Рекурсивные алгоритмы

Рекурсия – фундаментальное понятие в математике и компьютерных науках. В языках программирования рекурсивной программой называется программа, которая обращается *сама к себе* (подобно тому, как в математике рекурсивная функция определяется через понятия самой этой функции). Рекурсивная программа не может вызывать себя до бесконечности, следовательно, вторая важная особенность рекурсивной программы – наличие *условия завершения*, позволяющее программе прекратить вызывать себя.

Таким образом рекурсия в программировании может быть определена как *сведение задачи к такой же задаче, но манипулирующей более простыми данными.*

Как следствие, рекурсивная программа должна иметь *как минимум два пути выполнения*, один из которых предполагает рекурсивный вызов (случай «сложных» данных), а второй – без рекурсивного вызова (случай «простых» данных).

Примеры рекурсивных программ. В ряде случаев рекурсивную подпрограмму можно построить непосредственно из формального математического описания задачи.

Факториал	

$\begin{cases} n! = n * (n-1)!, \text{ при } n>0 \\ 1, n=0 \end{cases}$	`Function Fact(n:byte):longint;`
	`begin`
	`  if n=0`
	`  then Fact:=1`
	`  else Fact:=n*Fact(n-1)`
	`end;`

Числа Фибоначчи	

$\begin{cases} \Phi_n = \Phi_{n-1} + \Phi_{n-2}, \text{ при } n>1 \\ \Phi_0 = 1, \Phi_1 = 1 \end{cases}$	`Function F(n:byte):longint;`
	`begin`
	`  if n <= 1`
	`  then F:=1`
	`  else F:= F(n-1)+F(n-2)`
	`end;`

Рекурсия и итерация. Рекурсивную программу всегда можно преобразовать в нерекурсивную (итеративную, использующую циклы), которая выполняет те же вычисления. И наоборот, используя рекурсию, любое вычисление, предполагающее использование циклов, можно реализовать, не прибегая к циклам.

К примеру, вычисление факториала и чисел Фибоначчи можно реализовать без рекурсии:

Факториал	

`Function Fact(n:byte):longint;`
`var F, i : byte;`
`begin`
`  F:=1;`
`  for i:=1 to n do F:=F*i;`
`  Fact:=F`
`end;`
	`Function Fact(n:byte):longint;`
	`begin`
	`  if n=0`
	`  then Fact:=1`
	`  else Fact:=n*Fact(n-1)`
	`end;`

Числа Фибоначчи	

`Function F(n:byte):longint;`
`var F0, F1, F2 : longint; i : byte;`
`begin`
`  F0:=1; F1:=1;`
`  for i:=2 to n do`
`  begin`
`  F2:=F1+F0; F0:=F1; F1:=F2;`
`  end;`
`  F:=F1`
`end;`
	`Function F(n:byte):longint;`
	`begin`
	`  if n <= 1`
	`  then F:=1`
	`  else F:= F(n-1)+F(n-2)`
	`end;`

При выполнении рекурсивной подпрограммы сначала происходит «рекурсивное погружение», а затем «возврат вычисленных результатов». Например, вычисление 5! при помощи вызова Fact(5) будет происходить следующим образом:

```
Fact(5) → 5 * Fact(4)                    5 * 24    →   120
            ↓
          4* Fact(3)                     4 * 6
            ↓
          3 * Fact(2)                    3 * 2
            ↓
          2 * Fact(1)                    2 * 1
            ↓
            1            →                 1
```

Как видно на примере вычисления 5!, при «рекурсивном погружении» функция Fact вызывает точно такой же новый экземпляр самой себя. При этом сама функция как бы еще не завершилась, а новый ее экземпляр уже начинает работать. И только когда новый экземпляр завершит работу («вернет вычисленные результаты»), будет продолжена работа самой функции.

Стек. Информация о таких незавершенных вызовах рекурсивных подпрограмм (а это, в самом простом представлении, значения переменных, необходимых для работы подпрограммы) запоминается в специальной области памяти – стеке. При работе в среде Turbo Pascal содержимое стека и весть процесс появления и завершения рекурсивных вызовов можно просмотреть в окне «Call Stack», которое вызывается нажатием клавиш Ctrl+F3.

Размер стека по умолчанию – 16Кб. Если незавершенных рекурсивных вызов слишком много (хотя вы по ошибке могли написать и программу с бесконечной рекурсией – тогда вам ничто не поможет), то система выдаст сообщение «Stack overflow» («Переполнение стека»). В этом случае есть смысл увеличить размер стека используя меню «Options – Memory Sizes ”.

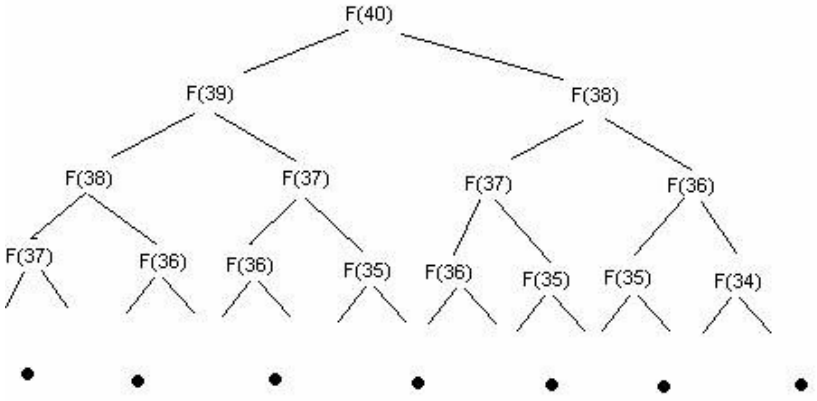
Также помните, что нажав клавиши **Ctrl+O O**, вы перед текстом программы получите список значений всех директив компилятора Turbo Pascal, в котором можно изменять признаки их активности и значения, установленные по умолчанию.

```
{$A+,B-,D+,E-,F-,G+,I-,L+,N+,O-,P-,Q-,R-,S-,T-,V+,X+}
{$M 16384,0,655360}
```

Размер стека определяется первым параметром директивы $M. Второй и третий ее параметры – нижняя и верхняя границы области динамически выделяемой памяти (кучи, heep). Размер стека можно увеличить непосредственно в такой строке, однако эти установки будут действовать только для данной программы.

Сложность рекурсивных вычислений. При относительной простоте написания, у рекурсивных подпрограмм часто встречается существенный недостаток – неэффективность. Так, сравнивая скорость вычисления чисел Фибоначчи с помощью итеративной и рекурсивной функции можно заметить, что итеративная функция выполняется почти «мгновенно», не зависимо от значения n. При использовании же рекурсивной функции уже при n=40 заметна задержка при вычислении, а при больших n результат появляется весьма не скоро.

Неэффективность рекурсии проявляется в том, что одни и те же вычисления производятся по многу раз. Так для вычисления 40-го числа Фибоначчи схема рекурсивных вызовов представлена на рисунке ниже.



Оценить сложность рекурсивных вычислений (количество рекурсивных вызовов) можно с помощью рекуррентных соотношений. **Рекуррентное соотношение** – это *рекурсивная функция с целочисленными значениями*. Значение любой такой функции можно определить, вычисляя все ее значения начиная с наименьшего, используя на каждом шаге ранее вычисленные значения для подсчета текущего значения.

Рекуррентные выражения используются, в частности, для определения *сложности рекурсивных вычислений*.

Например, пусть мы пытаемся вычислить числа Фибоначчи по рекурсивной схеме

```
F(i) = F(i-1) + F(i-2), при N >= 1; F(0) = 1; F(1) = 1;
```

с помощью указанной выше рекурсивной подпрограммы-функции F(n).

Требующееся при вычислении значения F(N) по такой схеме количество рекурсивных вызовов может быть получено из решения рекуррентного выражения

$$T_N = T_{N-1} + T_{N-2}, \text{ при } N >= 1;\ T_0 = 1;\ T_1 = 1$$

T_N приблизительно равно Φ^N , где $\Phi \approx 1.618$ - золотая пропорция («золотое сечение»), т.е. приведенная выше программа потребует экспоненциальных временных затрат на вычисления.

Метод «разделяй и властвуй». Многие алгоритмы используют два рекурсивных вызова, каждый из которых работает приблизительно с половиной входных данных. Такая рекурсивная схема, по-видимому, представляет собой наиболее важный случай хорошо известного метода «разделяй и властвуй» (divide and conquer) разработки алгоритмов.

В качестве примера рассмотрим задачу отыскания максимального из N элементов, сохраненных в массиве a[1], . . . , a[N] с элементами типа Item. Эта задача легко может быть решена за один проход массива:

```
Max:=a[1];
For i:=1 to N do
   if a[i] > Max then Max:=a[i];
```

Рекурсивное решение типа «разделяй и властвуй» - еще один простой (хотя совершенно иной) способ решения той же задачи:

```
Function Max (a : array of Item; l, r : integer) : Item;
var u, v : Item; m : integer;
begin
   m := (l+r) / 2;
   if (l = r)
      then Max := a[l]
      else begin
              u := Max (a, l, m);
              v := Max (a, m+1, r);
              if (u > v) then Max := u else Max := v
           end
end;
```

Чаще всего подход “разделяй и властвуй» используют из-за того, что он обеспечивает более быстрые решения, чем итерационные алгоритмы.

Основным недостатком алгоритмов типа «разделяй и властвуй» является то, что делят задачи на *независимые* подзадачи. Когда подзадачи независимы, это часто приводит к недопустимо большим затратам времени, так как одни и те же подзадачи начинают решаться по многу раз.

К примеру, приведенная выше рекурсивная схема вычисления чисел Фибоначчи абсолютно недопустима при больших значениях N, так как приводит к многократным повторным вычислениям, и экспоненциальной сложности вычислений (T_N приблизительно равно Φ^N , где $\Phi \approx 1.618$ есть золотая пропорция).

Обходить подобные ситуации позволяет подход, известный как динамическое программирование.

Динамическое программирование. Общий подход для реализации рекурсивных программ, который дает возможность получать эффективные и элегантные решения для обширного класса задач.

Технология, называемая *восходящим динамическим программированием* (bottom-up dynamic programming) основана на том, что значение рекурсивной функции можно определить, вычисляя все значения этой функции, начиная с наименьшего, используя на каждом шаге ранее вычисленные значения для подсчета текущего значения.

Она применима к любому рекурсивному вычислению *при условии*, что мы можем позволить себе хранить все ранее вычисленные значения. Что в результате позволит уменьшить временную зависимость с экспоненциальной на линейную !

Нисходящее динамическое программирование (top-down dynamic programming) – еще более простая технология. Она позволяет выполнять рекурсивные функции при том же количестве итераций, что и восходящее динамическое программирование. Технология требует введения в рекурсивную программу неких средств, обеспечивающих сохранение каждого вычисленного значения и проверку сохраненных значений во избежание их повторного вычисления.

К примеру, сохраняя вычисленные значения в статическом массиве K[1..100] (предварительно проинициализированном, к примеру, числом -1), мы явным образом исключим любые повторные вычисления.

Приведенная ниже программа вычисляет F(N) за время, пропорциональное N.

```
Function F( n : integer ) : longint;
begin
   if (K[n] <> -1)
      then F := K[n]
      else if n < 2
              then F := n
              else begin
                      t := F(n-1) + F(n-2)
                      K[n] := t;
                      F := t
                   end
end;
```

Примеры задач, решаемых с помощью рекурсии.

Поиск максимального элемента массива. Поскольку текст функции для поиска максимального элемента массива приведен выше, остановимся лишь на особенностях ее применения.

```
Type Item = integer;
Const Mas : array [1..10] of Item = (3,6,2,4,1,8,0,9,5,7);

Function Max (A : array of Item; l, r : integer) : Item;
. . . . .
end;

begin
  writeln (Max(Mas,0,9))
{l=0, r=9 т.к. параметр A в списке формальных параметров функции Max – открытый массив, т.е.
значения индексов начинаются от 0}
end
```

Поиск элемента в упорядоченном массиве (двоичный поиск).

Имеется упорядоченный массив и эталонный элемент. Требуется определить, содержится ли эталон в массиве. Если «да», то вернуть соответствующий номер позиции. Если «нет» - вывести сообщение.

Для решения задачи используется метод деления исходного массива пополам.

С эталоном сравнивается «средний» (расположенный по середине) элемент массива. Если он меньше эталона – поиск продолжается в правой половине массива. Если меньше – в левой.

Поиск ведется до тех пор, пока не будет обнаружено соответствие, или пока длина участков массива, в которых ведется поиск, не станет меньше 1.

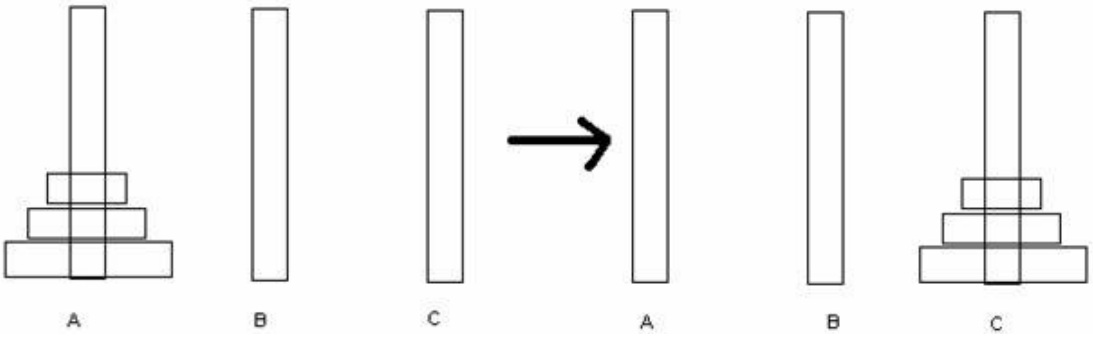
```
Type Item = integer;
Const A : array[1..10] of Item = (1,2,3,4,5,6,7,8,9,10);

Function Se (a : array of Item; e : Item; l, r : integer) : integer;
{a – массив, e – эталон поиска,
l, r – левая и правая границы подмассива, в котором производится поиск
Функция возвращает позицию найденного элемента (нумерация от 0) или -1 }
var m : integer;
begin
  if (r=l) and (a[r]<>e) then Se:=-1
    else begin
           m := (l+r) div 2;
           if (e = a[m])
             then Se := m
             else if e < a[m]
                    then Se := Se (a, e, l, m)
                    else Se := Se (a, e, m+1, r)
end;

begin
  P:= Se(A,0,0,9);
  if P<>-1 then writeln('Искомый элемент в позиции ', P+1)
           else writeln('Искомый элемент не присутствует в массиве')
end.
```

Ханойские башни. Имеется три стержня A, B, C. На стержне A надето N колец, причем кольцо меньшего диаметра должно располагаться над кольцом большего диаметра. **Требуется переместить N колец с A на C, используя B как промежуточный стержень.** При перемещении колец кольцо меньшего диаметра можно класть поверх кольца большего диаметра, но не наоборот. Программа должна распечатать протокол перемещений (цепочку команд вида **X -> Y**).

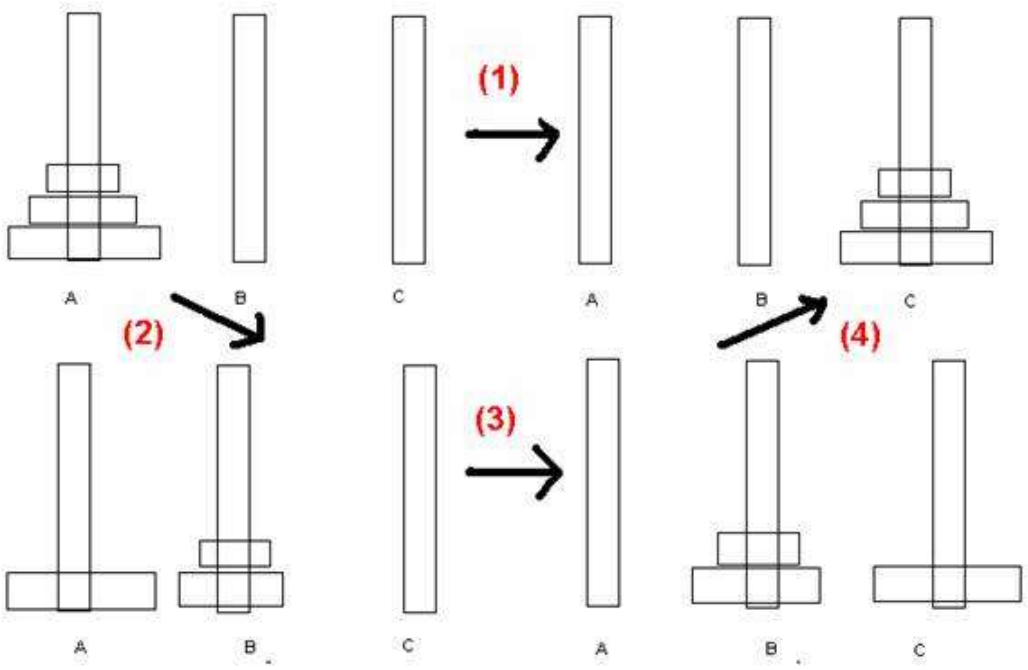
Например, в случае N=3 исходную и конечную ситуацию можно изобразить так:



Протокол действий будет иметь вид:

```
A → C, A → B, C → B,          (2)
A → C,                         (3)
B → A, B → C, A → C                 (4)
```

Я не случайно записал протокол в три строки. Определим смысл действий, записанных в каждой из строк, и проиллюстрируем их:



(1) – Переместить N колец с A на C, используя B как промежуточный

(2) – Переместить (N-1) кольцо с A на B, используя C как промежуточный

(3) – Переместить кольцо с A на C

(4) – Переместить (N-1) кольцо с B на C, используя A как промежуточный

К тому же (1) = (2) → (3) → (4). Т.е. исходная задача (1), работающая с N кольцами, сводится рекурсивно к таким же по сути задачам (2) и (4), в которых используется (N - 1) кольцо.

Программная реализация решения:

```
Procedure Towers(n:byte; A, C, B : char)
{переложить N колец с A на C, используя B как промежуточный}
begin
  if n=1 then writeln(A, ' → ', C)
         else begin
                Towers(n-1, A, B, C);
                writeln(A, ' → ', C);
                Towers(n-1, B, C, A)
              end
end;

begin
  Towers(4, 'A', 'C', 'B') { решаем задачу при N = 4 }
end.
```

Размен денег. Имеется некоторая сумма денег S и набор монет с номиналами a1, …, an. Монета каждого номинала имеется в единственном экземпляре. Необходимо найти все возможные способы разменять сумму S при помощи этих монет.

```
const n = 6;  {число монет}
      s = 33; {требуемая сумма}
      a : array[1..n]of integer =(1,2,3,5,10,15); {номиналы монет}
var kol : array[1..n] of integer; {сколько взято таких монет -0 или 1}
    found : boolean; { признак, что хотя бы одно решение найдено}

procedure rec(k,s1:integer);
{k - число уже использовавшихся монет, s1 - набранная сумма}
var i : integer;
  begin
    if s=s1 then begin  {вывод найденного решения}
      found:=true;
      for i:=1 to n do
        if kol[i]>0 then write(a[i],' ');
      writeln; exit
    end;
    for i:=k+1 to n do  {цикл по всем еще неиспользовавшимся монетам}
      begin
        {самое критичное место рекурсивного перебора -
         «взять-проверить-вернуть обратно, чтобы взять следующую»}
        kol[i]:=1;       {взять i-ю монету}
        rec(i,s1+a[i]); {проверить, годится ли}
```

```
        kol[i]:=0;       {вернуть i-ю обратно, чтобы взять следующую}
      end;
  end;

begin
  found:=false;
  rec(0,0);
  if not found then writeln('Разменять невозможно')
end.
```

Размен денег 2. К условию предыдущей задачи добавим, что монет каждого номинала может быть несколько.

В этом случае изменится только перебор очередных монет

```
const n = 6;  {число монет}
      s = 33; {требуемая сумма}
      a : array[1..n]of integer = (1,2,3,5,10,15); {номиналы монет}
      b : array[1..n]of integer = (1,1,5,1,3,1); {количество монет}
var kol   : array[1..n] of integer; {сколько взято таких монет}
    found : boolean; { признак, что хотя бы одно решение найдено}

procedure rec(k,s1:integer);
{k - число уже использовавшихся номиналов, s1 - набранная сумма}
var i, p : integer;
  begin
    if s=s1 then begin {вывод найденного решения}
      found:=true;
      for i:=1 to n do
        if kol[i]>0 then write(a[i] ,':',kol[i],' ');
      writeln; exit
    end;
    for i:=k+1 to n do  {цикл по всем еще неиспользовавшимся монетам}
      begin
        {самое критичное место рекурсивного перебора -
         «взять-проверить-вернуть обратно, чтобы взять следующие»}
         p:=b[i];          {запомнить количество i-х монет}
         while (b[i]>0) do
         begin
           inc(kol[i]);    {взять следующую i-ю монету}
           dec(b[i]);      {уменьшить число оставшихся i-х монет}
           rec(i,s1+a[i]*kol[i]);
         end;
         b[i]:=p;          {восстановить количество i-х монет}
         kol[i]:=0;        {вернуть все i-е монеты}
      end;
  end;

begin
  found:=false;
  rec(0,0);
  if not found then writeln('Разменять невозможно')
end.
```

Деревья. Изучение рекурсии неразрывно связано с изучением рекурсивно определяемых структур данных, называемых *деревьями* (trees). Деревья используются как для упрощения понимания и анализа рекурсивных программ, так и в качестве явных структур данных. В свою очередь, рекурсивные программы используются для построения деревьев. Глобальная связь между ними (и рекуррентными отношениями) используется при анализе алгоритмов.

Деревья рассмотрим позже, в теме «Динамические структуры данных».