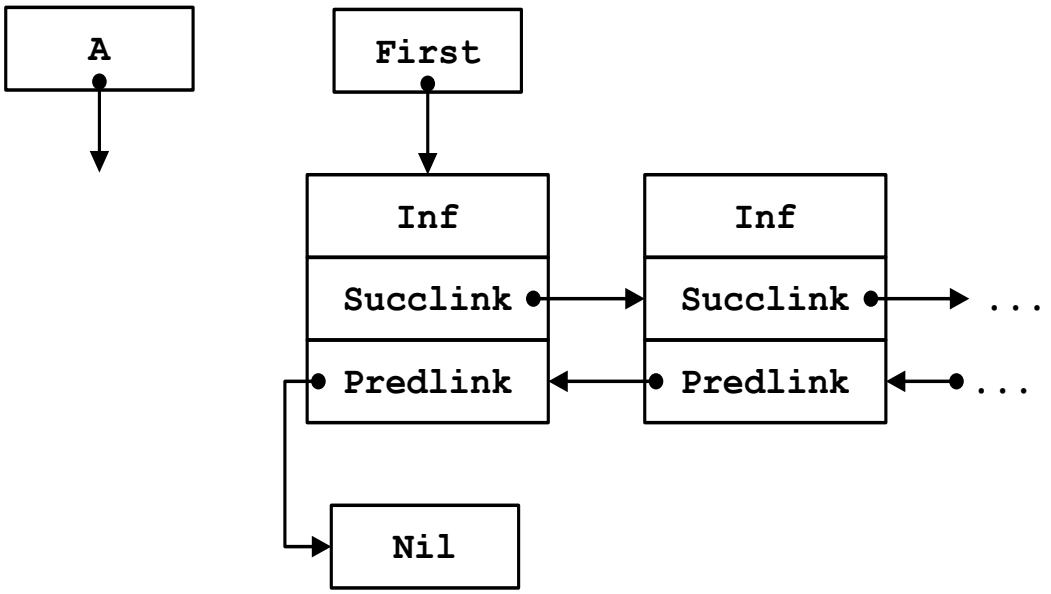
Процедуры `AddAfter_2` і `AddBefore_2` не «подправляют» указатели, если включаемый узел станет первым или последним, ведь мы такие процедуры сделали отдельно.

Процедуры удаления узла из двунаправленного списка

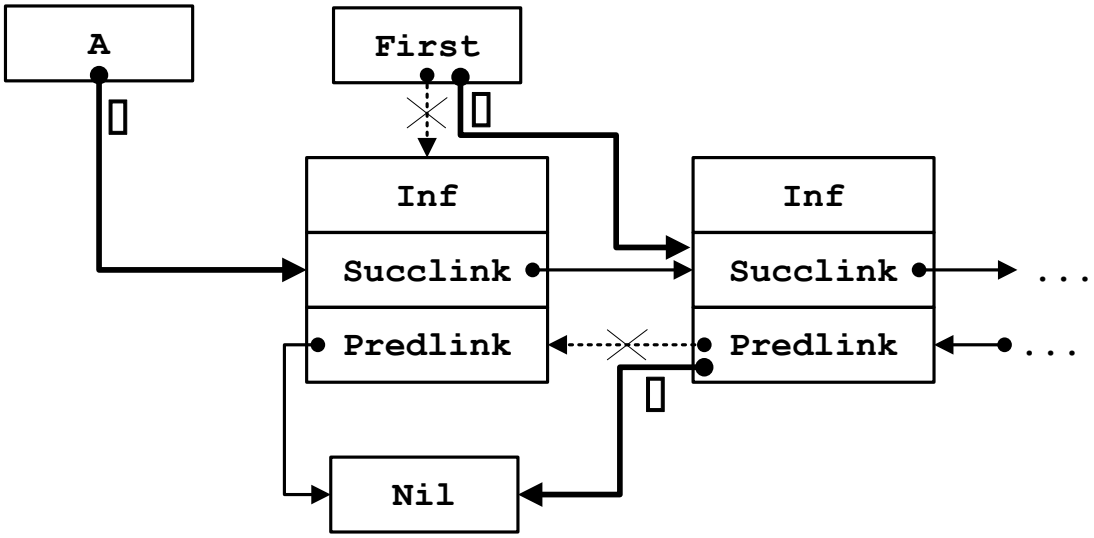
Рассмотрим ещё процедуры *удаления* узлов из двунаправленного списка.

Удалить первый узел

Исходное состояние удаления первого узла с двунаправленного списка:



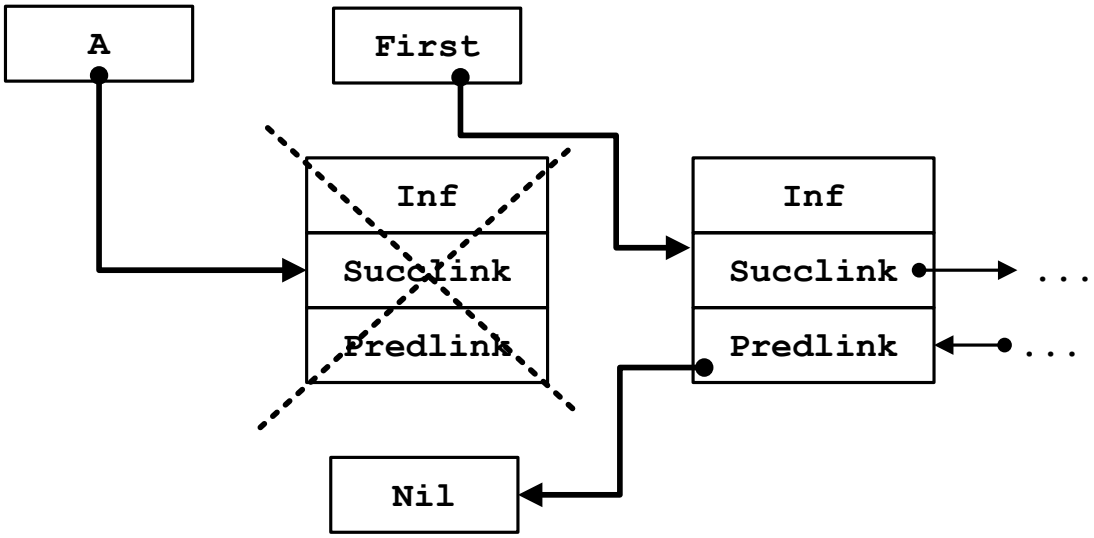
Этапы удаления первого узла списка:



Соответственно получим следующую процедуру удаления первого узла списка.

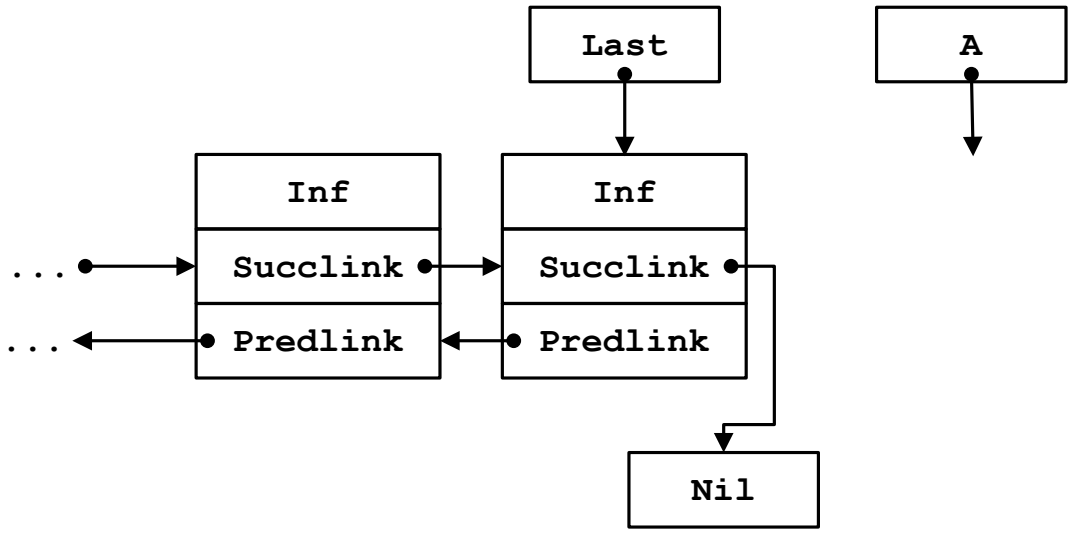
```
procedure DelFirst_2(var First, A : TLink);
begin
   A                          := First;
   First^.Succlink^.Predlink := nil;
   First                      := First^.Succlink;
end;
```

В указателе A получили ссылку на элемент, который вывели из списка. Потом можно освободить элемент с Heap (Dispose(A)).

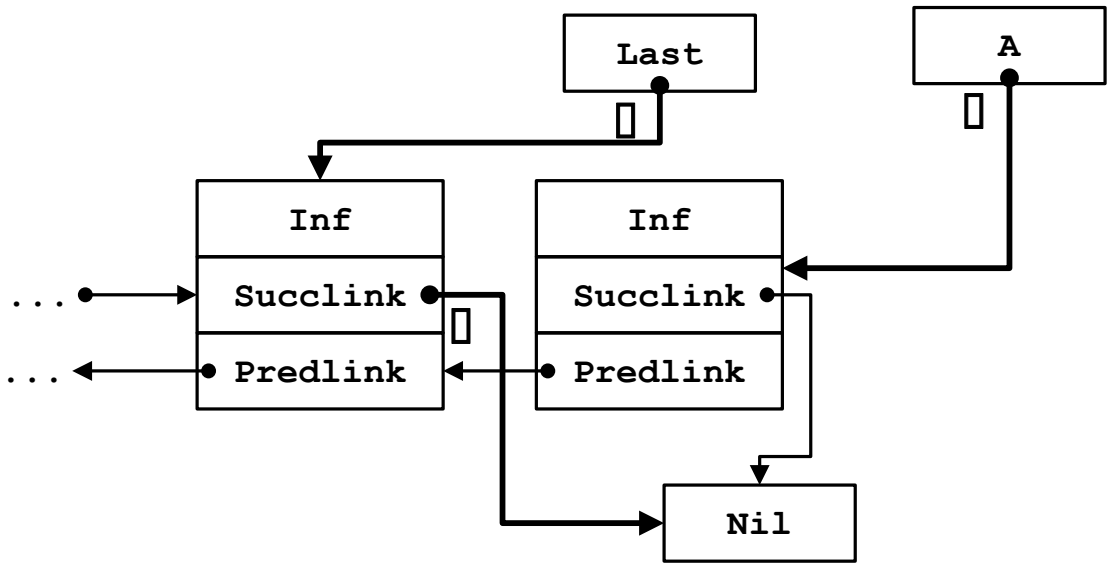


Удалить последний узел

Исходное состояние удаления последнего узла:



Этапы удаления последнего узла списка:



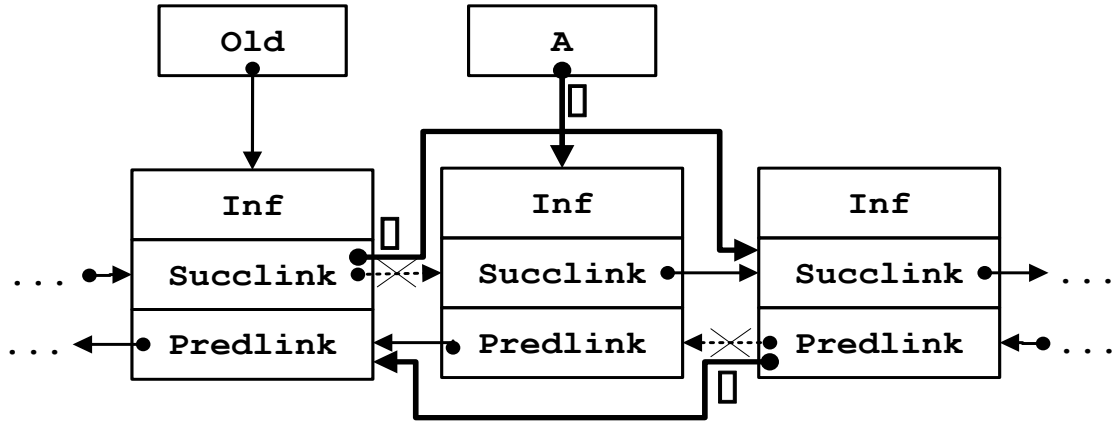
Соответственно получим следующую процедуру удаления последнего узла списка.

```
procedure DelLast_2(var First, A : Link);
begin
      A                          := Last;
      Last^.Predlink^.Succlink := nil;
      Last                       := Last^.Predlink;
end;
```

В указателе `A` получили ссылку на элемент, который вывели из списка. Память, отведенную под него, можно освободить оператором `Dispose(A)`.

Удалить узел после указателя на узел (середина списка)

Этапы удаления узла в середине двунаправленного списка после узла `Old`:

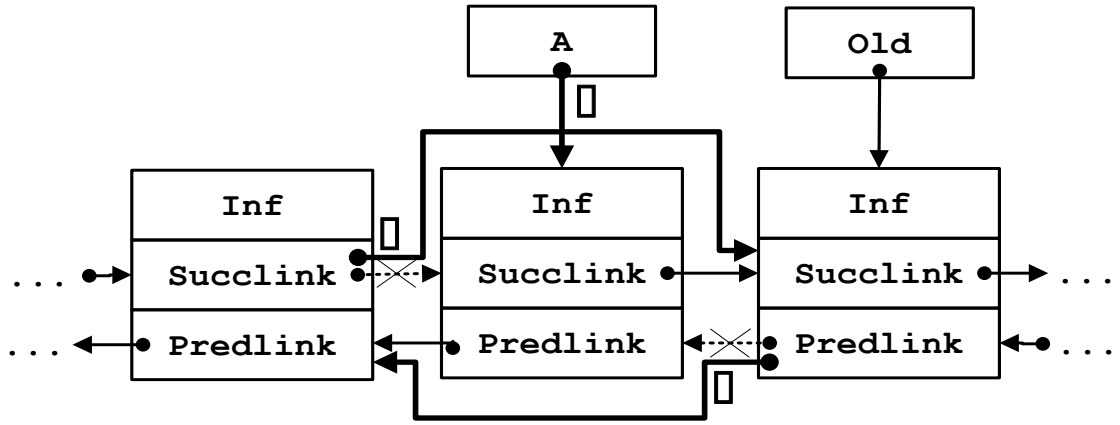


Предложенную схему удаления узла в середине списка после узла `Old` опишем в процедуры `DelAfter_2`.

```
procedure DelAfter_2(var A, Old : Link);
begin
      A                        := Old^.Succlink;
      Old^.Succlink            := A^.Succlink;
      Old^.Succlink^.Predlink := Old;
end;
```

Удалить узел перед указателем на элемент (середина списка)

Этапы удаления узла в середине двунаправленного списка перед узлом `Old`:

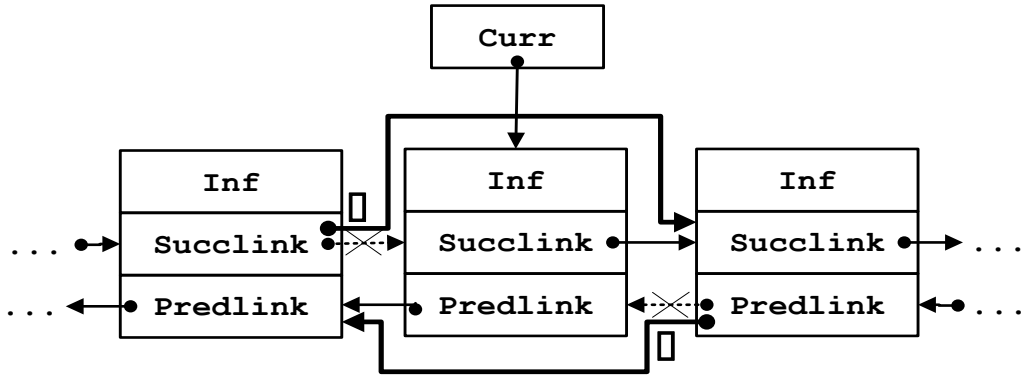


Получим следующую процедуру удаления узла в середине двунаправленного списка перед узлом `Old`.

```
procedure DelBefore_2(var A,Old:Link);
begin
   A                      := Old^.Predlink;
   A^.Predlink^.Succlink := Old;
   Old^.Predlink          := A^.Predlink;
end;
```

Удаление текущего узла (середина списка)

Этапы удаления текущего узла в середине двунаправленного списка:



Соответственно получим следующую процедуру удаления текущего узла в середине двунаправленного списка.

```
procedure  DelCurr_2(var Curr : Link);
begin
     Curr^.Predlink^.Succlink := Curr^.Succlink;
     Curr^.Succlink^.Predlink := Curr^.Predlink;
end;
```

После выполнения любой из пяти процедур удаления элемента из двунаправленного списка можно удалить элемент `A` или `Curr`.

Процедуры `DelAfter_2`, `DelBefore_2` и `DelCurr_2` предусматривают, что элемент удаляется строго с середины списка. Их нужно усовершенствовать на те случаи, когда в удалении будут задействованы «крайние» узлы – `First` и `Last`.

Удалить после указателя на элемент

Объединяя рассмотренные алгоритмы, получим следующую процедуру удаления узла после указателя на элемент из двунаправленного списка.

```
procedure DelAfter_2_Best(var A, Old, Last : Link);
begin
 A := Old^.Succlink;
 if A^.Succlink = nil then
   begin
    Last          := Old;
    Old^.Succlink := nil;
   end
 else
   begin
    Old^.Succlink           := A^.Succlink;
    Old^.Succlink^.Predlink := Old
   end;
end;
```

Удалить перед указателем

Удаление узла перед указателем на элемент из двунаправленного списка фактически есть или удаление первого узла, или удаление узла перед указателем на элемент в середине списка.

Объединяя алгоритмы удаления первого узла и перед указателем на элемент в середине списка, получим для двунаправленного списка следующую процедуру удаления узла перед указателем на элемент.

```
procedure DelBefore_2_Best(var  A, Old, First : Link);
begin
 A := Old^.predlink;
 if A^.predlink = nil then
   begin
    First         := Old;
    Old^.Predlink := nil;
   end
 else
   begin
    A^.Predlink^.Succlink := Old;
    Old^.Predlink         := A^.Predlink;
   end;
end;
```

Удаление текущего элемента

Удаление текущего узла из двунаправленного списка - это фактически реализация одного из следующих случаев: 1) удаление первого узла, но не единственного; 2) удаление последнего узла, но не единственного; 3) удаление текущего узла в середине списка; 4) удаление текущего узла в списке, который содержит только один узел.

```
procedure DelCurr_2_Best(var Curr, First,
                                   Last : Link);
begin
 if  Curr^.Succlink = nil  then
    begin
     Last                     := Last^.Predlink;
     Curr^.Predlink^.Succlink := nil
    end
 else
    if Curr^.Predlink = nil  then
       begin
        First                    := First^.Succlink;
        Curr^.Succlink^.Predlink := nil;
       end
    else
       begin
        Curr^.Predlink^.Succlink := Curr^.Succlink;
        Curr^.Succlink^.Predlink := Curr^.Predlink;
       end;
end;
```

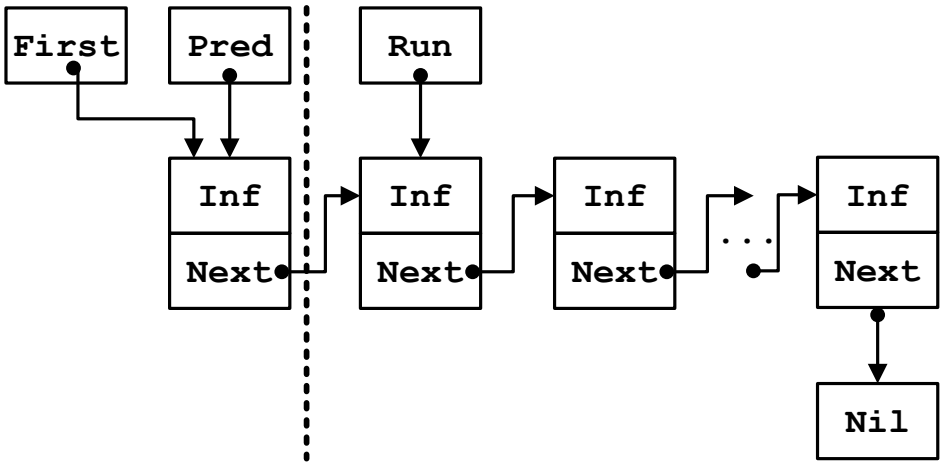
Прохождение связного списка не представляет никаких трудностей. Фактически нужно переходить от узла к узлу по указателю из ссылочной части до достижения указателя, равного `nil`, который свидетельствует об окончании списка.

Сортировка и слияние списков

Сортировка списков

Ввиду того, что списки представляют собой динамические структуры данных с последовательным, а не прямым доступом, то методы сортировок массивов для списков почти не применяют, за малым исключением. Рассмотрим их.

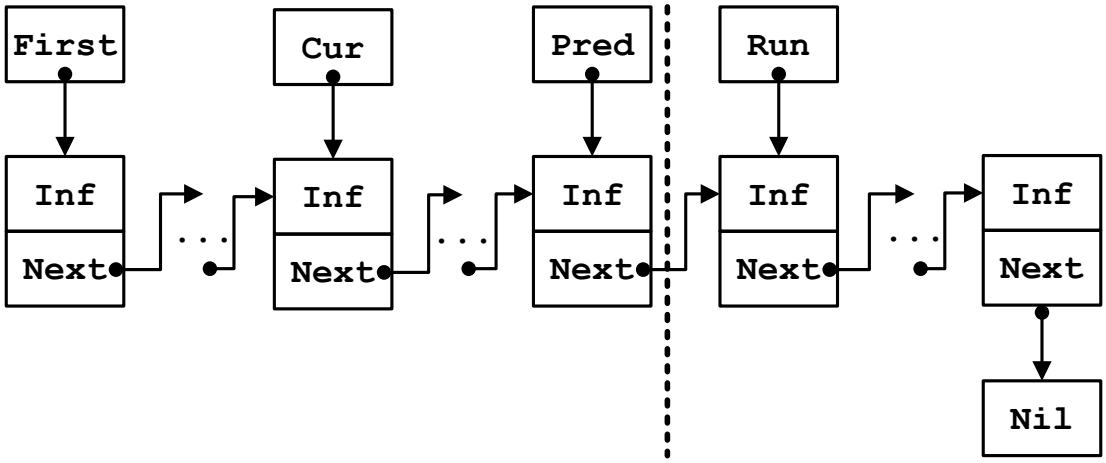
Первый алгоритм. Рассмотрим неупорядоченный список и применим для его сортировки метод «сортировка включениями»:



Для каждого узла списка (на него указывает `Run`), начиная со второго узла и до конца списка, ищем этому узлу место от начала списка (с `First`) до конца отсортированной части (на нее показывает `Pred`). Текущий узел включаем в нужное место или оставляем на старом. Здесь выделяются следующие этапы.

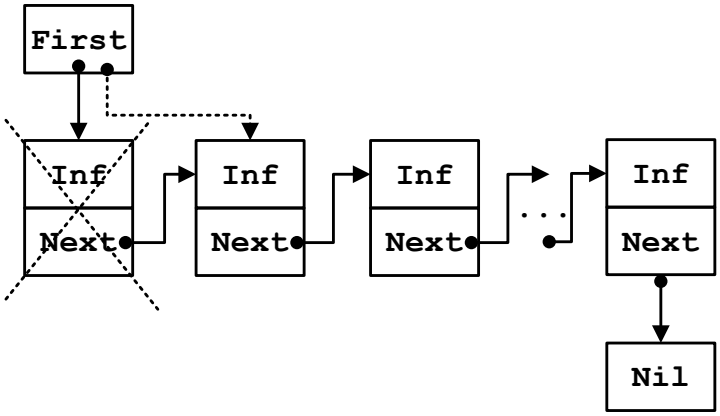
1. Поиск места, куда нужно вставить элемент.

2. Если вставляем после `Pred`, то оставляем узел на месте и сдвигаем `Pred` на `Run`; иначе вставляем в нужное место, а узел, на который указывает `Run`, удаляем из списка:

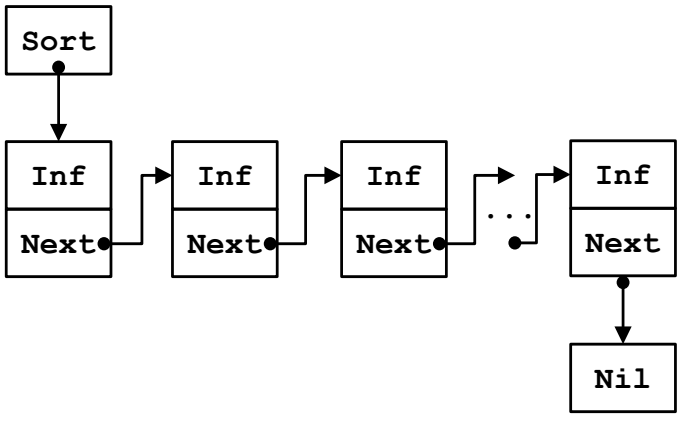


Поскольку здесь возникают сложности с указателями, лучше использовать другой подход.

Второй алгоритм. Проходя по первому списку, каждый раз вычленяем из него первый узел и вставляем на нужное место во второй отсортированный список:



Удаляем первый узел из одного списка и ищем ему место вставки в другой список:



Слияние упорядоченных списков

Первый алгоритм. Пусть заданы два упорядоченных списка. Один из них будем пополнять элементами из второго списка, не нарушая упорядочение.

Эта задача подобна предыдущей задаче сортировки (второй алгоритм), но тут поиск места вставки узла из первого списка во второй список надо начинать с места последней вставки, а не с самого начала второго списка.

Второй алгоритм. Является улучшением первого алгоритма. Он повторяет алгоритм слияния двух отсортированных файлов в третий отсортированный.

Стеки

Организация стека последовательным методом хранения

Стеки – это особый случай списка и простейшая динамическая структура данных. Добавление элементов в стек и выборка из него выполняются с одного конца, который называется *вершиной стека*. Другие операции со стеком не определены. Стеки широко используются в системном программировании, компиляторах, в различных рекурсивных алгоритмах.

Обычно стек обозначается английской аббревиатурой `LIFO` – `Last In First Out`, что можно перевести как «последний вошел, первый вышел». Это правило передает механизм работы со стеком.

Для стека нужна операция добавления элемента в стек – `AddFirst`, но для стека такую операцию называют `Push` – «затолкнуть». Вторая операция – `DelFirst` – «вытолкнуть» элемент из стека, для стека такую операцию называют `Pop`. Указатель на вершину стека называют `Top`.

Поскольку в переполненный стек нельзя затолкать очередной элемент и из пустого стека нельзя вытолкнуть элемент, то при работе со стеками нужны еще следующие дополнительные подпрограммы:

- **`function`** `Empty`, которая выдает значение `true`, если стек пуст, и значение `false` в противном случае;
- **`function`** `Full`, которая выдает значение `true`, если стек полностью заполнен, и `false` в противном случае;
- **`procedure`** `Mistake` – процедура, которая обрабатывает ошибочную ситуацию, параметр – код ошибки;
- **`procedure`** `Clear`, которая нужна, что бы «очистить» стек, параметр – указатель на вершину стека.

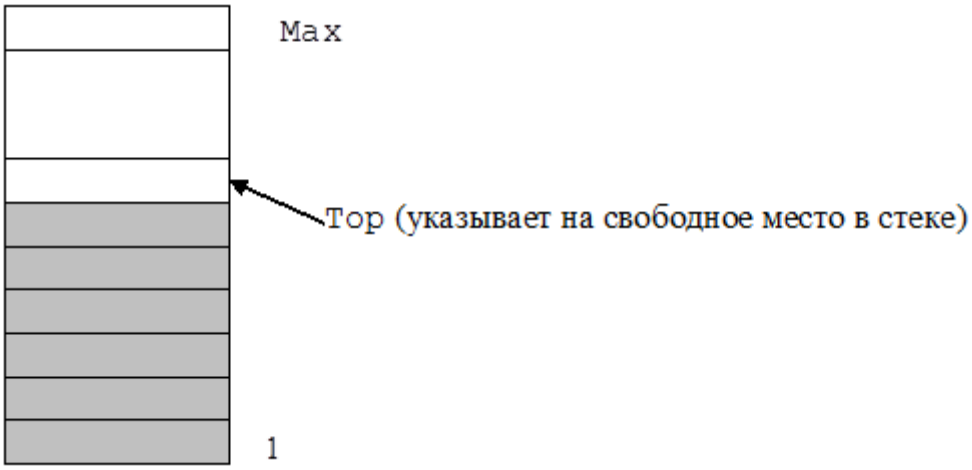
Поскольку для таких видов списка доступ односторонний, то их можно образовывать не только при помощи связных списков, используя разработанные выше процедуры, но и при помощи массивов.

Чем хорошо описывать работу со стеком отдельным модулем? Во-первых, это отдельная часть программы, которую можно модифицировать независимо от программы. Во-вторых, если все ресурсы по работе со стеками собрать в модуль, то реализацию стека можно «сохранить» в части `implementation`. Получим АТД «стек».

Работу со списками в динамической памяти мы уже усвоили, а теперь рассмотрим организацию стека в виде массива.

Количество элементов в массиве – глубину стека – объявим константой max. Индекс элемента массива, через который совершается доступ, – это указатель на вершину Top.

Если указатель Top равен единице, то стек «пустой» и в него можно затолкать очередной элемент, но нельзя из него вытолкнуть элемент. Если указатель Top = Max + 1, то стек «полный» и в него нельзя затолкать элемент, но можно вытолкнуть.



Опишем далее модуль, в котором соберем ресурсы по работе со стеком целых чисел (листинг 28).

```
unit StackInt;
interface
type
     TItem = Integer;
procedure Push    (x  : TItem);
function  Pop          : TItem;
procedure Mistake (Kod : Byte);
procedure Clear;
function  Empty        : Boolean;
function  Full         : Boolean;

implementation

const Max = 100;
var
      Stack : array[1..Max] of TItem;
      Top   : Word;

function  Empty;
begin
 Empty := (Top = 1);
end;

function  Full;
begin
 Full := (Top > Max);
```

```
end;

procedure  Clear;
begin
 Top := 1;
end;

procedure Push;
begin
 if Full then Mistake (1)
 else
   begin
    Stack[Top] := x;
    Inc(Top);
   end;
end;

function Pop;
begin
 if Empty then Mistake(2)
 else
    begin
     Dec(Top);
     Pop := Stack[Top]
    end;
end;

procedure Mistake;
begin
 case Kod of
   1:
  Write('Стек переполнен, увеличьте константу Max');
   2:
  Write('Стек пуст');
 else
  Write('Другие ситуации');
 end;
 Halt;
end;

begin
end.
```

Очереди

Очередь – это отдельный случай списка и простая динамическая структура. Добавление элементов в очередь выполняется в конец очереди, а выборка из очереди – с начала очереди. При выборке элемент исключается из очереди.

Другие операции с очередью не определены. Очереди широко используются в программировании.

Очередь реализует принцип FIFO – First In First Out. Это можно перевести как «первый вошел, первый вышел». Это правило напрямую передает механизм работы с очередью. Для очереди нужна операция Push – затолкать элемент в очередь, операция Pop – вытолкнуть элемент из очереди. Указатель на начало очереди называют Front, на конец – Rear.

Поскольку в переполненную очередь нельзя затолкать очередной элемент и с пустой очереди нельзя вытолкнуть элемент, то при работе с очередью нужны еще следующие подпрограммы:

- **function** Empty, которая выдаёт значение true, когда очередь пустая, и значение false в противном случае;
- **function** Full, которая выдаёт значение true, когда очередь переполнилась, и значение false в противном случае;
- **procedure** Mistake, которая обрабатывает ошибочную ситуацию, параметр – код ошибки;
- **procedure** Clear, которая нужна, чтобы «почистить» очередь, параметр – указатели Front и Rear.

Для такого вида списков доступ к элементу происходит на концах, поэтому их также можно реализовывать не только при помощи динамических структур данных, но и с помощью массивов.

Для моделирования очереди при помощи массива можно применить два подхода.

1. Массив с фиксированным количеством элементов, в котором элементы очереди занимают группу соседних компонент от Front до Rear, при этом, когда очередь достигает правого края массива, то все ее элементы сразу сдвигаются к левому краю (пересылка элементов будет занимать какое-то время).

2. Представление аналогично предыдущему, но массив как бы склеивается в кольцо, поэтому если элементы очереди достигают правого края массива, тогда новые элементы записываются на свободное место в начало массива. Второй подход более дешевый по времени работы программы.

Когда очередь пуста, индекс Front установим в 1 – первый незанятый элемент массива, за которым можно обслуживать, а индекс Rear установим в 0 – последний элемент очереди, за которым можно писать в очередь. Опишем переменную Size, в которой будем хранить текущую длину очереди.

Для работы с несколькими очередями напишем модуль, в нем будут описаны вышезаявленные ресурсы без использования динамических переменных (32). Получим следующую АТД «очередь».

32. Модуль для работы с очередьми

```
unit  TurnReal;
interface
const
       TurnMax  = 100;
       TurnMin  = 1;
```

```
       TurnSize = TurnMax - TurnMin + 1;
type
       TItem = Real;
       TMas  = array[TurnMin..TurnMax] of  TItem;
       Turn  = record
                 Front : Word;   {Начальный  индекс}
                 Rear  : Word;   {Конечный   индекс}
                 Size  : Word;   {Длина     очереди}
                 A     : TMas;   {          Очередь}
               end;
procedure  Push       (var Q : Turn;  x : TItem);
function   Pop        (var Q : Turn)    : TItem;
procedure  Mistake    (Kod   : Byte);
procedure  Clear      (var Q : Turn);
function   Empty      (var Q : Turn)    : Boolean;
function   Full       (var Q : Turn)    : Boolean;
procedure  CheckBounds(var  index       : Word);

implementation

function Empty;
begin
 Empty := Q.Size = 0;
end;

function Full;
begin
 Full := Q.Size = TurnSize;
end;

procedure Clear;
begin
 Q.Front := TurnMin;
 Q.Rear  := TurnMin - 1;
 Q.Size  := 0;
end;

procedure CheckBounds;
begin
 if index < TurnMin then index := TurnMax
 else
    if index > TurnMax then index := TurnMin;
end;

procedure Push;
begin
 if Full (Q) then Mistake(1)
 else
```

```
   begin
    Inc(Q.Rear);
    Inc(Q.Size);
    CheckBounds(Q.Rear);
    Q.A[Q.Rear] := x;
   end;
end;

function Pop;
begin
 if Empty(Q) then Mistake(2)
 else
   begin
    Pop := Q.A[Q.Front];
    Inc(Q.Front);
    CheckBounds(Q. Front);
    Dec(Q.Size);
   end;
 end;

procedure Mistake;
begin
   case Kod of
 1:
  Write('Очередь переполнена,увеличьте константу Max');
 2:
  Write('Очередь пустая');
 else
  Write('Другие ситуации');
   end;
 Halt;
end;

begin
end.
```

Очередь с двусторонним доступом

Двусторонняя очередь обладает свойствами, как стека, так и простой очереди. Данные в нее могут добавляться в любой конец и обслуживаться с любого конца. Обычно в программах, реализующих двустороннюю очередь, предусмотрены процедуры добавления данных в начало и в конец очереди, а также обслуживания, как с начала, так и с конца. Программную реализацию можно выполнять как на базе массива, так и при помощи двусвязных списков.

Очередь с приоритетом

Очередь с приоритетом – эта очередь, с каждым элементом которой ассоциирован некоторый приоритет. Приоритет задается номером. Обычно самый высокий приоритет имеет номер 1, более низкий – 2 и т.д. Однако бывает и наоборот.

Для очереди с приоритетом предусмотрены две операции: добавление нового элемента в очередь сообразно его приоритету и обслуживание элемента, который имеет самый высокий приоритет. Для такой очереди при реализации процедуры `Push` надо сначала найти место вставки элемента в очередь сообразно его приоритету. Тогда процедура `Pop` будет правильно обслуживать очередь с приоритетом.

Пример на использование очереди.

За какое минимальное количество ходов конем можно попасть из заданной начальной точки в конечную посещая шахматное поле один раз? По возможности указать последовательность действий.

Алгоритм.

Запомним начальный ход в очереди ходов и укажем, что он нулевого порядка.

Будем повторять пока не попадаем в конечную точку:

1. Извлечем из очереди координаты очередного хода.
2. Организуем на следующем проходе добавление в очередь всех возможных перемещений коня согласно таблице перемещения коня, если ход приемлем и, если клетка не занята.

Рассмотрим правила перемещения коня. Если задана начальная пара координат (x, y), то в наилучшем случае имеется восемь возможных координат (u, v) следующего хода. Отобразим их в таблице.

	1		8	
2				7
		x, y		
3				6
	4		5	

Из этой таблицы видно, что изменения координат в соответствии с номером хода будут следующими:

k	Δx	Δy
1	-1	2
2	-2	1
3	-2	-1
4	-1	-2
5	1	-2
6	2	-1
7	2	1
8	1	2

Деревья

Основные понятия и определения

Отношения между объектами могут носить нелинейный характер, например, определяться логическими условиями типа "один ко многим" или "многие ко многим".

Отношение "один ко многим" носит иерархический характер и отображается древовидными структурами (структура ВУЗа, УДК и др.).

Дерево – это набор узлов, между которыми установлены родительско-дочерние отношения.

На самом верхнем уровне такой иерархии всегда имеется только один узел, называемый *корнем дерева*.

Каждый узел, кроме корневого, связан только с одним узлом более высокого уровня, называемым *узлом-предком*. При этом каждый узел дерева может быть соединен с произвольным количеством узлов более низкого уровня, которые для данного узла являются *дочерними узлами* (*узлами потомками*). В некоторых источниках узлы-потомки называют *сыновьими узлами*.

Считается, что корень дерева размещен на уровне 0, остальные имеют следующую иерархию: если узел x имеет уровень i, то его непосредственный потомок – уровень $i+1$.

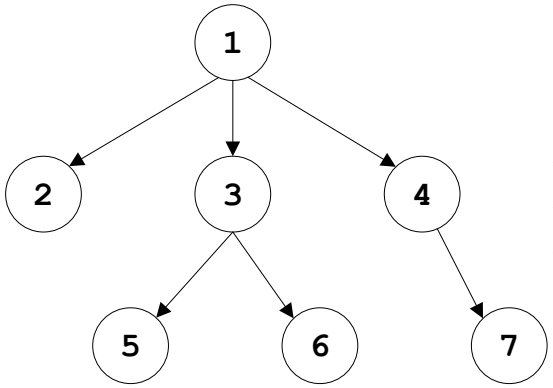
Любой узел дерева с его потомками также образует дерево, называемое *поддеревом* (относительно исходного дерева).

Основная терминология

Дерево – это конечное множество узлов с одним выделенным узлом B_0, который называется *корнем* дерева, а остальные узлы разбиты на $M > 0$ множеств $B_1, B_2, ..., B_M$, которые не пересекаются, каждое из них является *поддеревом*.

Эти соотношения между узлами дерева обладают следующими особенностями:

- существует узел (корень дерева), который не имеет предшественника;
- любой другой узел имеет одного предшественника.



$B_1 = \{2\};$
$B_2 = \{3, 5, 6\};$
$B_3 = \{4, 7\}.$

Узлы дерева, которые не имеют потомков, являются *листьями* дерева (нетерминальные элементы), остальные – внутренние (терминальные элементы).

От корня до любого узла всегда существует только один путь. Максимальная длина пути от корня до листьев называется *высотой дерева*.

Глубина узла – длина пути от корня до этого узла.

Степень узла n – это число его потомков. Наибольшая из степеней всех узлов считается степенью дерева.

Дерево степени *n* называется *полным*, когда степень любого его узла равна *n* или 0.

Дерево, каждый узел которого представлен одним и тем же типом записи, наз. *однородным*.

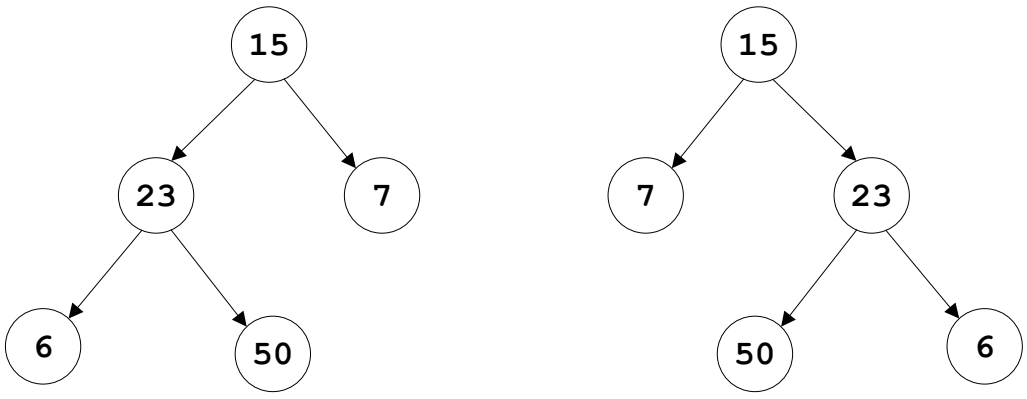
Если узлы дерева представлены разными типами записей, то оно называется *неоднородным*.

Типы деревьев

В зависимости от количества потомков (ветвистость) деревья разделяют на *бинарные* (*binary trees*) – не больше двух поддеревьев у произвольного элемента и *сильноветвистые* (*multiway trees*) – есть элементы, которые имеют больше двух поддеревьев.

Дерево *упорядочено*, когда для каждого потомка *k'* узла *k* степени *n* указано, каким он является: первым, вторым, ..., *n*-м (левым, средним, ..., правым).

Ниже показано одно и то же полное неупорядоченное дерево степени 2, или если считать их упорядоченными, то это разные деревья.



Бинарные деревья

Прямой и симметричный обходы деревьев

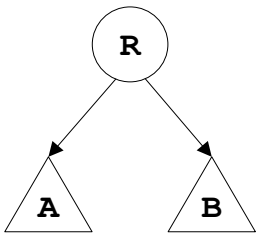
Рассмотрим упорядоченное дерево. Дочерние узлы обычно упорядочиваются слева направо – левостороннее упорядочивание, если не оговорено о правостороннем упорядочивании.

При прохождении узлов дерева *в прямом порядке* сначала посещается корень, затем узлы самого левого поддерева, далее узлы следующих поддеревьев.

При *обратном обходе* узлов дерева сначала посещаются в обратном порядке все узлы поддеревьев, затем корень.

Обход упорядоченного бинарного дерева

Рассмотрим упорядоченное бинарное дерево:



Обход (прохождение) упорядоченного бинарного дерева проводят шестью методами:

а) при левостороннем упорядочивании:

- сверху вниз: R, A, B (префиксная форма обхода – PreOrder);
- слева направо: A, R, B (инфиксная форма обхода – InOrder);
- снизу вверх: A, B, R (постфиксная форма обхода – Postorder);

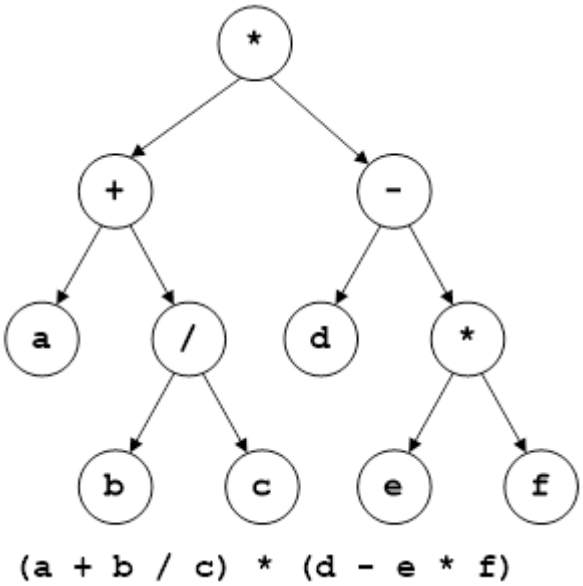
б) при правостороннем упорядочивании:

- сверху вниз: R, B, A;
- слева направо: B, R, A;
- снизу вверх: B, A, R.

Более очевидными становятся три первых варианта обхода на обходе деревьев-формул.

Дерево-формула получается в соответствии с арифметическим выражением, содержащим переменные величины, арифметические операции и скобки, согласно такому алгоритму, имеем: в корень дерева помещается операция, операнды же направляются в левое и правое поддеревья.

Очевидно, что арифметическому выражению $(a+b/c)*(d-e*f)$ соответствует рассмотренное дальше дерево.



(a + b / c) * (d - e * f)

При выполнении префиксного обхода R, A, B, получаем следующую последовательность: *, +, a, /, b, c, –, d, *, e, f.

Если же выполнить инфиксный обход A, R, B, получим такую последовательность: a, +, b, /, c, *, d, –, e, *, f.

Если тут своевременно проставить круглые левые и правые скобки, получим полную скобочную запись выражения, в котором потом в соответствии с приоритетом операций некоторые скобки можно опустить.

Постфиксный обход A, B, R дает последовательность: a, b, c, /, +, d, e, f, *, –, *.

Представление деревьев

Представление деревьев можно делать и при помощи массивов, и при помощи ссылок.

Приведенное выше дерево с помощью массива представляется так: описывается переменная-массив

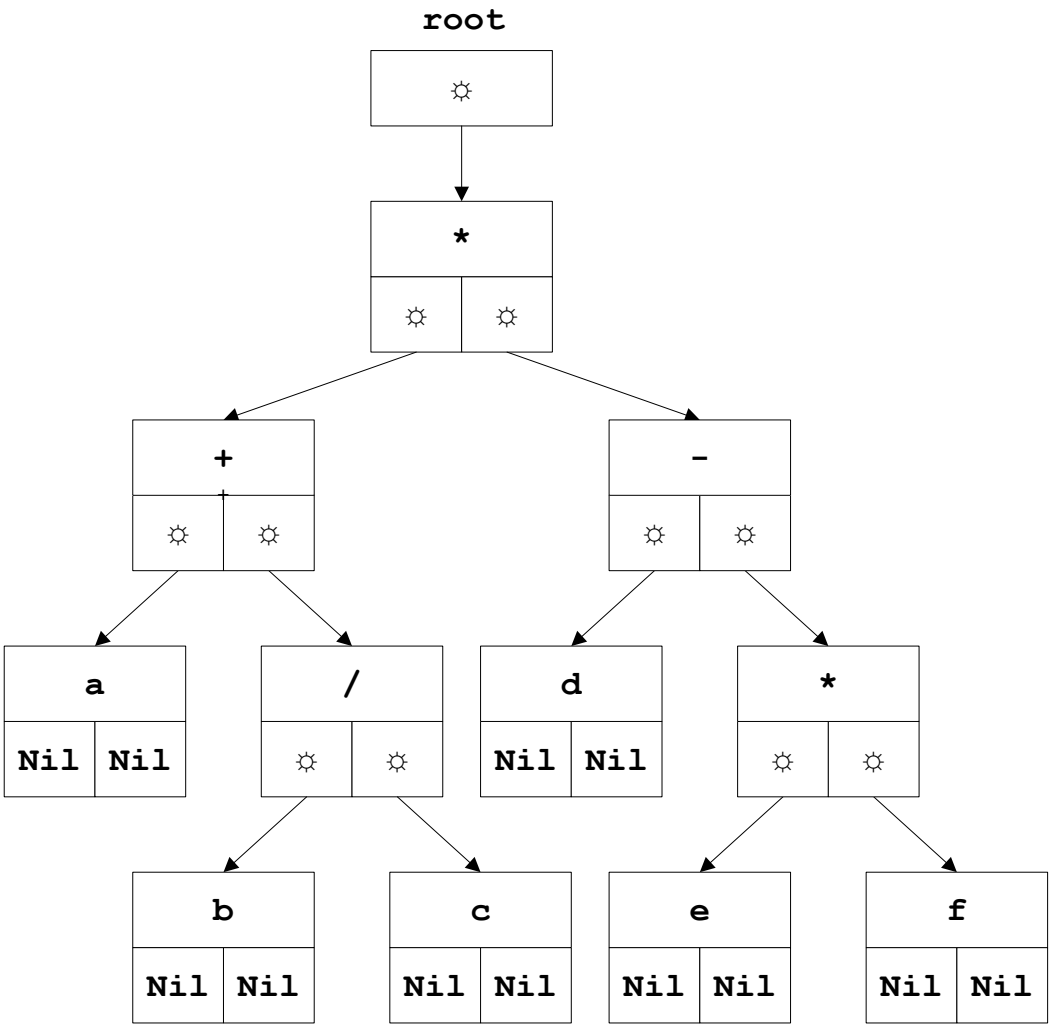
```
t : array [1..11] of
                    record
                     op    : char;
                     left,
                     rigth : Integer
                     end;
```

значения компонент которой следующие (таблица 11):

Таблица 11 – Пример представления дерева массивом.

Индекс элемента массива	Значение поля записи op	Значение поля записи left	Значение поля записи rigth
1	*	2	3
2	+	6	4
3	–	9	5
4	/	7	8
5	*	10	11
6	a	0	0
7	b	0	0
8	c	0	0
9	d	0	0
10	e	0	0
11	f	0	0

Представляя данное дерево с помощью ссылок, будем иметь следующее дерево:



Введем следующие определения:

```
Type
        TRef  = ^TNode;
        TNode = record
                  op          : char;
                  left, rigth : TRef;
                end;
var     root : TRef;
```

Над деревьями обычно выполняются следующие операции:

- добавить в дерево узел;
- исключить из дерева определенный узел;
- пройти все узлы дерева в заданном порядке;
- найти узел с заданным свойством;
- определить родительский узел заданного узла;
- определить дочерние узлы заданного узла и др.

Надо заметить, что само дерево определяется в терминах рекурсивных соотношений, значит, и действия над деревьями лучше всего задавать с помощью рекурсии.

Работа с бинарными деревьями

Рассмотрим сначала процедуры левостороннего обхода упорядоченного дерева.

Разные способы левостороннего обхода упорядоченного дерева

```
procedure PreOrder(t : TRef);
begin
 if t <> nil then
   begin
 { Обработка узла, например,  Write(t^.op : 4);}
    PreOrder(t^.left);
    PreOrder(t^.right);
   end;
end;

procedure InOrder(t : TRef);
begin
 if t <> nil then
   begin
    InOrder(t^.left);
 { Обработка узла, например,  Write(t^.op : 4);}
    InOrder(t^.right);
   end;
end;

procedure Postorder(t : TRef);
begin
 if t <> nil then
   begin
    Postorder(t^.left);
    Postorder(t^.right);
 { Обработка узла, например,  Write(t^.op : 4);}
   end;
 end;
```

хНапишем еще процедуру распечатки схемы дерева. Во-первых, представим дерево, повернув его на 90° против часовой стрелки. Поскольку каждый новый уровень будет отступать от предыдущего на несколько позиций, то в процедуру построения добавим параметр – номер уровня. Получим рекурсивный алгоритм:

- пустое дерево не печатается для поддерева уровня h;
- печатаем правое поддерево;
- печатаем узел, который выделяется предыдущими пробелами, соответствующими по количеству уровню *h*;
- печатаем левое поддерево.

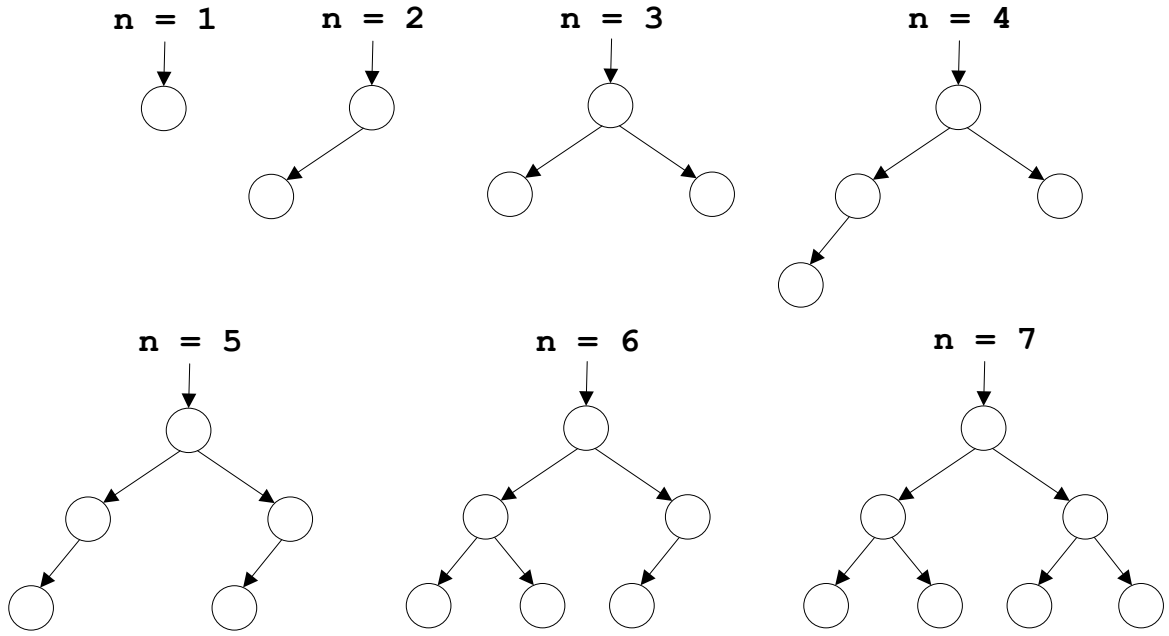
Процедура распечатки схемы дерева

```
procedure PrintTree(t : TRef; h : Byte);
var   i : Byte;
begin
  if  t = nil then Exit;
  PrintTree(t^.right, h + 1);
  for i := l to h do Write('    ');
  Writeln(t^.op);
  PrintTree(t^.left, h + 1);
end;
```

Задание. Напишите процедуру распечатки схемы дерева с использованием графического режима.

Идеально сбалансированные деревья

Мы уже рассматривали деревья-формулы. Существуют еще *идеально сбалансированные деревья*, которые имеют следующий вид:



Такие деревья дают минимальную глубину. Чтобы достичь наименьшей глубины при данном числе узлов, нужно на всех уровнях, кроме самого нижнего, распределять максимально возможное количество узлов.

Дерево *идеально сбалансированное*, если для каждого его уровня число узлов в левом и правом поддереве отличается не более чем на один.

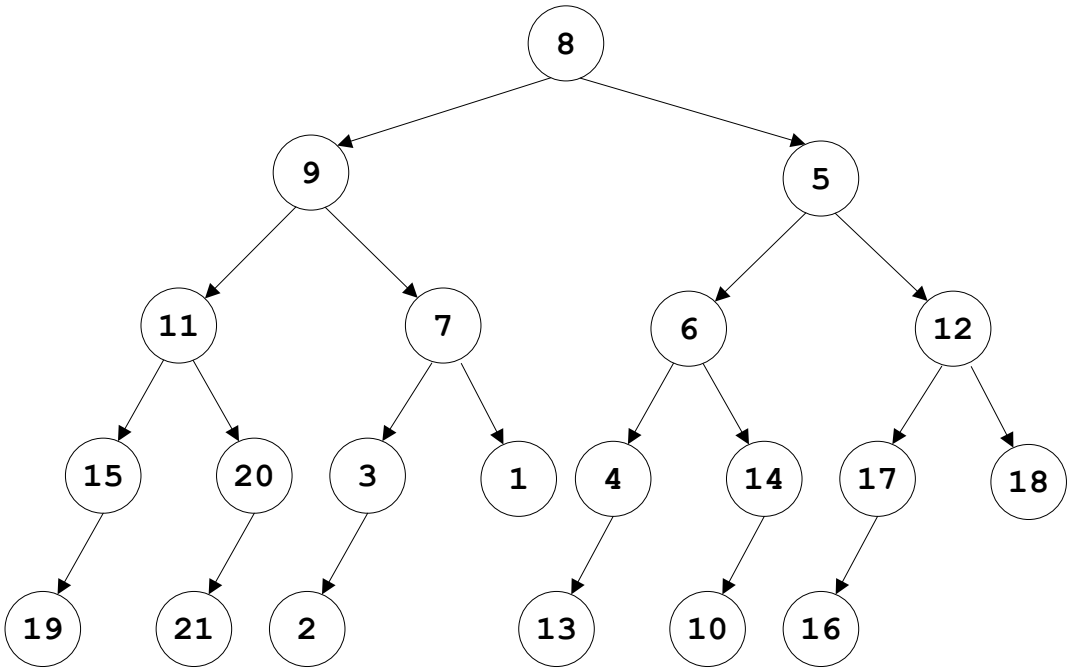
Это правило легко сформулировать при помощи рекурсии:

- взять один узел в качестве корня;
- построить левое поддерево с $n_л = n$ div 2;
- построить правое поддерево с $n_п = n - n_л - 1$.

Рассмотрим построение такого дерева на примере следующей задачи.

Задача. Построить идеально сбалансированное дерево из заданных чисел, находящихся в типизированном файле.

Построим сначала самостоятельно дерево из следующей последовательности чисел: 8, 9, 11, 15, 19, 20, 21, 7, 3, 2, 1, 5, 6, 4, 13, 14, 10, 12, 17, 16, 18 (n = 21):



Далее опишем рекурсивную функцию

`function Tree (n:Integer):TRef`,

которая строит идеально сбалансированное дерево.

Результатом работы функции будет указатель на корень дерева.

Если подключим рассмотренные ранее подпрограммы, получим следующую программу.

Построение идеально сбалансированного дерева

```
Program Build_Tree;
uses crt;
type
    TInf  = Integer;
    TRef  = ^TNode;
    TNode = record
              op         : TInf;
              left,right : TRef;
            end;
var  f    : file  of  TInf;
     root : TRef;
     n    : Integer;

function Tree(n : Integer) : TRef;
                 {Ссылка на корень}
var
      Newnode : TRef;
      nl,nr   : Integer;
      x       : TInf;
```

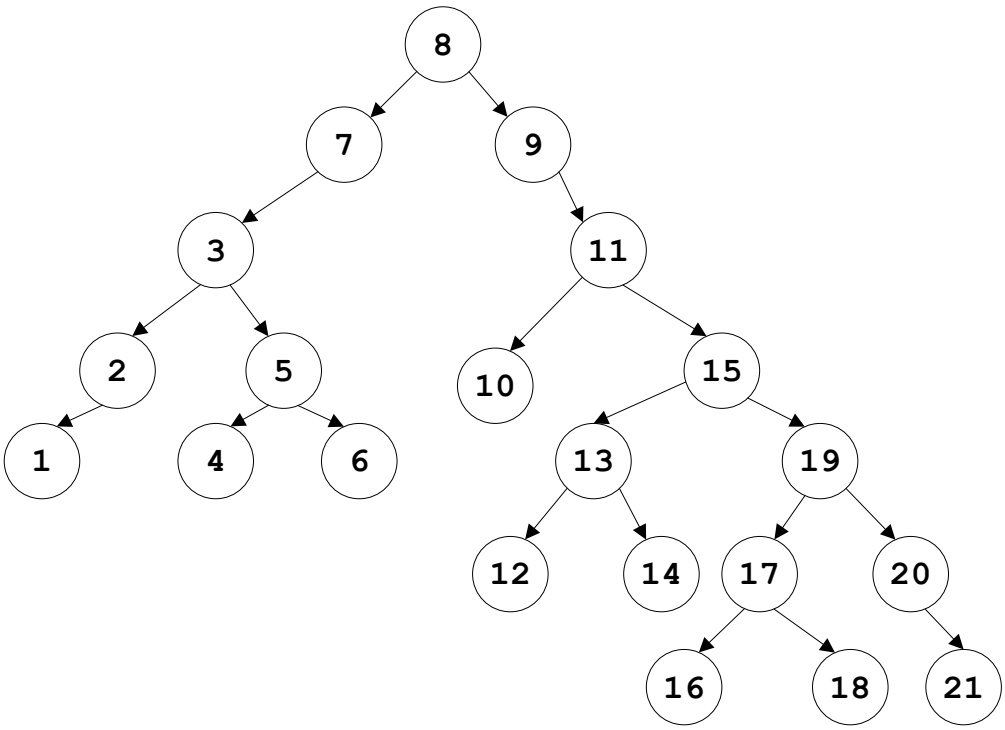
```
begin
 if n = 0 then  tree := nil
 else
   begin
    nl := n div 2;
    nr := n - nl - 1;
    New(Newnode);
    Read(f, x);
    Newnode^.op    := x;
    Newnode^.left  := tree(nl);
    Newnode^.right := tree(nr);
    tree           := Newnode;
   end;
end;

procedure PreOrder(t:TRef);
begin
 if t = nil then Exit;
 Write(t^.op : 4);
 PreOrder(t^.left);
 PreOrder(t^.right);
end;

procedure PrintTree(t:TRef; h : Byte);
var    i : Integer;
begin
 if t = nil then Exit;
 PrintTree(t^.right, h + 1);
 for i := 1 to h do Write('   ');
 Writeln(t^.op:10);
 PrintTree(t^.left, h + 1);
end;

begin
 ClrScr;
 Assign(f, 'c:\Pascal\DataWord.dat');
 Reset(f);
 n := FileSize(f);
 Root := Tree(n) ;
 PrintTree(root, 0);
 PreOrder(root);
 { другая работа }
end.
```

Дерево поиска

Двоичное дерево поиска (англ. *binary search tree*, *BST*) – это двоичное дерево, для которого выполняются следующие дополнительные условия: значения всех элементов, расположенных в левом поддереве, меньше значения в

корне дерева, а значения всех элементов, расположенных в правом поддереве, больше либо равны значения в корне дерева.

Для последовательности 8, 9, 11, 15, 19, 20, 21, 7, 3, 2, 1, 5, 6, 4, 13, 14, 10, 12, 17, 16, 18 получим следующее дерево:



Программу построения дерева поиска напишем далее.

Следующая задача получения частотного словаря строит дерево поиска, которое называется *лексикографическим деревом*.

Лексикографическое дерево поиска

Задача. Имеется массив слов (чисел). Подсчитать, сколько раз встречается каждое слово (число) в массиве. Распечатать слова (числа) в алфавитном порядке (порядке возрастания).

Решение выполним для целых чисел, хранящихся в файле. Начинаем с пустого дерева. Затем каждое число ищется (просеивается) в дереве по принципу дерева поиска (меньше, чем в вершине, – идем на левое поддерево, в противном случае – на правое). Если число найдено, увеличиваем счетчик его появлений, иначе, если числа нет в дереве, оно вставляется в дерево, причем счетчик его становится равным 1.

Распечатка чисел в порядке возрастания

```
Program Build_Tree_Praseivanne;
type
       TInf  = Integer;
       TRef  = ^TNode;
       TNode = record
                 key        : TInf;
                 count      : Integer;
```

```
               left,right : TRef;
              end;
var      f    : file  of  Integer;
         root : TRef;
         k    : TInf;
         P    : Pointer;
procedure Praseivanne(x : TInf; var p : TRef);
begin
 if p = nil  then
  begin
   New(p);
   P^.key   := x;
   P^.count := 1;
   P^.left  := nil;
   P^.right := nil;
  end
 else
    if P^.key>x then Praseivanne(x, P^.left)
    else
        if P^.key<x then Praseivanne(x,P^.right)
        else Inc(p^.count);
end;

procedure InOrder(t:TRef);
begin
 if t <> nil   then
   begin
    InOrder(t^.left);
    Write  (t^.key:4);
    InOrder(t^.right);
   end;
end;

procedure PrintTree(t : TRef; h : Byte);
var    i : Byte;
begin
if t = nil then Exit;
   PrintTree(t^.right, h + 1);
   for i := 1 to h do   Write('   ');
   Writeln(t^.key);
   PrintTree (t^.left, h + 1);
end;

begin
 Mark  (P);
 Assign(f, 'Z:\Home\ffff.dat');
 Reset (f);
 Root := nil;
```

```
while not Eof(f) do
    begin
      Read(f, k);
      Praseivanne(k, root);
    end;
PrintTree(root, 0);
InOrder  (root);
{іншая работа}
Readln;
Release(P);
end.
```

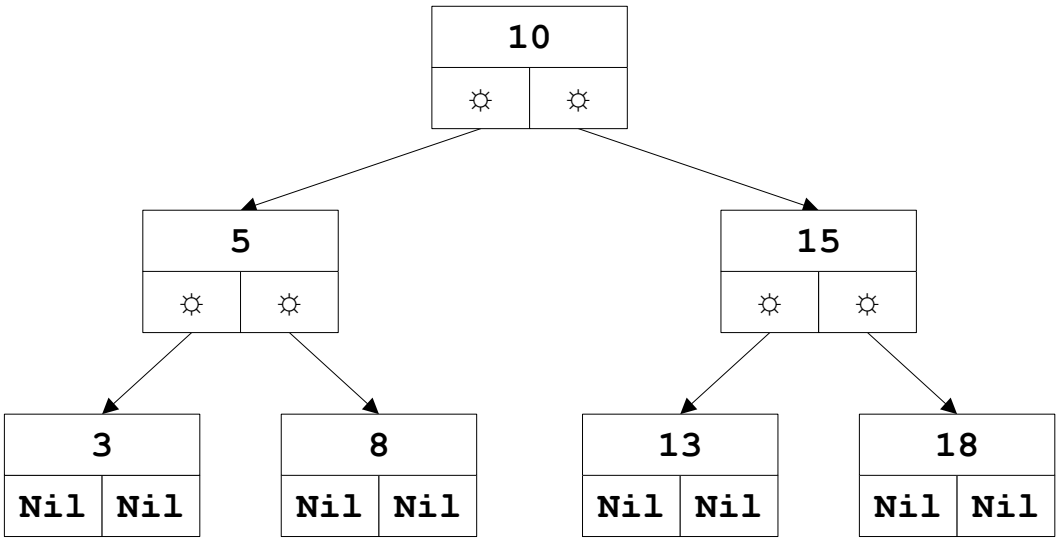
Симметричный обход построенного дерева слева направо даст отсортированный по возрастанию массив чисел, справа налево – по убыванию.

Удаление из бинарного дерева поиска

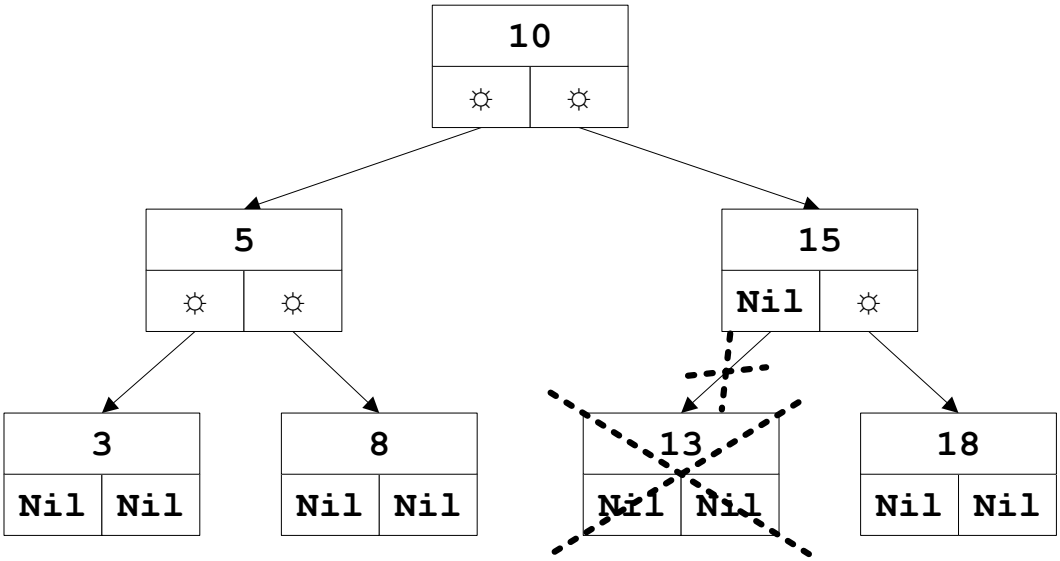
Когда встает задача удаления узла из дерева, то могут возникнуть различные обстоятельства.

Рассмотрим следующее дерево поиска и обсудим ситуации удаления узла так, чтобы после операции осталось также дерево поиска.

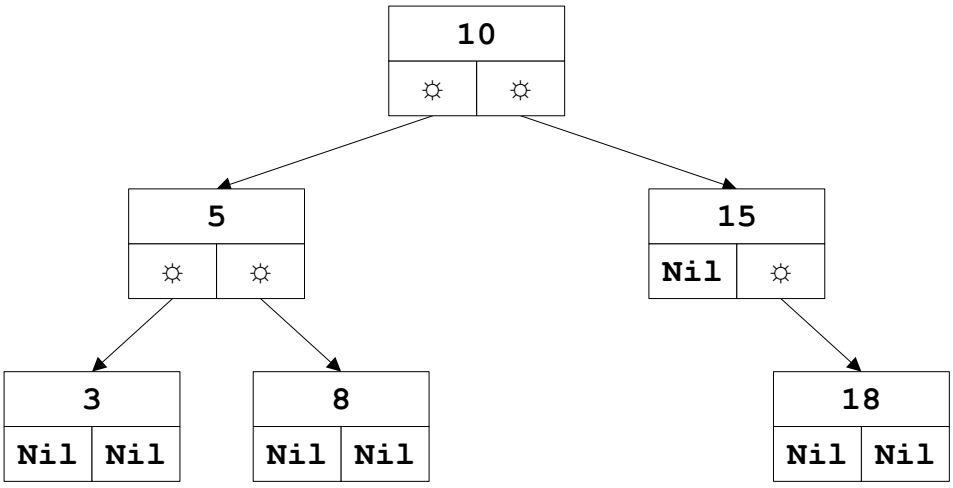
1. Пусть нужно удалить лист следующего дерева (например, элемент «13»):



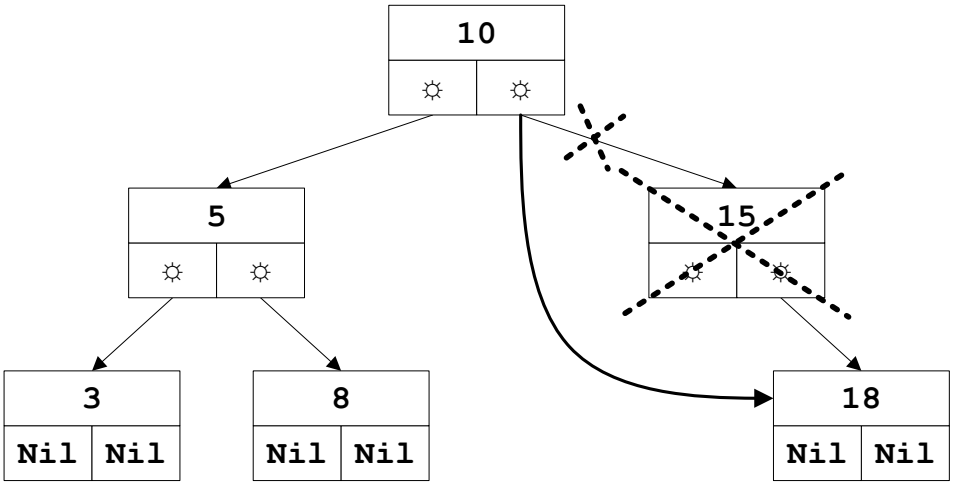
Тогда родительский узел «15» вместо ссылки на дочерний узел должен записать **nil**:



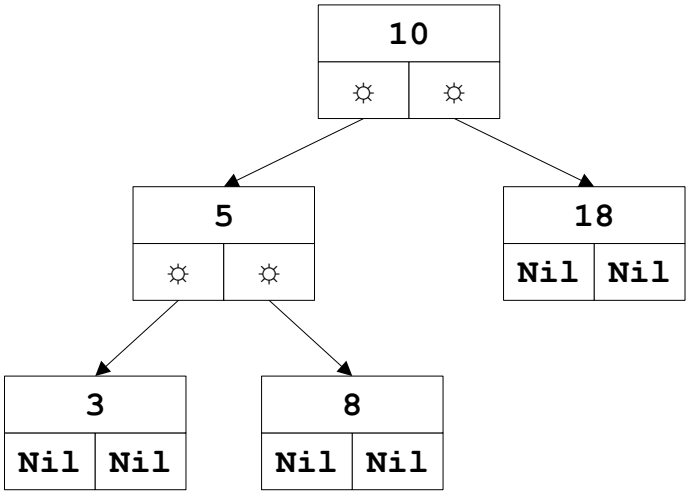
2. Пусть в полученном дереве нужно удалить узел, который имеет одного потомка (например, элемент «15»):



Тогда надо на место родительского узла переместить дочерний узел:

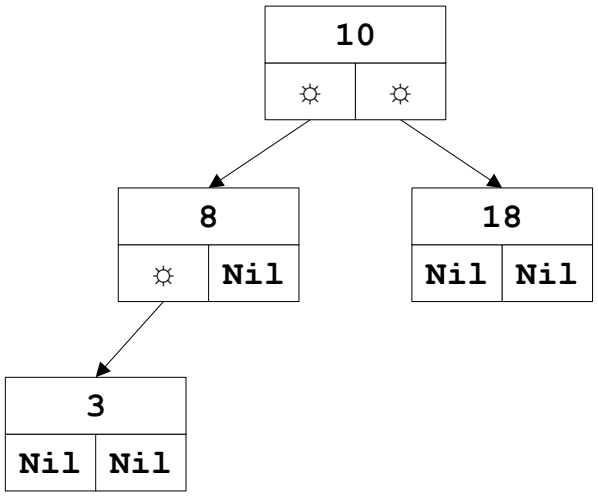


3. Пусть в последнем дереве нужно удалить узел, имеющий двоих потомков (например, элемент «5»):

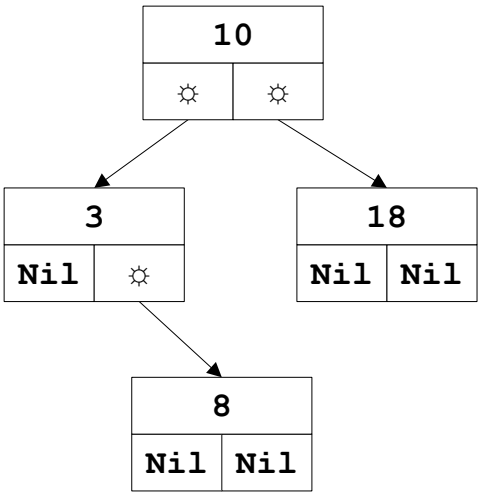


Это можно сделать двумя способами, так как на место родительского узла можно подставить левый (вариант 1) или правый (вариант 2) дочерний узел.

Получим следующие варианты схем удаления:



Вариант 1

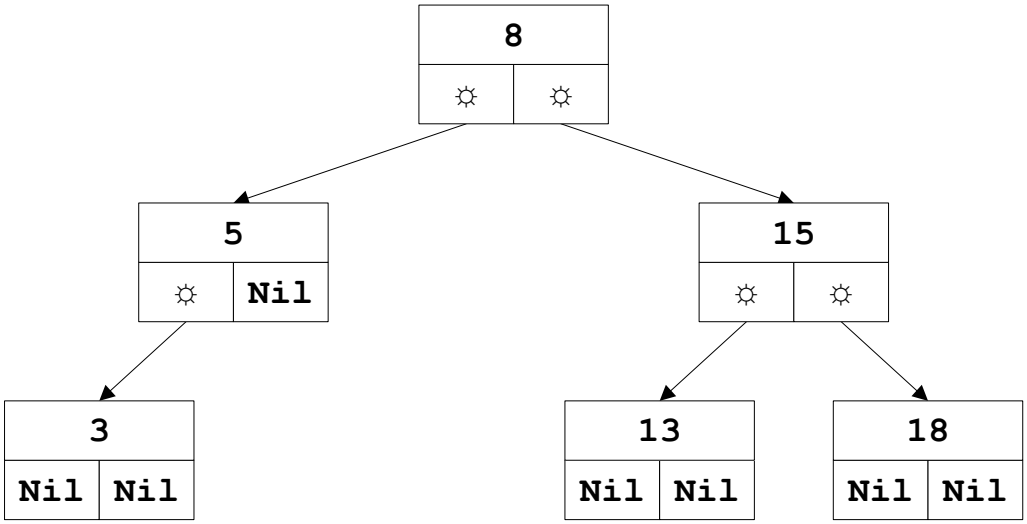


Вариант 2

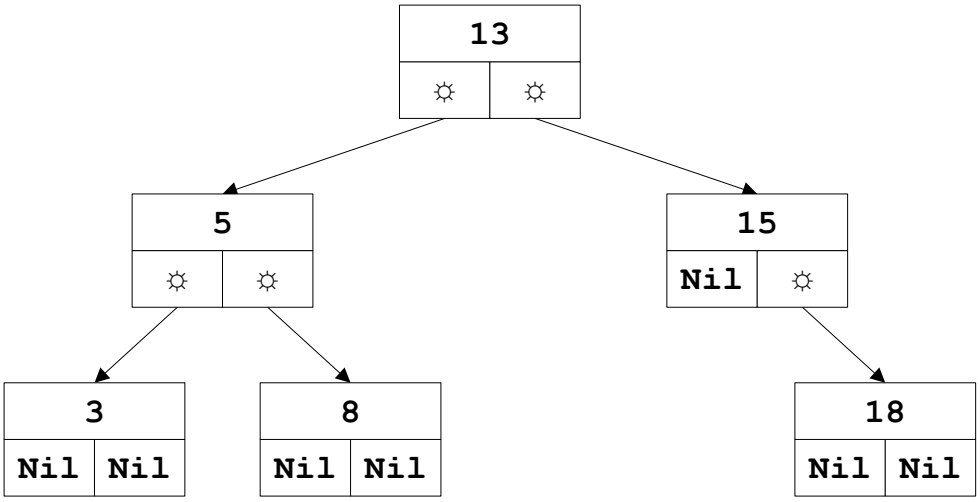
Можно заметить, что ситуация 3 является частным случаем следующей ситуации 4.

4. Пусть нужно удалить узел, имеющий много потомков (например, элемент «10» первоначального дерева).

Это можно сделать двумя способами: заменить (листом) узлом «8» (шаг влево и потом до конца вправо):



или узлом «13» (шаг вправо и потом до конца влево):



Задача. Построить алгоритм для удаления узла с ключом, равным *x*, из дерева поиска.

Р е ш е н и е . Напишем процедуру, которая различает три случая:

1) узел с ключом *x* не нашли;

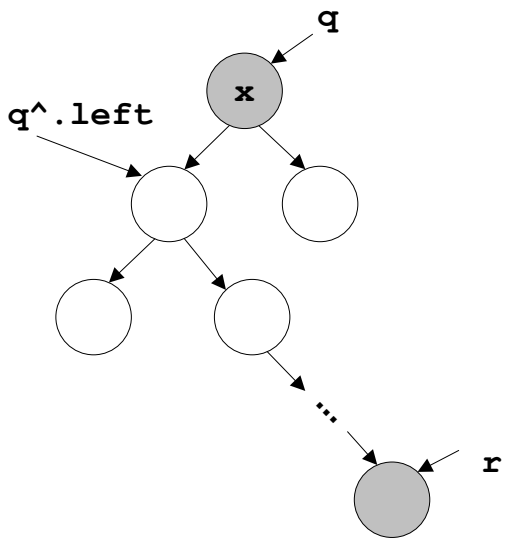
2) узел с ключом *x* имеет одного наследника;

3) узел с ключом *x* имеет двух наследников.

Удаление узла с ключом, равным *x*, из дерева поиска

```
procedure Delete(x : item; var p : TRef);
var
     q : TRef;    {глобальная  переменная }
procedure Del(var r : TRef);
begin
 if r^.rigth<>nil then Del(r^.rigth)
 else
   begin
    q^.key   := r^.key;
    q^.count := r^.count;
    q := r;      {ссылка на узел, который удаляется}
    r := r^.left;
```

```
   end;
end;

begin
 if p = nil then Writeln('элемент не нашли')
 else
    if x < p^.key then Delete(x, p^.left)
    else
      if x > p^.key then Delete(x, p^.rigth)
      else     {      x  =  p^.key}
         begin
          q := p;{ссылка на узел, который удаляется}
          if q^.rigth = nil then p := q^.left
          else
              if q^.left = nil then p := q^.rigth
              else Del(q^.left);
          Dispose(q);
         end;
end;
```



, 2021

1.4. Методы разработки алгоритмов

Алгоритмы «разделяй и властвуй»

Возможно, самым важным и наиболее распространенным методом проектирования эффективных алгоритмов является метод декомпозиции (метод «разделяй и властвуй», или метод разбиения). Этот метод предусматривает такую декомпозицию (разбиение) задачи на более мелкие задачи, что на основе их решения можно легко получить решение первоначальной задачи. Этот метод хорошо подходит при программировании рекурсивных алгоритмов. Рассмотрим

далее несколько практических задач и заметим, что таким способом можно решить множество других задач.

Задача о Ханойских башнях

В одной из древних легенд говорится следующее. «В храме Бенареса находится бронзовая плита с тремя алмазными стержнями. На один из стержней Бог при сотворении мира надел 64 диска различного диаметра из чистого золота так, что самый большой диск лежит на бронзовой плите, а остальные образуют пирамиду, которая сужается кверху. Это – башня Брамы. Работая день и ночь, жрецы переносят диски с одного стержня на другой, подчиняясь законам Брамы:

1) диски можно перемещать с одного стержня на другой только по одному;

2) нельзя класть больший диск на меньший.

Когда все 64 диска будут перенесены с одного стержня на другой, и башня, и храмы, и жрецы-бромиды превратятся в пыль, и наступит конец света». Задачу и легенду придумал математик Э. Люка в 1883 г.

Это легенда родила математическую задачу о Ханойской башне.

Задача. Существует три стержня A, B, C. На первом из них нанизана пирамида из n дисков ниспадающего диаметра. Нужно разместить диски на третий стержень при помощи второго в том же порядке, причем разрешено перекладывать только по одному диску и нельзя класть больший диск на меньший.

Разработайте алгоритм решения задачи о Ханойских башнях на ПК и просчитайте, на какой период времени откладывается наступление конца света в рамках постановки задачи для разных значений ***n***. Отобразите полученные расчеты в таблице 12.

Таблица 12 – Пример таблицы для заполнения.

Количество дисков		**Количество перемещений**	**Период времени выполнения задачи**

Куда уходит детство и задача о конце света…

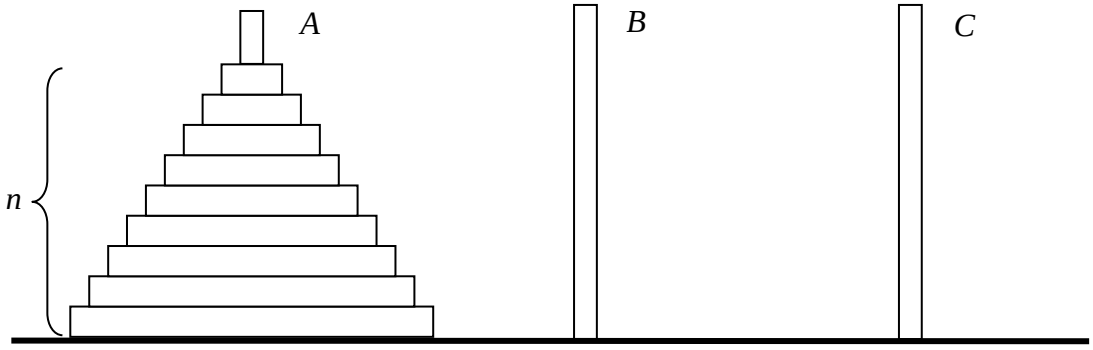
Подсказки для решения задачи

1. Если в детстве Вы снимали по одному кольцу, откладывая его в сторону, то с возрастом Вы можете снять большее количество?
2. У Вас появились еще 2 пустые пирамидки и их можно использовать для хранения снятых колец.
3. Как лучше это сделать, чтобы получить пирамидку на соседнем основании?

Предлагайте.

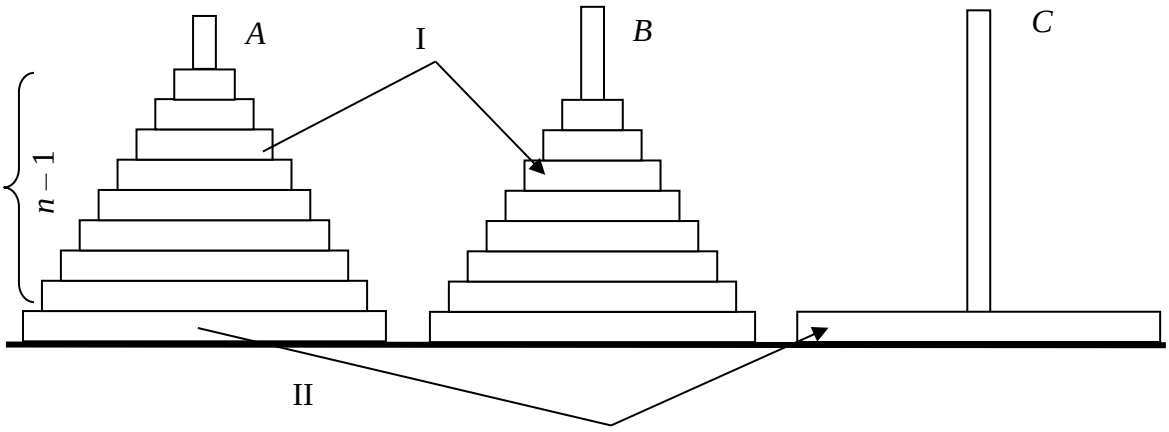
Задача о ханойских башнях

Обсуждение. Рассмотрим несколько подходов для решения этой задачи. Первый из них – рекурсивный. Сначала рассмотрим иллюстрацию, на которой проследим процесс перекладывания дисков.

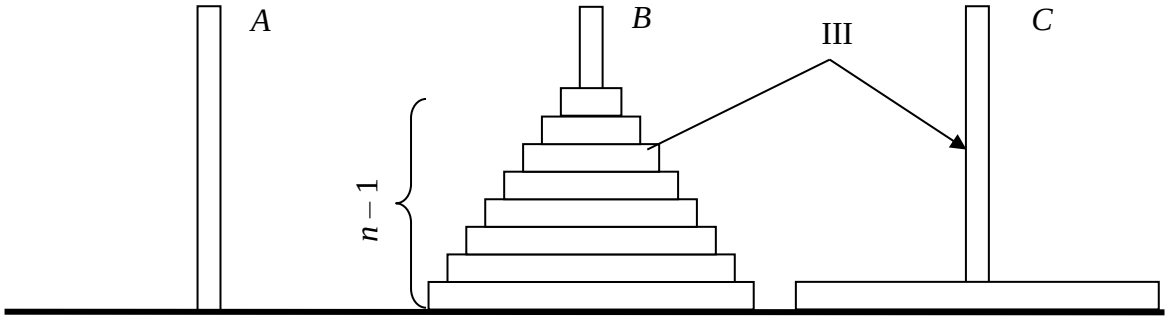


Алгоритм 1.

1. Переложить верхние $n-1$ дисков со стержня A на вспомогательный стержень B.
2. Нижний диск с первого стержня переложить на третий стержень C.



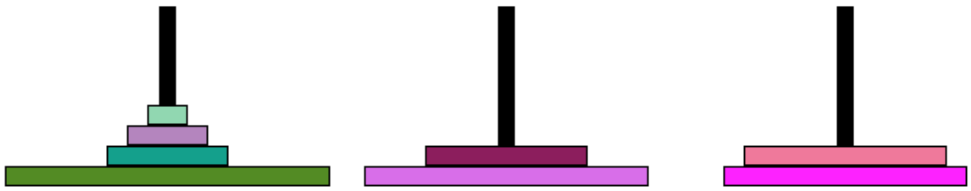
3. Переложить $n-1$ дисков с вспомогательного стержня B на третий стержень C.



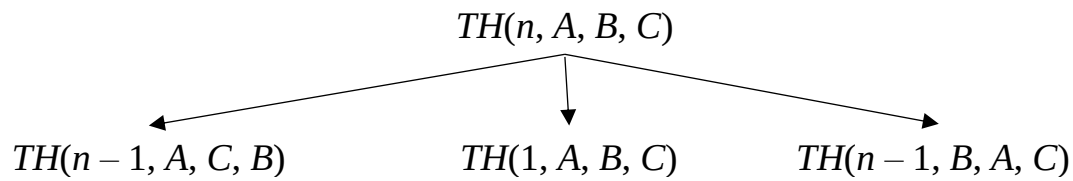
Обсуждение.

Пример для 8 дисков. Промежуточное состояние.

Число перемещений дисков равно 72



Число перемещений дисков равно 255



Если обозначить процесс перекладывания дисков алгоритмом $TH(n, A, B, C)$, то шаги 1–3 перейдут в следующие вызовы алгоритма TH:

$TH(n-1, A, C, B)$, $TH(1, A, B, C)$, $TH(n-1, B, A, C)$.

Давайте переложим самостоятельно диски для $n = 3$. Получим следующие указания для перемещения дисков:

1→3 1→2 3→2 1→3 2→1 2→3 1→3

Опишем рекурсивную процедуру, соответствующую полученному алгоритму.

```
program  Xan;

var
     m : Integer;

procedure TH(n, a, b, c : Byte);
  begin
    if n > 1 then
      begin
        TH(n - 1, a, c, b);
        TH(1, a, b, c);
        TH(n - 1, b, a, c);
      end
    else Write(a:4, '->', c);
  end;

begin
  Readln(m);
  Writeln('m = ', m);
  TH (m, 1, 2, 3);
  Readln;
end.
```

Лемма. Пусть $P(n)$ – количество перемещений n дисков. Наименьшее количество перемещений дисков $P(n) = 2^n - 1$.

Доказательство. Поскольку в рассматриваемом алгоритме выполняются перекладывания дисков по закону: $P(n+1) = P(n) + P(1) + P(n)$, то докажем формулу $P(n) = 2^n - 1$ по индукции.

1. $P(1) = 1$ правильно.

2. Пусть $P(n) = 2^n - 1$ правильно.

3. Докажем, что $P(n+1) = 2^{n+1} - 1$ правильно.

Подсчитаем с учетом пункта 2

$P(n+1) = P(n) + P(1) + P(n) = 2 * P(n) + P(1) = 2 * (2^n - 1) + 1 = 2^{n-1} - 1.$

Формула доказана.

Далее самостоятельно докажите, что это наименьшее число перемещений ■

Когда $n = 64,$ то число перемещений будет

$2^{64} - 1 \approx 2^4 * (2^{10})^6 = 16 * 1024^6 \approx \approx 16 * 10^{18}.$

Если на перемещение одного диска взять 1 с, то потребуется $> 10^{19}$ с, а это будут миллионы лет. Значит, конец света будет еще не скоро.

Рассмотрим еще другие подходы.

Построим дерево с тремя ветвями по следующему алгоритму, который фактически повторяет предыдущий алгоритм 1.

Алгоритм 2.

1. В корневом элементе поместим исходные данные для дисков: n; A, B, C.

2. В левом поддереве будем строить дочернее поддерево для решения такой задачи: $n-1;$ $A, C, B.$

3. В среднем поддереве сформируем узел для решения простой задачи: $1;$ $A, B, C.$

4. В правом поддереве будем строить дочернее поддерево для решения такой задачи: $n-1;$ $B, A, C.$

После того как дерево будет построено повторением пунктов 1-4, сделаем обход дерева слева направо, при этом если в вершине будет стоять один элемент, тогда напечатаем инструкцию для перемещения его с одного стержня на другой.

Получим следующую программу.

```
Program  Hanoy_Tree;
uses crt;
type
    TRef  = ^TNode;
    TNode = record
              key                : Byte;
              a, b, c            : Byte;
              left, mittl, rigth : TRef;
            end;
var
  root : TRef;
procedure  Tree(var t:TRef; n:Byte; i, j, k:Byte);
 begin
  New(t);
  with t^ do
      begin
       key   := n;
       a     := i;
       b     := j;
       c     := k;
       rigth := nil;
```

```
       mittl := nil;
       left  := nil;
      end;
  if n > 1 then
    begin
     tree(t^.left, n - 1, i, k, j);
     tree(t^.mittl, 1, i, j, k);
     tree(t^.rigth, n - 1, j, i, k);
    end;
 end;

procedure Order (t : TRef);
 begin
  if t <> nil then
     if t^.key = 1 then
        Write (t^.a, '->', t^.c, '    ')
     else
        begin
         Order(t^.left);
         Order(t^.mittl);
         Order(t^.rigth);
        end;
 end;

begin
 ClrScr;
 Writeln('  n = 4');
 Tree(root, 4, 0, 1, 2);
 Readln;
 Order(root);
 Readln;
end.
```

Получим следующие перемещения: `n = 4`:

```
0->1  0->2  1->2  0->1  2->0  2->1  0->1  0->2  1->2
1->0  2->0  1->2  0->1  0->2  1->2
```

Упражнение. Запрограммируйте задачу с помощью рекурсивной функции `Tree`.

Предлагается самостоятельно решить следующие задачи:

1. Разработайте алгоритм решения задачи о Ханойских башнях в предположении, что в наличии имеется 4 стержня.
2. Возможны ли такие алгоритмы и будут ли они с точки зрения времени выполнения оптимальнее предложенного?

Задание. Напишите программу, которая в графическом режиме по одному из полученных выше алгоритмов демонстрирует перекладывание дисков.

Перестановки чисел

Задача получения всех перестановок из заданных чисел является важной для большого количества комбинаторных задач. Рассмотрим ее решение методом «разделяй и властвуй».

Задача. Имеется n разных чисел. Напечатать все возможные перестановки этих чисел.

Решение 1. Перестановки индексов элементов массива будем получать при помощи рекурсивной процедуры и затем печатать перестановки собственно элементов массива.

```
Program Permutations;
uses Crt;
const
     n = 4;
type
    TItem   = Integer;
    TMasIndex    = array [1..n] of Byte;
    TMasItem = array [1..n] of TItem;
var
    i : Integer;
    C : TMasItem;

procedure Permutation (C: TMasItem; n : Byte);
  var  i : Byte;
       A : TMasIndex;

procedure Permut (A: TMasIndex; m: Byte);
 var
      j : Byte;
  procedure  Swap (m, j : Byte);
    var
        B : TItem;
    begin
     B    := A[m];
     A[m] := A[j];
     A[j] := B;
    end;
procedure  WriteMas;
  var  i : Byte;
  begin
   for  i := 1  to  n  do
        Write (C[A[i]] : 4);
   Write(' ' : 4);
  end;

  begin
   for  j :=  m  to  n  do
```

```
      begin
       if  j <> m  then
          begin
           Swap (m, j);
           WriteMas;
          end;
       Permut (A, m + 1);
      end;
  end;
 begin
  for  i := 1  to  n  do
     A[i] := i;
  for  i := 1  to  n  do
     Write (C[A[i]] : 4);
   Write(' ' : 4);
  Permut (A, 1);
 end;

begin
 ClrScr;
 for  i := 1  to  n  do
     C[i] := n+1-i;
 Writeln;
 Writeln('Array C: ');
 Permutation (C, n);
 Readln;
end.
```

Получим следующую последовательность перестановок индексов массива *C*:

```
1  2  3  4    1  2  4  3    1  3  2  4    1  3  4  2
1  4  2  3    1  4  3  2    2  1  3  4    2  1  4  3
2  3  1  4    2  3  4  1    2  4  1  3    2  4  3  1
3  1  2  4    3  1  4  2    3  2  1  4    3  2  4  1
3  4  1  2    3  4  2  1    4  1  2  3    4  1  3  2
4  2  1  3    4  2  3  1    4  3  1  2    4  3  2  1
```

Комментарии. Здесь расходуется память при хранении локальных параметров рекурсивной процедуры.

Рассмотрим еще один вариант решения.

Р е ш е н и е 2. Будем строить лексикографические перестановки элементов, и получается следующий алгоритм нерекурсивной перестановки.

1. От конца к началу перестановки ищем первый элемент $b[i]$, такой, что $b[i] < b[i + 1]$, и запоминаем его индекс i.

2. От элемента $i + 1$ до конца ищем последний элемент, который больше $b[i]$, и запоминаем его индекс k.

3. Меняем местами эти элементы – $b[i]$ с $b[k]$.

4. Всю группу элементов от (i + 1)-го элемента до последнего попарно меняем местами.

Формализованный алгоритм будет следующим.

Начало.

1. Ввод N.
2. Заполняем массив b последовательно числами от 1 до N.
3. Это первоначальная перестановка, выводим ее.
4. Пока «правда» повторять:
   1) i := N;
   2) пока (i > 0) и (b[i] >= b[i+1]) повторять: i := i – 1;
   3) когда i = 0, то конец работы;
   4) для j от i + 1 до N повторять:
      когда b[j] >= b[i], то k = j;
   5) обмен значений b[i] и b[k];
   6) для j от i + 1 до i + (N + 1 – i) **div** 2 повторять:
      обмен значений b[i] и b[N + 1 + i – j];
   7) вывод текущей перестановки b.

Конец.

Программу напишите самостоятельно.

«Жадные» алгоритмы

На каждом отдельном этапе любой «жадный» алгоритм выбирает тот вариант, который является *локально оптимальным* в том или ином смысле.

Когда единственным способом получения оптимального решения некоторой задачи является использование метода полного перебора, но для этого требуется слишком много времени, тогда «жадный» алгоритм, или другой эвристический метод получения приемлемого (не обязательно оптимального) решения, может оказаться единственным реальным способом его получения.

Однако, не всякий «жадный» алгоритм допускает оптимальное решение в целом, так как на каждом этапе гарантируется мгновенная выгода, но при этом общий результат может оказаться неприемлемым.

Если же нас удовлетворяет «почти оптимальный» результат, то «жадные» алгоритмы часто оказываются самыми быстрыми методами получения решения.

Рассмотрим эти подходы при решении различных задач.

Счастливые билеты

Талон для проезда в городском транспорте имеет шестизначный номер от 000000 до 999999.

Будем считать талон счастливым, если сумма первых трех цифр равна сумме последних трех.

Задача. Посчитать количество счастливых номеров среди шестизначных номеров.

Эвристический подход приводит нас к следующему *алгоритму* 1:

- перебираем все номера от 000000 до 999999,

- выделяем цифры в каждом числе и
- проверяем, является ли билет счастливым.

```
Program Happy_tickets_1;
   {Полный перебор с выделением цифр числа}
var
   k, i, i1 : Longint;  j : Integer;
   a        : array[1..6] of Integer;
begin
k:=0;
  for i := 0  to  999999  do
   begin
    i1 := i;   {Выделение цифр}
    for j := 6   downto  1   do
       begin
        a[j]:=i1 mod 10;
        i1:=i1 div 10;
       end;
    if a[1]+a[2]+a[3]=a[4]+a[5]+a[6] then Inc(k);
   end;
  Writeln('k= ',k); Readln;
end.
```

Разумеется, что это «жадный» алгоритм и он не самый худший, но есть и алгоритмы лучше.

Алгоритм 2. Во втором варианте формируем сразу все шесть цифр числа, т. к. в предыдущем варианте какое-то времени тратиться на выделение цифр.

```
Program Happy_tickets_2;
var
      i, j, k, l, m, n : Byte;
                 Count : Word;
begin
 Count := 0;
 for i := 0 to 9 do
  for j := 0 to 9 do
   for k := 0 to 9 do
    for l := 0 to 9 do
     for m := 0 to 9 do
      for n := 0 to 9 do
       if i + j + k = l + m + n then Inc(Count);
 Writeln('количество = ', Count);
 Readln;
end.
```

Напечатается 55252.

Анализ. Шесть вложенных циклов осуществляют перебор всех вариантов, а их 10^6 – миллион, а нашли 1/18 часть всего перебора. Разумеется, что это нерациональный алгоритм, неэффективный по времени ожидания.

Разработаем другой подход, который будет совершать подсчет только счастливых билетов.

Алгоритм 3. Будем считать, сколько раз выйдет сумма цифр, равная 0, 1, 2, …, 3 * 9 в одной половине и во второй.

Очевидно, что количество «счастливых» билетов с определенной суммой первых трех цифр равно $a[sc]^2$.

Найдем сумму первых трех цифр и увеличим соответствующий элемент массива на единицу.

```
Program Happy_ticket_3;
var
    i, j, k : Byte;
    Count   : Word;
    a       : array[0..9 * 3]  of  Word;

begin
 Count := 0;
 for i := 0 to 27 do a[i] := 0;
 for i := 0 to 9 do
    for j := 0 to 9 do
        for k := 0 to 9 do
            Inc(a[i + j + k]);
 for i := 0 to 27 do
    count := count + a[i] * a[i];
 Writeln('количество = ', Count);
 Readln;
end.
```

В этом варианте мы сэкономили время (примерно в 20 раз), но незначительно отняли память. Существуют и другие способы усовершенствования этого подхода, которые оптимизируют циклы, но они нас не очень интересуют, т. к. рассчитаны на шестизначные номера билетов.

И это существенный недостаток рассмотренных алгоритмов.

Если необходимо совершить подсчет для номеров, разрядность которых будет варьироваться или может стать известной, например, только во время выполнения программы, то необходимо иначе запрограммировать наши вложенные циклы.

Рассмотрим следующий неожиданный подход, который можно применять в процессе выполнения программы и там, где необходимо моделировать десятичные алгоритмы, и там, где нужно подсчитать значение вложенных сумм в не заданном наперед количестве.

Есть такой механизм, который называется «одометр». Проимитируйте работу одометра, например четырехразрядного. Заметили, что вы имеете дело со

схемой работы вложенных циклов? Применим этот принцип и улучшим предыдущую программу.

```
Program Happy_ticket_4;

const
      k = 4;     {2 * k разрядное число}

var
     i         : Byte;
     Sum_Digit : Byte;
     Count     : Longint;
     a         : array[0..9 * k]  of  Longint;
     Cp        : array[0..k] of  Byte;

begin
 for i := 0 to k * 9 do a[i] := 0;
    {               количество возможных сумм цифр}
 for i := 0 to k  do Cp[i] := 0; {Обнулили одометр}
   {одометр переполняется при Cp[0]=1. Конец вычислений}
 repeat
      Sum_Digit := 0;
      for  i := 1  to  k  do
           Sum_Digit := Sum_Digit + Cp[i];
           {    подсчитали текущую сумму цифр
                                на одометре}
      Inc(a[Sum_Digit]);
      i := k;
           {         переходим на следующую
                 комбинацию цифр одометра}
       while  Cp[i] = 9  do
              begin
               Cp[i] := 0;
               Dec(i);
              end;
      {Сбросили все цифры 9 в конце одометра
             и прибавили 1 в нужной позиции}
       Cp[i] := Cp[i] + 1;
 until  Cp[0] = 1;
 Count := 0;
 for  i := 0  to  k * 9    do
      count := count + a[i] * a[i];
 Writeln('количество = ', Count);
 Readln;
end.
```

Комментарии. Для сохранения последовательности цифр одометра используем массив `Cp[i], i = 0 ... k`, в элементах которого записывается текущий набор цифр.

Для перехода к следующему варианту нужно прибавить 1 к k-й (последней) цифре числа. Но это можно сделать только тогда, когда она не равна 9.

Если цифра равна 9, то ее необходимо сбросить в ноль, и прибавить 1 к предыдущей.

Так работаем до тех пор, пока не найдем цифру, не равную 9. Последовательность $\underbrace{09\ldots9}_{k}$ будет последней, так как по нашему алгоритму получим $\underbrace{10\ldots0}_{k}$, т. е. `Cp[0]=1,` а это признак окончания цикла.

Можно придумать еще несколько алгоритмов решения этой задачи, однако последний алгоритм можно применять для решения задач по математике.

Задание 1. Напишите программу, которая подсчитывает сумму: $S = \sum_{i_k=1}^{n} \cdots \sum_{i_1=1}^{n} \frac{1}{i_1 + \cdots i_k}$.

Задание 2. Напишите программу, которая подсчитывает сумму: $S = \sum_{i_k=a_k}^{b_k} \cdots \sum_{i_1=a_1}^{b_1} u(i_1,\ldots,i_k)$ для заданных чисел a_1, ..., a_k, b_1, ..., b_k..

Задача получения сдачи эвристическим методом

Рассмотрим упрощенную задачу выдачи сдачи монетами следующего номинала (копеек): 25, 10, 5 и 1. Пусть сдача составляет 63 копейки. Тогда можно сдачу выдать следующим образом:

$63 = 2 \cdot 25 + 1 \cdot 10 + 3 \cdot 1$.

Количество монет: $2 + 1 + 3 = 6$.

Этот эвристический подход принадлежит к «жадным» алгоритмам и состоит в следующем.

1. Для сдачи, равной 63, выбираем наибольшее количество монет самого большого номинала $\left[\frac{63}{25}\right] = 2$.
2. Сдача уменьшается: $63 - 2 \cdot 25 = 13$.
3. Для сдачи, равной 13, выбираем наибольшее количество монет самого приемлемого наибольшего номинала $\left[\frac{13}{10}\right] = 1$.

4. Сдача снова уменьшится: $13 - 1 \cdot 10 = 3$.
5. Для сдачи равной 3 выбираем наибольшее количество монет самого приемлемого наибольшего номинала $\left[\frac{3}{1}\right] = 3$.
6. Сдача уменьшится: $3 - 3 \cdot 1 = 0$.
7. Конец.

В результате, поскольку значение остатка сдачи с каждым разом уменьшалось и стало равным 0, мы получили в некотором смысле оптимальное решение. Нам понадобилось 6 монет, чтобы набрать сумму нужной сдачи.

Оказывается, что жадные алгоритмы не всегда дают оптимальный вариант.

Приведем пример.

Рассмотрим такой вариант: монеты номинала 11, 5 и 1, остаток – 15 копеек.

Наш жадный алгоритм даст результат: $15 = 1 \cdot 11 + 4 \cdot 1$.

Количество монет – 5.

Но очевидно, что остаток в 15 копеек таким номиналом монет можно отсчитать и так: $15 = 3 \cdot 5$. Количество монет – 3.

Значит, мало того, что «жадный» алгоритм в целом не всегда обеспечивает оптимальное решение, но бывают ситуации, когда он вообще не даёт решения.

Это может быть, например, когда каких-то монет определенного номинала – ограниченное количество.

Далее мы более подробно рассмотрим этот эвристический метод.

Задача. Разработать подпрограмму для микропроцессора, которая определяет остаток в кассовых аппаратах для систем монет в 1, 2, 3, 5, 10, 15, 20 и 50 копеек.

Параметры подпрограммы:

- x ($x > 0$) – сумма остатка;
- q[1..8] – первоначальное количество монет разного номинала соответственно перечню, данному выше.

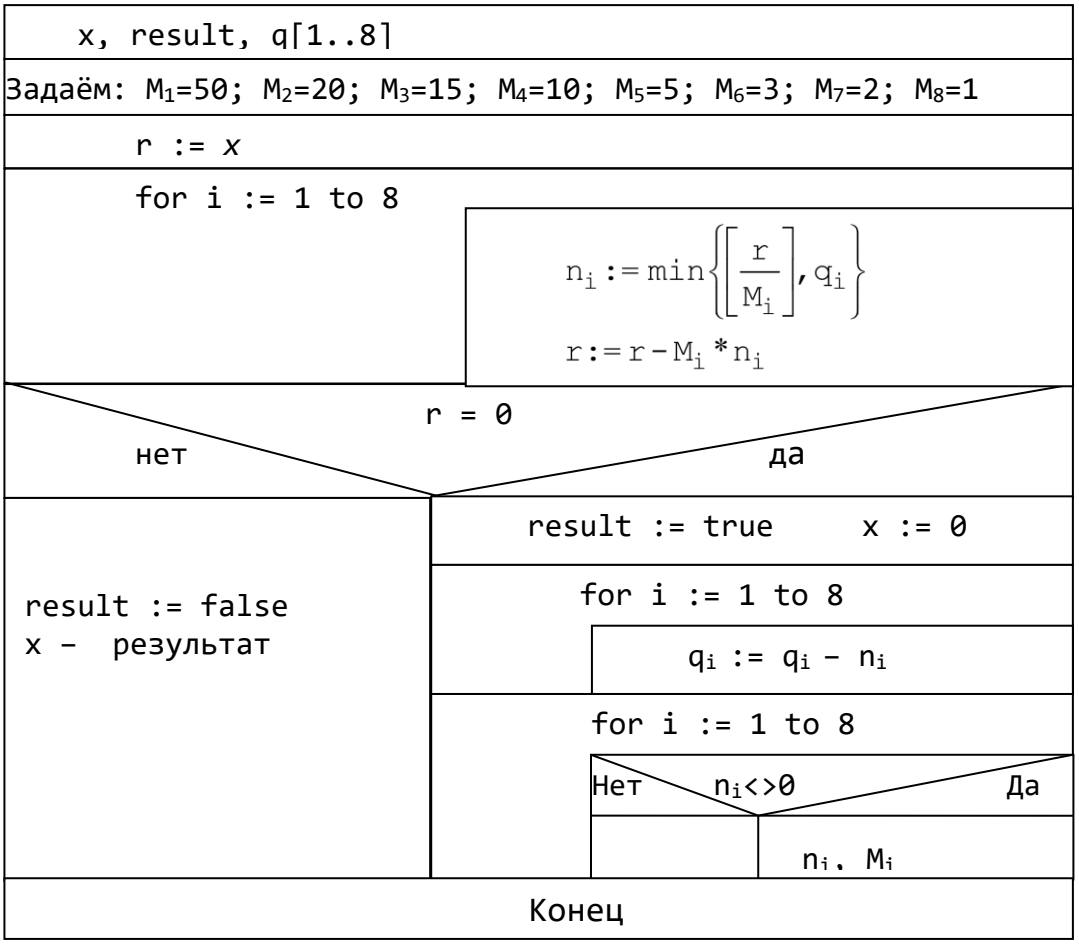
Результатом работы программы будет булевское значение **`true`**, массив количества монет разного номинала, который обеспечивает выплату сдачи, и измененный исходный массив наличия монет, или **false**, если сдачу набрать невозможно (или не хватает монет некоторого номинала).

Проверку алгоритма выполним на следующей совокупности монет (таблица 13):Таблица 13 – Исходные данные.

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
M_i	50	20	15	10	5	3	2	1
q_i	3	4	8	0	1	1	0	1

Эвристический подход, который построен по принципу "жадного" алгоритма и не позволяет рассмотреть разные варианты, по большому счету будет заводить в тупик.

Оговоренный ранее эвристический алгоритм представим структурограммой.



Очевидно, что для $x = 67$ и $x = 60$ мы не получим положительного ответа по нашему алгоритму, потому что нам не хватит монет малого номинала.

Однако для $x = 60$ можно было бы предположить такие совокупности: три монеты по 20 копеек или четыре монеты по 15 копеек.

В следующей лекции рассмотрим строгий подход, который позволит перебрать все варианты выдачи сдачи. Разумеется, что для микропроцессора он будет неприемлемым.

Алгоритмы на полный перебор

Задача получения сдачи полным перебором

Нужно заметить, что при каждом добавлении к выплате сдачи очередной монеты возникает такая же задача, но с уменьшенной суммой сдачи и с другим набором монет q_i, $i = \overline{1,n}$.

Определим алгоритм задачи получения остатка как булевскую функцию «остаток возможен» следующим образом:

$$
\begin{aligned}
RM(x;\ q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8) := (x = 0 \vee \\
\vee (x \geq 50 \wedge q_1 \geq 1 \wedge RM(x-50;\ q_1 - 1, q_2, q_3, q_4, q_5, q_6, q_7, q_8)) \\
\vee (x \geq 20 \wedge q_2 \geq 1 \wedge RM(x-20;\ q_1, q_2 - 1, q_3, q_4, q_5, q_6, q_7, q_8)) \\
\vee (x \geq 15 \wedge q_3 \geq 1 \wedge RM(x-15;\ q_1, q_2, q_3 - 1, q_4, q_5, q_6, q_7, q_8)) \\
\vee (x \geq 10 \wedge q_4 \geq 1 \wedge RM(x-10;\ q_1, q_2, q_3, q_4 - 1, q_5, q_6, q_7, q_8)) \\
\vee (x \geq 5\ \wedge q_5 \geq 1 \wedge RM(x-5;\ \ q_1, q_2, q_3, q_4, q_5 - 1, q_6, q_7, q_8)) \\
\vee (x \geq 3\ \wedge q_6 \geq 1 \wedge RM(x-3;\ \ q_1, q_2, q_3, q_4, q_5, q_6 - 1, q_7, q_8)) \\
\vee (x \geq 2\ \wedge q_7 \geq 1 \wedge RM(x-2;\ \ q_1, q_2, q_3, q_4, q_5, q_6, q_7 - 1, q_8)) \\
\vee (x \geq 1\ \wedge q_8 \geq 1 \wedge RM(x-1;\ \ q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8 - 1))).
\end{aligned}
$$

Если в некоторый момент $RM(y; q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8)$ будет иметь $y = 0,$ тогда решение $RM(\cdots)$ будет ***true*** (поскольку так работает $a \vee b$), иначе будет *false.*

В таком рекурсивном варианте, если решение работы функции *false,* это лишь означает, что на данном наборе q_i, $i = \overline{1,n},$ остаток получить вовсе невозможно.

Напишем программу по указанному алгоритму согласно интуитивной логике разработки.

```
Program Get_Change_Func;
{$R+}
Type
       TMas = array [1..8] of Byte;
const
      M          : TMas = (50, 20, 15, 10, 5, 3, 2, 1);
      QQ         : TMas = (4,   3,  8,  4, 1, 1, 1, 1);
Var
      b          : Boolean;
      Q          : TMas;
      k, S, i, x : Word;
  Count1, count2 :longint;

procedure Print(y : TMas);
var
  i : Integer;
begin
 Write('VARIANT: ', '   ');
 count2:=count2+1;
 for i := 1  to  8  do
    if y[i]<>0 then Write(M[i]:2, '-',y[i], '  ');
```

```
 Writeln;
end;
                                                {GT=Get Change}
function  GC (x : byte; Q : TMas) : Boolean;
            {2. На заглавие обратим внимание потом}

  var
       b   :  Boolean;
       I,j :  Integer;
       D   :  TMas;
  begin
  if  x = 0  then
     begin
      GC := true;
      for j := 1 to 8 do D[j] := QQ[j] – Q[j];
      Print (D);
     end
  else
     begin
{3. На следующее присваивание обратим внимание потом}
      I := 1;
      while (I <= 8) and (x < M[i]) do
             I := I + 1;
      for j := I  to  8  do
          if Q[j] > 0  then
            begin
             D    := Q;
             D[j] := D[j] – 1;
             count1:=count1+1;
{1. На следующее присваивание обратим внимание потом}
             b    := GC(x – M[j],D);
            end;
       end;
 end;

begin
 count1:=0;
 count2:=0;
 Q := QQ;
 x := 60;
 b := GC(x, Q);
 Writeln(' X = ', x);
 Writeln(  '   "50"':6, '"20"':6, '"15"':6,'"10"':6,
           ' 5"':6, '" 3"':6, '" 2"':6, '" 1"':6);
 for  I := 1  to  8  do  Write (q[i]:6);
 Writeln;
 Writeln(Count1:19, ' samples  ', count2:16, ' variants' );
```

```
 if  count2 = 0  then  Writeln('not  variants ');
 Readln;
end.
```

Программа напечатает результат:

```
VARIANT:    50-1  10-1
VARIANT:    50-1   5-1   3-1   2-1
VARIANT:    50-1   5-1   3-1   2-1
VARIANT:    50-1   5-1   3-1   2-1
VARIANT:    50-1   5-1   3-1   2-1
VARIANT:    50-1   5-1   3-1   2-1
VARIANT:    50-1   5-1   3-1   2-1
VARIANT:    20-3
VARIANT:    20-2  15-1   5-1
…

VARIANT:    15-2  10-2   5-1   3-1   2-1
VARIANT:    20-1  10-3   5-1   3-1   2-1
 X = 60
   "50"  "20"  "15"  "10"    5"  " 3"  " 2"  " 1"
     4     3     8     4     1     1     1     1
              62438 samples              3753 variants
```

Оказывается, что рассмотренный вариант программы на исходных данных выполняет 62438 проб, получает 3753 варианта. Видно, что повторов, не дающих новый уникальный вариант, очень много. А реально только 20 вариантов уникальные.

Чтобы уменьшить количество проб в программу внесем одно улучшение алгоритма: перед вызовом функции GC нужно проконтролировать, чтобы в наличии гарантированно была необходимая сумма денег, иначе очевидно, что мы все равно не сможем набрать необходимую сумму. А именно на месте комментария

```
{1. На следующее присваивание обратим внимание потом}
```

и оператора

```
b := GC(x – M[j],D);
```

вставим такую последовательность операторов

```
S    := 0;
for  k := I  to  8  do
     S := S + D[k] * M[k];
if S + M[j] >= x then
   b := GC(x – M[j],D);
```

После внесенных правок программа напечатает 60698 проб и снова 3753 варианта, из которых только 20 уникальных.

Очевидно, что получилось много повторов. Нужно от них избавиться.

Продумав, где мы напрасно рекурсивно вызываем функцию GC, избавимся от таких вызовов. Для этого проанализируем следующие варианты.

Пусть $x = 60$. Получим первый вариант: $20 + 2 \cdot 15 + 10$, второй вариант: $15 \cdot 2 + 20 + 10$, третий вариант: $10 + 20 + 2 \cdot 15$ и т. д. Нужно так запрограммировать алгоритм, чтоб оставался первый вариант, если монеты идут с номиналом, который уменьшается. Затем пробуя монеты с меньшим номиналом, например, на 10 копеек, не должно быть перехода на монеты в 20 или 50 копеек. Это место в программе отмечено комментарием 3:

```
{3. На следующее присваивание обратим внимание потом}
                              I := 1;
```

Переменная в операторе `I := 1;` должна получить значение не единицы, указывающей, что снова будут участвовать монеты с большим номиналом, а той позиции, с которой происходит вызов функции `GC`.

Поэтому внесем коррективы в описание заголовка функции, а именно:

```
function  GC (x : byte; Q : TMas; w : byte) : Boolean;
```

а также в вызов ее (на месте комментария 3).

Сейчас получим следующий вариант программы.

```
Program Get_Change_Func_New;
{$R+}
type
     TMas = array [1..8] of Byte;
const
      M  : TMas = (50, 20, 15, 10, 5, 3, 2, 1);
      QQ : TMas = (4, 3, 8, 4, 1, 1, 1, 1);
var
     b    : Boolean;
     Q    : TMas;
     k, S : Word;
     c    : char;

procedure Print(y : TMas);
  var
     i : Integer;
begin
   Write('VARIANT: ', '   ');
   for  i := 1  to  8  do
       if y[i]<>0 then Write(M[i]:2,'-',y[i],'   ');
   Writeln;
  end;

function GC(x:byte; Q:TMas; w:Byte): Boolean;
  var
       b   :  Boolean;
       I,j :  Integer;
       D   :  TMas;
  begin
```

```
   if  x = 0  then
     begin
       GC := true;
       for j:=1 to 8 do D[j]:=QQ[j]-Q[j];
       Print (D);
     end
   else
    begin
     I := w;
     while (I<=8) and (x<M[i]) do I:=I+1;
     for j := I  to  8  do
        if Q[j] > 0 then
          begin
           D    := Q;
           D[j] := D[j] - 1;
           S    := 0;
           for  k := I  to  8  do
                   S := S + D[k] * M[k];
           if  S + M[j] >= x then
               b := GC(x - M[j], D, j);
           end;
             if I > 8  then GC := false;
      end;
 end;

begin
 Q := QQ;
 b := GC(60, Q, 1);
 Writeln(b);
 Readln;
end.
```

Программа напечатает 20 вариантов выплаты сдачи, выполнив 268 проб. Данные изменения в программе принесли огромную вычислительную экономию.

```
    VARIANT:    50-1  10-1
    VARIANT:    50-1   5-1   3-1   2-1
    VARIANT:    20-3
    VARIANT:    20-2  15-1   5-1
    VARIANT:    20-2  15-1   3-1   2-1
    VARIANT:    20-2  10-2
    VARIANT:    20-2  10-1   5-1   3-1   2-1
    VARIANT:    20-1  15-2  10-1
    VARIANT:    20-1  15-2   5-1   3-1   2-1
    VARIANT:    20-1  15-1  10-2   5-1
    VARIANT:    20-1  15-1  10-2   3-1   2-1
    VARIANT:    20-1  10-4
```

```
VARIANT:    20-1  10-3   5-1   3-1   2-1
VARIANT:    15-4
VARIANT:    15-3  10-1   5-1
VARIANT:    15-3  10-1   3-1   2-1
VARIANT:    15-2  10-3
VARIANT:    15-2  10-2   5-1   3-1   2-1
VARIANT:    15-1  10-4   5-1
VARIANT:    15-1  10-4   3-1   2-1
 X = 60
   "50"  "20"  "15"  "10"    5"  " 3"  " 2"  " 1"
     4     3     8     4     1     1     1     1
                 268 samples                 20 variants
```

Задача получения остатка с помощью сильноветвистых деревьев

По функции *GC*(…) можно построить сильноветвистое дерево с восемью ветвями. Если $q_i > 0,\ i = \overline{1,n}$, то такое дерево полное, иначе там, где $q_i = 0$ или какое-то количество $x - M_i \le 0$, построение поддерева прекращается.

После построения дерева нужно обойти все построенные нетерминальные узлы, и, если встретится узел, в котором $y = 0,$ можно печатать результат: $n_i = Q_i - q_i,\ i = \overline{1,8}$, где $Q_i,\ i = \overline{1,n}$, изначально заданный набор монет.

Прежде чем писать программу, построим самостоятельно дерево для следующего набора монет и исходной суммы остатка $x = 30$ (таблица 14):

Таблица 14 – Пример исходных данных.

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**		**8**
$\boldsymbol{M}_i$	50	20	15	10	5	3	2		1
$\boldsymbol{q}_i$	4	3	8	4	1	1	1		1

Теперь напишем алгоритмически упрощенный вариант программы (исключительно с целью получения навыков работы с сильноветвистыми деревьями), которая построит дерево без учета последних замечаний, приведенных нами в предыдущем варианте.

В соответствии с рекурсивной определением функции *GC*(…) получим такое дерево, в котором каждый узел будет иметь максимально 8 ветвей, соответствующих всем $q_i > 0.$

Используя этот подход, далее найдем в дереве узлы, содержащие в информационной части *x*=0, и подсчитаем их количество.

```
{программа строит дерево}
Program  Get_Change_Tree_Simple;
uses crt;
```

```
type
     TMas = array[1..8]  of  byte;
     TRef = ^TNode;
     TNode = record
              key : word;
              g   : TMas;
              r   : array[1..8] of TRef;
             end;

var
    I, k, S    : word;
    L, L2      : LongInt;
    x          : Word;
    n          : TMas;
    root       : TRef;

const
    M          : TMas = (50, 20, 15, 10, 5, 3, 2, 1);
    q          : TMas = ( 4,  3,  8,  4, 1, 1, 1, 1);

procedure  Tree(var t:TRef; x:Word; a:TMas);
var
     I, j : Integer;
     d    : TMas;
begin
 L := L + 1;
 New (t);
 with t^  do
  begin
   key := x;
   for  I := 1  to  8  do
       r[i] := nil;
   if x>0 then g := a
          else
            for  I := 1  to  8  do
                 g[i] := q[i]-a[i];
  if  x > 0  then
   begin
    I := 1;
    while (I <= 8) and (x < M[i]) do
         I := I + 1;
    for  j := I  to  8  do
         with  t^  do
               if  g[j] > 0   then
                  begin
                   S := 0;
                   for  k := I  to  8  do
```

```
                        S := S + g[k] * M[k];
                    if  S + M[j] >= x then
                       begin
                        d    := g;
                        d[j] := g[j] - 1;
                        Tree(r[j], x - M[j], d);
                       end;
                               end;
        end;
    end;
end;

procedure  Count(t : TRef; Var L : longint);
  var
     I : Integer;
  begin

   if t <> nil  then
     if  t^.key = 0  then  L := L+1
                     else
                      for I:=1 to 8 do Count(t^.r[i], L);
  end;

begin
 clrscr;
 Root  := nil;
  x := 60;
 L := 0;
 Tree (root, x, q);
 Writeln (  '  Tree:    L = ', L, ' nodes');

 if  Root = nil  then  Writeln('Nil')
                 else  Writeln (' "Good"');
 L2 := 0;
 Count(Root, L2);
 writeln;
 Writeln (  '  Ord:    L2 = ', L2, ' variants');

 Writeln(' x = ', x);
 Writeln(  '   "50"':6, '"20"':6, '"15"':6,'"10"':6,
           '" 5"':6, '" 3"':6, '" 2"':6, '" 1"':6);
 for  I := 1  to  8  do  Write (q[i]:6);
 Writeln;
 Readln;
end.
```

Если в предложенной выше программе проводить вычисления для различных исходных значений *x* получим примерно (зависит от мощности компьютера) следующие результаты (таблица 15):

Таблица 15 – Результаты выполнения программы.

x	**Узлов дерева**	**Вариантов**	**Различных вариантов**
30	1047	118	7
31	1696	637	7
60	54819	3753	20
80	542121	31290	31
120	29629609	1177096	52
x>120	Heap Overflow		71

Понятно, что алгоритм нужно улучшить.

Воспользуемся предыдущим подходом: не строим ветви, когда может произойти переход от монеты с меньшим номиналом к монете с большим номиналом.

```
Program  Get_Change_Tree_New;
     {программа оптимально строит дерево и выводит результат}

type
     TMas = array[1..8]  of  shortInt;
     TRef = ^TNode;
     TNode = record
              key : Integer;
              g   : TMas;
              r   : array[1..8] of TRef;
             end;
var
    I, L, k, S : Integer;
    x          : Word;
    root       : TRef;
const
       M : TMas = (50, 20, 15, 10, 5, 3, 2, 1);
       q : TMas = ( 4,  3,  8,  4, 1, 1, 1, 1);

procedure  Tree(var t:TRef; x:Word; a:TMas; w:Byte);
var
     I, j : Integer;
     d    : TMas;
begin
 L := L + 1;
 New (t);
```

```
 with t^  do
  begin
   key := x;
   for  I := 1  to  8  do
       r[i] := nil;
   if x>0 then g := a
          else
            for  I := 1  to  8  do
                 g[i] := q[i]-a[i];
  if  x > 0  then
   begin
    I := w;
    while (I <= 8) and (x < M[i]) do
         I := I + 1;
    for  j := I  to  8  do
         with  t^  do
               if  g[j] > 0   then
                  begin
                   S := 0;
                   for  k := I  to  8  do
                        S := S + g[k] * M[k];
                   if  S + M[j] >= x then
                      begin
                       d    := g;
                       d[j] := g[j] - 1;
                       Tree(r[j], x - M[j], d, j);
                      end;
                      end;
   end;
  end;
end;

procedure  Order (t : TRef);
var    I : Integer;
begin
 if  t <> nil  then
    if  t^.key = 0  then
       begin
        L := L + 1;
        for  I := 1  to  8  do
            if  t^.g[i] <> 0 then
                 Write(M[i]:2, ' : ', t^.g[i], '  ');
       Writeln;
        end
    else
        for  I := 1  to  8  do
             Order (t^.r[i]);
```

```
end;

begin
 Root := nil;
 x    := 60;
 Writeln(' X = ', x);
 Writeln('"50"':6,'"20"':6,'"15"':6,'"10"':6,
         '" 5"':6, '" 3"':6, '" 2"':6, '" 1"':6);
for  I:=1 to 8 do Write(q[i]:6);
Writeln;
L := 0;
Tree(root, x, q, 1);
Writeln ('         L = ', L, ' nodes');
if  Root = nil  then  Writeln ('  Nil  ')
                 else  Writeln (' "Good"');
 L := 0;
 Order (Root);
 Writeln(   '  Order: L = ', L, ' variants');
 Readln;
end.
```

Если по построенной программе делать просчеты для различных исходных значений *x* и при одном и том же наборе наличия монет, получим примерно следующие результаты:

x	Узлов дерева	Различных вариантов
30	45	7
31	52	7
60	159	20
80	279	31
120	553	52
180	934	71
200	1005	72
260	1034	70

Поиск с возвратом. Задачи искусственного интеллекта

Особенно интересный раздел программирования – это задачи из области «искусственного интеллекта».

Здесь нужно строить алгоритмы, которые находят решение определенной задачи не по фиксированным правилам, а методом проб и ошибок.

Процесс проб и ошибок можно рассматривать в общем виде как поисковый процесс, который постепенно строит и просматривает (а также обрезает) деревья подзадач.

Во многих случаях такие деревья поиска растут очень быстро, обычно экспоненциально, в зависимости от заданного параметра.

Соответственно увеличивается стоимость поиска. Часто деревья поиска можно обрезать, используя только эвристические соображения, и тем самым сводить к количеству подсчетов в разумных пределах.

Рассмотрим общий принцип разбивки таких задач на подзадачи и использования в них рекурсии.

Задача обхода конем шахматной доски

Задана доска, $n \times n$, которая содержит n^2 полей. Конь, который ходит соответственно шахматным правилам, помещается на поле с заданными начальными координатами. Нужно покрыть всю доску ходами коня, т. е. найти обход доски, если он существует, за $n^2 - 1$ ходов такой, что в каждое поле конь заходит ровно один раз.

Очевидно, что эта задача разбивается на две более простые задачи:

- *или выполнить очередной ход,*
- *или установить, что никакой ход невозможен.*

Этот подход можно алгоритмизировать так:

```
procedure  попытка очередного хода;
begin
 инициализация выборки ходов;
 repeat
  выбор следующего возможного хода из списка
  очередных ходов;
  if  ход приемлимый  then
    begin запись хода;
     if  доска не заполнена  then
       begin
         попытка очередного хода;
         if неудовлетворительно then
            стирание последнего хода
       end
    end
 until (ход был удачный)∨(нет других возможных ходов)
end;
```

Мы видим, что получился рекурсивный алгоритм.

На примере этой задаче можно проследить общие свойствами таких алгоритмов, когда происходит возврат на предыдущую позицию после того, как мы зашли в тупик.

Чтобы детализировать этот алгоритм, нужно соответствующим образом подобрать представления данных.

Доску представим матрицей h размера $n \times n$, элементы которой сначала будут равны нулю, а после удачного хода будут равны номеру хода. Номер хода будем запоминать в переменной i, если доска не заполнена, то $i < n^2$,

очередной ход будем запоминать в переменных x, y, предполагаемый ход – в переменных u, v, булевская переменная q будет иметь значение *true*, если ход удачный, и *false* – в противном случае.

После такой детализации получим уже более четкий алгоритм пока на псевдокоде:

```
const
     n = 8;
var
     h : array(1..n,1..n] of Byte;

procedure  Try(i, x, y : Integer; var q : Boolean);
  var
      u, v : Integer;
      q1   : Boolean;
  begin
         инициализация выборки ходов;
   repeat
      пусть u, v - координаты следующего хода из
          списка очередных ходов;
    q1:=false;
    if (1<=u<=n) и (1<=v<=n) и (h[u,v]=0) then
      begin  h[u,v]:=1;
      if  i<sqr(n)  then
        begin
          Try(i+1, u, v, q1);
              {попытка очередного хода}
          if  not q1  then h[u,v]:=0
        {если q1 с false не изменилась на true,}
        {то такой ход завел в тупик            }
        {и надо его вычеркнуть                 }
        end
      else q1:=true          {i□n²}
      end
   until  q1 ∨ (нет других возможных ходов);
   q:=q1;
  end;
```

Здесь появились и другие уточнения.

Еще один этап уточнения, и мы напишем программу уже на алгоритмической языке. Заметим, что до сих пор мы намеренно не конкретизировали выбор следующего хода шахматной фигуры, так как это очень собственная особенность задачи. Более интересно было рассматривать, как происходит возврат, когда мы зашли в тупик.

Рассмотрим далее правила перемещения коня. Если задана начальная пара координат (x, y), то в наилучшем случае имеется восемь возможных координат (u, v) следующего хода. Отобразим их в таблице 16.

Таблица 16 – Возможные варианты хода коня.

	1		8	
2				7
		x, y		
3				6
	4		5	

Из этой таблицы видно, что изменения координат в соответствии с номером хода будут следующими (таблица 17):

Таблица 17 – Массивы по вариантам хода коня.

k	Δx	Δy
1	-1	2
2	-2	1
3	-2	-1
4	-1	-2
5	1	-2
6	2	-1
7	2	1
8	1	2

Для нумерации следующего очередного хода используем индекс $k\,(1 \le k \le 8)$. Изменения Δx, Δy зададим в массивах a, b. Рекурсивная процедура вызывается в первый раз с параметрами x_0, y_0 – координатами поля начала обхода.

Дальнейшие некоторые улучшения очевидны из текста программы.

```
Program KhightsTour;
const   n = 8;   nsqr = n * n;
type
    TA = array[1..8] of shortint;
    TIndex = 1..n;
const
      a : TA = (-1,-2,-2,-1, 1, 2, 2, 1);
      b : TA = (2, 1,-1,-2,-2,-1, 1, 2);
var
   i, j : TIndex;
   q    : Boolean;
   S    : set of TIndex;
```

```
   H    : array[TIndex, TIndex] of Byte;
   C    : Longint;

procedure Try(i:Integer;x,y:TIndex;var q:Boolean);
    var
        k, u, v : Integer;
        q1      : Boolean;
 begin
  k := 0;
  repeat
    Inc(k);
    q1 := false;
    u  := x + a[k];
    v  := y + b[k];
    if (u in S) and (v in S)  then
      if H[u,v] = 0  then
        begin
         H[u,v] := i;
         Inc(C);
         if  i < nsqr then
            begin
             Try(i + 1, u, v, q1);
             if not q1 then H[u,v] := 0
            end
          else  q1 := true
         end
  until  q1  or (k = 8);
  q := q1;
  end;     {Try}
begin
 C := 0;
 S := [];
 for  i := 1  to  n  do  S  := S + [i];
 for  i := 1  to  n do
     for  j := 1  to  n  do   H[i,j] := 0;
 H[1,1] := 1;
 Try(2, 1, 1, q);
 Writeln(' Attempts: ', C:16);
 if  q  then
    for  i := 1  to  n do
        begin
         for  j := 1  to  n  do
              Write (H[i,j]:5);
         Writeln
        end
 else
    Writeln (' not equation ');
```

```
 Readln;
end.
```

Получим следующий результат.

```
Attempts:          27241112
    1   10   23   64    7    4   13   18
   24   63    8    3   12   17    6   15
    9    2   11   22    5   14   19   32
   62   25   40   43   20   31   16   51
   39   44   21   58   41   50   33   30
   26   61   42   47   36   29   52   55
   45   38   59   28   57   54   49   34
   60   27   46   37   48   35   56   53
```

Задание. В графическом режиме на шахматной доске совершить движение коня по стратегии, когда ход коня выбирается не детерминировано, а случайно (из восьми позиций).

Задача о восьми ферзях

Требуется так расставить 8 ферзей на шахматной *доске* так, чтобы ни один ферзь не угрожал *другим.*

Возможны следующие варианты условия этой задачи:

- отыскать один вариант решения;
- отыскать все варианты решения.

Используя решение предыдущей задачи в качестве образца, мы легко получим первоначальную версию алгоритма.

```
procedure  Try(i:Integer);
begin
 инициировать выбор позиции для i-го ферзя;
 repeat
 выбрать позицию из списка очередных ходов;
 if  разрешенная then
   begin поставить ферзя;
    if  i<8  then
        begin
         try(i + 1);
         if неудовлетворительна then убрать ферзя
        end
   end
 until (ход был удачный)∨( нет других возможных ходов)
end;
```

Чтобы идти дальше, надо выбрать какое-то представление для данных.

Вспомним, что ферзь бьет все фигуры, расположенные на той же вертикали, горизонтали и диагоналях (левой и правой).

Отсюда имеем, что каждая вертикаль может содержать одного и только одного ферзя, так что *i*-го ферзя и будем помещать на *i*-ю вертикаль.

Таким образом, выбор позиции ограничивается 8 возможными значениями индекса горизонтали *j*.

Как же представить расположение ферзей на доске?

Нам потребуется информация, стоит ли ферзь на данной горизонтали и двух соответствующих диагоналях, чтобы из этого заключить, можно ли сюда поставить ферзя. А это булевское значение.

Пусть $x[i]$ показывает позицию ферзя на *i*-й вертикали;

$a[j] = true$ показывает, что на j-й горизонтали нет ферзя;

$b[k] = true$ показывает, что на k-й диагонали нет ферзя;

$c[k] = true$ показывает, что на k-й диагонали нет ферзя.

Выбор индексных границ в массивах b, c определяется так: на диагонали, которая параллельна побочной диагонали, все поля имеют одну и ту же сумму координат i, j. Значит, у индексов диапазон будет от 2 до 16. На диагонали, которая параллельна главной диагонали, все поля имеют одну и ту же разность координат i, j. Значит, у индексов диапазон будет от -7 до 7.

Опишем теперь переменные.

```
var   x : array[ 1..8 ]  of  Integer;
      a : array[ 1..8 ]  of  Boolean;
      b : array[ 2..16]  of  Boolean;
      c : array[-7..7 ]  of  Boolean;
```

При таких данных оператор «поставить ферзя» программируется так:

```
     x[i]    := j;        a[j]     := false;
     b[j + i]:= false;    c[i - j] := false;
```

Оператор «снять ферзя» программируется так:

```
      a[j] := true; b[j + i]:= true; c[i - j] := true;
```

Условие «разрешённая» программируется так:

```
 a[j] and b[j + i]  and  c[j - i] = true
```

Теперь напишем программу, которая разыскивает единственное решение и прекращает работу.

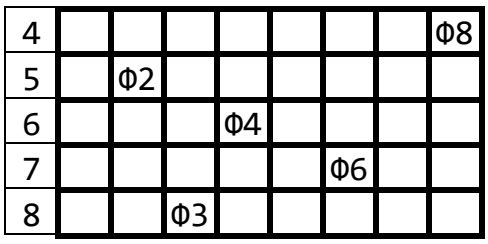
```
Program EightQueens1;
const
     n = 8;
var
     x : array[1..n]      of  shortint;
     a : array[1..n]      of  Boolean;
     b : array[2..2 * n]  of  Boolean;
     c : array[-n+1..n-1] of  Boolean;
     i : Integer;
     l : Integer;
     q : Boolean;

procedure Try(i:Integer;  var q : Boolean);
```

```
  var   j : Integer;
  begin
   j := 0;
   repeat
   Inc(j); q := false;
   if  a[j]  and  b[j + i]  and  c[i - j]  then
     begin
      x[i]   := j;
      a[j]   := false;
      b[i+j] := false;
      c[i-j] := false;
      if  i<n  then
         begin
          Try(i+1,q);
          if  not q  then
             begin
              a[j]   := true;
              b[i+j] := true;
              c[i-j] := true;
              Inc(l);
             end
         end
      else  q := true
      end
   until  q  or (j = n);
  end;     {Try}

begin
 l := 0;
 for   i := 1  to  n    do   a[i] := true;
 for   i := 2  to  2*n  do   b[i] := true;
 for   i :=-n+1 to n-1  do   c[i] := true;
 Try(1, q);
 Writeln('l = ',l);
 if  q  then
    for  i := 1  to  n do
         Writeln (i:4, x[i]:5)
 else
     Writeln (' not equation ');
 Readln;
end.
```

Получим следующую версию размещения:

	1	2	3	4	5	6	7	8
1	Ф1							
2							Ф7	
3					Ф5			

```
l = 105
   1    1
   2    7
   3    5
   4    8
   5    2
   6    4
   7    6
   8    3
```



Обобщением этой задачи является задача, которая ищет все решения.

К этому обобщению приводит следующий алгоритм.

```
procedure  Try(i:Integer);
var  j : Integer;
begin
 for  j := 1  to  n  do
    begin    выбор j-го пути;
     if  приемлимо  then
        begin     записать ход
         if i<n then try(i+1) else печатаем решение
         стираем запись хода
        end
    end
end;
```

Выделим процедуру распечатки текущего варианта решения задачи и получим следующую программу:

```
Program EightQueen2;
const
     n = 8;
type
     TMas = array[1..n] of shortint;
var
    x : TMas;
    a : array[1..n]        of  Boolean;
    b : array[2..2*n]      of  Boolean;
    c : array[-n+1..n-1]   of  Boolean;
    w, l, i : Integer;

procedure Print_Try (x: TMas);
  var  k : Integer;
  begin
   for  k := 1  to  n  do  Write(x[k]:4);
   Writeln(L:6);
```

```
   L := 0;
   Inc(w);
  end;

procedure Try(i:Integer);
  var
     j : Integer;
  begin
   for  j:=1  to  n  do
      begin
       Inc(L);
       if  a[j]  and  b[j+i]  and  c[i-j]  then
          begin
           x[i]   := j;
           a[j]   := false;
           b[i+j] := false;
           c[i-j] := false;
           if  i<n then  Try(i+1)
                   else  Print_try(x);
           a[j]   := true;
           b[i+j] := true;
           c[i-j] := true;
          end
      end;
    end;     {Try}
begin
 l := 0;
 for  i := 1    to  n     do  a[i] := true;
 for  i := 2    to  2*n   do  b[i] := true;
 for  i := -n+1 to  n-1   do  c[i] := true;
 W := 0;
 Writeln(' x1  x2  x3  x4  x5  x6  x7  x8  l: ');
 Try(1);
 Writeln('varians = ', w);
 Readln;
end.
```

Последняя программа находит все 92 варианта решения задачи, но только 12 из них принципиально разные, а остальные получаются поворотами и зеркальными отражениями. Число l указывает частоту проверок безопасности полей. Часть результатов приведена ниже.

```
x1  x2  x3  x4  x5  x6  x7  x8    L:
1   5   8   6   3   7   2   4   876
1   6   8   3   7   4   2   5   264
1   7   4   6   8   2   5   3   200
1   7   5   8   2   4   6   3   136
2   4   6   8   3   1   7   5   504
```

```
   2   5   7   1   3   8   6   4   400
   2   5   7   4   1   8   6   3    72
   2   6   1   7   4   8   3   5   280
        ........
   8   4   1   3   6   2   7   5   264
variants = 92
```

Задача о восьми ладьях

По аналогии с задачей о ферзях можно рассмотреть задачу о ладьях (турах).

Требуется так расставить 8 ладей на шахматной доске, чтобы ни одна ладья не угрожала другим.

Обсудим математический смысл поставленной задачи.

Понятно, что в каждой горизонтали шахматной доски может стоять только одна ладья.

Далее каждой ладье присвоим номер вертикали, в которой она стоит. Можно начать с ситуации $(1, 2, 3, 4, 5, 6, 7, 8)$.

Очевидно, что нужно получить все перестановки из 8 чисел, а их будет $8! = 40320$, эту задачу мы умеем решать при помощи рекурсивной подпрограммы.

Задача про устойчивые браки

Еще одной интересной задачей этого раздела является следующая **Задача**.

Заданы два непересекающихся множества A и B с одинаковыми мощностями n. Нужно найти некоторые множество n пар $< a, b >$, такое, что a in A, b in B удовлетворяют некоторым ограничениям.

Для выбора таких пар существует множество различных критериев, один из них называется «правилом устойчивых браков».

Предположим, что A – множество мужчин, а B – множество женщин. Каждый мужчина и каждая женщина устанавливают определенный набор наиболее желательных для себя возможных партнеров по браку. Если n пар выбрано таким образом, что существуют какие-то мужчина и женщина, которые не состоят в браке, но отдают предпочтение один другому, а не своим существующим мужу (жене), то такое множество браков считается *неустойчивым*. Если же такой пары не существует, то множество называется *устойчивым*.

Эта ситуация типична для многих подобных задач, в которых распределение зависит от каких-либо предпочтений. Пример с браками выбран интуитивно.

Решение можно искать следующим образом: пытаться последовательно объединять в пары члены двух множеств, пока оба множества не будут исчерпаны. Если нужно найти все устойчивые распределения, мы можем легко получить схему программы, опираясь на предыдущие разработки.

Задание 1. Указать маршрут коня, который начинается на одном заданном поле шахматной доски и заканчивается на другом. Никакое поле не должно встречаться в маршруте дважды.

Задание 2. Найти такую расстановку пяти ферзей на шахматной доске, при которой каждое поле будет находиться под ударом одного из них.

Задание 3. Найти такую расстановку двенадцати коней на шахматной доске, при которой каждое поле будет находиться под ударом одного из них.

Задание 4. Найти такую расстановку восьми ладей на шахматной доске, при которой каждое поле будет находиться под ударом одной из них.

Задание 5**.** Найдите все способы обхода конем шахматной доски, которые совмещались бы друг с другом при повороте доски на 90°.

Задание 6**.** Найдите все способы обхода конем шахматной доски, которые совмещались бы друг с другом зеркальным отражением.

Поиск с возвратом и локальный поиск

Существует целый класс задач, которые попадают в категорию теоретически разрешимых, но практически неосуществимых. Нет особого различия между программой, которая не заканчивает своей работы никогда, и программой, которая работает годами. Этот класс – класс переборных задач. Приведем пример такой задачи.

Задача о «рюкзаке»

Задача. *Существует набор грузов известного веса и машина известной грузоподъемности. Требуется определить, можно ли полностью загрузить машину, поместив в нее некоторые из грузов.*

Теоретическое решение этой задачи очевидно. Если у нас есть n грузов, то возможных комбинаций этих грузов будет 2^n. Перебрав все комбинации, можно дать положительный или отрицательный ответ. Оценим, однако, количество операций, необходимых для выполнения этого алгоритма. Например, когда грузов $n=100$, то $2^n = 2^{100} = (2^{10})^{10} = 1024^{10} \approx 1000^{10} = 10^{30}$, требуется надо перебрать примерно 10^{30} вариантов. Сколько же времени будет работать машина, выполняя этот полный перебор? Пусть мы имеем компьютер, который выполняет миллиард (10^9) операций в секунду. На проверку одного набора идет никак не меньше одной операции. Значит, требуется не менее $10^{30}/10^9 = 10^{21}$ секунд $> 10^{13}$ лет (!). Очевидно, что с любой практической точки зрения этот алгоритм непригоден.

Единственный выход – это всеми возможными способами уменьшить количество вариантов перебора.

Следует отметить, что существует ряд задач, которые называются *полными* и которые требуют выполнения перебора всех возможных ситуаций.

Примеры полных задач

Задача. (*Задача о «раскраске графа»*.) Задан граф – набор из нескольких вершин, некоторые из которых соединены линиями (дугами), и число h. Требуется определить, можно ли так раскрасить вершины графа в h цветов, чтобы любые две вершины, соединенные одной дугой, были разукрашены в разные цвета.

Задача. (*Задача о «разбиении»*.) Задано множество S и некоторое семейство его подмножеств. Требуется узнать, можно ли из этого семейства подмножеств выбрать несколько множеств, которые попарно не пересекаются, объединение которых есть S.

Для этих трех задач показано, что если одна из них может быть быстро решена, то и другая – тоже. Но дальше показано, что быстрого алгоритма не существует и нужно перебирать все варианты. Значит, у задач такого рода надо уметь отвергать так называемые тупиковые варианты.

Имея задачу, сначала нужно решить, к какому классу она принадлежит – полных задач или нет. Далее нужно или искать быстрый алгоритм, или выполнять полный перебор с улучшением, который называется *поиск с возвратом*.

Те задачи, к которым подходит идея «разделяй и властвуй» – сводят задачу к нескольким подзадачам того же типа, но меньшего размера – имеют быстрые алгоритмы. Обычно это рекурсивный подход. Однако существуют и нерекурсивные алгоритмы с возвратом.

Поиск с возвратом

Остановимся более подробно на задачах перебора, при решении которых можно реализовать нерекурсивные алгоритмы с возвратом.

Рассмотрим некоторый массив a_1, ..., a_n. Он обладает 2^n подмассивами: пустым, массивами по одному элементу, по два и, вообще, подмассивами a_i, a_j, ..., a_p.

Задача. *В заданном массиве натуральных чисел* $a_1, \ldots, a_n$ *нужно выбрать подмассив* $a_i, a_j, \ldots, a_p$ *такой, что* $a_i + a_j + \ldots + a_p = z$, *где* z *– заданное число.*

Эта задача принадлежит к полным задач, поэтому она решается перебором всех возможных ситуаций. При ее решении возникнет ряд подзадач, которые нам понадобятся и в дальнейшем.

Надо научиться фиксировать определенные подмассивы и потом проверять их элементы на удовлетворение поставленному условию.

Во многих случаях нужно решать задачу выбором из заданного массива одного или всех подмассивов, которые удовлетворяют определенному условию.

Для этого заведем вспомогательный массив $b_1, \ldots, b_n$ из нулей и единиц, в котором будем хранить такую информацию: если $b_i = 0$, тогда a_i не включается в подмассив, и если $b_j = 1$, тогда a_j включается в подмассив.

Например, при $n = 6$ массив $b = (0, 1, 0, 1, 1, 0)$ зафиксирует подмассив a_2, a_4, a_5, массив $b = (0, 0, 0, 0, 0, 0)$ зафиксирует пустой подмассив, а массив $b = (1, 1, 1, 1, 1, 1)$ зафиксирует исходный массив.

Поэтому далее мы будем работать не с условием $a_i + a_j + \ldots + a_p = z$, а с условием $z = \sum_{i=1}^{n} b_i a_i$.

Рассмотрим, как можно перебрать все 2^n подмассивов для решения поставленной задачи. Для этого будем строить массив $b_1, \ldots, b_n$ из нулей и единиц в порядке, который называется «словарным».

Если рассматривать массивы как слова в алфавите из двух цифр 0 и 1, считая, что 0 предшествует 1, то именно в таком порядке массивы попадут в словарь.

При n=3 получим подмассивы:

(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1).

Продумали, как строить подмножество. Теперь рассмотрим, как сократить перебор.

Чтобы определить n-элементный подмассив b (из нулей и единиц), который фиксирует текущий подмассив массива $a_1, \ldots, a_n$, надо о каждом элементе a_i $(1 \le i \le n)$ решить, принимается он в подмассив, или нет? Может возникнуть следующая ситуация: относительно элементов $a_1, \ldots, a_k$, $(k < n)$ приняты некоторые решения, но после этого оказалось, что как бы мы не распоряжались остальными элементами, нам не удастся получить подмассив, который даст решение поставленной задачи. Это так называемые *тупиковые* ситуации.

В таком случае можно исключить из рассмотрения все подмассивы, первые элементы которых выбраны из числа $a_1, ..., a_k$ $(k < n)$ в соответствии с принятыми ранее относительно их решениями.

Например, в условиях нашей задачи известно, что все $a_1, \ldots, a_n$ – положительные. Тогда когда зафиксированы начальные элементы $a_i, a_j, \ldots, a_r$ этого массива и их сумма уже больше чем z, то попытка подобрать к ним еще несколько составляющих из элементов $a_{r+1}, \ldots, a_n$, которые остались, будет бессмысленной (ведь от этого сумма только увеличится).

Это наводит на мысль, что в массиве b нужно распознавать тупиковые подмассивы и отбрасывать их без дальнейшего просмотра.

Для этого рассмотрим более подробно построение вспомогательного массива b. Поскольку мы пользуемся словарным методом его построения, то нужно в нем хранить и более короткие слова (как в обычном словаре, где

встречаются и слова на одну, две и более букв). Тогда нам легче будет отказаться от «удлинения» подмассива $a_1, ..., a_k$ $(k < n)$, когда мы попали в тупиковую ситуацию.

Будем считать, что нам известен алгоритм `Impasse`, позволяющий распознавать тупиковые ситуации и возвращать значение $t = 1$, если подмассив $a_1, ..., a_k$ $(k < n)$ тупиковый, и $t = 0$, если нет. Используем алгоритм `Impasse` для как можно большего уменьшения вариантов перебора.

Проанализируем задачу построения вспомогательного массива b.

В общем случае массив b будет содержать все возможные как односимвольные, так двух, трех и т. д. символьные «слова». Короткие массивы нас интересуют в том смысле, что к ним как раз и нужно применять алгоритм распознавания тупиков. Для $n = 3$ получим следующую последовательность слов:

0; 0, 0; 0, 0, 0; 0, 0, 1; 0, 1; 0, 1, 0; 0, 1, 1; 1; 1, 0; 1, 0, 0; 1, 0, 1; 1, 1; 1, 1, 0; 1, 1, 1.

Анализируя эту последовательность, получим следующий алгоритм построения вспомогательного массива b:

1) $b_1 = 0,\ k := 1$;

2) если массив $b_1, \ldots, b_k$ не тупиковый, тогда продлеваем его, дописывая справа нулевые элементы;

3) если зашли в тупик или получили массив длины n, тогда надо рассматривать массив, который:

а) не является удлинением рассматриваемого массива;

б) расположен в словаре дальше, чем рассматриваемый;

в) из всех массивов, удовлетворяющих а), б), встречается в словаре первым.

Такой массив, когда он существует, может быть построен из $b_1, \ldots, b_k$ просмотром элементов в обратном порядке, т. е. в порядке b_k, $b_{k-1}, \ldots$ до первого элемента $b_m = 0$; тогда берем $b_m = 1,\ k = m$. После этого группа элементов $b_1, \ldots, b_k$ удовлетворяет условиям 1) – 3).

Если получим массив $b_1 = b_2 = \cdots = b_n = 1$, тогда перебор закончен.

Напишем процедуру построения очередного варианта массива b.

```
procedure  Build_b;
begin
 while  B[k] = 1   do
      begin
       k := k - 1;
       if  k = 0  then  exit
      end;
 B[k] := 1;
end;
```

Процедура `Build_b` изменяет переменную k и, возможно, некоторые элементы $b_1, \ldots, b_k$, значит, или строит новую группу $b_1, \ldots, b_k$ для дальнейшего просмотра, или при $k = 0$ показывает, что новую группу построить нельзя.

Таким образом, вместе с добавлением новых нулей при движении слева направо время от времени возникает обратный просмотр элементов массива b. Это так называемый *бектрекинг* (backtracking) – *обратный просмотр*.

Перебор массивов при помощи бектрекинга может выполняться в зависимости от цели перебора:

- или до исчерпания всех массивов (найти все варианты решения задачи),
- или до возникновения первого массива, который удовлетворяет условию задачи (найти хотя бы один вариант).

Для нашей задачи возможны такие ситуации:

- $if \ \sum_{i=1}^{k} b_i a_i < z \quad \rightarrow$ продлеваем массив b;
- $if \ \sum_{i=1}^{k} b_i a_i > z \quad \rightarrow$ тупик, бектрекинг;
- $if \ \sum_{i=1}^{k} b_i a_i = z \quad \rightarrow$ решение.

Получим следующую программу.

```
Program Backtracking;
const
     n = 10;
type
     U = array[1..n] of Integer;
Var
     B       : U;
     t,k,i,Z : Integer;
     flag    : Boolean;
const
      A : U  = (10,2,3,4,5,6,4,8,7,5);

procedure Impasse;{Тупик}
  var
      i, S : Integer;
  begin
   S := 0;
   for i := 1 to  k  do
       S := S + A[i] * B[i];
      if  Z = S  then   t := 0  { нашли решение}
      else
          if (k<n)  and  (S<Z)  then
             t := 2      { не достигли решения }
```

```
          else   t := 1; { зашли в тупик       }
                         { нужен бектрекинг    }
  end;

procedure  Build_b;
  begin
   while  B[k] = 1   do
        begin
         k := k - 1;
         if  k = 0  then  exit
        end;
   B[k] := 1;
  end;

procedure  Happy_end;
  var
      i : Integer;
  begin
   flag   := false;
   for  i := 1  to  k  do
    if B[i]=1 then Write('a[',i, ']=',a[i], ' ');
   Writeln;
   t := 1;
  end;

begin
Write('  Z - ? ');
 Readln(Z);
 k    := 1;
 B[k] := 0;
 flag := true;
 t    := 2;
 while k > 0  do
   case t  of
    0 : Happy_end;
    1 : begin  Build_B;  Impasse;  end;
    2 : begin  k := k + 1;   B[k] := 0; Impasse;  end;
   end;
 if flag  then  Writeln(' no ');
 Readln; end.
```

Недостатком этой подпрограммы является пересчёт S при каждом новом обращении к процедуре `Impasse`. Мы можем указать много таких ситуаций, когда пересчёт суммы вообще не нужен (например, $(0,\ 1) \to (0,\ 1,\ 0)$), а в других ситуациях к сумме нужно добавить только одну составляющую (например, $(0,\ 1,\ 0) \to (0,\ 1,\ 1)$).

Чтобы исправить этот недостаток, заведем целочисленный массив $S_0, \ldots, S_n$, в котором $S_k = S_{k-1} + b_k a_k, \quad S_0 = 0.$

При увеличении k будем постепенно получать следующие суммы, при уменьшении k вернемся к предыдущим суммам.

Сделаем очевидные правки в процедурах `Impasse, Happy_end` и в главной программе.

Если нужно получить хотя бы одно решение, тогда после печати очередного варианта можно остановить выполнение программы. Если же требуется найти все варианты, тогда после печати очередного варианта нужно вызвать процедуру `Build_B`.

Получим следующий улучшенный вариант решения поставленной задачи.

```
Program Backtracking_Opt;
{$R+}
const
      n = 10;
type
     U = array[1..n] of Integer;
     V = array[0..n] of Integer;
var
   B         : U;
   S         : V;
   t,k,i,Z,L : Integer;
   flag      : Boolean;
   f         : text;
const
      A : U  = (10, 2, 3, 4, 5, 6, 4, 8, 7, 5);

procedure Impasse;
  var
      i : Integer;
  begin
   if  Z = S[k]  then   t := 0
   else
      if (k < n) and (S[k] < Z)  then   t := 2
      else   t := 1;
  end;

procedure  Build_b;
  begin
   while  B[k] = 1   do
         begin
         k := k - 1;
         if  k = 0  then  exit
         end;
   B[k] := 1;
```

```
  end;

procedure  Happy_end;
  var
      i : Integer;
  begin
   L    := L + 1;
   flag := false;
   for  i := 1  to  k  do
       if B[i] = 1 then
          Write(f, 'a[',i, '] = ',a[i], '  ');
   Writeln(f);
   {Halt;}
   Build_B;
  end;

begin
 L := 1;
 Assign(f, 'backtr.dat');
 ReWrite(f);
 Write('  Z - ? ');
 Readln(Z);
 k    := 1;
 B[k] := 0;
 flag := true;
 t    := 2;
 S[0] := 0;
 while k > 0  do
     begin
      S[k] := S[k - 1] + A[k] * B[k];
      Impasse;
      case t  of
      0 : Happy_end;
      1 : Build_B;
      2 : begin
          k    := k + 1;
          B[k] := 0;
          Impasse;
          end;
      end;
     end;
 Writeln('L = ',L);
 if flag  then  Writeln(' no ');
 Close(f);
 Readln;
end.
```

При $Z = 20$ получим следующие варианты (печатаются элементы заданного массива, которые нужно взять, чтобы накопить заданную сумму Z):

Z = 20

```
a[8] = 8  a[9] = 7  a[10] = 5
a[5] = 5  a[8] = 8  a[9] = 7
a[5] = 5  a[6] = 6  a[7] = 4  a[10] = 5
a[4] = 4  a[7] = 4  a[9] = 7  a[10] = 5
a[4] = 4  a[5] = 5  a[7] = 4  a[9] = 7
a[4] = 4  a[5] = 5  a[6] = 6  a[10] = 5
a[3] = 3  a[7] = 4  a[8] = 8  a[10] = 5
a[3] = 3  a[6] = 6  a[7] = 4  a[9] = 7
a[3] = 3  a[5] = 5  a[9] = 7  a[10] = 5
a[3] = 3  a[5] = 5  a[7] = 4  a[8] = 8
a[3] = 3  a[4] = 4  a[8] = 8  a[10] = 5
a[3] = 3  a[4] = 4  a[6] = 6  a[9] = 7
a[3] = 3  a[4] = 4  a[5] = 5  a[8] = 8
a[2] = 2  a[6] = 6  a[9] = 7  a[10] = 5
a[2] = 2  a[6] = 6  a[7] = 4  a[8] = 8
a[2] = 2  a[5] = 5  a[8] = 8  a[10] = 5
a[2] = 2  a[5] = 5  a[6] = 6  a[9] = 7
a[2] = 2  a[4] = 4  a[6] = 6  a[8] = 8
a[2] = 2  a[4] = 4  a[5] = 5  a[7] = 4  a[10] = 5
a[2] = 2  a[3] = 3  a[8] = 8  a[9] = 7
a[2] = 2  a[3] = 3  a[6] = 6  a[7] = 4  a[10] = 5
a[2] = 2  a[3] = 3  a[5] = 5  a[6] = 6  a[7] = 4
a[2] = 2  a[3] = 3  a[4] = 4  a[7] = 4  a[9] = 7
a[2] = 2  a[3] = 3  a[4] = 4  a[6] = 6  a[10] = 5
a[2] = 2  a[3] = 3  a[4] = 4  a[5] = 5  a[6] = 6
a[1] = 10  a[6] = 6  a[7] = 4
a[1] = 10  a[5] = 5  a[10] = 5
a[1] = 10  a[4] = 4  a[6] = 6
a[1] = 10  a[3] = 3  a[9] = 7
a[1] = 10  a[2] = 2  a[8] = 8
a[1] = 10  a[2] = 2  a[4] = 4  a[7] = 4
a[1] = 10  a[2] = 2  a[3] = 3  a[10] = 5
a[1] = 10  a[2] = 2  a[3] = 3  a[5] = 5
```

Мы построили программу, которая обрабатывает неотрицательные исходные данные и поэтому анализирует короткие массивы на тупиковые ситуации. Если же исходные данные могут быть и отрицательными числами, то очевидно, что массив $b_1,..., b_k$ нужно достраивать до конца и только тогда делать возврат (*бектрекинг*). Рассмотрим такую задачу:

Задача. *В заданном массиве целых чисел* $a_1,\ldots, a_n$ *нужно выбрать подмассив* $a_i, a_j,\ldots, a_p$ *такой, что* $a_i + a_j + \ldots + a_p = z$, *где* z – *заданное число*.

Небольшие изменения внесем в процедуру `Impasse` и в главную программу и получим решение поставленной задачи для целых чисел.

```
Program Bt_full;
{$R+}
const
      n = 10;
type
     U = array[1..n]  of Integer;
     V = array[0..n]  of Integer;
var
     B          : U;
     S          : V;
     f          : text;
     t,k,i,Z,L  : Integer;
     flag       : Boolean;
const
      A : U  = (10,-2,3,4,-5,6,4,-8,7,5);

procedure Variant;
  var
      i : Integer;
  begin
   if  Z = S[k] then   t := 0
   else
      if  k<n   then   t := 2
      else   t := 1;
  end;

procedure  Build_b;
  begin
   while  B[k] = 1   do
       begin
         k   := k - 1;
         if  k = 0  then  exit
       end;
   B[k] := 1;
  end;

procedure  Happy_end;
  var
      i : Integer;
  begin
   L    := L + 1;
   flag := false;
   for  i := 1  to  k  do
     if B[i]=1 then Write(f,'a[',i,']=',a[i],'  ');
```

```
   Writeln(f);
   Build_B;
  end;

begin
 L:=1;
 Assign(f, 'backtr.dat');
 ReWrite(f);
 Write('  Z - ?');
 Readln(Z);
 Writeln(f, 'z=',z);
 k := n;
 for  i:= 1  to  n  do
    begin
     S[i] := 0;
     B[i] := 0;
    end;
 B[k] := 1;
 flag := true;
 t    := 2;
 S[0] := 0;
 while  k > 0  do
      begin
       S[k] := S[k-1] + A[k] * B[k];
       Variant;
       case t  of
         0 :  Happy_end;
         1 : Build_B;
         2 : begin
             k := k + 1;
             B[k] := 0;
             Variant;
             end;
        end;
       end;
 Writeln('L = ',L);
 if flag  then  Writeln(' no ');
 Readln;
 Close(f);
end.
```

Программа напечатает следующий результат:

```
Z = 20
a[4]=4  a[7]=4  a[9]=7  a[10]=5
a[3]=3  a[6]=6  a[7]=4  a[9]=7
a[3]=3  a[5]=-5  a[6]=6  a[7]=4  a[9]=7  a[10]=5
```

```
a[3]=3  a[4]=4  a[6]=6  a[9]=7
a[3]=3  a[4]=4  a[5]=-5  a[6]=6  a[9]=7  a[10]=5
a[2]=-2  a[6]=6  a[7]=4  a[9]=7  a[10]=5
a[2]=-2  a[4]=4  a[6]=6  a[9]=7  a[10]=5
a[2]=-2  a[3]=3  a[4]=4  a[6]=6  a[7]=4  a[10]=5
a[1]=10  a[6]=6  a[8]=-8  a[9]=7  a[10]=5
a[1]=10  a[6]=6  a[7]=4
a[1]=10  a[5]=-5  a[6]=6  a[7]=4  a[10]=5
a[1]=10  a[4]=4  a[6]=6
a[1]=10  a[4]=4  a[5]=-5  a[7]=4  a[9]=7
a[1]=10  a[4]=4  a[5]=-5  a[6]=6  a[10]=5
a[1]=10  a[3]=3  a[9]=7
a[1]=10  a[3]=3  a[6]=6  a[7]=4  a[8]=-8  a[10]=5
a[1]=10  a[3]=3  a[5]=-5  a[9]=7  a[10]=5
a[1]=10  a[3]=3  a[4]=4  a[7]=4  a[8]=-8  a[9]=7
a[1]=10  a[3]=3  a[4]=4  a[6]=6  a[8]=-8  a[10]=5
a[1]=10 a[3]=3 a[4]=4 a[5]=-5 a[7]=4 a[8]=-8 a[9]=7 a[10]=5
a[1]=10 a[2]=-2 a[9]=7 a[10]=5
a[1]=10 a[2]=-2 a[5]=-5 a[6]=6 a[7]=4 a[9]=7
a[1]=10 a[2]=-2 a[4]=4 a[7]=4 a[8]=-8 a[9]=7 a[10]=5
a[1]=10  a[2]=-2  a[4]=4  a[5]=-5  a[6]=6  a[9]=7
a[1]=10  a[2]=-2  a[3]=3  a[7]=4  a[10]=5
a[1]=10 a[2]=-2 a[3]=3 a[6]=6 a[7]=4 a[8]=-8 a[9]=7
a[1]=10 a[2]=-2 a[3]=3 a[5]=-5 a[6]=6 a[7]=4 a[8]=-8 a[9]=7 a[10]=5
a[1]=10 a[2]=-2 a[3]=3 a[4]=4 a[10]=5
a[1]=10 a[2]=-2 a[3]=3 a[4]=4 a[6]=6 a[8]=-8 a[9]=7
a[1]=10 a[2]=-2 a[3]=3 a[4]=4 a[5]=-5 a[6]=6 a[8]=-8  a[9]=7  a[10]=5
a[1]=10 a[2]=-2 a[3]=3 a[4]=4 a[5]=-5 a[6]=6 a[7]=4
L=32
```

Фракталы

В настоящее время есть несколько определений термина «фрактал»:

- фрактал – это геометрическая фигура, в которой один и тот же фрагмент повторяется при каждом уменьшении масштаба. Это конструктивные фракталы и получаются они в результате рекурсивной процедуры, объединяющей сжимающие отображения подобия;
- фракталом называется структура, состоящая из частей, в каком-то смысле подобных целому;
- фрактал – это такое множество, которое имеет фрактальную размерность, большую топологической.

В первом определении слово «фрактал» происходит от латинского «fractus», означающее изломанный. Во втором и третьем определении оно связано с английским «fractional» – дробный.

Первые фракталы появились в математике в конце XIX начале XX века. К их числу относятся такие удивительные конструкции, как канторовское множество (Г. Кантор, 1883), или, например, заметающая целый квадрат, кривая

Гильберта-Пеано (Д. Гильберт – Дж. Пеано, 1890), множества Серпинского (В.Серпинский, 1916) и другие типичные фракталы (термин "фрактал" в то время еще не был введен). Эти конструкции были в свое время открыты математиками для того, чтобы показать, насколько наивными и хрупкими могут быть наши представления о столь знакомых, казалось, объектах, как функция и кривая. Функции, которые не являются достаточно гладкими или регулярными, часто игнорировались как не стоящие изучения.

Одним из первых описал динамические фракталы в 1918 году французский математик Гастон Жюлиа в своем объемном труде. Однако в нем отсутствовали изображения и только по истечении длительного времени компьютеры сделали видимыми ажурные множества. Фрактальные картины впечатляют? и появились термины: «компьютерное искусство», «художественный дизайн», «эстетический хаос».

Наука о фракталах оформилась в отдельную область математики в начале 70-х годов XX века. Началом этого процесса принято считать появление в 1977 году книги Б. Мандельброта "Фрактальная геометрия природы", в которой содержится огромное количество изображений различных фрактальных множеств и приведены доказательства существования фрактальных объектов в природе.

После выхода книги Б. Мандельброта "Фрактальная геометрия природы" началась настоящая "фрактальная лихорадка". Многим удалось по-новому взглянуть на объекты своих исследований, и оказалось, что они долгие годы изучают фракталы. Одна за другой стали появляться научные работы, где сообщалось о нахождении фрактальных объектов. Исследовались поверхности разломов твердых образцов, процессы агрегации кластеров и адсорбции, форма облаков и облачных зон над поверхностью Земли, шероховатость минералов, динамика экономических процессов, рост биологических популяций, волны в океане. В геологии и картографии, в физике и биологии – везде были обнаружены фракталы.

Теория фракталов стала междисциплинарной. Интерес к исследованию процессов, обуславливающих фрактальную геометрию природы, привел к рождению новых научных направлений в физике (фрактальная физика), биологии, материаловедении и т. д. Такое объединение различных научных направлений является следствием универсальных свойств фрактальных структур.

Многие крупные достижения науки о фракталах стали возможны только с использованием методов вычислительной математики.

Компьютерные просчеты позволили получить достаточно полное представление о разнообразных фрактальных структурах. Компьютерная графика способна воссоздать на экране монитора бесконечное разнообразие фрактальных форм и пейзажей при помощи сравнительно простых алгоритмов.

В настоящее время фракталы используются для сжатия изображений, которое состоит в нахождении подобных областей и сохранении в файле только коэффициентов преобразований подобия. Преобразования подобия – это сдвиг, отражение, поворот и изменение яркости. Фракталы используются при анализе и

классификации сигналов сложной формы, возникающих в разных областях, например при анализе колебаний курса валют в экономике. Они применяются в физике твердого тела, в динамике активных сред и т. д.

Классификация фракталов

Существуют разные классификации фракталов.

По одной классификации фракталы делятся на группы. Самые большие группы это:

- алгебраические фракталы;
- геометрические фракталы;
- системы итерируемых функций;
- стохастические фракталы.

По другой классификации фракталы делятся на:

- детерминированные:
  - алгебраические,
  - геометрические;
- недетерминированные:
  - стохастические.

По третьей классификации фракталы делятся на:

- динамические:
  - алгебраические;
- конструктивные:
  - системы итерируемых функций,
  - геометрические,
  - стохастические.

Алгебраические фракталы

Большая группа фракталов – алгебраические. Свое название они получили за то, что их строят на основе алгебраических формул, иногда очень простых.

Классическими примерами алгебраических фракталов являются множества Жюлиа, Мандельброта, ньютоновские и модели магнетизма.

Методов получения алгебраических фракталов несколько. Один из методов – это многократный (итерационный) подсчет функции $z_{n+1} = f(z_n)$, где z – комплексное число, а f – некоторая функция. Подсчет данной функции продолжается до выполнения определенного условия завершения. И если это условие выполняется, на экран выводится точка соответствующего цвета. При этом значение функции для разных точек комплексной плоскости может меняться следующим образом:

1. С течением времени стремится к бесконечности.

2. С течением времени стремится к 0 или какому-то другому конечному числу.

3. Принимает конечные фиксированные значения и не выходит за их пределы.

4. Поведение хаотическое, без каких-либо тенденций.

В центре внимания оказалась природа границ между такими разными областями. Здесь можно представить себе центры – аттракторы (точки притяжения), которые ведут борьбу за влияние на плоскости.

Любая начальная точка z_0 или в течение процесса приходит к тому или иному центру, либо лежит на границе и не может принять определенное решение. Со сменой параметра меняются и области, принадлежащие аттракторам, а вместе с ними и границы. Граница, которая разделяет области притяжения аттракторов, называется *множеством Жюлиа*. Может случиться, что граница превращается как бы в пыль.

Б. Мандельброт исследовал предельные поведение последовательности комплексных чисел (при $k \to \infty$)

$$z_{k+1} = z_k^2 + c,\ k = 0,\ 1,\ 2,\ \ldots,\quad z_0 = c, \qquad (1)$$

при различных значениях комплексных чисел *c*.

Будем говорить, что если точка z_k вышла за пределы круга заданного радиуса $r_{\min}$, то расхождение достигнуто за $k < k_{\max}$ итераций и процесс итерирования прерывается. При $|z_k| \le r_{\min}$ и до $k = k_{\max}$ присутствует сходимость.

Последовательность z_k с ростом числа итераций демонстрирует поведение двух типов в зависимости от выбора начальной точки *c*. Элементы последовательности или постепенно уходят в бесконечность, или всегда остаются в определенной замкнутой области, совершая циклическое движение, или стремясь в конечную точку. Математиками строго доказано, что когда при некотором k модуль $|z_k| > r_{\min}$, где $r_{\min} = 2$ – минимальный радиус расходимости множества Мандельброта, то дальше последовательность расходится и $\lim\limits_{k\to\infty} |z_k| = \infty$.

Множество точек *c*, для которых эта последовательность *сходится,* называется множеством Мандельброта.

Очень трудно доказывается, что это множество связное. Однако, чем ближе к границе множества находится точка, тем больше итераций необходимо выполнить, чтобы узнать, стремится ли точка в бесконечность, или нет.

Графическая интерпретация фрактальных множеств удивительно красивая и бесконечно разнообразная. Устанавливаются простые соотношения между цветом и временем $k \le k_{\max}$, за которое точка (x_k, y_k) уходит на бесконечность или приближается к другому аттрактору.

Множество Мандельброта

Итерационный процесс

$$z_{k+1} = z_k^2 + c,\ k = 0,\ 1,\ 2,\ \ldots,\quad z_0 = c,$$

при различных значениях комплексных чисел c можно выполнять как с комплексной арифметикой, так и с арифметикой для действительных чисел (x_k, y_k), т. е. представляя $z_k = x_k + iy_k$, $c = c_x + ic_y$, легко получить:

$$x_{k+1} = x_k^2 - y_k^2 + c_x,\ y_{k+1} = 2x_k y_k + c_y,\ k = 0,\ 1,\ 2, \ldots,\ x_0 = c_x,\ y_0 = c_y.$$

Множество Мандельброта получается так: выбирается фиксированная точка (x, y) и просчитывается ее путь при разных значениях параметров (c_x, c_y). Результаты наносятся, точка за точкой, на плоскости (c_x, c_y). Таким образом получается рисунок типа множества Мандельброта.

Алгоритм

1. Задаются отрезки изменения действительной части c_x и мнимой части c_y комплексной переменной c.

2. На экране формируется прямоугольник, параметры которого определяются разрешением экрана.

3. Подсчитывается шаг изменения переменных c_x и c_y.

4. Задается максимальное количество итераций $k_{\max}$ и радиус круга сходимости $r_{\min}$ $(r_{\min} = 2)$.

5. Образуется цикл на количество пикселей по оси Ox $(i = \overline{1, n})$.

6. Образуется цикл на количество пикселей по оси Oy $(j = \overline{1, m})$.

7. Каждому пикселю экрана сопоставляется текущая точка комплексной плоскости C (а фактически по оси Ox задается значение действительной части c_x и по оси Oy – мнимой части c_y).

8. Запускается итерационный процесс $z_{k+1} = z_k^2 + c,\ k = 0,\ 1,\ 2,\ \ldots,\ z_0 = c.$

9. Если за максимальное число итераций значение $|z_k| \le r_{\min}$, то заданная точка $c(c_x, c_y)$ принадлежит множеству Мандельброта. На экране такие точки обозначаются черным цветом. Если же на некотором шаге k итерации значение $|z_k| > r_{\min}$, то считается, что значение идет к бесконечности и цвет такого пикселя выбирается по формуле $k \bmod 256$ (или $k \bmod 16$) в зависимости от количества цветов палитры.

10. Конец цикла по оси Oy.

11. Конец цикла по оси Ox.

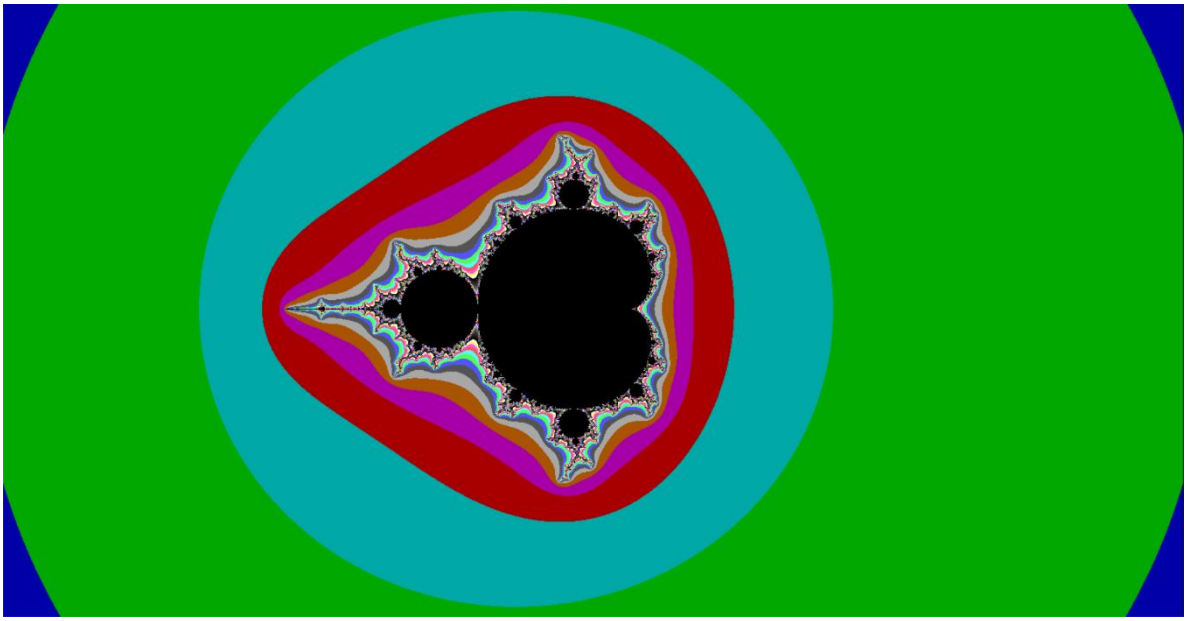
Время подсчетов можно уменьшить в 2 раза по причине симметрии процесса. Точки (x, y) и $(-x, -y)$ имеют одну судьбу.

Ниже приведен код программы построения классического множества Мандельброта с использованием комплексной арифметики.

```
Program Maldelbrot_1;
uses Graph, Crt;
```

```
type
     Complex = record
                Re, Im : Real;
               end;
const
     MaxIter = 100;
     BailOut = 16;
var
   W1, W2, Backup      : Complex;
   X, Y, Count, dr, dm : Integer;
   Max_X, Max_Y, Color : Integer;

procedure Sqr_C(a : Complex; var r : Complex);
begin
with a  do
    begin
     r.re := sqr(re) - sqr(im);
     r.im := 2 * re * im;
    end;
end;

procedure Add_C(a,b : Complex; var r : Complex);
begin
with r  do
    begin
     re := a.re + b.re;
     im := a.im + b.im;
    end;
end;

procedure Init_C(a, b  : Real; var r : Complex);
begin
with r  do
    begin
     re := a;
     im := b;
    end;
end;

begin
 dr:=Detect;
 InitGraph(dr,dm, '');
 Max_X := GetMaxX  div  2;
 Max_Y := GetMaxY  div  2;
 for Y := - Max_Y  to  0  do
    for X := -Max_X   to  Max_X    do
       begin
```

```
        Count := 0;
        Init_C(X / 200, Y / 200, Backup);
        Init_C(0, 0, W1);
        while (Sqr(W1.re)+Sqr(W1.im)<BailOut) and
                    (Count < MaxIter)  do
              begin
               W2 := W1;
               Sqr_C(W2, W1);
               Add_C(W1, Backup, W2);
               W1 :=W2;
               Inc(Count);
              end;
        if count<>MaxIter then Color:=Count mod 256)
        else Color :=0;
            PutPixel(Max_X+X, Max_Y+Y,Color);
            PutPixel(Max_X+X,Max_Y+Abs(Y),Color);
      end;
 Readln;
end.
```

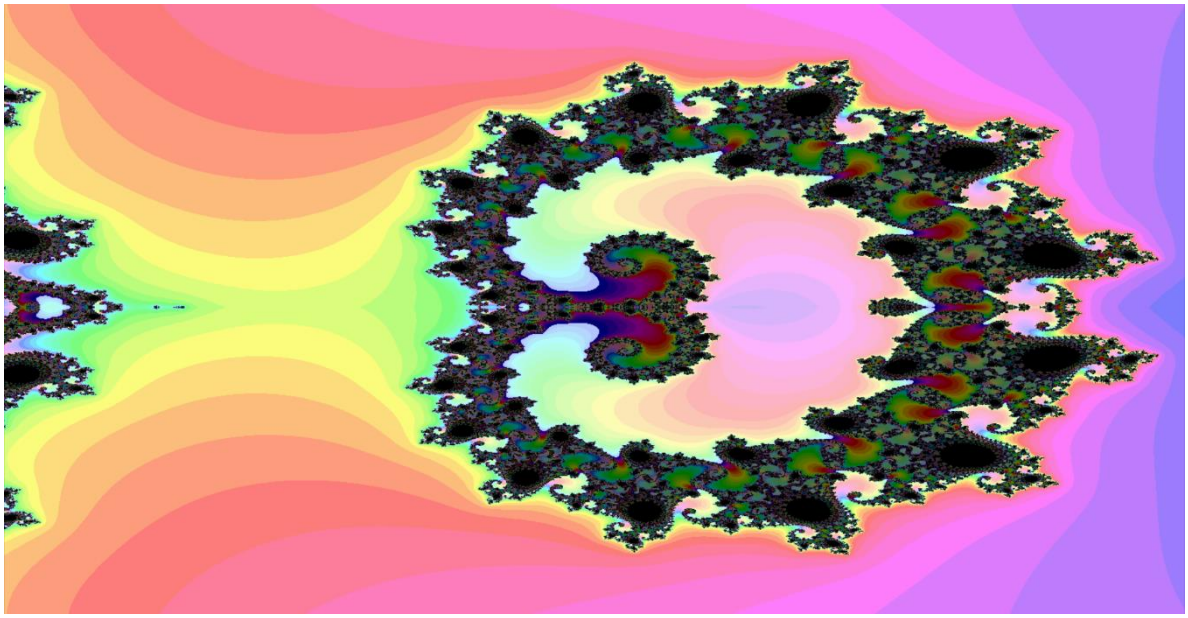


Ниже приведен код программы построения множества Мандельброта с использованием вещественной арифметики.

```
Program Maldelbrot_2;
uses  Graph, Crt;
var
   gd, gm : Integer;

function f(x, y, p : Real) : Real;
  begin
   f := x * x – y * y + p;
  end;
```

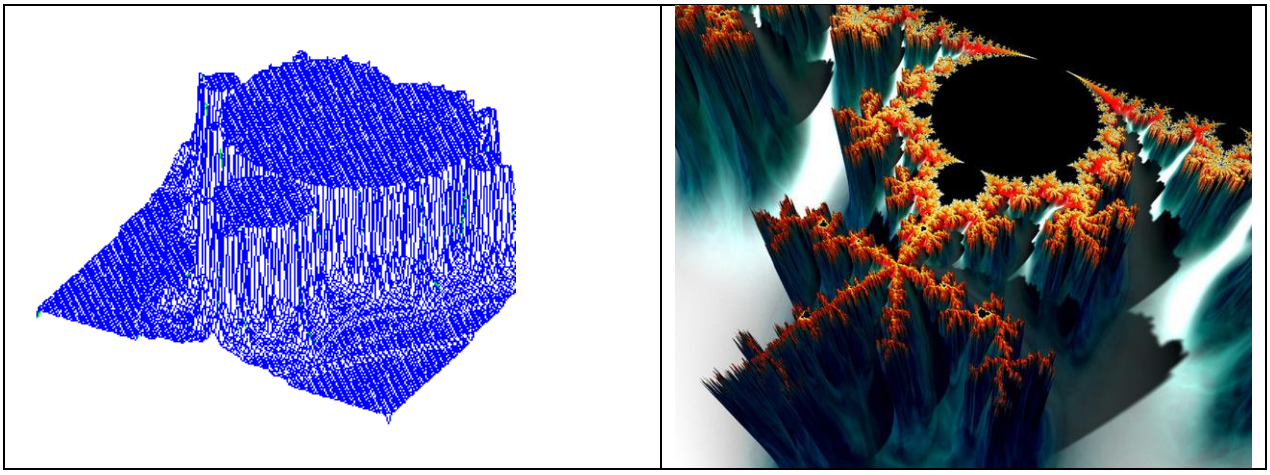
```
function g(x, y, q : Real) : Real;
  begin
   g := 2 * x * y + q;
  end;

procedure Mandelbrot(P_min, P_max, Q_min, Q_max,
                     L : Real; MaxIter : Integer);
var
     Max_x, Max_y   : Integer;
     r, t, xk, yk   : Real;
     p, q, dp, dq   : Real;
     color, i, j, k : Integer;
     flag           : Boolean;
begin
 gd := detect;
 InitGraph(gd, gm, '');
 max_x := GetMaxX;
 max_y := GetMaxY;
 dp := (p_max - p_min) / max_x;
 dq := (q_max - q_min) / max_y;
 p  := p_min;
for i := 0  to  max_x  do
   begin
    q := q_min;
    for j := 0  to  max_y div 2  do
       begin
        k    := 0;
        flag := true;
        xk   := p;
        yk   := q;
        while flag  and  (k<=maxiter)  do
             begin
              Inc(k);
              t    := xk;
              xk   := f(xk, yk, p);
              yk   := g(t , yk, q);
              r    := xk * xk + yk * yk;
              flag := r < L;
             end;
         if  flag  then color := 0
                   else  color := k mod 256;
         PutPixel(i, j, color);
         PutPixel(i, max_y - j, color);
         q := q + dq;
        end;
    p := p + dp;
   end;
```

```
end;
begin
 Mandelbrot(-0.74591,       -0.74448,       0.11196,       0.11339,
                 225, 255);
 Readln;
end.
```



Выполните эксперименты на компьютере, изменяя значения параметров `P_min, P_max, Q_min, Q_max.`

Замечание. Для каждой точки экрана $c_{ij} \in C,\ i = \overline{1,n},\ j = \overline{1,m},$ можно запустить итерационный процесс, сформировать матрицу $M = (m_{ij}),$ элемент $m_{ij} \in [1,\ k_{\max}]$ который равен номеру итерации $k,$ на которой процесс остановлен. Полагая числа m_{ij} вершинами некоторой поверхности в точках $c_{ij},$ можно получить объёмный образ множества Мандельброта или его части, которая при специально подобранном освещении может выглядеть и как скала с плоской вершиной, и как водопад, и как горная вершина. Примеры приведены на рис. ниже.



Множество Жюлиа

Еще один известный алгоритмический фрактал получается на основе формулы (1) $z_{k+1} = z_k^2 + c,\ k = 0,\ 1,\ 2,\ \ldots,$ при ее следующей модификации. Фиксируется комплексное число, например, $c = 0{,}36 + 0{,}36i,$ а каждый пиксель экрана соответствует числу z_0, действительная и мнимая части которого пробегают все значения отрезка $[-1, 1]$.

Такое предельное поведение последовательности комплексных чисел (при $k \to \infty$) исследовал *Г. Жюлиа*.

В 1918 году Гастон Жюлиа написал подробный «мемуар» в несколько сотен страниц, который был награжден призом Французской Академии.

«*Этот труд написан на высоком уровне, но ... едва ли можно найти в нем какие-то изображения*». Работа Жюлиа игнорировалась в течение почти полувека. Компьютеры сделали видимым то, что не могло быть изображено во времена Жюлиа. Визуальные результаты превзошли все ожидания.

Множество Жюлиа получается так: плоскость (x, y) рассматривается при фиксированных значениях (c_x, c_y) и анализируется динамическая связь точки с соответствующим аттрактором. В этом случае исследуется структура областей притяжения и их границы. Граница, которая разделяет области притяжения аттракторов, называется *множеством Жюлиа.*

Фрактал состоит из бесконечного ряда островов, которые касаются друг друга попарно на оси x. Примечательно, что фрактал не является связным, а состоит из отдельных компонент, подобно точечному множеству Кантора. Фракталы такого типа обычно называют «*пылью Фату*» в честь математика Фату.

Алгоритм

1. Задаются отрезки изменения действительной части x_0 и мнимой части y_0 переменной z_0.

2. На экране формируется прямоугольник, размеры которого определяются разрешением экрана.

3. Подсчитывается шаг изменения переменных x_0 и y_0.

4. Задается максимальное количество итераций $k_{\max}$ и радиус круга сходимости к бесконечному аттрактору $r_{\max}$ $(r_{\max} \approx 100)$, $r_{\min}$ $(r_{\min} = 2)$.

5. Образуется цикл на количество пикселей по оси Ox $(i = \overline{1, n})$.

6. Образуется цикл на количество пикселей по оси Oy $(j = \overline{1, m})$.

7. Каждому пикселю экрана сопоставляется текущая точка комплексной плоскости C (а фактически по оси Ox задается значение действительной части x_0 и по оси Oy – мнимой части y_0).

8. Запускается итерационный процесс $z_{k+1} = z_k^2 + c,\ k = 0,\ 1,\ 2,\ \ldots$.

9. Если на некотором шаге $k \le k_{\max}$ итерации значение $|z_k| > r_{\max}$, то считается, что значение стремится к бесконечности, и цвет такого пикселя выбирается по формуле $k \mod 256$ (или $k \mod 16$ в зависимости от количества цветов палитры), иначе заданная точка $z_0(x_0, y_0)$ на экране обозначается черным цветом.

10. Конец цикла по оси Oy.

11. Конец цикла по оси Ox.

Аттракторов может быть и несколько, тогда надо смотреть точки сгущения.

За счет задания других отрезков изменения переменных c и z_0 вероятны теоретически бесконечно глубокие опускания внутрь этих фрактальных множеств, рисунки очень разнообразные и красивые, однако всегда на этих рисунках будут появляться важнейшие особенности фракталов – симметрия и самоподобие. Практическая глубина опускания ограничена только разрядностью процесса.

Другие фрактальные множества

Фрактальные множества можно получить для итерационных процедур общего вида $z_{k+1} = F(z_k, c),\ k = 0, 1, 2, \ldots,$ которые порождают разные рисунки. Например, если взять $F(z_k, c) = z_k^n + c$ с целым числом n, то получим симметричное фрактальное множество, которое объединяет n-1 множество Мандельброта. При $n = 7$ получается почти настоящая шести лучевая снежинка.

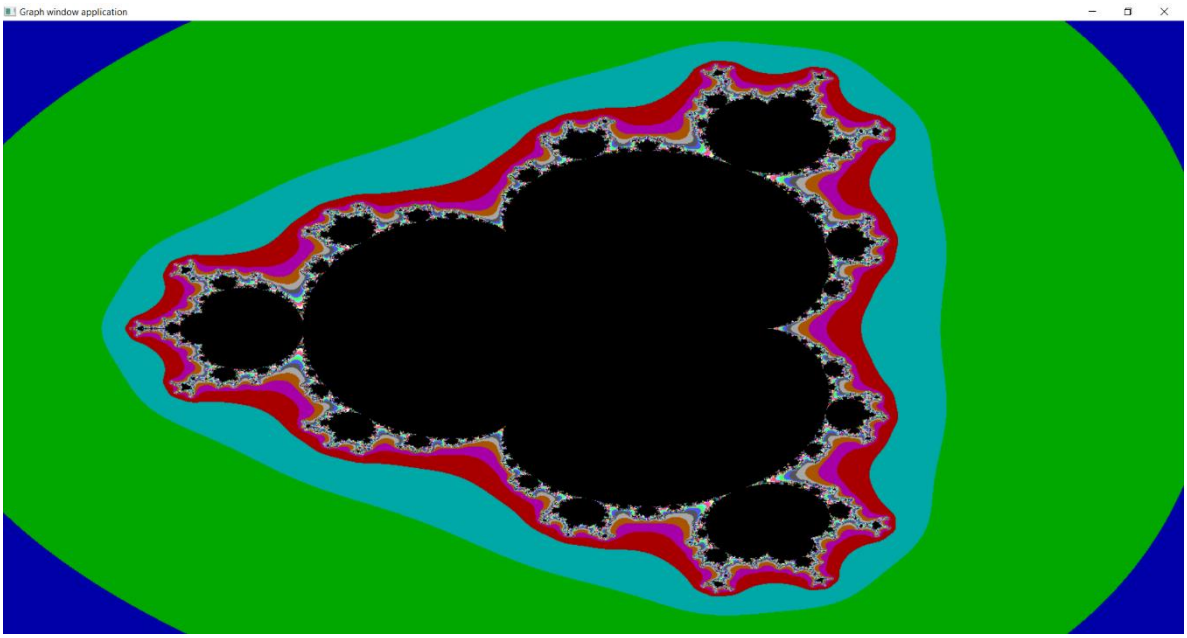
Рассмотрим сначала $n = 3, c = p + iq.$

Получим итерационный процесс:

$$\begin{cases} x_{n+1} = x_n^3 - 3x_n y_n^2 + p, \\ y_{n+1} = 3x_n^2 y_n - y_n^3 + q. \end{cases}$$

```
Program Raznoe_3;
uses  Graph, Crt;
var
   gd, gm : Integer;

function f(x, y, p : Real) : Real;
  begin
   f := x * x * x - 3 * x * y * y + p;
  end;

function g(x, y, q : Real) : Real;
  begin
   g := 3 * x * x * y - y  * y * y + q;
  end;

procedure Raznoe(P_min, P_max, Q_min,  Q_max,
```

```
                  L : Real; MaxIter : Integer);
var
     Max_x, Max_y   : Integer;
     r, t, xk, yk   : Real;
     p, q, dp, dq   : Real;
     color, i, j, k : Integer;
     flag           : Boolean;
begin
 gd := detect;
 InitGraph(gd, gm, '');
 max_x := GetMaxX;
 max_y := GetMaxY;
 dp := (p_max - p_min) / max_x;
 dq := (q_max - q_min) / max_y;
 p  := p_min;
for i := 0  to  max_x  do
   begin
    q := q_min;
    for j := 0  to  max_y div 2  do
       begin
        k    := 0;
        flag := true;
        xk   := p;
        yk   := q;
        while flag  and  (k <= maxiter)  do
             begin Inc(k);
              t    := xk;
              xk   := f(xk, yk, p);
              yk   := g(t , yk, q);
              r    := xk * xk + yk * yk;
              flag := r < L;
             end;
        if  flag  then  color := 0
                  else  color := k mod 16;
        PutPixel(i, j, color);
        PutPixel(i, max_y - j, color);
        q := q + dq;
       end;
    p := p + dp;
   end;
end;

begin
 Raznoe(-1.0, 1.0, -1.5, 1.5, 96, 100);
 Readln;
end.
```



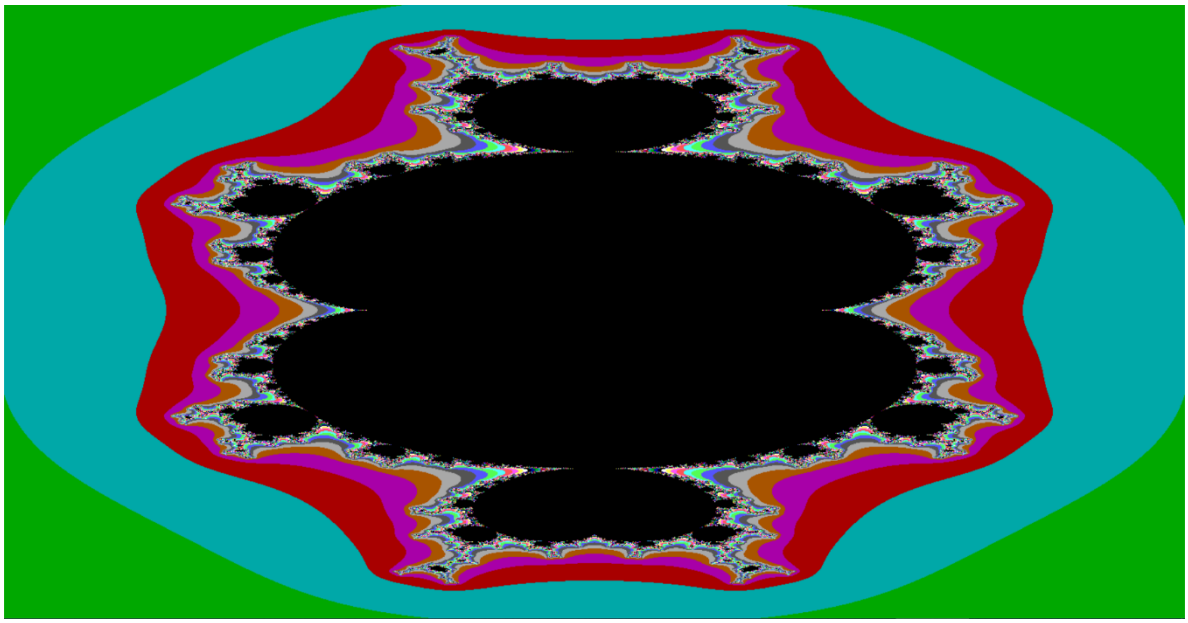
Для $n = 4$, изменяя в предыдущей программе функции:

```
function f(x, y, p : Real) : Real;
  begin
   f := x*x*x*x - 6 *x*x*y*y + y*y*y*y + p;
  end;

function g(x, y, q : Real) : Real;
  begin
   g := -4*x*y*y*y + 4 *x*x*x*y + q;
  end;
```

и вызов процедуры:

```
Raznoe(-1.6, 1.6, -1.3, 1.3, 96, 100);
```

Получим следующий рисунок:



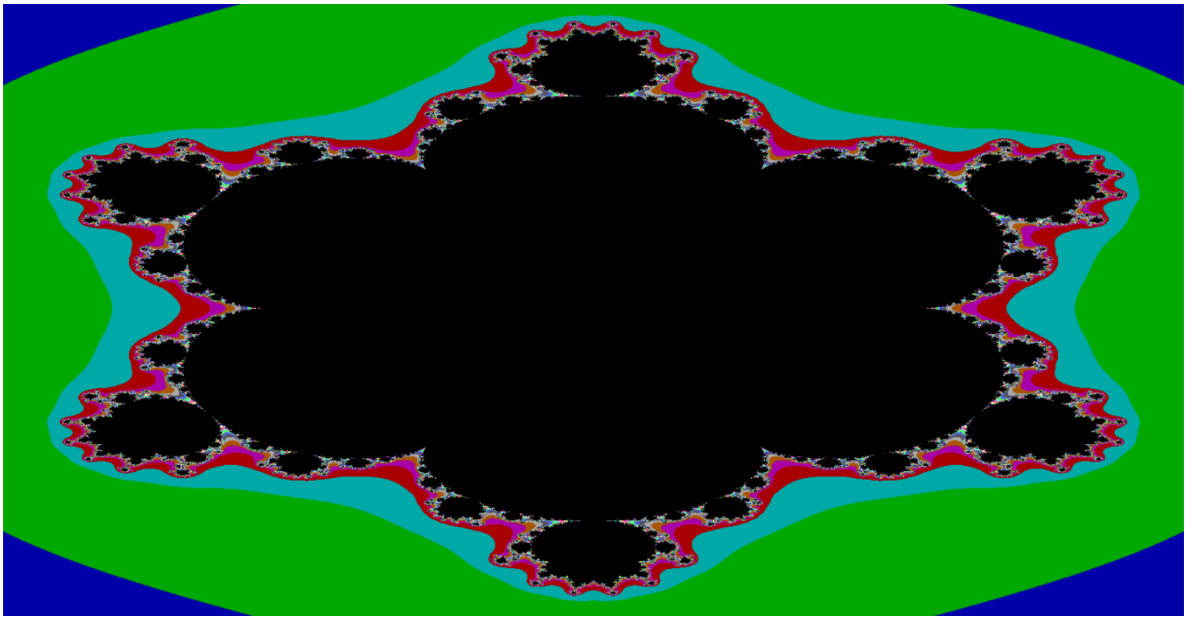
Для $n = 7$, изменяя в предыдущей программе функции:

```
function f(x, y, p : Real) : Real;
 begin
  f:=x*x*x*x*x*x*x-21*x*x*x*x*x*y*y
     +35*x*x*x*y*y*y*y-7*x*y*y*y*y*y*y*y+p;
 end;

function g(x, y, q : Real) : Real;
  begin
   g := -y*y*y*y*y*y*y+21*x*x*y*y*y*y*y
        -35*x*x*x*x*y*y*y+7*x*x*x*x*x*x*y+q;
  end;
```

и вызов процедуры:

```
    Raznoe(-1.1, 1.1, -1.2, 1.2, 96, 100);
```

получим следующий рисунок (снежинка!):



Ньютоновские фракталы

Фрактальные свойства можно выявить в самых обычных алгоритмах. Например, при расчете корней функции $f(z)$ в комплексном пространстве итерационным методом $z_{n+1} = F(z_n)$ обычно полагают, что области сходимости начальных приближений z_0 к корням функции $f(z) = 0$ имеют фиксированные и гладкие границы.

Моделирование, однако, показало, что они действительно фрактальные. Например, в качестве метода $z_{n+1} = F(z_n)$ выберем метод Ньютона $z_{n+1} = z_n - \dfrac{f(z_n)}{f'(z_n)}$ с условием завершения $|f(z_{n+1})| \le 0{,}01$, а принадлежность начальной точки z_0 к области сходимости корня $z*$ определим по минимальному расстоянию $|z_n - z*| \to \min$.

Выполним итерирование, как и в предыдущих случаях, и получим границу притяжения разных корней, которые будут аттракторами. Она будет иметь фрактальную структуру.

Модели магнетизма

Рассматриваются следующие процессы.

1. Первая модель магнетизма $z = \left(\frac{z^2 + q - 1}{2z + q - 2} \right)^2$, q – комплексный параметр.

2. Вторая модель магнетизма $z = \left(\frac{z^3 + 3(q-1)z + (q-1)(q-2)}{3z^3 + 3(q-2)z + q^2 - 3q + 3} \right)^2$, q – комплексный параметр.

Для них можно построить множество типа множеств Мандельброта – рисунок на плоскости z – для зафиксированного значения q.

В обеих моделях точки $z = 1$ и $z = \infty$ являются сверхустойчивыми аттракторами. Кроме того, могут быть еще один или два дополнительных аттрактора, ведь есть еще два критических значения $z = 0$ и $z = (1-q)^2$.

Интересно провести исследование параметров по подсчету времени, который необходим для того, чтобы эти критические точки приблизились к $z = 1$ (один основной цвет) или к $z = \infty$ (второй основной цвет). Когда q лежит в черных областях, критические точки сходятся к одному из дополнительных аттракторов.

Геометрические фракталы

Фракталы этого класса самые очевидные, они конструктивны. В двухмерном случае их получают с помощью некоторой ломаной (или поверхности в трехмерном случае), которая называется *генератором*.

За один шаг алгоритма каждый из отрезков, который составляет ломаную, заменяется на ломаную-генератор в соответствующем масштабе. В результате бесконечного повторения этой процедуры получается геометрический фрактал.

Примерами геометрических фракталов являются:

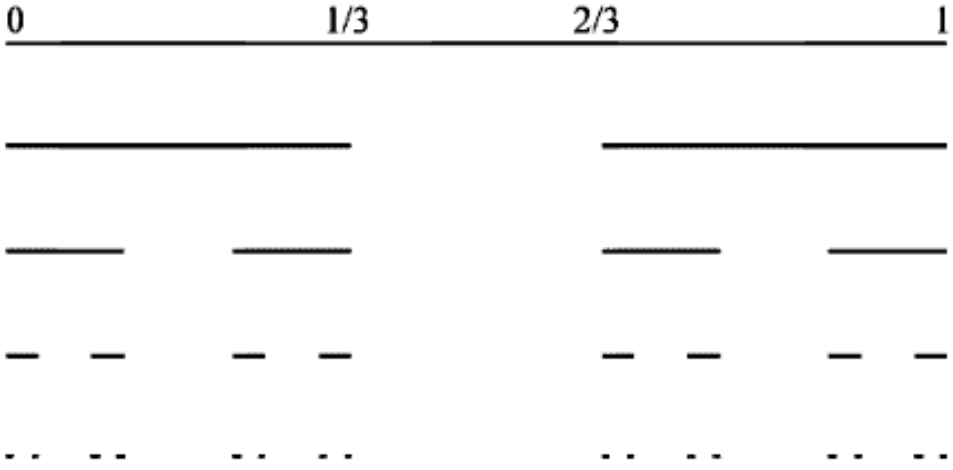
- множества Кантора (1-3D);
- H-фрактал;
- треугольник Серпинского;
- кривая Гильберта;
- кривая Серпинского;
- кривая Коха;
- кривая Леви и другие;
- дерево.

Для построения геометрических фракталов хорошо приспособлены так называемые L-Systems. Суть этих систем заключается в том, что существует набор определенных знаков системы, каждый из которых обозначает собственное действие, и набор правил выбора этих символов.

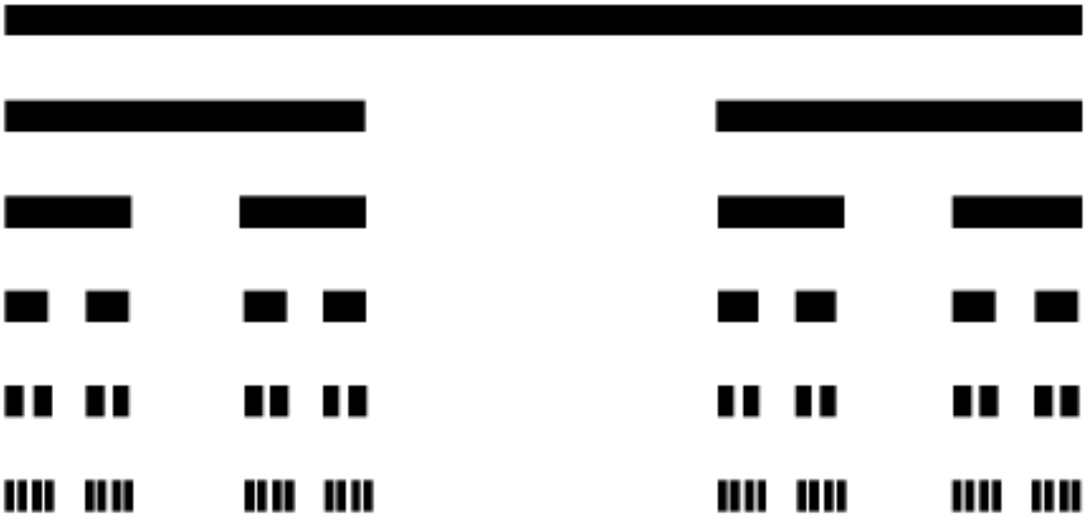
Такие фракталы хорошо программировать рекурсивными алгоритмами.

Фрактал Кантора

Кантор (1845-1918) также придумал один из старейших фракталов (1883):



Другая иллюстрация:



Третья иллюстрация – в виде гребня:

Двумерное множество Кантора. Алгоритм:

1. Построить квадрат размером *M*.
2. Вырезать расположенный в центре квадрат размером *M* / 2.
3. Поделить исходный квадрат на четыре ровные части размером *M* / 2.
4. Для каждого из четырех квадратов повторить шаги 2 и 3.

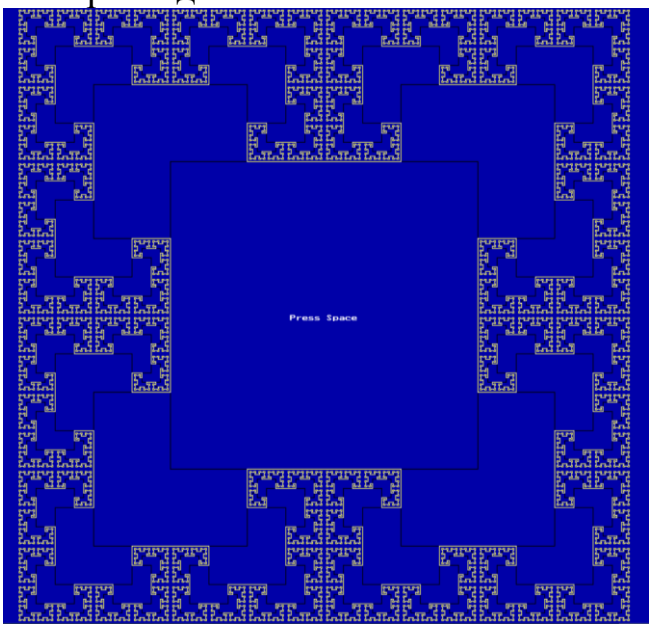
В итоге получится самоподобное множество – фрактал.

В следующей программе сохраняется образ границы вырезанного квадрата, и построение множества идет не от большего квадрата к меньшему, а наоборот.

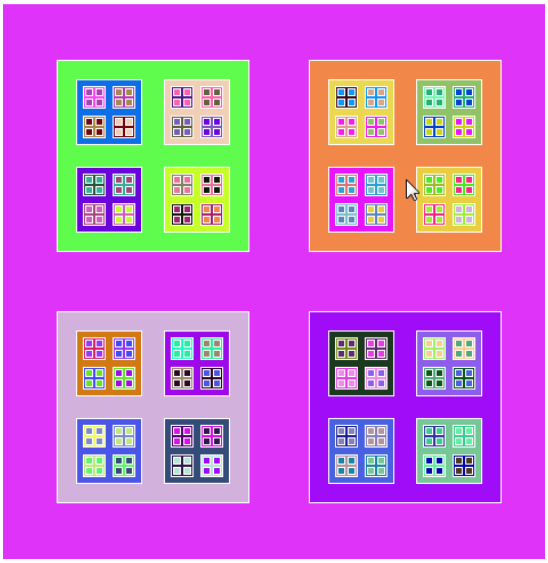
```
Program Cantor;
uses  Graph, crt;
const
      min_size = 1;
var
     Gd, Gm : Integer;
     Ch : char;

procedure draw(x,y : Integer; size : Word);
var
     S : Word;
begin
 if size > min_size  then
  begin
   S := size  div  2;
   draw(x - size,  y + size, s);
   draw(x - size,  y - size, s);
   draw(x + size,  y + size, s);
   draw(x + size,  y - size, s);
  end;
 Rectangle(x-size, y-size,x+size,y+size);
 Bar(x-size+1, y-size+1, x+size-1, y+size-1);
end;
begin
 Gd:= Detect;
 initGraph(Gd,Gm, '');
 setbkcolor(1);
 Clearviewport;
 SetfillStyle(solidfill,Black);
 Setcolor(15);
 draw(getmaxX div 2, getmaxY div 2, getmaxY div 4);
OutTextXY((getmaxX-TextWidth('Press Space')) div 2, getmaxY div 2,
'Press Space');
ch := ReadKey;
 Setcolor(0);
 SetWriteMode(XorPut);
 draw(getmaxX div 2, getmaxY div 2, getmaxY div 4);
 Setcolor(15);
OutTextXY((getmaxX-TextWidth('Press Space')) div 2, getmaxY div 2,
'Press Space');
 ch := ReadKey;
 CloseGraph;
```

```
end.
```

Итогом будет сначала картина (множество Кантора), которая после нажатия на клавишу изменится. Проследите это самостоятельно.



Другая интерпретация:



Треугольник Серпинского

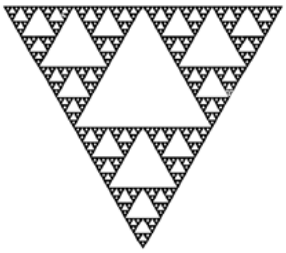
В 1915 году польский математик Вацлав Серпинский придумал занимательный объект, известный как решето Серпинского. Этот треугольник один из самых ранних известных примеров фракталов.

Строится он следующим образом. Соединяются середины сторон исходного треугольника, тем самым он разбивается на 4 равные треугольника. Удаляется центральный треугольник. К оставшимся трем, применяют те же действия и так далее.

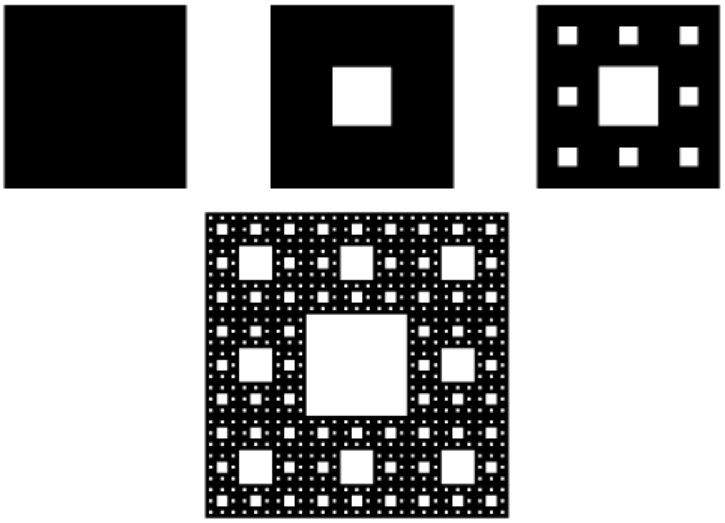
Генератор для решета Серпинского:

(*А*) (*Б*)

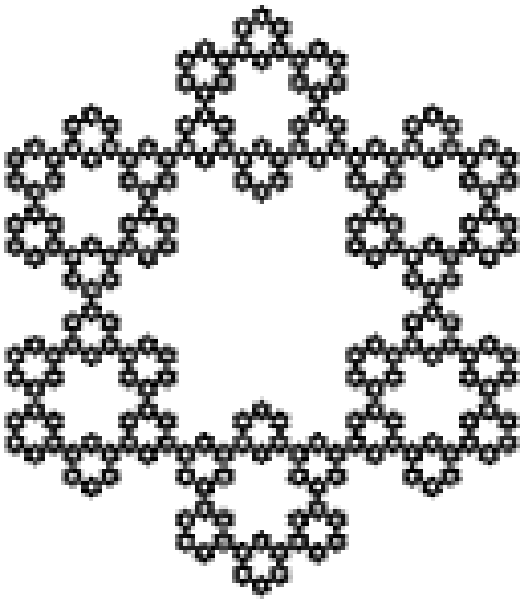
Другой вариант:



Еще один вариант ковра Серпинского. В квадрате с единичной стороной каждая из сторон делится на 3 равные части, а весь квадрат, соответственно, на 9 одинаковых квадратиков. Удаляется центральный квадрат. К оставшимся 8 квадратам применяется этот же процесс.

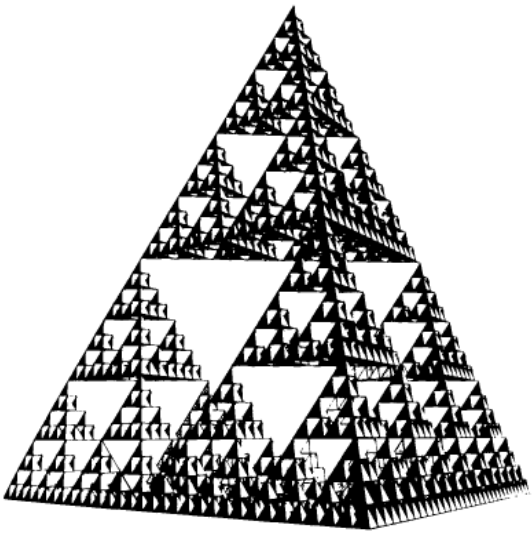


Шестиугольник Серпинского:



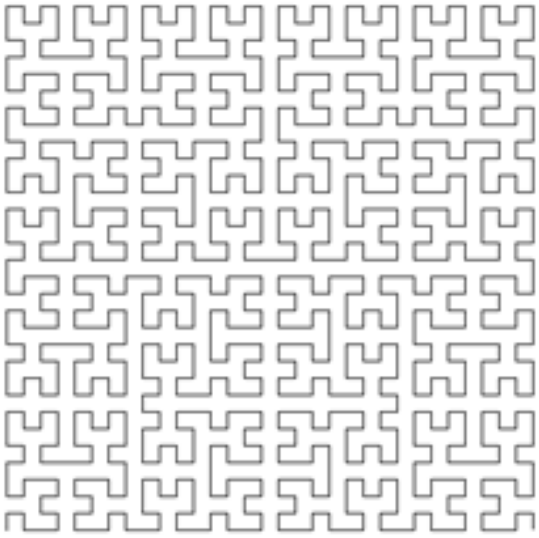
Трехмерное обобщение ковра Серпинского

Построение его начинается с правильного тетраэдра, из которого вырезается центральный перевернутый правильный тетраэдр вдвое меньших размеров. Процесс повторяется.



Кривые Гильберта

Эта кривая связана с любопытным понятием теории функций, а именно – всюду плотными кривыми. Кривая на плоскости называется всюду плотной в некоторой области, если она проходит через любую сколь угодно малую окрестность каждой точки этой области.



Известные математики Гильберт и Серпинский построили примеры всюду плотных кривых. Хотя эти примеры различны, схема получения соответствующих кривых одинакова.

По определенному правилу строятся кривые (соответственно Гильберта и Серпинского) первого, второго, ..., *n*-го порядка, вписанные в заданный квадрат.

При неограниченном возрастании *n* они стремятся к некоторой предельной кривой, которая является всюду плотной в заданном квадрате.

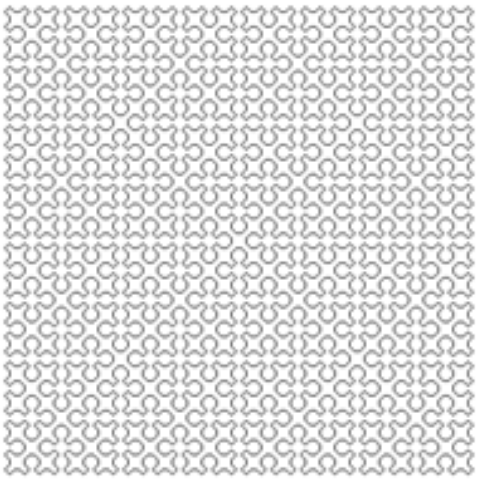
До работы Гильберта математики предполагали, что сделать это невозможно.

Разумеется, повторить алгоритм бесконечное число раз нельзя; говоря о бесконечностях, следует оперировать пределами, ε – окрестностями и прочими понятиями математического анализа.

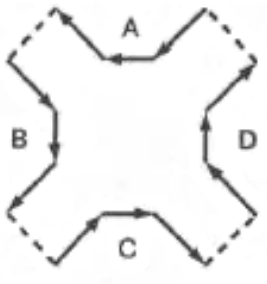
Кривые Серпинского

Подобно Гильбертовым кривым, кривые Серпинского – это самоподобные кривые, которые обычно определяются рекурсивно. Кривые Серпинского проще строить с помощью четырех отдельных процедур, работающих совместно, – SierpA, SierpB, SierpC и SierpD.

Эти процедуры косвенно рекурсивные – каждая из них вызывает другие, которые после этого вызывают первоначальную процедуру. Они выводят верхнюю, левую, нижнюю и правую части кривой Серпинского соответственно.



На рисунке показано, как эти процедуры образуют кривую глубины 1. Отрезки, составляющие кривую, изображены со стрелками, которые указывают направление их рисования. Сегменты, используемые для соединения частей, представлены пунктирными линиями.



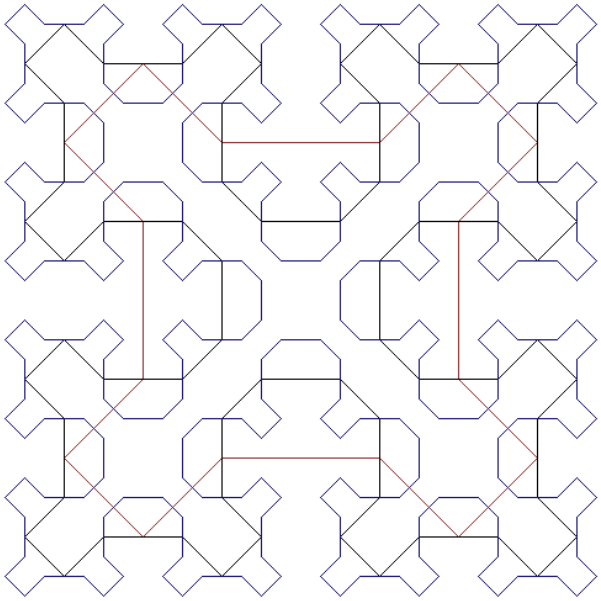
Каждая из четырех основных кривых составлена из линий диагонального сегмента, вертикального или горизонтального и еще одного диагонального сегмента.

Кривую 3-го порядка следует строить из подкривых 2-го порядка, как они показаны на рисунке выше. Рекурсивная зависимость между четырьмя типами кривых выглядит так:

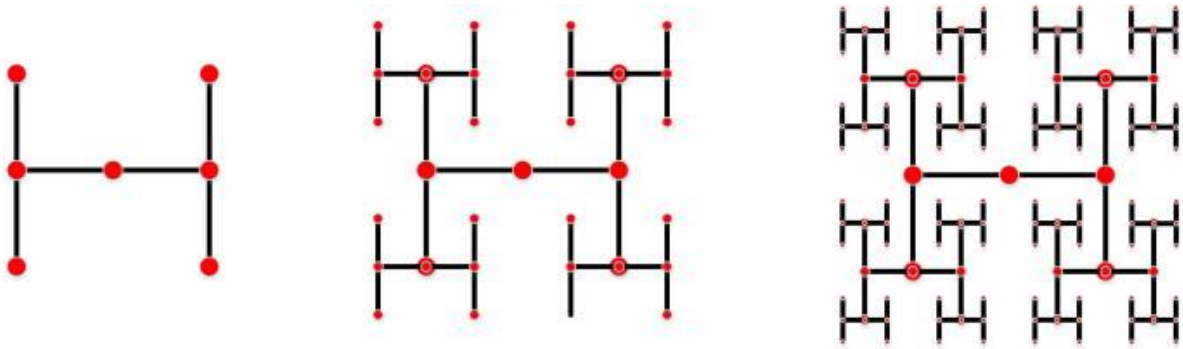
A: A` B` D` A`
B: B` C` A` B`
C: C` D` B` C`
D: D` A` C` D`

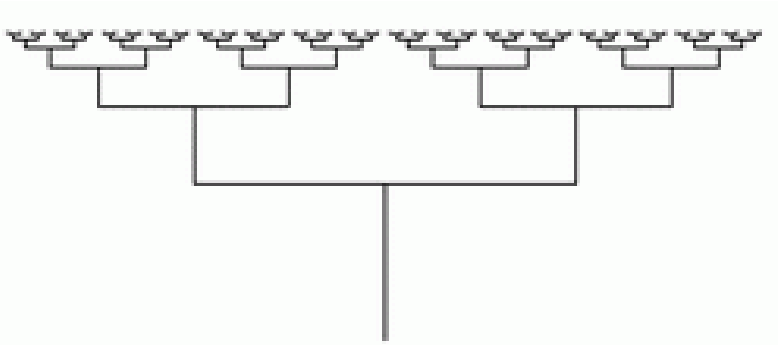
где A`, B`, C`, D` - кривые предыдущего порядка.

На рисунке приведены примеры кривых Серпинского.



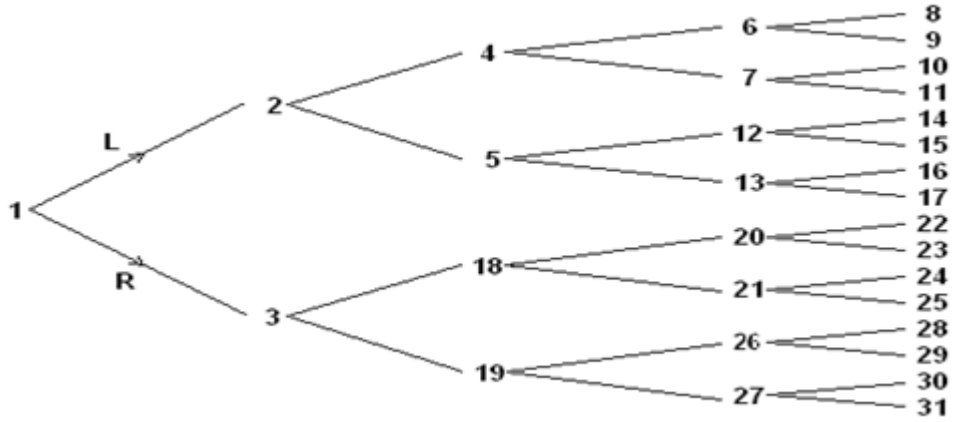
H-фрактал





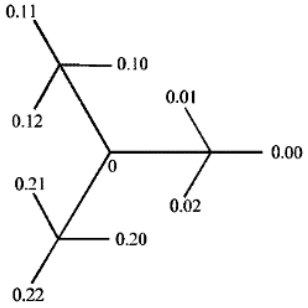
H-фрактал относится к так называемым «дендритам», от греческого «dendron» – дерево.

Дерево вызовов рекурсии - двоичное дерево

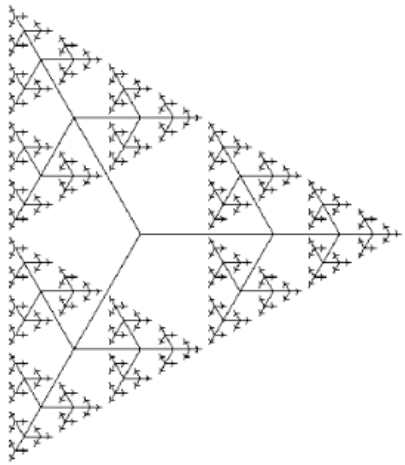


Фракталы и системы счисления

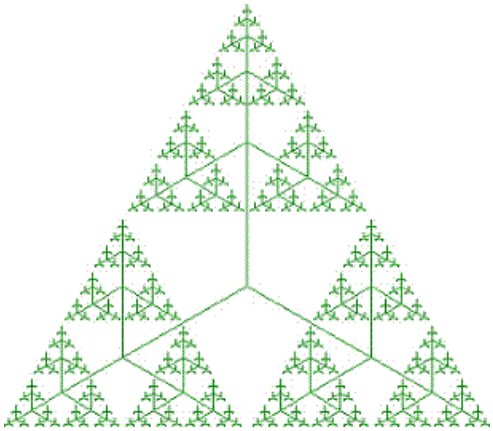
Рассмотрим дерево – дендрит, основанного на троичной системе счисления. Из одной точки под углом 120° друг к другу выходят три главные ветви. Каждый из трех концов сам является точкой, из которой выходят три более мелкие ветви, и т. д. Направление вправо мы помечаем «0». Направление влево-вверх – «1», влево-вниз – «2».



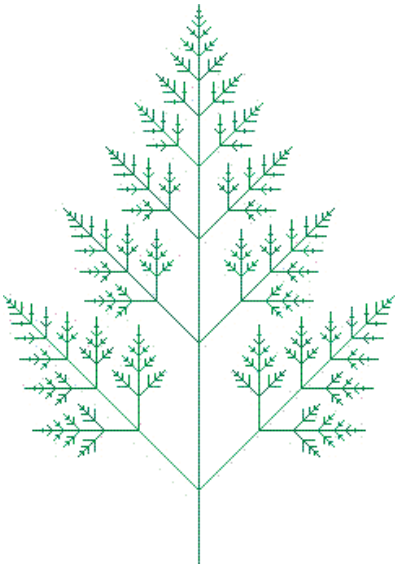
Используя данный алгоритм можно построить следующее дерево.



Или с другой ориентацией:



А можно построить и ветвь:



Системы итерируемых функций

Метод «систем итерируемых функций» (Iterated Functions System – IFS) появился в середине 1980-х гг. как простое средство получения фрактальных структур.

Метод IFS – это система функций из некоторого фиксированного класса функций, отражающих одно многомерное множества в другое. Наиболее простая IFS состоит из аффинных преобразований плоскости:

$$\begin{cases} X' = A \cdot X + B \cdot Y + C, \\ Y' = D \cdot X + E \cdot Y + F. \end{cases}$$

После задания начальной точки (x, y), если задать отображение несколькими аффинными преобразованиями, можно запустить итерационный процесс.

Для построения IFS применяют и другие классы простых геометрических преобразований – проекционные преобразования плоскости:

$$\begin{cases} X' = (A_1 \cdot X + B_1 \cdot Y + C_1) / (D_1 \cdot X + E_1 \cdot Y + F_1), \\ Y' = (A_2 \cdot X + B_2 \cdot Y + C_2) / (D_2 \cdot X + E_2 \cdot Y + F_2); \end{cases}$$

квадратичные

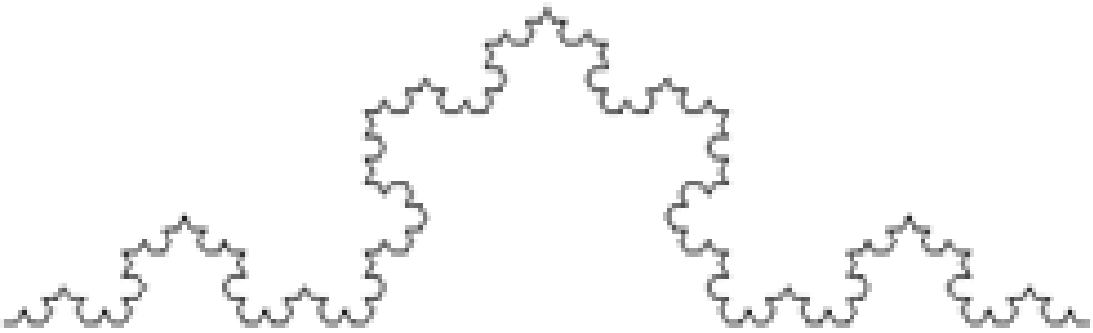
$$\begin{cases} X' = A_1 \cdot X \cdot X + B_1 \cdot X \cdot Y + C_1 \cdot Y \cdot Y + D_1 \cdot X + E_1 \cdot Y + F_1, \\ Y' = A_2 \cdot X \cdot X + B_2 \cdot X \cdot Y + C_2 \cdot Y \cdot Y + D_2 \cdot X + E_2 \cdot Y + F_2 \end{cases}$$

и другие.

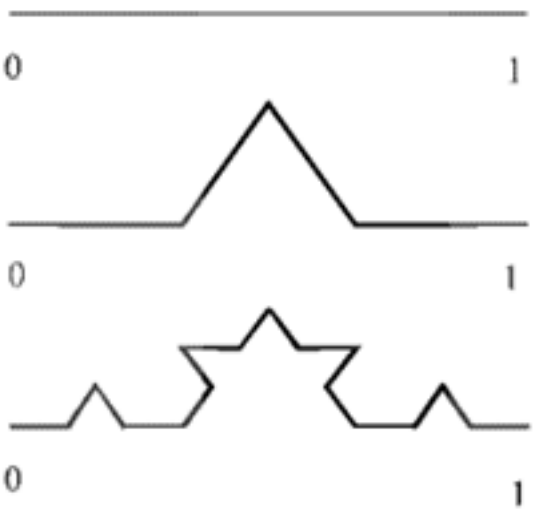
Триадная кривая Коха, дракон Хартера-Хейтуэя и лист папоротника Барнсли задаются аффинными преобразованиями и отличаются лишь матрицами преобразований.

Кривая Коха

В 1904 году математик Кох дал пример кривой, которая нигде не имеет касательной. Представьте кривую, состоящую из частей, каждая из которых бесконечной длины. Рисунок является хорошим приближением кривой Коха. Построение кривой Коха похоже на построение точек множества Кантора. Начинаем с отрезка-основы: удаляем его среднюю третью часть и заменяем ее сторонами равностороннего треугольника.



Мысленно мы можем представить кривую Коха как предел таких операций. Если основа имеет длину 1, то фрагмент будет состоять из четырех отрезков, каждый длины 1/3 и, следовательно, общей длины 4/3. На следующем шаге получаем ломаную, состоящую из 16 отрезков и имеющую общую длину 16/ 9 или (4/3) 2 и т. д.



Системы итерируемых функций:

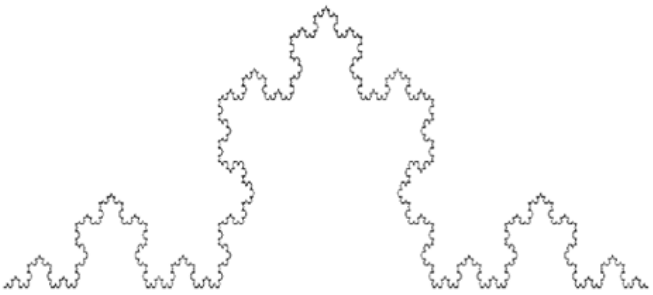
$$\begin{cases} X' = 0.333 \cdot X + 13.333 \\ Y' = 0.333 \cdot Y + 200 \end{cases}$$

$$\begin{cases} X' = 0.333 \cdot X + 413.333 \\ Y' = 0.333 \cdot Y + 200 \end{cases}$$

$$\begin{cases} X' = 0.167 \cdot X + 0.289 \cdot Y + 130 \\ Y' = -0.289 \cdot X + 0.167 \cdot Y + 256 \end{cases}$$

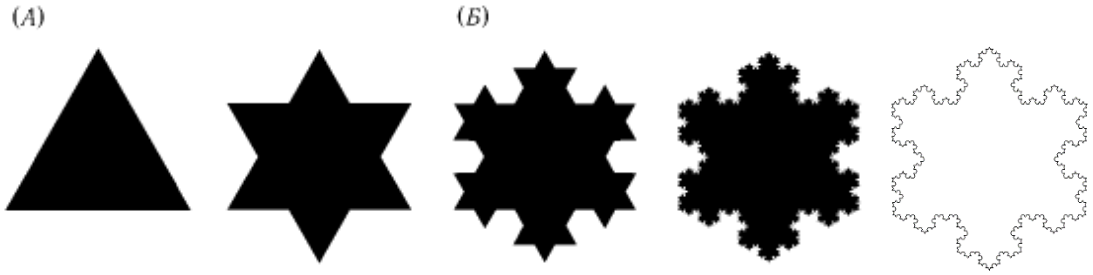
$$\begin{cases} X' = 0.167 \cdot X - 0.289 \cdot Y + 403 \\ Y' = -0.289 \cdot X + 0.167 \cdot Y + 71 \end{cases}$$

Результат применения этого аффинного преобразования после десятой итерации можно увидеть далее.

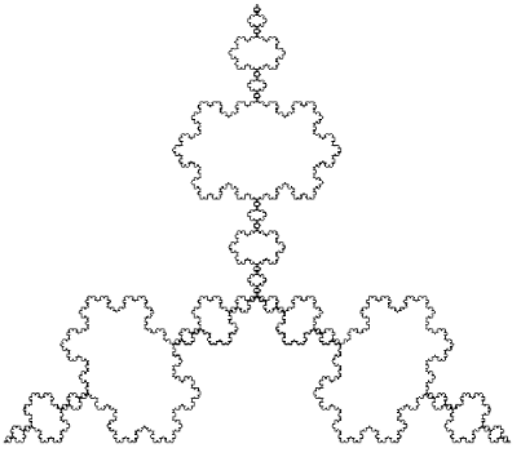


Остров Коха

Кривую Коха можно строить на сторонах правильного многоугольника (т. е. основа – правильный многоугольник). Если в качестве основы взять равносторонний треугольник, а в качестве фрагмента – фрагмент Коха, ориентированный наружу треугольника, то получим фигуру, представленную на следующем рисунке. Инициатор и генератор для снежинки (острова) Коха (А) и промежуточные стадии построения снежинки фон Коха.

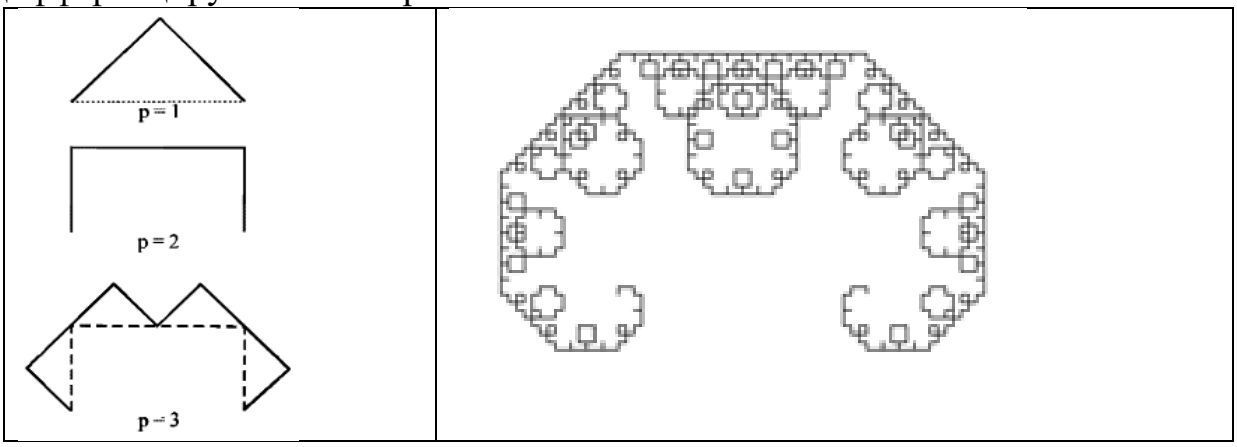


Остров Коха, ориентированный вдоль треугольника

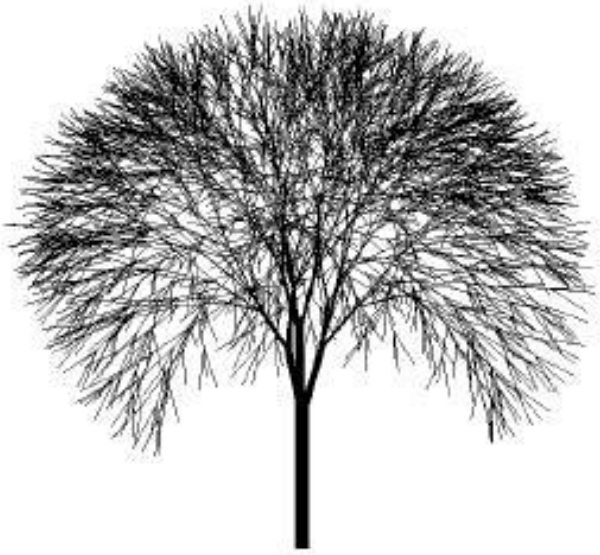


Кривая Леви

Кривая Леви – фрактал. Предложен французским математиком П. Леви (1886-1971). Получается, если заменить прямую на половину квадрата вида /\, а затем каждую сторону заменить таким же фрагментом, и, повторяя эту операцию, в пределе получим кривую Леви. Кривая Леви нигде не дифференцируема и не спрямляема.



Дерево



Построение фрактального дерева происходит просто. Берется основа – "ствол", и от его вершины рисуются несколько "веток" меньшей длины, которые располагаются под некоторым углом к стволу.

Далее каждая ветка рассматривается как отдельный ствол, к которому пририсовываются пропорционально уменьшенные ветки под теми же углами.

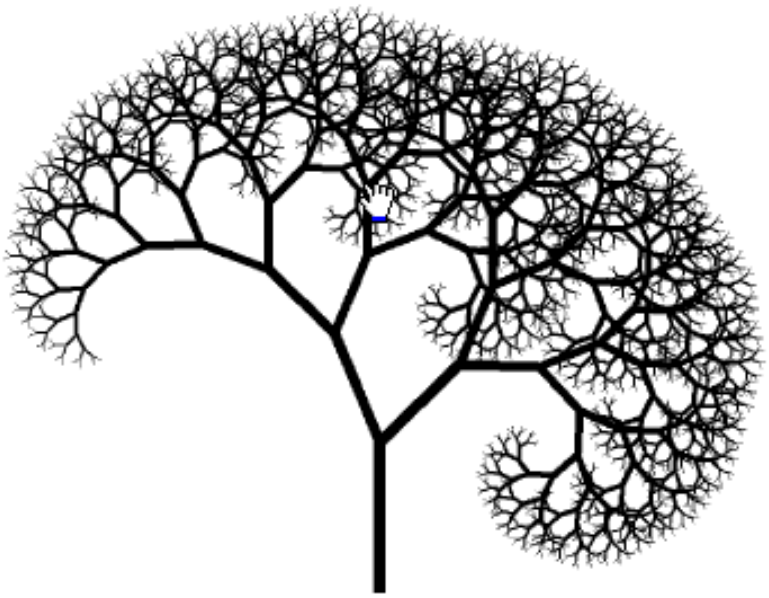
Получившиеся ветки снова рассматриваются как стволы и т. д.

Стохастические фракталы

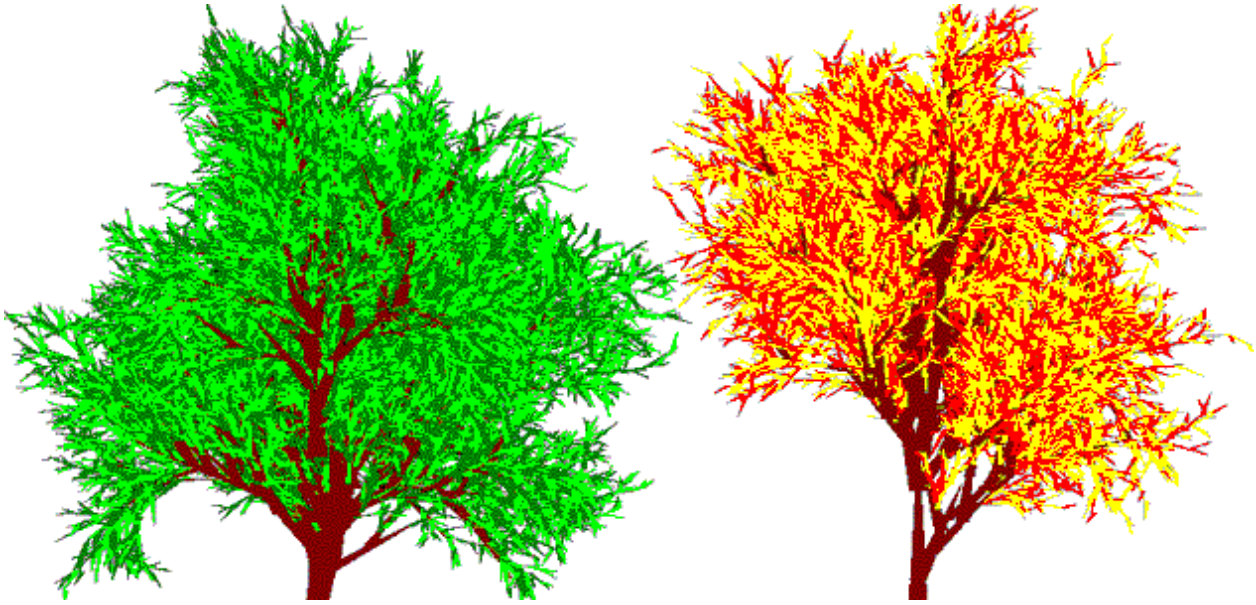
Стохастические фракталы получаются в том случае, когда в итерационном процессе построения рисунка случайно менять какие-то параметры.

При этом получаются объекты, очень похожие на природные: несимметричные деревья, порезанные береговые линии и т. д.

Обдуваемое ветром дерево Пифагора



Деревья



Лист папоротника

Одним из наиболее ярких примеров среди различных систем итерируемых функций, несомненно, является открытая М. Барнсли система из четырех сжимающих аффинных преобразований, аттрактором для которой является множество точек, поразительно напоминающее по форме изображение листа папоротника.

Систему из четырех сжимающих аффинных преобразований можно представить в виде следующей таблицы 18:

Таблица 18 – Пример сжимающих аффинных преобразований.

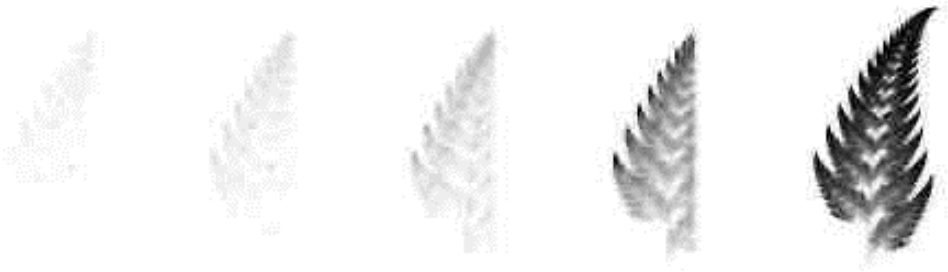
a	*b*	*c*	*d*	*e*	*f*	*p*
0	0	0	0.16	0	0	0.01
0.85	0.04	-0.04	0.85	0	1.6	0.85
0.2	-0.26	0.23	0.22	0	1.6	0.07
-0.15	0.28	0.26	0.24	0	0.44	0.07

Каждая строчка этой таблицы соответствует одному аффинному преобразованию с коэффициентами a, b, c, d, e, f в соответствии с выражением

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}.$$

В последнем столбце таблицы приведены вероятности p, в соответствии с которыми в методе случайных итераций выбирается то или иное преобразование.

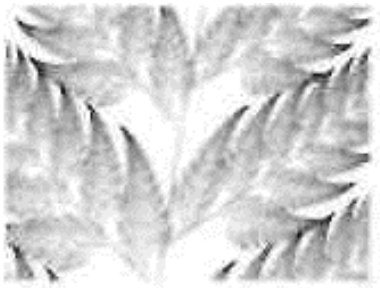
Результат действия этой системы функций на некоторую начальную точку для разного числа итераций приведен на рисунке.



Слева направо показано 2000, 4000, 10000, 50000 и 200000 итераций.

Видно, как с ростом числа итераций действительно возникает все более и более четкое изображение листа папоротника, удивительным образом напоминающее существующее в природе растение.

Это множество точек бесконечно само подобно, как и полагается всякому фракталу. Как следует из следующего рисунка, увеличенные малые фрагменты изображения подобны целому. Для разрешения этих фрагментов необходимо только, чтобы число итераций было достаточно велико.



Таким образом, чем больше используемое разрешение, тем больше точек требуется внести в память компьютера для того, чтобы построить соответствующее изображение. С другой стороны, запоминать координаты этих точек вовсе не требуется, так как они каждый раз могут быть заново получены с использованием системы функций, заданных предыдущей таблицей.

В результате всего 28 чисел содержат всю необходимую информацию об этом нетривиальном рисунке!

Возникла в свое время мысль, а нельзя ли подобным образом «кодировать» и другие изображения. Эта идея, будучи реализованной на практике. позволила бы сжимать изображения в десятки или даже в сотни раз.

И действительно, в 1988 г. она была успешно воплощена Барнсли и Слоаном в созданной ими совместно компании по кодированию и сжатию графической информации с помощью соответствующим образом подобранной системы функций.

В книге «Фракталы и мультифракталы» С. В. Божокина и Д. А. Паршина очень подробно разобрана эта система на примере квадрата и дана

геометрическая интерпретация числам, приведенным в таблице (а это сжатие, поворот, отражение и параллельный перенос).

Построение стохастического фрактала «папоротник»

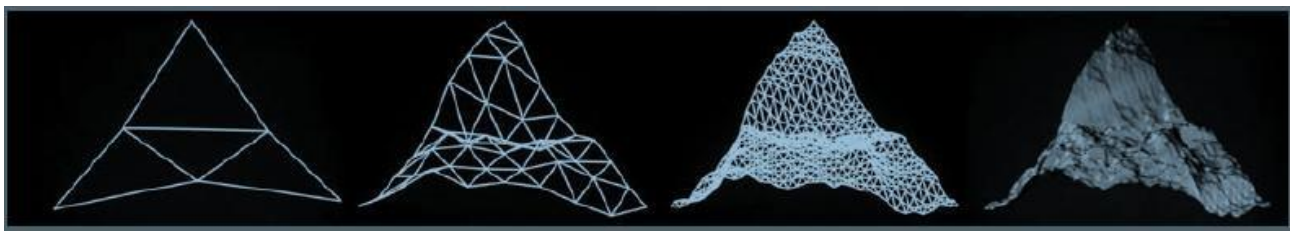
```
Program Paparatz;
Uses
     Graph, Crt;
var
      Ch      : Char;
      dr, dm  : Integer;

procedure Draw;
const
      Iteration = 50000;
var
    t, x, y, p           : Real;
    k                    : Longint;
    mid_x, mid_y, radius : Integer;
begin
 mid_x := GetMaxX div 2;
 mid_y := GetMaxY;
 radius := Trunc(0.1 * mid_y);
 Randomize;
 x := 1.0;
 y := 0.0;
 for k:=1  to iteration  do
   begin
    p := random;
    t := x;
    if p <= 0.85  then
      begin
       x := 0.85 * x - 0.04 * y;
       y := -0.04 * t + 0.85 * y + 1.6;
      end
    else
        if p <= 0.92  then
          begin
           x := 0.20 * x - 0.26 * y;
           y := 0.23 * t + 0.22 * y + 1.6;
          end
        else
            if p <= 0.99  then
              begin
               x :=-0.15 * x + 0.28 * y;
               y := 0.26 * t + 0.24 * y + 0.44;
              end
            else
```

```
                begin
                 x := 0.0;
                 y := 0.16 * y;
                end;
                    Delay(20);
    PutPixel(mid_x + Round(radius * x),
             mid_y - Round(radius * y), 4);
   end;
end;

begin
 dr := 0;
 InitGraph(dr, dm, '');
 SetBkColor(blue);
 Clearviewport;
 draw;
 OutTextXY(0, 465, 'Press <Space>:');
 Ch := ReadKey;
end.
```

На следующем рисунке приведен другой пример стохастического фрактала «папоротник»

Примеры практического использования фракталов

1. Лорен Карпентер (Loren C. Carpenter) в 1967 году применил в компьютерной графике фрактальный алгоритм для построения изображений. Он смог визуализировать реалистичное изображение горной системы на своем компьютере, разделяя более крупную геометрическую фигуру на мелкие элементы, а те, в свою очередь, деля на аналогичные фигуры меньшего размера пока у него не получался реалистичный горный ландшафт.



После этого в мировой практике стали использовать фрактальный алгоритм для имитации реалистичных природных форм.

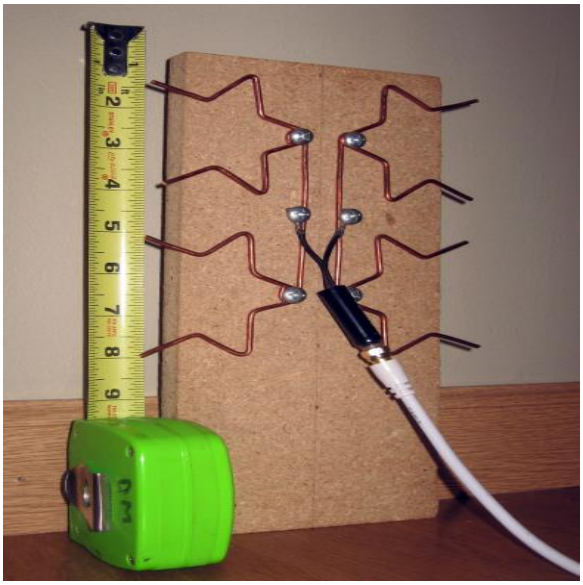
Всего через несколько лет свои наработки Лорен Карпентер смог применить в куда более масштабном проекте. Аниматор создал на их основе двухминутный демонстрационный ролик Vol Libre, который был показан на Siggraph в 1980 году. Это видео потрясло всех, кто его видел, и Лоурен получил приглашение от Lucasfilm.

2. Радиолюбитель Натан Коэн стремился создать антенну, обладающую как можно более высокой чувствительностью. Единственный способ улучшить параметры антенны, который был известен на то время, заключался в увеличении ее геометрических размеров. Натан стал экспериментировать с различными формами антенн, стараясь получить максимальный результат при минимальных размерах. Загоревшись идеей фрактальных форм, Коэн сделал из проволоки один из самых известных фракталов «снежинку Коха».

После серии экспериментов будущий профессор Бостонского университета понял, что антенна, сделанная по фрактальному рисунку, имеет высокий КПД и покрывает гораздо более широкий частотный диапазон по сравнению с классическими решениями. Кроме того, форма антенны в виде кривой фрактала позволяет существенно уменьшить геометрические размеры.

Автор запатентовал свое открытие и основал фирму по разработке и проектированию фрактальных антенн Fractal Antenna Systems, справедливо

полагая, что в будущем благодаря его открытию сотовые телефоны смогут избавиться от громоздких антенн и станут более компактными.



2. СЕМЕСТР

1.5. Методы сортировки данных

Под *сортировкой* понимают процесс целенаправленной перестановки элементов данных в указанном порядке по возрастанию или убыванию согласно определенным линейным отношениям их порядка, таким как отношения «≥» (больше или равно) или «≤» (меньше или равно) для чисел.

Цель сортировки – облегчить потом *поиск* элемента в отсортированной (упорядоченной) совокупности элементов. Аналогом является поиск элемента в словаре, телефонном справочнике, ведомостях и др.

Сортировка – одна из распространенных задач в информатике. Алгоритмы сортировки являются одним из наиболее изученных направлений информатики. Сортировка является идеальным примером огромного разнообразия алгоритмов, выполняющих одну и ту же задачу.

Объемы информационных массивов непрерывно и стремительно растут, соответственно возрастают и требования к скорости сортировки, что обусловливает актуальность оптимизации используемых алгоритмов.

Реализация алгоритмов сортировки имеет большое количество разного рода ухищрений. На примере сортировки можно убедиться в необходимости сравнительного анализа алгоритмов, так как при помощи усложнения алгоритма можно получить существенное увеличение эффективности при сравнении с более простыми и очевидными алгоритмами. Во многих случаях определить характеристики выполнения сортировки довольно просто.

Существующие алгоритмы сортировки значительно различаются по уровню сложности, скорости, устойчивости, требованиям к памяти и другим параметрам. Однако практически каждый алгоритм оказывается наиболее удобным в какой-либо конкретной ситуации. Востребованными являются даже очень медленные алгоритмы, которые из-за своей простоты находят применение в образовательных целях.

Зависимость выбора алгоритма от структуры данных – явление довольно частое, и в случае сортировки она очень сильная, поэтому методы сортировки обычно разделяют на две категории:

- сортировка данных типа массив, сортировка данных типа связанные списки;
- сортировка последовательных файлов.

Эти два класса часто называют *внутренней* и *внешней* сортировками.

Внутренняя сортировка. Массивы и списки располагаются в оперативной памяти быстрого доступа к каждому элементу. Данные обычно упорядочиваются на том же месте без дополнительных затрат. В современных архитектурах персональных компьютеров широко применяется подкачка и кэширование памяти. Алгоритм сортировки должен хорошо сочетаться с применяемыми алгоритмами кэширования и подкачки.

Внешняя сортировка оперирует запоминающими устройствами большого объёма, но не с произвольным доступом, а последовательным: в каждый момент времени можно считать или записать только элемент, следующий за текущим.

Объём данных не позволяет им разместиться в оперативной памяти. Это накладывает некоторые дополнительные ограничения на алгоритм и приводит к специальным методам упорядочения, обычно использующим дополнительное дисковое пространство. Отметим, что доступ к данным во внешней памяти производится намного медленнее, чем операции с оперативной памятью.

Сортировка строк – это ранжирование данных строкового типа по алфавиту. Сортировка строк напоминает во всем сортировку массивов, так как доступ имеем к любому элементу строки, т. е. символу, который в памяти хранится своим кодом. Происходит сравнивание строковых данных по закону «≥» (больше или равно) или «≤» (меньше или равно). Следует учитывать, что применение сравнения к строкам, представляющим собой числа в естественной записи, выдаёт контр интуитивные результаты: например, «9» оказывается больше, чем «11», так как первый символ первой строки имеет большее значение, чем первый символ второй строки. Для исправления этой проблемы алгоритм сортировки может преобразовывать сортируемые строки в числа и сортировать их как числа. Такой алгоритм называется «числовой сортировкой», а описанный ранее – «строковой сортировкой».

Сортировка элементов файла с прямым доступом (а не последовательным) также напоминает во всём сортировку массивов, так как доступ имеем к любому элементу, однако нужно помнить, что работа с файлами происходит при помощи буферов обмена, в который идет подкачка информации, и это накладывает свои негативные отпечатки.

Оценка алгоритма сортировки

Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти:

• **время** – основной параметр, характеризующий быстродействие алгоритма, называемый также вычислительной сложностью. Для упорядочения важны худшее, среднее и лучшее поведение алгоритма в терминах размера n входной последовательности A. Для типичного алгоритма хорошее поведение – это $O(n \log n)$ и плохое поведение – это $O(n^2)$. Идеальное поведение для упорядочения – $O(n)$;

• **память** — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке не учитывается место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы (так как всё это потребляет $O(1)$). Алгоритмы сортировки, не потребляющие дополнительной памяти, относят к сортировкам на месте.

Свойства алгоритма и классификация

• **Устойчивость** (англ. *stability*). При использовании алгоритмов устойчивой (стабильной) сортировки при наличии в наборе данных нескольких равных элементов в отсортированном наборе они сохраняются в том же порядке, в котором были в исходном наборе. Таким образом, устойчивая сортировка не меняет относительный порядок сортируемых элементов, имеющих одинаковые

ключи. Сохранение взаимного расположения равных элементов важно при сортировке по одному полю данных, состоящих из нескольких полей.

• **Естественность поведения** – эффективность метода при обработке уже упорядоченных или частично упорядоченных данных. Алгоритм ведёт себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.

• **Использование операции сравнения**. Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях.

• **Алгоритмы, не основанные на сравнениях**. Как следует из самого их названия, такие алгоритмы совсем не используют сравнений сортируемых элементов.

• **Прочие алгоритмы сортировки.**

1.6. Сортировка массивов

Введём систему обозначений, которую будем использовать далее. Пусть нам дан массив – последовательность элементов $a_1,\ a_2,\ \ldots,\ a_n$.

Сортировка по возрастанию (убыванию) означает перестановку этих элементов в порядке $a_{k_1},\ a_{k_2},\ \ldots,\ a_{k_n}$ так, что при заданной функции упорядочения f справедливы отношения $f(a_{k_1}) \le f(a_{k_2}) \le \ldots \le f(a_{k_n})$ (или, наоборот, $f(a_{k_1}) \ge f(a_{k_2}) \ge \ldots \ge f(a_{k_n})$).

Обычно функция упорядочения не подсчитывается по какому-то специальному правилу, а присутствует в каждом элементе в виде явной компоненты (поля элемента), которую называют *ключом* элемента. Таким образом, для представления элемента a_i особенно хорошо подходит структура записи.

Для дальнейшего изложения материала определим тип `item` [`aitem`], который будет использоваться в алгоритмах сортировки:

```
const   n = 100;
type
      inf = record
    ... {описание полей записи без ключевого поля}
      end;
type
     item = record
         key   : Integer;  {описание ключевого поля}
         zmest : inf;      {«другие компоненты»   }
            end;
type
     index = 1..n;
```

«Другие компоненты» – это все существенные данные элемента. Поле `key` – ключ служит только для идентификации элементов. Но если мы говорим об алгоритмах сортировки, ключ для нас – единственная существенная компонента и

нам нет необходимости сейчас определять остальные. Тип ключа может быть любым типом, для которого заданы отношения всеобщего порядка «больше или равно» («меньше или равно»).

Сделаем еще такое определение:

```
var  A: array [1..n] of item;
```

Разберем подробно некоторые интересные алгоритмы сортировки. Полный обзор методов сортировки можно найти в книгах [2, 7–9] и др.

Основное требование к методам сортировки массивов – экономное использование памяти. Это значит, что сортировку элементов нужно выполнять на том же месте (*in situ*), и поэтому методы, которые пересылают элементы из массива A в массив B, для нас на данном этапе не представляют интереса.

Таким образом, выбирая метод сортировки и руководствуясь критериями экономии памяти, классификацию алгоритмов мы будем проводить в соответствии с их *эффективностью*, это значит *экономией* времени и *быстротой действия*. При этом мы будем подсчитывать *C – необходимое количество сравнений ключей* и *M – перенаправления* (*перестановку*) *элементов* (что отнимает много времени). Это будут функции от *n* – числа элементов, которые сортируются.

Исторически повелось, что методы, которые сортируют элементы *in situ,* можно разбить на три основных класса в зависимости от заложенного в их основе базового алгоритма:

- сортировка обменом;
- сортировка включением;
- сортировка выбором.

Рассмотрим методы сортировки массивов в соответствии с этой классификацией и сразу будем искать те подходы, которые позволят улучшать базовый алгоритм. Не снижая общности, сортировку элементов будем проводить в *порядке возрастания ключа*.

При различных методах сортировки интерес будут вызывать такие вопросы:

1) как ведет себя алгоритм в крайних ситуациях:
   - массив упорядочен в нужном порядке;
   - массив упорядочен в обратном порядке;
   - массив не упорядочен;
2) на каких массивах алгоритм дает количество действий:
   - минимальную;
   - максимальную;
3) чему равняется среднее количество действий.

Первый тип. Сортировка обменом

Первый метод. Сортировка простым обменом

Это один из простейших методов сортировки, который обычно входит в школьный курс информатики.

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок упорядочения в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются n-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает – массив отсортирован.

Если выполнять проходы справа налево, то при каждом проходе алгоритма по внутреннему циклу очередной наименьший элемент массива ставится на своё место в начале массива рядом с предыдущим «наименьшим элементом», а наибольший элемент перемещается на одну позицию к концу массива.

Это *сортировка простым обменом*, или *«пузырьком»*. Минимальный элемент как бы всплывает (как *пузырёк*) в начале массива. Вместо поднятия «самого легкого» можно «топить» самый «тяжелый».

Сначала рассмотрим работу такого алгоритма на примере: A = {44, 55, 12, 42, 94, 18, 06, 67} (n = 8).

На Рис. 1 в строках отражены промежуточные состояния массива A при сортировке простым обменом (n = 8).

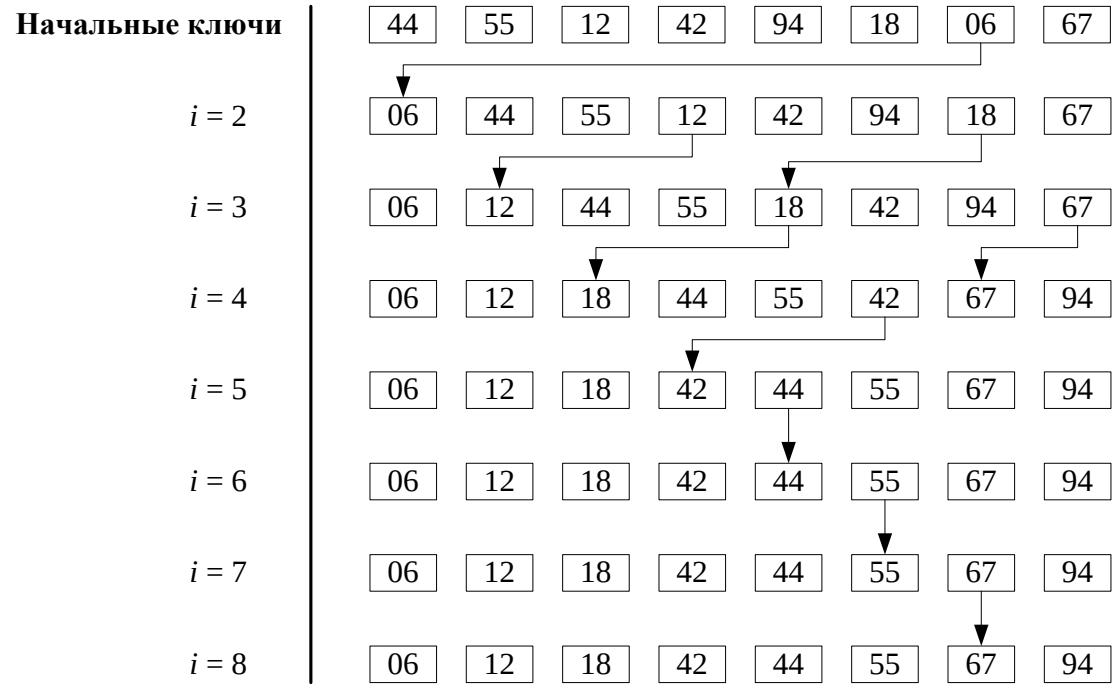


Рис. 1

Фрагмент кода, который реализует сортировку простым обменом, приведён в листинге.

```
procedure BubbleSort;      {проход справа налево}
var
   i, j : index;
   temp : item;
begin
for i := 2  to  n  do
    for j := n   downto   i do
        if a[j-1].key > a[j].key then
           begin
```

```
            temp   := a[j-1];
            a[j-1] := a[j];
            a[j]   := temp;
            end;
end; {BubbleSort}
```

Подсчитаем количество сравнений:

$$C = (n-1) + (n-2) + ... + 2 + 1 = \frac{n-1+1}{2}(n-1) = \frac{n^2 - n}{2} = O(n^2).$$

Количество обменов:

$$M_{\min} = 0;\ M_{\max} = \frac{3}{2}(n^2 - n);\ M = \frac{3}{4}(n^2 - n)\ \text{(среднее)}.$$

Таким образом, $M_{\max} = 3C$.

Можно заметить, что массив был отсортирован на каком-то промежуточном шаге ($i = 5$), но алгоритм построен так, что такое явление в нём не анализируется.

Второй метод. Сортировка методом пузырька с преждевременным выходом

Этот метод – оптимизация предыдущего метода: нужно запомнить, проводился ли на данном проходе какой-либо обмен, и если нет, то можно преждевременно закончить работу.

Сортировка простым обменом с преждевременным выходом

```
procedure BubbleSort_2;
var
      i, j : index;
      temp : item;
      flag : Boolean;
begin
 for i := 2  to  n do
   begin
    flag := true;
    for j := n  downto  i do
        if a[j - 1].key > a[j].key then
          begin
           temp    := a[j - 1];
           a[j-1]  := a[j];
           a[j]    := temp;
           flag    := false;
          end;
     if flag then exit;
    end
end; {BubbleSort2}
```

Третий метод. Оптимальная обменная сортировка

В этом методе сортировки нужно запомнить не только факт обмена, но и индекс последнего обмена. Очевидно, что все пары соседних элементов с индексами, меньшими чем этот индекс, уже расположены в нужном порядке. Поэтому проход можно заканчивать на этом индексе, вместо того, чтобы двигаться до конца. Программу алгоритма напишите самостоятельно.

Четвертый метод. Алгоритм шейкер-сортировки

Анализируя метод пузырьковой сортировки, можно отметить два обстоятельства:

- если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, её можно исключить из рассмотрения;
- при движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

Например:

а) если $A = \{94, 06, 12, 18, 42, 44, 55, 67\}$, то просмотр слева направо сортирует массив за один проход, а справа налево – за семь;

б) если $A = \{12, 18, 42, 44, 55, 67, 94, 06\}$, то просмотр слева направо сортирует массив за семь проходов, а справа налево – за один.

Это приводит к следующей модификации метода пузырьковой сортировки:

- границы сортируемой части устанавливаются в месте последнего обмена на каждом проходе;
- массив просматривается поочередно справа налево и слева направо.

Шейкер-сортировка (с запоминанием места последнего обмена)

```
procedure ShakerSort;
var
    j, k, L, r : index;
          temp : item;
begin
 L := 2; r := n; k := n;
 repeat
{1}   for j := r downto L do                    {←}
          if a[j - 1].key > a[j].key then
             begin
             temp := a[j - 1]; a[j - 1] := a[j];
             a[j] := temp ; k := j; {последний обмен}
             end;
      L := k + 1;
 {2}  for j := L to r do                      {→}
          if a[j - 1].key > a[j].key then
             begin
```

```
            temp := a[j - 1]; a[j - 1] := a[j];
            a[j] := temp ; k := j;  {последний обмен}
            end;
        r := k - 1;
    {когда в циклах не было больше обмена, то
              L = k + 1; r = k - 1 i L > r - выход!}
 until L > r;
end; {ShakerSort}
```

Схема работы алгоритма. Пусть $A=\{a_1, a_2, \ldots, a_n\}$ – исходный массив. Выполним первый проход справа налево для $j := n, n-1, \ldots, 2$ и запомним номер k последней перестановки. Заметим, что элементы $\{a_1, a_2, \ldots, a_k\}$ упорядочены. Обозначим $l := k+1$. Выполнив второй проход слева направо для $j := l, l+1, \ldots, n$ и запомнив номер k последней перестановки, заметим, что элементы $\{a_k, a_{k+1}, \ldots, a_n\}$ упорядочены. Обозначим $r := k-1$. Осталось упорядочить элементы $\{a_l, \ldots, a_k\}$. Для их сортировки повторяем чередование просмотров справа налево и слева направо (рис. 2).

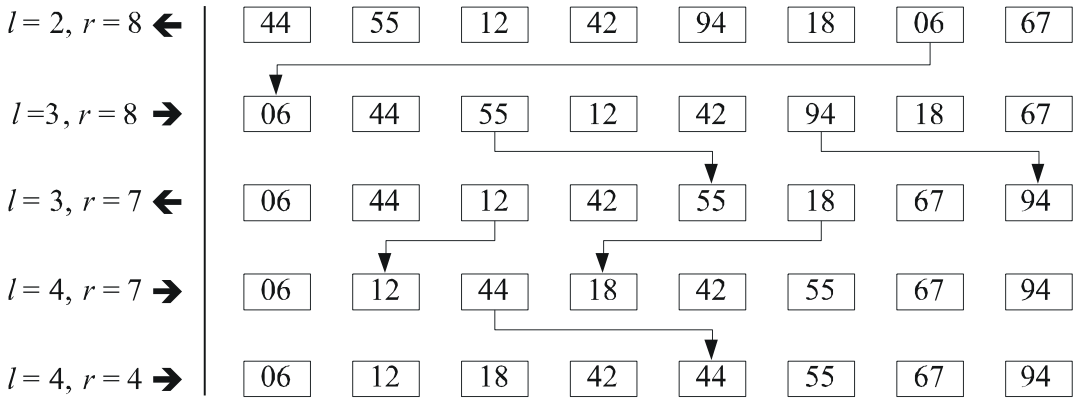


Рис. 2

Лучший случай для этой сортировки – отсортированный массив ($O(n)$), худший – отсортированный в обратном порядке ($O(n^2)$).

Отметим, что сортировка пузырьком массива A = {44, 55, 12, 42, 94, 18, 06, 67} требует $4 \cdot 7 = 28$ просмотров, пузырьком с преждевременным выходом – $7 + 6 + 5 + 4 + 3 = 25$ просмотров, шейкер-сортировкой – $7 + 6 + 5 + 4 + 1 = 23$ просмотра, а количество перестановок одинакова – $O(n^2)$.

Шейкер-сортировку (англ. *Cocktail sort*) в литературе еще называют сортировкой *перемешиванием*, или *двунаправленной*.

Пятый метод. Обменная сортировка с разделением

Сортировка с разделением – это ещё одно улучшение метода, основанного на принципе обмена. Алгоритм настолько хорош, что его автор английский информатик К. Хоар назвал *быстрой сортировкой* (англ. *quicksort*). Ранее рассмотренные методы обменивали соседние элементы. Быстрая сортировка

находит элементы, расположенные не на своих местах: «тяжелый» – в «легком» конце, а «легкий» – в «тяжелом» конце, а потом обменивает их.

Быстрая сортировка относится к алгоритмам «разделяй и властвуй».

Алгоритм состоит из трёх шагов:

1. По вспомогательному алгоритму в массиве выбирается какой-то элемент *x – опорный элемент*.
2. Выполняется разбиение: перераспределение элементов в массиве таким образом, что элементы меньшие опорного помещаются перед ним, а большие или равные после.

Это происходит следующим образом. Движемся по совокупности элементов слева направо, пока не встретим элемент $a_i \geq x$, а затем – справа налево, пока не найдем элемент $a_j < x$. Если $i < j$, обменяем a_i с a_j. Продолжаем просмотр дальше от места $i+1$ до $j-1$. Выполняем такие поиски до тех пор, пока два просмотра не встретятся.

3. Рекурсивно применяются первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.

Рассмотрим данный алгоритм на следующем примере. Пусть для массива A = {55, 12, 42, 94, 06, 18, 44, 67} опорный элемент x равен 42 (средний среди первых трех неравных). Получим такую последовательность перемещений:

A = {55, 12, 42, 94, 06, 18, 44, 67};

A = {18, 12, 42, 94, 06, 55, 44, 67};

A = {18, 12, 06, 94, 42, 55, 44, 67}.

Числа, меньшие чем 42 (42 – опорны элемент) | Числа, не меньше чем 42

Просмотр закончился слева направо на элементе 94, а справа налево – на элементе 06. Они не обмениваются. Получили две части последовательности: в одной числа, меньшие, чем опорный элемент *x*, во второй числа, не меньшие чем x.

От выбора опорного элемента многое зависит. Заметим, что если бы мы выбрали элемент, стоящий в середине последовательности на 4-ом или 5-ом месте, а это соответственно наибольший и наименьший элементы массива, то на первом шаге разбиения получились бы неудачные подмассивы. Проследите этот момент самостоятельно.

Сейчас это разделение исходного массива представим в виде процедуры.

Вспомогательный алгоритм для быстрой сортировки

```
procedure Partition; {разделение}


var
     w, x : item;
begin
```

```
 i := 1;
 j := n;
        {          выбрать случайно x:
     можно посередине или средний среди неравных }
repeat
   while a[i].key < x.key do Inc(i);
   while a[j].key > x.key do Dec(j);
   if i <= j then
     begin
      w    := a[i];
      a[i] := a[j];
      a[j] := w;
      Inc(i);
      Dec(j)
     end;
 until i > j;
end; {разделение}
```

Разделив массив опорным элементом на две части, нужно сделать то же самое с каждой из них, затем с частями этих частей и так далее, пока каждая часть не будет содержать только один или ни одного элемента. Значит, процедура разделения будет сама себя вызвать. Получим следующую процедуру сортировки.

Быстрая сортировка

```
procedure QuickSort;
   procedure Partition(L, r : index);
   var temp, x : item;
   begin
       {Дополнить нахождением опорного элемента x}
    i := L; j := r;
    repeat
      while a[i].key < x.key do Inc(i);
      while a[j].key > x.key do Dec(j);
      if i <= j then
        begin
         temp := a[i]; a[i] := a[j];
         a[j] := temp; Inc(i); Dec(j)
        end;
    until i>j;
    if  L<j  then Partition(L, j);
    if  i<r  then Partition(i, r);
   end; {Sort}

begin
 Partition(1, n);
end; {QuickSort}
```

Существует нерекурсивный вариант алгоритма быстрой сортировки [7].

Анализ быстрой сортировки. Анализ методов сортировки – не всегда простая задача. Вероятностный подход позволил получить такие оценки:

- ожидаемое число обменов ~n / 6;
- общее количество сравнений есть $O(n \log n)$.

Надо отметить, что рекурсивная версия программы требует для рекурсии дополнительную память и время, но для больших n это оправдано.

Это один из известных универсальных алгоритмов сортировки массивов. Из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

Алгоритмы выбора опорного элемента

Для выполнения быстрой сортировки необходимо знать два алгоритма более низкого уровня: как выбирать опорный элемент и как наиболее эффективно переставить элементы последовательности таким образом, чтобы получить два набора элементов – со значениями, меньшими чем значения опорного элемента, и со значениями, большими чем значение опорного элемента.

Рассмотрим некоторые алгоритмы выбора опорного элемента на примере сортировки чисел.

Было бы идеально, если бы выбирался средний элемент последовательности, и тогда слева и справа относительно опорного элемента получалось примерно одинаковое количество элементов. Подсчет среднего элемента последовательности (или медианы последовательности) представляет собой достаточно сложный процесс, к тому же стандартный алгоритм его определения использует метод разделения быстрой сортировки, которую мы сейчас обсуждаем.

Худший случай имеет место, если в качестве опорного элемента выбирается элемент с максимальным или минимальным значением. Тогда фактически никакого существенного разделения на две части не будет и глубина рекурсии будет наибольшей.

Таким образом, после рассмотрения этих двух предельных случаев можно сказать, что желательно выбирать опорный элемент, который был бы как можно ближе к среднему элемента и как можно дальше от минимального и максимального.

В литературе встречаются следующие подходы к выбору опорного элемента последовательности $a_l, a_{l+1}, ..., a_r$ $(1 \le l \le r \le n)$:

- выбирается средний элемент последовательности $a_{(l+r)/2}$;
- выбирается средний элемент среди трех первых неравных;
- выбирается случайный элемент среди элементов последовательности, который потом обменивается со средним $k = l + \text{random}(r - l + 1)$; $m = (l + r)$ div2; $a_k \Leftrightarrow a_m$;
- выбирается элемент с индексом $k = (xl + yr) / (x + y)$, где x и y – произвольные два числа;

• выбирается средний элемент среди a_l, a_r, $a_{(l+r)/2}$. Затем элемент с наименьшим значением попадает в позицию l, средний – в середину последовательности, а элемент с наибольшим значением – в позицию r. Таким образом, при выборе опорного элемента размер частей последовательности сокращается на два элемента, так как уже известно, что они находятся в правильных частях последовательности относительно опорного элемента.

Замечание. Еще одна модификация метода быстрой сортировки. Если сначала количество элементов последовательности велико, то выполняется рекурсивный алгоритм разделения последовательности на две последовательности относительно опорного элемента. Однако если получили последовательность небольшого размера, то уже сортировать можно более медленными методами (метод пузырька, метод вставки, метод выбора).

Второй тип. Сортировка включением

Первый метод. Сортировка простым включением

Сортировка вставками (англ. *Insertion sort*) – алгоритм сортировки, в котором элементы массива $A=\{a_1, a_2, \ldots, a_n\}$ условно делятся на две части: отсортированную

$$\{a_1, a_2, \ldots, a_{i-1}\}$$

и неотсортированную

$$\{a_i, a_{i+1}, \ldots, a_n\}.$$

Сначала отсортированная часть состоит только из одного элемента $\{a_1\}$. На каждом шаге, начиная с $i = 2, 3, \ldots$, берут очередной элемент a_i и вставляют его на соответствующее место в отсортированную часть массива.

```
for i := 2 to n do
  begin
   x := a[i];  {вставить его на подходящее место}
  end;
```

В отсортированную часть массива $\{a_1, a_2, \ldots, a_{i-1}\}$ нужно вставить элемент x, не нарушив порядок упорядочивания. Сделать это возможно, используя просеивание, которое может закончиться при двух условиях:

• найден элемент a_j такой, что $a_j \le x \le a_{j+1}$, $1 \le j \le i-1$;

• достигли левого конца отсортированной части массива.

Цикл на просеивание можно усовершенствовать, если ввести фиктивный элемент («барьер») $a_0 = x$. Тогда окончание просеивания будет всегда только при выполнении первого условия, и второе условие можно не проверять.

Если уже ясно, куда вставить элемент x, тогда нужно освободить для него место, сдвигая элементы a_{j+1}, ..., a_{i-1} на шаг вправо. Однако можно уже при сравнении делать это перемещение. Заметим, что массив $A = \{a_1, a_2, \ldots, a_n\}$ с учетом необходимости хранения барьера нужно описать так:

```
var  a : array[0..n]  of  item;
```

Фрагмент кода, который реализует сортировку простым включением, приведенный в листинге.

Сортировка простым включением

```
procedure StraightInsertion;
var     i, j : index;
        x    : item;
begin
 for i := 2 to n do
    begin
     x      := a[i];
     a[0]   := x;
     j      := i - 1;
     while x.key < a[j].key do
         begin
          a[j + 1]   := a[j];
          j          := j - 1
         end;
     a[j + 1] := x
    end
end;
```

Анализ сортировки простым включением. Число сравнений ключей C_i на i-ом просеивания составляет самое большее $C_{\max} = i - 1$, а меньшее $C_{\min} = 1$, в среднем $C_{\text{c}} = i/2$. Число пересылок M_i (придание значений) равно $C_i + 2$ (учтем барьер). Поэтому M_i принимает наименьшее значение, если элементы изначально упорядочены, а наибольшее – когда элементы расположены в обратном порядке.

$$C_{\min} = \sum_{i=2}^{n} 1 = n - 1, \quad M_{\min} = 3(n-1), \quad C_{\text{c}} = \sum_{i=2}^{n} \frac{i}{2} = \frac{1}{4}(n+2)(n-1).$$

$$M_{\text{c}} = \sum_{i=2}^{n} (\frac{i}{2} + 2) = \frac{1}{4}(n+2)(n-1) + 2(n-1) = \frac{1}{4}(n-1)(n+10).$$

$$C_{\max} = \sum_{i=2}^{n} (i-1) = \frac{(n-1)n}{2}, \quad M_{\max} = \sum_{i=2}^{n} (i-1+2) = \frac{(n-1)(n+4)}{2}.$$

Второй метод. Сортировка простым включением с улучшением

Улучшение можно получить, если поиск места вставки элемента x выполнять *бинарными делениями*. Этим уменьшается количество сравнений. Затем сдвигаем часть элементов в массиве, освобождая место для вставки. Количество перемещений M в таком случае остается прежним, а пересылки занимают гораздо больше времени, чем сравнения. Программу напишите самостоятельно.

Третий метод. Сортировка Шелла

Сортировка Шелла была названа в честь её изобретателя – Дональда Шелла, который опубликовал этот алгоритм в 1959 году.

Сортировка Шелла (англ. *Shell sort*) – алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами – это сортировка вставками с предварительными «грубыми» проходами. Аналогичный метод усовершенствования пузырьковой сортировки называется *сортировка расчёской* (*методом прочесывания*).

Это сортировка включениями с приращением, которое уменьшается. Рассмотрим ее на примере сортировки массива из восьми элементов $A = \{a_1, a_2, \ldots, a_8\}$.

Сначала отдельно группируются и сортируются все элементы, которые отстают друг от друга на четыре позиции. Таким образом сортируются четыре пары элементов массива: $\{a_1, a_5\}$; $\{a_2, a_6\}$; $\{a_3, a_7\}$; $\{a_4, a_8\}$.

Далее группируются и сортируются элементы, которые отстают друг от друга на две позиции. На этом этапе сортируются две группы элементов массива: $\{a_1, a_3, a_5, a_7\}$; $\{a_2, a_4, a_6, a_8\}$.

На последнем этапе выполняется обычная сортировка всех элементов массива.

На первый взгляд, добавились два лишних шага (сортировка групп элементов, которые отстают друг от друга на четыре и две позиции). Но на каждом этапе сортируется относительно мало элементов или элементы уже довольно хорошо упорядочены.

Пример. Над исходным массивом $A = \{44, 55, 12, 42, 94, 18, 06, 67\}$ выполним сортировку каждого четвертого элемента (рис. 3).

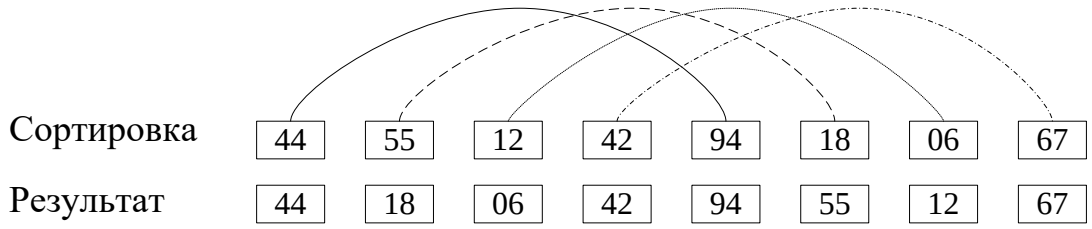


Рис. 3. Четверная сортировка

Результатом будет массив $A = \{44, 18, 06, 42, 94, 55, 12, 67\}$. Выполним в нем сортировку каждого второго элемента (рис. 4).

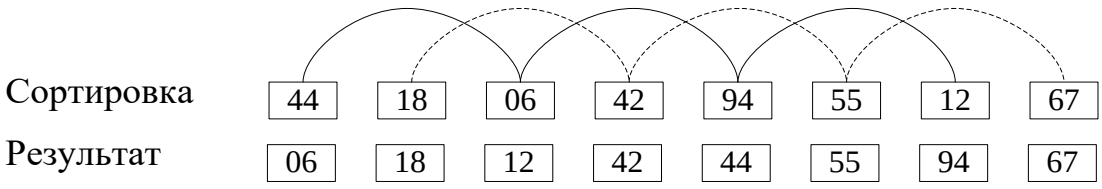


Рис. 4. Двойная сортировка

Получим следующий массив: A = {06, 18, 12, 42, 44, 55, 94, 67}. Выполним сортировку и получим полностью отсортированный массив A = {06, 12, 18, 42, 44, 55, 67, 94} (рис. 5).

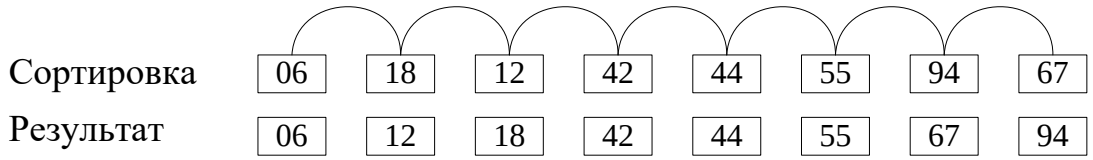


Рис. 5. Одинарная сортировка

Продвижение легкого элемента вперед идет быстрыми темпами.

Очевидно, что этот метод в итоге дает упорядоченный массив, и также ясно, что каждый проход будет использовать результаты предыдущего прохода, поскольку каждая *i*-ая сортировка объединяет две группы, отсортированные предыдущей сортировкой. Анализ показал, что допустимо произвольная последовательность приращений, важно чтобы последнее было равное 1, так как в худшем случае вся работа будет выполнена на последнем проходе и, как показывает практика, даже лучшие результаты получаются, если приращение не является степенью двойки.

Обозначим все t приращения через $h_1, h_2, \ldots, h_t$ с условиями: $h_t = 1$, $h_{i+1} < h_i$. Каждая h_i-сортировка программируется как сортировка простыми включениями, при этом, для того, чтобы условие окончания поиска места включения была простой, используется барьер. Поэтому массив a нужно определить так:

```
var   a : array[-h1..n]  of  Integer;
```

Фрагмент кода, который реализует сортировку Шелла, приведенный в листинге.

Сортировка Шелла

```
procedure ShellSort;
const
      t = 4;
      h : array[1..t] of  Byte = (9, 5, 3, 1);
var
      i, j, k, s : Integer;
      x          : item;
      m          : 1..t;
begin
 for m := 1 to t do
   begin
    k := h[m];
    s := -k;
    for i := k + 1  to   n   do
       begin
        x := a[i];
        j := i - k;
        if s = 0  then  s := -k;
        s := s + 1;     a[s] := x;   {  барьер  }
        while x.key < a[j].key do
```

```
                begin
                 a[j + k] := a[j];
                 j        := j - k;
                end;
           a[j + k] := x;
          end
     end
end;
```

Анализ сортировки Шелла

До сих пор неизвестно, какая последовательность приращений дает наилучший результат, но оказалось, что лучше, если приращения не кратны друг другу. В противном случае каждый проход сортировки объединяет две цепочки, которые ранее не взаимодействовали, а лучше, чтобы взаимодействие происходило как можно чаще. Имеет место следующее утверждение: если *k*-отсортированная последовательность *i* сортируется, то она остается *k*-отсортированной.

Чем больше будут взаимодействовать последовательности, которые сортируются, тем быстрее получится результат. Например, в 1969 Г. Д. Кнут предложил последовательность приращений (используется в обратном порядке) 1, 4, 13, 40, 121, ... ($h_{i+1} = 3h_i + 1$), которая дает в среднем случае скорость $O(n^{5/4})$, в худшем случае – $O(n^{3/2})$. Известно, что самая быстрая последовательность, разработанная Р. Седжвиком: 1, 5, 19, 41, 109 и т. д., дает в среднем случае $O(n^{7/6})$, а в худшем – $O(n^{4/3})$. Хороший результат дает последовательность: 1, 3, 7, 15, 31, ..., где $h_{k-1} = 2 \cdot h_k + 1$ и $t = [\log_2 n] - 1$.

Третий тип. Сортировка выбором

Первый метод. Сортировка простым выбором

Шаги алгоритма:

- Среди *n* элементов выбирается элемент с наименьшим ключом и меняется местами с первым.
- Потом действие повторяется с остальными *n-1* элементами, *n-2* элементами и т д., пока не останется один последний элемент.

Рассмотрим работу такого алгоритма при сортировке массива
A = {44, 55, 12, 42, 94, 18, 06, 67}.

На Рис. 6 в строках показано промежуточное состояние массива при сортировке простым выбором.

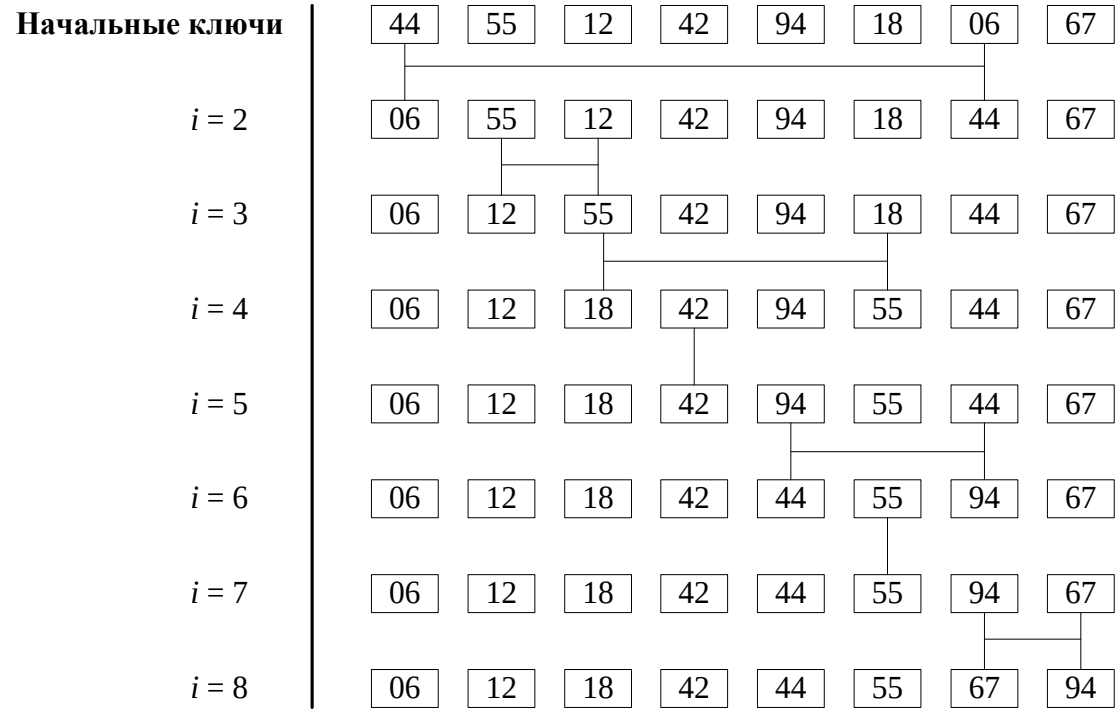


Рис. 6

Фрагмент кода, который реализует сортировку простым выбором, приведенный в листинге.

Сортировка простым выбором

```
 Program StraightSelection;
var     i, j, k : index;
         x       : item;
 begin
  for  i := 1  to  n - 1  do
    begin
     k := i;
     x := a[i];
     for  j := i + 1  to  n  do
          if  a[j].key < x.key  then
             begin
              k := j;
              x := a[j];
             end;
     a[k] := a[i];
     a[i] := x;
    end;
 end;
```

Анализ сортировки простым выбором. Сравнение:

$$C = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}.$$

Минимальное число пересылок: $M_{\min} = 3(n-1)$.

Максимальное число пересылок получается, если ключи расположены в обратном порядке:

$$M_{\max} = \sum_{i=1}^{n-1}(n-(i+1)+1) + 3(n-1) = n(n-1) - \frac{n}{2}(n-1) + 3(n-1).$$

Сортировка простым выбором (англ. *Straight Selection sort*) на массиве из n элементов имеет время выполнения в худшем, среднем и лучшем случае $O(n^2)$, если сравнения делаются за постоянное время.

Второй метод. Сортировка при помощи дерева

Улучшить предложенный выше метод сортировки простым выбором можно, если после каждого сравнения хранить больше информации об элементах. Например, на первом шаге при помощи $n/2$ сравнений можно определить наименьший ключ из каждой пары; на втором шаге при помощи $n/4$ сравнений из выбранных ключей можно определить наименьший из каждой пары и т. д. Получим наименьший из элементов в корне дерева (рис. 7).

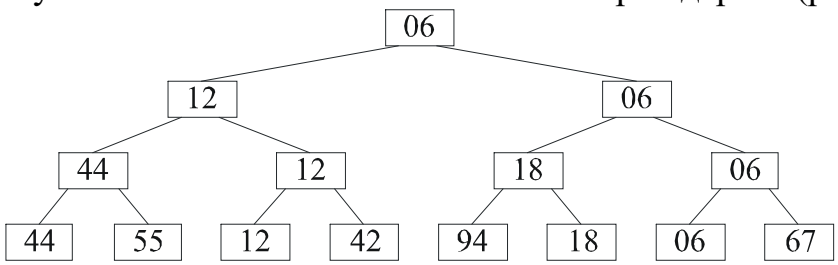


Рис. 7

На втором этапе мы спускаемся по пути, указанному наименьшим ключом, и исключаем его в исходной последовательности, заменяя, например, на ∞ (рис. 8).

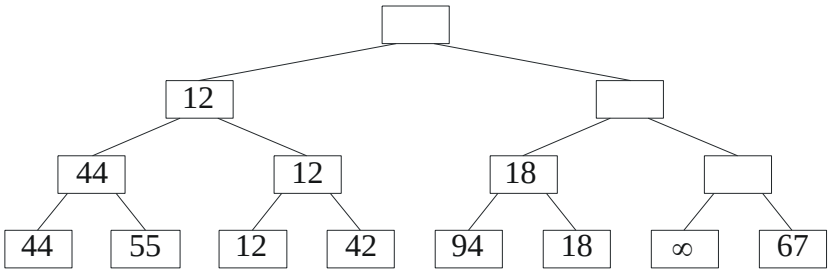


Рис. 8

Далее по такой схеме находим наименьший среди $n-1$ элементов (рис. 9).

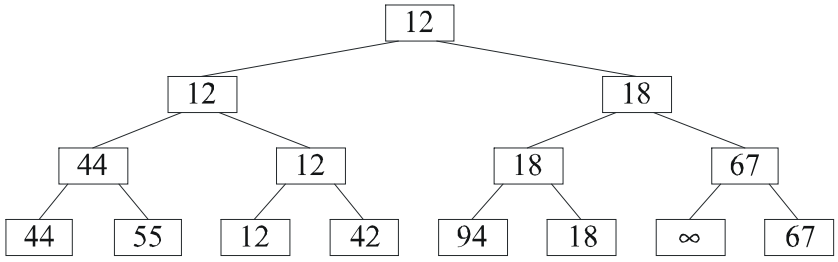


Рис. 9

После n таких шагов последовательность состоит только из ∞, значит, процесс сортировки закончился.

Каждый из n шагов требует $\log_2 n$ сравнение (так как уменьшение шло каждый раз в 2 раза), тогда за n шагов получаем $n\log_2 n$ сравнений. Сортировки же простым выбором требует n^2 сравнений.

Но реализация этого метода требует дополнительную память для хранения дерева – информации после каждого из сравнений. Нужно найти более удобный метод хранения дерева и каким-то образом освободиться от необходимости хранить ∞.

Оригинальный метод изобрел Дж. Вильямс и назвал его *пирамидальной сортировкой.*

Третий метод. Пирамидальная сортировка

Пирамида определяется как последовательность ключей $h_l, h_{l+1}, \ldots, h_r$, такая, что выполняются условия

$$\begin{cases} h_i \le h_{2i}, \\ h_i \le h_{2i+1} \end{cases} \qquad (*)$$

для любого $i = l, ..., r/2.$

Отсюда $h_1 = \min(h_1, ..., h_n).$

Замечание. Условия

$$\begin{cases} h_i \ge h_{2i}, \\ h_i \ge h_{2i+1} \end{cases} \qquad (**)$$

для любого $i = l, ..., r/2,$ так же определяют пирамиду, у которой $h_1 = \max(h_1, ..., h_n).$

Фактически получается следующее дерево (рис. 10):

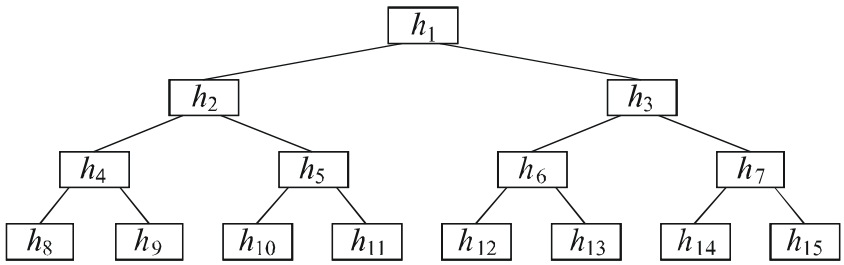


Рис. 10

Встают вопросы: как строить и сохранять пирамиду. Оказывается, это можно делать *in situ.*

Допустим, что дана пирамида с элементами $h_{l+1}, ..., h_r$ (l, r – зафиксированы), удовлетворяющими условию (*). Нужно добавить новый элемент x так, чтобы сформировать расширенную на этот элемент пирамиду $h_l, h_{l+1}, \ldots, h_r.$

Если новый элемент сначала добавить в вершину дерева, а затем «просеивать» по пути, на котором находятся меньшие по сравнению с ним элементы, которые одновременно поднимаются вверх, тогда сформируется новая (расширенная на один элемент) пирамида, так как при этом выполняется условие (*).

Рассмотрим пример.

Пусть имеется пирамида: h_2, ..., h_7: $h_2 = 42$, $h_3 = 6$, $h_4 = 55$, $h_5 = 94$, $h_6 = 18$, $h_7 = 12$.

Нужно добавить в нее новый элемент 44. Берем $h_1 = 44$ (рис.11).

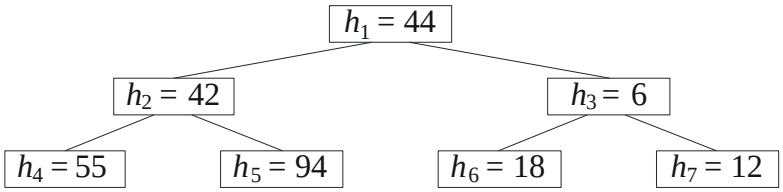


Рис.11

По условию (*) $h_1 = 44$ сравнивается с меньшим среди $h_2 = 42$ и $h_3 = 6$. Это элемент $h_3 = 6$, и он обменивается с $h_1 = 44$.

Получится: $h_1 = 6$, $h_3 = 44$.

Далее $h_3 = 44$ сравнивается с меньшим среди $h_6 = 18$ и $h_7 = 12$. Это $h_7 = 12$, и он обменивается с $h_3 = 44$.

Получилась расширенная пирамида, 42, 12, 55, 94, 18, 44 (рис.12).

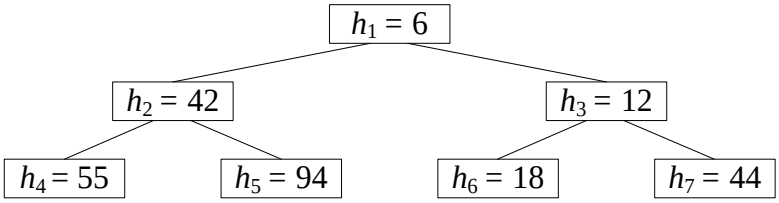


Рис.12

Основываясь на этом приеме просеивания, красивый способ построения пирамиды in situ предложил Р.В. Флойд. Рассмотрим этот способ.

Пусть задан массив $h_1, h_2, \ldots, h_n$. Ясно, что элементы $h_{n/2+1}, \ldots, h_n$ уже образуют пирамиду, потому что для $n/2+1 \le i \le n$ здесь не существует элементов с номерами 2i, 2i + 1. Фактически эти элементы используют как самый нижний ряд бинарного дерева (как на 10).

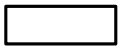
Первый этап. Начиная с элемента i = n / 2 с шагом от -1 до 1, будем просеивать через предыдущую пирамиду очередной h_i элемент. В конце этой процедуры в вершину пирамиды перенесется самый маленький элемент. Но последовательность еще не отсортирована. Сделаем следующее.

Второй этап. Обменяем первый элемент с последним и просеем его через пирамиду, не задевая последнего. Новый первый элемент обменяем с предпоследним и просеем его через пирамиду, не задевая двух последних. И так сделаем n-1 раз. Получим упорядоченный массив по убыванию значений.

Отобразим первый этап построения пирамиды для массива A = {44, 55, 12, 42, 94, 18, 06, 67}. Далее на 13-16 введены следующие обозначения (таблица 19).

Таблица 19 – Возможные обозначения блоков.

Обозначения на рисунках пирамидальной сортировки

Элемент	Обозначение
	Элемент массива h_i, i = 1, 2, ..., 8
	Линия, справа от которой элементы удовлетворяют определению пирамиды (*), слева - еще не просеянные элементы
	Элемент массива h_i, который просеивается
	Элементы массива h_{2i} и h_{2i+1}, с которыми сравнивается просеиваемый элемент h_i; наименьший из двух обведен штриховой линией, если он меньше чем элемент, который просеивается
	Обмен элемента h_i, который просеивается, с наименьшим элементом из пары h_{2i} и h_{2i+1}

Элементы h_i, i = 5, 6, 7, 8, уже образуют пирамиду, поскольку в массиве A не существует элементов h_{2i} h_{2i+1}, а значит, условие (*) выполнено. Просеивание начнем с элемента $h_4 = 42$. Элемент $h_4 = 42$ сравнивается с $h_8 = 67$ (элемента h_9 не существует); поскольку $h_4 < h_8$, то h_4 остается на месте и элементы h_i, i = 4, 5, 6, 7, 8, уже образуют пирамиду (рис.13).

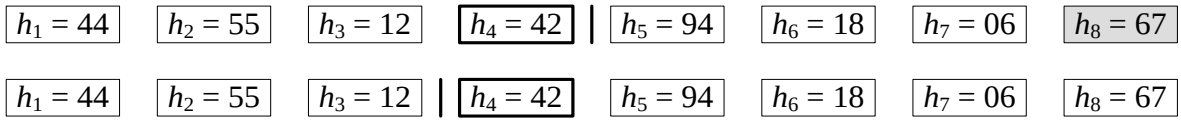


Рис.13

Просеиваем элемент $h_3 = 12$. Он сравнивается с меньшим среди $h_6 = 18$ и $h_7 = 06$; поскольку $h_3 > h_7$, то h_3 обменивается с h_7. В итоге элементы h_i, i = 3, 4, 5, 6, 7, 8, уже образуют пирамиду (рис.14).

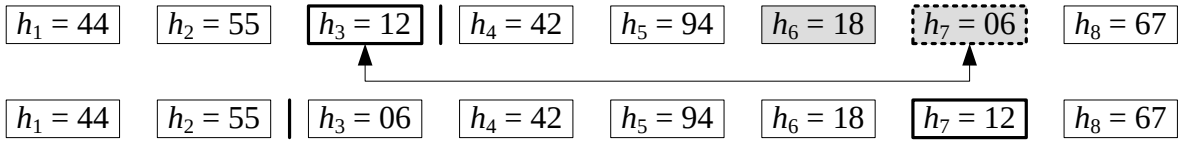


Рис.14

Просеиваем элемент $h_2 = 55$. Он сравнивается с меньшим среди $h_4 = 42$ и $h_5 = 94$; поскольку $h_2 > h_5$, то h_2 обменивается с h_5. Потом элемент $h_4 = 55$ сравнивается з $h_8 = 67$ (элемента h_9 не существует); поскольку $h_4 < h_8$, то h_4 остается на месте и элементы h_i, i = 2, 3, 4, 5, 6, 7, 8, уже образуют пирамиду (рис. 15).

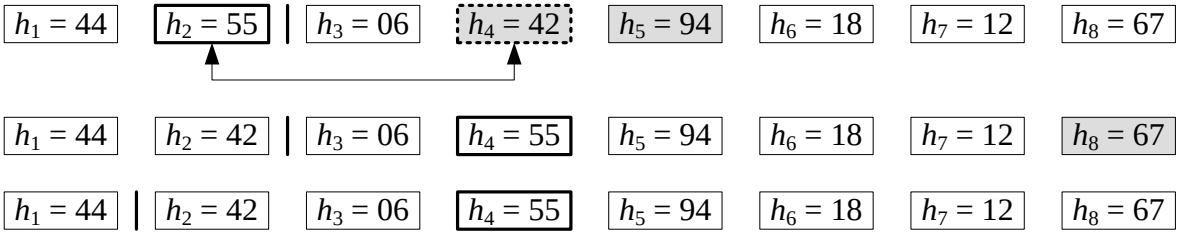


Рис.15

Просеиваем элемент $h_1 = 44$. Он сравнивается с меньшим среди $h_2 = 42$ и $h_3 = 06$; поскольку $h_1 > h_3$, то h_1 обменивается с h_3. Потом элемент $h_3 = 44$ сравнивается с меньшим среди $h_6 = 18$ и $h_7 = 12$; поскольку $h_3 > h_7$, то h_3 обменивается с h_7 и все элементы h_i, $i = 1, 2, 3, 4, 5, 6, 7, 8$, образуют пирамиду (рис. 16).

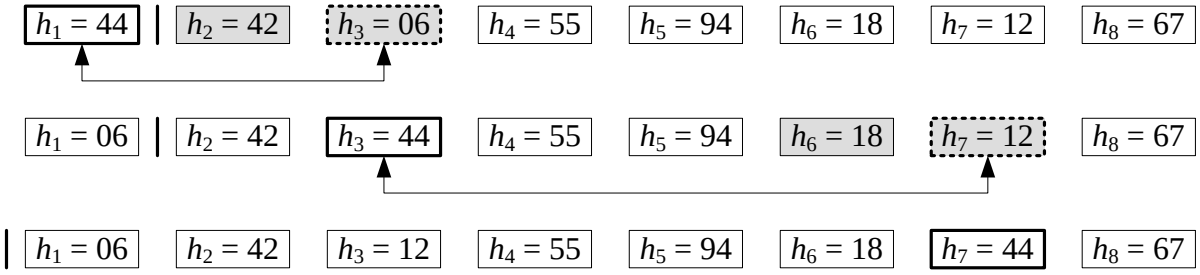


Рис. 16

Первый этап построения пирамиды для массива $A = \{44, 55, 12, 42, 94, 18, 06, 67\}$, который отражен на 13–16, компактно представлен на Рис. 17.

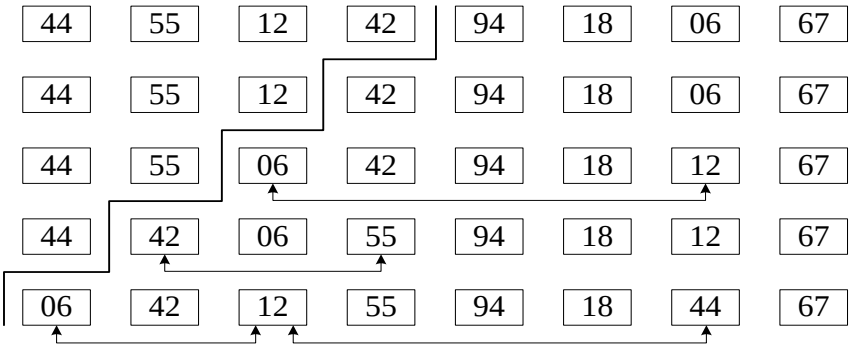


Рис. 17

Этим мы сформировали пирамиду, и наименьший элемент переместился наверх. Далее будем обменивать его согласно этапу 2. Обменяем первый элемент с последним и просеем его через пирамиду, не затрагивая последний (рис. 18).

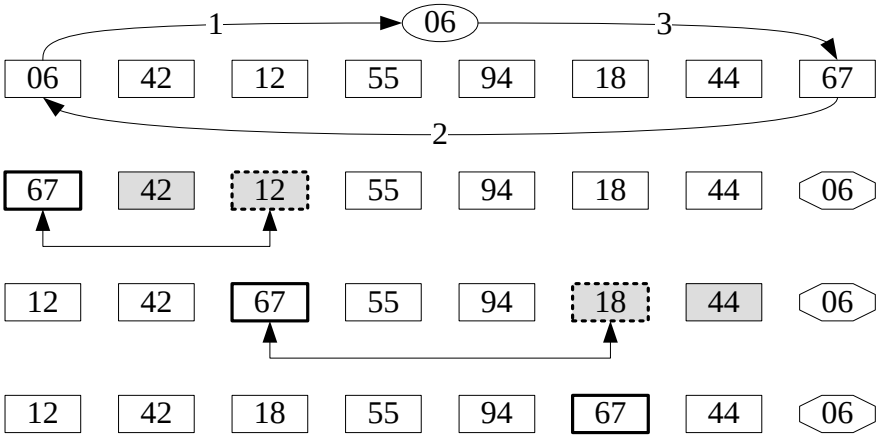


Рис. 18

Обозначения на 18–24 совпадают с приведенными в табл. 1. Дополнительно используются ⬭ – для вспомогательного элемента, который используется для обмена первого элемента массива и последнего неотсортированного, ⯃ – для отсортированных элементов массива *A*.

Новый первый элемент обменяем с предпоследним и просеем его через пирамиду, не затрагивая двух последних (рис. 19).

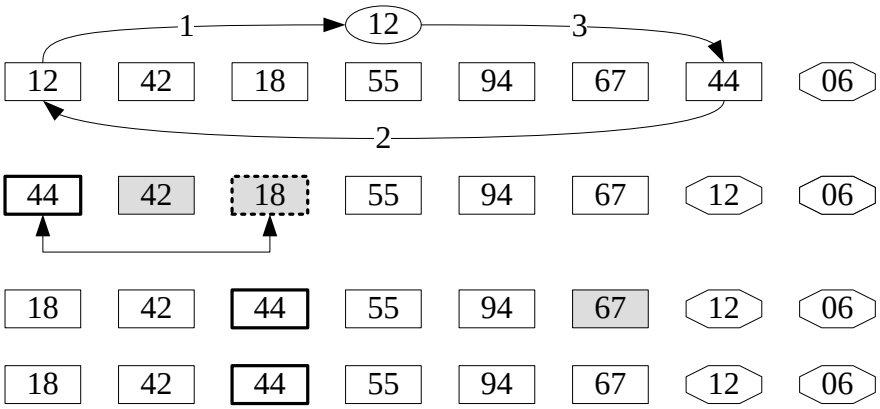


Рис. 19

Новый первый элемент обменяем с последним неотсортированным (с шестым) и просеем его через пирамиду, не затрагивая трех последних (рис. 20).

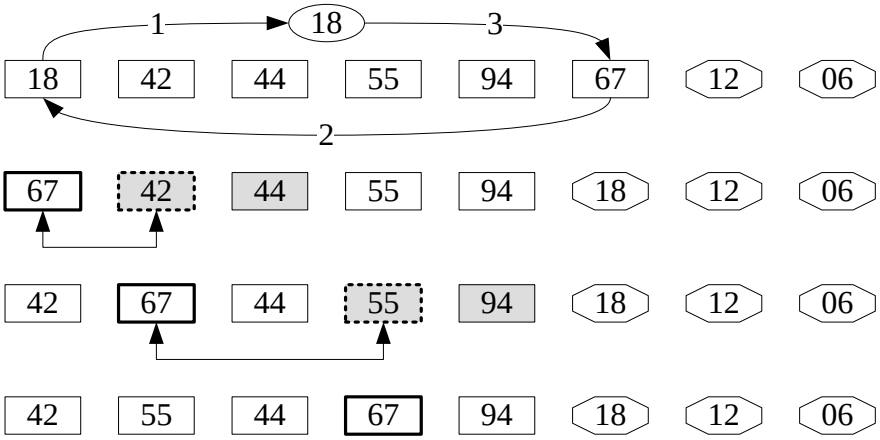


Рис. 20

Новый первый элемент обменяем с последним неотсортированным (с пятым) и просеем его через пирамиду, не затрагивая четырех последних (рис. 21).

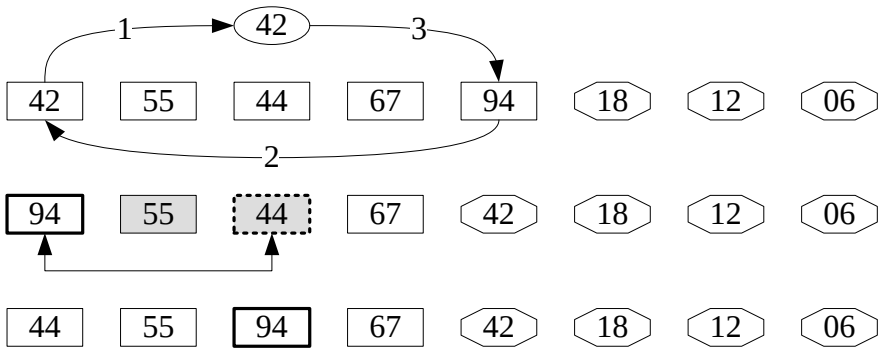


Рис. 21

Новый первый элемент обменяем с последним неотсортированным (с четвертым) и просеем его через пирамиду, не затрагивая пяти последних (рис. 22).

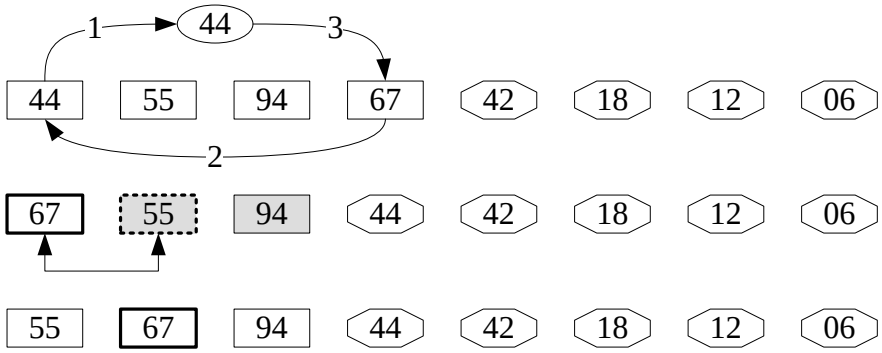


Рис. 22

Новый первый элемент обменяем с последним неотсортированным (с третьим) и просеем его через пирамиду, не затрагивая шести последних (рис. 23).

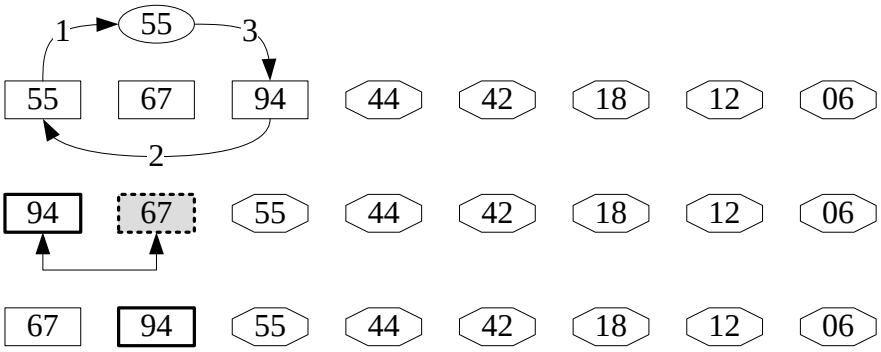


Рис. 23

Новый первый элемент обменяем с последним неотсортированным (со вторым) и получим отсортированный массив (рис. 24).

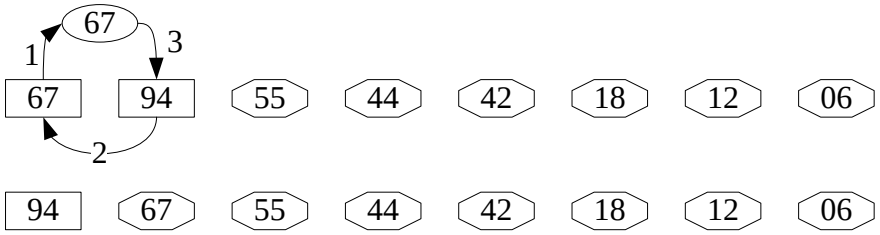


Рис. 24

Получили упорядоченный массив по убыванию значений. Значит, чтобы получить упорядочение в другую сторону, нужно изменить условие (*) на условие (**).

Напишем процедуру просеивания. Заданы: a_l, $a_{l+1}, ..., a_r$. Просеем a_l через пирамиду $a_{l+1}, ..., a_r$.

```
procedure Sift(L,r:index);
var
      j,i : index;
        x : item;
begin
 x := a[L]; i := L; j := 2 * i;
 if (j < r) and (a[j + 1].key < a[j].key)
 then j := j + 1;
             {чтобы дальше сравнивать с меньшим}
 while (j <= r) and (x > a[j].key) do
                    {  подъем вверх}
     begin
      a[i] := a[j]; i := j; j := i * 2;
      if (j < r) and (a[j + 1].key < a[j].key)
      then j := j + 1;
     end;
 a[i] : = x;
end; {Sift}
```

Мы умеем строить пирамиду из n элементов, при этом наименьший элемент перемещается алгоритмом в a_1.

Напишем фрагмент программы построения пирамиды.

```
L := (n div 2) + 1;
 while (L > 1) do
      begin
       L := L - 1;
       Sift(L, n);
      end;
```

Для того чтобы получить не только частичную, но и полную упорядоченность среди элементов, нужно сделать n шагов, причем тот элемент, который переносится

каждый раз на вершину дерева, есть очередной наименьший элемент. Мы их будем отправлять в конец последовательности и не рассматривать в очередной проход, а очередной последний в свою очередь просеивать через пирамиду.

Получим такой алгоритм сортировки по убыванию элементов.

Пирамидальная сортировка

```
procedure HeapSort;
var
      L, r : index;
      y    : item;

  procedure  Sift(L, r: index);
  var
      j, i : index;
      x    : item;
  begin
   x := a[L]; i := L; j := 2 * i;
   if (j < r) and (a[j + 1].key < a[j].key)
   then j := j + 1;
   while (j <= r) and (x > a[j].key) do
     begin
      a[i] := a[j];
      i    := j;
      j    := i * 2;
      if (j<r) and (a[j+1].key<a[j].key) then Inc(j);
     end;
   a[i] := x;
  end; {Sift}

begin
 L := (n div 2) + 1;
 while (L > 1) do
      begin
       L := L - 1;
       Sift(L,n);
      end;                    {построили пирамиду}
 r := n;
 while r >1 do
     begin
      y    := a[1];
      a[1] := a[r];
      a[r] := y;
      r    := r - 1;
      Sift(1,r);
     end;
end;  {HeapSort}
```

Анализ пирамидальной сортировки. Пирамидальную сортировку (англ. *Heapsort*, «Сортировка кучей») рекомендуется использовать для больших n. Среднее число пересылок приблизительно равно $1{,}5 \cdot n \cdot \log_2 n$.

Недостатки алгоритма:

- Неустойчив – для обеспечения устойчивости нужно расширять ключ.
- На почти отсортированных массивах работает столь же долго, как и на хаотических данных.
- На одном шаге выборку приходится делать хаотично по всей длине массива – поэтому алгоритм плохо сочетается с кэшированием и подкачкой памяти.
- Методу требуется «мгновенный» прямой доступ; не работает на связанных списках и других структурах памяти последовательного доступа.

Из-за сложности алгоритма выигрыш получается только на больших n. На небольших n (до нескольких тысяч) быстрее сортировка Шелла.

Некоторые другие методы сортировок

«Карманная» сортировка

Это сортировка массивов за линейное время.

Если исходная последовательность чисел $a_1, a_2, ..., a_n$ является последовательностью натуральных чисел, и они не повторяются, то можно предложить для их сортировки по возрастанию следующий подход. Берется вспомогательный массив $b_1, \ldots, b_n$ и организуется такой цикл:

```
for i := 1 to n do
    B[A[i]] := A[i];
```

Этот код подсчитывает, где в массиве B должен находиться элемент `A[i]`, и помещает его туда. Весь этот цикл требует времени порядка $O(n)$ и работает корректно только тогда, когда значения всех ключей различны и являются натуральными числами из интервала от 1 до n.

Если же не разрешается использовать вспомогательный массив B, то организовывается такой цикл:

```
for i := 1 to n do
    while A[i] <> i do
         begin
          обмен A[i] и A[A[i]]
         end;
```

Этот алгоритм также требует времени порядка $O(n)$. По очереди проверяем элементы $a_1, a_2, \ldots, a_n$. Если в a_i стоит число $j \neq i$, то меняются местами элементы a_i и a_j. Если после этой перестановки новое число в a_i имеет значение $k \neq i$, то совершается перестановка a_i и a_k элемента. Каждая перестановка смещает хотя бы один элемент в нужное место.

Сортировка элементов методом подсчета

Это сортировка массивов за линейное время.

Каждый элемент сравнивается с остальными элементами. Если какой-то элемент больше, чем k элементов этого же массива, то после упорядочения он должен занимать $(k + 1)$ место. Значит, необходимо сравнивать попарно все элементы и подсчитывать в отдельном массиве, сколько из них меньше за каждый текущий элемент. После этого необходимо переставить элементы в соответствии с вычисленными местами.

Introsort или интроспективная сортировка

Introsort или **интроспективная сортировка** – алгоритм сортировки, предложенный Дэвидом Мюссером в 1997 году. Он использует быструю сортировку и переключается на пирамидальную сортировку, когда глубина рекурсии превысит некоторый заранее установленный уровень (например, логарифм от числа сортируемых элементов). Этот подход сочетает в себе достоинства обоих методов с худшим случаем $O(n \log n)$ и быстродействием, сравнимым с быстрой сортировкой. Так как оба алгоритма используют сравнения, этот алгоритм также принадлежит классу сортировок на основе сравнений.

Мюссер выяснил, что на худшем наборе данных для алгоритма быстрой сортировки «медиана из трёх» (рассматривался массив из 100 тысяч элементов) *introsort* работает примерно в 200 раз быстрее.

Timsort

Timsort – гибридный алгоритм сортировки, сочетающий сортировку вставками и сортировку слиянием, опубликованный в 2002 году Тимом Петерсом.

Основная идея алгоритма в том, что в реальном мире сортируемые массивы данных часто содержат в себе упорядоченные подмассивы. На таких данных Timsort существенно быстрее многих алгоритмов сортировки.

Основная идея алгоритма:

- По специальному алгоритму входной массив разделяется на подмассивы.
- Каждый подмассив сортируется сортировкой вставками.
- Отсортированные подмассивы собираются в единый массив с помощью модифицированной сортировки слиянием.

Принципиальные особенности алгоритма в деталях, а именно в алгоритме разделения и модификации сортировки слиянием.

Терпеливая сортировка

Терпеливая сортировка (англ. *patience sorting*) – алгоритм сортировки с худшей сложностью $O(n \log n)$. Позволяет также вычислить длину наибольшей

возрастающей подпоследовательности данного массива. Алгоритм назван по одному из названий карточной игры "Солитёр" – "Patience".

Алгоритм:

Имеем массив *A,* элементы которого нужно отсортировать по возрастанию. Разложим элементы массива по стопкам: для того чтобы положить элемент в стопку, требуется выполнение условия – новый элемент меньше элемента, лежащего на вершине стопки; либо создадим новую стопку справа и сделаем наш элемент её вершиной. Используем жадную стратегию: каждый элемент кладётся в самую левую стопку из возможных, если же таковой нет, справа от существующих стопок создаётся новая. Для получения отсортированного массива сначала построим массив стопок, затем выполним n шагов (здесь и далее нумерация шагов начинается с единицы): на i-м шаге выберем из всех вершин стопок наименьшую, извлечём её и запишем в массив B на i-ю позицию.

Плавная сортировка

Плавная сортировка (англ. *Smoothsort*) – алгоритм сортировки выбором, разновидность пирамидальной сортировки, разработанная Э. Дейкстрой в 1981 году. Как и пирамидальная сортировка, имеет сложность в худшем случае равную $O(n \log n)$. Преимущество плавной сортировки в том, что её сложность приближается к $O(n)$, если входные данные частично отсортированы, в то время как у пирамидальной сортировки сложность всегда одна, независимо от состояния входных данных.

Общее представление:

Как и в пирамидальной сортировке, в массиве накапливается куча из данных, которые затем сортируются путём непрерывного удаления максимума из кучи. В отличие от пирамидальной сортировки, здесь используется не двоичная куча, а специальная, полученная с помощью чисел Леонардо. Куча состоит из последовательности куч, размеры которых равны одному из чисел Леонардо, а корни хранятся в порядке возрастания. Преимущества таких специальных куч перед двоичными состоят в том, что если последовательность отсортирована, её создание и разрушение займёт $O(n)$ времени, что будет быстрее.

Гномья сортировка

Гномья сортировка (англ. *Gnome sort*) – алгоритм сортировки, похожий на сортировку вставками, но в отличие от последней перед вставкой на нужное место происходит серия обменов, как в сортировке пузырьком. Название происходит от предполагаемого поведения садовых гномов при сортировке линии садовых горшков.

Параллельная сортировка Бэтчера

Параллельно в первой и во второй половинах выполняется сортировка простым обменом, а затем идет слияние отсортированных массивов.

Сортировка методом прочесывания

В сортировке пузырьком или шейкер-сортировке, когда сравниваются два элемента, промежуток (расстояние друг от друга) равен 1. Основная идея метода прочесывания в том, чтобы первоначально брать достаточно большое расстояние между сравниваемыми элементами и по мере упорядочивания массива сужать это расстояние вплоть до минимального. Таким образом, мы как бы причёсываем массив, постепенно разглаживая на всё более аккуратные пряди. Сортировка также основана на этой идее, но она является модификацией сортировки вставками, а не сортировки пузырьком.

Сортировка методом прочесывания или расчёской (англ. *comb sort*) – это довольно упрощённый алгоритм сортировки, изначально спроектированный В. Добосевичем в 1980 г.

Позднее он был переоткрыт и популяризован в статье Стивена Лэйси и Ричарда Бокса в 1991 г.

Сортировка расчёской улучшает сортировку пузырьком, и конкурирует с алгоритмами, подобными быстрой сортировке. Основная идея – устранить черепах, т. е. маленькие значения в конце списка, которые крайне замедляют сортировку пузырьком (кролики, большие значения в начале списка, не представляют проблемы для сортировки пузырьком).

Сравнение методов сортировки массивов *in situ*

Мы получили следующие алгоритмы.

I. Обменные сортировки.
   1. Метод простого обмена.
   2. Метод простого обмена и выход, если нет обмена.
   3. Метод простого обмена не с начала, а от последнего обмена.
   4. Шейкер-сортировка.
   5. Быстрая сортировка.
   6. Гномья.
   7. Расчёской.

II. Сортировка вставками.
   1. Сортировка простыми вставками.
   2. Сортировка простыми вставками с бинарным поиском.
   3. Сортировка Шелла.

III. Сортировка выбором.
   1. Сортировка простым выбором.
   2. Пирамидальная сортировка.
   3. Плавная.

IV. Сортировка слиянием.

V. Без сравнений.
   1. Подсчётом.
   2. Поразрядная.
   3. Блочная.

VI. Гибридные.
1. Introsort.
2. Timsort.

Попробуем сравнить их эффективность. Для усовершенствованных методов нет сколько-нибудь осмысленных простых и точных оценок (формул). Существенно, что для сортировки Шелла вычислительные затраты составляют $O(n^{1,2})$, а для *QuickSort* (быстрой сортировки) и *HeapSort* (пирамидальной) – $O(n \log_2 n)$. Эксперимент показывает, что *Quicksort* лучше в 2-3 раза по сравнению с *HeapSort*. Эти оценки позволяют разбить алгоритмы на методы продуктивности порядке $O(n^2)$ и на усложнённые порядка $O(n \log n)$.

Для сравнения алгоритмов сортировки удобно использовать матричный метод, который предполагает сравнение по двум параметрам. Так, в зависимости от устойчивости и скорости алгоритмы сортировки можно разделить на шесть групп (таблица 20).

Таблица 20 – Сравнения алгоритмов сортировки.

	$<n^2$	**n^2**	**$> n^2$**
Устойчивые	• Сортировка с помощью двоичного дерева. • Сортировка слиянием. • Сортировка методом Тима Петерса.	• Сортировка методом простого обмена. • Сортировка методом простого обмена и выход, если нет обмена. • Сортировка методом простого обмена не с начала, а от последнего обмена. • Шейкер-сортировка. • Сортировка вставками. • Сортировка выбором.	• Гномья сортировка.
Неустойчивые	• Сортировка Шелла. • Сортировка расчёской. • Пирамидальная сортировка. • Быстрая сортировка. • Интроспективная сортировка. • Терпеливая сортировка.	• Сортировка выбором.	

Классификация алгоритмов сортировки

Алгоритмы устойчивой сортировки

- Сортировка пузырьком (англ. *Bubble sort*).
- Сортировка перемешиванием (англ. *Cocktail sort*).
- Сортировка вставками (англ. *Insertion sort*).

- Гномья сортировка (англ. *Gnome sort*); первоначально опубликована под названием «глупая сортировка» (англ. *stupid sort*) за простоту реализации.
- Сортировка слиянием (англ. *Merge sort*).
- Сортировка с помощью двоичного дерева (англ. *Tree sort*).
- Сортировка Timsort (англ. *Timsort*).
- Сортировка подсчётом (англ. *Counting sort*).

Алгоритмы неустойчивой сортировки

- Сортировка выбором (англ. *Selection sort*).
- Сортировка расчёской (англ. *Comb sort*).
- Сортировка Шелла (англ. *Shell sort*).
- Пирамидальная сортировка (англ. *Heapsort*) .
- Плавная сортировка (англ. *Smoothsort*).
- Быстрая сортировка (англ. *Quicksort*).
- Интроспективная сортировка (англ. *Introsort*).
- Терпеливая сортировка (англ. *Patience sorting*).**Эволюция способов и алгоритмов сортировки**

Выделяют следующие пять этапов в эволюции способов и алгоритмов машинной сортировки данных в массивах.

Первый этап начался в 1870 году и длился до начала 1940-х годов. Его ознаменовал переход от ручной сортировки к сортировке с помощью статистических табуляторов. При этом использовался алгоритм поразрядной сортировки.

Второй этап – с начала 1940-х годов до середины 1950-х. На смену счетно-перфорационным машинам пришли ЭВМ первого поколения, для которых был разработан ряд новых алгоритмов сортировки.

Третий этап начался в середине 1950-х годов и продолжался до середины 1970-х. Активное развитие алгоритмов сортировки началось с разработкой ЭВМ второго поколения (увеличилась производительности компьютеров до 30 тысяч операций в секунду, резко уменьшились их габариты и стоимость). Разработка первых языков программирования высокого уровня (Фортран, Алгол, Кобол) создало предпосылки для ускорения написания программ для ЭВМ.

Четвертый этап продолжался с середины 1970-х до середины 1990-х годов. Появление вычислительных центров, объединяющих мощности отдельных вычислительных машин и позволяющих работать с разделением времени потребовало разработки новых алгоритмов сортировки и модификации существующих. Началось исследование задач сортировки в классе параллельных алгоритмов, были достигнуты значительные успехи в увеличении скорости сортировки за счет повышения эффективности уже известных к тому времени алгоритмов путем их доработки или комбинирования.

Пятый этап начался с середины 1990-х годов и продолжается по настоящее время. Особую актуальность получило исследование задач сортировки на частично упорядоченных множествах. Актуальность этих задач объясняется появлением и широким распространением компьютеров на сверхсложных микропроцессорах с параллельно-векторной структурой, а также высокоэффективных сетевых компьютерных систем.

1.7. Сортировка последовательных файлов

Простое слияние

Когда происходит сортировка массивов в оперативной памяти, тогда имеется доступ к любому элементу массива. Что нельзя сказать в случае последовательного файла, так как в каждый момент времени существует доступ только к одному элементу. Такое ограничение строгое по сравнению с возможностями, которые дает массив, и поэтому здесь приходится применять другие методы сортировки. Основной метод – сортировка слиянием. Слияние означает объединение двух (или более) упорядоченных последовательностей в одну упорядоченную последовательность.

Рассмотрим один из методов сортировки слиянием – *простое слияние*.

1. Последовательность *A* разбивается на две половины – *B* и *C*.

2. Последовательности *B* и *C* сливаются при помощи объединения отдельных элементов в упорядоченные пары.

3. Полученной последовательности дается имя *A*, и повторяются шаги 1 и 2; на этот раз упорядоченные пары сливаются в упорядоченые четверки.

4. Предыдущие шаги повторяются: четверки сливаются в восьмерки, и весь процесс продолжается до тех пор, пока не будет упорядочена вся последовательность, так как длина последовательностей, которые сливаются, каждый раз удваивается.

Рассмотрим этот алгоритм на нашем примере:

44	55	12	42	|	94	18	06	67	{пополам и пары}
44	94	18	55	|	06	12	42	67	{пополам и четверки}
06	12	44	94	|	18	42	55	67	{пополам и восьмерки}
06	12	18	42		44	55	67	94	{все}

Получили 3 прохода (элементов $n = 8 = 2^3$).

Для выполнения сортировки требуются три последовательности, поэтому процесс называется *трехленточным слиянием*. Исторически последовательные файлы хранились на магнитных лентах.

Далее можно улучшать этот алгоритм и получать различные модификации.

Каждый проход состоит из фазы разделения и фазы слияния. Фазы разделения не относятся к сортировке непосредственно, потому что это просто переписывание элементов; в каком-то смысле они непродуктивны, хотя и составляют половину всех операций перезаписи.

Их предложили уничтожить, объединив фазы разделения и фазы слияния так: результат слияния сразу распределяется на две ленты, которые на следующем проходе будут входными. Это уже однофазное, или

сбалансированное, слияние. Оно требует вдвое меньше операций перезаписи, но это достигается за счет использования четвертой ленты.

Анализ сортировки слиянием. Поскольку на каждом проходе p длина подпоследовательностей, которые объединяются слиянием, удваивается и сортировка заканчивается, как только $p >= n$, то понадобится $\log_2 n$ проходов. По определению при каждом проходе все множество из n элементов копируется только один раз, значит, общее количество пересылок равна $M = [\log_2 n] \cdot n$.

Количество сравнений примерно такое же, так как при копировании остатка последовательности (если есть остаток) сравнение не проводится.

Этот алгоритм хорошо переводится и на сортировку массива, который будет просматриваться строго последовательно.

Естественное слияние

В случае простого слияния ничего не выигрывается, когда данные уже частично отсортированы. Фактически можно было бы сразу сливать какие-либо упорядоченные подпоследовательности длин m и n в одну последовательность из $m + n$ элементов.

Метод сортировки, при котором каждый раз сливаются две самые длинные возможные подпоследовательности, называется *естественным слиянием.*

Упорядоченную подпоследовательность назовем *серией*. Это такие элементы a_i, a_2, ..., a_j, которые удовлетворяют условиям:

$$a_{i-1} > a_i; \quad a_k <= a_{k+1}, \quad k = i, \ldots, j-1; \quad a_j > a_{j+1}.$$

Значит, сортировка естественным слиянием сливает не последовательности фиксированной, ранее заданной длины, а максимальные серии. Таким образом, на каждом проходе общее число серий уменьшается вдвое, и число необходимых пересылок элементов в худшем случае $[\log_2 n] \cdot n$, а в обычном случае меньше.

Сбалансированное многоленточное слияние

Затраты на последовательную сортировку пропорциональны числу проходов, так как по определению на каждом проходе происходит переписывание всего множества данных. Один из способов уменьшить это число – распределять серии более чем на две ленты. Слияние r серий, которые поровну распределены на N лентах, дает в результате последовательность из r / N серий.

Здесь мы имеем массив файлов. За второй проход число серий уменьшается до r / N^2, и после k проходов остается r / N^k отрезков. Все они каждый раз распределяются на N лент.

Таким образом, общее число проходов при сортировке n элементов N-путевым слиянием $k = [\log_N n]$. Поскольку на каждом проходе происходит N операций перезаписи, то общее число операций перезаписи в худшем случае будет $M = [\log_N n] \cdot n$.

На Рис. 25 покажем наиболее известные алгоритмы внешних сортировок.

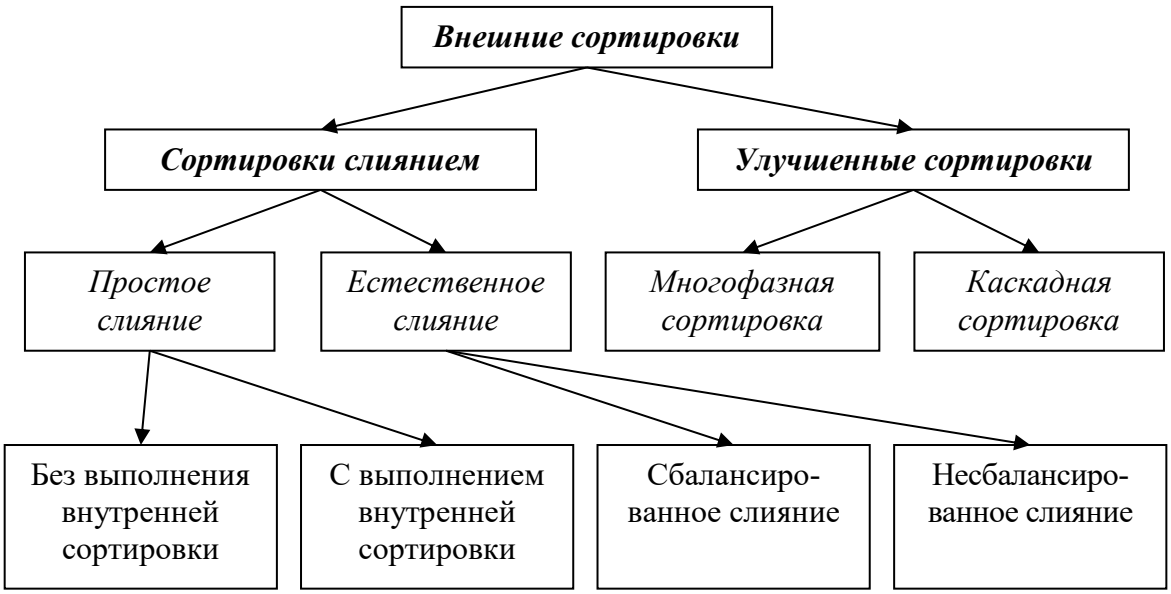


Рис. 25

1.8. Объектно-ориентированное программирование

Структурная программа состоит из совокупности подпрограмм, связанных с помощью интерфейса. Подпрограммы работают с данными, которые или являются локальными, или передаются им в качестве параметров.

Ошибки часто связаны с тем, что в подпрограмму передаются неверные данные. Естественный путь избежать таких ошибок – связать в единое целое данные и все подпрограммы, которые предназначены для их обработки, и назвать эту совокупность новым типом – объектом.

Объектно-ориентированное программирование основано на трех важнейших принципах, придающих объектам новые свойства. Этими принципами являются инкапсуляция, наследование и полиморфизм.

Инкапсуляция есть объединение в единое целое данных и алгоритмов обработки этих данных.

Инкапсуляция позволяет при необходимости изменять реализацию объекта без модификации основной части программы, если его интерфейс остался тем же. Простота же модификации программы – важный критерий качества программы.

Наследование есть свойство объектов порождать потомков. Объект-потомок автоматически получает от родителя все поля и методы, и может дополнять их.

Наследование позволяет создать иерархию объектов. В языке Pascal любой объект может иметь одного родителя и произвольное количество потомков. Иерархия представляется в виде дерева, в котором общие объекты располагаются ближе к корню, а специализированные – на ветках и листьях.

В рамках ООП поведенческие свойства объекта определяются набором методов, входящих в объект. Изменяя алгоритм того или иного метода в потомках объекта, программист может дать этим потомкам специфические свойства, отсутствующие у родителя. Для изменения метода нужно перекрыть

его в потомке, т. е. объявить в потомке одноименный метод и реализовать в нем нужные действия. В результате в объекте-родителе и объекте-потомке будут действовать два одноименных метода, имеющие разную алгоритмическую основу и, таким образом, дающие объектам различные свойства.

Полиморфизм – это свойство родственных объектов (т. е. объектов, имеющих одного общего родителя) решать схожие по смыслу проблемы разными способами.

В Borland Pascal полиморфизм расширяется виртуализацией метода (`virtual` – эффективный), когда из родительских методов обращаются к методу потомков. Объявление метода виртуальным дает возможность дочернему классу произвести замену виртуального метода своим собственным. Этот механизм рассмотрим позже.

Применение ООП позволяет писать гибкие программы, которые легко расширяются и читаются. Во многом это обеспечивается благодаря изменчивости объектов, т. е. такому свойству как полиморфизм, когда имеется возможность при наследовании менять реализацию методов так, что они будут иметь один и тот же интерфейс и разную алгоритмическую реализацию.

Механизм объявления объектов

Объектный тип может быть определен как полный, самостоятельный тип, подобный описанию записей в Pascal, но он может определяться и как потомок существующего типа объекта.

Объект – это тип данных, поэтому он определяется в разделе описания типа с использованием служебного слова `object`. В других языках объектный тип называют классом. Объект похож на тип `record`, но кроме *полей данных*, в нем можно описывать *заголовки методов*.

Методами называются подпрограммы, предназначенные для работы с полями объекта.

Поля объекта описываются аналогично обычным переменным: для каждого поля задается его имя и тип. Значения полей определяют состояние объекта.

Поля и ***методы называются элементами объекта***.

Для реализации принципа инкапсуляции, т. е. ограничения видимости элементов, объекты обычно описывают в модулях. В интерфейсной части модуля описывается тип объекта, а тела методов объектов описываются в разделе реализации.

Хранение описаний в объектах

Согласно принципу инкапсуляции во многих случаях требуется ограничение доступа к составляющим объект компонентам (методам и переменным). Доступ к полям объектов напрямую нежелателен. Например, объект описывает ноутбук и содержит в том числе поля характеризующие его размер. Очевидно, что значения длины, ширины и толщины корпуса должны быть положительны, и более того находиться в некотором диапазоне, поскольку не существует ноутбуков толщиной 2 метра или шириной 3 миллиметра. Для того, чтобы пользователь мог изменить размеры устройства должны быть реализованы методы для изменения параметров,

внутри которых новое значение должно проверяться на пригодность, т. е. валидироваться. В таких случаях предусматриваются объекты, в которых есть открытые (публичные, доступные) поля и методы и такие поля и методы, прямой доступ к которым запрещен, – закрытые (тайные, личные) поля и методы.

Видимостью элементов объекта управляют директивы `public` и `private`. В объекте может быть произвольное количество разделов `public` и `private`. По умолчанию все элементы объекта считаются видимыми извне, т. е. `public`. Действие директивы распространяется до другой директивы или до конца объекта.

Поля и методы, которые идут сразу за заголовком объектного типа или за директивой `public`, не имеют никаких ограничений на область действий. Поля и методы, объявленные после директивы `private`, считаются закрытыми (собственными, личными) и ограничены использованием в пределах модуля.

Полное описание объекта примерно следующее:

```
type
      NEWObject = object
        Поля;   {Открытые}
        Методы; {Открытые}
      private
        Поля;   {Закрытые}
        Методы; {Закрытые}
      public
        Поля;   {Открытые}
        Методы; {Открытые}
                 end
```

С наследованием:

```
type
     NEWObject=object(Имя_объекта_родителя)
        Поля;   {Открытые}
        Методы; {Открытые}
      private
        Поля;   {Закрытые}
        Методы; {Закрытые}
      public
        Поля;   {Открытые}
        Методы; {Открытые}
      end
```

Каждое действие, которое должен выполнять объект, оформляется в виде отдельной процедуры или функции, т. е. метода.

Описания реализаций методов (текст подпрограмм) размещаются вне объекта в разделе описания процедур и функций.

При определении содержимого методов исходят из нужного поведения объекта.

При объявлении объектов должны соблюдаться следующие требования:

- описание типа объект может происходить только в секции **type** основной программы или в разделах модуля; нельзя описывать локальные объекты в подпрограммах;
- при описании типа объекта в каждом разделе все поля данных должны находиться перед описаниями методов;
- компоненты объекта не могут быть файлами, а файлы, в свою очередь, не могут содержать компоненты типа «объект»;
- при наследовании полей в дочерних типах уже нельзя объявлять их идентификаторы, определенные в одном из родительских типов, однако дочерние объекты могут переопределять любой из наследуемых методов.

Область действия полей данных объекта неявно распространяется на тело процедур и функций, реализующих методы этого объекта.

В случае если в дочернем типе описывается новая процедура инициализации, в ней обычно сначала вызывается процедура родительской инициализации. Это самый естественный способ проинициализировать поля, полученные в наследство.

Механизм определения метода

В объектах (в разделах `private` или `public`) сначала определяются поля объекта, а затем методы.

Параметры метода имеют уникальные имена, которые не совпадают с именами всех полей, как собственных, так и наследованных. Сами методы описываются вне определения объекта как отдельная процедура или функция, при этом имя метода дополняется именем типа объекта, которому принадлежит метод, с последующей точкой, список параметров можно опускать.

Замечание. Фактически имя метода – сложное:

```
имя_объекта.имя_метода
```

Значит, в программе может быть объявлена другая подпрограмма, имя которой будет совпадать с именем метода, и здесь не будет ошибки.

Переопределение методов

Наследование дочерними типами информационных полей и методов их родительских типов выполняется соответственно следующим правилам:

Правило 1. Информационные поля и методы родительского типа наследуются всеми его дочерними типами независимо от количества промежуточных уровней иерархии.

Правило 2. Доступ к полям и методам родительских типов в рамках описания любых дочерних типов выполняется так, будто они описаны в самом дочернем типе.

Правило 3. Нельзя в дочерних типах использовать идентификаторы полей, совпадающие с идентификаторами полей какого-либо родительского типа.

Правило 4. Дочерний тип может доопределять любое количество собственных методов и информационных полей.

Правило 5. Любое изменение кода в родительском методе автоматически влияет на все методы порожденных дочерних типов, которые его вызывают.

Правило 6. Идентификаторы метода в дочерних типах могут совпадать с именами метода в родительских типах. Тогда дочерний метод подавляет тождественный ему родительский, и в рамках дочернего типа при указании имени такого метода будет вызываться именно дочерний метод. Для вызова родительского метода перед именем метода нужно написать

```
<имя_родительского_объекта.метод>
```

или использовать конструкцию **`inherited`** (наследуемый) метод.

Важным аспектом наследственности являются *правила вызова наследуемых методов*:

1. При вызове метода компилятор сначала ищет метод, имя которого определено внутри типа объекта.

2. Если в типе объекта не определен метод с указанным в операторе вызова именем, тогда компилятор в поисках метода с таким именем поднимается выше к родительскому или прародительскому типу. Вызовы метода с ниже размещенных по иерархии типов (дочерних) не разрешаются.

3. Если наследуемый метод найден и его адрес подставлен, нужно помнить, что метод, который вызывается, будет работать так, как он определен в родительском типе. И если этот наследуемый родительский метод вызывает еще и другие методы, тогда вызываться уже будут только родительские или методы предков.

Раннее связывание

После компиляции и запуска программы ее исполняемые операторы в виде инструкций – команд процессору, находятся в сегменте кода. Каждая подпрограмма имеет точку входа. *При компиляции вызов подпрограммы заменяется последовательностью команд, передающих управление в эту точку входа*.

Этот механизм называется *ранним связыванием*, так как все ссылки на подпрограммы, компилятор строит до выполнения программы.

Очевидно, что с помощью раннего связывания не удается обеспечить возможность вызова из одной и той же подпрограммы метода то одного объекта, то другого.

В Pascal существует и другой механизм вызова методов, когда ссылки определяются уже на этапе выполнения программы в момент вызова метода. Такой механизм реализуется с помощью, так называемых, виртуальных методов и называется ***поздним связыванием***.

Возможность использования методов то одного объекта, то другого нарушает правило соответствия типа по присваиванию, которое мы изучали до этого. Для объектов понятие совместимости расширено.

Экземпляры объектов

Переменная объектного типа называется *экземпляром* объекта.

Часто экземпляры просто называют объектами.

Экземпляры объектов можно создавать и в статической и в динамической памяти, можно определять массивы экземпляров объектов, или указателей на объекты, или другие структуры данных.

Доступ к элементам экземпляра объекта осуществляется либо с использованием составного имени, когда указывается *имя экземпляра объекта* и *через точку имя поля* или *имя метода*, либо с помощью оператора присоединения `with`.

Перед использованием элементов объекта требуется проинициализировать его поля и, возможно, выполнить другие подготовительные действия. Метод, выполняющий такие действия, обычно называют именем `Init`.

Если объект описан в модуле, получить или изменить значения элементов с директивой `private` в программе можно только через обращение к соответствующим методам.

При создании каждого экземпляра объекта выделяется память, достаточная для хранения всех его полей.

Коды методов объекта хранятся в одном экземпляре. Для того чтобы методу было известно, с данными какого экземпляра объекта он работает, при вызове ему в неявном виде передается параметр `self`, который и определяет место размещения данных этого объекта.

Фактически внутри метода обращение к полю «x» объекта имеет вид `self.x` (`@self` представляет собой адрес начала области оперативной памяти, в которой хранятся поля объекта).

Экземпляры объектов одного типа *можно присваивать друг другу, при этом выполняется копирование всех полей*. Позже будем рассматривать правила расширенной совместимости типов объектов.

Совместимость объектных типов

Наследование несколько изменяет правила совместимости типов в Turbo Pascal: порожденный тип наследует совместимость со всеми родительскими типами. Другими словами: объекты дочерних типов могут свободно использоваться вместо объектов родительских, но не наоборот.

Совместимость объектных типов бывает трех видов:

- между экземплярами объектов;
- между указателями на экземпляры объектов;
- между формальными и фактическими параметрами.

Во всех трех случаях совместимость односторонняя: *родительскому экземпляру объекта может быть присвоен экземпляр такого же типа или любого из его потомков*.

Совместимость между экземплярами объектов

Смысл совместимости объектов в том, что в результате присваивания значения все информационные поля объекта приемника, который стоит в левой части оператора присвоения значения, должны получить значения. Объект же потомка может иметь и свои собственные поля.

Из источника в приемник будут копироваться *только те поля*, которые являются общими и для объекта родительского типа, и для объекта дочернего типа.

Совместимость между указателями на экземпляры объектов

Указатель на объект дочернего типа может быть присвоен указателю на родительский тип, т. е. совместимость типов работает также для указателей на типы объектов, *при этом тип вызываемого метода объекта соответствует типу указателя*, а не типу того объекта, на который он ссылается.

Если известно, что указатель на предка в самом деле хранит ссылку на потомка, можно обратиться к элементам, определенным в потомке, с помощью такого приема как явное преобразование типа.

Совместимость между формальными и фактическими параметрами

Если экземпляр объекта является параметром подпрограммы, ему может соответствовать *аргумент того же типа или любого из его наследников*, но есть разница между передачей экземпляров объекта по значению (параметр-значение) и по адресу (параметр-переменная).

Другими словами, формальный параметр (параметр-значение, параметр-переменная) данного объектного типа может принимать в качестве фактического параметра объект своего же типа или объекты всех своих дочерних типов. Но вспомним механизм передачи параметров.

Важно помнить, что параметр, передаваемый по *значению*, представляет собой *копию* экземпляра объекта-аргумента, *приведенную к типу параметра*. Копия содержит только те поля данных и методы, которые соответствуют типу параметра.

При передаче экземпляра объекта по адресу подпрограмме передается указатель (адрес) на фактический экземпляр объекта, т. е. *приведение типов не выполняется*.

Поэтому в подпрограмме тип экземпляра объекта, передаваемого по адресу, может изменяться в зависимости от аргумента.

Замечание. Процитируем еще раз описанное выше свойство: «Указатель на объект дочернего типа может быть присвоен указателю на родительский тип, *при этом тип вызываемого метода соответствует типу указателя*, а не типу того объекта, на который он ссылается».

Это все объясняется тем, что методы являются статическими. Для реализации полиморфизма в Turbo Pascal существуют *виртуальные* методы, которые динамически связываются с экземплярами объектов во время выполнения программы.

Виртуальные методы. Конструкторы

Экземпляры объектов, фактический тип которых может изменяться во время выполнения программы, называются полиморфными. Исходя из предыдущего, можно сделать вывод, что полиморфным может быть экземпляр объекта, определенный через указатель или переданный в подпрограмму по адресу.

Полиморфные объекты обычно применяются вместе с виртуальными методами.

Метод становится виртуальным, если за его определением в типе объекта относится служебное слово `virtual`:

```
procedure имя_метода (параметры);              virtual;
function имя_метода (параметры) : тип_вызова; virtual;
```

При описании реализации метода слово `virtual` уже не ставится.

Различие между вызовом статического и динамического методов заключается в том, что в первом случае компилятору сразу известна связь объекта с методом, и он устанавливает ее на этапе компиляции (*раннее связывание).* Во втором случае компилятор как бы откладывает решение связи до момента выполнения программы (*позднее связывание*).

Для реализации позднего связывания необходимо, чтобы адреса виртуальных методов хранились там, где ими можно будет воспользоваться в любой момент во время выполнения программы. Поэтому *компилятор формирует* для этих методов внутреннюю таблицу виртуальных методов – ТВМ (VMT). В первое поле этой таблицы записывается размер объекта, а затем идут *адреса кодов* процедур или функций, реализующих каждый из его виртуальных методов, как описанных в объекте, так и унаследованных. Такая таблица – *одна для каждого объектного типа*.

Каждый экземпляр объекта *во время выполнения программы* при своем реальном создании должен иметь доступ к ТВМ. Эта связь экземпляра объекта с ТВМ устанавливается с помощью специального метода, называемого *конструктором*.

Значит, объект, имеющий хотя бы один виртуальный метод, должен содержать конструктор – это специальный метод, который инициирует объект. В нем может выполняться выделение памяти под динамические переменные или структуры, если они есть в объекте, и присваиваться начальные значения. Обычно ему дают имя `INIT`.

В этом методе служебное слово `procedure` в объявлении и реализации заменяется словом `constructor`, обозначающее особый вид процедуры – конструктор, который помимо своей собственной работы (которой может вообще и не быть) выполняет установочную работу для механизма виртуальных методов. По ключевому слову `constructor` компилятор вставляется в начало метода фрагмент, который записывает ссылку на ТВМ в специальное поле экземпляра объекта (память под это поле выделяется компилятором).

Конструктор всегда должен вызываться до первого вызова виртуального метода, потому что конструктор устанавливает связь между экземпляром объекта, который вызывает конструктор, и таблицей виртуальных методов данного

объектного типа. Конструктор должен быть *вызван для каждого создаваемого экземпляра объекта*. Присваивание одного экземпляра объекта другому возможно только после конструирования обоих. Если же конструктор не будет вызван перед обращением к виртуальному методу, тогда компьютер не будет знать, где искать этот метод. Это и приведет к фатальной ситуации.

Если программу компилировать с директивой {$R+}, тогда программа сама проверяет корректность объектов, анализируя их размеры. При этом генерируется вызов подпрограммы проверки правильности ТВМ перед каждым вызовом виртуального метода. Если контрольные значения размера в ТВМ указывают на сбой, происходит фатальная ошибка 210. После отладки директиву можно отключить.

Сами *конструкторы* могут быть только *статическими*, хотя внутри конструктора могут вызываться и виртуальные методы. Если в конструкторе есть поля, которые также являются объектами с виртуальными методами, то в теле конструктора вызываются конструкторы этих объектов.

В объекте может быть определено несколько конструкторов. Повторный вызов конструктора вреда программе не принесет.

Правила описания виртуальных методов

Когда объявляется виртуальный метод в каком-нибудь родительском типе, то это накладывает следующие ограничения на его дочерние типы:

- все методы дочерних типов одноименные с виртуальными родительскими также должны быть виртуальными. Статический метод не может подавить виртуальный;
- после того как метод стал виртуальным, его *заголовок не может меняться в объектах более низкого уровня иерархии*. Заголовки всех реализаций одного и того же виртуального метода должны быть идентичными (одинаковое число параметров, порядок их следования, их типы и т. д.). Статический же метод, переопределив другой статический метод, может иметь другую совокупность параметров;
- переопределять виртуальный метод в каждом из потомков нужно только по необходимости (если хочется изменить действия, описанные в наследуемом методе);
- объект, имеющий хотя бы один виртуальный метод, должен содержать конструктор.

Каждый вызов виртуального метода проходит через обращение к ТВМ, а статические методы вызываются «напрямую», вот почему вызов статического метода происходит быстрее, чем виртуального. Поэтому при выборе метода следует оценивать гибкость, которую дает виртуальный метод, и некоторое увеличение скорости подсчетов, которое дают статические методы.

О внутреннем представлении объектов

Поля объекта записываются в порядке их описаний как непрерывная последовательность переменных. Если объектный тип определяет виртуальные методы, то компилятор размещает в нем дополнительное 16-ти битовое поле, называемое полем таблицы виртуальных методов. Оно используется для запоминания адреса смещения таблицы виртуальных методов в сегменте данных. Если объект унаследовал родительские поля, то его собственные поля записываются после родительских полей. Если объектный тип наследует виртуальные методы, то он также наследует и поле таблицы виртуальных методов, благодаря чему новое 16-ти битовое поле не размещается.

Таблица виртуальных методов автоматически создается компилятором для объекта, имеющего виртуальные методы, и программа никогда не манипулирует ими непосредственно.

Заполнение полей таблицы виртуальных методов экземпляра объекта осуществляется конструктором объектного типа. Программа никаким другим способом не имеет к таблице виртуальных методов прямого доступа.

Первое слово (16 бит) таблицы виртуальных методов содержит размер экземпляра соответствующего объектного типа. Эта информация используется конструктором и деструктором для определения количества байтов, которое затем передается операторам `New` и `Dispose` при использовании расширенного их синтаксиса.

Второе слово таблицы виртуальных методов содержит отрицательный размер экземпляра соответствующего объектного типа. Эта информация используется программой контроля вызовов виртуальных методов для выявления неинициализированных экземпляров объектов в случае включения директивы {`$R+`}.

Третье и четвертое слова таблицы виртуальных методов содержит 0. Далее следует список 32-разрядных указателей методов – адресов точки входа в порядке их описания. Каждый указатель ссылается на свой виртуальный метод.

Для непосредственной работы с ТВМ используются две функции.

Стандартная функция

```
TypeOf (ИмяТипаИлиИмяЭкземпляра): Pointer
```

возвращает указатель на ТВМ для конкретного экземпляра или самого типа объекта и применяется только к объектам, имеющим ТВМ, иначе будет ошибка.

Функция `TypeOf` может быть использована для проверки фактического экземпляра по схеме

```
if TypeOf (Self) = TypeOf (ObjVAR) then ...
```

Стандартная функция `SizeOf` при применении к экземпляру типа объект, который имеет связь с таблицей ТВМ, возвращает размер, который хранится в ТВМ. Для типов объектов, имеющих ТВМ, функция `SizeOf` всегда возвращает фактический размер экземпляра, который может отличаться от заявленного в описании типа.

Задание 1. Технологией ООП написать программу для решения уравнений до третьей степени включительно $a_3 x^3 + a_2 x^2 + a_1 x + a_0 = 0$ с вещественными коэффициентами на множестве комплексных чисел, если степень уравнения и коэффициенты уравнения – последовательность некоторых случайных чисел.

Экземпляры объектов в динамической памяти

Все приведенные до сих пор экземпляры объектов были статическими, они описывались в секции **var**, им присваивались имена, и сами экземпляры объектов размещались в сегменте данных программы (глобальные переменные) или в стеке (локальные переменные).

Но экземпляры объектов могут быть размещены и в динамической памяти. Подобно любым другим динамическим структурам данных динамические экземпляры объектов объявляться как ссылки:

```
var
    ИмяССылкиНаОбъект : ^Тип_Объекта;
```

Дальнейшее обращение к экземплярам объектов, их полям и методам тоже будет обычным:

- ИмяССылкиНаОбъект – ссылка на экземпляр объекта;
- ИмяССылкиНаОбъект^ – экземпляр объекта в целом;
- ИмяССылкиНаОбъект^.ИмяПоля – поле экземпляра объекта;
- ИмяССылкиНаОбъект^.ИмяМетода – метод экземпляра объекта.

Для выделения памяти под динамические экземпляры объекта используется стандартная процедура New:

```
New(ИмяССылкиНаОбъект)
```

Процедура New, как и обычно, выделяет в динамической памяти область, достаточную для хранения экземпляра типа, определяемого указателем, и возвращает адрес этой области в указателе. Если же динамический объект содержит виртуальные методы, то он должен быть инициализирован посредством вызова конструктора до вызова всех его остальных методов:

```
ИмяССылкиНаОбъект^.ИмяКонструктора(параметры)
```

В Turbo Pascal процедура New расширена и позволяет *в одной операции выделить память под объект и вызвать конструктор*:

```
New(ИмяССылкиНаОбъект, ИмяКонструктора(параметры))
```

Сначала выполняется New для выделения памяти для объекта в Heap. Когда она закончилась без ошибок, то вызывается конструктор объектов. Компилятор определяет правильность вызова конструктора, проверяя тип указателя, передаваемого в качестве первого параметра.

Оператор New может вызываться и как функция:

```
ИмяССылкиНаОбъект:=
New(ТипОбъекта,ИмяКонструктора(Параметры))
```

Использование оператора New как функции применимо ко всем типам данных, а не только к объектам.

Когда поля объектов, в свою очередь, динамические, тогда выделение памяти в Heap для них должно происходить в конструкторе.

В конструкторе должно происходить необходимая инициализация полей данных (в том числе и вызовы конструкторов для унаследованных полей).

Освобождение динамических экземпляров объектов. Деструкторы

Для освобождения памяти от динамических объектов применяется стандартная процедура

```
Dispose (ИмяСсылкиНаОбъект)
```

Такой вызов освободит динамический экземпляр объекта в целом. При выполнении этой процедуры освобождается количество байтов, равное размеру объекта, соответствующего типу указателя.

Но когда поля данного объекта были динамическими, и под них выделялась дополнительная память при выполнении конструктора или иной процедуры инициализации, тогда их нужно освободить до уничтожения самого объекта.

Для корректного освобождения памяти из-под полиморфных объектов, для которых создавалась ТВМ, вводится специальный вид метода – *деструктор*.

Деструктор объявляется среди других методов служебным словом destructor вместо procedure. Обычно деструктору дается имя Done («Завершено»).

Исполняемый код деструктора никогда не бывает пустым, потому что компилятор по служебному слову destructor вставляет в конец тела метода операторы получения размера объекта из ТВМ.

Смысл и необходимость введения деструктора заключается в том, что его можно использовать в расширенной процедуре Dispose так же, как конструктор в New:

```
Dispose (ИмяСсылкиНаОбъект, ИмяДеструктора);
```

Действует такой вызов следующим образом: сначала вызывается деструктор и выполняются описанные в нем завершающие действия как обычного метода. Далее, если объект содержит виртуальные методы, тогда деструктор осуществляет поиск размера объекта в ТВМ и передает размер процедуре Dispose, которая освобождает правильное количество байт памяти.

Объект может иметь деструкторы даже в том случае, если все его методы статические.

Поэтому для динамических объектов всегда имеет смысл объявлять виртуальный деструкторы, даже и пустой, который нужен для нормальной работы процедуры Dispose:

```
destructor ИмяТипоОбъекта.Done; virtual;
 begin
 end;
```

Обработка ошибок при работе с динамическими объектами

При работе с динамическими объектами в программе могут возникать следующие нештатные ситуации:

- при размещении динамического экземпляра объекта в `Heap` при помощи указателя не хватает памяти для экземпляра объекта. Тогда указатель получает значение `nil`. Инициализация дальше не возможна и нужно было бы прекратить работу конструктора;
- успешно выполнилось выделение памяти для экземпляра объекта в динамической памяти, однако поля объекта – большие динамические массивы или другие динамические объекты, для выделения памяти под которые места не хватает.

Правильное размещение данных в динамической области контролируется системной стандартной функцией, имеющей заголовок

```
function HeapFunc (Size:Word) : Integer;    Far;
```

которая возвращает следующие коды завершения операций `New` и `GetMem`:

0: попытка выделить блок необходимого размера закончилась неудачно. Происходит аварийное завершение программы;

1: попытка выделить блок необходимого размера закончилась неудачно, но вместо ошибки процедуры `GetMem` и `New` возвращают указатель со значением `nil`;

2: попытка выделить блок необходимого размера закончилась удачно.

Turbo Pascal позволяет установить пользовательскую функцию обработки ошибок динамической памяти при помощи переменной `HeapError`, являющейся стандартной и не требующей описания в разделе переменных. Она содержит адрес стандартной функции обработки ошибок, которая может быть замещена на пользовательскую функцию обработки ошибок формата:

```
function HeapFunc (Size:Word) : Integer;    Far;
```

путем присвоения адреса пользовательской функции переменной `HeapError` таким образом:

```
HeapError := @HeapFunc
```

Данная возможность полезна при использовании конструкторов, если, например, при выделении памяти под динамические поля произошел сбой (не хватает памяти). Будет разумно, если в подобной ситуации конструктор отменит все предыдущие действия по выделению памяти для экземпляра объекта.

Для этой цели введена стандартная процедура `Fail` (неуспешно), которая может быть вызвана только из конструктора.

Вызов этой процедуры освобождает память, которая была выделена для экземпляра объекта еще до входа в конструктор, и возвращает в ссылке значение nil. Получение nil означает неудачное выделение памяти.

Ситуация, когда недостаточно памяти возможна и в случае статических объектов с динамическими полями. Так, при выделении памяти конструктором для динамических полей в куче может возникнуть ситуация нехватки памяти. Но

поскольку объект статический, нельзя передать значение nil в ссылку – ее просто нет. Вместо этого предлагается использовать имя конструктора как логическую функцию.

Если внутри конструктора была вызвана процедура Fail, то вернется значение true, в остальных случаях – false. Подобным способом анализа можно пользоваться и для проверки работы наследственных конструкторов.

Напомним, что если при попытке поместить динамический экземпляр объекта свободной памяти будет недостаточно, то вызов расширенной процедуры New сгенерирует код фатальной ошибки выполнения 202. А если переписать системную функцию HeapFunc таким образом:

```
function HeapFunc(Size:Word):Integer;  Far;
 begin
  HeapFunc := 1
 end;
```

чтобы она возвращала значение 1 во всех случаях, то в ссылочную переменную в случае ошибки вернется значение nil, а программа не прервется.

Далее эту ситуацию можно программно проанализировать и как-то отреагировать. Значит, если при запросе памяти для объекта процедурой

```
New(ИмяССылкиНаОбъект, ИмяКонструктара(параметры))
```

функция HeapFunc выдаст значение 1, то конструктор не будет выполняться, а в ИмяССылкиНаОбъект запишется nil.

Если же начинает выполняться тело конструктора, то для экземпляра объекта уже гарантированно выделено место.

Но сам конструктор также может выполнять действия по выделению памяти для динамических полей.

Опишем функцию Check, которую будем использовать при выделении памяти.

```
procedure Check (var P: Pointer; Size: Word);
 var
     OldHeapError: Pointer;
 begin
    OldHeapError := HeapError;
    HeapError    := Addr (HeapFunc);
    GetMem          (P, Size);
    HeapError    := OldHeapError;
 end;
```

Перед тем как использовать стандартные процедуры выделения динамической памяти GetMem или New, можно было бы проверить функцией MaxAvail, которая возвращает размер в байтах наибольшего непрерывного участка памяти в Heap области, имеется ли в Heap требуемая память.

Примеры учебных задач по ООП

Задача 1. Модуль по работе с векторами и матрицами

Технологией ООП разработать модуль по работе с векторами и матрицами, размещенными в динамической памяти, размеры которых становятся известными во время выполнения программы, для последующего использования их при решении задач линейной алгебры.

Алгоритм. При размещении вектора в динамической памяти используем тип – указатель на вектор, сделав следующие определения:

```
Item          = Real;
VectorType    = array [1..1] of Item;
VectorTypePtr = ^ VectorType;
```

и далее выделим память под вектор такого типа процедурой GetMem.

При размещении матрицы в динамической памяти используем тип – указатель на вектор указателей на строку матрицы:

```
MatrixType    = array [1..1] of VectorTypePtr;
MatrixTypePtr = ^ MatrixType;
```

В модуле unit V_OOP опишем два независимых объектных типа Vector и Matrix и соберем простейшие ресурсы по работе с вектором и матрицей, полями которых будут не только переменные типа VectorType и MatrixType, но и их текущие размеры:

- размещение вектора и матрицы в динамической памяти (GetMemory);
- освобождение вектора и матрицы из динамической памяти (FreeMemory);
- инициализация вектора и матрицы действительными случайными числами (Init_R);
- инициализация вектора и матрицы информацией из типизированных файлов (Init_File);
- распечатка вектора и матрицы (Show);
- обработка вектора и матрицы (Work).

Описание объектов по работе с массивами, размещенными в динамической памяти

```
unit V_OOP;
interface {$R-}
type
     Item          = Real;
     TFile         = Text;
     VectorType    = array[ 1..1]   of   Item;
     VectorTypePtr = ^VectorType;
     MatrixType    = array[1..1] of VectorTypePtr;
     MatrixTypePtr = ^MatrixType;
     Vector        = object
            V : VectorTypePtr;
```

```
       n : Word;
        procedure    Init_n(n_:Word);
        procedure    GetMemory;
        procedure    FreeMemory;
        procedure    Init_R;
        procedure    Init_File(name: String);
        procedure    Show;
                end;
Matrix        = object
       A : MatrixTypePtr;
       n, m : Word;
        procedure    Init(n_, m_ : Word);
        procedure    GetMemory;
        procedure    FreeMemory;
        procedure    Init_R;
        procedure    Init_File(name: String);
        procedure    Show;
                end;
implementation
 procedure  Vector.Init_n;
   begin
    n := n_;
   end;
 procedure  Vector.GetMemory;
   begin
    GetMem(V, n * SizeOf(Item));
   end;
 procedure  Vector.FreeMemory;
   begin
    FreeMem(V, n * SizeOf(Item));
   end;
  procedure   Vector.Init_R;
   var  i : Word;
    begin
      for  i := 1  to  n  do
           V^[i] := 10 * Random - 5;
    end;
 procedure   Vector.Init_File;
  var  i : Word;
       F : TFile;
   begin
       Assign(F, name);
       Reset (F);
       for  i := 1  to  n  do  Read(F,V^[i]);
       Close(F);
   end;
 procedure   Vector.Show;
```

```
 var i : Integer;
  begin
   for  i := 1  to  n  do
     Write (V^[i]:8:2);
   Writeln;
   Writeln;
  end;
procedure  Matrix.Init;
  begin
   n := n_;
   m := m_;
  end;
procedure  Matrix.GetMemory;
 var  i : Word;
  begin
    GetMem(A, n * SizeOf(Pointer));
    for  i := 1  to n do
       GetMem ( A^[i], m * SizeOf(item) );
  end;
procedure  Matrix.FreeMemory;
 var  i : Word;
  begin
      for  i := 1  to n do
        FreeMem ( A^[i], m * SizeOf(item));
      FreeMem(A, n * SizeOf(Pointer));
  end;
 procedure   Matrix.Init_R;
  var  i, j : Word;
   begin
    for  i := 1  to n do
      for  j := 1  to  m do
         A^[i]^[j] := 10 * Random - 5;
   end;
 procedure   Matrix.Init_File;
  var  i, j : Word;
       F    : TFile;
   begin
     Assign(F, name);
     Reset(F);
     for  i := 1  to n do
      for  j := 1  to  m do
         Read(F, A^[i]^[j]);
     Close(F);
   end;
 procedure   Matrix.Show;
  var
     i, j : Integer;
```

```
        begin
          Writeln;
          for  i := 1  to n do
           begin
            for  j := 1  to  m do
              Write ( A^[i]^[j] :8:2 );
            Writeln;
           end;
          Writeln;
        end;
end.
```

Для отладки методов напишем следующую главную программу.

```
Program OOP00;
 uses CRT, V_OOP;
 var
    Vec     : Vector;
    Matr    : Matrix;
 {$R+}
 begin
        ClrScr;
        Vec.Init_n(6);
        Vec.GetMemory;
        Vec.Init_R;
        Vec.Show;
        Vec.Init_File('Tv.txt');
        Vec.Show;
        Readln;
        Matr.Init(5,4);
        Matr.GetMemory;
        Matr.Init_R;
        Matr.Show;
        Matr.Init_File('TM.txt');
        Matr.Show;
        Readln;
 end.
```

Задача 2. Модуль по решению некоторых задач линейной алгебры

Используя ресурсы модуля из предыдущей задачи, технологией ООП разработать модуль по решению некоторых задач линейной алгебры.

Алгоритм. Разработаем модуль `VLin_alg`, в котором подключим предыдущий модуль `V_OOP`. Опишем объектный тип `Lin_Alg`, согласно последующему объявлению, в котором в разделе `public` объявим методы для решения классических задач линейной алгебры, назначение которых видно из их названий. В разделе `private` объявим поля объектных и простых типов, позволяющие работать с векторами и матрицами разных размерностей. Еще объявим методы по их инициализации не только случайными числами, но и

данными из соответствующих файлов. Далее объявим методы для получения произведения двух матриц, умножения матрицы на вектор, суммы двух векторов.

Описание объектов по решению задач линейной алгебры

```
unit VLin_alg;
interface
uses V_OOP;
{$R-}
type
Lin_Alg = object
          public
  procedure Show_Mult_Matr_F(nameA1, nameA2 : String;
                             nA1_, mA1_, nA2_, mA2_:Word);
  procedure Show_Summ_Vect_F(nameV1,nameV2:String;
                             nV1_:Word);
  procedure Show_Mult_Matr_R(nA1_, mA1_,
                             nA2_, mA2_:Word);
  procedure Show_Summ_Vect_R(nV1_:Word);
  procedure Show_Mult_Matr_Vect_R(nA1_, mA1_:Word);
  procedure Show_Mult_Matr_Vect_F
           (nameA1, nameV1:String; nA1_, mA1_:Word);
          private
     OV1, OV2, OV3                : Vector;
     nV1, nV2, nV3                : Word;
     OA1, OA2, OA3                : Matrix;
     nA1, mA1, nA2, mA2, nA3, mA3 : Word;
 procedure Init_Vect_R(var V : Vector; nV   : Word );
 procedure Init_Matr_R(var A:Matrix; nA, mA : Word );
 procedure Init_Vect  (var V:Vector; nV     : Word );
 procedure Init_Matr  (var A:Matrix; nA, mA : Word );
 procedure Init_Vect_F(name:String;
                       var V:Vector;nV:Word);
 procedure Init_Matr_F(name:String; var A:Matrix;
                       nA,mA:Word);
 procedure Mult_Matr;
 procedure Mult_Matr_Vect;
 procedure Summ_Vect;
 end;
implementation
 procedure Lin_Alg.Show_Mult_Matr_Vect_R;
     begin
         nA1 := nA1_;
         mA1 := mA1_;
         Init_Matr_R ( OA1, nA1, mA1 );
```

```
        Init_Vect_R ( OV1, mA1 );
        Init_Vect   ( OV2, nA1 );
        Mult_Matr_Vect;
        OV2.Show;
        OA1.FreeMemory;
        OV1.FreeMemory;
        OV2.FreeMemory;
   end;

procedure  Lin_Alg.Show_Mult_Matr_Vect_F;
   begin
        nA1 := nA1_;
        mA1 := mA1_;
        Init_Matr_F ( nameA1, OA1, nA1, mA1 );
        Init_Vect_F ( nameV1, OV1, mA1 );
        Init_Vect   ( OV2, nA1 );
        Mult_Matr_Vect;
        OV2.Show;
        OA1.FreeMemory;
        OV1.FreeMemory;
        OV2.FreeMemory;
   end;

procedure Lin_Alg.Mult_Matr_Vect;
 var i, j, k : Word;
       t       : item;
   begin
   for i := 1  to  nA1 do
     begin
      t := 0;
      for  j := 1  to mA1 do
          t := t + OA1.A^[i]^[j] * OV1.V^[j];
      OV2.V^[i] := t;
     end;

   end;

procedure Lin_Alg.Mult_Matr;
 var i, j, k : Word;
       t       : item;
   begin
   for i := 1  to nA1 do
    for  j:= 1  to mA2 do
     begin

      t := 0;
      for  k := 1  to mA1 do
         t := t + OA1.A^[i]^[k] * OA2.A^[k]^[j];
      OA3.A^[i]^[j] := t;
     end;
```

```
    end;
 procedure  Lin_Alg.Show_Mult_Matr_R;
    begin
         nA1 := nA1_;  mA1 := mA1_;
         nA2 := nA2_;  mA2 := mA2_;
         nA3 := nA1_;  mA3 := mA2_;
         Init_Matr_R ( OA1, nA1, mA1 );
         Init_Matr_R ( OA2, nA2, mA2 );
         Init_Matr   ( OA3, nA3, mA3 );
         Mult_Matr;
         OA3.Show;
         OA1.FreeMemory;
         OA2.FreeMemory;
         OA3.FreeMemory;
    end;
 procedure  Lin_Alg.Show_Mult_Matr_F;
    begin
         nA1 := nA1_;  mA1 := mA1_;
         nA2 := nA2_;  mA2 := mA2_;
         nA3 := nA1_;  mA3 := mA2_;
         Init_Matr_F ( nameA1, OA1, nA1, mA1 );
         Init_Matr_F ( nameA2, OA2, nA2, mA2 );
         Init_Matr   ( OA3, nA3, mA3 );
         Mult_Matr;
         OA3.Show;
         OA1.FreeMemory;
         OA2.FreeMemory;
         OA3.FreeMemory;
    end;
 procedure  Lin_Alg.Summ_Vect;
  var i : Word;
    begin
      for i := 1  to  nV1  do
        OV3.V^[i] := OV1.V^[i] + OV2.V^[i];
    end;


procedure  Lin_Alg.Show_Summ_Vect_R;
    begin
      nV1 := nV1_;
      nV2 := nV1_;
      nV3 := nV1_;
      Init_Vect_R (OV1, nV1);
      Init_Vect_R (OV2, nV2);
      Init_Vect   (OV3, nV3);
      Summ_Vect;
      OV3.Show;
```

```
      OV1.FreeMemory;
      OV2.FreeMemory;
      OV3.FreeMemory;
    end;
procedure  Lin_Alg.Show_Summ_Vect_F;
    begin
      nV1 := nV1_;
      nV2 := nV1_;
      nV3 := nV1_;
      Init_Vect_F (nameV1, OV1, nV1);
      Init_Vect_F (nameV2, OV2, nV2);
      Init_Vect   (OV3, nV3);
      Summ_Vect;
      OV3.Show;
      OV1.FreeMemory;
      OV2.FreeMemory;
      OV3.FreeMemory;
    end;
procedure Lin_Alg.Init_Vect_R;
  begin
    V.Init_n(nV);
    V.Getmemory;
    V.Init_R;
    V.Show;
  end;
procedure Lin_Alg.Init_Matr_R;
  begin
    A.Init(nA, mA);
    A.Getmemory;
    A.Init_R;
    A.Show;
  end;
procedure Lin_Alg.Init_Vect;
  begin
    V.Init_n(nV);
    V.Getmemory;
  end;
procedure Lin_Alg.Init_Matr;
  begin
    A.Init(nA, mA);
    A.Getmemory;
  end;
procedure Lin_Alg.Init_Vect_F;
  begin
    V.Init_n(nV);
```

```
    V.Getmemory;
    V.Init_File(name);
    V.Show;
   end;
 procedure Lin_Alg.Init_Matr_F;
   begin
    A.Init(nA, mA);
    A.Getmemory;
    A.Init_File(name);
    A.Show;
   end;
 end.
```

В главной программе подключим предыдущий модуль и выполним модельные расчеты согласно следующей программе.

Проверка объектов по решению задач линейной алгебры

```
Program OOP11;
  uses CRT, VLin_Alg; {$R-}
  var
        D    : Lin_Alg;
  begin
      ClrScr;
      Writeln('random-chisla');
      Writeln('MULT Matrises');
      D.Show_Mult_Matr_R (5, 4, 4, 5);
      Readln;
      Writeln('ADD Vectors');
      D.Show_Summ_Vect_R (5);
      Readln;
      Writeln('Iz file''s');
      Writeln('MULT Matrises');
      D.Show_Mult_Matr_F('TM2.txt','TM1.txt',5,4,4,5);
      Readln;
      Writeln('ADD Vectors');
      D.Show_Summ_Vect_F ( 'TV1.txt', 'TV2.txt',5 );
      Readln;
      Writeln( 'Iz file''s' );
      Writeln( 'MULT Matrisa_Vector' );
      D.Show_Mult_Matr_Vect_F('TM1.txt',
                              'TV1.txt',5,4);
      Readln;
      Writeln( 'random-chisla');
      Writeln( 'MULT Matrisa_Vector');
      D.Show_Mult_Matr_Vect_R( 5, 4 );
      Readln;
  end.
```

1.9. Введение. Основы работы в MathCad

Использование систем компьютерной математики в процессе информатизации образования

В настоящее время в научных исследованиях, инженерных разработках и экономико-математических расчётах самое широкое применение находят системы компьютерной математики (СКМ) — системы для численных вычислений, которые становятся также одним из обязательных компонентов компьютерных технологий, используемых в образовании. Изучение СКМ начинать лучше со школы. Во-первых, школьника проще заинтересовать решением и проверкой трудных для него задач, а во-вторых, в будущем останутся навыки ориентирования в сложном мире математики.

Использование СКМ в учреждениях образования позволяет эффективно усваивать и закреплять знания, получаемые учащимися при изучении общих и специальных математических дисциплин, а также использовать возможности компьютерной математики для выполнения самостоятельных научно-исследовательских работ, подготовке курсовых и дипломных проектов по различным дисциплинам.

Когда-то системы символьной математики были ориентированы исключительно на узкий круг профессионалов и работали на больших компьютерах (мэйнфреймах). С появлением ПК эти системы были переработаны и доведены до уровня массовых серийных программных систем разной ориентации – от рассчитанной на широкий круг потребителей системы Mathcad до компьютерных монстров Mathematica, MatLab, Maple, Statictika и др. Эти программы обладают средствами выполнения различных численных и аналитических математических расчетов.

Спектр задач, решаемых системами компьютерной математики (СКМ), очень широк:

- проведение математических исследований, требующих вычислений и аналитических выкладок;
- разработка и анализ алгоритмов;
- математическое моделирование и компьютерный эксперимент;
- анализ и обработка данных;
- визуализация, научная и инженерная графика и т.д.

К СКМ обычно относят:

- табличные процессоры, например, Microsoft Excel, IBM Lotus Symphony Spreadsheets, Gnumeric, Apache OpenOffice.org Calc;

- системы для статистических расчётов, например, STATISTICA, PASW Statistics;

- системы для моделирования, анализа и принятия решений, например, GPSS, AnyLogic, DSS;

- универсальные математические системы, куда включаются системы компьютерной алгебры.

Программные средства, ориентированные на решение математических задач, условно дифференцированы на: встроенные средства различной степени развития той либо иной системы программирования (например, Basic, Pascal, C и др.); специальные языки программирования (например, Fortran, ISETL — интерактивный язык множеств, Prolog — язык и система логического программирования и др.); узко-специальные, специальные и общие пакеты прикладных программ (Reduce, Maple, Matlab, Mathematica, MathCAD, и др.).

Каждый из прикладных математических пакетов имеет свою область применения и работает под управлением конкретных операционных систем. Они, как правило, содержат библиотеки и пакеты дополнений, расширяющие базовые возможности пакета, и поэтому их в настоящее время называют системами.

Основное назначение систем компьютерной алгебры (СКА) – работа с математическими выражениями в символьной форме.

Символьный процессор системы выполняет требуемые или неявные преобразования или вычисления и выдаёт ответ в математической нотации. Алгоритмы внутренних преобразований имеют алгебраическую природу, что и отражено в названии систем — системы компьютерной алгебры. Справочная система всех СКА содержит и обеспечивает пользователей описаниями функциональных возможностей и демонстрационными примерами работы, информационными сообщениями о текущем состоянии системы, а также сведениями о математических основах алгоритмов. Справедливо утверждение, что многие СКА, по сути, являются не только инструментами для получения и анализа решений, но и математическими энциклопедиями.

Приведём основные функциональные возможности СКА.

Используя СКА и компьютер, можно выполнять в аналитической форме:

- упрощение выражений или приведение к стандартному виду,
- подстановки символьных и численных значений в выражения,
- выделение общих множителей и делителей,
- раскрытие произведений и степеней, факторизацию,
- разложение на простые дроби,
- нахождение пределов функций и последовательностей, операции с рядами,
- дифференцирование в полных и частных производных,
- нахождение неопределённых и определённых интегралов,
- анализ функций на непрерывность,
- поиск экстремумов функций и их асимптот,
- операции с векторами,
- матричные операции,
- нахождение решений линейных и нелинейных уравнений,
- символьное решение задач оптимизации,
- алгебраическое решение дифференциальных уравнений,
- интегральные преобразования,
- прямое и обратное быстрое преобразование Фурье,
- интерполяция, экстраполяция и аппроксимация,

- статистические вычисления,
- машинное доказательство теорем.

Если задача имеет точное аналитическое решение, пользователь СКА может получить это решение в явном виде (разумеется, речь идёт о задачах, для которых известен алгоритм построения решения). Также большинство СКА обеспечивают:

- числовые операции произвольной точности,
- целочисленную арифметику для больших чисел,
- вычисление фундаментальных констант с произвольной точностью,
- поддержку функций теории чисел,
- редактирование математических выражений в двумерной форме,
- построение графиков аналитически заданных функций,
- построение графиков функций по табличным значениям,
- построение графиков функций в двух или трёх измерениях,
- анимацию формируемых графиков разных типов,
- использование пакетов расширения специального назначения,
- программирование на встроенном языке,
- автоматическую формальную верификацию,
- синтез программ.

В СКА можно производить вычисления в арифметике с плавающей точкой и указывать точность, реализована точная рациональная арифметика, т.е. можно производить численные расчёты без потери точности.

Широкое распространение в настоящее время имеют СКА MatLab, Mathematica, Maple, Reduce, Derive, Maxima, MathCAD.

При использовании систем компьютерной алгебры в обучении можно выделить два направления, а именно: первое — использование СКА при подготовке студентов, специализирующихся по информатике, и, второе — применение СКА при изучении общих математических курсов. Наиболее привлекательным для начинающих является СКМ MathCAD, основанный на визуально-ориентированном языке программирования.

Пакет MathCAD ориентирован в первую очередь на проведение численных расчетов, но имеет встроенный символический процессор Maple, что позволяет выполнять аналитические преобразования.

Проводя вычисления с помощью MathCad кажется, что вы просто работаете с бумагой. MathCad позволяет представлять результаты расчетов таким образом, что их понимают все пользователи. Все вычисления здесь осуществляются на уровне визуальной записи выражений в общеупотребительной математической форме. MathCad имеет хорошие подсказки, обладает широкими возможностями символьных вычислений.

Maple, MATLAB и Mathematica – это языки программирования, гибкие и мощные, но трудные в использовании и требующие длительного времени на изучение. Поэтому, в отличие от MathCad, пользовательский интерфейс их сложен, в нем легко допускать ошибки, которые вынуждают проверять и отлаживать весь код. Программирование не визуально и не интерактивно. Невозможно поменять несколько строк в программе и автоматически увидеть

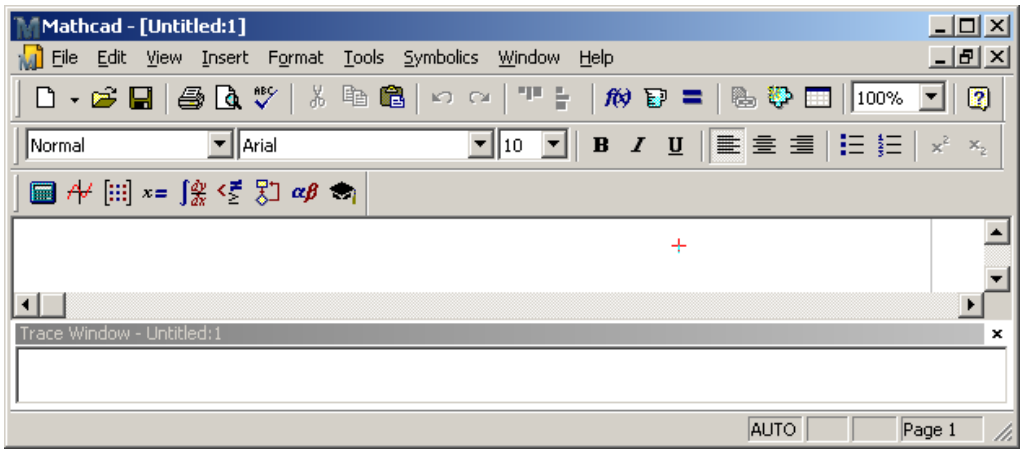
результаты. Для этого пользователю потребуется перекомпилировать и перезапустить программу. Не программисты заниматься этим не смогут. Рабочие же листы MathCad можно легко перепроверить. Вычислительные процедуры и важные для расчетов параметры выносятся так, что их можно легко менять и тут же следить за результатами.

В состав MathCAD входят несколько интегрированных между собой компонентов: мощный текстовый редактор, позволяющий вводить, редактировать и форматировать как текст, так и математические выражения; вычислительный процессор, умеющий проводить расчёты по введённым формулам, используя встроенные численные методы; символьный процессор, позволяющий проводить аналитические вычисления и являющийся, фактически, системой искусственного интеллекта; огромное хранилище справочной информации, как математической, так и инженерной, оформленной в качестве интерактивной электронной книги.

Основы работы в MathCad

Базовые возможности MathCad

После запуска пакета MathCad из Windows через некоторое время на экране возникает рабочее окно MathCad.



Данное окно Windows-приложения включает в себя следующие части.

1. *Заголовок окна MathCad*. Содержит имя текущего рабочего документа.
2. *Меню*. Предназначено для выбора необходимых действий.

Меню состоит из следующих пунктов:

- File – работа с файлами, сетью интернет и электронной почтой;
- Edit – редактирование документов;
- View – изменение способов представления документа и скрытие / отображение элементов интерфейса;
- Insert – вставка объектов и их шаблонов;
- Format – изменение форматов объектов;
- Tools – управление процессом вычислений;

• Symbolic – выбор операций символьного процессора;
• Window – управление окнами системы;
• Help – работа со справочной базой, центром ресурсов и электронными книгами.

Меню MathCad контекстные: число позиций в них и их назначение зависят от состояния системы. В раскрытом меню представлен список доступных и недоступных в данный момент команд.

3. *Рабочая область MathCad* – наибольшая по размерам часть. В ней осуществляется отображение и работа с окнами рабочих документов пакета MathCad.

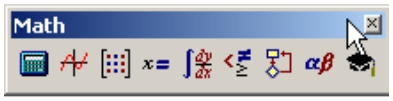
4. *Панель инструментов*. Состоит из кнопок, предназначенных для быстрого вызова наиболее важных пунктов меню.

5. *Панель форматирования*. Состоит из кнопок, предназначенных для быстрой работы с различными шрифтами и для форматирования частей документа. Она также имеет аналоги в меню.

6. *Строка состояния*. В ней отображается различная информация о работе системы, а также сообщения для пользователя MathCad об ошибках со стороны пользователя.

7. *Любые другие панели*, которые подключаются через View ► ToolBars. Например, панель «Палитры математических знаков».

8. *Палитры математических знаков*. Панель расположена ниже полосы инструментов, но можно выделить ее в отдельное окно (разной конфигурации).



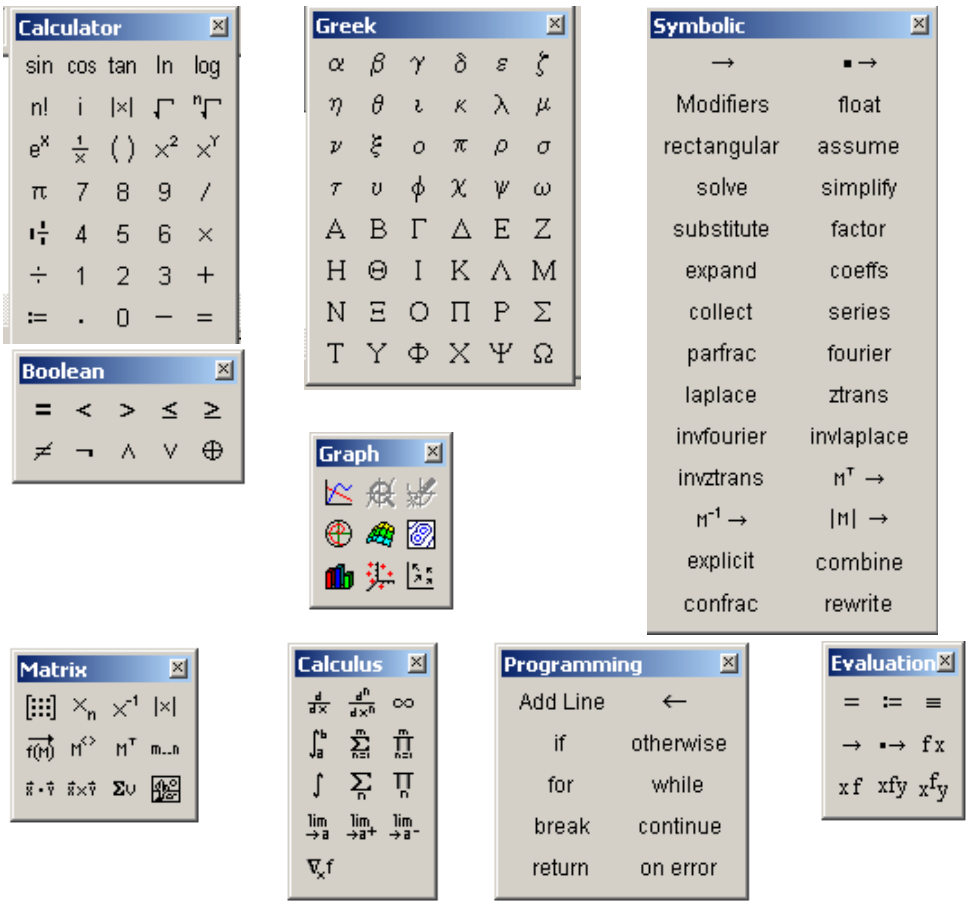
Палитры предназначены для ввода шаблонов операторов, различных специальных символов, графиков и др. Их назначение отображено рисунками на кнопках (таблица 21):

Таблица 21 – Палитры для ввода шаблонов операторов.

Кнопка	Назначение палитры	Название палитры
	Средства для вычислений	Calculator
	Векторные и матричные операторы	Matrix
	Операторы суммирования, интегрирования и дифференцирования	Calculus

	Конструкции программирования	Programming
	Операторы символьной математики	Symbolic
	Построение различных графиков	Graph
	Соотношения	Evaluation
	Знаки отношений	Boolean
	Буквы греческого алфавита	Greek

Открывая соответствующую палитру, можно получить доступ к ее средствам.



Окна документов располагаются в рабочей области окна системы MathCad и носят название соответствующего рабочего документа или *Untitled:N*, где *N* – номер этого пустого окна. Новое рабочее окно открывается посредством щелчка на соответствующей кнопке панели инструментов или пункта New из меню File.

Вычисление значений числовых выражений

В MathCad можно вычислять значения выражений − формул с любыми арифметическими операциями, причем выражение отображается в обычном виде и вычисляется после ввода, если принудительно не отключена опция автоматического пересчета формул в меню
Tools ► Calculate ► Automatic Calculation.

Формула – это математическое выражение, состоящее из операндов, соединенных знаками математических операций.

MathCad выполняет действия как над вещественными, так и над комплексными данными. Отметим, что в числе целая часть отделяется от дробной части десятичной точкой. Комплексные числа записываются в обычной форме, как $a+bi$, и могут употребляться в большинстве случаев так же, как и вещественные числа. При вводе комплексного числа мнимую единицу i необходимо представлять, как $1i$, иначе MathCad расценит символ i как переменную.

Для *ввода выражения* необходимо щелчком мыши в нужном месте документа выбрать начало ввода. На этом месте появится красный крестик-визир. Затем начинается ввод самого выражения, и крестик-визир превращается в *курсор ввода*. Это более протяженный *уголок* или, другими словами, *выделяющая рамка*. С выделяющей рамкой связано следующее *основное правило*: все выражение, заключенное в выделяющую рамку, используется как операнд для следующей вводимой операции. Если выделяющая рамка левосторонняя, то это левый операнд, а если правосторонняя – правый.

Пример. Вычислим значение выражения 25−76+3.

Вариант 1. Введем числитель, затем нажмем на клавишу «ПРОБЕЛ», чтобы курсор ввода охватил числитель, и на знак деления. Наберем число 6 и нажмем на клавишу «ПРОБЕЛ», чтобы курсор ввода охватил всю дробь, и далее наберем число 3:

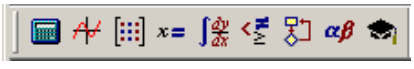
Набираем	Получаем
25 − 7 [ПРОБЕЛ] / 6 [ПРОБЕЛ] + 3	$\frac{25-7}{6}+3$

Вариант 2. Нажмем на клавишу «/» и получим шаблон для дроби. Заполним числитель выражением 25−7, а знаменатель числом 6 и нажмем на клавишу «ПРОБЕЛ», чтобы курсор ввода охватил всю дробь, а далее наберем число 3:

Набираем	Получаем
/ 25 − 7 [Клавиша Tab] 6 [ПРОБЕЛ] + 3	$\frac{25-7}{6}+3$

В меню Help ► Tutorials ► Getting Started Primers на английском языке можно найти примеры как построения выражений, так и других возможностей MathCad.

С клавиатуры напрямую можно ввести не все математические операции. Любую математическую операцию, известную MathCad, можно ввести с помощью одной из палитр панели «Палитры математических знаков».



Наиболее часто используемые операции находятся на палитре Calculator с изображением калькулятора. Вводятся они щелчком мыши по значку нужной

операции. Но можно использовать комбинации клавиш – так называемые *горячие клавиши*, за которыми закреплены определенные операции. При выборе на палитре соответствующей операции появляются сочетания закрепленных за ней горячих клавиш.

П р и м е р. Вычислить значение выражения $\frac{12\cdot 1{,}5-8^3}{1+\sqrt{35-5^2}}$

Вариант 1. Для операций возведения в степень и вычисления корня соответственно воспользуемся горячими клавишами «^» и «\»:

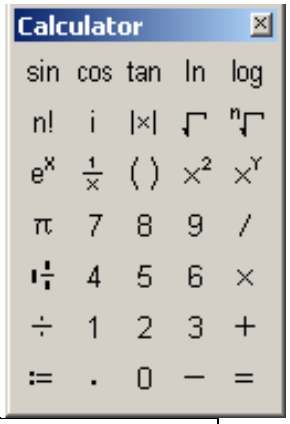
Набираем	Получаем
12*1.5−8^3 [ПРОБЕЛ] / 1+\35 – 5^2	$\frac{12\cdot 1.5-8^3}{1+\sqrt{35-5^2}}$

Нажмем клавишу «равно» (=) и получим ответ.

Вариант 2. Можно использовать и другую последовательность действий:

Набираем	Получаем
/12*1.5−8^3 [Клавиша Tab] 1+\35 – 5^2	$\frac{12\cdot 1.5-8^3}{1+\sqrt{35-5^2}}$

Вариант 3. Откроем палитру Calculator. Используя соответствующие кнопки на этой палитре, получим следующий порядок операций:



Набираем	Получаем
/ 1 2 × 1 . 5 − 8 x^y 3 [Клавиша Tab] 1 + $\sqrt{}$ 3 5 − 5 x^2	$\frac{12\cdot 1.5-8^3}{1+\sqrt{35-5^2}}$

Перечислим основные операции, которые можно вводить с клавиатуры или через палитру операторов (таблица 22):

Таблица 22 – Основные операции.

Символ операции	Операция	Пояснения
+	Сложение	
–	Вычитание	
*	Умножение	В итоговой формуле заменяется символом «·»
/	Деление	Вводится клавишей [/] как косая черта, но заменяется горизонтальной чертой
^	Возведение в степень	
$\sqrt{}$	Квадратный корень	Клавиша [\]
$\sqrt[n]{}$	Корень n-й степени	Клавиши ctrl + [\]
!	Факториал	
$\frac{d}{dx} f(x)$	Производная	Клавиша [?]
$\frac{d^n}{dx^n} f(x)$	Производная n-го порядка	Клавиши [ctrl] + [?]
$\int_a^b f(x)\,dx$	Определенный интеграл	Клавиша [&]

При редактировании неверно набранных выражений нужно следить за положением рамки ввода. Удалять символы можно и клавишей [Backspace] (предыдущего) и клавишей [Del] (последующего).

Сочетание клавиш [Shift] [←] или [Shift] [→] помогут выделить другим цветом все, что охвачено рамкой ввода. Затем это можно удалить или взять в качестве операнда для следующей операции. Если требуется удалить (или перетащить в другое место) несколько набранных фрагментов, то мышкой охватываем их и далее клавишей [Del] удаляем (или перетаскиваем) все выделенное.

При редактировании выражений можно оперативно пользоваться и мышкой, и меню Edit. Навыки достигаются путем экспериментов и тренировок.

Основные правила редактирования выражений

- Перемещение курсора ввода достигается клавишами [→] и [←].
- Расширение курсора ввода достигается клавишей [ПРОБЕЛ].
- Изменение направления ввода на противоположное направление достигается клавишей [Ins].
- Вставка оператора достигается установкой курсора между операндами и вводом символа оператора.

• Вставка оператора после части выражения достигается выделением части выражения и затем ввода оператора.

• Вставка квадратного корня достигается выделением левосторонним (вид курсора «⌊») курсором ввода подкоренного выражения и затем ввода знака «\».

• Вставка унарного минуса достигается выделением фрагмента выражения, перед которым вставляется минус, левосторонним курсором ввода и нажатием клавиши минус [–].

• Удаление оператора достигается установкой курсора ввода после оператора (вид курсора «⌋») и нажатием клавиши [Backspace] или установкой курсора ввода перед оператором и двойным нажатием клавиши [Del].

• Освобождение выражения от скобок достигается установкой курсора ввода справа от открывающей скобки и нажатием клавиши [Backspace] (или слева от закрывающей скобки и нажатием клавиши [Del]).

• Вставка фрагмента выражения в скобки достигается охватом выражения курсором ввода и набором символа одиночной кавычки «‘».

• Задание выражения в качестве аргумента функции достигается так: охватываем выражение курсором ввода, вводим скобки, расширяем курсор ввода для охвата всего выражения со скобками и вводим имя функции (например, с применением кнопки f(x) на панели инструментов).

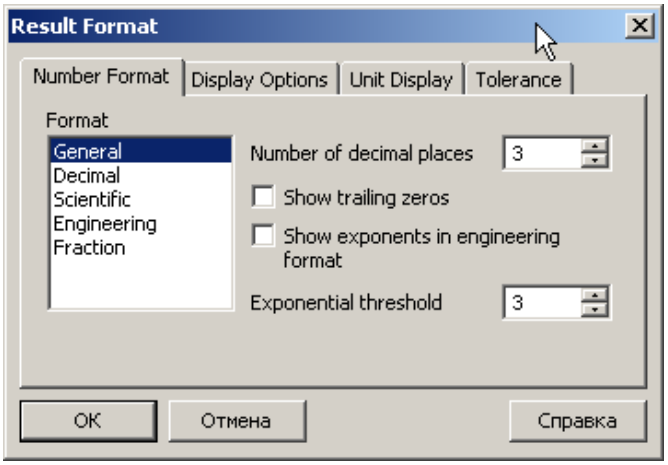
• Отказ от неверно проделанной операции достигается выбором команды Undo в меню Edit или щелчком кнопки Undo на панели инструментов.

О представлении результатов вычислений

MathCad проводит вычисления с высокой точностью и выводит результаты на экран в соответствии с заданными соглашениями о их представлении:

Набираем	Получаем
5–0.000001=	5
5+0.000001=	5

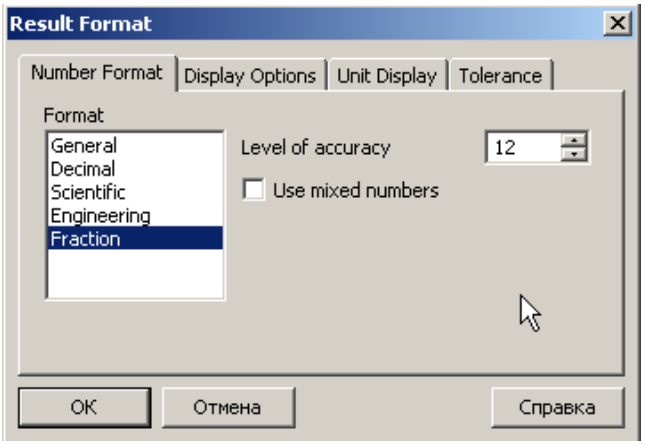
По умолчанию это три знака, но эту величину можно установить по своему усмотрению. Для этого необходимо в меню Format выбрать пункт Result. На экране появится окно. На закладке Number Format в поле Format выбирается опция General.



В поле Number of decimal places укажем число 6. В поле Exponential threshold установлена точность «3 знака», которую можно в дальнейшем увеличить. Получим следующее:

Набираем	Получаем
5–0.000001=	4.999999
0.000001+0.000002=	3×10^{-6}

Чтобы представить результат в виде рационального числа, выбирается опция Fraction.



По умолчанию задается величина точности в 12 знаков. Ее можно изменить.

Получим следующее:

Набираем	Получаем
5–0.000001=	$\frac{4999994}{999999}$
0.000001+0.000002=	$\frac{3}{1000000}$

На других закладках (Display Options, Unit Display, Tolerance) можно изменять значения по умолчанию некоторых других параметров.

Переменные в выражениях

При помощи *переменных* обозначаются скалярные величины, векторы и матрицы. Имена переменных могут быть любой длины и состоять из латинских, греческих букв, цифр от 0 до 9, символа подчеркивания, %, ∞. Переменная может быть набрана любым шрифтом, однако MathCad считает *разными имена*, набранные в разных регистрах и разными шрифтами: например, F, f, f – это разные переменные.

Некоторые переменные в MathCad имеют предопределенные значения: π, e, ∞, %, TOL, outn, inn, ORIGIN, PRNCOLWIDTH, PRNPRECISION, FRAME.

Переменные бывают локальными и глобальными.

Присвоение значения локальной переменной записывается так:

Набираем	Видим
Имя_переменной : выражение	Имя_переменной := выражение

Примеры локальных переменных:

$$x := 2.7 \qquad A := \begin{pmatrix} -2.13 & 4.5 \\ 0 & -1.35 \end{pmatrix} \qquad S := \sum_{n=1}^{10} \frac{1}{2 \cdot n + 1} \qquad I := \int_0^{\pi} \sin(x)\, dx$$

Присвоение значения *глобальной переменной* записывается следующим образом:

Набираем	Видим
Имя_переменной ~ выражение	Имя_переменной ≡ выражение

Знак «≡» можно также набрать из палитры Evaluation.

Примеры глобальных переменных:

$$D \equiv -12.7 \qquad A \equiv \begin{pmatrix} -4.73 & 8.5 \\ 0.9 & -4.35 \end{pmatrix} \qquad C \equiv \sum_{n=1}^{10} \frac{1}{2 \cdot n^3 + 1} \qquad J \equiv \int_0^{\pi} \cos(x)\, dx$$

MathCad читает рабочий документ *сверху вниз* и *слева направо*. При первом просмотре начальные значения присваиваются глобальным переменным, при втором просмотре – локальным. Если переменной не присвоено значение, тогда в выражении она будет отражена другим цветом. Значения переменных можно далее переопределять.

Для создания комментариев открывается *текстовая область*: меню Insert ► Text Region (вставка текста). В появившемся прямоугольнике вводится текст или вставляется формула (вставка математической области). В текстовой области курсор ввода имеет вид красной вертикальной черты.

Комплексные данные

Рассмотрим примеры некоторых несложных операций с комплексными числами и переменными.

Пусть $r = 2,\ Q = \frac{3}{4} \cdot \pi$. Зададим комплексные переменные z1 и z2: $z1 = \sqrt{-1}$, $z2 = r \cdot e^{i \cdot Q}$. Выведем их значения и получим:

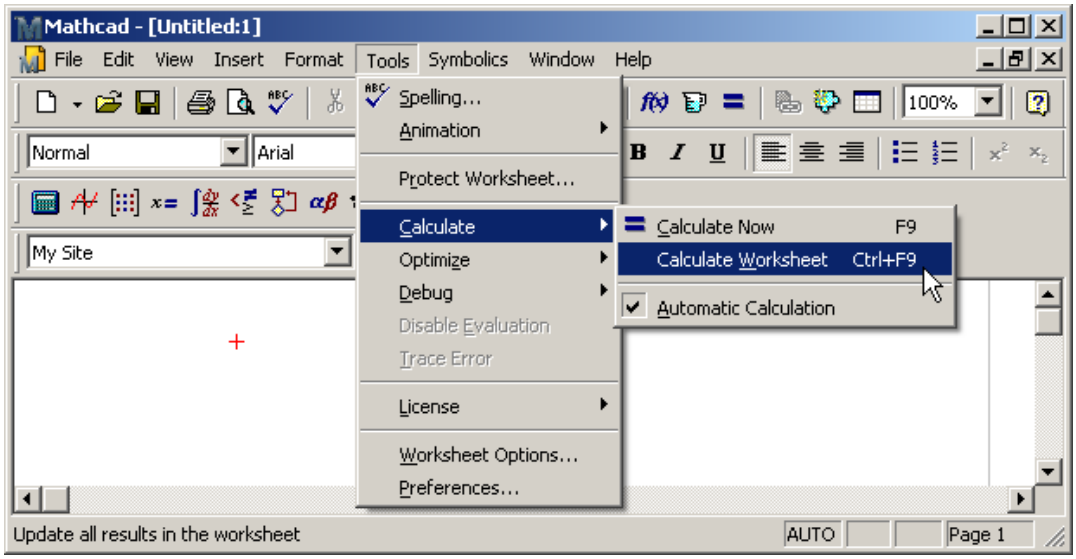
r := 2	$Q := \frac{3}{4}\pi$	$z1 := \sqrt{-1}$	$z2 := r \cdot e^{i \cdot Q}$
z1 = i	z2 = –1.414 + 1.414i		

Выполним с $z1$ и $z2$ следующие операции:

z1 + z2 = –1.414 + 2.414i z1 – z2 = 1.414 – 0.414i z1·z2 = –1.414 – 1.414i $\frac{z2}{z1}$ = 1.414 + 1.414i	Арифметические операции сложения, вычитания, умножения, деления
Re(z2) = –1.414 Im(z2) = 1.414	Выделение реальной и мнимой части числа
$\lvert z2 \rvert = 2$	Модуль числа
arg(z2) = 2.356	Функция возвращает аргумент числа
sin(z2) = –2.152 + 0.302i cos(z2) = 0.34 + 1.911i ln(z2) = 0.693 + 2.356i log(z2) = 0.301 + 1.023i tan(z2) = –0.041 + 1.118i	Функции, доступные через палитру Calculator

Управление вычислениями

В меню Tools объединены команды управления вычислительным процессом, позволяющие менять режимы перерасчета документа.



Прерывание вычислений

Для прерывания вычислений следует нажать клавишу «Esc». Появляется окно, посредством которого нужно подтвердить прерывание процесса. Возобновить работу можно, нажав клавишу «F9» или кнопку с изображением знака «=».

Дискретные переменные

Дискретные переменные принимают не одно, а несколько значений, но это не массивы. Дискретная переменная аккумулирует связанную с переменной совокупность значений, и если используется эта переменная, то сразу используется вся совокупность ее значений и невозможно использовать какое-то одно из них.

В математике часто встречаются задачи, в которых какое-нибудь выражение зависит от переменной, а переменная принимает не одно, а несколько значений, и для каждого из них нужно вычислить значение выражения. На этот случай в MathCad существуют так называемые *дискретные переменные* (их еще называют *ранжированными*). Сразу подчеркнем, что речь не идет о массивах.

Дискретные переменные можно определять непосредственно или используя палитру Matrix. Рассмотрим оба случая.

Первый способ задания дискретной переменной:

Набираем	Видим
Имя_переменной:нач_знач,след_знач; посл_знач	Имя_переменной:=нач_знач, след_знач..посл_знач

Например, t:=10,11..17 означает, что переменная t принимает значения от 10 до 17 с шагом 1, т. е. переменная t последовательно проходит все следующие значения: 10, 11, 12, 13, 14, 15, 16, 17.

Набирая t=, на экран выведется столбик значений этой переменной. Если количество значений таково, что все они сразу не отразились (по умолчанию больше за 15), то надо щелкнуть мышкой на таблице и появится полоса прокрутки, позволяющая просмотреть все значения.

В случае, когда шаг изменения дискретной переменной равен единице, шаблон задания переменной можно взять в сокращенном варианте:

Набираем	Видим
Имя_переменной:нач_знач; посл_знач	Имя_переменной:=нач_знач..посл_знач

На первый взгляд дискретная переменная напоминает вектор – одномерный массив индексированных переменных. Но мы уже отмечали, что дело обстоит не так. Дискретная переменная аккумулирует связанную с переменной совокупность значений, и если мы используем эту переменную, то сразу используем всю совокупность ее значений и невозможно (как при использовании массивов) использовать какое-то одно из них.

Второй способ задания дискретной переменной связан с использованием шаблона  из палитры Matrix, в котором нужно заполнить соответствуюшие поля:

Набираем	Видим
Имя_переменной:  нач_знач[Tab] посл_знач	Имя_переменной**:=**нач_знач**..**посл_знач

Отсюда видно, что шаг изменения дискретной переменной при таком способе задания равен единице.

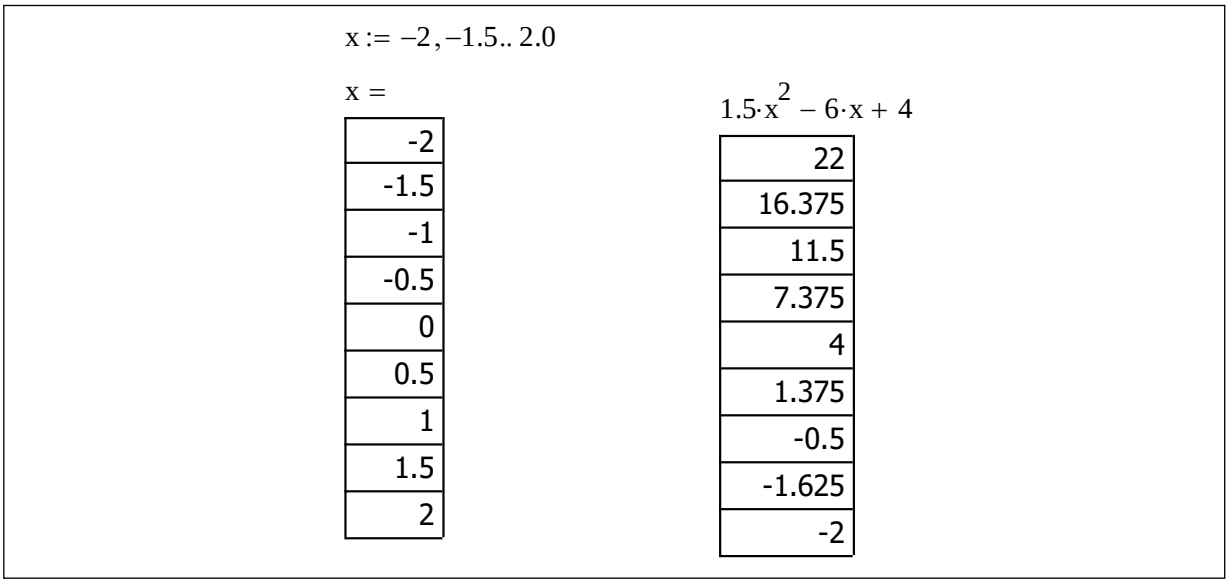
Примеры описания дискретных переменных:

Набираем	Видим
b:  1[Tab]10	b:=1..10
k:5.5;9	k:=5.5..9
a: –3.5, –3;2.5	a:=–3.5, –3..2.5
times:1s,2.5s;7s	times:=1s,2.5s..7s
q:1/4[ПРОБЕЛ],1/2[ПРОБЕЛ];7/4	$q := \frac{1}{4}, \frac{1}{2} .. \frac{7}{4}$

Рассмотрим задачу, использующую дискретные переменные.

Построить таблицу значений функции $1{,}5x^2 - 6x + 4$ при $x = -2, -1.5, \ldots, 2.5$.

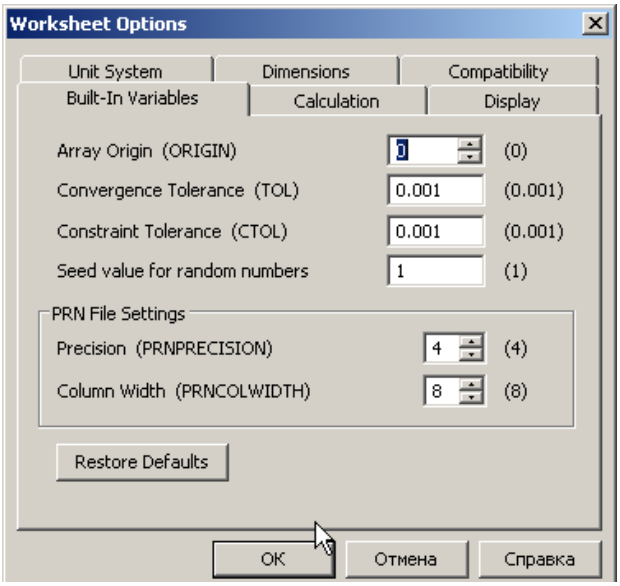
Документ MathCad будет иметь следующий вид:



1.10. Векторы и матрицы

Для задания векторов и матриц можно использовать дискретные переменные, которые принимают все значения изменения индекса. Вводить индексы можно через палитру Matrix или клавишей «[». Пример получения значений вектора $x_0, x_1, \ldots, x_{10}$:

Таким способом можно получать одномерные, двухмерные, трехмерные и так далее массивы. Заполняются они поэлементно, как правило, по некоторым формулам. По умолчанию индексы начинаются с нуля. Переустановить значение индекса можно один раз во всем документе, например, присвоив переменной `ORIGIN` значение 1.



При выводе на экран значений одномерных массивов получим столбец значений, а если переменная зависит от двух индексов − прямоугольную таблицу, например:

$$i := 0..2 \qquad j := 0..2 \qquad A_{i,j} := \frac{i\cdot\left(j^2+1\right)}{i^2+j^2+1} \qquad A = \begin{pmatrix} 0 & 0 & 0 \\ 0.5 & 0.667 & 0.833 \\ 0.4 & 0.667 & 1.111 \end{pmatrix}$$

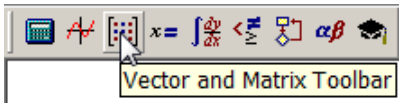
$$Z_i := \sin\left(\frac{\pi}{i+1}\right) \qquad Z = \begin{pmatrix} 0 \\ 1 \\ 0.866 \end{pmatrix}$$

Матрицы и операции над ними

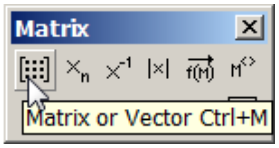
Если элементами матрицы являются константы и размеры матрицы невелики, то ее целесообразно создавать и заполнять через шаблон.

Для создания числовой матрицы нужно выполнить определенную последовательность действий:

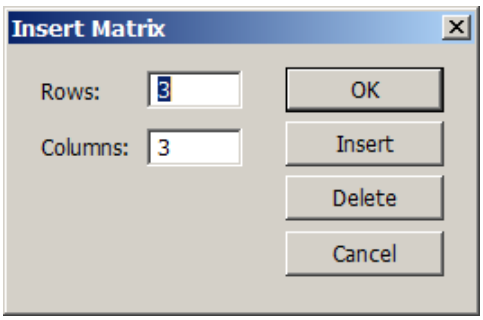
1. подвести указатель мыши к нужному место рабочего документа и щелкнуть левой кнопкой мыши;
2. щелкнуть на палитре математических знаков на кнопке «Vector and Matrix Toolbar»;

1. 

3. выбрать кнопку «Matrix or Vector»;

2. 

4. в диалоговом окне задать количество строк и столбцо; щелнуть «OK».

3. 

Получим следующий шаблон (для матрицы размерностью 3×3): $\begin{pmatrix} \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare \end{pmatrix}$.

Передвигая курсор по матрице, заполним соответствующие поля. В итоге получим требуемую матрицу.

Если задать количество столбцов, равное единице, то получим вектор-столбец.

Для изменения размеров матрицы нужно проделать следующие действия:

4. Щелкнуть на элементе матрицы, от позиции которого будут осуществляться изменения.

5. Щелкнуть на палитре математических знаков на кнопке Vector and MatrixToolbar, выбрать кнопку Vector or Matrix, задать нужное число столбцов и строк и выбирть кнопку Insert (вставить) либо Delete (удалить).

П р и м е р. Дана матрица: $A := \begin{pmatrix} 1 & -3 & -2 \\ 5 & 3 & 6 \\ -1 & 2 & 8 \\ 7 & 9 & 15 \end{pmatrix}$.

Выделим элемент данной матрицы, равный 3, и удалим строку и столбец, в которых этот элемент расположен.

Получим матрицу: $A := \begin{pmatrix} 1 & -2 \\ -1 & 8 \\ 7 & 15 \end{pmatrix}$.

Аналогично происходит и добавление строк и/или столбцов.

Основные операции с матрицами

Пусть даны матрицы:

$$A := \begin{pmatrix} -5 & 2 \\ 6 & 0 \end{pmatrix} \qquad B := \begin{pmatrix} 4 & 1 \\ 2 & -2 \end{pmatrix} \qquad C := \begin{pmatrix} 3 & -2 \\ 1 & 0 \\ 6 & 12 \end{pmatrix}$$

Простейшие операции с данными матрицами, которые можно задавать непосредственно или через палитру Matrix:

Операция	Название	Результат
+	Сумма матриц	$A + B = \begin{pmatrix} -1 & 3 \\ 8 & -2 \end{pmatrix}$
*	Произведение матриц	$A \cdot B = \begin{pmatrix} -16 & -9 \\ 24 & 6 \end{pmatrix}$
| |	Определитель матрицы	$\lvert A \rvert = -12$
M^T	Транспонирование матрицы	$C^T = \begin{pmatrix} 3 & 1 & 6 \\ -2 & 0 & 12 \end{pmatrix}$
M^{-1}	Обратная матрица	$B^{-1} = \begin{pmatrix} 0.2 & 0.1 \\ 0.2 & -0.4 \end{pmatrix}$
$M^{\langle\rangle}$	Выделение столбца матрицы	$B^{\langle 1 \rangle} = \begin{pmatrix} 1 \\ -2 \end{pmatrix}$

Формирование новых матриц из существующих

В Mathcad есть две функции для объединения матриц вместе – рядом или одна над другой, а также функция для извлечения подматрицы:

Имя функции	Возвращается...
`Augment` (A, B)	Массив, сформированный расположением `A` и `B` рядом. Массивы `A` и `B` должны иметь одинаковое число строк.
`Stack` (A, B)	Массив, сформированный расположением `A` над `B`. Массивы `A` и `B` должны иметь одинаковое число столбцов.
`Submatrix` (A, ir, jr, ic, jc)	Субматрица, состоящая из всех элементов, содержащихся в строках с `ir` по `jc` и столбцах с `ic` по `jc`. Чтобы поддерживать порядок строк и/или

	столбцов, удостоверьтесь, что `ir<=jr` и `ic<=jc`, иначе порядок строк и/или столбцов будет обращен.

Решение систем линейных алгебраических уравнений

Решение системы линейных алгебраических уравнений $Ax = b$ можно получить следующим способом:

$$A := \begin{pmatrix} -5 & 2 \\ 6 & 1 \end{pmatrix} \qquad b := \begin{pmatrix} 3 \\ 1 \end{pmatrix} \qquad x := A^{-1} \cdot b \qquad x = \begin{pmatrix} -0.059 \\ 1.353 \end{pmatrix}$$

или используя встроенную функцию `lsolve(A,b)`:

$$A := \begin{pmatrix} -5 & 2 \\ 6 & 1 \end{pmatrix} \qquad b := \begin{pmatrix} 3 \\ 1 \end{pmatrix} \qquad x := \mathrm{lsolve}(A, b) \qquad x = \begin{pmatrix} -0.059 \\ 1.353 \end{pmatrix}$$

Последовательности и прогрессии

С помощью пакета MathCad можно работать с различными последовательностями, производя операции над их членами, такие, например, как нахождение их суммы, произведения.

Последовательности можно задавать в виде формулы *n*-го члена. Но можно также задавать последовательность посредством рекуррентной формулы, то есть в виде закона, по которому очередной член последовательности выражается через предыдущий (или предыдущие). В частности, такими формулами задаются геометрические и арифметические прогрессии.

Пример 1. Найти первые 10 членов последовательности, заданной формулой *n*-го члена $x_n = 0,5 \cdot 4^n$.

Решение. Очевидно, что решение данной задачи довольно простое, но требует очень большого объема вычислений при расчете вручную. Рассмотрим последовательность действий в MathCad по шагам:

1. Так как нам нужно узнать первые 10 членов данной последовательности, то n должно изменяться от 1 до 10 с шагом 1. Зададим соответствующую дискретную переменную: $n := 1, 2..10$.

2. Зададим общий член нашей последовательности: $x_n := 0.5 \cdot 4^n$ (чтобы ввести нижний индекс, необходимо сначала нажать клавишу «[» или воспользоваться соответсвующей кнопкой палитры Matrix).

3. Выведем на экран все десять членов последовательности x_n и их номера.

4. Получим результат:

$n := 1, 2 .. 10$ $x_n := 0.5 \cdot 4^n$

$n =$ $x_n =$

n	x_n
1	2
2	8
3	32
4	128
5	512
6	$2.048 \cdot 10^3$
7	$8.192 \cdot 10^3$
8	$3.277 \cdot 10^4$
9	$1.311 \cdot 10^5$
10	$5.243 \cdot 10^5$

Аналогично можно рассчитать любое количество членов произвольных последовательностей.

П р и м е р 2. Проверить, является ли число –122 членом арифметической прогрессии 23; 17,2; … и если является, то каким?

Р е ш е н и е.

1) По определению арифметическая прогрессия записывается рекуррентной формулой $a_{n+1} = a_n + d$. В нашем случае $a_1 = 23, \quad d = 17,2 - 23 = -5,8.$

2) Зададим дискретную переменную n, присвоим начальные значения для a_1 и d, зададим рекуррентную формулу для a_n и проверим сначала, например, первые 30 членов данной прогрессии: $n := 1, \ 2 .. 30.$

3) Выведем значения этих 30 членов прогрессии на экран.

Как мы видим, число –122 является 26-м членом нашей прогрессии. Если бы среди этих 30 членов числа –122 не было, мы бы посмотрели на 30-й член и если он меньше –122, то в прогрессии вообще нет такого числа, иначе мы бы искали дальше это число.

Фрагмент документа приведен ниже.

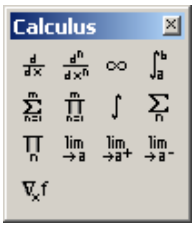
$n := 1, 2 .. 30$

$d := -5.8$ $a_1 := 23$ $a_{n+1} := a_n + d$

n =	a_n =
1	23
2	17.2
3	11.4
4	5.6
5	-0.2
6	-6
7	-11.8
8	-17.6
9	-23.4
...	...

Вычисление сумм и произведений

При вычислении сумм или произведений нужно пользоваться палитрой Calculus и выбирать соответствующий шаблон.



Если вычисляется сумма с конечным числом слагаемых, тогда используем шаблон $\sum_{n=1}^{m}$ («Ctrl» + «Shift» + «$»), если по значениям ранжированной переменной – тогда шаблон $\sum_{n}$ («Shift» + «$»), если же суммируются значения компонент некоторого вектора – тогда шаблон $\sum \square$ который появляется после нажатия клавиш «Ctrl» [4]. Затем заполняем поочередно все поля шаблона.

Аналогично выбираются шаблоны и для вычисления произведений.

Пример 1. Найти сумму первых десяти членов геометрической прогрессии b_n, в которой $b_1 = 3$ и $q = 0{,}5$.

Решение.

1) Зададим дискретную переменную n, присвоим начальные значения для b_1 и q, зададим рекуррентную формулу для b_n.

2) Нажмем комбинацию клавиш [Ctrl] [Shift] [$] или щелкнем по шаблону в математической палитре. Появится значок оператора суммы: $\sum_{\square=\square}^{\square} \square$, где ■ – пустые поля ввода.

3. Заполним соответствующие поля. В первом нижнем поле будет переменная n, которая является индексом нашей прогрессии, во втором нижнем поле – ее начальное значение, а в верхнем поле – конечное значение переменной n. В среднем поле под знаком суммы запишем n-й член нашей последовательности. Получим: $\sum_{n=1}^{10} b_n$.

4. Нажав клавишу «=», получим ответ: 5,994.

$$n := 1..10$$

$$q := 0.5 \qquad b_1 := 3 \qquad b_{n+1} := b_n \cdot q$$

$$\sum_{n=1}^{10} b_n = 5.994$$

П р и м е р 2. Найти значение конечной суммы $\sum_{n=1}^{1000} \frac{2n+1}{n^2(n+1)^2}$.

Р е ш е н и е:

$$N := 1000 \qquad n := 1, 2..N \qquad \sum_n \left[\frac{2 \cdot n + 1}{n^2 \cdot (n+1)^2} \right] = 1$$

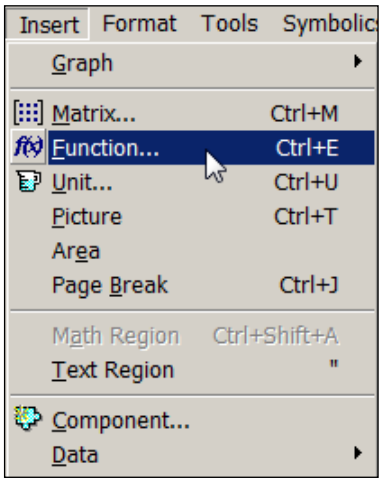
Аналогичным образом можно посчитать и произведение членов любой последовательности. Операция произведения вызывается из палитры кнопкой  («Ctrl» + «Shift» + «#») и имеет следующий вид: $\prod_{\square=\square}^{\square} \square$, также кнопкой  («Shift» + «#») и имеет вид: $\prod_{\square} \square$, где ■ – поля ввода такие же, как в операции суммирования.

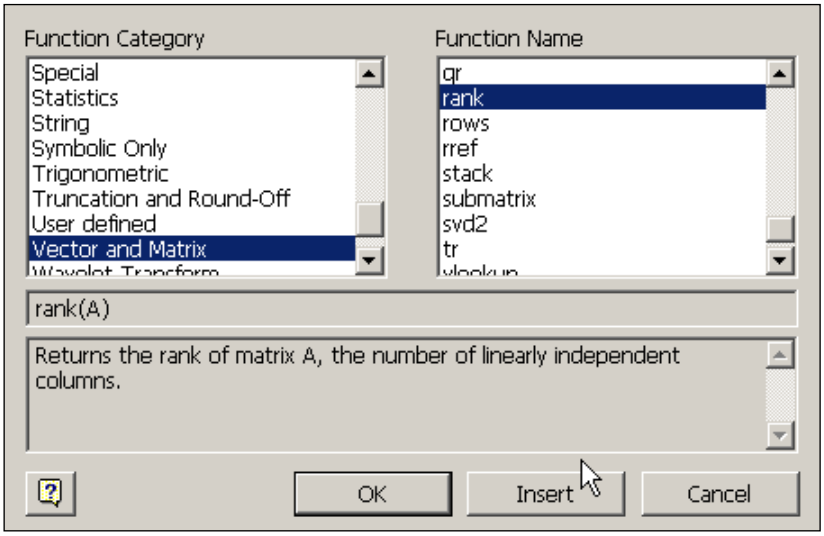
Функции

MathCad обладает множеством различных функций, но кроме них пользователь может определять собственные. Вызывается функция посредством ее имени, после которого следуют в круглых скобках аргументы функции (через запятую, если их несколько).

Функции можно вставлять в выражение с помощью кнопки  на панели меню или пункта Function меню Insert.

В появившемся далее окне можно просмотреть список всех встроенных функций MathCad, выбрать из них необходимую и на месте параметров вставить нужные имена.





Функции пользователя

Определение функции пользователя:

<имя_функции>(*<список_аргументов>*) := *<выражение>*

Тип аргумента не указывается, а распознается системой при вызове функции. Аргументы могут быть скалярами, матрицами, функциями. Для задания функций сложной структуры используют подпрограммы. Примеры описания и использования функций:

Набираем	Видим	Вызов функций
f(x):1+x^2[ПРОБЕЛ]+x^3	$f(x) := 1 + x^2 + x^3$	$f(5) = 151$
h(t):t+6[ПРОБЕЛ]/t^2[ПРОБЕЛ]+1	$h(t) := \frac{t + 6}{t^2 + 1}$	$h(5) = 0.423$

Заданные функции можно вызывать и от аргументов других типов. Например, в следующем случае аргументом заданных выше функций является дискретная переменная и, следовательно, результатом будет вектор:

$f(x) := 1 + x^2 + x^3$ $\quad h(t) := \dfrac{t + 6}{t^2 + 1}$

$x := 0 .. 6$

$f(x) =$

1
3
13
37
81
151
253

$h(x) =$

6
3.5
1.6
0.9
0.588
0.423
0.324

Производные и первообразные

При вычислении производных или первообразных нужно пользоваться палитрой Calculus, выбирать соответствующий шаблон и заполнять поля согласно поставленной задаче.

$f(x) := 1 + x^2 + x^3$ $\quad h(t) := \dfrac{t + 6}{t^2 + 1}$ $\quad x := 0 .. 5$

$\dfrac{d}{dx} f(x) =$

$2.556 \cdot 10^{-14}$
5
16
33
56
85

$\dfrac{d^2}{dx^2} h(x) =$

-12
2.5
1.088
0.348
0.136
0.063

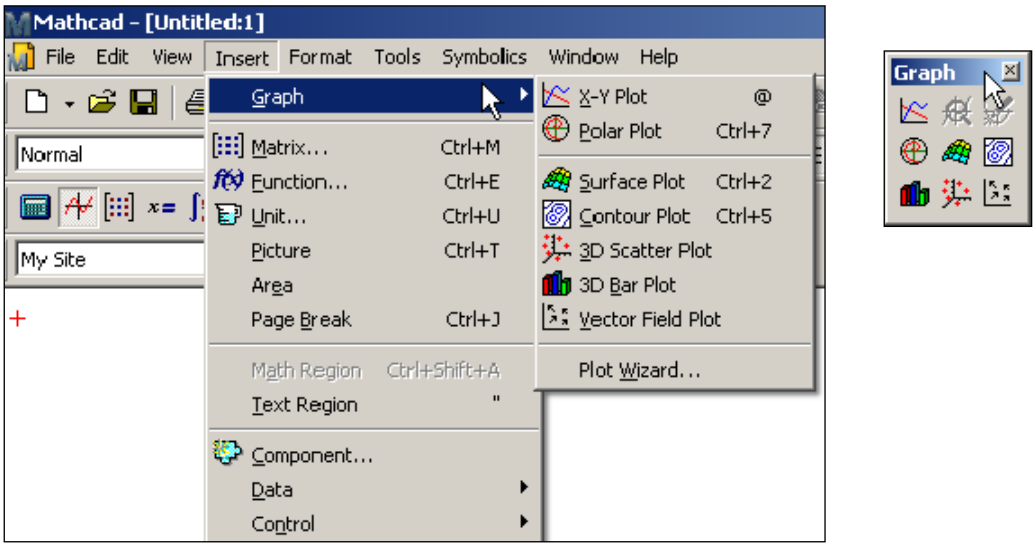
$g(x) := \int f(x)\, dx \rightarrow \dfrac{x^4}{4} + \dfrac{x^3}{3} + x$

$a := \int_{-1}^{1} h(t)\, dt$

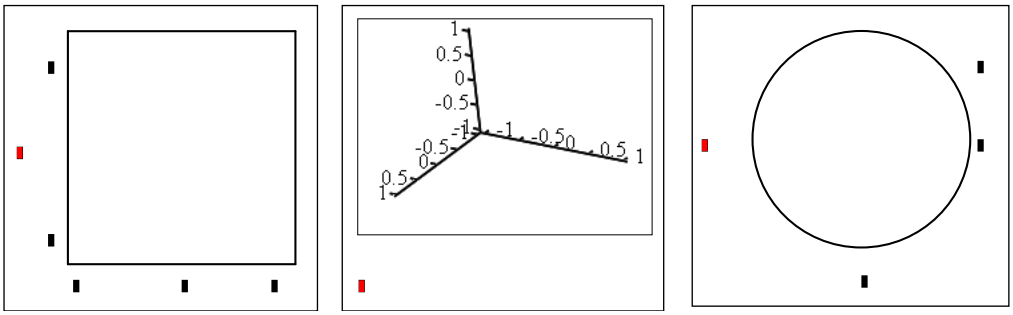
$a = 9.425$

Работа с графикой

Команда Insert ► Graph или же палитра Graph дает доступ к шаблонам графиков.



Шаблонов графиков всего три: в двумерной и трехмерной декартовой системах координат, а также в полярной системе координат. В появившемся шаблоне есть поля, которые заполняет пользователь, а есть и заполняющиеся автоматически.

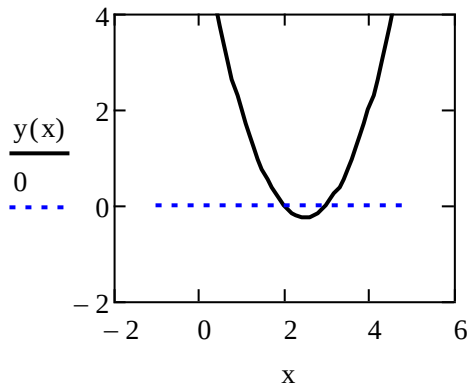


Построение двумерных графиков в декартовой системе координат

Для построения графиков функций одной переменной $y(x)$ в декартовой системе координат MathCad предусматривает два способа: упрощенный способ без первоначального задания значений дискретной переменной x (тогда пределы изменения аргумента x автоматически задаются от –10 до 10) и обычный способ с заданием дискретной переменной x. При построении графиков, заданных параметрически, поступим, как и ранее. Слева укажем функцию $y(t)$, а внизу – функцию $x(t)$ от дискретной переменной t.

С помощью одного шаблона можно строить графики нескольких функций, перечислив их через запятую (*они отображаются разными линиями и под каждым именем функции на графике рисуется образец этой линии*):

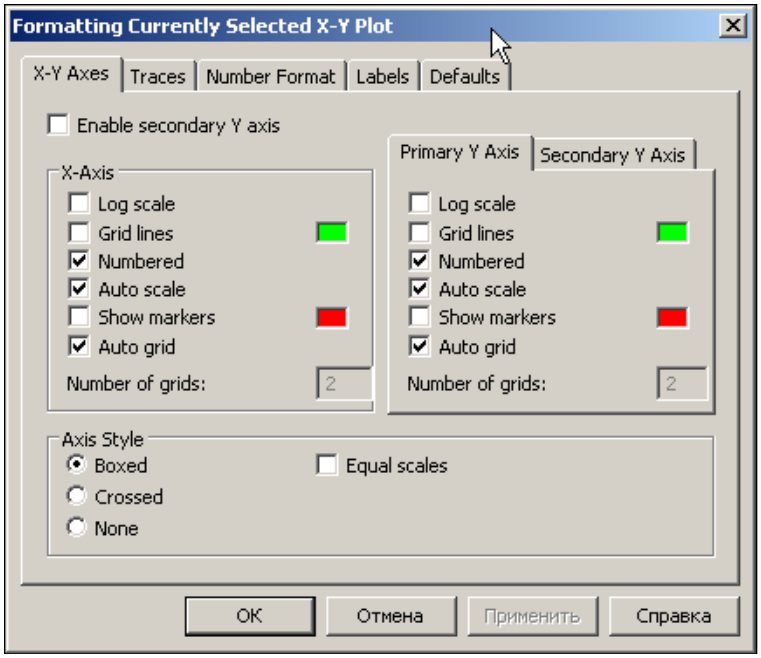
$x := -1, -0.9 .. 5$ $y(x) := x^2 - 5 \cdot x + 6$



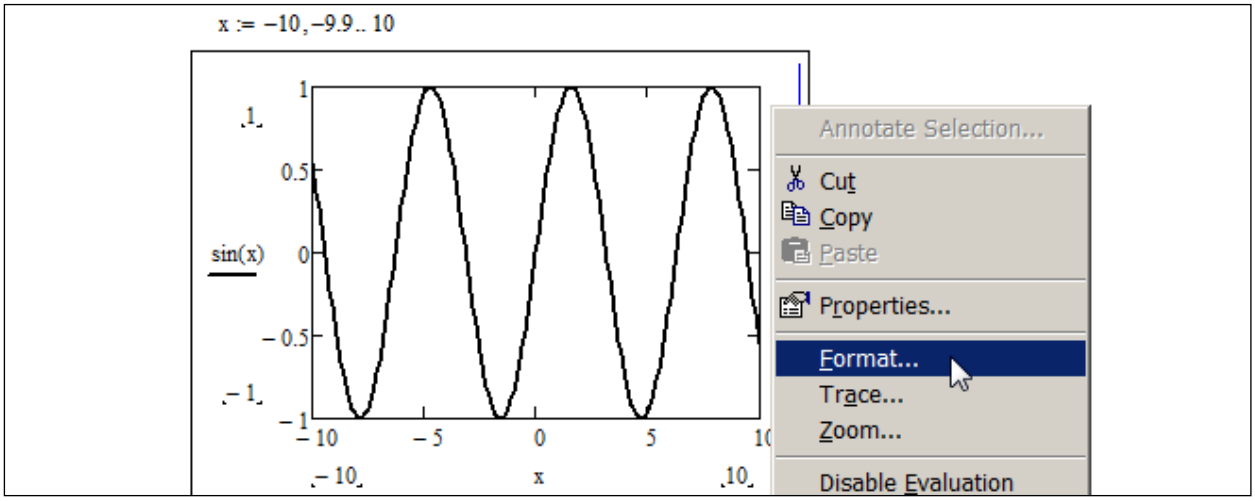
Простейшие приемы форматирования графиков

Если дважды щелкнуть мышью на графике, появится окно форматирования, которое имеет пять вкладок:

- Axes (оси X–Y) – задание параметров отображения осей;
- Traces – задание параметров отображения линий графика;
- Number Format – задание параметров отображения чисел графика;
- Labels – задание параметров отображения меток (надписей) у осей;
- Defaults – задание параметров по умолчанию.



Кроме того, ряд команд форматирования графиков имеется в команде Format контекстного меню, которое вызывается щелчком на графике правой кнопкой мыши.

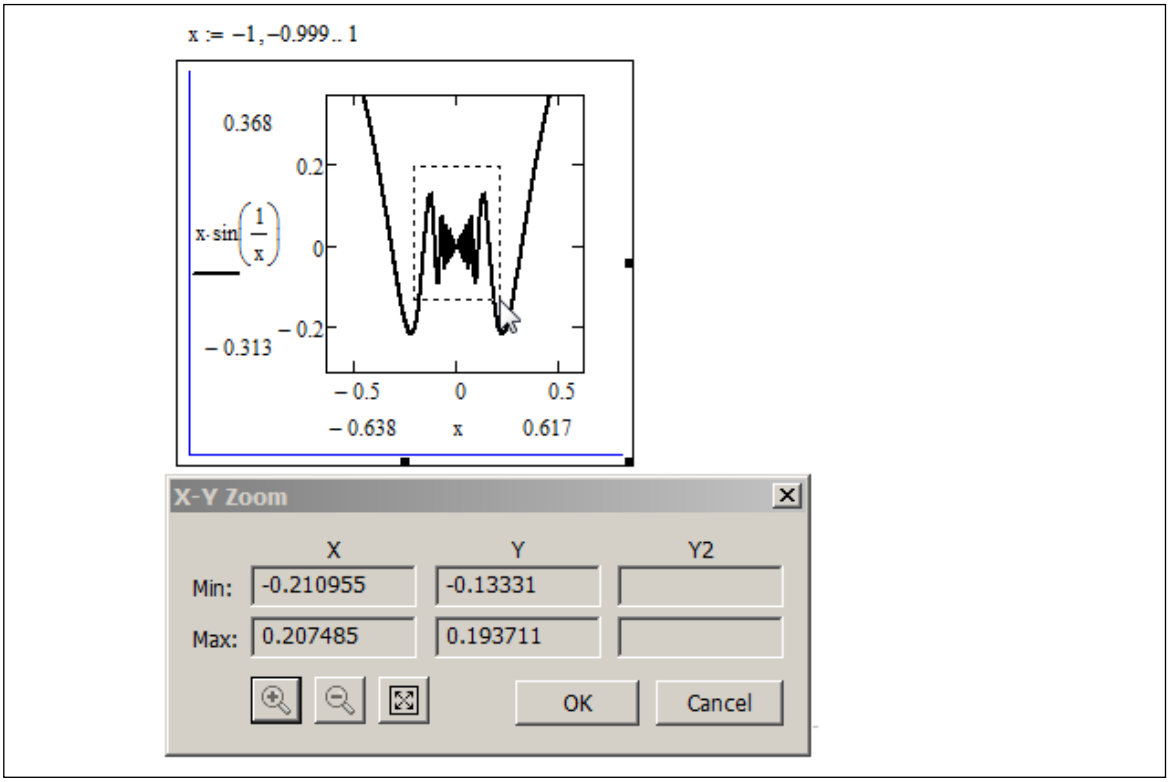


Команда Format содержит вкладки, которые позволяют изменять такие параметры, как масштаб по осям (линейный или логарифмический), задавать координатную сетку (линии сетки с нумерацией или без нее) и значения координат по осям, менять внешний вид графика, цвет и толщину линий, угол зрения и т.п. Все эти параметры достаточно очевидны.

Для того чтобы получить координаты любой точки графика, сначала выделим график, щелкнув мышью на графике, – MathCad заключит его в синюю рамку. Затем щелкнем правой кнопкой мыши и вызовем команду Trace. Появится окно X–Y Trace. Теперь, если нажать кнопку мыши на кривой графика и двигать мышь по этой кривой, то в окне отобразятся координаты (x, y) точки, на которую указывает мышь.

Кнопки (Zoom), (Unzoom) и (Full View) позволяют увеличить (уменьшить) выделенную часть графика или снять выделение и вернуться к просмотру всего графика.

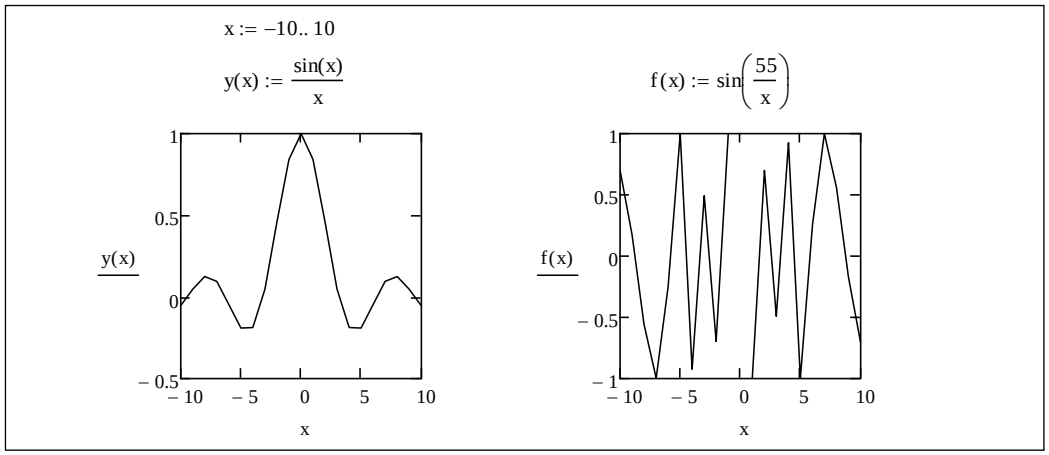
Выделенный участок графика увеличивается до размера всего окна просмотра. При этом поведение функции становится более понятным.



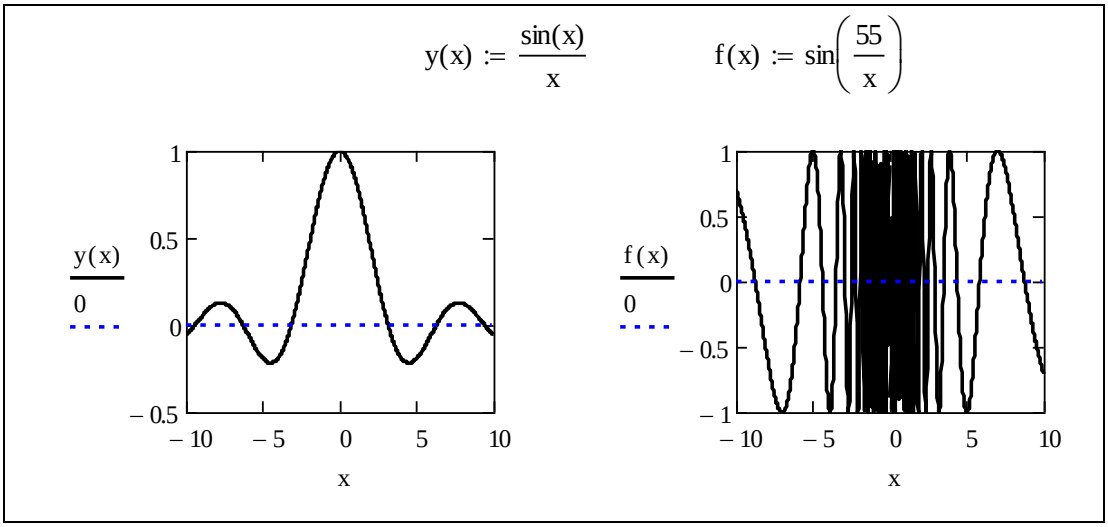
Мы видели, что MathCad позволяет наложить друг на друга графики нескольких функций (они отображаются разными линиями и под каждым именем функции на графике рисуется образец этой линии). Это дает широкие возможности в исследовании функций, такие, как нахождение точек пересечения их графиков, исследование промежутков, на которых, например, одна из функций принимает большие значения, чем другая (таким образом, кстати, можно решать неравенства).

З а м е ч а н и е**.** При построении графиков нужно аккуратно относиться к заданию ранжированной переменной, так как в некоторых случаях это может привести к глубокому искажению формы графиков. Рассмотрим следующие примеры. Зададим вначале большой шаг изменения дискретной переменной x при построении функций $y(x)=\dfrac{\sin(x)}{x}$, $f(x)=\sin\left(\dfrac{55}{x}\right)$ и построим их графики, а затем изменение дискретной переменной x оставим по умолчанию и получим следующие графики:

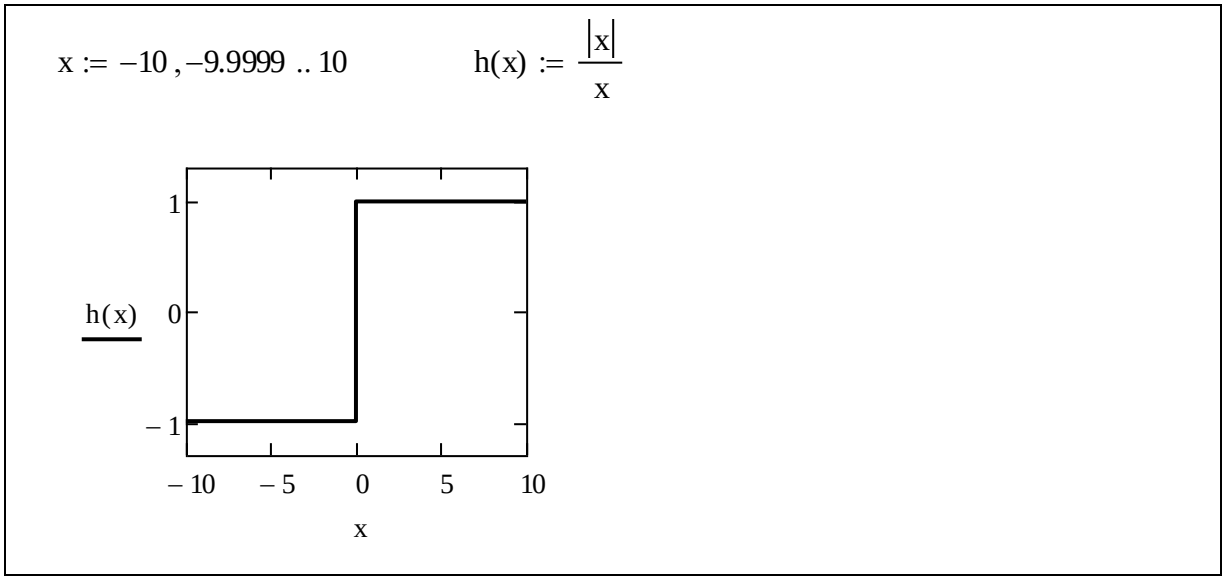
Первый вариант:



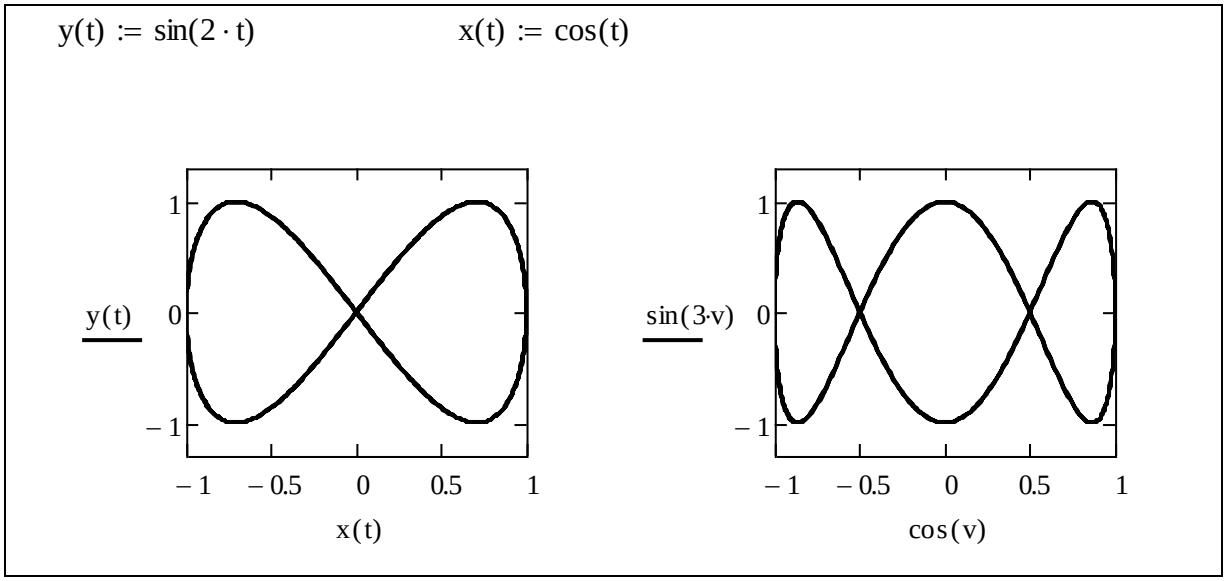
Второй вариант:



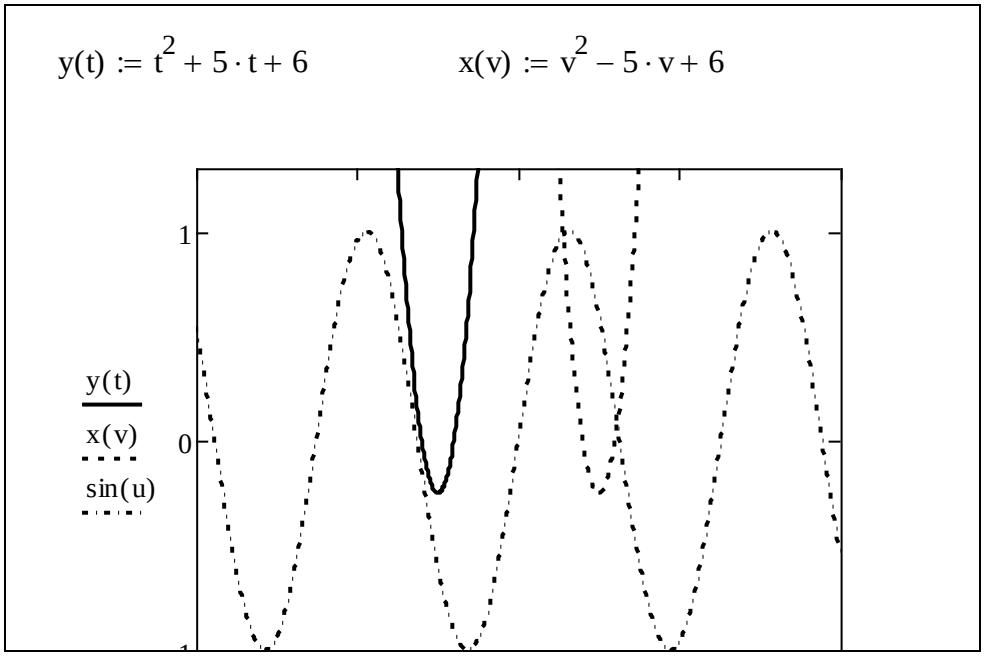
Надо всегда иметь в виду: как бы мы ни задавали шаг изменения дискретной переменной *x*, ступенчатые функции на графике отражаются неверно (так как на месте ступеньки рисуется сплошная линия). Это хорошо видно на следующем примере:



При построении графиков, заданных параметрически, следует поступать так же, как и ранее. Слева указывается функция $y(t)$, а внизу – функция $x(t)$. Задавать значения дискретной переменной необязательно:



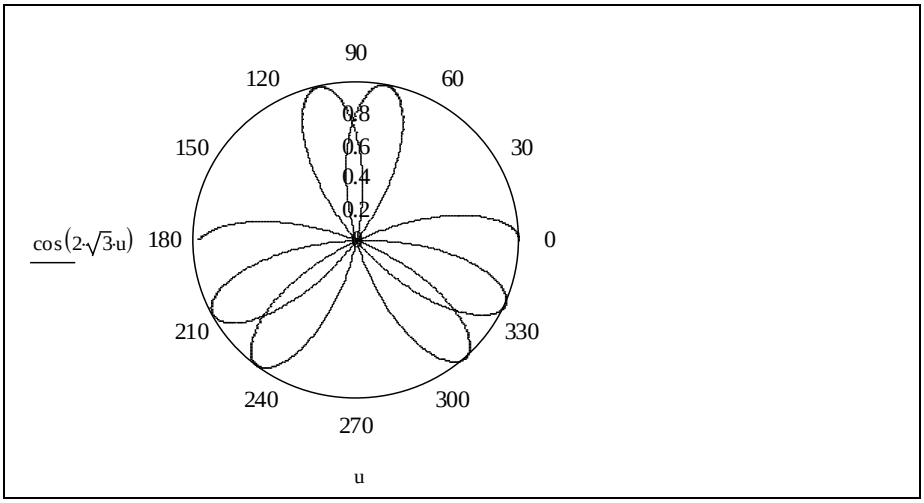
Приведем еще один пример построения нескольких функций (каждой от своего аргумента) на одном графике. Как видно из нижеприведенного изображения, это делается очень просто. Если в дальнейшем отформатировать этот график, то по нему можно выполнить некоторые исследования поведения совокупности функций:



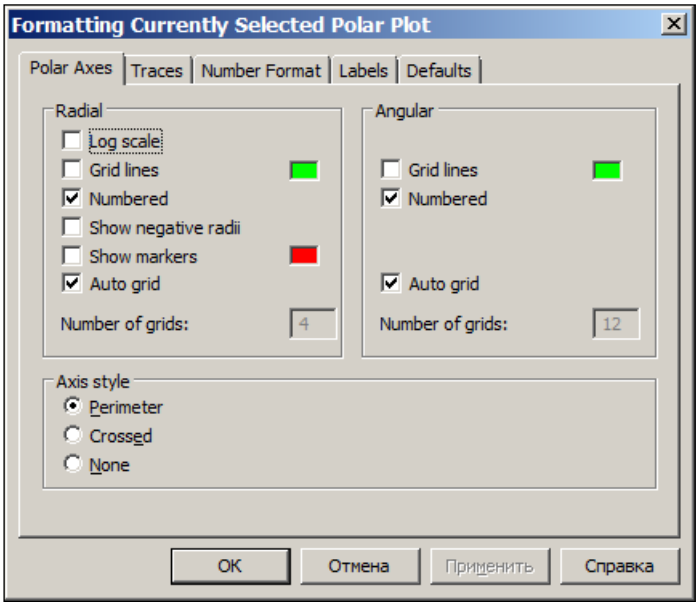
Построение графиков в полярной системе координат

В полярной системе координат каждая точка задается углом u и длиной его радиус-вектора $R(u)$. Как и ранее, можно строить графики, не задавая значений

дискретной переменной, и тогда по умолчанию эти значения будут выбираться из множества $[0, 2\pi]$. Чтобы получить шаблон графика в полярной системе координат, достаточно нажать клавиши «Ctrl» + «7» и заполнить два поля – слева указать одну или несколько функций, а внизу отобразить аргументы этих функций:



Двойной щелчок на графике предоставляет доступ к форматированию графиков в полярных координатах.



Как видно, окно форматирования имеет пять вкладок:

- Polar axes («Полярные оси») – задание параметров отображения осей;
- Traces («Графики») – задание параметров отображения линий графика;
- Number Format («Числовой формат») – задание параметров отображения чисел графика;
- Labels («Надписи») – задание параметров отображения меток (надписей) у осей;
- Defaults («По умолчанию») – задание параметров по умолчанию.

Работа с трехмерной графикой

Уравнением поверхности называется уравнение

$$F(x, y, z) = 0,$$

которому удовлетворяют координаты любой точки поверхности и только они.

Линия в пространстве как пересечение двух поверхностей определяется двумя уравнениями:

$$\begin{cases} F_1(x, y, z) = 0, \\ F_2(x, y, z) = 0. \end{cases}$$

Трехмерная графика используется тогда, когда надо получить визуальное представление о сложных поверхностях и объемных фигурах, расположенных в пространстве. С ее помощью можно строить пересекающиеся объекты, потом, изменяя угол видимости, анализировать линии пересечения.

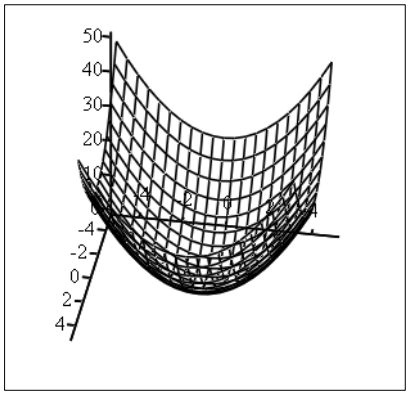
Построение поверхностей по матрице аппликат их точек

Если поверхность описывается по закону $z = f(x, y)$, то можно построить поверхность по точкам $z = f(x_i, y_j)$, $i = \overline{0, n}$, $j = \overline{1 = 0, m}$, значения которых хранятся в некоторой матрице. После этого командой Surface Plot (или клавишами [Ctrl] [2]) ввести шаблон графика поверхности, который содержит единственное место ввода – темный прямоугольник у левого нижнего угла. В него надо занести имя матрицы со значениями аппликат поверхности и щелкнуть вне графика.

Построение трехмерных графиков без задания матрицы

Пусть снова поверхность описывается по закону $z = f(x, y)$. В последних версиях MathCad поверхность можно построить уже без явного задания матрицы аппликат столь же просто, как и в случае двумерного графика. Для этого командой Surface Plot (или клавишами [Ctrl] [2]) вводится шаблон графика поверхности, и указывается зависимость $z = f(x, y)$.

$f(x,y) := x^2 + y^2$



f

Отличительным моментом в данных вариантах изображения поверхностей являются обозначения на осях, вернее неопределенность в масштабировании. Поэтому для получения более точного изображения поверхности потребуется ее дальнейшее форматирование.

Построение параметрически заданных поверхностей

Не всегда можно поверхность задать зависимостью $z = f(x, y)$. Но в математике имеется второй способ задания поверхности – параметрический: $x = f_1(u,v),\ y = f_2(u,v),\ z = f_3(u,v),\ u_0 \le u \le u_1,\ \ v_0 \le v \le v_1$.

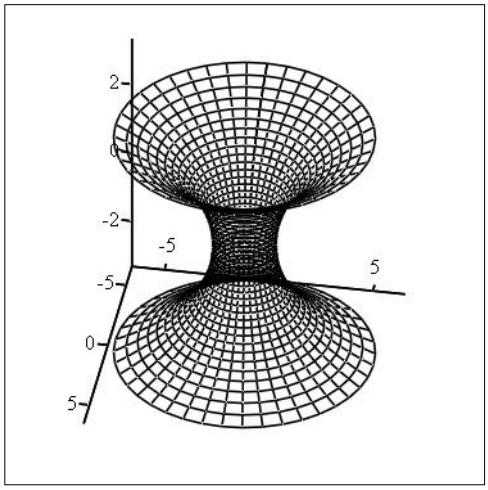
Для построения поверхности такого вида нужно сформировать три матрицы: *X* – для задания значений по оси *Ox*, *Y* – для задания значений по оси *Oy*, *Z* – для задания значений по оси *Oz* и далее указать их блоком в шаблоне графика.

Пусть, например, задана поверхность в виде: $x = c\,\mathrm{ch}(u/c)\cos(v)$, $y = c\,\mathrm{ch}(u/c)\sin(v)$, $z = u$ (c – параметр). Построим ее:

$n := 20$ $c := 1.5$

$i := 0 .. 2n$ $u_i := (i - n)\cdot\frac{\pi}{n}$ $j := 0 .. 2n$ $v_j := (j - n)\cdot\frac{\pi}{n}$

$X_{i,j} := c\cdot\cosh\left(\frac{u_i}{c}\right)\cdot\cos(v_j)$ $Y_{i,j} := c\cdot\cosh\left(\frac{u_i}{c}\right)\cdot\sin(v_j)$ $Z_{i,j} := u_i$

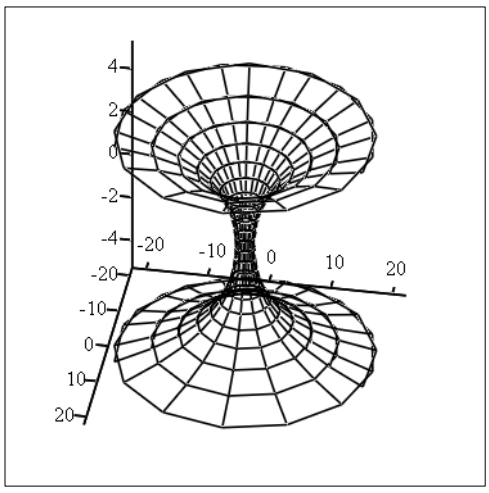


(X, Y, Z)

Можно поступить и так:

$c := 1.5$

$X(u,v) := c\cdot\cosh\left(\frac{u}{c}\right)\cdot\cos(v)$ $Y(u,v) := c\cdot\cosh\left(\frac{u}{c}\right)\cdot\sin(v)$ $Z(u,v) := u$



(X, Y, Z)

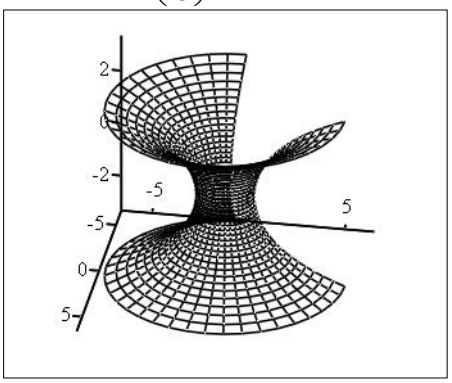
Построение трехмерных фигур с вырезом

Если в предыдущем примере изменение аргумента задать в меньших границах, тогда получим фигуру с вырезом:

$$n := 20 \qquad np := \frac{\pi}{n} \qquad i := 0..2n \qquad u_i := (i - n)\cdot np$$

$$c := 1.5 \qquad j := 0..1.5n \qquad v_j := (j - n)\cdot np$$

$$X_{i,j} := c\cdot\cosh\left(\frac{u_i}{c}\right)\cdot\cos\left(v_j\right) \qquad Y_{i,j} := c\cdot\cosh\left(\frac{u_i}{c}\right)\cdot\sin\left(v_j\right) \qquad Z_{i,j} := u_i$$

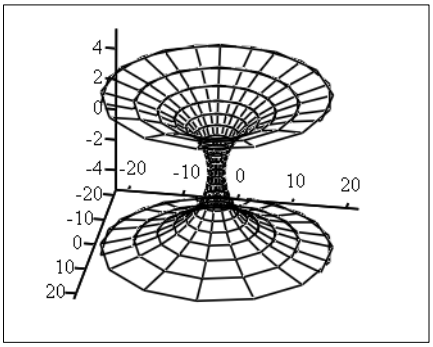


(X, Y, Z)

Построение графика поверхности, заданной в векторной параметрической форме

Уравнения поверхности записываются в векторной форме, и на шаблоне графика в заполняемом поле указывается имя этого вектора:

$$c := 1.5 \qquad F(u,v) := \begin{pmatrix} c\cdot\cosh\left(\frac{u}{c}\right)\cdot\cos(v) \\ c\cdot\cosh\left(\frac{u}{c}\right)\cdot\sin(v) \\ u \end{pmatrix}$$



F

Судя по обозначениям на осях, получены несколько другие границы изменения параметров u, v.

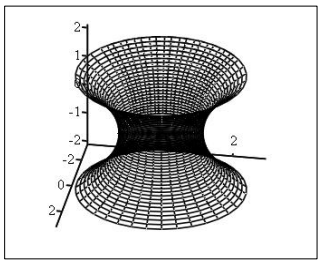
Применение функции CreateMesh

В MathCad встроена графическая функция для задания поверхностей $\mathrm{CreateMesh}(F[,u0,u1,v0,v1][,u,v][,fmap])$, которая возвращает массив из трех векторов, представляющий абсциссы, ординаты и аппликаты параметрически

заданной поверхности $F(u,v)$. Параметры $u0,u1,v0,v1$ задают пределы изменения переменных u,v. Параметр *fmap* – трехэлементный вектор, задающий число линий в сетке изображаемой функции.

$$c := 1.5 \qquad F(u,v) := \begin{pmatrix} c\cdot\cosh\left(\frac{u}{c}\right)\cdot\cos(v) \\ c\cdot\cosh\left(\frac{u}{c}\right)\cdot\sin(v) \\ u \end{pmatrix}$$

$$A := \text{CreateMesh}(F,-2,2,-4,4,45)$$



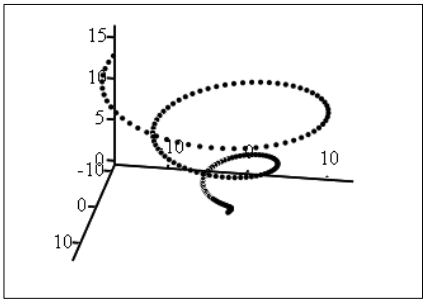
A

Применение функции CreateSpace

График типа Scatter Plot позволяет получить изображение пространственной кривой. Для этого в MathCad встроена графическая функция CreateSpace, которую можно использовать в двух форматах.

Функция CreateSpace($F[,t0,t1][,t][,$ *fmap*$]$) возвращает массив из трех векторов, представляющий абсциссы, ординаты и аппликаты параметрически заданной пространственной линии в виде вектор–функции F от трех переменных. Параметры $t0,t1$ задают пределы изменения переменной t. Параметр *fmap* – трехэлементный вектор, задающий число линий в сетке изображаемой функции. Например:

$$H(t) := \begin{pmatrix} t\cdot\sin(t) \\ t\cdot\cos(t) \\ t \end{pmatrix} \qquad A := \text{CreateSpace}(H,0,16,160)$$

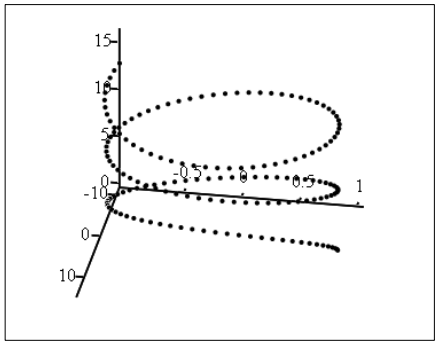


A

Функция CreateSpace($F1,F2,F3[,t0,t1][,t][,$ *fmap*$]$) возвращает массив из трех векторов, представляющий абсциссы, ординаты и аппликаты параметрически заданной пространственной линии в виде вектор–функции F от трех переменных.

$F1(t) := t \cdot \sin(t)$ $F2(t) := \cos(t)$ $F3(t) := t$

$A := \text{CreateSpace}(F1, F2, F3, 0, 16, 160)$



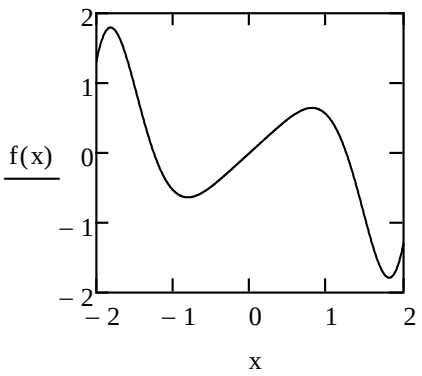
A

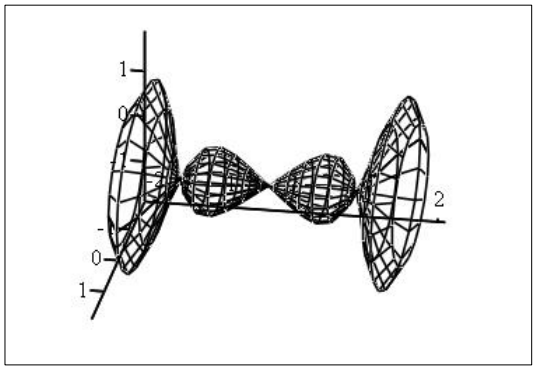
Построение объемной фигуры, образованной вращением кривой

При построении объемной фигуры, образованной вращением кривой, заданной функцией $y = f(x)$, вокруг оси Oy, можно снова воспользоваться функцией CreateMesh:

$f(x) := x \cdot \cos\left(x^2\right)$ $F(u, v) := \begin{pmatrix} f(u) \cdot \cos(v) \\ u \\ f(u) \cdot \sin(v) \end{pmatrix}$

$A := \text{CreateMesh}(F, -2, 2, 0, 2 \cdot \pi)$

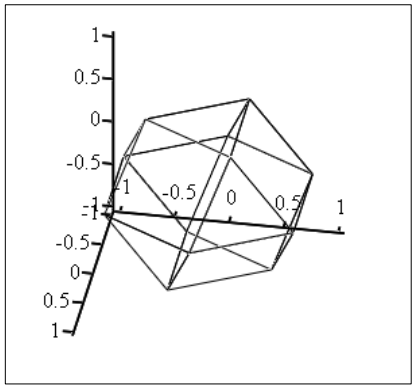




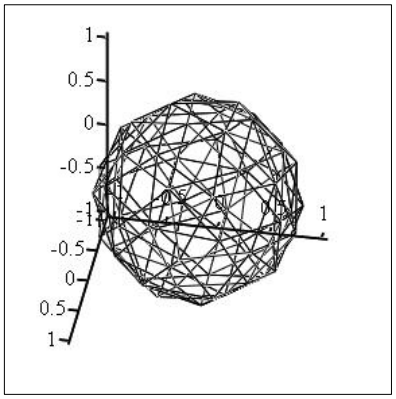
A

Построение объемной фигуры
с помощью функции Polyhendron

Для построения ряда известных в математике стандартных объемных фигур (полиэдров) можно использовать встроенную функцию Polyhendron(“name”), где “name” – это или имя фигуры, или ее номер, заданный как “#N”.



Polyhedron("#12")



Polyhedron("#37")

Для получения характеристик полиэдра можно использовать встроенную функцию PolyLoop(“name”), где “name” – это или имя фигуры, или ее номер, заданный как “#N”, или “описатель”:

$$\text{PolyLookup("\#12")} = \begin{pmatrix} \text{"cuboctahedron"} \\ \text{"rhombic dodecahedron"} \\ \text{"2|3 4"} \end{pmatrix}$$

Работу с трехмерной графикой закрепим на примере решения нескольких следующих задач.

Задача 1

Построить поверхность $z = f(x, y)$ по матрице аппликат ее точек.

Задание

По матрице аппликат построить поверхность $z = x^2 - y^2$.

Алгоритм решения

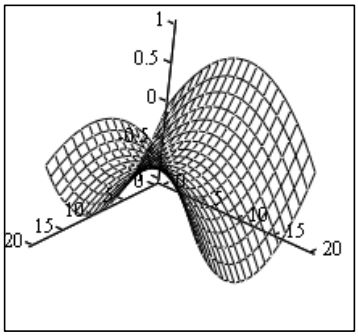
1. Построим матрицу *M*, содержащую точки
   $z = f(x_i, y_j),\ i = \overline{0, n},\ \ j = \overline{0, m}.$
2. Командой Surface Plot (или клавишами «Ctrl» + «2») введем шаблон графика поверхности.
3. В место ввода матрицы аппликат занесем имя матрицы *M* и щелкнем вне графика.

Выполнение задания

$i := 0..20 \quad j := 0..20$

$x_i := -1 + i \cdot 0.1 \quad y_j := -1 + j \cdot 0.1$

$f(x,y) := x^2 - y^2 \quad M_{i,j} := f(x_i, y_j)$ - матрица апликат



M

Замечание. На графике появилась поверхность в виде каркаса, которую нужно отформатировать. При форматировании изменяется масштаб построений, угол поворота фигуры относительно осей, удаляются невидимые линии, изменяется механизм окраски и т.д. Все это становится доступным после вызова команды Format. Сделаем это несколько позже.

Задача 2

Построить поверхность $z = f(x, y)$ без задания матрицы аппликат ее точек.

Задание

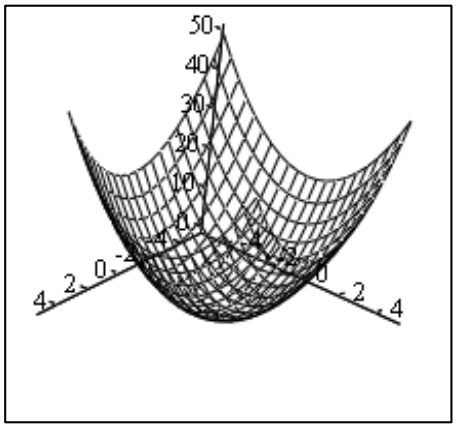
Построить поверхность $z = x^2 + y^2$.

Алгоритм решения

1. Командой Surface Plot (или клавишами «Ctrl» + «2») введем шаблон графика поверхности.
2. В место ввода матрицы аппликат занесем имя функции $f(x, y) = x^2 + y^2$.
3. Щелкнем вне графика.

Выполнение задания

$f(x,y) := x^2 + y^2$



f

З а м е ч а н и е . Отличительным моментом в данных вариантах изображения поверхностей являются обозначения на осях, т.е. неопределенность в масштабировании. Поэтому для получения более точного изображения поверхности потребуется ее дальнейшее форматирование.

Задача 3

Построить поверхность, заданную параметрическим способом: $x = f_1(u,v)$, $y = f_2(u,v)$, $z = f_3(u,v)$, $u_0 \le u \le u_1$, $v_0 \le v \le v_1$.

Задание

Построить следующую поверхность: $x = c\sin u\cos v$, $y = c\sin u\sin v$, $z = c(\mathrm{tg}\frac{u}{2} + \cos u)$ для разных значений параметра c.

Алгоритм решения

1. Сформируем три матрицы: X – для задания значений по оси Ox, Y – для задания значений по оси Oy, Z – для задания значений по оси Oz.
2. Командой Surface Plot (или клавишами «Ctrl» + «2») введем шаблон графика поверхности.
3. В место ввода матриц занесем их блоком.
4. Щелкнем вне графика.

Выполнение задания

$n := 10$ $\qquad c := 1.5$

$i := 1 .. 2n - 1$

$u_i := (i - n) \cdot \frac{\pi}{2n}$

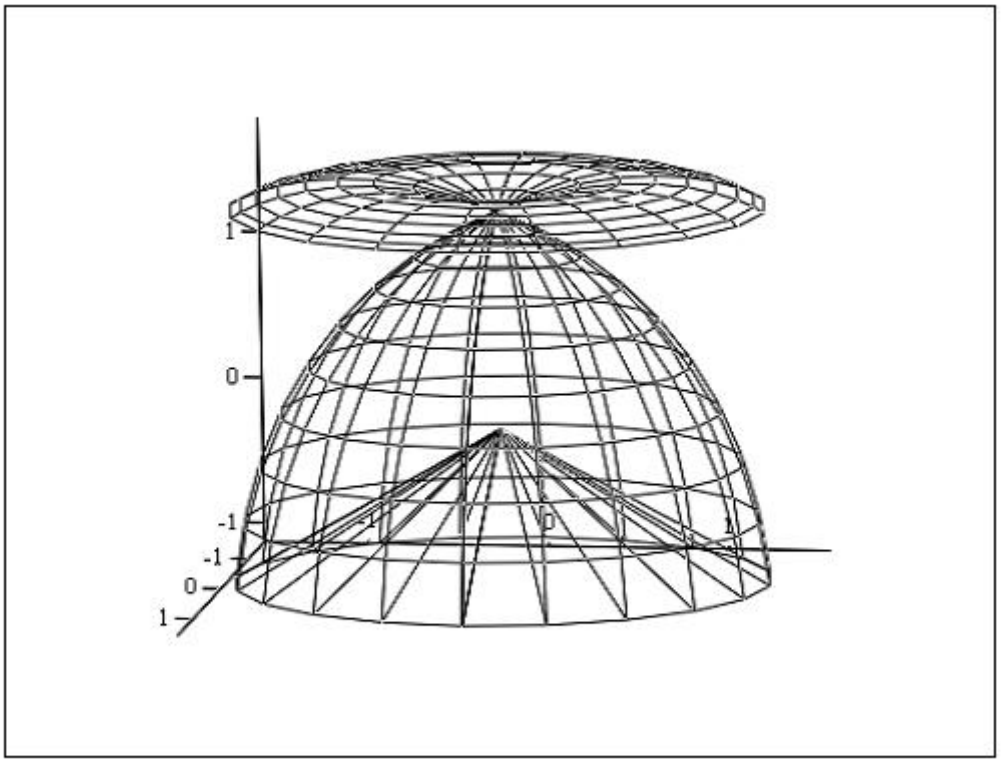
$j := 0 .. 2n$

$v_j := (j - n) \cdot \frac{\pi}{n}$

$X_{i,j} := c \cdot \sin(u_i) \cdot \cos(v_j)$

$Y_{i,j} := c \cdot \sin(u_i) \cdot \sin(v_j)$

$Z_{i,j} := c \cdot \left(\tan\left(\frac{u_i}{2}\right) + \cos(u_i) \right)$



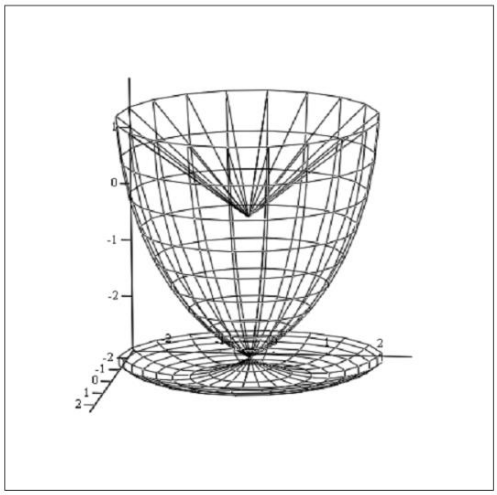
(X, Y, Z)

$c := -2.5$

$X_{i,j} := c \cdot \sin(u_i) \cdot \cos(v_j)$

$Y_{i,j} := c \cdot \sin(u_i) \cdot \sin(v_j)$

$Z_{i,j} := c \cdot \left(\tan\left(\frac{u_i}{2}\right) + \cos(u_i) \right)$



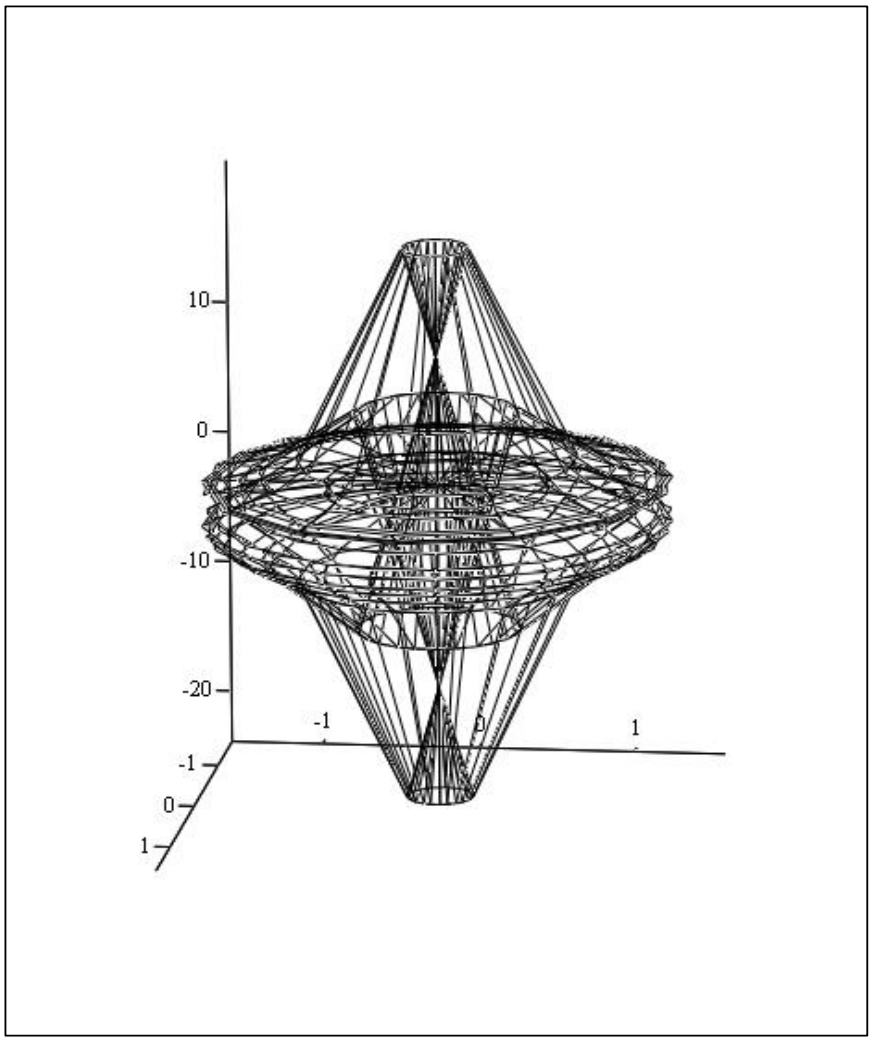
(X,Y,Z)

Можно поступить и так:

$c := 1.5$

$X(u,v) := c \cdot \sin(u) \cdot \cos(v)$ $Y(u,v) := c \cdot \sin(u) \cdot \sin(v)$

$Z(u,v) := c \cdot \left(\tan\left(\frac{u}{2} \right) + \cos(u) \right)$

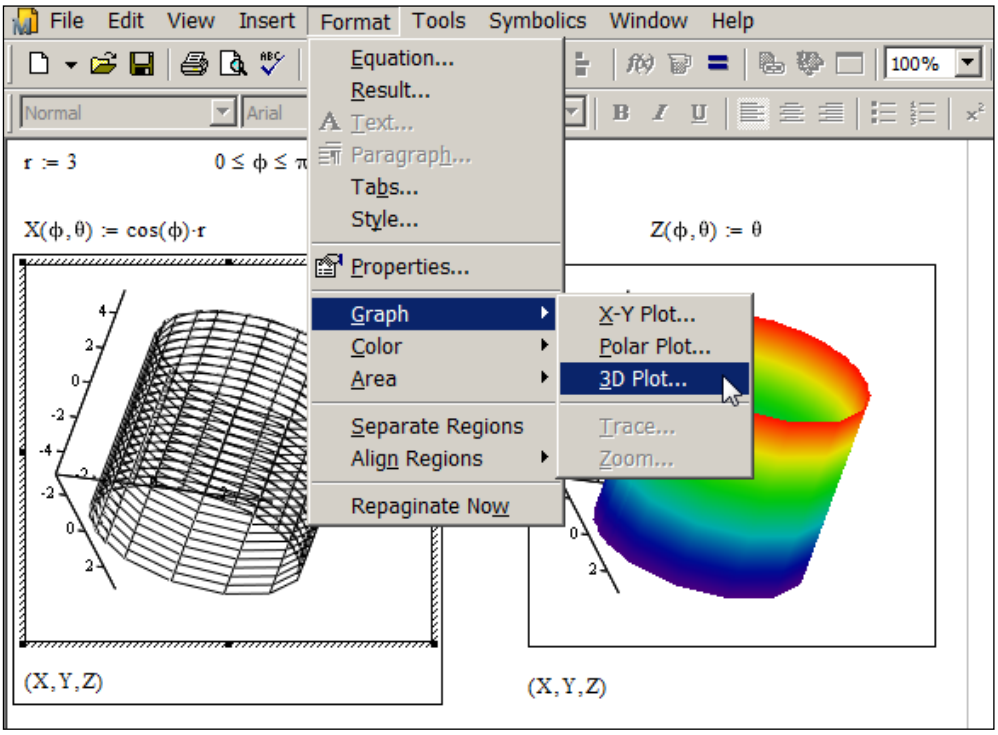


(X,Y,Z)

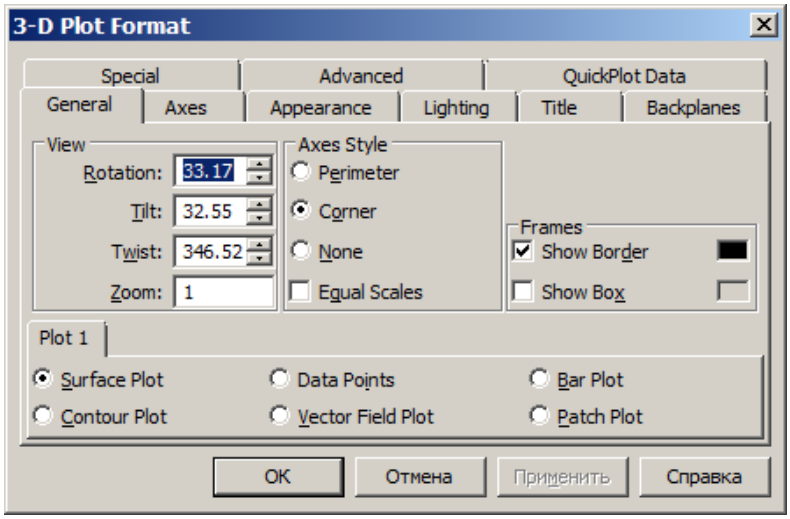
Замечание. Здесь снова появилась неопределенность в масштабировании. Поэтому для получения более точного изображения поверхности потребуется ее дальнейшее форматирование.

Форматирование трехмерных графиков

Построив трехмерный график, приступим к его форматированию. Для этого выполним двойной щелчок мышкой в окне графика – и появится окно форматирования. Его можно вызвать также из контекстного меню, которое появляется после щелчка правой клавишей мышки, или через меню MathCad.



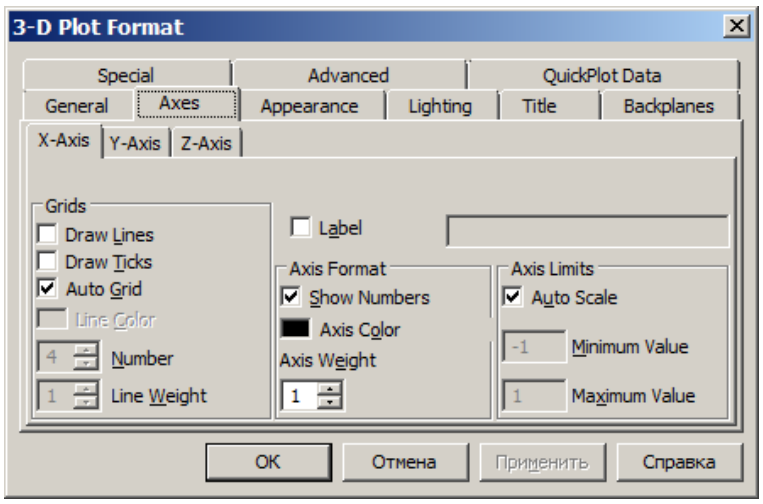
Щелкнув на 3D Plot, получим окно форматирования.



Каждая вкладка отражает определенное состояние графика.

1. General («Общие») – задание общих параметров изображения. Эта вкладка содержит сведения об углах обзора фигуры, объединенных в группе View, стиле осей, объединенных в группе Axes Style, и внешнем обрамлении графика, которые объединены в группе Frame.

2. Axes («Оси») – задание параметров координатных осей (тип, толщина и цвет линий осей, число отметок, их нумерация, масштаб и др.). Внутри этой вкладки имеется еще три: X-Axes, Y-Axes, Z-Axes.



На этих вкладках задаются объединенные в группы параметры для каждой из координатных осей *Ox*, *Oy*, *Oz*. Группа параметров Grids («Сетка») позволяет установить вид координатной сетки (с выводом линии сетки или без, с выводом делений по осям или без и др.). Группа параметров Axes Format («Формат осей») обеспечивает установку формата координатных осей. Группа параметров Axes Limit («Предельные значения по осям») позволяет задать пределы изменения координат.

3. Appearance («Вид») – задание параметров изображения графика. Группа команд Fill Options («Параметры закрашивания») позволяет установить параметры окраски поверхностей и контурных линий. Группа команд Line Options («Параметры линий») позволяет установить параметры отображения линий и их цвет. Группа команд Point Options («Параметры точек») позволяет установить параметры отображения разными символами и их цвет. В каждой группе имеются переключатели для выбора схемы окрашивания Colormap («Цветовая карта») или Solid Color («Основной цвет»). Параметры данной группы существенно влияют на вид рисунка.

4. Title («Заголовок») – задание титульных надписей и их параметров.

5. Lighting («Освещение») – задание условий освещения и выбор схемы освещения.

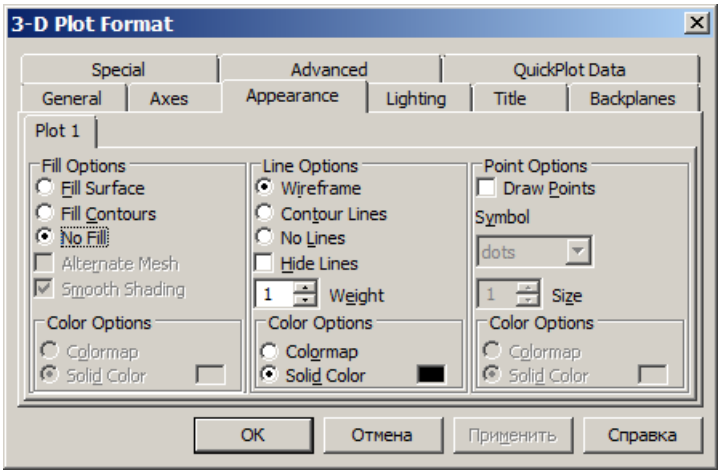
6. Backplanes («Грани») – задание параметров трех граней.

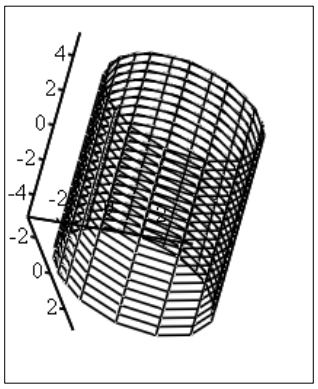
7. Special («Специальные») – задание видов графиков (контурные линии, столбцы, интерполяция по цвету и другие параметры).

8. Advanced («Дополнительно») – задание дополнительных параметров отображения (перспектива, световые эффекты, качество печати и др.).

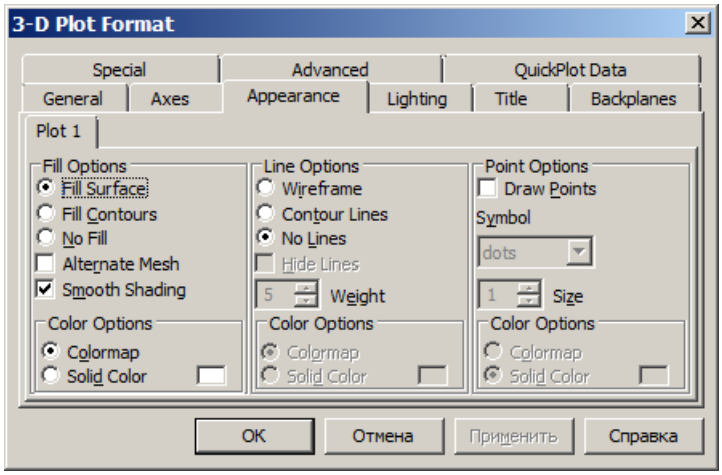
9. QuickPlot Data («Быстрое построение графика по данным») – задание параметров быстрого построения графиков без задания матриц аппликат поверхностей.

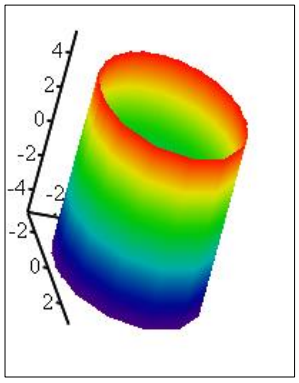
Со многими вкладками вам придется поэкспериментировать самостоятельно, а мы изменим только некоторые параметры вида. Сравните вкладки до форматирования и после внесения изменений.





(X, Y, Z)



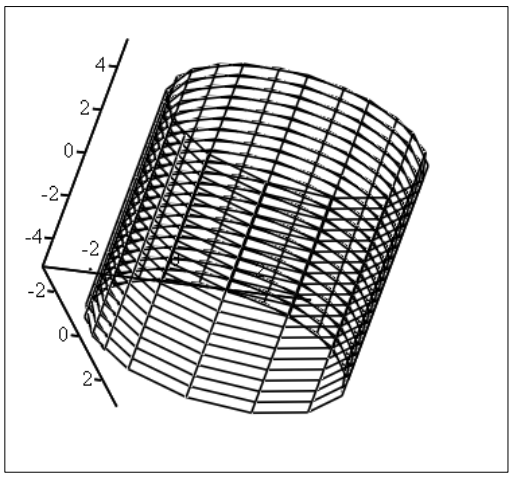


(X, Y, Z)

Проанализируйте, что было изменено на вкладке, и какой график получился в итоге:

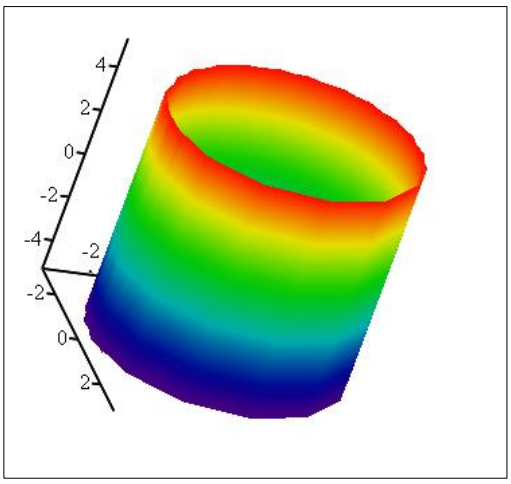
$r := 3$	$0 \le \phi \le \pi$	$0 \le \theta \le \pi \cdot 2$
$X(\phi, \theta) := \cos(\phi) \cdot r$	$Y(\phi, \theta) := \sin(\phi) \cdot r$	$Z(\phi, \theta) := \theta$

До форматирования



(X, Y, Z)

После форматирования



(X, Y, Z)

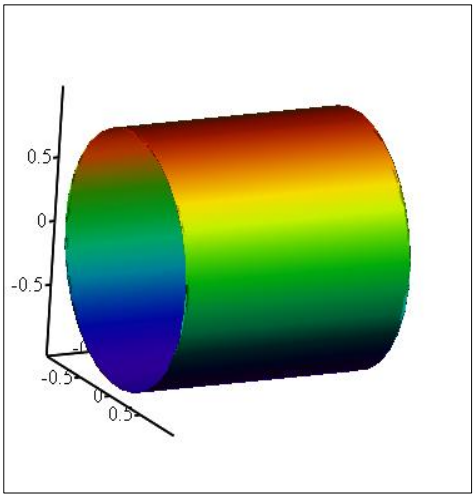
Построение на одном графике нескольких трехмерных объектов

Выполнив построение нескольких графиков в одной системе координат, можно отследить их взаимное положение.

П р и м е р 1. Рассмотрим взаимное положение двух цилиндров:

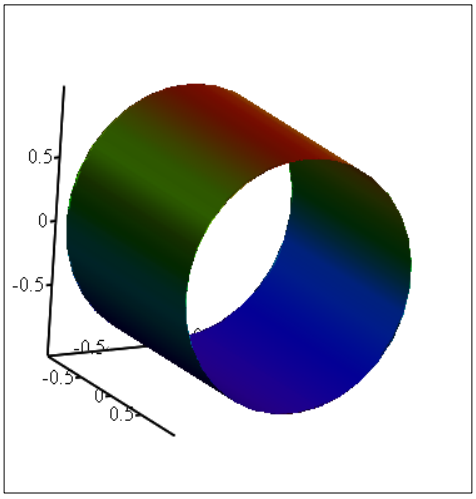
$X(u, v) := \cos(u)$ $Y(u, v) := \sin(v)$ $Z(u, v) := \sin(u)$

$N1 := CreateMesh(X, Y, Z, 50, 50)$



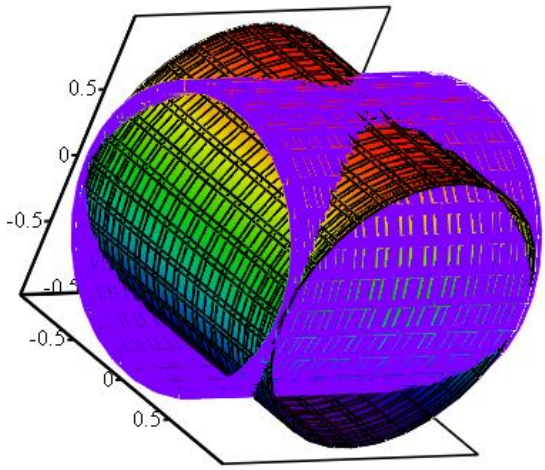
N1

$N2 := CreateMesh(Y, X, Z, 50, 50)$



N2

Их пересечение выглядит так:



N1, N2

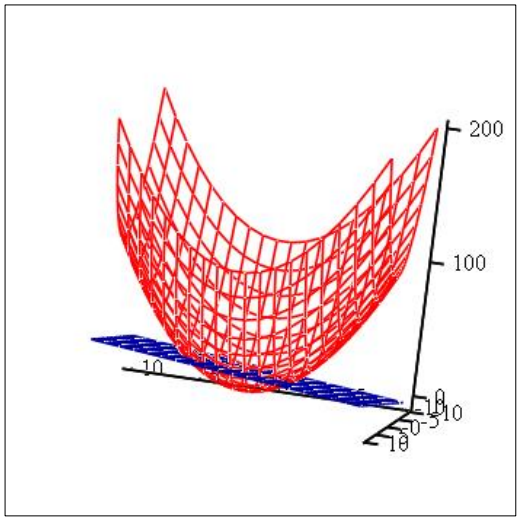
П р и м е р 2:

$F(x,y) := x^2 + y^2$

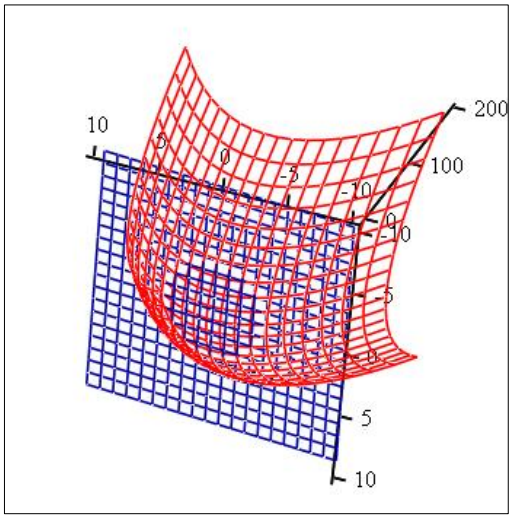
$H(x,y) := x + y + 15$

$S1 := \text{CreateMesh}(F, -10, 10, -10, 10)$

$S2 := \text{CreateMesh}(H, -10, 10, -10, 10)$



S1, S2



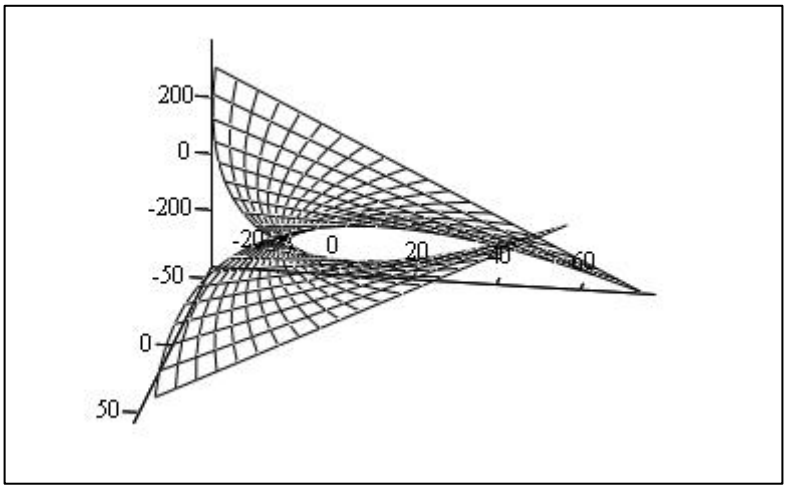
S1, S2

Можно строить график поверхности, заданной в векторной параметрической форме: $D(u,v)=\begin{pmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{pmatrix}$.

Пример 3. Построим поверхность $D(u,v)=\begin{pmatrix} cu+v \\ u^2+2uv \\ u+3u^2v \end{pmatrix}$ для разных значений параметра *c*. Для этого уравнение поверхности запишем в векторной форме. Командой Surface Plot (или клавишами «Ctrl» + «2») введем шаблон графика поверхности. В месте ввода укажем имя этого вектора. Щелкнем вне графика.

$c := 10$

$$D(u,v) := \begin{pmatrix} c\cdot u + v \\ u^2 + 2u\cdot v \\ u + 3\cdot u^2\cdot v \end{pmatrix}$$



D

Замечание. Так как появилась неопределенность в масштабировании, для получения более точного изображения поверхности потребуется ее дальнейшее форматирование.

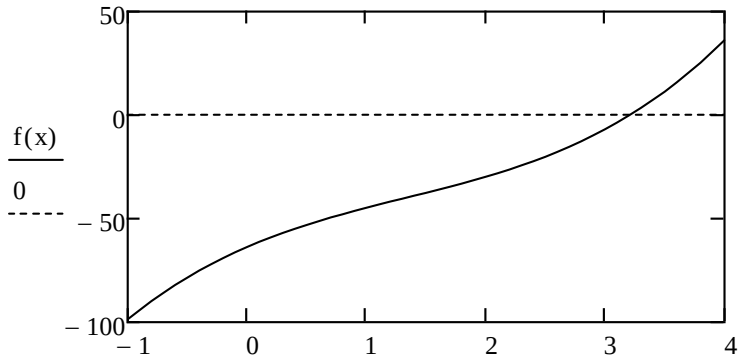
Решение нелинейных уравнений

Поиск корня нелинейного уравнения

Для простейших уравнений вида $f(x)=0$ решение находится с заданной точностью (значение системной переменной TOL) итерационным методом с помощью функции $root(f(x),x)$ и перед ее применением надо задать начальное приближение к корню.

Значит, если, например, из графика функции $y = f(x)$ стало ясно, что уравнение имеет несколько корней, то перед вызовом функции $root(f(x), x)$ надо уточнить, какой корень нужно искать. Например:

$f(x) := 2x^3 - 8x^2 + 25x - 64$ $\quad x := -1, -0.9 .. 4$



Из графика этой функции видно, что уравнение имеет один действительный корень в районе точки $x = 3$. Найдем его и два других комплексных корня.

$f(x) := 2x^3 - 8x^2 + 25x - 64$

$x := 3$	$x1 := root(f(x), x)$	$x1 = 3.211$
$x := 1 + i$	$x2 := root\left(\frac{f(x)}{x - x1}, x\right)$	$x2 = 0.395 + 3.132i$
$x := -i$	$x3 := root\left(\frac{f(x)}{x - x1}, x\right)$	$x3 = 0.395 - 3.132i$

В данной версии MathCad эта функция может записываться в расширенном формате: $root(f(x), x, a, b)$. Здесь $[a, b]$ – отрезок изоляции одного корня.

$x := root\left(2x^3 - 8x^2 + 25x - 64, x, 0, 5\right)$	$x = 3.211$
$t := root\left(e^x - x^2, x, -1, 1\right)$	$t = -0.703$

Поиск всех корней многочлена

Для поиска корней многочлена степени n можно использовать функцию *polyroots*(*V*), где V – вектор, размерности $n+1$, содержащий коэффициенты многочлена, начиная с нулевой степени. Например, чтобы найти корни многочлена из предыдущего примера, нужно поступить так:

$V_0 := -64 \quad V_1 := 25 \quad V_2 := -8 \quad V_3 := 2$

$$polyroots(V) = \begin{pmatrix} 0.395 + 3.132i \\ 0.395 - 3.132i \\ 3.211 \end{pmatrix}$$

Вычислительный блок решения системы уравнений

При решении систем нелинейных уравнений используется специально оформленный вычислительный блок, содержащий не только систему уравнений, но и условия, накладываемые на корни. Вычислительный блок должен начинаться служебным словом Given. Он имеет такую структуру:

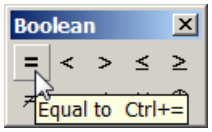
Начальные условия на корни

Given

Уравнения

Ограничительные условия

Выражения с функциями Find, Minerr, Maximize, Minimize



Начальные условия определяют начальные (примерные) значения искомых переменных. Уравнения задаются с применением жирного знака равенства между левой и правой частями. Этот знак находится на палитре Boolean или получается нажатием клавиш [Ctrl] [=].

Ограничительные условия задаются в виде равенств или неравенств, которые должны удовлетворяться при решении системы уравнений. Можно задавать начальные условия, уравнения и ограничительные условия в векторной форме, если решается система относительно нескольких неизвестных. При этом решение может искаться и в символьной форме (если оно существует).

В блоке используется функция, которая возвращает значение переменных *v*1,*v*2,…,*vn* для точного решения. Функция же Minerr(*v*1,*v*2,…,*vn*) возвращает значение переменных *v*1,*v*2,…,*vn* для приближенного решения. Если начальные условия заданы некорректно, тогда система подсвечивает переменную, в которой должно быть получено решение красным цветом.

$x := 1 \qquad y := 1$

Given

$x^2 + y^2 = 25$ Используйте [Ctrl][=], задавая нужный знак равенства

$x - y = 2$

$x > 0$

$y > 0$

$$\text{Find}(x,y) = \begin{pmatrix} 4.391 \\ 2.391 \end{pmatrix}$$

Приведенный выше код определяет точку пересечения окружности $x^2 + y^2 = 25$ и прямой $x - y = 2,$ расположенную в первой координатной четверти.

1.11. Аналитические (символьные) вычисления

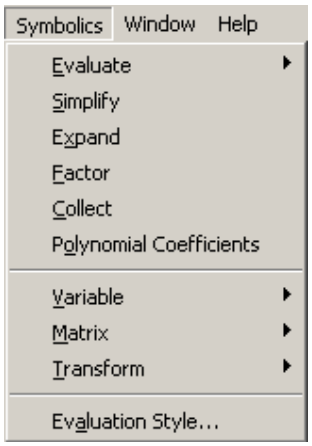
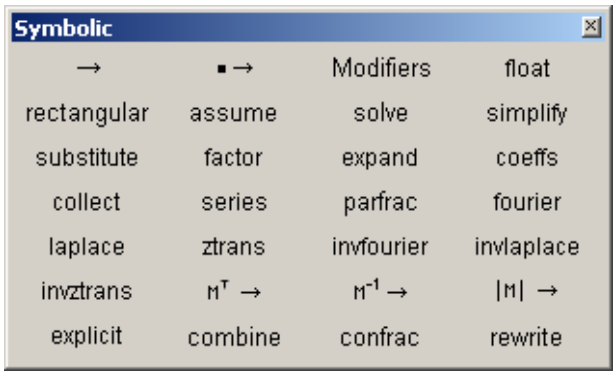
Символьными называют такие вычисления, результаты которых представляются в аналитическом виде, т.е. в виде формул (в частном случае – числом).

Вычисления в символьном виде отличаются большей общностью и позволяют судить о некоторых закономерностях решаемых задач.

Системы компьютерной математики, выполняющие символьные вычисления, принято называть *системами компьютерной алгебры*. Такие системы снабжены специальным процессором для выполнения аналитических (символьных) преобразований. Его основой является ядро, хранящее всю совокупность формул и формульных преобразований, с помощью которых производятся аналитические вычисления. Команды, относящиеся к работе символьного процессора, содержатся в меню Symbolics.

Символьные вычисления можно выполнять, используя разные возможности:

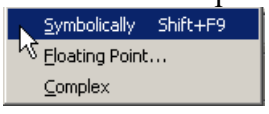
- в командном режиме – через различные команды меню Symbolics;
- при применении специальных символьных операторов палитры Symbolic; доступ к символьным операторам палитры появляется, если нажать кнопку  .

Командный режим	Палитра символьных операторов
	

Символьные вычисления в командном режиме

Символьные операции с выделенными выражениями

С выделенными выражениями допустимы следующие операции:

Операция меню Symbolics	Действия
Evaluate («Вычислить»)	Преобразовать выражение с дальнейшим выбором вида преобразования из подменю: 

Операция меню Symbolics	Действия
Simplify («Упростить»)	Упростить выделенное выражение, выполняя такие операции, как приведение подобных слагаемых, использование основных тригонометрических тождеств, приведение дробей к общему знаменателю и т.д.
Expand («Разложить по степеням»)	Получить выражение в бесскобочном варианте
Factor («Разложить на множители»)	Разложить число или выражение на множители
Collect («Разложить по подвыражениям»)	Результатом будет выражение – многочлен относительно выбранного выражения
Polynomial Coefficients («Полиномиальные коэффициенты»)	Найти по заданной переменной коэффициенты многочлена, приближающего выражение, в котором эта переменная использована

СИМВОЛЬНЫЕ ОПЕРАЦИИ С ВЫДЕЛЕННЫМИ ПЕРЕМЕННЫМИ

С выделенными переменными допустимы следующие операции:

Операция меню Symbolics ►Variable	Действия
Solve («Решить»)	Решить уравнение или неравенство относительно выделенной переменной
Substitute («Подстановка»)	Заменить указанную переменную во всем выражении содержимым буфера обмена
Differentiate («Дифференцировать»)	Дифференцировать все выражение по выделенной переменной (остальные переменные рассматриваются как константы)
Integrate («Интегрировать»)	Интегрировать все выражение по выделенной переменной
Expand to Series («Разложить в ряд»)	Найти несколько членов разложения выражения в ряд Тейлора относительно выделенной переменной
Convert to Partial Fraction («Разложить на элементарные дроби»)	Разложить на элементарные дроби выражение, которое рассматривается как рациональная дробь относительно выделенной переменной

СИМВОЛЬНЫЕ ОПЕРАЦИИ С ВЫДЕЛЕННЫМИ МАТРИЦАМИ

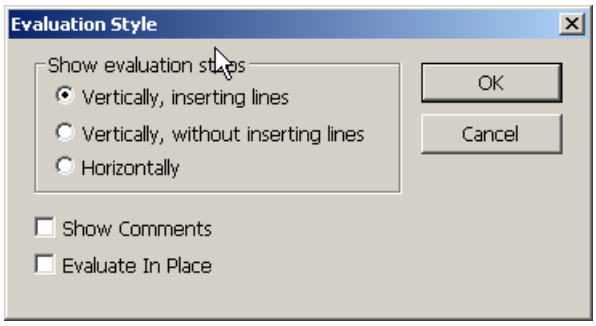
С выделенными матрицами допустимы следующие операции:

Операция меню Symbolics ►Matrix	Действия
Transpose («Транспонировать»)	Получить транспонированную матрицу
Invert («Обратить»)	Получить обратную матрицу
Determinant («Определитель»)	Вычислить определитель матрицы

Если элементы матрицы – числа, то соответствующие операции выполняются в числовой форме.

Операции вывода результатов

Чтобы выполнить символьные вычисления, необходимо указать, над каким выражением надо проводить действия, значит, предварительно надо выделить выражение. Символьные вычисления выполняются только над явными выражениями, поэтому присутствие в выражении функций недопустимо. Результат может выводиться ниже исходного выражения, справа от него или вместо него. Способ задаем командой меню Symbolics ► Evaluate Style:



Выполнение символьных вычислений в командном режиме

Выделение объектов символьных операций

Для проведения символьных операций нужно, прежде всего, выделить объект, над которым эти операции выполняются. Если объект не выделен, соответствующие команды меню Symbolics недоступны. Объектом для выполнения операции может быть самостоятельное математическое выражение, часть математического выражения или заданной пользователем функции, например, переменная, результат предшествующей операции и т. д.

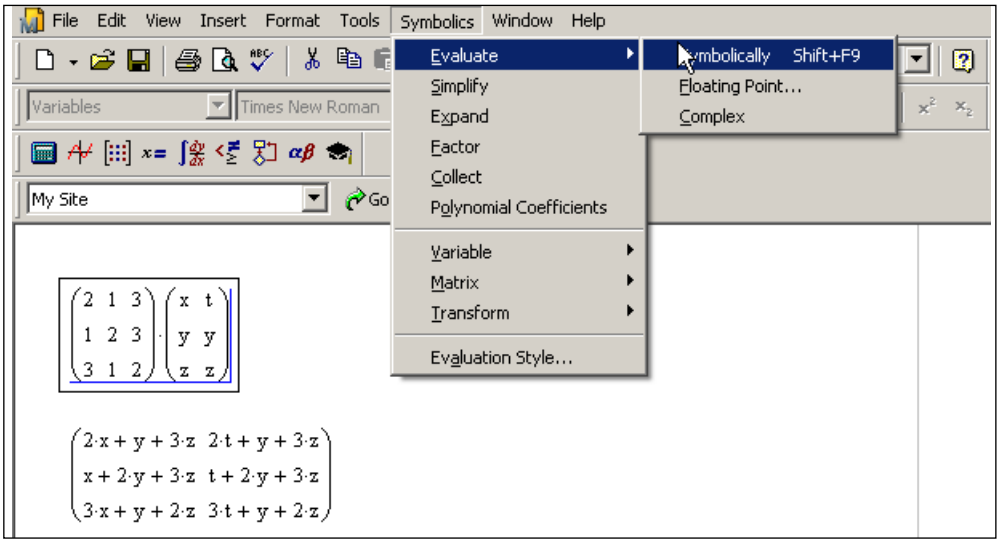
Для выделения в объекте некоторой переменной нужно установить указатель мыши после этой переменной и щелкнуть левой кнопкой. Переменная будет помечена синим уголком, расположенным следом за ней. Перемещая указатель мыши над объектом при нажатой левой кнопке, можно выделить отдельные части выражения или выражение целиком. Для выделения частей выражений полезно также использовать клавиши перемещения курсора при нажатой клавише Shift.

Часть символьных операций производится указанием на объект (на выражение или его часть). Например, упрощение выражения требует такого указания на объект. Другие операции, такие как вычисление производной или интеграла, требуют указания переменной, относительно которой производится операция дифференцирования или интегрирования.

Если заданная операция невыполнима, система выводит в дополнительном окне сообщение об ошибке или просто повторяет выделенное выражение. Это означает, что операция задана корректно, но результат не может быть получен (например, если делается попытка разложить на множители объект, уже

разложенный на множители или не допускающий такого разложения в принципе).

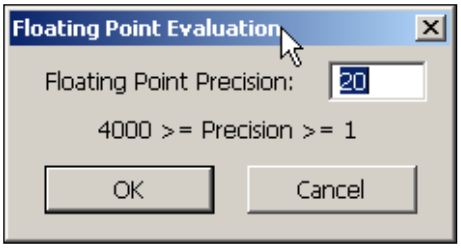
Ниже представлено описание символьной операции над матрицами.



Команда Floating Point

В MathCad имеется возможность выполнения обычных числовых вычислений с повышенной точностью – до 20 знаков после десятичной точки. Для перехода в такой режим вычислений нужно числовые константы в вычисляемых объектах задавать с обязательным указанием десятичной точки, например, 10.0 или 3.0, а не 10 или 3. Этот признак является указанием на проведение вычислений такого типа.

Команда Evaluate ► Floating Point практически лишена ограничения на количестве цифр результата. Точнее говоря, она может дать результат с 4000 верных цифр. При выборе этой команды появляется окно, в котором надо задать число цифр результата от 1 до 4000:



Однако если количество цифр результата велико, система предлагает поместить результат вычислений в буфер обмена. Проэкспериментируйте самостоятельно на примере выражения, содержащего иррациональные числа, в частности число π, или факториал большого числа. Надо отметить, что система Mathematica, к примеру, способна выдавать результаты вычислений с миллионом верных цифр.

Команда Complex

Многие вычисления имеют смысл, только если задан режим комплексных вычислений. Таковы, например, вычисления квадратного корня из выражения, дающего отрицательное значение и т. д. Режим комплексных вычислений, таким образом, существенно расширяет возможности вычислений.

Команда Evaluate ► Complex задает режим комплексных вычислений.

Пример выполнения команд Complex и Floating Point при вычислении значения *arcsin*(2) представлен на следующем рисунке. Обратите внимание, что в случае команды Evaluate ► Complex вместо числа получено выражение.

Исходное выражение	Результат выполнения команды **Symbolics ►Evaluate ►Complex**
asin(2)	$\frac{\pi}{2} - \ln\left(\sqrt{3} + 2\right)\cdot i$
Исходное выражение	**Результат выполнения команды Symbolics ►Evaluate ►FloatingPoint**
asin(2)	1.5707963267948966192 – 1.31695789692481670

Упрощение выражений

Команда Simplify (упростить) позволяет выполнить одну из самых важных символьных операций, которая позволяет упрощать математические выражения, содержащие алгебраические и тригонометрические функции, выражения с многочленами; также можно выполнять символьные вычисления производных и определенных интегралов. Упрощение означает замену сложных фрагментов выражения более простыми, однако результатом вычислений могут быть специальные математические функции

Рассмотрим еще раз порядок действий по упрощению выражений на следующих примерах:

а) $\frac{x^2 + 4\cdot x - 5}{x + 5} - 3\cdot x + 4$; б) $\sqrt{4107\cdot a^2\cdot b}$; в) $\sin^2 x + \cos^2 x + 2(\sin 0 + x)$.

Для упрощения каждого из этих выражений необходимо: ввести выражение; с помощью клавиши [Пробел] заключить упрощаемую часть в выделяющую рамку (в данном случае все выражение); воспользоваться пунктом Simplify меню Symbolics.

MathCad упростит выражение и тут же выведет результат:

Исходное выражение	Результат выполнения команды **Symbolics ►Simplify**
$\frac{x^2 + 4\cdot x - 5}{x + 5} - 3\cdot x + 4$	$3 - 2\cdot x$
$\sqrt{4107\cdot a^2\cdot b}$	$37\cdot\sqrt{3\cdot a^2\cdot b}$
$\sin(x)^2 + \cos(x)^2 + 2\cdot(\sin(0) + x)$	$2\cdot x + 1$

Упрощение дробей

С помощью символьного процессора можно производить различные операции с дробями, такие, как сокращение дробей, арифметические операции над дробями, разложение на элементарные дроби.

При помощи пункта Simplify меню Symbolics выполнить сокращение дробей:

а) $\frac{(3\cdot x+6\cdot y)^2}{5\cdot x+10\cdot y}$; б) $\frac{4\cdot x^2-y^2}{(10\cdot x+5\cdot y)^2}$.

Для решения данной задачи нужно действовать так: ввести выражение; выделить с помощью клавиши [Пробел] всю дробь выделяющей рамкой; вызвать пункт Simplify меню Symbolics.

MathCad упростит выражение и выведет результат:

Исходное выражение	Результат выполнения команды **Symbolics ►Simplify**
$\frac{(3\cdot x + 6\cdot y)^2}{5\cdot x + 10\cdot y}$	$\frac{(3\cdot x + 6\cdot y)^2}{5\cdot x + 10\cdot y}$
$\frac{4\cdot x^2 - y^2}{(10\cdot x + 5\cdot y)^2}$	$-\frac{y^2 - 4\cdot x^2}{(10\cdot x + 5\cdot y)^2}$

Вычисление производных

С помощью команды Simplify вполне возможно вычисление производных как первого, так и высшего порядка. Для вычисления производных с помощью команды Simplify запись производных надо задать в явном виде – с применением операторов вычисления производных, например:

Исходное выражение	Результат выполнения команды **Symbolics ►Simplify**
$\frac{d}{dx}\frac{(3\cdot x + 6\cdot y)^2}{5\cdot x + 10\cdot y}$	$\frac{9}{5}$
$\frac{d}{dx}\sin(x)$	$\cos(x)$

Вычисление интегралов

Система MathCad содержит встроенную функцию для вычисления значений определенных интегралов приближенным численным методом. Ею целесообразно пользоваться, когда нужно просто получить значение определенного интеграла в виде числа. Команда Simplify ищет аналитическое выражение для интеграла. Она способна делать это и при вычислении кратных интегралов, пределы которых – функции.

Вычисление интегралов с помощью команды Simplify требует записи вычисляемых интегралов в явном виде – с применением шаблонов интегралов, например:

Исходное выражение	Результат выполнения команды **Symbolics ►Simplify**
$\int \frac{(3\cdot x + 6\cdot y)^2}{5\cdot x + 10\cdot y}\,dx$	$\frac{9\cdot(x + 2\cdot y)^2}{10}$

Вычисление сумм и произведений

При вычислении суммы и произведения, так же как и при вычислении интегралов, их надо задавать в явном виде и результат операции получится в символьной форме (если таковая существует).

Исходное выражение	Результат выполнения команды **Symbolics ►Simplify**
$\sum_{n=1}^{9} \frac{1}{n^2}$	$\frac{9778141}{6350400}$

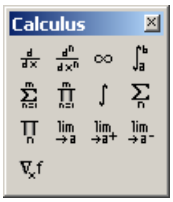
Замечания по поводу упрощения выражений

Следует помнить, что символьный процессор системы MathCad обладает заметно урезанной библиотекой функций и преобразований (в сравнении с библиотеками других систем). Поэтому часто система не находит решения в замкнутом виде, хотя оно и приводится в справочнике. Тогда повторяется

введенное выражение или выводится сообщение об ошибке. В результате преобразований могут появляться специальные функции, которые нельзя использовать при создании математических выражений. Однако эксперименты показали, что слепо доверять результатам символьных вычислений в MathCad не стоит.

Пределы

Для вычисления предела нужно получить шаблон операции из палитры Calculus, затем заполнить соответствующие поля и вызвать команду символьного вычисления предела либо через меню символьных операций, либо клавишами [Shift] [F9]. Для получения шаблона операции можно воспользоваться горячими клавишами: [Ctrl] [L] – предел, [Ctrl] [Shift] [A] – предел справа, [Ctrl] [Shift] [B] – предел слева.



Исходное выражение	Результат выполнения команды **Symbolics ►Simplify**
$\lim_{x \to -1} \frac{x^2 - 2x + 1}{x + 1}$	undefined
$\lim_{x \to 1} \frac{x^2 - 2x + 1}{x + 1}$	0
$\lim_{x \to -1^+} \frac{x + 1}{x^2 + 2x + 1}$	∞
$\lim_{x \to -1^-} \frac{x + 1}{x^2 + 2x + 1}$	$-\infty$

Разложение по степеням

Действие команды Expand (Разложить по степеням) в некотором смысле противоположно действию команды Simplify. При преобразовании выражений командой Expand система старается более простые функции представить через более сложные, свести алгебраические выражения, представленные в сжатом виде, к выражениям в развернутом виде и т. д.:

Исходное выражение	Результат выполнения команды **Symbolics ►Expand**
$(x - y)^5$	$x^5 - 5 \cdot x^4 \cdot y + 10 \cdot x^3 \cdot y^2 - 10 \cdot x^2 \cdot y^3 + 5 \cdot x \cdot y^4 - y^5$

Разложение выражений

Команда Factor (Разложить на множители) используется для разложения выражений или чисел на простые множители. В том случае, когда разложение части полинома содержит комплексно-сопряженные корни, порождающее их

выражение представляется квадратичным трехчленом. В MathCad разложение чисел на простые множители записывается как произведение множителей, причем повторяющиеся *n* раз множители записываются в степени *n*, например:

Исходное выражение	Результат выполнения команды **Symbolics ►Expand**
$x^3 - y^3$	$(x - y)\cdot\left(x^2 + x\cdot y + y^2\right)$
2048	2^{11}

Разложение по подвыражениям

Команда Collect (Разложить по подвыражениям) обеспечивает замену указанного выражения другим – скомплектованным по указанной переменной, если такое представление возможно. Эта команда особенно удобна, когда заданное выражение есть функция ряда переменных и нужно представить его в виде функции заданной переменной, имеющей вид степенного многочлена. При этом другие переменные входят в сомножители указанной переменной, представленной в порядке уменьшения ее степени.

Исходное выражение	Результат выполнения команды **Symbolics ►Collect**
$(x + b + c)^2$	$x^2 + (2\cdot b + 2\cdot c)\cdot x + (b + c)^2$

Вычисление коэффициентов полиномов

Команда Polynomial Coefficients (коэффициенты полинома), служит для вычисления коэффициентов полинома. Команда применяется, если заданное выражение – полином или может быть представлено таковым относительно выделенной переменной. Результатом операции является вектор с коэффициентами полинома.

Исходное выражение	Результат выполнения команды **Symbolics ►Polynomial Coefficientst**
$(x + b + c)^2$	$\begin{pmatrix} b^2 + 2\cdot b\cdot c + c^2 \\ 2\cdot b + 2\cdot c \\ 1 \end{pmatrix}$

Операции относительно заданной переменной

Дифференцирование

Команда Variable ► Differentiate (Переменная ► Дифференцировать) дифференцирует выражение по той переменной, которая указана курсором:

Исходное выражение	Результат выполнения команды **Symbolics ►Variable ►Differentiate**
$(x + b + c)^2$	$2\cdot b + 2\cdot c + 2\cdot x$

Интегрирование

Команда Variable ► Integrate (Переменная ► Интегрировать) возвращает символьное значение неопределенного интеграла по переменной, которая указана курсором:

Исходное выражение	Результат выполнения команды **Symbolics ►Variable ►Integrate**
$(x + b + c)^2$	$\frac{(b + c + x)^3}{3}$

Решение уравнений

Если задано некоторое выражение и выделена в нем переменная, то команда Variable ► Solve (Переменная ► Решить) возвращает символьное значение указанной переменной, при котором это выражение равно нулю.

Исходное выражение	Результат выполнения команды **Symbolics ►Variable ►Solve**
$a\cdot x^2 + b\cdot x + c$	$\begin{pmatrix} -\frac{\frac{b}{2} - \frac{\sqrt{b^2 - 4\cdot a\cdot c}}{2}}{a} \\ -\frac{\frac{b}{2} + \frac{\sqrt{b^2 - 4\cdot a\cdot c}}{2}}{a} \end{pmatrix}$

Подстановка

Команда Variable ► Substitute (Переменная ► Подстановка) возвращает новое выражение, полученное путем подстановки вместо указанной переменной другого выражения, которое должно быть подготовлено в буфере обмена, например, командой [Копировать] меню [Правка].

Скопируем в буфер обмена выражение $z - y$

$z - y$

Выделим в выражении $\frac{(3\cdot x + 6\cdot y)^2}{5\cdot x + 8\cdot y}$ переменную x

и выполним команду Symbolics ► Variable ► Substitute

Исходное выражение	Результат выполнения команды **Symbolics ►Variable ► Substitute**
$\frac{(3\cdot x + 6\cdot y)^2}{5\cdot x + 8\cdot y}$	$\frac{9\cdot(y + z)^2}{3\cdot y + 5\cdot z}$

Разложение в ряд Тейлора

Команда Variable ► Expand to Series (Переменная ► Разложить в ряд) выполняет разложение выражения в ряд Тейлора относительно выделенной переменной с заданным по дополнительному запросу числом членов ряда:

Исходное выражение	Результат выполнения команды **Symbolics ►Variable ► Expand to Series**
$\cos(x)$	$1 - \frac{x^2}{2} + \frac{x^4}{24}$

Разложение на правильные дроби

Команда Variable ► Convert to Partial Fraction (Переменная ► Разложить на элементарные дроби) возвращает разложение выражения в виде суммы правильных элементарных дробей относительно выделенной переменной, например:

Исходное выражение	Результат выполнения команды **Symbolics ►Variable ►Convert to Partial Fraction**
$\frac{2}{x^2 - 1}$	$\frac{1}{x - 1} - \frac{1}{x + 1}$

Матричные операции

С *выделенными* матрицами допустимы следующие операции:

- Получить транспонированную матрицу – Transpose, например:

Исходное выражение	Результат выполнения команды **Symbolics ► Matrix ► Transpose**
$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & j \end{pmatrix}$	$\begin{pmatrix} a & d & g \\ b & e & h \\ c & f & j \end{pmatrix}$

- Получить обратную матрицу – Invert, например:

Исходное выражение	Результат выполнения команды **Symbolics ► Matrix ► Invert**
$\begin{pmatrix} x & y \\ z & t \end{pmatrix}$	$\begin{pmatrix} \frac{t}{t\cdot x - y\cdot z} & -\frac{y}{t\cdot x - y\cdot z} \\ -\frac{z}{t\cdot x - y\cdot z} & \frac{x}{t\cdot x - y\cdot z} \end{pmatrix}$

- Вычислить определитель матрицы – Determinant, например:

Исходное выражение	Результат выполнения команды **Symbolics ► Matrix ► Determinant**
$\begin{pmatrix} x & y \\ z & t \end{pmatrix}$	$t\cdot x - y\cdot z$

Надо заметить, что для матриц большого размера выполнить такие операции не всегда возможно.

Символьные вычисления в явном режиме

Символьные вычисления в явном виде предпочтительнее, так как после обновления рабочего листа такие операции пересчитываются. Вычисления в символьном виде возможны с использованием клавиш «Shift» + «F9» или одиночной специальной операции «→» палитры Symbolic как в таком одиночном виде, так и в сочетании с предварительно заданным перед операцией «→» действием. После операции «→» надо нажать клавишу «Ввод», чтобы увидеть результат.

Символьные операции с выделенными выражениями

С выделенными выражениями допустимы следующие операции (таблица 23):

Таблица 23 – Символьные операции с выделенными выражениями.

Операция палитры Symbolic	Действия
→	Преобразовать выражение. Эквивалентно Evaluate ► Symbolically. Горячие клавиши «Ctrl» + «.»
• →	Выполнить указанное перед операцией «→» действие •
Float, n →	Вычислить, преобразовывая результат в десятичную дробь с *n* знаками после запятой
Explicit →	Вычислить (возможно, с подстановкой значений, входящих в выражение переменных), не преобразовывая результат в десятичную дробь
Simplify →	Упростить выделенное выражение, выполняя такие операции, как приведение подобных слагаемых, использование основных тригонометрических тождеств, приведение дробей к общему знаменателю и т.д.
Expand →	Получить выражение в бесскобочном варианте
Factor →	Разложить число или выражение на множители
Collect	Результат – многочлен относительно выбранного выражения
Coeffs	Найти по заданной переменной коэффициенты многочлена, приближающего выражение, в котором эта переменная использована
Confrac	Разложить выражение в цепную дробь
Combine	Упростить выражение, используя стандартные формулы
Rewrite	Преобразовать выражение, используя элементарные функции

Символьные операции с выделенными переменными

С выделенными переменными допустимы следующие операции (таблица 24):

Таблица 24 – Символьные операции с выделенными переменными.

Операция палитры Symbolic	Действия
Solve, • →	Решить уравнение или неравенство относительно указанной вместо символа «•» переменной
Substitute	Заменить указанную переменную во всем выражении содержимым буфера обмена
Parfrac	Разложить на элементарные дроби выражение, которое рассматривается как рациональная дробь относительно выделенной переменной
Assume, x=type	Присвоить переменной тип

Многие из перечисленных выше операций применяются с определенными уточнениями действий – директивами. Например, при приведении к некоторому новому типу переменной необходимо указать явно этот тип. Некоторые их этих ключевых слов собраны в операции Modifiers палитры Symbolic: integer, real,

RealRange, complex, fully. Эти, а также другие модификаторы команд перечислены ниже (таблица 25):

Таблица 25– Модификаторы команд.

Модификаторы команд	Действия	Где используются
All	Применить для всех переменных в выражении	assume, explicit
atan	Упростить выражение, используя стандартные формулы для функции арктангенса	combine
complex	Указание на необходимость выполнения операций в комплексной форме	assume, factor, parfrac
degree	Возвратить второй столбец, в котором содержатся степени одночленов	coeffs
domain	Задать область определения (домен) переменной	assume, factor, parfrac
exp	Упростить или преобразовать выражение, используя стандартные формулы для экспоненциальной функции	combine, rewrite
fully	Возвратить детальное решение уравнения	В любом операторе
integer	Для V=integer означает целочисленное значение переменной *V*	assume
ln	Упростить или преобразовать выражение, используя стандартные формулы для логарифмической функции	combine, rewrite
log	Упростить или преобразовать выражение, используя стандартные формулы для логарифмической функции	combine, rewrite
raw	Возвратить результат без проведения упрощений	assume, factor, parfrac
real	Для V=real означает вещественное значение переменной *V*	assume, factor
RealRange	Для V=RealRange(a, b) означает принадлежность вещественной переменной *V* к интервалу (*a*, *b*)	assume
sincos	Упростить или преобразовать выражение, используя стандартные формулы для синуса и косинуса	combine, rewrite
sinhcosh	Упростить или преобразовать выражение, используя стандартные формулы для гиперболического синуса и косинуса	combine, rewrite

`using`	Заменить переменную-параметр в полученном решении уравнения	В любом операторе

Символьные операции с выделенными матрицами

С выделенными матрицами допустимы следующие операции (таблица 26):

Таблица 26 – Символьные операции с выделенными матрицами.

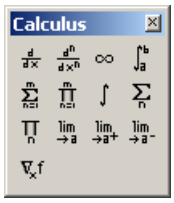
Операция палитры Symbolic	Действия
$M^T \rightarrow$ (транспонировать)	Получить транспонированную матрицу
$M^{-1} \rightarrow$ (обратить)	Получить обратную матрицу
$\lvert M \rvert \rightarrow$ (определитель)	Вычислить определитель матрицы

Выполнение символьных вычислений в явном виде

Большинство рассмотренных выше операций, выполненных в командном режиме, можно использовать для вычисления в явном виде.

Пределы

Для вычисления предела нужно получить шаблон операции из палитры Calculus.



Затем следует заполнить соответствующие поля и вызвать команду символьного вычисления предела либо через меню символьных операций, либо клавишами [Shift] [F9]. Напомним, что для получения шаблона операции можно воспользоваться горячими клавишами: [Ctrl] [L] – предел, [Ctrl] [Shift] [A] – предел справа, [Ctrl] [Shift] [B] – предел слева.

Примеры вычисления пределов:

$$\lim_{x \to -1} \frac{x^2 - 2x + 1}{x + 1} \to \text{undefined}$$

$$\lim_{x \to 1} \frac{x^2 - 2x + 1}{x + 1} \to 0$$

$$\lim_{x \to -1^+} \frac{x + 1}{x^2 + 2x + 1} \to \infty$$

$$\lim_{x \to -1^-} \frac{x + 1}{x^2 + 2x + 1} \to -\infty$$

Действия символьной математики

Рассмотрим далее простейшие примеры символьных вычислений в явном виде, не требующих комментариев:

$$\frac{x^2 + 4 \cdot x - 5}{x + 5} - 3 \cdot x + 4 \to \frac{x^2 + 4 \cdot x - 5}{x + 5} - 3 \cdot x + 4$$

$$\sqrt{4107 \cdot a^2 \cdot b} \to 37 \cdot \sqrt{3} \cdot \sqrt{a^2 \cdot b}$$

$$\frac{d}{dx} \cos(x) \to -\sin(x)$$

$$\frac{d^3}{dx^3} \cos(x) \to \sin(x)$$

$$\sin(x)^2 + \cos(x)^2 + 2 \cdot (\sin(0) + x) \to \cos(x)^2 + \sin(x)^2 + 2 \cdot x$$

Необходимо отметить, что результат вычислений в последнем примере не совсем приемлем, так как при упрощении не было применено основное тригонометрическое тождество. Чтобы получить более завершенный результат, оператор упрощения запишем в более полном виде, используя операцию combine:

$$\sin(x)^2 + \cos(x)^2 + 2\cdot(\sin(0) + x) \rightarrow \cos(x)^2 + \sin(x)^2 + 2\cdot x$$

$$\sin(x)^2 + \cos(x)^2 + 2\cdot(\sin(0) + x)\ \text{combine, sincos} \rightarrow 2\cdot x + 1$$

$$a^n\cdot a^m \rightarrow a^m\cdot a^n$$

$$a^n\cdot a^m\ \text{combine} \rightarrow a^{m+n}$$

$$e^b\cdot e^{2t} \rightarrow e^b\cdot e^{2\cdot t}$$

$$e^b\cdot e^{2t}\ \text{combine} \rightarrow e^b\cdot e^{2\cdot t}$$

$$e^b\cdot e^{2t}\ \text{combine, exp} \rightarrow e^{b+2\cdot t}$$

$$\ln(x) + \ln(2) + 3\cdot\ln\left(\frac{3}{2}\right)\ \text{combine, ln} \rightarrow \ln\left(\frac{27\cdot x}{4}\right)$$

$$2\log(5, x) + 5\cdot\log(5, x)\ \text{combine, log} \rightarrow \frac{\log(78125)}{\log(x)}$$

Применение операции Rewrite («Преобразовать»):

$$\sin(x)\ \text{rewrite, exp} \rightarrow -\frac{e^{x\cdot i}\cdot i}{2} + \frac{1}{2}\cdot e^{x\cdot(-i)}\cdot i$$

$$-\frac{e^{x\cdot i}\cdot i}{2} + \frac{1}{2}\cdot e^{x\cdot(-i)}\cdot i\ \text{rewrite, sincos} \rightarrow \sin(x)$$

Упрощение выражений:

$$(x - 1)^2 + 2x^2 + 4x - 1\ \text{simplify} \rightarrow x\cdot(3\cdot x + 2$$

Если нужно знать числовое значение функции $F(x) := \sum_{k=0}^{5}\left[\frac{5!}{k!(5-k)!}x^k 2^{5-k}\right]$. при некотором конкретном *x*, то используется символ «=» или «→». Затем нажимается клавиша «Ввод» и получается результат:

$$n := 5$$

$$F(x) := \sum_{k=0}^{n}\left[\frac{n!}{k!\,(n-k)!}x^k 2^{n-k}\right]$$

$$F(2) = 1.024\times 10^3 \qquad F(-5) = -243$$

$$F(x) \rightarrow x^5 + 10\cdot x^4 + 40\cdot x^3 + 80\cdot x^2 + 80\cdot x + 32$$

Разложение на множители выполним следующим образом: щелкнем на команде factor меню Symbolic, появится шаблон команды. Заполним поле, нажмем [Ввод] и, как видно из примеров ниже, результат над полем действительных чисел получается не всегда:

$$2x^3 - 2x - 3x^2 + 3 \text{ factor } \rightarrow (2\cdot x - 3)\cdot(x - 1)\cdot(x + 1)$$

$$x^4 - 4x^3 - 1 \text{ factor } \rightarrow x^4 - 4\cdot x^3 - 1$$

Разложение дробей в сумму элементарных дробей и разложение на множители в более полном варианте продемонстрировано ниже:

$$\frac{1}{x^2 + 2} \text{ parfrac}, x, \text{domain} = \text{complex} \rightarrow \frac{\sqrt{2}\cdot i}{4\cdot\left(x + \sqrt{2}\cdot i\right)} - \left[\frac{\sqrt{2}}{4\cdot\left(x - \sqrt{2}\cdot i\right)}\right].$$

$$x^4 - 4\cdot x^3 - 1 \text{ factor}, \text{domain} = \text{complex}, \text{float}, 3 \rightarrow$$

$$(x + 0.601)\cdot[x - (0.293 - 0.573i)]\cdot[x - (0.293 + 0.573i)]\cdot(x - 4.02)$$

Коэффициентами исходной дроби должны быть либо рациональные дроби, либо целые числа, иначе символьный процессор не сможет с ней работать, а подключится числовой процессор:

Функция IsPrime

Функция IsPrime(n) возвращает значение 1, если аргумент есть число простое, и значение 0, если составное, например:

$$\text{IsPrime}(2049) \rightarrow 0 \qquad 2049 \text{ factor } \rightarrow 3\cdot 683$$

$$\text{IsPrime}(2053) \rightarrow 1 \qquad 2053 \text{ factor } \rightarrow 2053$$

Функции Numer и Denom

Значения, возвращаемые функциями, это соответственно числитель и знаменатель некоторой дроби, например:

$$\text{numer}\left(\frac{x^2 + y^2}{a^6 + b^6}\right) \rightarrow x^2 + y^2 \qquad \text{denom}\left(\frac{x^2 + y^2}{a^6 + b^6}\right) \rightarrow a^6 + b^6$$

Модифицированные команды (Modifier)

Рассмотрим еще примеры использования модифицированных команд для переменных и решений ALL, domain и для коэффициентов degree:

$$\sqrt{a^2} + \sqrt{b^2} + \sqrt{c^2}\ \text{assume, ALL} < 0 \rightarrow -a - b - c$$

$$x := 5 \qquad y := 3 \qquad z := -1$$

$$\frac{x \cdot y - y \cdot z}{2 \cdot z - x}\ \text{explicit, x, y} \rightarrow \frac{(5 \cdot 3 - 3 \cdot z)}{(2 \cdot z - 5)}$$

$$\frac{x \cdot y - y \cdot z}{2 \cdot z - x}\ \text{explicit, ALL} \rightarrow \frac{(5 \cdot 3 - 3 \cdot -1)}{(2 \cdot -1 - 5)}$$

$$4 - 5 \cdot t^3 + 6 \cdot t\ \text{coeffs, degree} \rightarrow \begin{pmatrix} 4 & 0 \\ 6 & 1 \\ 0 & 2 \\ -5 & 3 \end{pmatrix}$$

Команда domain также используется совместно с командами factor или parfrac. Рассмотрим следующие примеры:

$$\frac{1}{x^2 + 9.}\ \text{parfrac, x, domain = complex} \rightarrow \frac{i}{6 \cdot (x + 3i)} - \left[\frac{1}{6 \cdot (x - 3i)}\right] \cdot i$$

$$\frac{1}{x^2 + 8.1}\ \text{parfrac, x, domain = complex} \rightarrow \frac{1.0}{x^2 + 8.1}$$

$$x^3 - 27 \left| \begin{array}{l} \text{factor, domain = real} \\ \text{float, 2} \end{array} \right. \rightarrow (x - 3.0) \cdot \left(3.0 \cdot x + x^2 + 9.0\right)$$

$$x^3 - 27 \left| \begin{array}{l} \text{factor, domain = complex} \\ \text{float, 2} \end{array} \right. \rightarrow (x - 3.0) \cdot (x + 1.5 - 2.6i) \cdot (x + 1.5 + 2.6i)$$

Решение уравнений и систем

Рассмотрим далее разные приемы решения уравнений и систем с помощью символьной математики.

Решение некоторых уравнений можно проводить, используя команду solve:

$$x^2 - 2 \cdot x + 3\ \text{solve} \rightarrow \begin{pmatrix} 1 - \sqrt{2} \cdot i \\ 1 + \sqrt{2} \cdot i \end{pmatrix}$$

$$x^2 - 2 \cdot x + 3.0\ \text{solve} \rightarrow \begin{pmatrix} 1.0 - 1.4142135623730950488i \\ 1.0 + 1.4142135623730950488i \end{pmatrix}$$

При решении второго уравнения, очевидно, подключился числовой процессор, так как коэффициенты уравнения – вещественные числа.

Некоторые уравнения MathCad решает символьно в радикалах, если находит такие представления результата, а большинство в виде десятичных чисел, например:

$$x^5 - x^2 + 1 \text{ solve} \rightarrow \begin{pmatrix} 0.8692775018425938612 - 0.3882694065997403553\epsilon \\ 0.8692775018425938612 + 0.3882694065997403553\epsilon \\ -0.46491220160289785433 - 1.071473840270269409 \\ -0.46491220160289785433 + 1.071473840270269409 \\ -0.80873060047939201374 \end{pmatrix}$$

$$x^5 - x^2 + 1 \left| \begin{matrix} \text{solve} \\ \text{float},9 \end{matrix} \right. \rightarrow \begin{pmatrix} 0.869277502 - 0.388269406i \\ 0.869277502 + 0.388269406i \\ -0.464912202 - 1.07147384i \\ -0.464912202 + 1.07147384i \\ -0.8087306 \end{pmatrix}$$

$$x^5 - x^4 - 2\cdot x^3 + 2\cdot x^2 - 2\cdot x - 4 \text{ solve} \rightarrow \begin{bmatrix} -1 \\ 2 \\ -\frac{(-2)^{\frac{1}{3}}\cdot\left(1 + \sqrt{3}\cdot i\right)}{2} \\ \frac{(-2)^{\frac{1}{3}}\cdot\left(-1 + \sqrt{3}\cdot i\right)}{2} \\ (-2)^{\frac{1}{3}} \end{bmatrix}$$

Рассмотрим еще, как меняется результат решения одного и того же уравнения, в зависимости от уточнений на корни на следующих примерах.

$$\left(x^3-1\right)\cdot\left(x^2-2\right)\ \text{solve}\ \rightarrow\begin{bmatrix}1\\-\sqrt{2}\\-\frac{1}{2}-\left(\frac{\sqrt{3}}{2}\right)\cdot i\\-\frac{1}{2}+\frac{1}{2}\cdot\sqrt{3}\cdot i\\\sqrt{2}\end{bmatrix}$$

$$\left(x^3-1\right)\cdot\left(x^2-2\right)\ \left|\begin{array}{l}\text{solve}\\\text{assume},x=\text{real}\end{array}\right.\rightarrow\begin{pmatrix}1\\-\sqrt{2}\\\sqrt{2}\end{pmatrix}$$

$$\left(x^3-1\right)\cdot\left(x^2-2\right)\ \left|\begin{array}{l}\text{solve}\\\text{assume},x=\text{RealRange}(0,2)\end{array}\right.\rightarrow\begin{pmatrix}1\\\sqrt{2}\end{pmatrix}$$

$$\left(x^3-1\right)\cdot\left(x^2-2\right)\ \left|\begin{array}{l}\text{solve}\\\text{assume},x=\text{integer}\end{array}\right.\rightarrow 1$$

Решение уравнений с параметром

Решение уравнения с параметром в неполном варианте выполним следующим образом: щелкнем на команде solve меню Symbolic, появится шаблон команды. Заполним поле, нажмем [Ввод] и получим результат:

$$(x-1)^2+x^2+4x-z=0\ \text{solve},x\ \rightarrow\begin{pmatrix}\frac{\sqrt{2\cdot z-1}}{2}-\frac{1}{2}\\-\frac{\sqrt{2\cdot z-1}}{2}-\frac{1}{2}\end{pmatrix}$$

$$(x-1)^2+x^2+4.0x-z=0\ \text{solve},x\ \rightarrow\begin{bmatrix}0.5\cdot(2.0\cdot z-1.0)^{0.5}-0.5\\-0.5\cdot(2.0\cdot z-1.0)^{0.5}-0.5\end{bmatrix}$$

Решение уравнения с параметром по полному варианту выполним так: щелкнем на команде solve меню Symbolic, появится шаблон команды. Заполним поле, нажмем [Ввод] и получим результат:

$$z\,x - z = 0\ \text{solve}, x \rightarrow 1$$

$$z\,x - z = 0\ \text{solve}, x, \text{fully} \rightarrow \begin{cases} 1 & \text{if } z \neq 0 \\ \_c1 & \text{if } \_c1 \in \mathbb{C} \wedge z = 0 \end{cases}$$

$$\sin(x) = \cos(x)\ \text{solve} \rightarrow \frac{\pi}{4}$$

$$\sin(x) = \cos(x)\ \text{solve}, \text{fully} \rightarrow \begin{cases} \frac{\pi}{4} + \pi \cdot \_n & \text{if } \_n \in \mathbb{Z} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\sin(x) = \cos(x)\ \text{solve}, \text{fully}, \text{using}, \_n = m, \text{fully} \rightarrow \begin{cases} \frac{\pi}{4} + \pi \cdot m & \text{if } m \in \mathbb{Z} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Результат символьного решения можно оформить функцией:

$$f(a) := (x + 2)\cdot(a - 1) - 1 = a^2\ \text{solve}, x, \text{fully} \rightarrow \begin{cases} \frac{a^2 - 2\cdot a + 3}{a - 1} & \text{if } a \neq 1 \\ \text{undefined} & \text{if } a = 1 \end{cases}$$

$$f\left(\frac{3}{4}\right) \rightarrow -\frac{33}{4} \qquad f(1) \rightarrow \text{undefined} \qquad f(1) = \text{NaN}$$

Здесь при попытке вычислить символьно значение функции $f(1)$ мы получили сообщение «неопределено», а в случае численного вычисления $f(1)$ MathCad ответил «NaN», что означает - «Not a Number» (не число). Отметим, что если оформить функцию следующим вариантом, то мы вообще получим ошибку:

$$g(a) := (x + 2)\cdot(a - 1) - 1 = a^2\ \text{solve}, x \rightarrow \frac{a^2 - 2\cdot a + 3}{a - 1}$$

$$g\left(\frac{3}{4}\right) \rightarrow -\frac{33}{4} \qquad g(1) \rightarrow \qquad g(1) = \blacksquare$$

Divide by zero.

Решение систем уравнений

При символьном решении систем уравнений можно воспользоваться блоком Given – Find, например:

Given

$x^2 + y^2 = r^2$ Используйте [Ctrl][=], задавая нужный знак равенства

$x - y = 2$

$$M(r) := \text{Find}(x,y) \rightarrow \begin{pmatrix} 1 - \frac{\sqrt{2}\cdot\sqrt{r^2-2}}{2} & \frac{\sqrt{2}\cdot\sqrt{r^2-2}}{2} + 1 \\ -\frac{\sqrt{2}\cdot\sqrt{r^2-2}}{2} - 1 & \frac{\sqrt{2}\cdot\sqrt{r^2-2}}{2} - 1 \end{pmatrix}$$

Также можно задать систему уравнений как вектор (знак равенства задается жирным знаком как **[**Ctrl] **[=]**) и решить следующим образом:

$$\begin{pmatrix} 0.3\cdot w + 0.2\cdot x + 6.6\cdot y = 1 \\ 4.5\cdot w - 1.8\cdot x - 0.3\cdot y = 1 \\ -7\cdot w + 9.7\cdot x + 10.9\cdot y = 2 \end{pmatrix} \left| \begin{matrix} \text{solve}, x, y, w \\ \text{float}, 2 \end{matrix} \right. \rightarrow (0.33 \quad 0.13 \quad 0.36$$

Решение неравенств

Неравенства можно решать, используя команду solve:

$x^2 - 2\cdot x - 3 > 0$ solve $\rightarrow x < -1 \vee 3 < x$

$x^2 - 2\cdot x - 3.0 < 0$ solve $\rightarrow -1 < x < 3$

$x^3 - 22x < 24 - 3x^2$ solve $\rightarrow x < -6 \vee -1 < x < 4$

$x^2\cdot 3^x \le 3^{x-1}$ solve $\rightarrow -\frac{\sqrt{3}}{3} \le x \le \frac{\sqrt{3}}{3}$

$(x-2)\cdot(x+4)\cdot(x-3)^2\cdot(x+1)^5 > 0$ solve $\rightarrow 3 < x \vee 2 < x < 3 \vee -4 < x < -1$

Пакет MathCad не имеет собственных средств для решения систем неравенств. Однако можно воспользоваться тем, что он способен решать отдельные неравенства в символьном виде. Рассмотрим, например, как можно решить следующую систему неравенств: $\begin{cases} x^2 + 2x > 35 \\ x^2 + 42 \le 17x \end{cases}$.

Решаем по отдельности каждое неравенство:

$$x^2 + 2x > 35 \text{ solve} \rightarrow 5 < x \vee x < -7$$

$$x^2 + 42 \leq 17x \text{ solve} \rightarrow 3 \leq x \leq 14$$

Из полученных ответов можно определить, что решением системы будет интервал $x \in (5,14]$.

1.12. Программирование в системе MathCad

Составляющие программирования

Процесс написания программ на любом алгоритмическом языке условно можно определить как соединение трех начал: ремесла, науки и искусства. Кодирование заданного алгоритма – ремесло, разработка эффективных приемов и технологий – наука, поиск уникальных по эффективности и возможностям алгоритмов – искусство.

Нельзя овладеть программированием, прочитав сколь угодно много руководств и прослушав курсы лекций. Только продолжительная работа с компьютером, многочисленные эксперименты позволят овладеть ремеслом, если же эти навыки подкрепить теоретически – то и наукой, а наличие интереса и изобретательского подхода позволит дорасти до искусства.

Традиционные средства программирования

В MathCad имеется возможность задания завершенных программных модулей (программ), имеющих вид набора инструкций, выделенных в тексте жирной вертикальной чертой. Модули органично входят в состав документови дают возможность пользоваться всеми средствами не только математически ориентированного входного языка MathCad, но и классического программирования.

Существуют сложные математические задачи, которые можно решить в системе MathCad только с использованием в явном виде программных средств, особенно это удобно, когда имеется описание их программной реализации на каком-либо языке программирования.

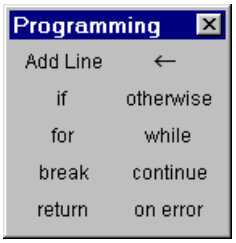
Много интересных и поучительных примеров применения программных модулей можно найти в QuickSheet («быстрые шпаргалках») центра ресурсов системы.

Модуль может вести себя как *безымянная функция* без параметров, как *именованная функция* с параметрами или без них, но все модули возвращают результат и не только скалярные величины, но и векторы или матрицы. Используемые в блоке переменные документа ведут себя как глобальные переменные, если они не переопределены в модуле.

Программы содержат конструкции, подобные программным конструкциям языков программирования:

- оператор присваивания;
- условные передачи управления;
- операторы циклов;
- области видимости переменных;
- использование подпрограмм и рекурсии.

Средства программирования сосредоточены в палитре программных элементов  , содержащей следующие инструкции:



Add Line – выполняет функции создания или расширения программного блока;

← – инструкция локального присваивания (в теле модуля);

if – условная инструкция;

for – инструкция задания цикла с фиксированным числом повторений;

otherwise – инструкция иного выбора (применяется с **if**);

while – инструкция задания цикла с неизвестным числом повторений;

break – инструкция прерывания;

continue – инструкция продолжения;

return – инструкция возврата значения;

on error – инструкция обработки ошибки.

После вызова многих инструкций появляется их шаблон, в котором необходимо заполнить поля по смыслу задачи.

У п р а ж н е н и е. Существуют «горячие клавиши» для быстрого вызова инструкций. Самостоятельно найдите их сочетания и изучите.

Поясним применение перечисленных инструкций.

Оператор внутреннего присваивания ←

Оператор ← выполняет функции локального внутреннего присваивания.

П р и м е р 1. Программа – составное выражение: $f(x,w) = \log\left(\frac{x}{w}\right)$.

Набираем $f\left(x, w\right)$. Открываем панель программированния. Нажимаем **Add Line** столько раз, столько полей понадобится.

Потом мышкой или клавишей Tab возвращаемся в 1-е поле. Набираем $z \leftarrow x / w$. Переходим ниже и набираем $\log(z)$. Переменная z – локальная, вне подпрограммы она не определена.

$$f(x,w) := \left| \begin{array}{l} z \leftarrow \dfrac{x}{w} \\ \log(z) \end{array} \right.$$

П р и м е р 2**.** Рассмотрим линейный алгоритм в задаче нахождения корней квадратного уравнения $a \cdot x^2 + b \cdot x + c = 0$:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}.$$

$$Quadr(a,b,c) := \left| \begin{array}{l} d \leftarrow b^2 - 4 \cdot a \cdot c \\ d \leftarrow \sqrt{d} \\ x1 \leftarrow \dfrac{(-b - d)}{2 \cdot a} \\ x2 \leftarrow \dfrac{(-b + d)}{2 \cdot a} \\ \begin{pmatrix} x1 \\ x2 \end{pmatrix} \end{array} \right.$$

Вызов:

$$Quadr(1,10,25) = \begin{pmatrix} -5 \\ -5 \end{pmatrix}$$

$$Quadr(1,1,1) = \begin{pmatrix} -0.5 - 0.866i \\ -0.5 + 0.866i \end{pmatrix}$$

Результат возвращается как вектор из двух компонент.

Инструкция on error и функция error

Инструкция **on error** позволяет создавать процедуры обработки ошибок. Формат инструкции:

Выражение2 **on error** Выражение1

Если при выполнении *Выражения1* возникает ошибка, то выполняется *Выражение2*.

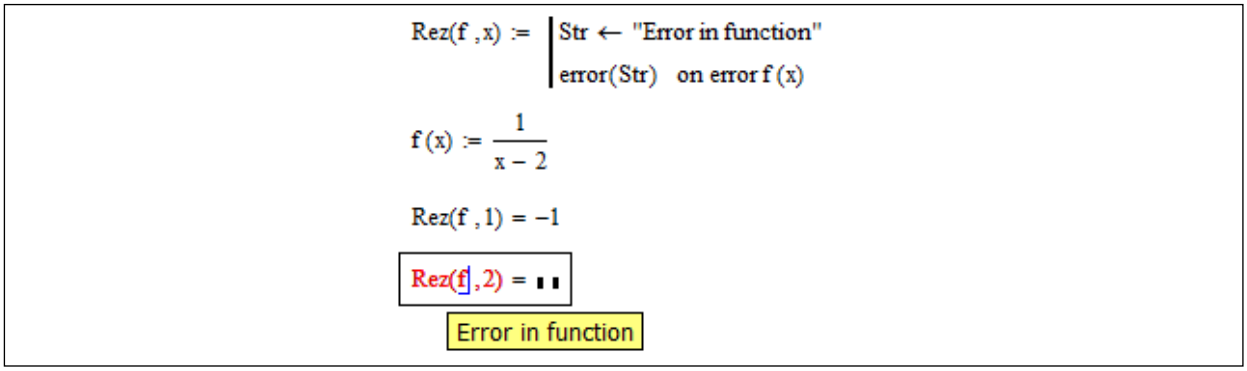
$$Rez(f,x) := \text{"Ошибка"} \quad \text{on error } f(x)$$

$$f(x) := \frac{1}{x - 2} \qquad g(z) := \log(z - 1) \qquad h(x) := x^{\frac{-1}{3}}$$

$$Rez(f,1) = -1 \qquad Rez(g,2) = 0 \qquad Rez(h,8) = 0.5$$

$$Rez(f,2) = \text{"Ошибка"} \qquad Rez(g,1) = \text{"Ошибка"} \qquad Rez(h,0) = \text{"Ошибка"}$$

Для обработки ошибок полезна также функция **error(S)**, которая, будучи помещенной в программный модуль, при возникновении ошибки выводит всплывающую подсказку с сообщением, хранящимся в символьной переменной S.



Условный оператор if

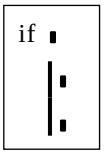
Условная инструкция **if** позволяет строить разветвляющиеся алгоритмы. Совместно с ней могут использоваться инструкции прерывания **break** и иного выбора **otherwise**. Инструкция прерывания **break** вызывает прерывание выполнения программы всякий раз, как она встречается и чаще всего используется совместно с инструкциями **while**, **for**, обеспечивая переход в конец тела цикла.

Формат: Выражение **if** Условие

Шаблон: ▮ if ▮

Из внешнего вида шаблона очевидно, что здесь поле для «иначе» не предусмотрено. Значит, в альтернативных ситуациях используется инструкция иного выбора **otherwise**.

Если при истинности условия необходимо программировать более одного действия, то нажимаем **Add Line** на поле слева и **if** изменит свой шаблон так:

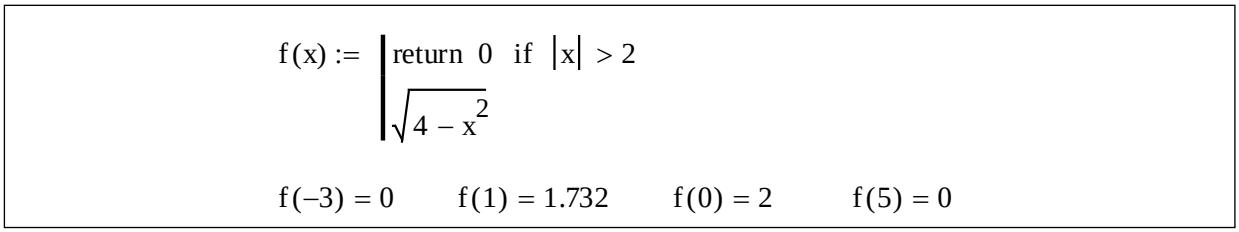


Особая инструкция **return** прерывает выполнение программы и возвращает значение операнда, стоящего за ней.

П р и м е р 1. Функцию

$$f(x)=\begin{cases}0, & \text{если } |x|>2,\\ \sqrt{4-x^2}, & \text{если } |x|\le 2\end{cases}$$

можно запрограммировать таким способом:



Это же можно представить, например, и так – при помощи оператора **if**:

$$f(x) := \text{if}\left(|x| > 2, 0, \sqrt{4 - x^2}\right)$$

$f(-3) = 0 \qquad f(1) = 1.732 \qquad f(0) = 2 \qquad f(5) = 0$

Рассмотрим применение инструкции **if** в сочетании с инструкциями **return, on error** и функции **error**, задающей вывод всплывающей подсказки при указании мышкой на выражение, содержащее ошибку. Инструкция **on error** может применяться и вне программного модуля, тогда она играет роль процедуры обработки ошибок.

П р и м е р 2. Применение инструкций **return** и **on error**:

$$\text{Ret}(i) := \begin{cases} \text{return "One"} & \text{if } i = 1 \\ \text{return "Two"} & \text{if } i = 2 \\ \text{error("No value")} & \text{otherwise} \end{cases}$$

$\text{Ret}(1) = \text{"One"}$

$\text{Ret}(2) = \text{"Two"}$

$\text{Ret}(3) =$

$$S(x) := \begin{cases} \text{return } 1 & \text{if } x = 0 \\ \dfrac{\sin(x)}{x} & \text{otherwise} \end{cases}$$

$$Yy(x) := \left(\frac{1}{x}\right) \cdot \sin(x)$$

$Y(x) := 1 \quad \text{on error } Yy(x)$

$S(0) = 1$ $\qquad Y(0) =$

$S(1) = 0.841$ $\qquad Y(1) = 0.841$

$S(2) = 0.455$ $\qquad Y(2) = 0.455$

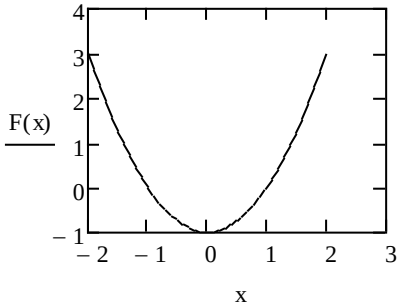
У п р а ж н е н и е. Прокомментируйте описанный выше пример.

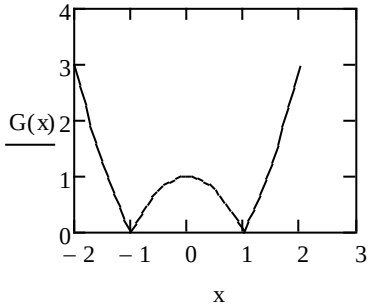
П р и м е р 3.

$x := -2, -1.9 .. 2$

$F(x) := x^2 - 1$

$$G(x) := \begin{cases} -F(x) & \text{if } -1 \le x \le 1 \\ F(x) & \text{otherwise} \end{cases}$$

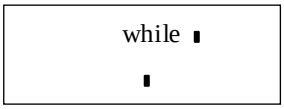




Оператор цикла While

Через панель программирования вводим инструкцию **While.** Появится **While** и два поля ввода: справа для ввода логического выражения на организацию цикла, снизу – для задания операторов цикла. Если в цикле необходимо выполнить несколько операторов, то добавляем поля для них клавишей “**]**” или **Add Line**.

Шаблон:



Такой цикл может зациклиться, если не наступит условие завершения цикла. Тогда предварительно используется оператор прерывания **break**. Для продолжения работы после прерывания программы используется инструкция **continue**.

П р и м е р 1. Для вычисления $\sqrt{a},\ \ a>0$ реализовать метод Ньютона решения уравнения $x^2-a=0$ в виде $x_{n+1}=\frac{1}{2}\left(x_n+\frac{a}{x_n}\right)$.

Решение представлено ниже:

$$\text{SqRoot}(a,\varepsilon):=\left|\begin{array}{l} xn\leftarrow\frac{a}{2} \\ n\leftarrow 0 \\ \text{while}\ \left|xn^2-a\right|>\varepsilon \\ \quad\left|\begin{array}{l} xn\leftarrow\frac{1}{2}\left(xn+\frac{a}{xn}\right) \\ n\leftarrow n+1 \\ \text{break if } n\geq 100 \end{array}\right. \\ \text{return}\ \begin{pmatrix} xn \\ n \end{pmatrix} \end{array}\right.$$

$$\text{SqRoot}\left(1234,10^{-1}\right)=\begin{pmatrix} 35.128368495194096 \\ 7 \end{pmatrix}$$

$$\text{SqRoot}\left(1234,10^{-6}\right)=\begin{pmatrix} 35.128336140515486 \\ 8 \end{pmatrix}$$

$$\text{SqRoot}\left(1234,10^{-12}\right)=\begin{pmatrix} 35.12833614050059 \\ 9 \end{pmatrix}$$

$$\text{SqRoot}\left(1234,10^{-13}\right)=\begin{pmatrix} 35.12833614050059 \\ 100 \end{pmatrix}$$

$$\sqrt{1234}=35.12833614050059$$

П р и м е р 2. Дан вектор, компоненты которого являются натуральными числами. Посчитать сумму и количество нечётных элементов в векторе.

```
Odd(v) := | n ← length(v)
          | S ← 0
          | k ← 0
          | for i ∈ 0 .. n − 1
          |     continue if mod(v_i, 2) = 0
          |     S ← S + v_i
          |     k ← k + 1
          | return (S)
          |        (k)
```

$v := (2\ \ 7\ \ 5\ \ 6\ \ 4\ \ 1\ \ 3\ \ 12)^T$ $\quad Odd(v) = \begin{pmatrix} 16 \\ 4 \end{pmatrix}$

$b := (1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8\ \ 9\ \ 10\ \ 11\ \ 12)^T$ $\quad Odd(b) = \begin{pmatrix} 36 \\ 6 \end{pmatrix}$

Оператор цикла for

На панели программирования находим конструкцию **for**. Появится шаблон, в котором по смыслу задачи заполняются поля. Указывается локальная переменная цикла, диапазон ее изменения, один или несколько операторов цикла.

П р и м е р 1. Посчитать сумму первых n натуральных чисел.

```
Sum(n) := | s ← 0
          | for i ∈ 1 .. n
          |   s ← s + i
```

$n := 44$ $\quad Sum(n) = 990$

П р и м е р 2.

```
join(r, s) := | m ← 0
              | for x ∈ r, s
              |     V_m ← x
              |     m ← m + 1
              | V
```

$r := \begin{pmatrix} 100 \\ 101 \\ 102 \end{pmatrix}$ $\quad s := \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ $\quad join(r,s) = \begin{pmatrix} 100 \\ 101 \\ 102 \\ 1 \\ 2 \end{pmatrix}$ $\quad join(s,r) = \begin{pmatrix} 1 \\ 2 \\ 100 \\ 101 \\ 102 \end{pmatrix}$

Программы в программах

Программу можно определить, а потом вызывать ее в других программах так, будто она есть подпрограмма. Функцию можно определить рекурсивно.

П р и м е р 1. Метод Ньютона:

$$\text{nroot}(f, df, x) := \left| \begin{array}{l} xn \leftarrow x - \dfrac{f(x)}{df(x)} \\ \text{while } |xn - x| > TOL \\ \quad \left| \begin{array}{l} x \leftarrow xn \\ xn \leftarrow x - \dfrac{f(x)}{df(x)} \end{array} \right. \\ xn \end{array} \right.$$

$$f(x) := x^2 - 4$$

$$df(x) := 2 \cdot x$$

$$\text{nroot}(f, df, 1) = 2$$

П р и м е р 2. Подсчитать интеграл адаптивным методом, используя подпрограмму метода Симпсона:

$$\text{IntSimp}(f, a, b, N) := \left| \begin{array}{l} s \leftarrow 0 \\ h \leftarrow \dfrac{b - a}{N} \\ \text{for } i \in 1..N \\ \quad \left| \begin{array}{l} x \leftarrow a + (i - 1) \cdot h \\ s \leftarrow s + \dfrac{h}{6} \cdot \left(f(x) + 4 \cdot f\left(x + \dfrac{h}{2} \right) + f(x + h) \right) \end{array} \right. \\ s \end{array} \right.$$

$$\text{Adapt}(f, a, b) := \left| \begin{array}{l} x \leftarrow \text{IntSimp}(f, a, b, 10) \\ x \quad \text{if } |x - \text{IntSimp}(f, a, b, 5)| \le TOL \\ \left(\text{Adapt}\left(f, a, \dfrac{a + b}{2} \right) + \text{Adapt}\left(f, \dfrac{a + b}{2}, b \right) \right) \quad \text{otherwise} \end{array} \right.$$

$$a := 0 \qquad b := 1$$

$$ft(x) := \sin(x)$$

$$\text{Adapt}(ft, a, b) = 0.45969771 \qquad \int_a^b ft(x)\, dx = 0.45969769$$

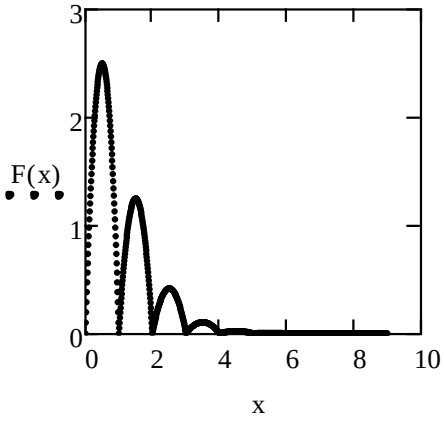
Здесь функция **Adapt** определена рекурсивно.

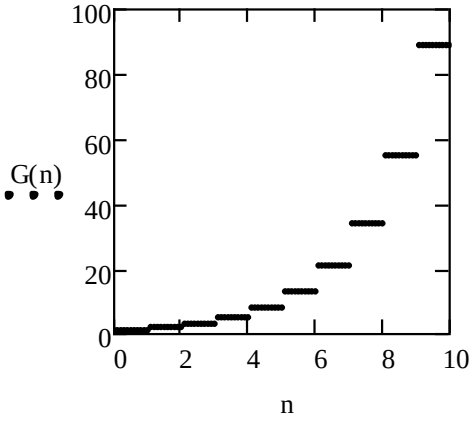
П р и м е р 3. Приведем пример рекурсивных программ:

$$G(n) := \text{if}(n \le 1, 1, G(n-1) + G(n-2))$$

$$F(x) := \text{if}\left(0 \le x < 1, 10 \cdot x - 10 \cdot x^2, \frac{F(x-1)}{\text{ceil}(x)}\right)$$

$$x := 0, 0.01 .. 9 \qquad n := 0, 0.1 .. 10$$





П р и м е р 4. Программа для нахождения пересечения двух множеств чисел, размещенных в векторах *v* и *w*.

```
in(v, x) :=
    t ← 0
    for y ∈ v
        t ← 1 if x = y
    t
```

```
intersect(v, w) :=
    u ← 0
    i ← 0
    for x ∈ v
        if in(w, x)
            u_i ← x
            i ← i + 1
    u
```

$$a := \begin{pmatrix} 1 \\ 3 \\ 2 \\ 9 \\ 8 \end{pmatrix} \qquad b := \begin{pmatrix} 3 \\ 0 \\ 7 \\ 8 \\ 1 \\ 5 \end{pmatrix}$$

$$\text{intersect}(a, b) = \begin{pmatrix} 1 \\ 3 \\ 8 \end{pmatrix}$$

П р и м е р 5. “Решето Эратосфена”.

```
prime(n) := | for i ∈ 2..√(n + 1)
            |   for k ∈ 2..n/i
            |     S_(k·i) ← 1
            | m ← 0
            | for j ∈ 2..n
            |   if S_j = 0
            |     | P_m ← j
            |     | m ← m + 1
            | P
```

prime(2500) =

	0
0	2
1	3
2	5
3	7
4	11
5	13
6	17
7	19
8	23
9	29
10	...

Последние простые числа видны на экране, а остальные будут доступны через полосу прокрутки после того, как щелкнуть по этому массиву.

Файлы данных

MathCad включает многие функции для чтения и записи данных. Рассмотрим некоторые из них.

READPRN, **WRITEPRN** и **APPENDPRN** – считывают целую матрицу из файла со строками и столбцами данных или записывают в виде такога файла матрицу с рабочего листа MathCad.

1. Перечисленные функции доступа должны печататься большими буквами или вставляться через меню.

2. Если MathCad не может найти файл данных, то он сразу выдает ошибку “**File not found**”; если файл не соответствующего формата – “**File error**”.

3. Каждое новое равенство с использованием функции доступа заново открывает файл данных.

Применение функций:

A:=**READPRN**(File) – каждая строка в файле становится строкой в матрице A;

WRITEPRN(File):=A – каждая строка матрицы становится строкой в файле;

APPENDPRN(File):=A – дописывает в существующий файл матрицу по строкам.

Здесь File – строковая переменная, содержащая имя файла, если каталог известен по умолчанию, или путь к файлу и имя файла.

При работе с документом MathCad считает каталогом по умолчанию тот, с которого документ загружен, или в котором он последний раз запоминался.

Каждая строка в файле данных должна содержать одинаковое количество значений. Функция **READPRN**(*file*) читает весь файл данных целиком, сама определяет число строк и столбцов и создает матрицу из этих данных. Она игнорирует текст в файле данных.

П р и м е р 1. Пусть в файле **matrix.txt** хранится матрица 4×3. Выполним следующее операторы:

M := READPRN("matrix.txt")

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 0 & 1 & 2 \end{pmatrix} \qquad V0 := M^{\langle 0 \rangle} \qquad V1 := M^{\langle 1 \rangle} \qquad V0 = \begin{pmatrix} 1 \\ 4 \\ 7 \\ 0 \end{pmatrix} \qquad V1 = \begin{pmatrix} 2 \\ 5 \\ 8 \\ 1 \end{pmatrix}$$

С массивами V0 и V1 можно работать как с простыми одномерными массивами.

Встроенные переменные **PRNCOLWIDTH** (ширина столбца, по умолчанию равна 8) и **PRNPRECISION** (число значащих цифр, по умолчанию равно 4) определяют формат данных, создаваемых MathCad.

Чтобы изменить эти значения, нужно или изменить значения через меню Math ─ Options… ─ Built-In Variables..., или сделать это явно в программе.

П р и м е р 2. Создание массивов, объединенных в матрицу:

$i := 0 .. 99 \qquad x_i := \sin\left(\frac{i}{10}\right) \qquad y_i := \cos\left(\frac{i}{10}\right) \qquad z_i := \tan\left(\frac{i}{10}\right)$

WRITEPRN("trigs.txt") := augment(augment(x, y), z)

A := READPRN("trigs.txt") A =

	0	1	2
0	0	1	0
1	0.1	0.995	0.1
2	0.199	0.98	0.203
3	0.296	0.955	0.309
4	0.389	0.921	0.423
5	0.479	0.878	...

Используя функцию **augment(*x, y*)**, можно объединить отдельные переменные в массивы и затем записать их все в файл данных.

П р и м е р 3.

$i := 0..5 \qquad j := 0..7$

$A_{i,j} := i \cdot \sin\left(j \cdot \frac{\pi}{5}\right)$

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.59 & 0.95 & 0.95 & 0.59 & 0 & -0.59 & -0.95 \\ 0 & 1.18 & 1.9 & 1.9 & 1.18 & 0 & -1.18 & -1.9 \\ 0 & 1.76 & 2.85 & 2.85 & 1.76 & 0 & -1.76 & -2.85 \\ 0 & 2.35 & 3.8 & 3.8 & 2.35 & 0 & -2.35 & -3.8 \\ 0 & 2.94 & 4.76 & 4.76 & 2.94 & 0 & -2.94 & -4.76 \end{pmatrix}$$

PRNPRECISION := 3

PRNCOLWIDTH := 10

WRITEPRN("sinevals.txt") := A

M := READPRN("sinevals.txt")

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.588 & 0.951 & 0.951 & 0.588 & 0 & -0.588 & -0.951 \\ 0 & 1.18 & 1.9 & 1.9 & 1.18 & 0 & -1.18 & -1.9 \\ 0 & 1.76 & 2.85 & 2.85 & 1.76 & 0 & -1.76 & -2.85 \\ 0 & 2.35 & 3.8 & 3.8 & 2.35 & 0 & -2.35 & -3.8 \\ 0 & 2.94 & 4.76 & 4.76 & 2.94 & 0 & -2.94 & -4.76 \end{pmatrix}$$

Так создали файл **sinevals.txt** со столбцами шириной в 10 символов, который содержит числа с тремя значащими цифрами.

2. ПРАКТИЧЕСКИЙ РАЗДЕЛ

2.1. Темы лабораторных работ

3 семестр

Тема 1. Основные понятия и подходы технологий программирования.

Тема 2. Топология языков программирования разных поколений

Тема 3. Версии реализации и среды разработки Borland Pascal, Free Pascal.

Тема 4. Free Pascal.

Тема 5. Динамические структуры.

Тема 6. Списки.

Тема 7. Стеки.

Тема 8. Очереди.

Тема 9. Деревья.

Тема 10. Деревья сильноветвящиеся.

Тема 11. Деревья сильноветвящиеся.

Тема 12. Методы разработки алгоритмов.

Тема 13. Алгоритмы типа «разделяй и властвуй».

Тема 14. «Жадные» алгоритмы.

Тема 15. Алгоритмы с возвратом.

Тема 16. Поиск с возвратом.

Тема 17. Поиск с возвратом и локальный поиск.

Тема 18. Фракталы.

4 семестр

Тема 1. Подготовка модуля для анализа трудозатрат методов сортировки.

Тема 2. Сортировка обменом.

Тема 3. Сортировка включениями.

Тема 4. Сортировка выбором.

Тема 5. Сортировка файлов.

Тема 6. Технологии реализации алгоритмов.

Тема 7. ООП.

Тема 8. Совместимость Объектных Типов

Тема 9. Экземпляры объектов в динамической памяти

Тема 10. ООП. Практика использования.

Тема 11. Архитектура MathCad и его интерфейс.

Тема 12. MathCad. Векторы и матрицы.

Тема 13. MathCad. Встроенные операторы и функции.

Тема 14. MathCad. Программирование.

Тема 15. MathCad. Решение уравнений и систем.

Тема 16. Файлы данных. Анимация.

Тема 17. Тест "Mathcad".

2.2. Контролирующие задания «Основы работы в Mathcad»

Введение в MathCad

1. Ознакомится с подменю каждого из пунктов меню.

2. Выяснить назначение кнопок панелей инструментов, наезжая на них мышкой и читая всплывающие подсказки.

3. Изменить вид окна MathCad, убрав и добавив отдельные панели.

4. Переместить панели инструментов и форматирования.

5. Раскрывая палитры, ознакомится с их содержанием.

6. Ввести шаблоны различных операторов и подумать, сможете ли вы заполнить их самостоятельно.

7. Загрузить подряд несколько документов из «быстрых шпаргалок» (Help ► QuickSheets) и проанализировать их.

8. Распечатать какой-нибудь документ с текстом, формулами и графиками.

9. Опробовать работу справочной системы по контексту и по индексу.

Вычисление значений числовых выражений

1. Вычислить значения следующих выражений:

$$\frac{456}{\left(168-\frac{3}{5}\right)\cdot 1{,}45}+5; \qquad \frac{\sqrt{17-12\sqrt{3}}}{\sqrt{(2-\sqrt{5})^2}}+4\sqrt[3]{\sqrt{2}}.$$

2. Вычислить значения выражений (здесь i мнимая единица):

$$\sqrt[4]{\sqrt{3}+i}; \qquad \left(\frac{\sqrt{3}}{2}-\frac{1}{2}i\right)^{12}; \qquad \frac{5+2i}{2-5i}-\frac{3-4i}{4+3i}-\frac{1}{i}.$$

Векторы и матрицы, решение систем линейных алгебраических уравнений

1. Создать вектор значений cos(x) при x = 0, 1, …, 10.

2. Решить по формулам Крамера системы уравнений:

$$\begin{cases} 2x+y+3z=-9, \\ 8x+3y+5z=-13, \\ 2x+5y-z=-5, \end{cases} \qquad \begin{cases} 2x-4y-7z+13=0, \\ -2x+5y-3z-1=0, \\ 5x-8y-10z+3=0. \end{cases}$$

3. Решить следующие системы уравнений:

$$\begin{cases} 2x+19y=-1, \\ 8x+31y=-1, \end{cases} \quad \begin{cases} -25x+55y-11=0, \\ 59x-67y+39=0, \end{cases} \quad \begin{cases} 29x-43y+137=0, \\ 57x-89y+314=0. \end{cases}$$

Вычисление сумм и произведений

1. Найти суммы и произведения первых девяти членов следующих прогрессий:

а) арифметической b_n, где $b_1=6,4,\ d=0,8$;

б) геометрической c_n, где $c_1=1,\ q=-2$.

2. Найти следующие суммы:

$$\frac{2}{1\cdot 3}+\frac{2}{3\cdot 5}+\ldots+\frac{2}{(2n-1)\cdot(2n+1)}; \qquad \sum_{n=1}^{M}(-1)^{n-1}\frac{x^{2n-1}}{(2n-1)!}..$$

3. Найти следующие произведения:

$$\left(1-\frac{1}{2^2}\right)\left(1-\frac{1}{3^2}\right)\cdots\left(1-\frac{1}{n^2}\right); \quad \left(1-\frac{1}{2^3}\right)\left(1-\frac{1}{3^3}\right)\cdots\left(1-\frac{1}{n^3}\right).$$

Функции, функции пользователя, производные и первообразные

1. Описать следующие функции и построить для них таблицу значений:

$$f(x)=\frac{2\cdot x+1}{x}+\frac{4\cdot x}{2\cdot x+1}-5; \qquad S(x)=\log_3 x+\log_x 9-3.$$

2. Найти значения производных следующих функций:

$f(x)=15\cdot x\cdot\cos x-e^x$ при x = 1; 0; 3,5.

3. Найти одну из первообразных для каждой из следующих функций:

$$f(x)=x\sin x; \qquad f(x)=\frac{1+x^2}{x^2}.$$

4. Вычислить интегралы: $\int_0^{\pi}\left(x^2+\sin(x)\right)dx$; $\int_0^{1}\left(\sin(3x)\cdot\sin(2x)-1\right)dx$.

Работа с графикой, построение двумерных графиков, работа с трехмерной графикой

1. Построить графики нескольких элементарных функций и отформатировать их для получения наибольшей наглядности.

2. Построить графики функции $y(x)=\sin(1/x)$, когда аргумент изменяется от –1 до 1 с шагом 0.1 и 0.01. Сравнить их.

3. Построить графики функций $y=\sin 5x$ и $y=x^2-2$. Найти точки пересечения графиков этих функций и промежутки, на которых одна из функций принимает значения большие, чем вторая.

4. Отформатировать поверхность, полученную в примере 2.

5. Построить известные канонические поверхности второго порядка.

6. Вычислить (предварительно сделав рисунок) площадь фигуры, ограниченной линиями: $y=a\cdot x,\quad y=0,\quad x=b,\quad x=3,\quad (a,b>0)$.

7. Вычислить объем тела, ограниченного однополостным гиперболоидом $\frac{x^2}{a^2}+\frac{y^2}{b^2}-\frac{z^2}{c^2}=1$ и плоскостями $z=0,\ z=h,\ (h>0)$.

8. Вычислить объем тела, образованного вращением вокруг оси *Ox* плоских фигур, ограниченных линиями: $x^2+y^2=1,\ x+y=1$.

Символьные вычисления в командном режиме

1. Разложить на элементарные следующие дроби:

$$\frac{13\cdot x^2-24\cdot x-54}{3\cdot x^3-27\cdot x^2+54\cdot x};\quad \frac{12\cdot x^3-72\cdot x^2+2\cdot x+18}{x^2-6\cdot x};\quad \frac{-72\cdot x-396}{x^4-4\cdot x^3-5\cdot x^2}.$$

2. Упростить следующие выражения: $\frac{a^2+2\cdot a\cdot c+c^2}{a^2+a\cdot c-a\cdot x-c\cdot x}$.

3. Символьно выполнить над дробями следующие действия:

$$\frac{b-6}{4-b^2}+\frac{2}{2\cdot b-b^2};\qquad \frac{a-30\cdot y}{a^2-100\cdot y^2}-\frac{10\cdot y}{10\cdot a\cdot y-a^2}.$$

4. Символьно упростить следующее выражение: $\sqrt{3698\cdot x^3-1849\cdot x^2}$.

Вычисление производных, вычисление интегралов

1. Символьно упростить следующие выражения:

$$\frac{d}{dx}\sin x;\quad \frac{d}{dx}\ln x;\quad \frac{d}{dx}\frac{1}{x^n};\quad \frac{d^2}{dx^2}\frac{x^5-3\cdot x+x^2-3\cdot x}{x^3+1}.$$

2. Символьно упростить следующие выражения:

$$\iiint \ln(x)dxdxdx;\quad \int\sqrt{ax+b}dx.$$

Вычисление сумм и произведений, пределы

1. Найдя частичные суммы как функцию верхнего предела суммирования, вычислить следующие суммы:

$$\sum_{n=1}^{\infty}\frac{x^{n+1}}{a^n(n+1)}\quad (a>0); \qquad \sum_{n=1}^{\infty}(-1)^{n-1}\frac{x^{2n-1}}{(2n-1)!}.$$

2. Найдя частичные произведения как функцию верхнего предела оператора накопления произведения, расчитать следующие произведения:

$$\left(1+\frac{1}{2}\right)\left(1+\frac{1}{3}\right)\cdots\left(1+\frac{1}{n}\right)\cdots; \qquad \left(1-\frac{1}{2^3}\right)\left(1-\frac{1}{3^3}\right)\cdots\left(1-\frac{1}{n^3}\right)\cdots.$$

3. Найти следующие пределы: $\lim\limits_{y\to a}\dfrac{2e^y}{y^2-6}$; $\lim\limits_{x\to\pm\infty}\left(\sqrt{x^2+1}-x\right)\cdot x.$

4. Найти обычный, левосторонний и правосторонний пределы следующих выражений: $\dfrac{\sin^2(ax)-\mathrm{tg}^2 x}{x^4-a}$ при x, стремящемся к $0,\ a^{\frac{1}{4}},\ \dfrac{\pi}{2}$.

5. Найти по определению производные следующих функций:

$g(x)=28\cdot\cos x-13\cdot\sin^2 x$ при $x=0,\ \dfrac{\pi}{2},\ \pi.$

Разложение по степеням, разложение выражений

1. Символьно упростить следующие выражения:

$$\ln\exp(n\cdot x);\quad \frac{a^2+2\cdot a\cdot b+b^2}{a+b};\quad \sin^2 x+\cos^2 x.$$

Операции относительно заданной переменной

1. Найдите решение следующего уравнения: $\sqrt{x^2-3}+|x-1|=x.$

2. Решить уравнения относительно обеих переменных:

$x^4-a\cdot x^2-3\cdot a=0;\quad x^{a-5}=5.$

Выполнение символьных вычислений в явном виде

1. Найдя частичные суммы как функцию верхнего предела суммирования, вычислить следующую сумму: $\sum\limits_{n=1}^{\infty}(-1)^{n-1}\dfrac{x^{2n-1}}{(2n-1)!}.$

2. Найдя частичные произведения как функцию верхнего предела оператора накопления произведения, расчитать следующие произведения:

$$\left(1+\frac{1}{2}\right)\left(1+\frac{1}{3}\right)\cdots\left(1+\frac{1}{n}\right)\cdots; \quad \left(1-\frac{1}{2^3}\right)\left(1-\frac{1}{3^3}\right)\cdots\left(1-\frac{1}{n^3}\right)\cdots.$$

3. Найти следующие пределы: $\lim\limits_{y\to a}\dfrac{2e^{y}}{y^{2}-6}$; $\lim\limits_{x\to\pm\infty}\left(\sqrt{x^{2}+1}-x\right)\cdot x$.

4. Найти обычный, левосторонний и правосторонний пределы следующих выражений: $\dfrac{\sin^{2}(ax)-\operatorname{tg}^{2}x}{x^{4}-a}$ при x, стремящемся к 0, $a^{\frac{1}{4}}$, $\dfrac{\pi}{2}$.

5. Найти по определению производные следующих функций:

$g(x)=28\cdot\cos x-13\cdot\sin^{2}x$ при $x=0$, $\dfrac{\pi}{2}$, π.

Решение уравнений и систем, решение неравенств

1. Найти решение уравнения: $\sqrt{x^{2}-3}+|x-1|=x$.

2. Решить уравнения относительно обеих переменных:

$x^{4}-a\cdot x^{2}-3\cdot a=0$; $x^{a-5}=5$.

3. Решить следующие неравенства: $\log_{\frac{1}{3}}\log_{3}(x-1)>0$; $\dfrac{-2}{|x|+1}\geq|x|-2$.

4. Решить иррациональные неравенства: $\sqrt{x+1}>x-1$; $\sqrt{2\cdot x}>x-2$.

5. Методом интервалов решить неравенство: $\sin x^{2}-\sqrt{3}\cos x>\sqrt{3}$.

6. Решить систему уравнений: $\begin{cases}x\cdot y+y\cdot z=8,\\ y\cdot z+z\cdot x=9,\\ x\cdot z+y\cdot x=5.\end{cases}$

7. Решить систему неравенств: $\begin{cases}\dfrac{2\cdot x^{2}-7\cdot x+41}{x^{2}+7\cdot x+12}\leq\dfrac{x+2}{x+4},\\ \dfrac{4}{x-5}<\dfrac{-5}{x}.\end{cases}$

3. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

3.1. Перечень рекомендуемых средств диагностики

Для оценки профессиональных компетенций студентов используется следующий диагностический инструментарий:

– тест – выполнение заданий в тестовой форме;

– коллоквиум – устный опрос на лабораторных занятиях или тестирование;

– текущие контрольные работы.

3.2. Примерный перечень заданий для управляемой самостоятельной работы студентов

В качестве управляемой самостоятельной работы предлагается выполнение тестов.

Тема 1.2. Топология языков программирования разных поколений (2 ч)

Особенности работы с различными диалектами и реализациями языка Pascal. Версии реализации и среды разработки Borland Pascal, Free Pascal, Pascal ABC.

Форма контроля – компьютерное тестирование на портале *edummf.bsu.by* БГУ в LMS Moodle.

Тема 2.1. Динамические структуры данных (2 ч)

Однонаправленные и двунаправленные списки. Стеки. Очереди. Бинарные деревья. Создание идеально сбалансированного дерева. Создание дерева поиска. Обход дерева.

Форма контроля – компьютерное тестирование на портале *edummf.bsu.by* БГУ в LMS Moodle.

Тема 3.1 – 3.4. Методы разработки алгоритмов (2 ч)

Рекурсивные и не рекурсивные алгоритмы. Алгоритмы типа «разделяй и властвуй». «Жадные» алгоритмы и алгоритмы на полный перебор. Поиск с возвратом и локальный поиск. Фракталы. Алгебраические, геометрические, стохастические фракталы.

Форма контроля – компьютерное тестирование на портале *edummf.bsu.by* БГУ в LMS Moodle.

Тема 3.5. Сортировка данных (2 ч)

Внутренняя и внешняя сортировки. Сортировка массивов «in situ». Классы алгоритмов сортировок. Базовые алгоритмы. Улучшенные алгоритмы. Характеристики методов сортировки, анализ их ресурсов.

Сортировка обменом. Сортировка включениями. Сортировка выбором.

Форма контроля – компьютерное тестирование на портале *edummf.bsu.by* БГУ в LMS Moodle.

Тема 4.1. Основы объектно-ориентированной методологии разработки программ (ООП) (2 ч)

Основные принципы: абстрагирование, инкапсуляция, иерархия, модульность, наследование, полиморфизм. Описание объектов. Поля и методы. Виртуальные и динамические методы. Наследование. Экземпляры объектов. Динамические и виртуальные объекты. Конструкторы. Освобождение объектов. Деструкторы.

Форма контроля – компьютерное тестирование на портале *edummf.bsu.by* БГУ в LMS Moodle.

Тема 5.4. Система компьютерной алгебры MathCad (2 ч)

Программирование в MathCad. Условная передача управления, операторы циклов, использование подпрограмм и рекурсий.

Форма контроля – компьютерное тестирование на портале *edummf.bsu.by* БГУ в LMS Moodle.

3.3. Примерный перечень вопросов к зачету

2 курс 3 семестр

Исторический и современный взгляд на разработку ПО

1. Основные понятия и подходы технологий программирования.
2. Топология языков разных поколений.
3. Версии реализации и среды разработки Borland Pascal, Free Pascal, Pascal ABC.

Динамические структуры данных

4. Работа со связными динамическими структурами.
5. Списки.
6. Стеки.
7. Очереди.
8. Деревья.
9. Задачи поиска элемента в разных структурах данных.

10. Последовательный поиск в массиве, в списке, в дереве.
11. Бинарный поиск в массиве, в списке, в дереве поиска.

Методы разработки алгоритмов

12. Алгоритмы типа «разделяй и властвуй».
13. Рекурсивные и не рекурсивные алгоритмы.
14. Задача о ханойских башнях: рекурсивная реализация и реализация при помощи дерева.
15. Задача на получение перестановок чисел (рекурсивный и не рекурсивный варианты решения).
16. Фракталы.
17. **«Жадные» алгоритмы**. Задача на счастливые билеты.
18. Решение задачи получения сдачи эвристическим методом и методом полного перебора всех возможных вариантов (рекурсивным методом и с помощью сильно ветвящегося дерева).
19. Алгоритмы с возвратом. Задача обхода конем шахматной доски. Задача о восьми ферзях. Задача о восьми ладьях.
20. Поиск с возвратом и локальный поиск. Задача о рюкзаке.

3.4. Примерный перечень вопросов к экзамену

Методы разработки алгоритмов

1. Сортировка данных.
2. Внутренняя сортировка.
3. Классы сортировок по базовому алгоритму.
4. Сортировки обменом.
5. Сортировки включениями.
6. Сортировки выбором.
7. Внешняя сортировка.

Технологии программирования

8. Методологии программирования. ООП.
9. Основные принципы: инкапсуляция, наследование, полиморфизм.
10. Описание объектов. Поля и методы.
11. Виртуальные и динамические методы.
12. Наследование.
13. Экземпляры объектов.
14. Совместимость объектных типов. Совместимость между экземплярами объектов. Совместимость между указателями на экземпляры объектов. Совместимость между формальными и фактическими параметрами.
15. Динамические и виртуальные объекты.
16. Конструкторы. Освобождение объектов.
17. Деструкторы. Обработка ошибок при работе с динамическими объектами.

18. Объекты и модули.

Динамические структуры данных

19. Работа со связными динамическими структурами.
20. Списки.
21. Стеки.
22. Очереди.
23. Деревья.
24. Задачи поиска элемента в разных структурах данных.
25. Последовательный поиск в массиве, в списке, в дереве.
26. Бинарный поиск в массиве, в списке, в дереве поиска.

Математическое моделирование

27. Системы компьютерной математики.
28. Характерные черты MathCad.
29. Архитектура MathCad и его интерфейс.
30. Векторы и матрицы.
31. Встроенные операторы и функции.
32. Программирование.
33. Решение уравнений и систем.
34. Использование MathCad в курсах высшей математики.

4. ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ

4.1. Список рекомендуемой литературы

Основная:

1. Расолько, Г. А. Теория и практика программирования на языке Pascal / Г. А. Расолько, Ю.А. Кремень. - Минск. : Вышэйшая школа, 2022. – 533 с.

Дополнительная:

2. Расолька, Г.А. Метады праграміравання. Алгарытмы апрацоўкі данных / Г. А. Расолько, Ю. А. Кремень. . – Мн.: БДУ, 2008.

3. Расолько, Г.А. Аналитическая геометрия. Практикум с использованием Mathcad / Г. А. Расолько, Ю. А. Кремень. – Минск : Вышэйшая школа, 2019.– 271 с.

4. Кремень, Е. В. Численные методы. Практикум в MathCad. / Е. В. Кремень, Ю. А. Кремень, Г. А. Расолько. – Минск : Вышэйшая школа, 2019.– 256 с.

5. Ахо, А. В. Структуры данных и алгоритмы : учеб. пособие / А. В. Ахо, Д. Хопкрофт, Д. Д. Ульман. М. : Вильямс, 2000.

6. Кетков, Ю.Л., Кетков А.Ю. Свободное программное обеспечение Free Pascal для студентов и школьников / Ю.Л. Кетков, А.Ю. Кетков. – СПб.: БХВ-Петербург, 2011.

7. Васин, В. В. Элементы нелинейной динамики от порядка к хаосу / В. В. Васин. Екатеринбург : Изд-во Уральского ун-та, 2003.

4.2. Электронные ресурсы

1. Расолько Г. А., Кремень Ю. А., Кремень Е. В. Технологии программирования и алгоритмы обработки данных : пособие / Г. А. Расолько, Ю. А. Кремень, Е. В. Кремень. – Минск : БГУ, 2021. – [Электронный ресурс] – Режим доступа: https://elib.bsu.by/handle/123456789/271484 – Дата доступа: 24.02.2023.

2. Расолько Г. А., Кремень Е. В., Кремень Ю. А. Технологии программирования : учеб.-метод. пособие. В 2 ч. Ч. 1. Технологии реализации алгоритмов и обработка структур данных / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень . – Минск : БГУ, 2022. – [Электронный ресурс] – Режим доступа: https://elib.bsu.by/handle/123456789/277931 – Дата доступа: 24.02.2023.

3. Расолько Г. А., Кремень Е. В., Кремень Ю. А. Технологии программирования: учеб.-метод. пособие. В 2 ч. Ч. 2. Методы разработки алгоритмов и среды программирования языка Pascal / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень . – Минск : БГУ, 2022. – [Электронный ресурс] – Режим доступа: https://elib.bsu.by/handle/123456789/277933 – Дата доступа: 24.02.2023.

4. Расолько Г. А., Кремень Е. В., Кремень Ю. А. Технологии программирования: математическое моделирование и система компьютерной математики MathCad : учеб.-метод. пособие / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень . – Минск : БГУ, 2022. – – [Электронный ресурс] – Режим доступа: https://elib.bsu.by/handle/123456789/277920 – Дата доступа: 24.02.2023.

5. Расолько Г.А. Фракталы. Учебные материалы по вычислительной практике // Расолько Г.А., Е. В. Кремень, Ю. А. Кремень. – Минск : БГУ, 2020. –

42 с. – [Электронный ресурс] – Режим доступа: https://elib.bsu.by/handle/123456789/248829 – Дата доступа: 24.02.2023.

6. Расолько Г.А. Задания вычислительной практики по курсу «Методы программирования и информатика» : практикум. В 2 ч. Ч. II / Расолько Г.А., Е. В. Кремень, Ю. А. Кремень. – Минск : БГУ, 2020. – 85 с. – [Электронный ресурс] – Режим доступа: https://elib.bsu.by/handle/123456789/248828 – Дата доступа: 24.02.2023.

4.3. УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ

Номер раздела, темы	Название раздела, темы	Количество аудиторных часов		Количество часов УСР	Формы контроля знаний
		Лекции	Лабораторные занятия		
	2	3	4	5	6
3 семестр					
1	**Исторический и современный взгляд на разработку ПО**	**6**	**4**	**2**	
1.1.	Основные понятия и подходы технологий программирования	2			Тест “Технологии программирования”
1.2	Топология языков разных поколений	4	4	2	Тест “Отличительные особенности Free Pascal”
2	**Структуры данных. Абстракция данных**	**16**	**14**	**2**	
2.1	Динамические структуры данных	16	14	2	Тест “Динамические структуры данных”.
3	**Методы разработки алгоритмов**	**14**	**12**	**2**	
3.1	Рекурсивные и не рекурсивные алгоритмы	14	12	2	Тест “Методы разработки алгоритмов”
4 семестр					
3	**Методы разработки алгоритмов**	**8**	**6**	**2**	
3.2	Сортировка данных	8	6	2	Тест “Сортировки”.
4	**Технологии программирования**	**8**	**8**		
4.1	Методологии программирования. ООП	8	8		Тест “ООП”
5	**Математическое моделирование**	**18**	**16**	**2**	
5.1-5.3	Системы компьютерной математики. Характерные черты и архитектура MathCad.	8	6		
5.4	Программирование	4	4	2	Тест “Программирование в MathCad”
5.5	Использование MathCad в курсах высшей математики	6	6		