Рекурсия в программировании

Лекция №10

Понятие рекурсии

Рекурсивным называется объект, который частично определяется через самого себя.

Факториал числа

Нерекурсивное определение:

N!= 1*2*..*N, при N>0,

и N!=1 при N=0.

Рекурсивное определение:

$$N! = \begin{cases} 1, если N = 0, \\ N*(N-1)!, если N > 0. \end{cases}$$

Пример

5!= 1*2*3*4*5=120.

Рекурсивное определение:

5!=5*4!

4!=4*3!

3!=3*2!

2!=2*1!

1!=1*0!

0!=1

Числа Фибоначчи

- **Нерекурсивное определение:**

$$F(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

- **Рекурсивное определение:**

$$F(n) = \begin{cases} F(1) = F(2) = 1, \\ F(n) = F(n-1) + F(n-2), n > 2. \end{cases}$$

$F_6 = F_5 + F_4$
$F_5 = F_4 + F_3$
$F_4 = F_3 + F_2$
$F_3 = F_2 + F_1$
$F_2 = 1$
$F_1 = 1$

$F_1 = 1$
$F_2 = 1$
$F_3 = 1+1=2$
$F_4 = 1+2=3$
$F_5 = 2+3=5$
$F_6 = 3+5=8$

Содержание и мощность рекурсивного определения, а также его главное назначение, состоит в том, что оно позволяет с помощью конечного выражения определить бесконечное множество объектов.

Использование рекурсии позволяет легко (почти дословно) запрограммировать вычисления по рекуррентным формулам.

Рекурсивное определение состоит из двух частей:

- **Базисного выражения**

  (оно нерекурсивно),
- **Рекурсивного выражения**

и строится так, чтобы полученная в результате повторных применений цепочка определений сходилась к базисному утверждению.

базисное выражение

$$N!=\begin{cases}1, если N=0,\\ N*(N-1)!, если N>0.\end{cases}$$

Рекурентное выражение

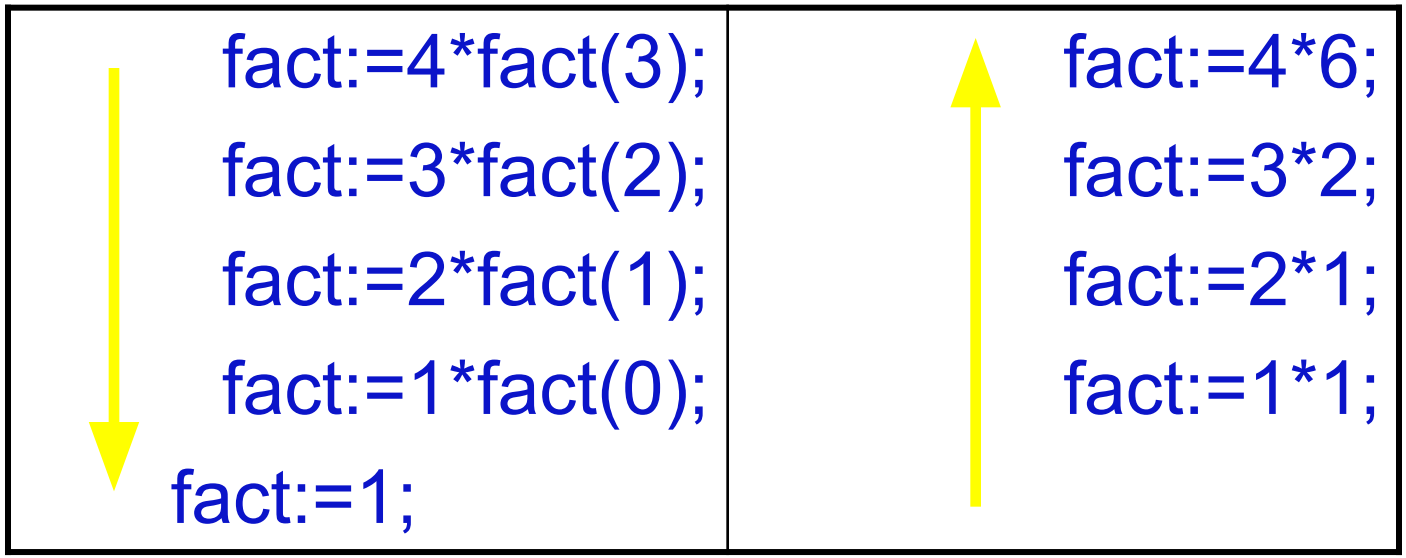
5!=5*4!
4!=4*3!
3!=3*2!
2!=2*1!
1!=1*0!
0!=1

В программировании *рекурсивной* называется подпрограмма, которая в процессе выполнения вызывает сама себя.

```
function fact (n:word):longint;
 begin
   if n=0 then fact:=1
          else fact:=n*fact(n-1);
 end;
```

Программы, в которых используются рекурсивные алгоритмы отличаются простотой, наглядностью и компактностью текста. Это обуславливается тем, что рекурсивная процедура указывает, что нужно делать, а нерекурсивная больше акцентирует внимание на том, как нужно делать.

N:=fact(4); N=24


fact:=4*fact(3);
fact:=3*fact(2);
fact:=2*fact(1);
fact:=1*fact(0);
fact:=1;
fact:=4*6;
fact:=3*2;
fact:=2*1;
fact:=1*1;

Формула рекурсивной п/п

Чтобы рекурсия не зацикливалась, необходимо соблюдать два условия:

1) **Обязательно должно быть условие окончания рекурсии (базисное выражение);**
2) **Если есть параметр рекурсии, то он должен изменяться таким образом , чтобы привести к условию окончания рекурсии (базису).**

```
Procedure Rec (t:<тип>);
Begin
<действия на входе в  рекур-ю>;
If <усл> then Rec(t+f);
 <действия на выходе из рекур-ии>;
End;
```

```
Function Func (t:<тип>):тип;
Begin
<действия на входе в  рекурсию>;
If <усл> then Func:=Func(t+f);
 <действия на выходе из рекурсии>;
End;
```

```
procedure Bit  (n:word);
  begin
       if n>1 then bit  (n div 2);
       write (n mod 2);
 end;
```

Рекурсия и Итерации

While

```
i:=0;  N:=10;
While i<=n do
    begin
        inc(i);
        readln(a);
     end;
```

Рекурсия

```
Procedure Inp (i,n:byte);
Var a:integer;
Begin
   readln(a);
    if i<=n then Inp(i+1,n);
End;

…
i:=0; n:=10;
Inp(0,10);
```

Рекурсия и Итерации

While

```
i:=0;
While n>0 do begin
  inc(i);
  M[i]:=n mod 2;
  n:=n div 2;
end;
For j:=i downto 1 do
    Write(m[j]);
```

Рекурсия

```
procedure Bit  (n:word);
  begin
  if n>1 then bit  (n div 2);
  write (n mod 2);
 end;
```

Однако за простоту рекурсивных алгоритмов приходится расплачиваться неэкономным использованием оперативной памяти, так как выполнение рекурсивных процедур требует значительно большего размера памяти во время выполнения.

При каждом рекурсивном вызове для локальных переменных, а также для параметров процедуры, выделяются новые ячейки памяти.

Таким образом, какой-либо переменной на разных уровнях рекурсии будут соответствовать различные ячейки памяти.

Поэтому воспользоваться значением переменной *i*-го уровня можно только находясь на этом уровне.

```
function fact (n:word):longint;
  begin
   if n=0 then fact:=1
          else fact:=n*fact(n-1);
      end;
```

N=5 | n=4 | n=3 | n=2 | n=1 | n=0 |

Каждое обращение к рекурсивной подпрограмме вызывает независимую *активацию* этой подпрограммы. Совокупность данных, необходимых для одной активации рекурсивной подпрограммы, называется *фреймом активации*.

Фрейм активации включает:

- **Копии всех локальных переменных подпрограммы;**
- **Копии параметров-значений;**
- **4-байтовые адреса параметров-переменных и параметров-констант;**
- **копию результата (для функции);**
- **служебную информацию – около 12 байт (в зависимости от способа вызова: дальний или ближний).**

Все фреймы размещаются в стеке, и при большом количестве вызовов возможно переполнение стека.

(Размер стека устанавливается в настройках среды, по умолчанию =16 Кб, его максимальный размер 64 Кб).

Одной из наиболее встречающихся ошибок при создании рекурсивных подпрограмм является «зациклившаяся» или бесконечная рекурсия.

В отличие от бесконечного цикла, обычно она завершается аварийно при переполнении стека.

Пример зацикленной рекурсии

```
Procedure PopeAndDog;
  Begin
    Writeln('У попа была собака, он ее любил.');
    Writeln('Она съела кусок мяса, он ее убил,');
    Writeln('В землю закопал и надпись написал:');
    PopeAndDog;
  End;
```

Разработать программу определения наибольшего общего делителя двух чисел, используя алгоритм Евклида.

- **Базисное утверждение**:

  если a=b, то НОД=a,

- **Рекурсивное утверждение**:

$$НОД(a,b)=\begin{cases} НОД(a-b,b), & если \quad a>b \\ НОД(a,b-a), & если \quad b>a \end{cases}$$

```
Function  NOD (a,b:integer):integer;
  Begin
     If a=b then NOD:=a
          Else
             If a>b then  NOD:=NOD (a-b,b)
                    Else NOD:=NOD (a,b-a)
  End;
```

Задача

Написать программу ввода четных элементов и вывода их в обратном порядке. Массивы не использовать.

```
Procedure vvod;
  var n: integer;
  begin
       write('введите четное  число: ');
       readln(n);
       if n mod 2=0 then vvod;
       write (n:4);
 end;
```

Разработать программу определения корней уравнения с заданной точностью ε на заданном отрезке методом половинного деления.

```
function func(x:real):real;
  begin
    func:=x+12;
  end;
Procedure Root ( a,b:real; var r:real);
     var x:real;
     Begin
         if abs(b-a)/2<eps then r:=(b+a)/2
         else begin
                x:=(a+b)/2;
                if func(a)*func(x)>0
                    then root(x,b,r)
                    else root(a,x,r);
              end;
     end;
```

```
BEGIN
  writeln('введите интервал:');
  readln(a,b);
  writeln('введите точность:');
  readln(eps);
  if func(a)*func(b)>0
      then writeln('на заданном интервале корней
  нет')
      else  begin
              Root(a,b,x);
              writeln('x=',x:10:6);
            end;
  readln;
End.
```

Во всех рассмотренных выше примерах рекурсивные подпрограммы имели общую черту: рекурсивная подпрограмма вызывает себя один раз. Такая организация рекурсии называется линейной.

Кроме линейной достаточно часто встречается рекурсия, получившая название древовидной. При такой структуре подпрограмма в течении одной активации вызывает себя более одного раза. Полученная в данном случае последовательность вызовов имеет форму дерева.

Быстрая сортировка

Быстрая сортировка основывается на принципе «разделяй и властвуй» и является улучшенной модификацией пузырьковой сортировки.

Сначала берется весь массив и частично упорядочивается особенным образом: выбирается серединный элемент, назовем его ключом, а остальные элементы упорядочиваются относительно этого ключа так, чтобы слева располагались элементы меньшие ключа (если массив сортируется по возрастанию), а справа – большие. В итоге ключевой элемент встает на «свое место».

Затем к левой и правой частям (относительно ключа) применяется этот же метод, то есть выбирается новый ключ и упорядочивание подмассива относительно его.

И так до тех до тех пор, пока каждая из частей не будет содержать ровно один элемент.

Реализация данного метода сортировки основывается на рекурсивном вызове процедуры упорядочивания. Она была разработана в 1962 году профессором Оксфордского университета К.Хоаром.

$A=\{6, 23, 17, 8, 14, 25, 6, 3, 30, 7\}$

```
procedure quick(var A:array of integer; l,r:integer);
var     X,i,j, V:integer;

 begin
       X:=A[(l+r)div 2];
       i:=l; j:=r;
       while (i<j) do
          begin
             while A[i]<X do inc(i);
             while A[j]>X do dec(j);
             If i<j then begin
                              V:=A[i];
                              A[i]:=A[j];
                              A[j]:=V;
                            end;
          end;
    if i>l then quick(A,l,j-1);
    if j<r then quick(A,j+1,r);
 end;
```

Эффективность метода быстрой сортировки

На каждом шаге делим массив на две части, для каждой из этих частей – N/2 сравнений (всего – 2*N/2=N),

затем на 2-ом шаге снова делим, получаем N/4 сравнений (4*N/4=N) и т.д.

Так как глубина рекурсии (количество разбиений) равна Log_2N, а количество сравнений на каждом уровне – N, то сложность алгоритма $T = \Theta (N*log_2N)$.