БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

В двух частях

Часть 2

МЕТОДЫ РАЗРАБОТКИ АЛГОРИТМОВ И СРЕДЫ ПРОГРАММИРОВАНИЯ ЯЗЫКА PASCAL

Рекомендовано
Учебно-методическим объединением
по естественно-научному образованию в качестве
учебно-методического пособия для студентов,
обучающихся по специальности «математика (по направлениям)»,
направление специальности «математика
(научно-педагогическая деятельность)»

Учебное электронное издание

Минск, БГУ, 2022

ISBN 978-985-881-164-8 (ч. 2)
ISBN 978-985-881-169-3

© Расолько Г. А., Кремень Е. В., Кремень Ю. А., 2022
© БГУ, 2022

УДК 004.432Pascal(075.8)
ББК 32.973.26-018.1я73-1

Р е ц е н з е н т ы:
кафедра математического анализа дифференциальных уравнений
и их приложений Брестского государственного
университета им. А. С. Пушкина (заведующий кафедрой
кандидат физико-математических наук, доцент *Н. Н. Сендер*);
кандидат технических наук *О. Г. Смолякова*

Расолько, Г. А. Технологии программирования [Электронный ресурс] : 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. – 1 электрон. опт. диск (CD-ROM). – ISBN 978-985-881-164-8.

Рассматриваются классические методы разработки алгоритмов, а также особенности работы с различными диалектами и реализациями языка Pascal. Издание направлено на развитие алгоритмического мышления, формирование знаний о свойствах алгоритмов и приобретение практических навыков разработки программ. Учебный материал сопровождается большим количеством примеров реализации алгоритмов на языке Pascal.

Минимальные системные требования:
PC, Pentium 4 или выше;
RAM 1 Гб; Windows XP/7/10; Adobe Acrobat.

Оригинал-макет подготовлен в программе MicroSoft Word.

В авторской редакции

Ответственный за выпуск *Е. В. Кремень*. Дизайн обложки *В. П. Явуз*.
Технический редактор *В. П. Явуз*. Компьютерная верстка *Е. В. Кремень*.

Подписано к использованию 31.01.2022. Объем 3,84 МБ.

Белорусский государственный университет.
Управление редакционно-издательской работы.
Пр. Независимости, 4, 220030, Минск.
Телефон: (017) 259-70-70.
e-mail: urir@bsu.by
http://elib.bsu.by/

СОДЕРЖАНИЕ

ТЕМА 1. МЕТОДЫ РАЗРАБОТКИ АЛГОРИТМОВ 5

Алгоритмы «разделяй и властвуй» 5
Задача о Ханойских башнях 5
Перестановки чисел 12

ТЕМА 2. «ЖАДНЫЕ» АЛГОРИТМЫ 15

«Жадные» алгоритмы 15
Счастливые билеты 15
Задача получения сдачи эвристическим методом 20

ТЕМА 3. АЛГОРИТМЫ НА ПОЛНЫЙ ПЕРЕБОР 23

Задача получения сдачи полным перебором 23
Задача получения остатка с помощью сильноветвистых деревьев 29

ТЕМА 4. ПОИСК С ВОЗВРАТОМ. ЗАДАЧИ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА 36

Задача обхода конем шахматной доски 36
Задача о восьми ферзях 41
Задача о восьми ладьях 46
Задача про устойчивые браки 47

ТЕМА 5. ПОИСК С ВОЗВРАТОМ И ЛОКАЛЬНЫЙ ПОИСК 49

Задача о «рюкзаке» 49
Поиск с возвратом 51

ТЕМА 6. ФРАКТАЛЫ 63

О Фракталах 63
Классификация фракталов 65
Алгебраические фракталы 65
Геометрические фракталы 79
Стохастические фракталы 94
Примеры практического использования фракталов 98

ТЕМА 7. ВЕРСИИ РЕАЛИЗАЦИИ И СРЕДЫ РАЗРАБОТКИ PASCAL. FREE PASCAL 101

Отличительные особенности Free Pascal 101
Данные языка Free Pascal 102
Числовые данные Free Pascal 103
Модуль Math 103
Символьные данные Free Pascal 105
Строки символов 105
Строки WideString 109
Строки UnicodeString 109
Преобразование строк 109
Новые функции преобразования числовых данных 110
Массивы в языке Free Pascal 112
Базовые операторы языка 116

Тема 8. Процедуры и функции Free Pascal 117
Процедуры и функции 117
Параметры 118
Параметры подпрограмм по умолчанию 118
Расширенный вызов функций 119
Перегрузка функций и процедур 120
Тема 9. Особенности работы с файлами, звуком, экраном 123
Управление файлами в стиле Windows 123
Функции для работы с дисками 123
Функции для работы с директориями 124
Функции для работы с файлами 124
Стандартные модули Free Pascal 127
Программирование с объектами 128
Особенности работы со звуком в Free Pascal 132
Особенности работы с экраном в текстовом режиме в Free Pascal 132
Особенности работы с графикой в Free Pascal 134
Особенности инициализации графического режима в Free Pascal 136
Особенности управления цветом в Free Pascal 140
Список литературы 143

ТЕМА 1
МЕТОДЫ РАЗРАБОТКИ АЛГОРИТМОВ

Содержание темы

- Алгоритмы «разделяй и властвуй»:
  - задача о Ханойских башнях,
  - перестановки чисел.
- «Жадные» алгоритмы.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

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 $\boldsymbol{n}$. Отобразите полученные расчеты в таблице.

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

Куда уходит детство и задача о конце света…

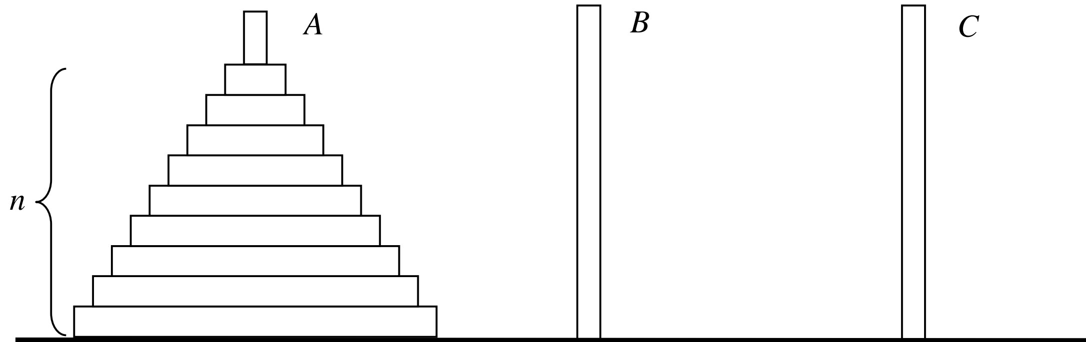
Подсказки для решения задачи

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.

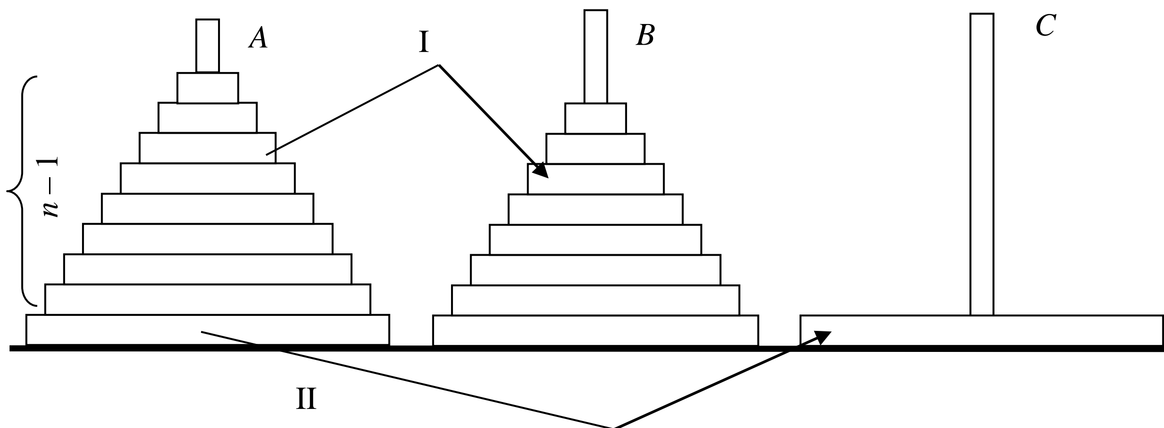
Задача о ханойских башнях

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

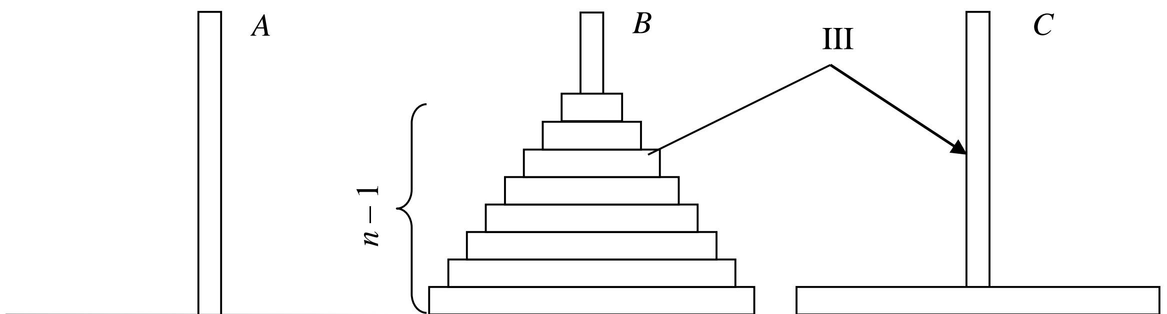


Алгоритм 1.

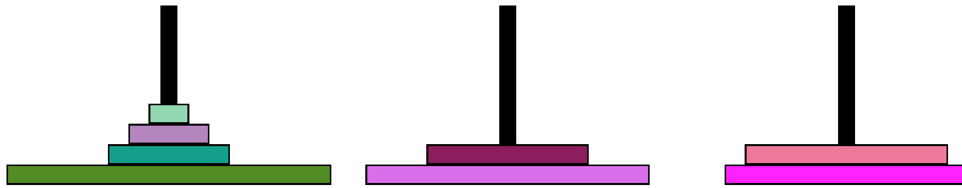
1. Переложить верхние $n-1$ дисков со стержня A на вспомогательный стержень B.
2. Нижний диск с первого стержня переложить на третий стержень C.



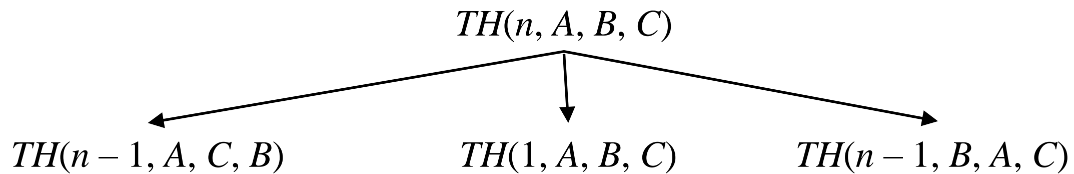
3. Переложить $n-1$ дисков с вспомогательного стержня B на третий стержень C.



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:

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

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.

Р е ш е н и е 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.

Решение 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.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 2
«ЖАДНЫЕ» АЛГОРИТМЫ

Содержание темы

- «Жадные» алгоритмы:
  - счастливые билеты,
  - задача получения сдачи эвристическим методом.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

«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 $\overset{k}{09\ldots9}$ будет последней, так как по нашему алгоритму получим $\overset{k}{10\ldots0}$, 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:

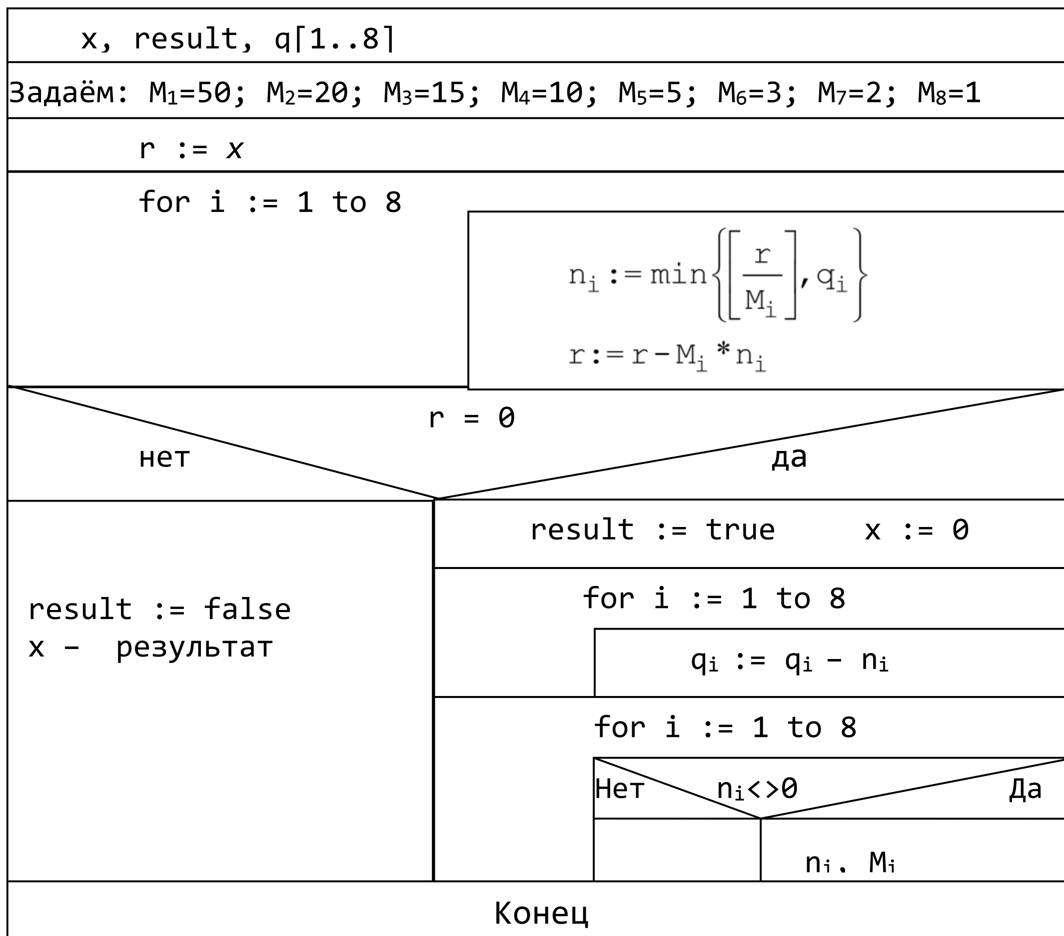
- x ($x > 0$) – сумма остатка;
- q[1..8] – первоначальное количество монет разного номинала соответственно перечню, данному выше.

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:

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.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?

ТЕМА 3
АЛГОРИТМЫ НА ПОЛНЫЙ ПЕРЕБОР

Содержание темы

- Задача получения сдачи полным перебором.
- Задача получения остатка с помощью сильноветвистых деревьев.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

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:

$$
\begin{aligned}
&RM(x;\ q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8) := (x = 0 \vee \\
&\quad \vee (x \ge 50 \wedge q_1 \ge 1 \wedge RM(x-50;\ q_1-1, q_2, q_3, q_4, q_5, q_6, q_7, q_8)) \\
&\quad \vee (x \ge 20 \wedge q_2 \ge 1 \wedge RM(x-20;\ q_1, q_2-1, q_3, q_4, q_5, q_6, q_7, q_8)) \\
&\quad \vee (x \ge 15 \wedge q_3 \ge 1 \wedge RM(x-15;\ q_1, q_2, q_3-1, q_4, q_5, q_6, q_7, q_8)) \\
&\quad \vee (x \ge 10 \wedge q_4 \ge 1 \wedge RM(x-10;\ q_1, q_2, q_3, q_4-1, q_5, q_6, q_7, q_8)) \\
&\quad \vee (x \ge 5 \wedge q_5 \ge 1 \wedge RM(x-5;\ q_1, q_2, q_3, q_4, q_5-1, q_6, q_7, q_8)) \\
&\quad \vee (x \ge 3 \wedge q_6 \ge 1 \wedge RM(x-3;\ q_1, q_2, q_3, q_4, q_5, q_6-1, q_7, q_8)) \\
&\quad \vee (x \ge 2 \wedge q_7 \ge 1 \wedge RM(x-2;\ q_1, q_2, q_3, q_4, q_5, q_6, q_7-1, q_8)) \\
&\quad \vee (x \ge 1 \wedge q_8 \ge 1 \wedge RM(x-1;\ q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8-1))).
\end{aligned}
$$

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 · 15 + 10, второй вариант: 15 · 2 + 20 + 10, третий вариант: 10 + 20 + 2 · 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 $GC(\ldots)$ 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 $x = 30$:

i	1	2	3	4	5	6	7	8
M_i	50	20	15	10	5	3	2	1
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 $GC(\ldots)$ 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:

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

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 4
ПОИСК С ВОЗВРАТОМ.
ЗАДАЧИ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Содержание темы

- Задача обхода конем шахматной доски.
- Задача о восьми ферзях.
- Задача о восьми ладьях.
- Задача про устойчивые браки.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

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.

	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

Для нумерации следующего очередного хода используем индекс $k\ (1 \le k \le 8)$. Изменения $\Delta x,\ \Delta y$ 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:

```
    l = 105
       1    1
       2    7
       3    5
       4    8
       5    2
       6    4
       7    6
       8    3
```

	1	2	3	4	5	6	7	8
1	Ф1							
2							Ф7	
3					Ф5			
4								Ф8
5		Ф2						
6				Ф4				
7						Ф6		
8			Ф3					

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.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 5
ПОИСК С ВОЗВРАТОМ И ЛОКАЛЬНЫЙ ПОИСК

Содержание темы

- Задача о «рюкзаке».
- Примеры полных задач.
- Поиск с возвратом.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

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:

- $if \ \sum_{i=1}^{k} b_i a_i < z$ $\rightarrow$ продлеваем массив b;
- $if \ \sum_{i=1}^{k} b_i a_i > z$ $\rightarrow$ тупик, бектрекинг;
- $if \ \sum_{i=1}^{k} b_i a_i = z$ $\rightarrow$ 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
```

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 6
ФРАКТАЛЫ

Содержание темы

- О Фракталах.
- Классификация фракталов:
  - алгебраические фракталы,
  - геометрические фракталы,
  - стохастические фракталы.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

О ФРАКТАЛАХ

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 $\lim_{k \to \infty} |z_k| = \infty$.

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

$$z_{k+1} = z_k^2 + c, \ k = 0, 1, 2, \ldots, \ \ z_0 = c,$$

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 $k \mod 256$ (или $k \mod 16$) в зависимости от количества цветов палитры.

10. Конец цикла по оси Oy.

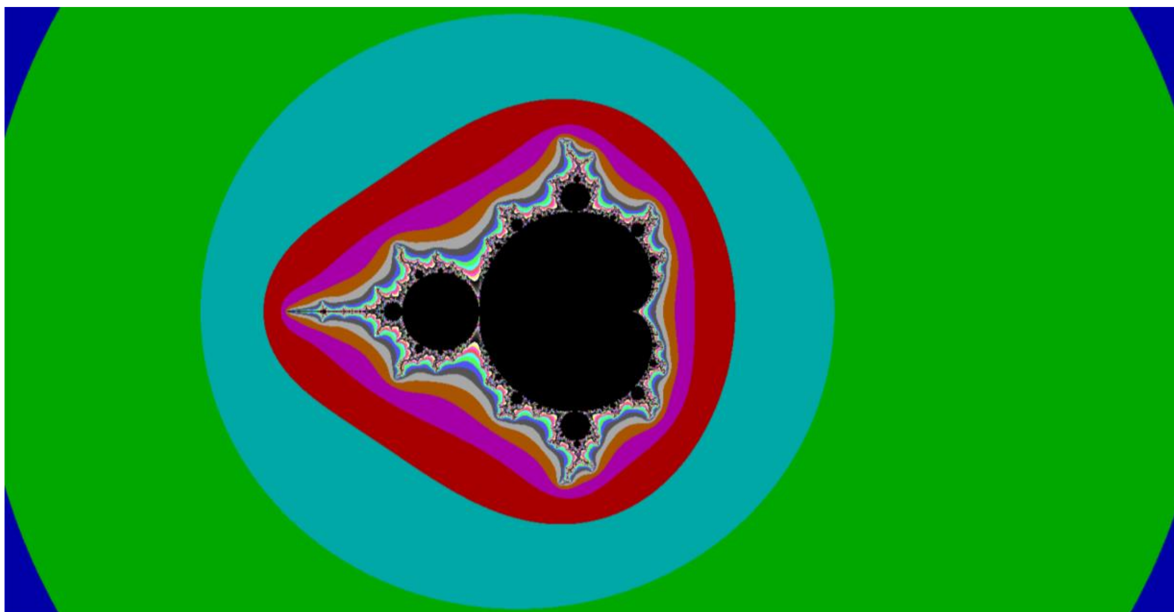
11. Конец цикла по оси Ox.

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
```

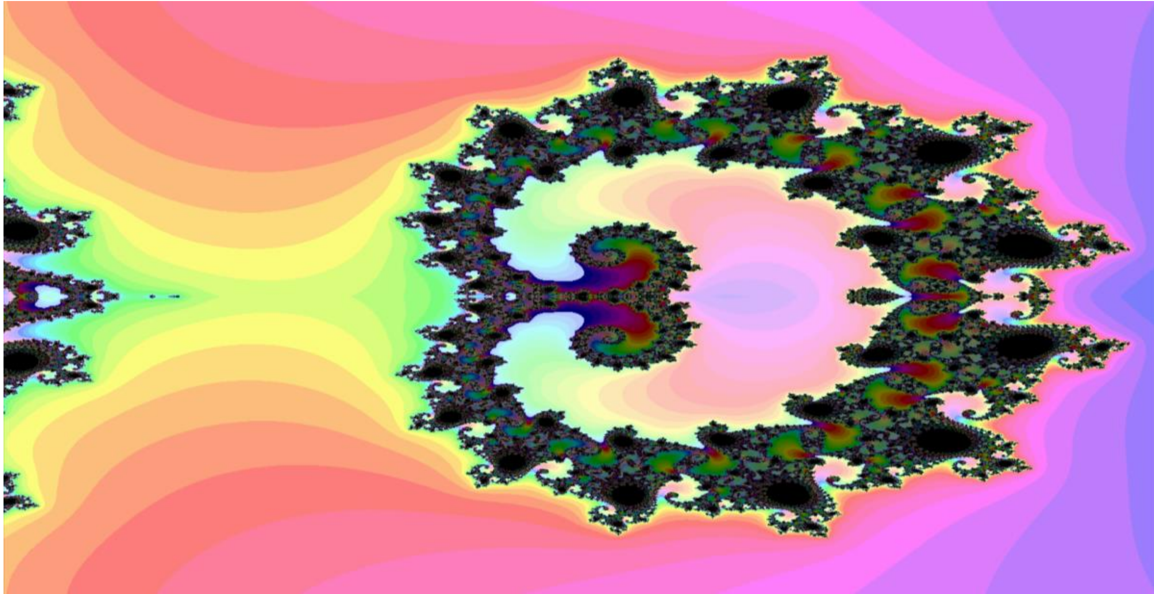
```
with r  do
    begin
     re := a.re + b.re;
     im := a.im + b.im;
    end;
end;

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.

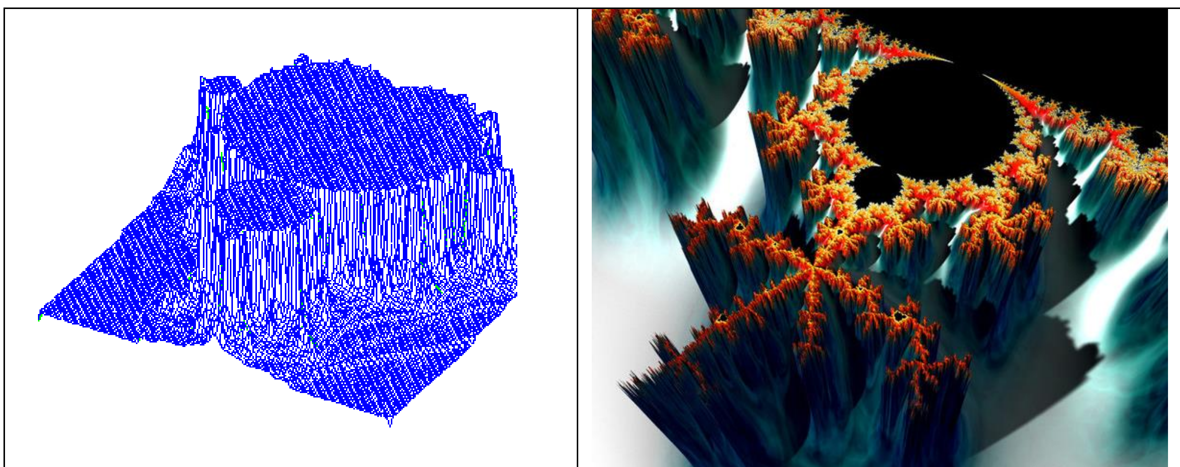
```
Program Maldelbrot_2;
uses  Graph, Crt;
var
   gd, gm : Integer;

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 $n = 3,\ c = p + iq$.

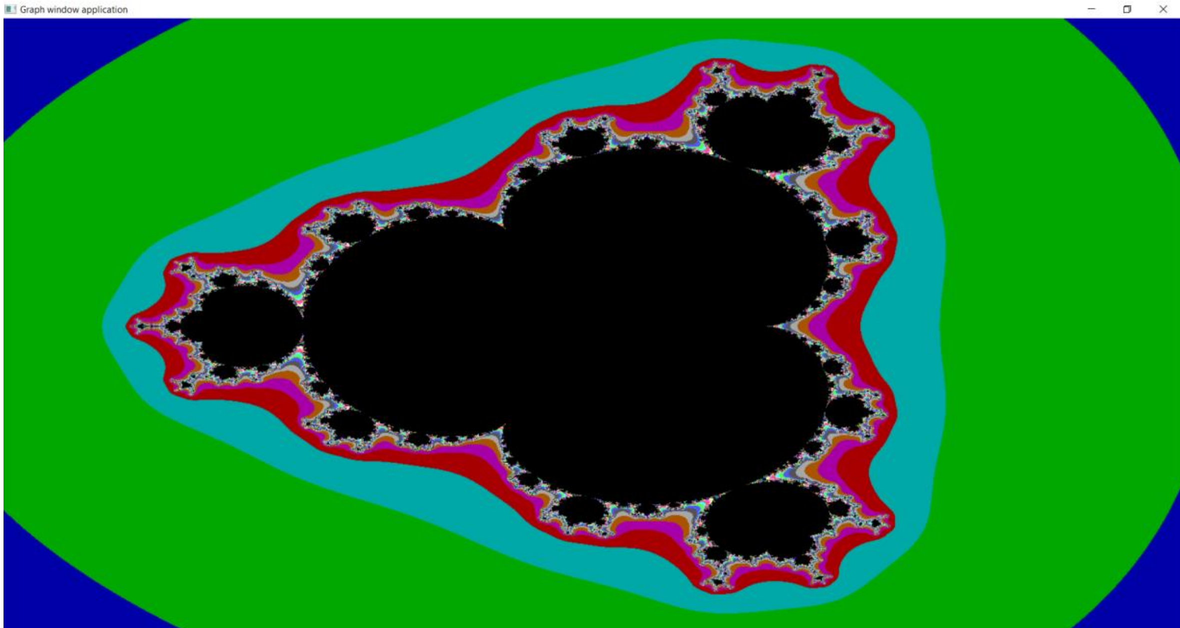
Получим итерационный процесс:

$$\begin{cases} x_{n+1} = x_n^3 - 3x_n\, y_n^2 + p, \\ y_{n+1} = 3x_n^2\, y_n - y_n^3 + q. \end{cases}$$

```
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.
```



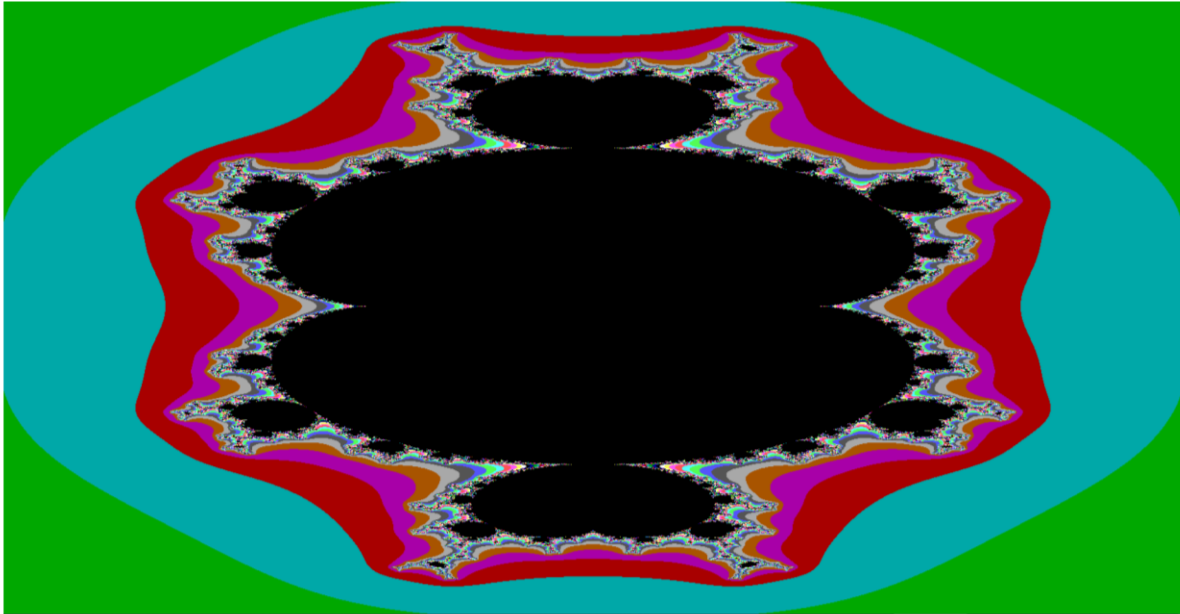
Для $n = 4$, изменяя в предыдущей программе функции:

```
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:



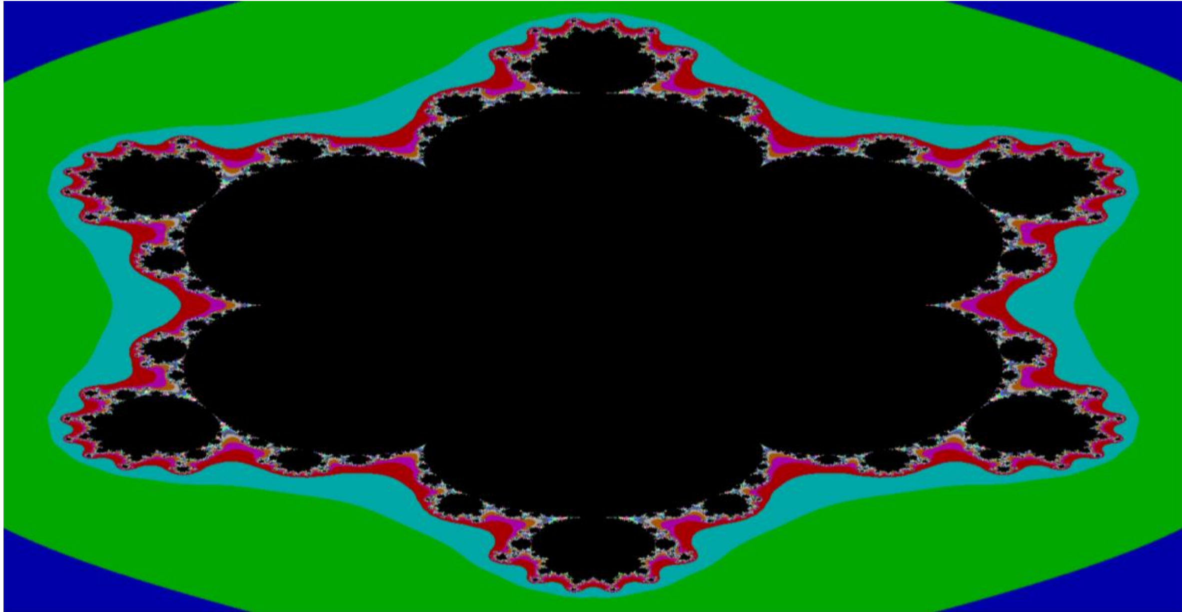
Для $n = 7$, изменяя в предыдущей программе функции:

```
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+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 $z=\left(\dfrac{z^2+q-1}{2z+q-2}\right)^2$, 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:

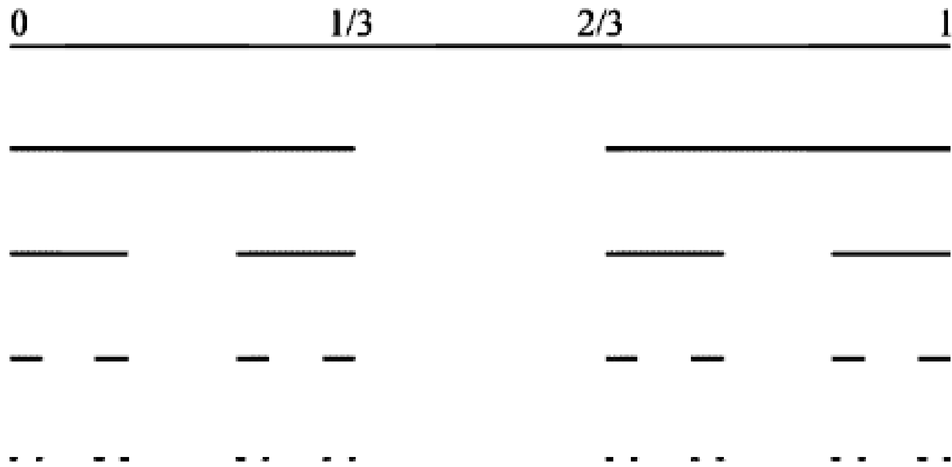
- множества Кантора (1-3D);
- Н-фрактал;
- треугольник Серпинского;
- кривая Гильберта;
- кривая Серпинского;
- кривая Коха;
- кривая Леви и другие;
- дерево.

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.

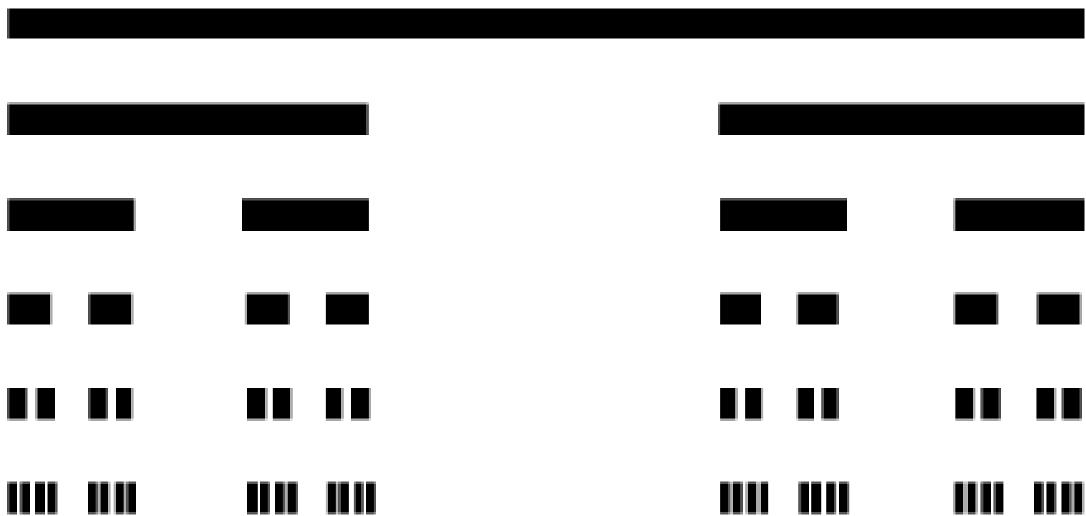
Такие фракталы хорошо программировать рекурсивными алгоритмами.

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.

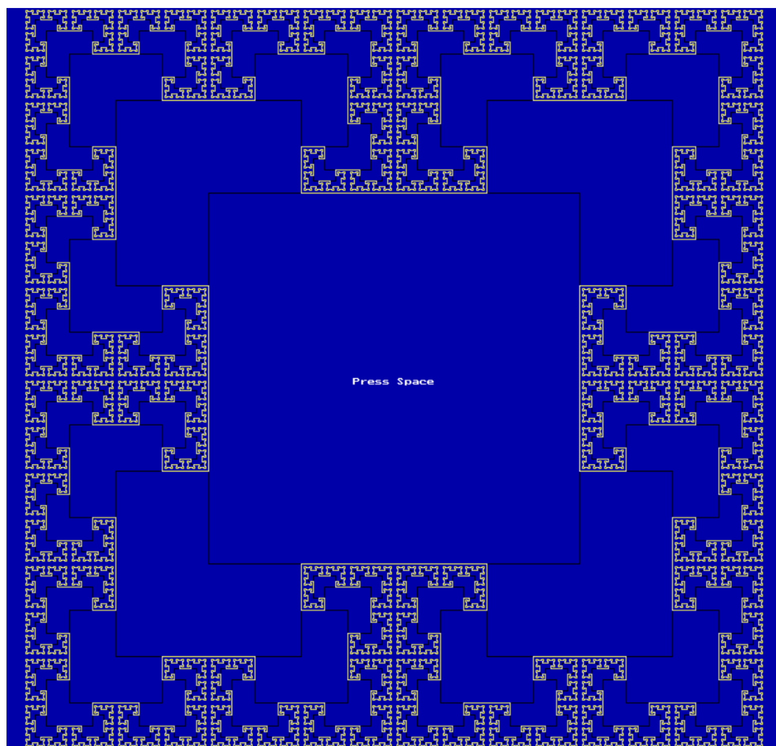
В итоге получится самоподобное множество – фрактал.

В следующей программе сохраняется образ границы вырезанного квадрата, и построение множества идет не от большего квадрата к меньшему, а наоборот.

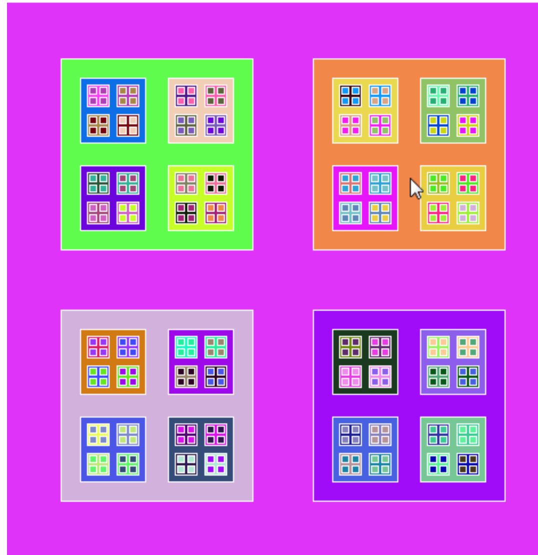
```
Program Cantor;
uses  Graph, crt;
const
      min_size = 1;
var
     Gd, Gm : Integer;
     Ch : char;

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:

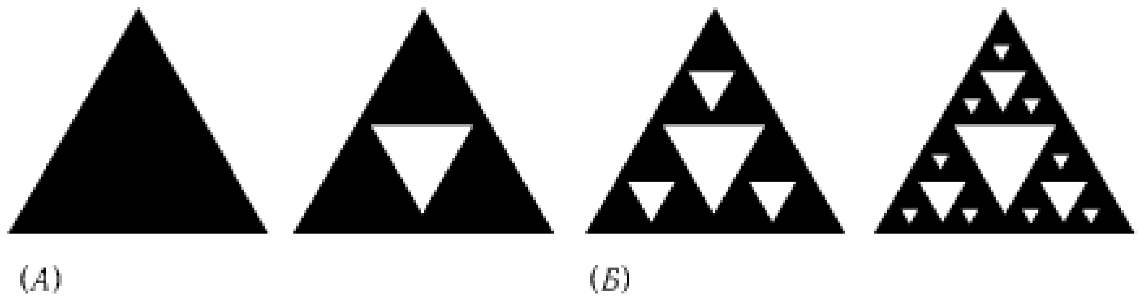


Треугольник Серпинского

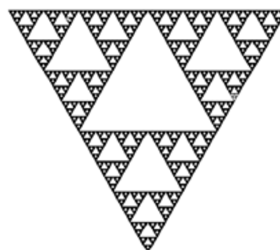
В 1915 году польский математик Вацлав Серпинский придумал занимательный объект, известный как решето Серпинского. Этот треугольник один из самых ранних известных примеров фракталов.

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.

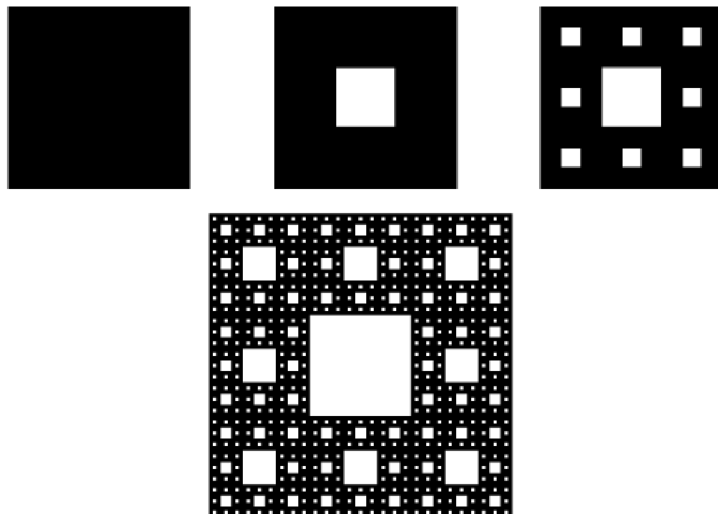
Генератор для решета Серпинского:



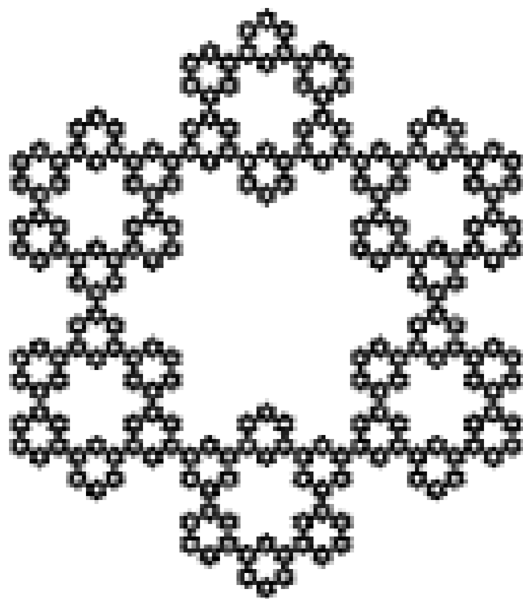
Другой вариант:



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.

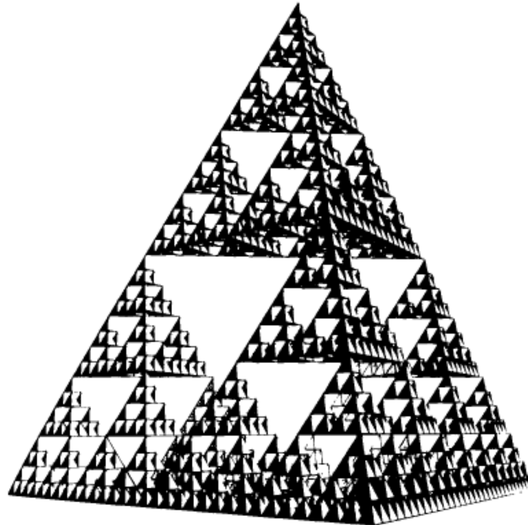


Шестиугольник Серпинского:



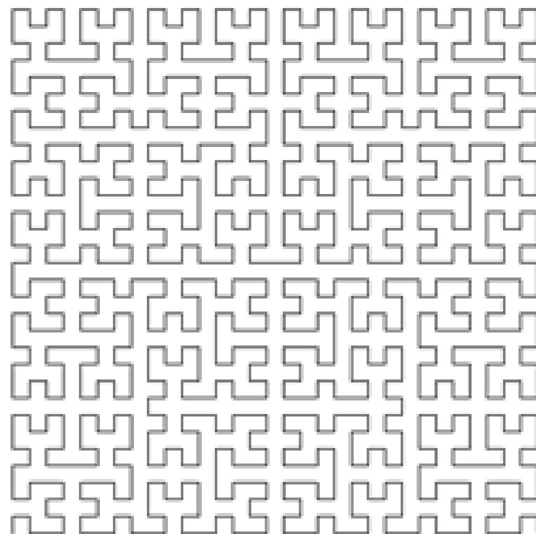
Трехмерное обобщение ковра Серпинского

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.

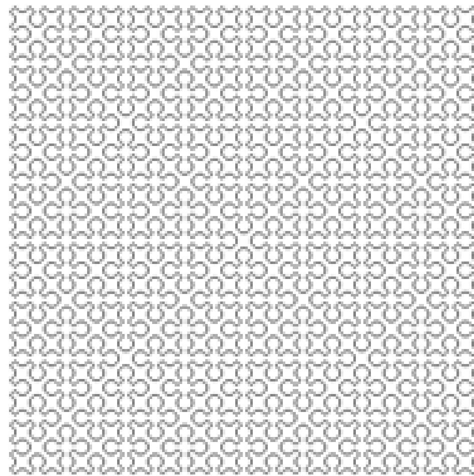
До работы Гильберта математики предполагали, что сделать это невозможно.

Разумеется, повторить алгоритм бесконечное число раз нельзя; говоря о бесконечностях, следует оперировать пределами, ε – окрестностями и прочими понятиями математического анализа.

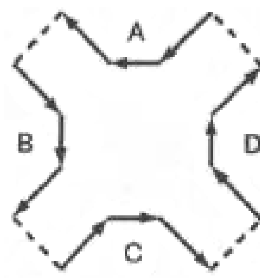
Кривые Серпинского

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.



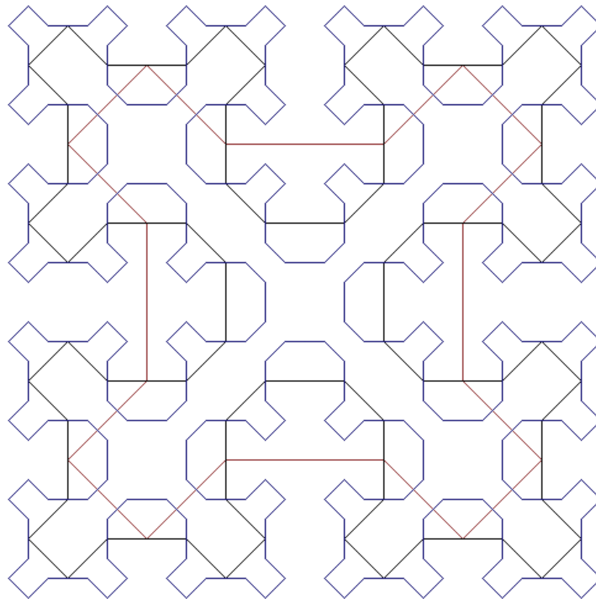
Каждая из четырех основных кривых составлена из линий диагонального сегмента, вертикального или горизонтального и еще одного диагонального сегмента.

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:

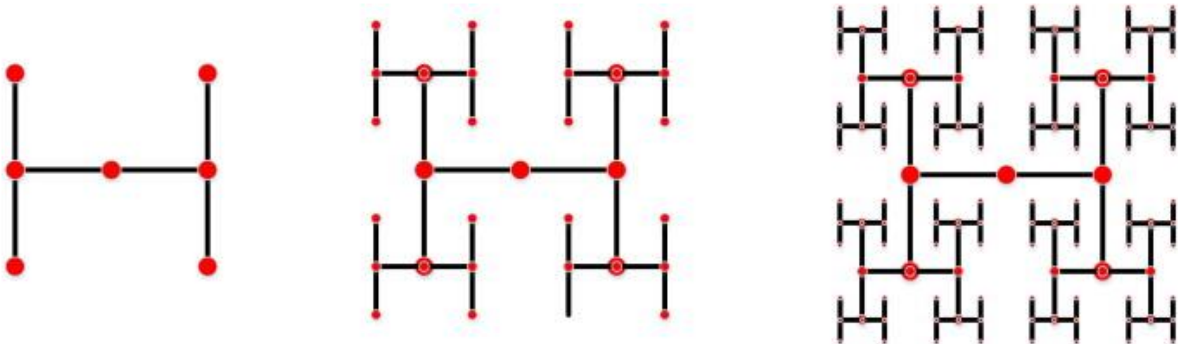
A: A` B` D` A`
B: B` C` A` B`
C: C` D` B` C`
D: D` A` C` D`

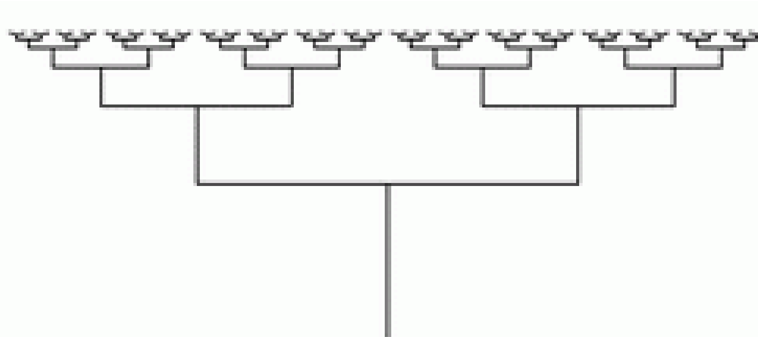
где A`, B`, C`, D` - кривые предыдущего порядка.

На рисунке приведены примеры кривых Серпинского.



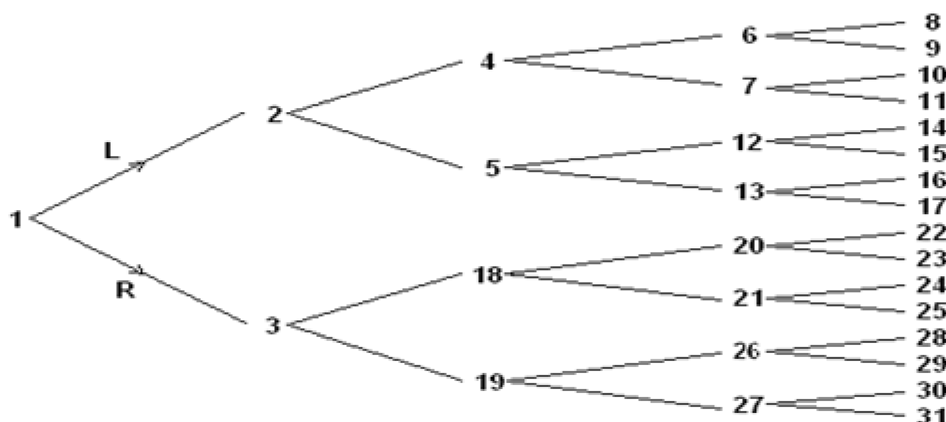
H-фрактал





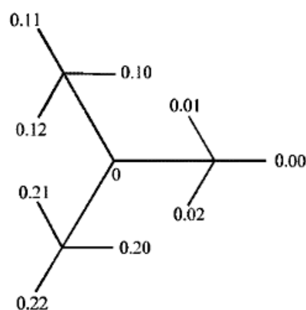
Н-фрактал относится к так называемым «дендритам», от греческого «dendron» – дерево.

Дерево вызовов рекурсии - двоичное дерево

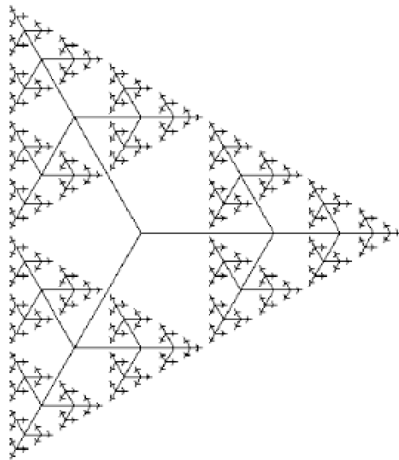


Фракталы и системы счисления

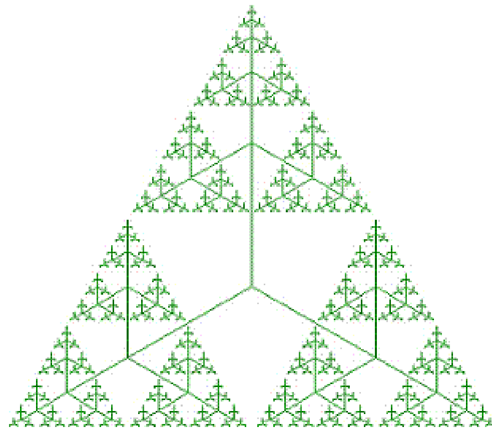
Рассмотрим дерево – дендрит, основанного на троичной системе счисления. Из одной точки под углом 120° друг к другу выходят три главные ветви. Каждый из трех концов сам является точкой, из которой выходят три более мелкие ветви, и т. д. Направление вправо мы помечаем «0». Направление влево-вверх – «1», влево-вниз – «2».



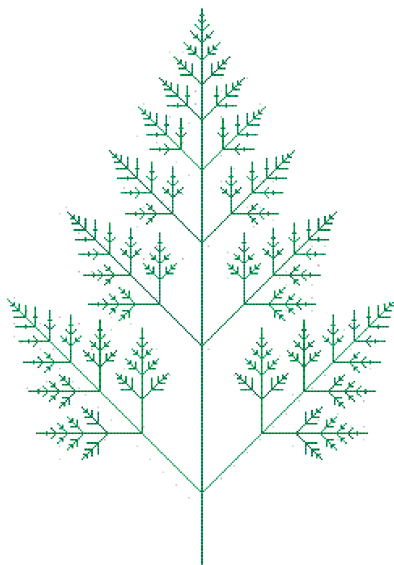
Используя данный алгоритм можно построить следующее дерево.



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

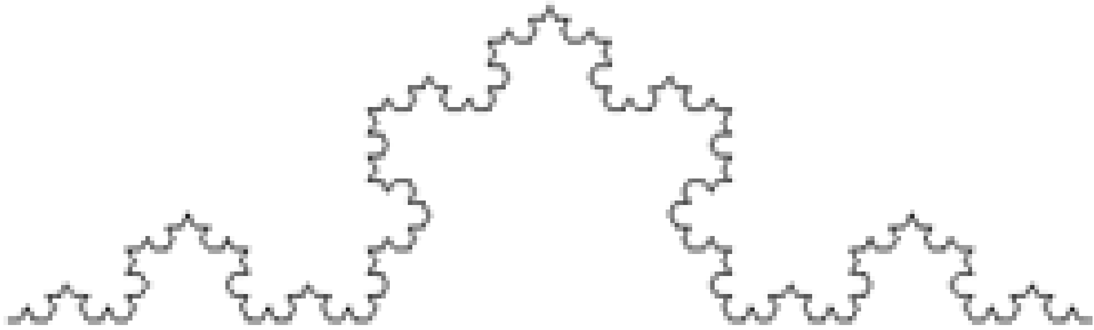
$$\begin{cases} X' = A_1 \cdot X \cdot X + B_1 \cdot X \cdot Y + C_1 \cdot Y \cdot Y + D_1 \cdot X + E_1 \cdot Y + F_1, \\ Y' = A_2 \cdot X \cdot X + B_2 \cdot X \cdot Y + C_2 \cdot Y \cdot Y + D_2 \cdot X + E_2 \cdot Y + F_2 \end{cases}$$

и другие.

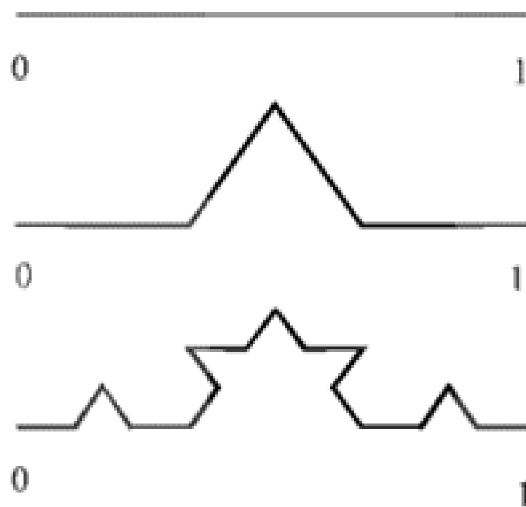
Триадная кривая Коха, дракон Хартера-Хейтуэя и лист папоротника Барнсли задаются аффинными преобразованиями и отличаются лишь матрицами преобразований.

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:

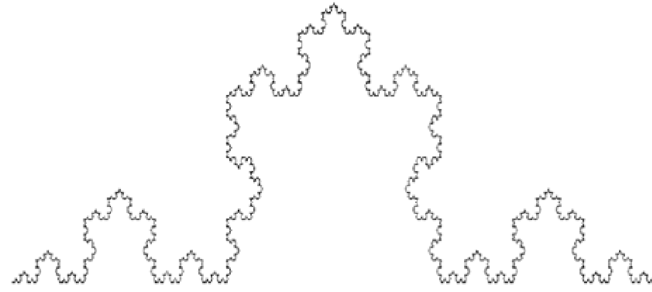
$$\begin{cases} X' = 0.333 \cdot X + 13.333 \\ Y' = 0.333 \cdot Y + 200 \end{cases}$$

$$\begin{cases} X' = 0.333 \cdot X + 413.333 \\ Y' = 0.333 \cdot Y + 200 \end{cases}$$

$$\begin{cases} X' = 0.167 \cdot X + 0.289 \cdot Y + 130 \\ Y' = -0.289 \cdot X + 0.167 \cdot Y + 256 \end{cases}$$

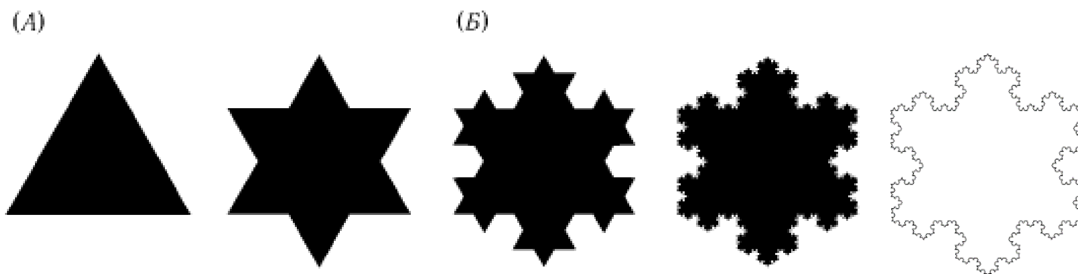
$$\begin{cases} X' = 0.167 \cdot X - 0.289 \cdot Y + 403 \\ Y' = -0.289 \cdot X + 0.167 \cdot Y + 71 \end{cases}$$

Результат применения этого аффинного преобразования после десятой итерации можно увидеть далее.

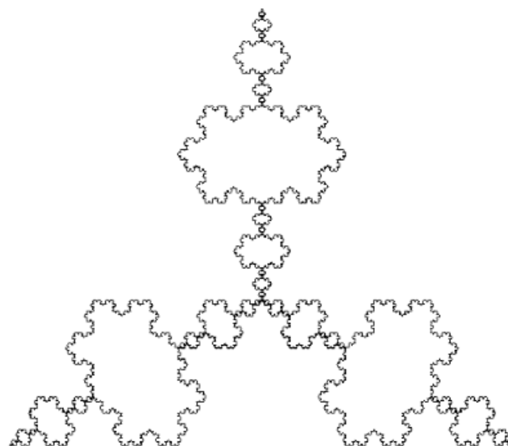


Остров Коха

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.

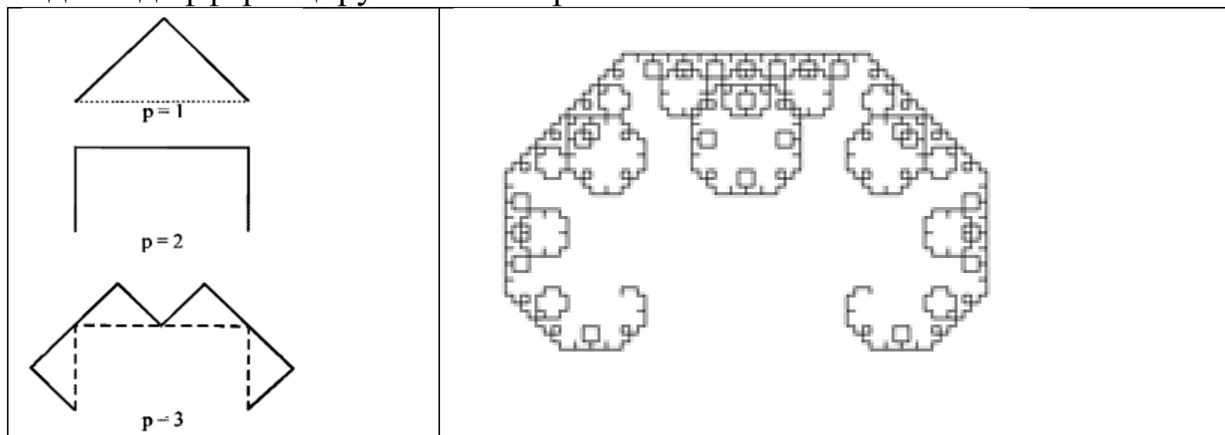


Остров Коха,
ориентированный вдоль треугольника

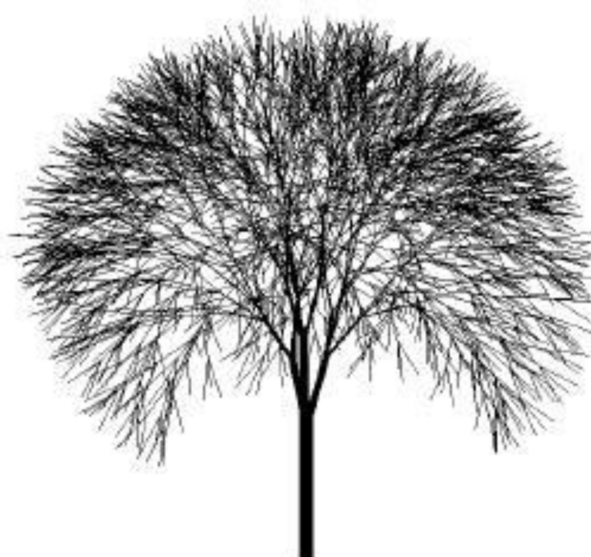


Кривая Леви

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.

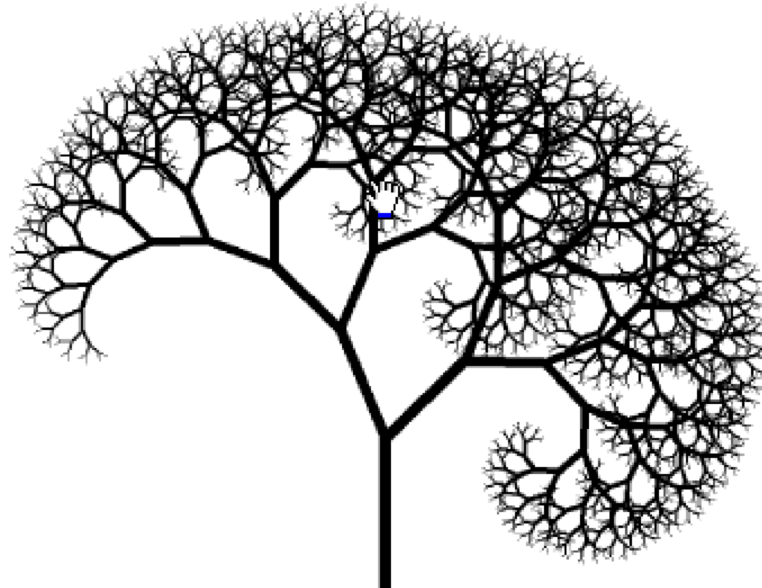
Получившиеся ветки снова рассматриваются как стволы и т. д.

Стохастические фракталы

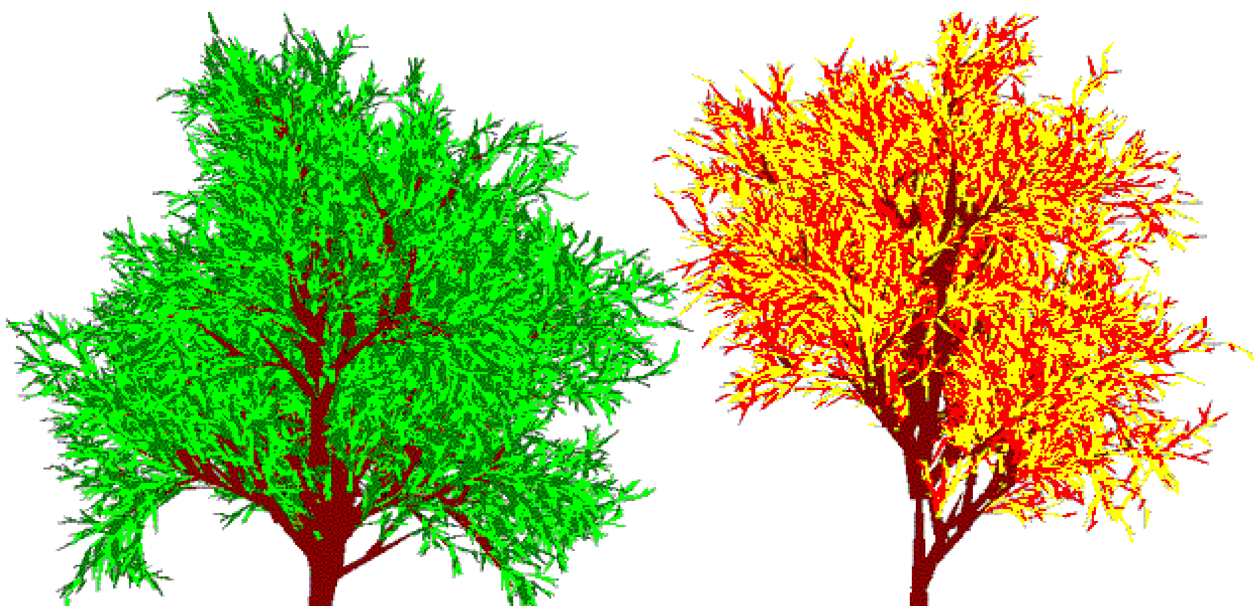
Стохастические фракталы получаются в том случае, когда в итерационном процессе построения рисунка случайно менять какие-то параметры.

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

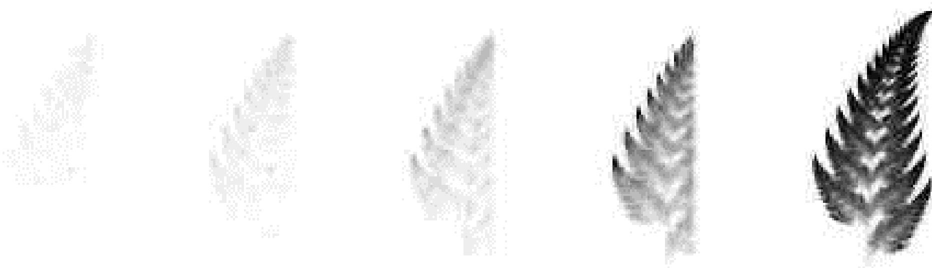
a	***b***	***c***	***d***	***e***	***f***	***p***
0	0	0	0.16	0	0	0.01
0.85	0.04	-0.04	0.85	0	1.6	0.85
0.2	-0.26	0.23	0.22	0	1.6	0.07
-0.15	0.28	0.26	0.24	0	0.44	0.07

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.

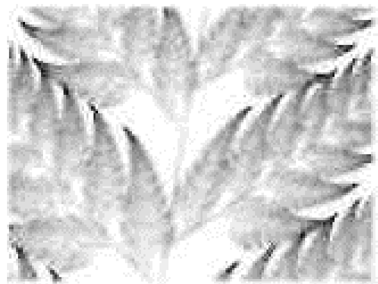
Результат действия этой системы функций на некоторую начальную точку для разного числа итераций приведен на рисунке.



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;
```

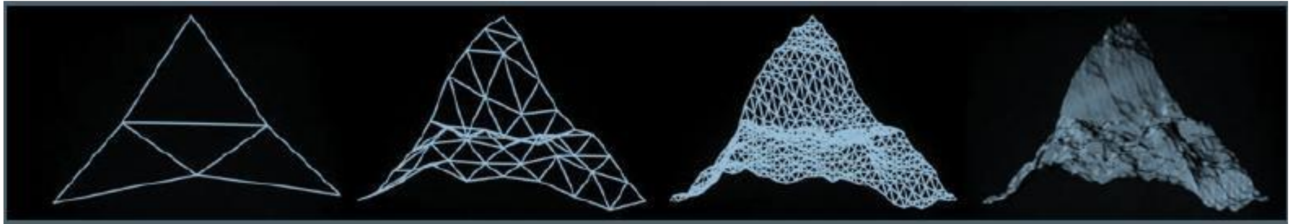
```
procedure Draw;
const
      Iteration = 50000;
var
    t, x, y, p           : Real;
    k                    : Longint;
    mid_x, mid_y, radius : Integer;
begin
 mid_x := GetMaxX div 2;
 mid_y := GetMaxY;
 radius := Trunc(0.1 * mid_y);
 Randomize;
 x := 1.0;
 y := 0.0;
 for k:=1  to iteration  do
   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.

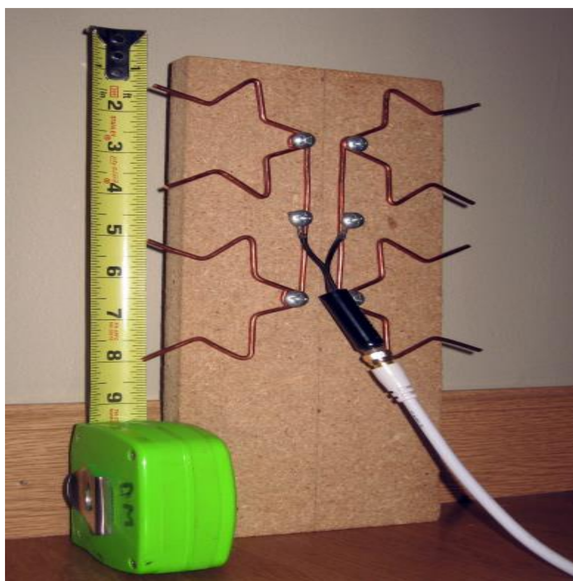
Всего через несколько лет свои наработки Лорен Карпентер смог применить в куда более масштабном проекте. Аниматор создал на их основе двухминутный демонстрационный ролик Vol Libre, который был показан на Siggraph в 1980 году. Это видео потрясло всех, кто его видел, и Лоурен получил приглашение от Lucasfilm.

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.



Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 7
ВЕРСИИ РЕАЛИЗАЦИИ И СРЕДЫ РАЗРАБОТКИ PASCAL. FREE PASCAL

Содержание темы

- 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.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

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.

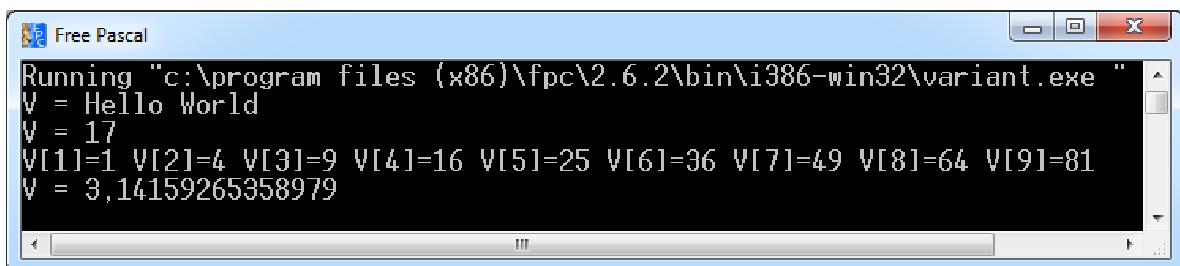
Данные языка Free Pascal

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.

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`	$1.5 \cdot 10^{-45}..3.4 \cdot 10^{38}$	7-8	4
`Double`	$5.0 \cdot 10^{-324}..1.7 \cdot 10^{308}$	15-16	8
`Extended`	$1.9 \cdot 10^{-4932}..1.1 \cdot 10^{4932}$	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.

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.

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.

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 С++. 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

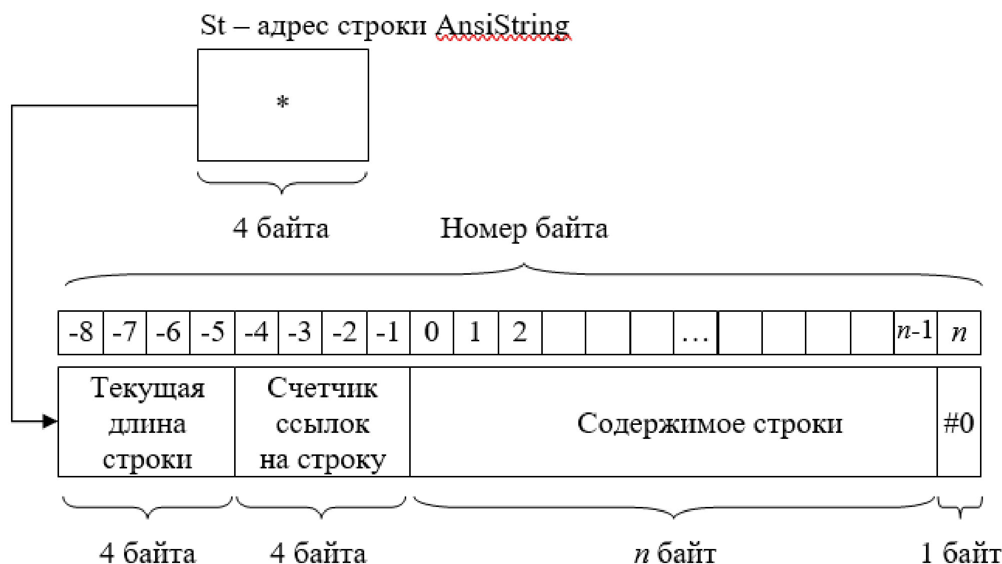
Строки типа AnsiString являются строковым типом данных неограниченной длины, где каждый символ представлен однобайтовым ANSI кодом. Объявляются при использовании String без указания длины строки при включенной директиве компилятора {$H+} или при помощи предопределенного типа AnsiString. Например, в приведенной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

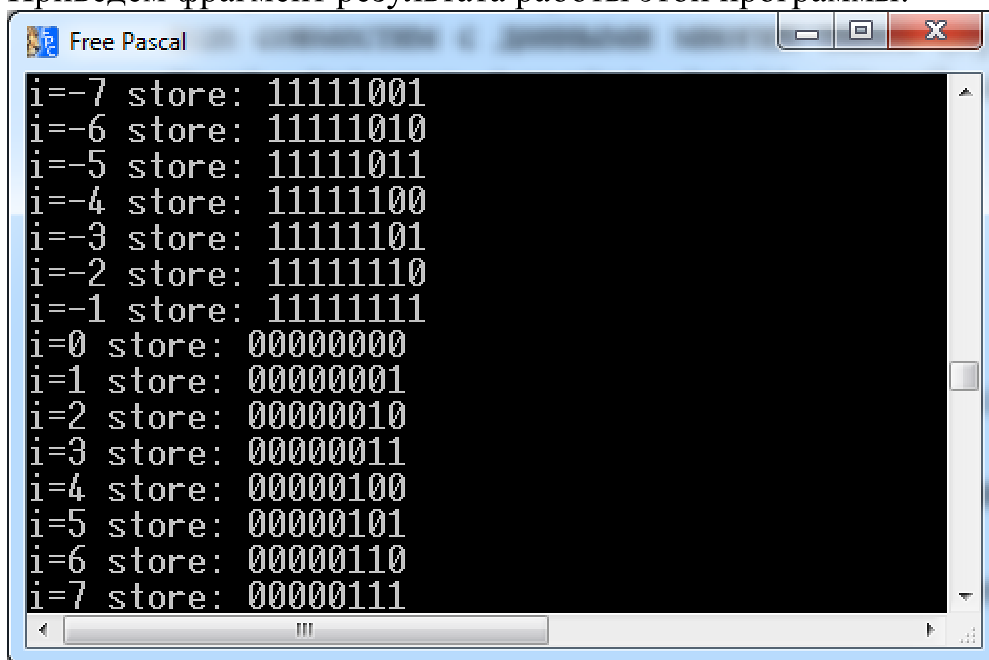
LongInt, Int64, QWord), а результат, возвращаемый функцией, имеет тип ShortSTRING.

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.

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.

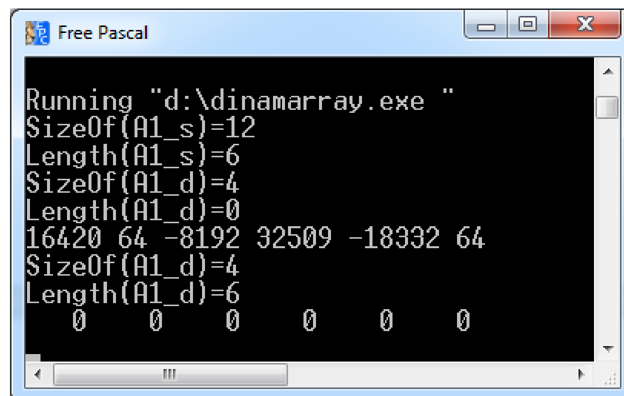
Функция Length выдает фактическое значение, равное текущему количеству элементов динамического массива. Если ещё не выполнялась процедура SetLength, то значение Length(Al_d) равно 0.

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}.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 8
ПРОЦЕДУРЫ И ФУНКЦИИ FREE PASCAL

Содержание темы

- Процедуры и функции.
- Параметры:
  - параметры подпрограмм по умолчанию,
  - расширенный вызов функций.
- Перегрузка функций и процедур.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

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

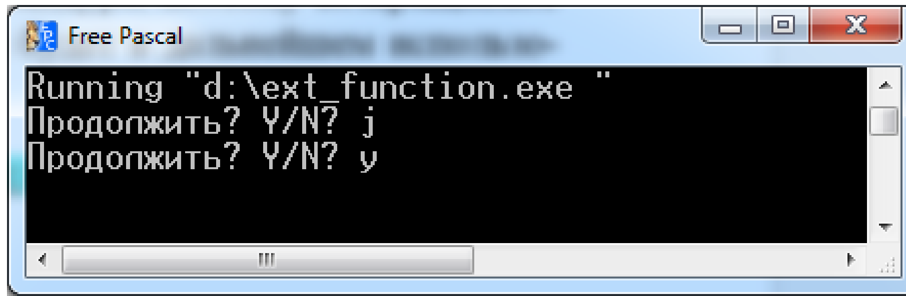
Расширенный вызов функций

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;
```

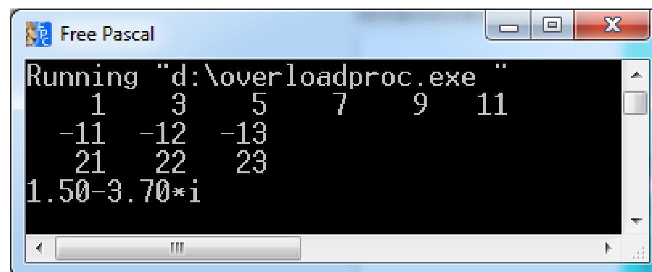
```
 Mat[0,2]:=-13;
 Mat[1,0]:=21;
 Mat[1,1]:=22;
 Mat[1,2]:=23;
 CompNum.Re:=1.5;
 CompNum.Im:=-3.7;
 Print(Vec);
 Print(Mat);
 Print(CompNum);
end.
```

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.



Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 9
ОСОБЕННОСТИ РАБОТЫ С ФАЙЛАМИ, ЗВУКОМ, ЭКРАНОМ

Содержание темы

- Управление файлами в стиле Windows.
- Стандартные модули Free Pascal.
- Особенности работы со звуком в Free Pascal.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

УПРАВЛЕНИЕ ФАЙЛАМИ В СТИЛЕ WINDOWS

В модуле sysutils 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:

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.

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:

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.

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

Имя модуля	Назначение
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
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.

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);
```

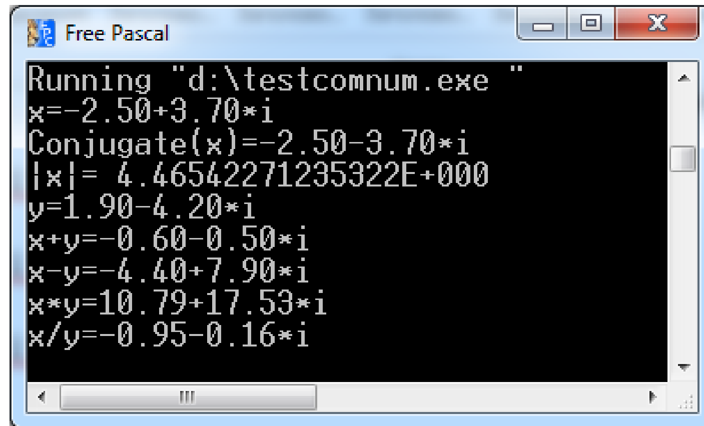
```
 begin
 if (x.Re<>0) And (x.Im<>0) then
   Write(x.Re:0:n,IfThen(x.Im>0,'+','-'),
         Abs(x.Im):0:n,'*i')
   else
    if (x.Im=0) And (x.Re=0) then Write (0)
       else
        if x.Re=0 then Write (x.Im:0:n,'*i')
           else Write(x.Re:0:n);
 end;
end.
```

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.

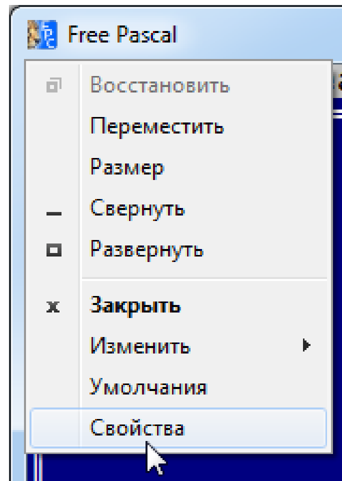
ОСОБЕННОСТИ РАБОТЫ С ЭКРАНОМ В ТЕКСТОВОМ РЕЖИМЕ В FREE PASCAL

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.

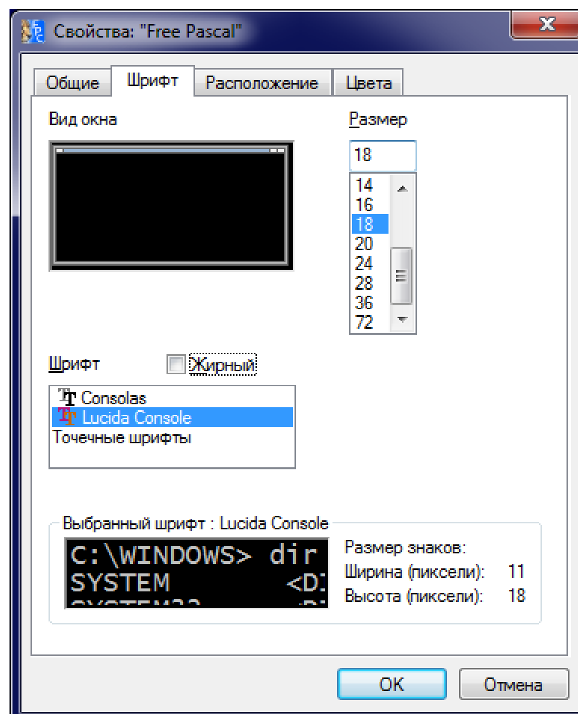
В настоящее время при работе в Windows текстовый режим поддерживается программными средствами. Поскольку при работе в

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.

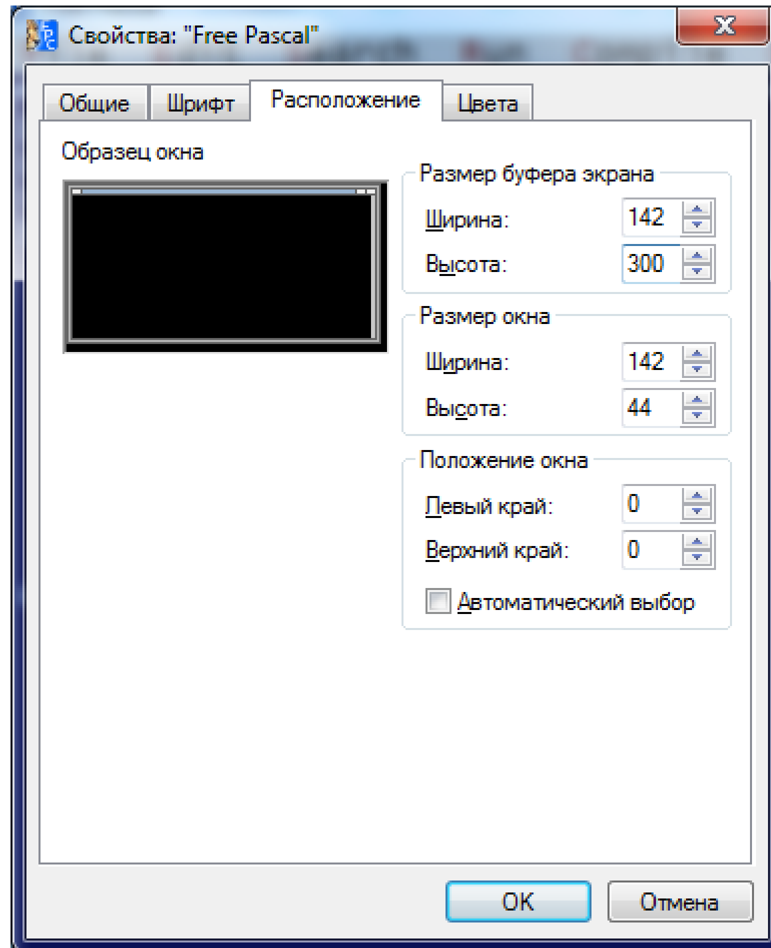


В открывшемся диалоговом окне на вкладке **Шрифт** можно выбрать одну из трех гарнитур шрифта (Consolas, Lucida Console, Точечные шрифты) и установить размер шрифта.



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

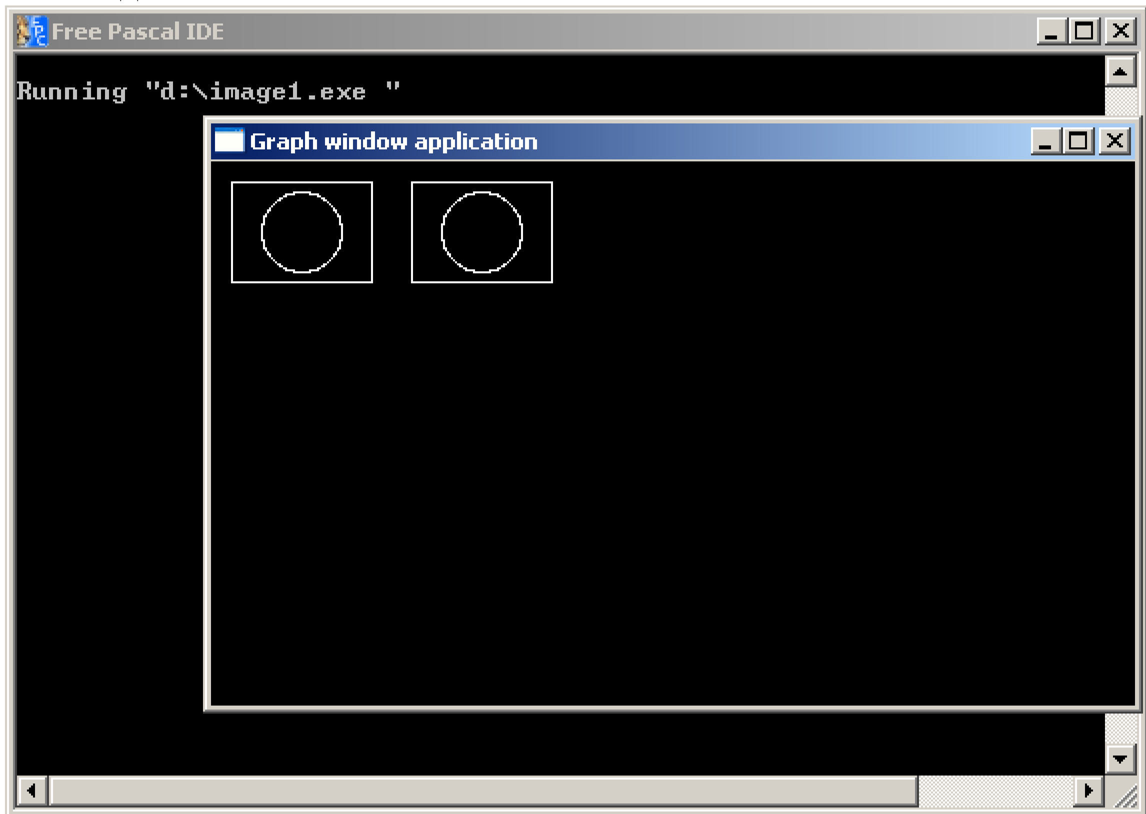
аппаратному обеспечению через OpenGL и Direct3D. Рассмотрение работы с `sdlGraph` выходит за рамки данного пособия.

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.

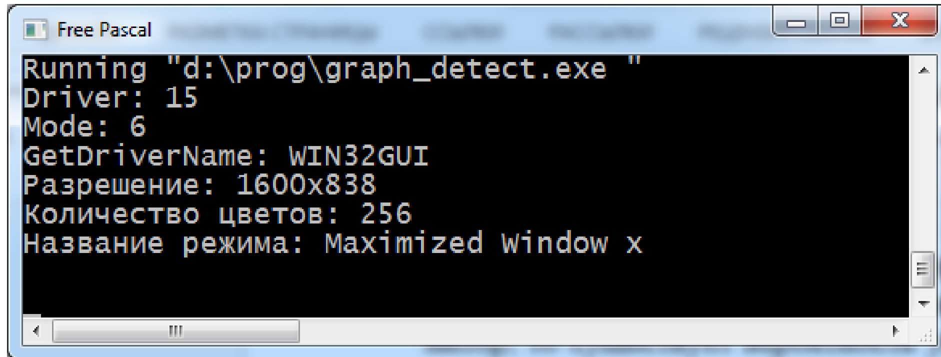
Для автоматического выбора `Driver` и `Mode`, обеспечивающих максимальное разрешение экрана и максимальное количество цветов при данном разрешении можно использовать процедуру DetectGraph.

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

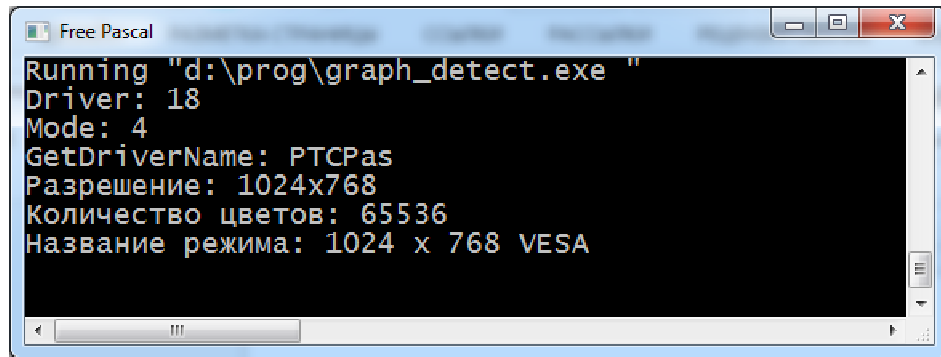
```
program Graph_detect;
uses Graph;
var
    Driver: SmallInt;
    Mode: SmallInt;
begin
  DetectGraph(Driver, Mode);
  InitGraph(Driver, Mode, '');
  Writeln('Driver: ',Driver);
  Writeln('Mode: ', Mode);
  Writeln('GetDriverName: ', GetDriverName);
  Writeln('Разрешение: ', GetMaxX+1,'x',GetMaxY+1);
```

```
  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.

```
Free Pascal

Running "d:\prog\gmodeinfo.exe "
----------------------------------------------------------------------------------
Driver Moda Название режима     Разрешение Кол. цветов Кол. страниц
----------------------------------------------------------------------------------

   9      2  640 x 480 x 16 Win      640x480       16          1
   9      0  640 x 200 x 16 Win      640x200       16          1
   9      1  640 x 350 x 16 Win      640x350       16          1
  10    256  640 x 400 x 256 Wi      640x400      256          1
  10    257  640 x 480 x 256 Wi      640x480      256          1
  10    258  800 x 600 x 16 Win      800x600       16          1
  10    259  800 x 600 x 256 Wi      800x600      256          1
  10    260  1024 x 768 x 16 Wi     1024x768       16          1
  10    261  1024 x 768 x 256 W     1024x769      256          1
  10    240  Largest Window x 1     1584x862       16          1
  10    241  Largest Window x 2     1584x862      256          1
  10    245  Maximized Window x     1600x838       16          1
  10    246  Maximized Window x     1600x838      256          1
----------------------------------------------------------------------------------
```

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:

```
Free Pascal
Running "d:\prog\gmodeinfo.exe "
-----------------------------------------------------------------------
Driver Moda Название режима     Разрешение Кол. цветов Кол. страниц
-----------------------------------------------------------------------

   1     0    320 x 200 CGA C0     320x200          4          1
   1     1    320 x 200 CGA C1     320x200          4          1
   1     2    320 x 200 CGA C2     320x200          4          1
   1     3    320 x 200 CGA C3     320x200          4          1
   1     4       640 x 200 CGA     640x200          2          1
   2     0    320 x 200 CGA C0     320x200          4          1
   2     1    320 x 200 CGA C1     320x200          4          1
   2     2    320 x 200 CGA C2     320x200          4          1
   2     3    320 x 200 CGA C3     320x200          4          1
   2     4       640 x 200 CGA     640x200          2          1
   2     5      640 x 480 MCGA     640x480          2          1
   7     0  720 x 348 HERCULES     720x348          2          2
   3     0       640 x 200 EGA     640x200         16          3
   3     1       640 x 350 EGA     640x350         16          2
   9     0       640 x 200 EGA     640x200         16          3
   9     1       640 x 350 EGA     640x350         16          2
   9     2       640 x 480 VGA     640x480         16          1
   6     0       320 x 200 VGA     320x200        256          1
   6     1     320 x 200 ModeX     320x200        256          4
  10   256      640 x 400 VESA     640x400        256          2
  10   257      640 x 480 VESA     640x480        256          2
  10   269      320 x 200 VESA     320x200      32768          2
  10   272      640 x 480 VESA     640x480      32768          2
  10   270      320 x 200 VESA     320x200      65536          2
  10   273      640 x 480 VESA     640x480      65536          2
  10   258      800 x 600 VESA     800x600         16          2
  10   259      800 x 600 VESA     800x600        256          2
  10   275      800 x 600 VESA     800x600      32768          2
  10   276      800 x 600 VESA     800x600      65536          2
  10   260     1024 x 768 VESA    1024x768         16          2
  10   261     1024 x 768 VESA    1024x768        256          2
  10   278     1024 x 768 VESA    1024x768      32768          2
  10   279     1024 x 768 VESA    1024x768      65536          2
-----------------------------------------------------------------------
```

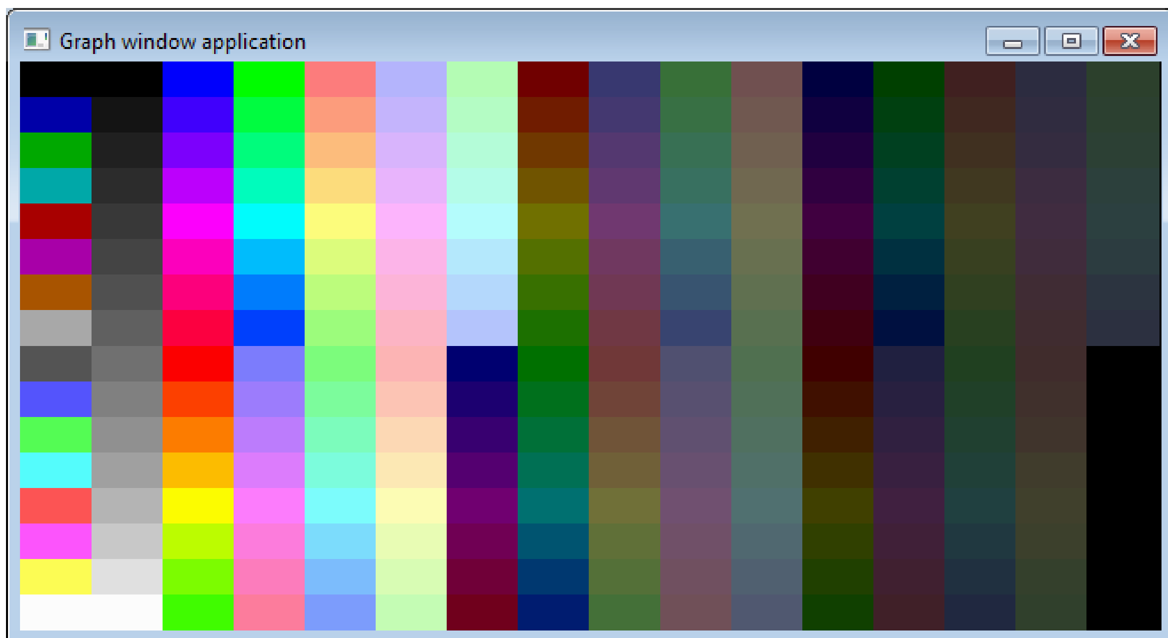
При использовании модуля `ptcGraph` максимальное разрешение графического окна получилось меньше, всего 1024×768, но увеличилось количество цветов до 65 536 и количество видеостраниц до 2.

Особенности управления цветом в Free Pascal

При использовании модуля `Graph` становятся доступными 256 цветов, а при использовании `ptcGraph` – 65 536. 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 65 536 цветов, но размер каждого прямоугольника будет очень маленьким, на тестируемом компьютере всего 4×3 пиксела.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

СПИСОК ЛИТЕРАТУРЫ

Расолько, Г. А. Теория и практика программирования на Pascal / Г. А. Расолько, Ю. А. Кремень. – Минск : Выш. шк., 2015.

Расолька, Г. А. Метады праграміравання. Алгарытмы апрацоўкі данных / Г. А. Расолько, Ю. А. Кремень. – Мінск : БДУ, 2008.

Расолько, Г. А. Электронный учебно-методический комплекс по учебной дисциплине «Методы программирования и информатика» для специальностей 1-31 03 01 «Математика (по направлениям)», 1-31 03 01-02 «Математика (научно-педагогическая деятельность)» / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень. – Минск : БГУ, 2014. http://elib.bsu.by/handle/123456789/97905

Расолько, Г. А. Методы программирования. Free Pascal: основы работы : учеб. материалы для студентов мех.-мат. фак. спец. 1-31 03 01-02 «Математика (научно-педагогическсная деятельность)» / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень. – Минск : БГУ, 2014. http://elib.bsu.by/handle/123456789/94223

Расолько, Г. А. Методы программирования. Free Pascal: работа с внешними устройствами : учеб. материалы для студентов мех.-мат. фак. спец. 1-31 03 01-02 «Математика (научно-педагогическая деятельность)» / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень. – Минск : БГУ, 2014. http://elib.bsu.by/handle/123456789/94228