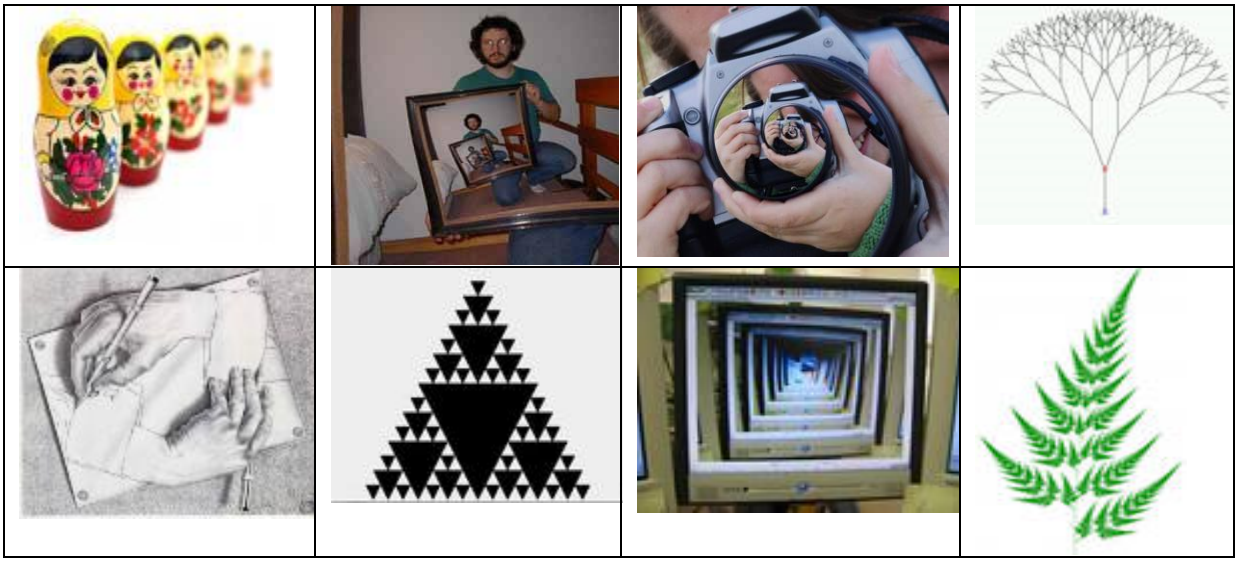
Рекурсия. От простого к сложному.

На протяжении многих лет при изучении в школе темы «Рекурсия», я сталкивалась с тем, что для многих учеников это был «камень преткновения». Не смотря на то, что достаточно много материала и в учебниках и в интернете на эту тему, я все-таки решила создать свой конспект. В нем я старалась изложить понятие рекурсии в более доступной форме, переходя от простых примеров рекурсии (картинки, детские стишки, игра «Ханойская башня») к классическим алгоритмам (факториал, числа Фибоначчи и т.д.) и заканчивая рекурсивной графикой.
Наблюдая за учениками на уроках, я заметила, что «Рекурсия» перестала быть для них «страшным» словом.
Текст конспекта представляю вашему вниманию.

“Итерация – от человека, а рекурсия – от бога”

Рекурсивным называется объект, частично состоящий или определяемый с помощью самого себя. Рекурсия встречается не только в математике, но и в повседневной жизни.



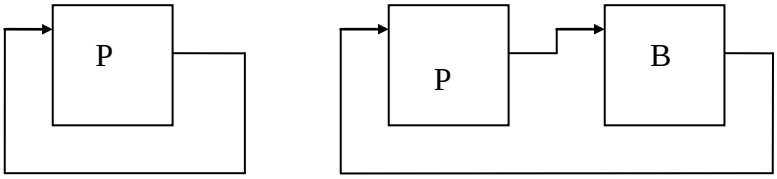
В жизни Вам не раз приходилось сталкиваться с рекурсией. Вспомните хотя бы стихотворение "У попа была собака" или “10 негритят пошли купаться в море…”

Или то, как, сидя в поезде, вы ловили свое отражение в зеркале, которое отражалось в зеркале напротив, которое отражалось в зеркале напротив…

Слово «рекурсия» **происходит от латинского слова** «recursio» **- возвращение.**

Определение: ***Если подпрограмма обращается сама к себе как к подпрограмме непосредственно или через цепочку подпрограмм, то это называется*** *рекурсией*. ***А такие подпрограммы называются*** *рекурсивными*.

Если некоторая процедура P содержит явную ссылку на саму себя, то ее называют *прямо рекурсивной,* **если же P ссылается на другую процедуру B, содержащую ссылку на P, то P называют** *косвенно рекурсивной*.



Прямая **Косвенная**

Рассмотрим сначала прямую рекурсию.

Сразу же после определения может возникнуть вопрос: «А не зациклится ли, то есть, не будет ли бесконечно выполняться такая рекурсивная программа?» И, действительно, опасность зацикливания вполне реальна. Например, стихотворение "У попа была собака" может повторяться до бесконечности. А со считалкой про 10 негритят дело обстоит иначе.

Вот текст стихотворения:
«10 негритят пошли купаться в море,
10 негритят резвились на просторе,
Один из них пропал и вот вам результат:

9 негритят пошли купаться в море,
9 негритят резвились на просторе,
Один из них пропал – и вот вам результат:
…
1 (из) негритят пошли(ел) купаться в море,
1 (из) негритят резвились(ся)на просторе,
Один из них пропал – и вот вам результат:
Нет больше негритят!»

Первые три строчки этого стихотворения повторяются 10 раз с небольшим изменением – число негритят уменьшается с каждым разом на единицу. И только когда число негритят уменьшилось до нуля, стихотворение заканчивается единственной строчкой «Нет больше негритят!».

Напишем процедуру:

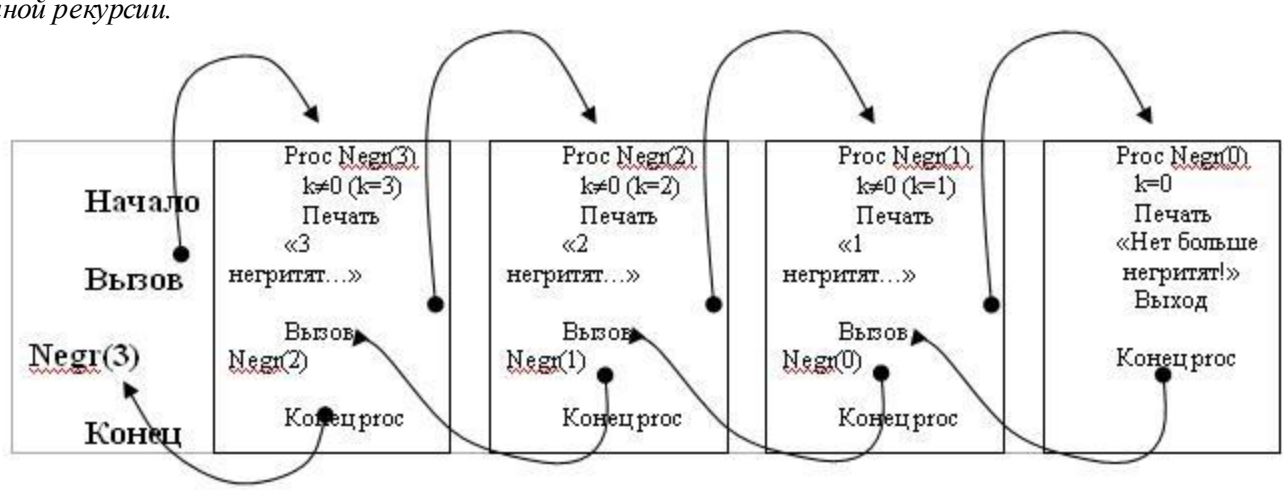
```
Procedure Negr(k: integer);   {k - число негритят, параметр процедуры}
Begin
        If k=0  {проверка, что число негритят равно нулю}
        then    Writeln('Нет больше негритят!')  {выход из рекурсии }
        else begin
                Writeln(k,' негритят пошли купаться в море,');
                Writeln(k,' негритят резвились на просторе,');
                Writeln('Один из них пропал - и вот вам результат:');
                Negr(k-1);  {Вызов процедуры с уменьшенным на 1 параметром}
                end
End;
Begin
   Negr(10).
End.
```

Каким же образом выглядит рекурсивная процедура-подпрограмма, выполняющая эту задачу? Проиллюстрируем для k = 3.

При обращении подпрограммы к самой себе происходит то же самое, что и при обращении к любой процедуре или функции: в стек записывается адрес возврата, резервируется место под локальные переменные, происходит передача параметров, после чего управление

передается первому исполняемому оператору программы. **При повторном вызове этот процесс повторяется.**

Число копий переменных, одновременно находящихся в стековой памяти, называется *глубиной рекурсии.*



Для завершения рекурсивных вызовов в рекурсивной подпрограмме обязательно должно быть условие выхода из нее, заканчивающееся возвратом в вызывающую программу. **В данном случае это происходит при k=0.**

При завершении подпрограммы область ее локальных переменных освобождается, а управление передается оператору, следующему за рекурсивным вызовом.

Такое последовательное обращение как бы к другим экземплярам одной и той же процедуры (другим также является значение параметра) напоминает матрешку, которая раскрывается до тех пор, пока может открываться, потом же матрешек закрывают по очереди, начиная с внутренней (самой маленькой).

Из этого примера можно сделать **выводы:**

1. Рекурсивная подпрограмма должна иметь условие выхода**, чтобы не быть бесконечной. Поэтому в первую очередь надо оформлять выход из рекурсии.**
2. **Рекурсия не может иметь слишком много вложений.** Ограничение по стековой памяти**, которое не может превышать 64 Кб.**

Рекурсия достаточно широко применяется в программировании, что основано на рекурсивной природе многих математических алгоритмов.

Рекурсивным ***называется объект, частично состоящий или определяемый с помощью самого себя.***

Рекурсивные определения представляют собой мощный аппарат в математике. Например:

1. Натуральные числа:
   а) 1 есть натуральное число,
   б) число, следующее за натуральным, - есть натуральное число.
2. Деревья:
   а) 0 есть дерево ("пустое дерево"),
   б) если A1 и A2 - деревья, то построение, содержащее вершину с двумя ниже расположенными деревьями, опять дерево.
3. Функция n! "факториал" (для неотрицательных целых чисел):
   а) 0!=1,
   б) n>0: n!=n·(n-1)!

Хорошей иллюстрацией механизма рекурсии является функция для вычисления факториала натурального числа.

Факториал вычисляется следующим образом:

N! = 1 * 2 * 3 * … * (N-1) * N, *(1)*

то есть представляет собой произведение натуральных чисел от 1 до N включительно.

На эту функцию можно посмотреть и под другим углом зрения. Внимательнее изучив формулу *(1)*, мы можем прийти к выводу, что

N! = N * (N-1)! *(2)*

Таким образом, мы определили факториал через сам факториал! Казалось бы, такое определение совершенно бесполезно, так как неизвестное понятие определяется через опять же неизвестное понятие, и мы приходим к бесконечному циклу. Однако это впечатление обманчиво, потому что на самом деле факториал N определяется через факториал (N-1), то есть нахождение неизвестной функции сводится к вычислению той же неизвестной функции от другого аргумента, на единицу меньшего, чем исходный. Таким образом, есть надежда, что каждая следующая задача будет решаться чуть легче предыдущей.

Окончательно разорвать замкнутый круг мы сможем, если дополним рекурсивное определение *(2)* еще одним, которое служит чем-то вроде тупика, предназначенного для остановки процесса:

0! = 1 *(3)*

Правила *(2)* и *(3)* совместно позволяют вычислить значение факториала от любого аргумента. Попробуем для примера вычислить значение 5!, несколько раз применив правило *(2)* и однократно – правило *(3)*:

5! = 5 * 4! = 5 * 4 * 3! = 5 * 4 * 3 * 2! = 5 * 4 * 3 * 2 * 1! = 5 * 4 * 3 * 2 * 1

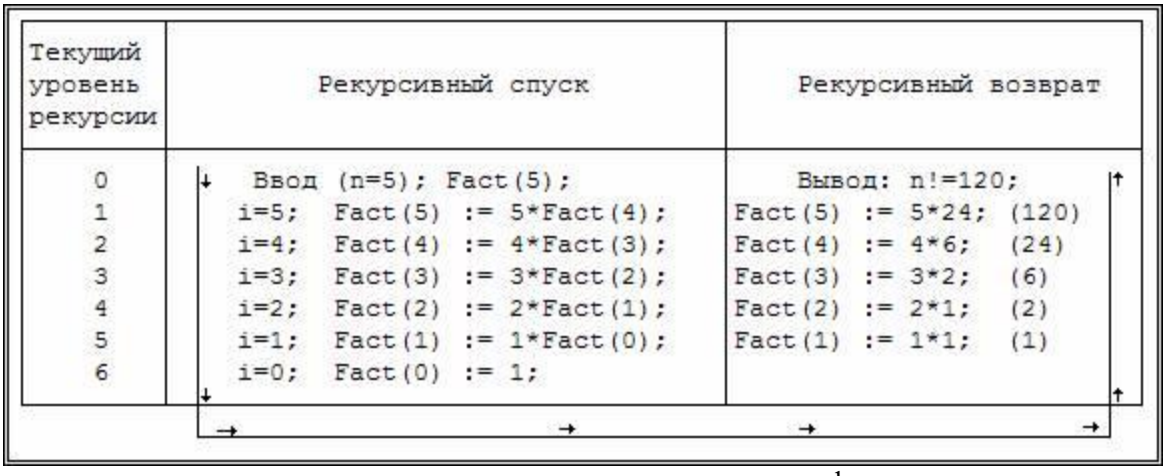
Мы получили результат, который совпадает с определением *(1)* с точностью до перестановки сомножителей.

Как же выглядит рекурсия на практике?

```
Function fakt (n: integer): integer;
begin
if n=0 then fakt:=1
else fakt:= n*fakt(n-1);
end;
```

Рассмотрим работу функции при n=5.

Полезно проанализировать рекурсивные алгоритмы с точки зрения последовательности их выполнения. Под последовательностью выполненного рекурсивного алгоритма будем понимать последовательность вызовов алгоритма с различными значениями аргументов и очередью определения результатов.



Из рисунка видно, что рекурсия связана с многократными вызовами функции, а это несколько менее эффективно при выполнении по сравнению с использованием цикла. К тому же требует больше памяти. Однако рекурсивные версии программ, как правило, гораздо короче и нагляднее, не требуют наличия цикла и параметра цикла.

Резюме: Итерация требует меньше места в памяти и машинного времени, чем рекурсия, которой необходимы затраты на управление стеком**. Итак, если для некоторой задачи возможны два**

решения, предпочтение следует отдать итерации. Правда, для многих задач рекурсивная формулировка совершенно прозрачна, в то время как построение итерации оказывается весьма сложным делом. Короче, какой алгоритм выбрать - решать вам.

Задача1**. Числа Фибоначчи.**

Еще один из наиболее часто используемых примеров применения рекурсии - это числа Фибоначчи. Они определяются следующим образом:

x[1]=x[2]=1

x[n]=x[n-1]+x[n-2] при n > 2

Каждый элемент ряда Фибоначчи является суммой двух предшествующих элементов, т.е.

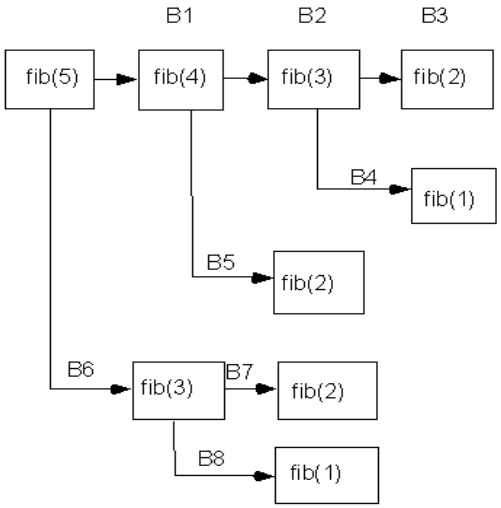
1 1 2 3 5 8 13 21 34 55 …

Решим задачу двумя способами: через рекурсию и итерацию. Определите, сколько раз выполняется операция сложения при =45.

```
program fibonacci;
uses crt;
var n,k:integer;
      function fibit(n:integer):longint;
      var a,b,c,i:integer;
      begin
          a := 1; b := 1;
          if (n=1) or (n=2)
      then fibit :=1
      else begin
                for i:= 3 to n do
                begin c :=a+b; a := b; b :=c; end;
                fibit :=c;
                   end;
            end;
           function fib(n:integer):longint;
           begin
           if (n=1) or (n=2) then fib:=1 else fib:=fib(n-1)+fib(n-2);
           k:=K+1; {writeln(k); }
           end;
      begin
          clrscr;  k:=0;
          write('n = ');
          readln(n);
          writeln('Итеративно:',fibit(n):5);
          writeln('Рекурсивно:',fib(n));
          write ('Глубина рекурсии:',k);
      end.
```

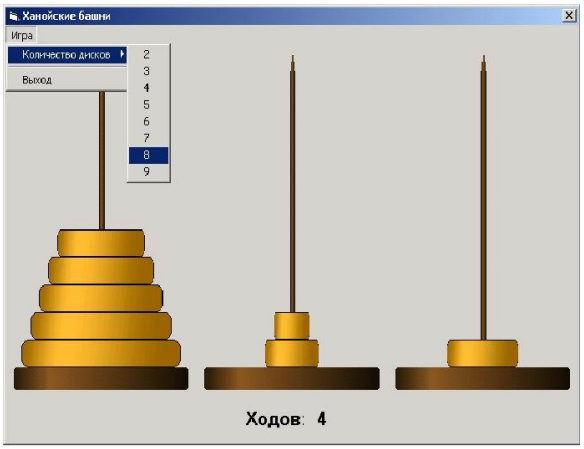
Схема нахождения 5-го члена ряда Фибоначчи изображена на рисунке ниже.

Чтобы определить значение 5-го элемента Фибоначчи, для этого необходимо определить значения fib(2), fib (1), fib (3), fib (2). Из схемы видно также, что в рассматриваемом случае значения fib (1), fib (3), fib (2) определяются дважды. При нахождении члена последовательности с большим номером число повторных вычислений значительно увеличивается. В результате при определения значения fib (17) компьютер выполнит свыше 1000, значения fib (31) свыше 1000000, значения fib (45) свыше 1000000000 операций сложения. В тоже время при использовании не рекурсивного алгоритма для вычисления 45-го члена потребуется всего 43 операции сложения.



Это позволяет сделать вывод о неэффективности использования рекурсии для решения рассматриваемой задачи.

Задача 2. **Ханойские башни. (Задачу и рассказ придумал французский математик Люка в 1883 году.)**



В центре мира в землю вбиты 3 алмазных шпиля. На одном из них 64 золотых диска убывающих радиусов (самый большой – нижний). Буддийские монахи день и ночь переносят диски с одного шпиля на другой.

Правила: переносить только по одному диску и нельзя больший класть на меньший.

Когда все диски перенесут, наступит конец света.

Обозначим для определенности шпили (или стержни) как A, B, C (или 1, 2, 3). По условию задачи надо перенести N дисков с A на B, используя промежуточный стержень C. Если N=1, то все просто (перекладываем диск с A на B) и заканчиваем процедуру переноса. Иначе (если N>1), переложим сначала N-1 дисков с A на C (используя B в качестве промежуточного стержня), затем перекладываем 1 диск (самый большой) с A на B и, наконец, переносим N-1 диск с C на B (используя стержень A).

Используя метод математической индукции и вышеуказанный алгоритм легко доказать, что необходимо совершить $2^N - 1$ действий. Если тратить на каждое действие по одной секунде, то посчитайте, сколько (миллиардов) лет понадобится легендарным жрецам, чтобы исполнить свою работу.

Предположим, с первого столба A надо перенести на третий C n дисков. Диски пронумерованы в порядке возрастания их диаметров. Предположим, что мы умеем переносить n-1 дисков. В этом случае n дисков перенесем посредством следующих шагов:

1. верхние n-1 дисков перенесем с первого на второй, пользуясь свободным третьим столбом;
2. последний диск наденем на третий столб;
3. n-1 дисков перенесем на третий, пользуясь свободным первым столбом.

Аналогичным образом можно перенести n-1, n-2 и т.д. дисков. Когда n=1, перенос осуществляется непосредственно с первого столба на третий.

```
Var
```

```
a,b,c: char;
n, k: integer;
Procedure Move (n:integer; a,b,c: char);
begin
if N>=1 then
begin
move(n-1,a,c,b);
Writeln(a,'-->',c,' ');  k:=k+1;
move(n-1,b,a,c);
end;
end;
BEGIN k:=0;
write('Введите количество колец'); readln (n);
move (n,'1','2','3');
writeln('Количество переносов=',k)
end.
```

Запустите игру в режиме on-line с сайта http://www.log-in.ru/games/865/ или http://www.tehnoflot.ru/test_33/ и попробуйте решить эту задачу для n=7.

Задача 3. **Перевод десятичного числа в 8-ю систему счисления.**

Рекурсивная процедура convert переводит десятичное число z в восьмеричную систему путем деления его на 8 и выдачи остатка в обратной последовательности.

```
Program p10to8;     {Перевод десятичного числа в восьмеричное
}
var z:integer;
procedure convert(z:integer);
begin
    if z > 7 then convert(z div 8);
    write(z mod 8);
end;
begin
    writeln('Введите число:');
    readln(z);
    writeln('Десятичное чмсло:',z);
    writeln('Восьмеричное число:    ');
    convert(z);
    writeln;
end.
```

Задача 4. **У попа была собака**

Полный текст стихотворения:

У попа была собака
Он ее любил
Она съела кусок мяса
Он ее убил и на камне написал:

А дальше по кругу то же самое.

Конечно, его можно было бы проще напечатать с помощью цикла, но это не интересно. Рекурсия больше отвечает духу стихотворения. Конечный цикл рано или поздно закончится, т.е. не не выполнит задание до конца, а бесконечный подразумевает бессмысленность труда автора (попа), поскольку какой смысл трудиться, точно зная, что стихотворение никогда не будет написано до конца? С рекурсией дело хитрее. Вызывая в который раз саму себя, функция считает, что скоро будет конец, что в ближайшем вызове дело закончится полным успехом. А чтобы во время печати не пришлось ждать слишком долго, пусть программа продолжается до тех пор, пока не будет нажата какая-нибудь клавиша (**keypressed.)**

```
program pop;
  uses crt;
  procedure absaz(c:word);{ Процедура,
   печатающая один абзац }
  { В качестве параметра цвет абзаца }
  begin
    textcolor(c);
    writeln(' У попа была собака');
    writeln(' Он ее любил');
    writeln(' Она съела кусок мяса');
    writeln(' Он ее убил');
    writeln(' И на камне написал:');
    delay(500);{ Задержка перед выводом
        следующего абзаца }
    if not keypressed then absaz((c+1)mod 15+1);
       { Продолжать, пока не нажата клавиша }
  end;
  begin
    clrscr;
    absaz(1);
  end.
```

Задача 5. **Написать рекурсивную функцию вычисления x^n, где $n \geqslant 0$.**
Для решения будем использовать равенства $x^n=1$, при $n=0$, и $x^n=x \cdot x^{n-1}$, при $n>0$.

```
var  x:real; n: byte;
Function pow (x:real; n: byte):real;
begin
if n=0 then pow:=1
else pow:=x*pow(x,n-1);
end;
begin
write('Введите число x:');
readln(x);
write('Введите степень n:');
readln(n);
writeln('Результат:', pow(x,n));
end.
```

Замечания:

1. В случае переполнения стека программа завершится с соответствующим сообщением об ошибке. Проверка переполнения стека задается ключом компиляции {$S+}, который включен по умолчанию.
2. При отладке рекурсивных программ полезно отслеживать глубину рекурсии либо визуально, вставив оператор вывода в начало подпрограммы, либо с помощью типизированной константы, которая увеличивается на единицу при каждом вызове подпрограммы: Const num: word=0;
   Локальная типизированная константа, в отличии от локальной переменной, размещается в сегменте данных и сохраняет свое значение между вызовами подпрограммы.

Домашнее задание:

Задача Написать рекурсивную функцию и процедуру вычисления НОД двух чисел, используя первую модификацию алгоритма Евклида.

И напоследок, еще несколько советов**:**

1. Всегда предусматривайте выход из рекурсии. Если выхода вы не предусмотрели, ваша программа зависнет.
2. При вызове рекурсивных функций их код перемещается в стек, и локальные переменные размещаются там же, поэтому хорошо рассчитывайте максимальную глубину "самовызовов" вашей функции. И если вы получили ошибку в процессе выполнения типа "out of memory" - это из-за переполнения стека. Пересмотрите свой алгоритм: наиболее вероятно, что рекурсия заходит слишком глубоко (или бесконечно глубоко, если неправильно реализован из нее выход).
3. Рекурсия - это не способ быстрого решения. Сначала придумайте алгоритм, а потом сделайте его рекурсивную (при надобности) реализацию.

Рекурсивные графические алгоритмы

1. **Рассмотрим построение изображения, показанного на рис. 1**

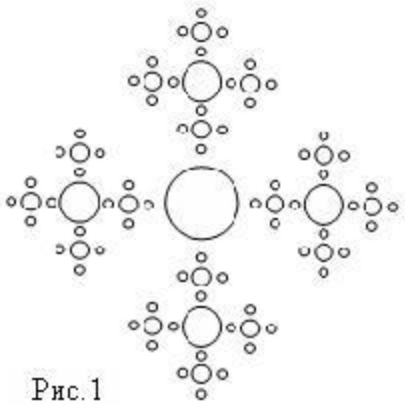


Рис.1

На рисунке бóльшая окружность окружена четырьмя окружностями поменьше, каждая из которых в свою очередь окружена четырьмя окружностями с еще меньшим радиусом. Всего изображено четыре уровня окружностей.

Для того, чтобы составить программу построения этого изображения, можно:

- описать процедуру изображения одной окружности с четырьмя окружностями поменьше;
- для изображения каждой окружности следующего уровня использовать эту же процедуру, только с другими значениями параметров (координат центров, величин радиусов и проч.).

Опишем алгоритм рисования окружности радиуса r с с центром в точке (x, y) с четырьмя окружностями вокруг. При этом необходимо знать расстояния $r1$ от точки (x,y) до центров окружностей окружения (они, очевидно равны). Пусть $\frac{r'}{r} = k1$, где r' - радиус окружности окружения, $\frac{r1}{r} = k2$.

На языке Паскаль данный алгоритм будет иметь следующий вид:

```
uses Crt, Graphabc;
var x,y,n,r,r1,a,b: integer;
    k1,k2: real;
procedure p(x,y,r,r1,n:integer);
  var x1,y1,i: integer;
    begin
      if n>0
        then
          begin
            SetPenColor(clblack);circle (x,y,r);{sleep(20);}
```

```
            r1:=round(r*k2);
            for i:=1 to 4  do
              begin
                x1:=round(x+r1*cos(pi/2*i));
                y1:=round(y+r1*sin(pi/2*i));
                p(x1,y1,round(r*k1),r1,n-1);
              end;
          end;
      end;
begin  clearWindow(clwhite);
  n:=4;
  x:=windowwidth div 2;
  y:=windowheight div 2;
  k1:=0.3;k2:=3;r:=40;
  p(x,y,r,r1,n);
end.
```

Таким образом, в описанной процедуре осуществляется вызов ее же самой в качестве вспомогательной.

Для вычисления значений *x1* и *y1* воспользуемся определениями тригонометрических функций sin и cos: $x1 = x + r1 \cdot \cos\alpha$, $y1 = y + r1 \cdot \sin\alpha$, *где* $\alpha = \frac{\pi}{2}, \pi, \frac{3\pi}{2}, 2\pi.$

Для того, чтобы программа с данной рекурсивной процедурой работала не бесконечно, необходимо в качестве аргумента процедуры ввести некоторую величину *n* (здесь логично за величину *n* взять число уровней окружностей), которая при каждом новом вызове процедуры будет уменьшаться на 1, а в тело процедуры включить условие, что его операторы будут выполняться только при *n>0*. Данное условие будет играть роль своеобразной «заглушки» (граничное условие), ограничивающее число вызовов процедуры.

В основной программе запрашивается количество уровней *n*, задаются координаты центра большой окружности *(x, y)* и ее радиус *r*, а так же коэффициенты *k1* и *k2*. Центр самой большой окружности располагается в центре экрана.

Изменяя в процессе демонстрации работы программы значения *k1, k2, n* можно получить множество привлекательных рисунков.

Что надо изменить в программе, чтобы получить следующие рисунки:

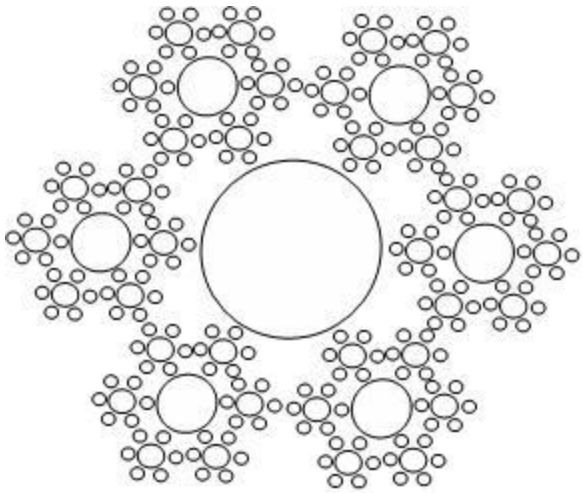


рис. 2

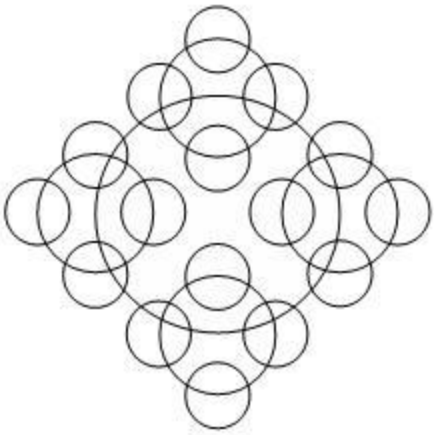


рис. 3

2. **Рассмотрим построение некоторых фрактальных кривых.**

Математическое описание бесконечно дробимых объектов уравнениями линий или поверхностей чрезвычайно громоздко из-за необъятного количества мельчайших объектов. Для преодоления этой трудности математиком Исследовательского центра корпорации IBM Бенуа Мандельбротом в 1975 году был введен термин “фрактал” (от латинского fractus – раздробленный, разбитый, состоящий из фрагментов), а в 1982 году опубликована основополагающая книга “Фрактальная геометрия природы”, где описаны фрактальные множества, их свойства, методы получения и изображения.

В машинной графике фракталы применяются при генерации искусственных облаков, гор, поверхности моря. Образы фракталов похожи на природные объекты — деревья, растения, узоры, звездные скопления т.п. Фракталы позволяют создавать с помощью нескольких коэффициентов линий и поверхности очень сложной формы.

Ниже рассмотрим некоторые из этих фрактальных множеств.

а) Множество Кантора (рис. 4):

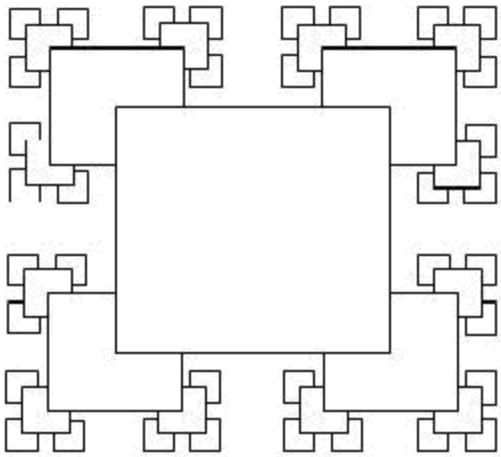


Рис. 4

Алгоритм рисования “множества Кантора”:

- построить большой квадрат;
- на его вершинах, как на центрах рисуются квадраты, в четыре раза меньшие первоначального;
- это повторяется для каждого оставшегося квадрата (*n*), причем большие квадраты перекрывают меньшие.

```
uses crt, graphabc;
 var r:integer;
procedure star(x,y,r:integer);
begin
if r>2 then begin
star(x+r,y+r,r div 2);
star(x+r,y-r,r div 2);
star(x-r,y-r,r div 2);
star(x-r,y+r,r div 2);
rectangle(x-r,y-r,x+r,y+r);   {sleep(50);}
end;
```

```
end;
begin
      clearWindow(clwhite);
      SetPenColor(clblack);  {SetBrushStyle(bsClear); }
      star(windowwidth div 2, windowheight div 2,windowheight div 4
);
end.
```

Данная программа даёт возможность изменения количества уровней *n* (глубину рекурсии), а так же размеры квадратов.

б) Самым знаменитым примером площадного геометрического фрактала является треугольник Серпинского (см. рис.5), строящийся путем разбиения треугольника, необязательно равностороннего – средними линиями на четыре подобных треугольника, исключением центрального и рекурсивного разбиения угловых треугольников до получения площадных элементов желаемого разрешения.

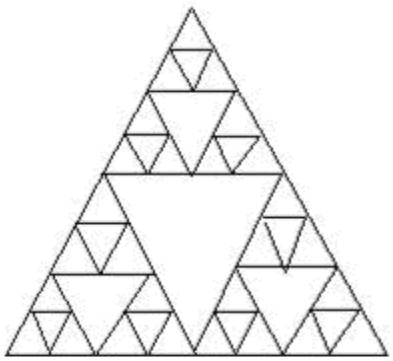


Рис. 5

Алгоритм построения треугольника Серпинского довольно прост:

- строится большой внешний треугольник (А);
- строится треугольник, получающийся при соединении середин сторон большого треугольника (Б);
- строятся треугольники, получающиеся аналогично элементу Б, но в качестве большого треугольника берутся треугольники, образованные элементами А и Б.

Изображение состоит из однотипных элементов, связанных между собой зависимостью каждого следующего элемента от координат предыдущего

```
Uses Crt,Graphabc;
Var x1,y1,x2,y2,x3,y3, a,b,n: integer;
PROCEDURE TRI(x1,y1,x2,y2,x3,y3, N: integer);
Var x12,y12,x23,y23,x31,y31: integer;
Begin If N<>0 then
      begin
       x12:=(x1+x2) div 2;    y12:=(y1+y2) div 2;
       x23:=(x2+x3) div 2;    y23:=(y2+y3) div 2;
       x31:=(x3+x1) div 2;    y31:=(y3+y1) div 2;
       SetPenColor(clblack);
MoveTo(x31,y31); LineTo(x12,y12);
                 LineTo(x23,y23);
```

```
                  LineTo(x31,y31);
TRI(x1,y1,x12,y12,x31,y31, N-1);
TRI(x2,y2,x12,y12,x23,y23, N-1);
TRI(x3,y3,x31,y31,x23,y23, N-1)
  end; end;
Begin  clearWindow(clwhite); SetWindowSize(800,600);
write('n= ');readln(n);
x1:=320; y1:=0; x2:=639; y2:=479; x3:=0; y3:=479;
Moveto(x1,y1);   Lineto(x2,y2);
                  LineTo(x3,y3);
                  LineTo(x1,y1);
TRI(x1,y1,x2,y2,x3,y3, n);
 end.
```

На рис. 5 представлено изображение, состоящее из трех уровней, а данная программа позволяет рисовать изображение в зависимости от введённого пользователем *n* уровней.

Преимущество использования рекурсии очевидно - без рекурсии построение такого рисунка состоящего более чем из шести уровней весьма проблематично, а рекурсия позволяет увеличивать количество уровней, не ограничиваясь минимальными размерами самого нижнего уровня. Например, с помощью этой программы можно увеличить количество уровней до пятнадцати при этом будет ощутима только некоторая задержка при выводе изображения на экран, а вот без рекурсии такой рисунок построить будет практически невозможно, так как изображение будет состоять более чем из тридцати одной тысячи треугольников.

в) «Ветка» (см. рис. 6)

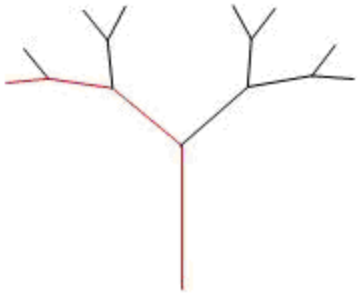


Рис. 6

Длина первоначального звена, коэффициент уменьшения следующих звеньев задаются пользователем.

Алгоритм работы над задачей:

- Введем обозначения: *n* – количество звеньев ветки, длина первоначального (вертикального) отрезка ветки (в точках) - *dl*, коэффициент уменьшения каждого звена - *t* .
- Начинаем рисовать с первоначального (вертикального) отрезка, далее, если это не последнее звено, просчитывается длина следующего отрезка и строится этот отрезок; так продолжается до тех пор, пока не прорисуются все n звеньев.
- Далее сначала прорисовывается левая часть ветви, а затем правая.

```
uses Graphabc,Crt;
var c,t,n,dl: integer; m:real;
```

```
procedure vetka(x,y,L,n: integer; a: real);
Const
b=pi/4;
var x0,y0: integer;
begin
if n+1<>0 then begin
Moveto(x,y);
x0:=x; y0:=y;
x:=x+round(L*cos(a)); y:=y+round(L*sin(a));
SetPenColor(clblack);
Lineto(x,y);
vetka(x,y,round(L*m),n-1,a-b/2);  vetka(x,y,round(L*m),n-1,a+b/2);
end; end;
begin clearWindow(clwhite); SetWindowSize(800,600);
write('n=');readln(n);
{write('dl=');readln(dl);
write('m=');readln(m);  }
dl:=150; m:=0.6;
vetka(300,450,dl,n,-pi/2);
end.
```

При работе с данной программой обратите внимание на ограничения, накладываемые на ввод данных: *n* - количество звеньев (>1); *dl* - длина первоначального отрезка в точках, брать 10 точек …(начните со 100); 0<*m*<*1*.

Ссылки:

http://pascal.proweb.kz/index.php?page=252
http://uvd45.ru/4-kurs/metodika-i-soderzhanie-podgotovki-uchashchikhsia-k-olimpiadam-po-progra/
http://festival.1september.ru/articles/506555/
http://www.tvd-home.ru/recursion

Батракова Людмила Васильевна