Лекция 5.
Рекурсия (начало)

До настоящего времени мы строили алгоритмы три вида алгоритмов: *линейный, разветвленный, циклический*. И только с помощью только этих трех видов алгоритмов можно написать любой алгоритм — это основы структурного программирования. То есть сводим решение задачи, например к *линейному* алгоритму (ввод — вывод — решение — вывод), затем Ввод сводим к *линейному* алгоритму (ввод размеров матрицы — ввод самой матрицы). Следом — ввод размеров матрицы сводим к *линейному* алгоритму (ввод количества строк — ввод количества столбцов), а ввод матрицы сводим к *циклическому* алгоритму ввода одной строки, а ввод строки в свою очередь сводим к *циклическому* алгоритму ввода элемента строки. Аналогично надо вывод и решение свести к одному из видов алгоритмов, и затем и каждую часть этого решения. Заканчивается такое разложение (декомпозиция), когда мы достигнем *базовых* решений. Для ЯВУ (языков высокого уровня) это операции присваивания, ввода, вывода. Такой прием сведения задачи к решению серии подобных задач вплоть до базовых случаев и называется рекурсией.

Цикл представляет собой краткую запись повторение постоянного или переменного количества операторов. Такой алгоритм невозможно или очень длинно записывать без использования цикла. Если таких операторов фиксированное количество, например, девять, то их можно записать линейно. Если их переменное количество N, то без цикла этот алгоритм становится неоправданно длинным.

Ровно 9 раз	Ровно N (нельзя)	Ровно N (от 1 до 9) (длинно)	Ровно N (с циклом)

Sum:=0;
Sum:= Sum+ A[1];
Sum:= Sum+ A[2];
Sum:= Sum+ A[3];
Sum:= Sum+ A[4];
Sum:= Sum+ A[5];
Sum:= Sum+ A[6];
Sum:= Sum+ A[7];
Sum:= Sum+ A[8];
Sum:= Sum+ A[9];
	Sum:=0;
	Sum:= Sum + A[1];
	.. //нельзя так
	Sum:= Sum + A[N];
		Sum:=0;
		Sum:= Sum + A[1];
		If N>1 then
		Sum:= Sum + A[2];
		If N>2 then
		Sum:= Sum + A[3];
		If N>3 then
		Sum:= Sum + A[4];
		If N>4 then
		Sum:= Sum + A[5];
		If N>5 then
		Sum:= Sum + A[6];
		If N>6 then
		Sum:= Sum + A[7];
		If N>7 then
		Sum:= Sum + A[8];
		If N>8 then
		Sum:= Sum + A[9];
			Sum:=0;
			For i:=1 to N do
			Sum:= Sum + A[1];

Параметрический цикл существенно укорачивает запись того же кода, а чем короче запись, тем меньше дублей, и тем меньше шансов сделать ошибку и легче ее найти и исправить.

С процедурами аналогично – самый короткий способ – написать подзадачу с использованием **параметров**, которые позволяют повторно использовать уже написанный код и избегать дублей с разными ошибками в них. Пример – в 11 лабораторной работе первой части курса. Если собрать процедуры в библиотеку (модуль), то можно повторно использовать написанный код не только в одной, но и в других программах.

Рекурсия тоже позволяет записать в укороченной форме, *определенный вид алгоритма*, который без использования рекурсивных вызовов через ***стек вызовов*** можно записать только с помощью своего стека, практически имитирующего работу стека вызовов.

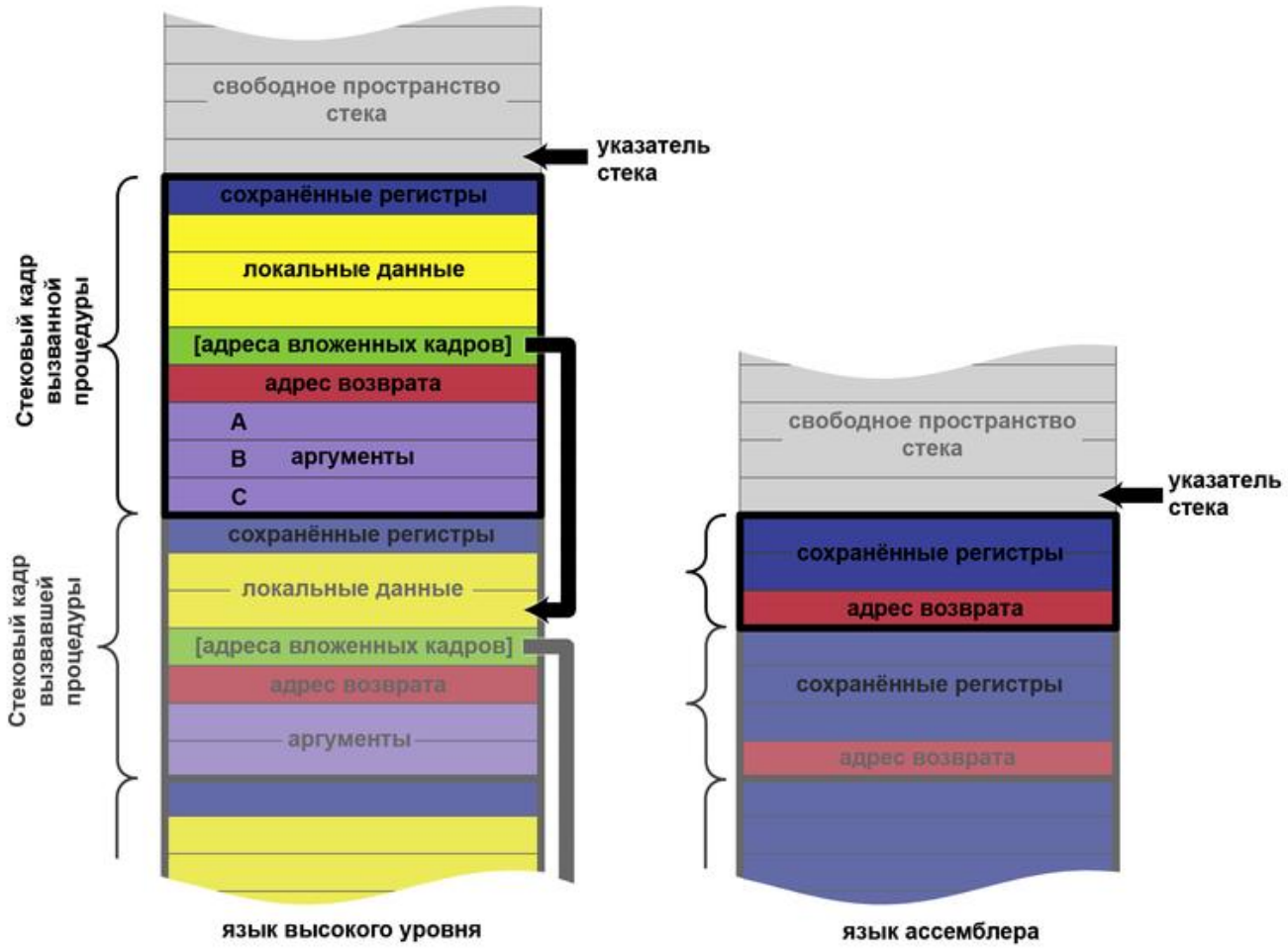


Рис.5.1 Пара способов заполнения **стека вызовов** процедур и функций.

Примером такого изначально *рекурсивного алгоритма* служат задачи полного перебора, например, задача нахождения всех **перестановок без повторений** для строки из разных символов, в которой надо найти все разные последовательности заданных символов без их повторения, которые можно получить из исходной последовательности перестановкой элементов. Например, из символов “ABC” без повторения можно составить следующие строки:

ABC - исходная строка, и ее перестановки:

ACB

BAC

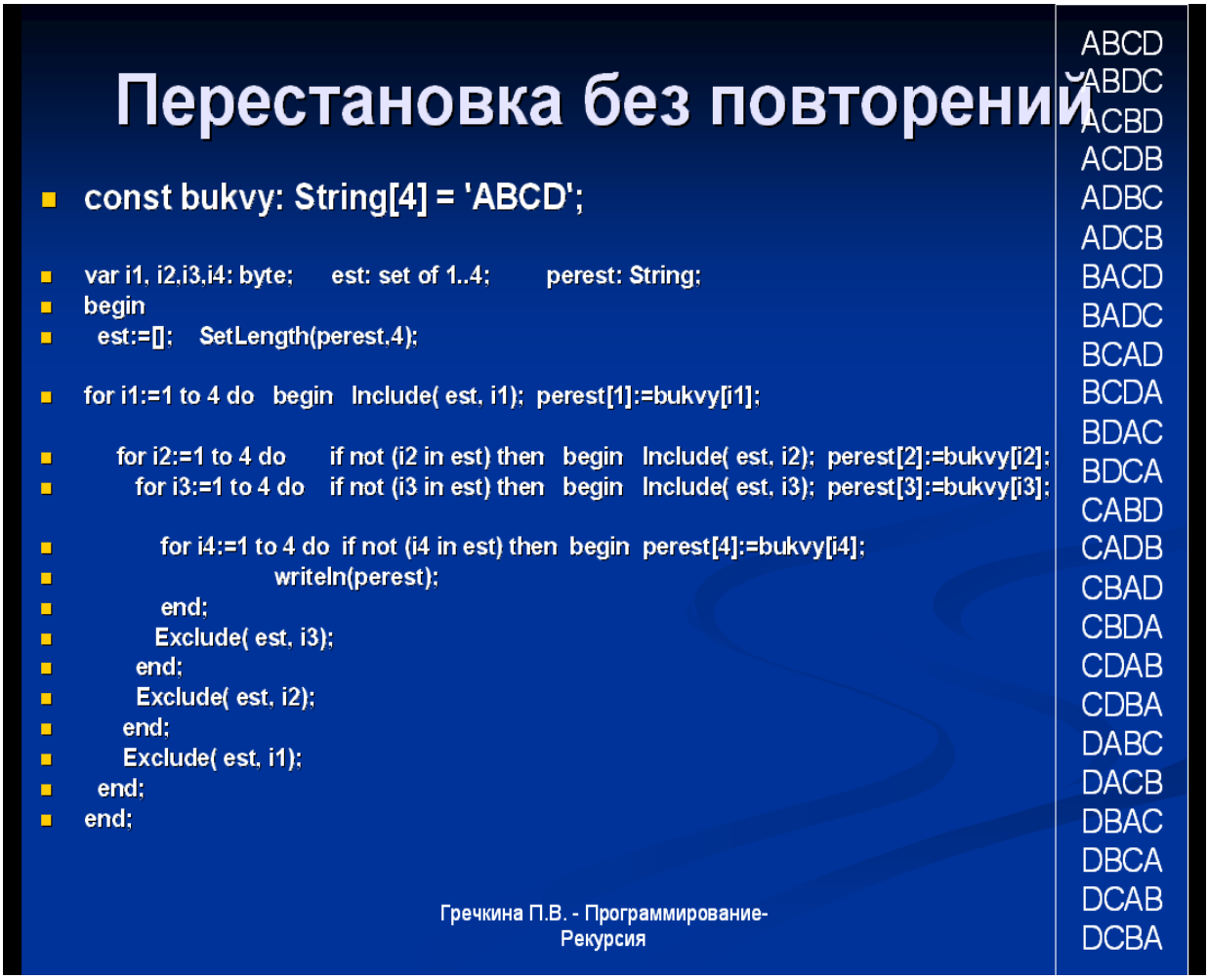
BCA

CAB

CBA

С повторениями добавились бы еще строки: AAA, AAB, AAC. ABA. BAA и так далее и т.п..

При фиксированной длине исходной последовательности, например, четыре, этот алгоритм можно записать ***четырьмя <u>вложенными</u> циклами***:



Если длина ***не фиксирована***, например, надо вывести все перестановки из разных символов N (от 1 до 16), то надо вложить N циклов, чего ***прямо сделать нельзя***, уже потому, что *количество вложений циклов в языке ограничено* меньшим числом, и надо написать конкретное число циклов:

Перестановка без повторений

```
const bukvy: String[16] = 'ABCDEFGHIJKLMNOP';

var i: array [1..16] of byte;    est: set of 1..16;    perest: String;    K: 1..16;
begin
  est:=[];                    readln(K);   if K<1 then K:=1; if K>16 then K:=16;
  SetLength(perest, K);

  for i[1]:=1 to K do  begin  Include( est, i[1]);  perest[1]:=bukvy[i[1]];

    for i[2]:=1 to K do     if not (i[2] in est) then  begin  Include( est, i[2]);  perest[2]:=bukvy[i[2]];
      . . . . . . . //не возможно!

      for i[K-1]:=1 to K do  if not (i[K-1] in est) then begin Include(est, i[K-1]); perest[K-1]:=bukvy[i[K-1]];

        for i[K]:=1 to K do       if not (i[K] in est) then  begin     perest[K]:=bukvy[i[K]];
                  writeln(perest);
        end;
        Exclude( est, i[K-1]);
      end;
      . . . . . . . .
      Exclude( est, i[2]);
    end;
    Exclude( est, i[1]);
  end;
end;
```

Гречкина П.В. - Программирование-Рекурсия

Но для меньшего количества это можно сделать, если добавить несколько строк, например, для N от 1 до 4 код может быть таким:

Перестановка без повторений

```
const bukvy: String[4] = 'ABCD';

var i1, i2,i3,i4, N: byte;    est: set of 1..4;      perest: String;
begin

  write('N=(1..4)'); Readln(N);  if N<1 then N:=1;  if N>4 then N:=4;
  est:=[];    SetLength(perest,N);

 for i1:=1 to N do   begin   Include( est, i1);  perest[1]:=bukvy[i1];
     if N=1 then writeln(perest)
     else
     for i2:=1 to N do      if not (i2 in est) then   begin   Include( est, i2);  perest[2]:=bukvy[i2
       if N=2 then writeln(perest)
       else
       for i3:=1 to N do    if not (i3 in est) then   begin   Include( est, i3);  perest[3]:=bukvy[i3
          if N=3 then writeln(perest)
          else
          for i4:=1 to N do  if not (i4 in est) then  begin  perest[4]:=bukvy[i4];
                             if N=4 then writeln(perest);
          end;
         Exclude( est, i3);
       end;
       Exclude( est, i2);
     end;
     Exclude( est, i1);
  end;
end.
```

```
N=(1..4)1
A
```

```
N=(1..4)2
AB
BA
```

```
N=(1..4)3
ABC
ACB
BAC
BCA
CAB
CBA
```

Гречкина П.В. - Программирование-Рекурсия

Код станет короче, и можно преодолеть ограничение на количество вложенных циклов, если использовать ***рекурсивную процедуру***, в каждой из которой только один цикл, но в теле цикла присутствует ***вызов такой же процедуры***:

Перестановка без повторений

```
const bukvy: String[16] = 'ABCDEFGHIJKLMNOP';
type Tset= set of 1..16;

  procedure perestanovki (n, K: byte; var est: Tset; var perest: string);
  var i: byte;
  begin
    if n>K then writeln(perest)
    else
      for i:=1 to K do      if not (i in est) then      begin   Include( est, i);  perest[n]:=bukvy[i];
                      perestanovki( n+1, K, est, perest);
        Exclude( est, i)
      end;
  end;
```

Обязательной частью рекурсии является ***терминальная ветвь*** (см. все определения в следующей лекции), в которой нет рекурсивных вызовов и решается простейшая или базовая задача, и ***рабочая ветвь для декомпозиции или уменьшения сложности и размерности*** задачи.

Добавим программу для вызова этой рекурсивной функции.

Перестановка без повторений

```
const bukvy: String[16] = 'ABCDEFGHIJKLMNOP';
type Tset= set of 1..16;

  procedure perestanovki (n, K: byte; var est: Tset; var perest: string);
  var i: byte;
  begin
   if n>K then writeln(perest)
   else
    for i:=1 to K do     if not (i in est) then     begin  Include( est, i);  perest[n]:=bukvy[i];
                    perestanovki( n+1, K, est, perest);
     Exclude( est, i)
    end;
  end;

Var    est: Tset;    perest: String;    K: 1..16;
begin
  readln(K);   if K<1 then K:=1; if K>16 then K:=16;

  est:=[];    SetLength(perest, K);

  perestanovki(1, K, est, perest);

  Readln
end.
```

```
ABCDEFG
ABCDEGF
……
GFEDABC
GFEDACB
GFEDBAC
GFEDBCA
GFEDCAB
GFEDCBA
Затрачено 24,83 сек
```

Гречкина П.В. - Программирование-Рекурсия

В некоторые определения изначально имеют рекурсивное описание, например, вычисление факториала на множестве натуральных чисел:

N!= N* (N-1)!

1!=1

0!=1

Здесь первая строка описывает способ декомпозиции., а вторые две – базовые случаи.

Его же можно записать в виде системы правил и в виде соответствующей рекурсивной функции, но и в виде **не**рекурсивной функции тоже:

Нерекурсивный алгоритм более эффективен, и если он достаточно прост и читабелен, лучше использовать его. Некоторые рекурсивные алгоритмы достаточно просто преобразовать в циклические, но есть задачи, которые проще и понятней записать и оставить именно в рекурсивном виде – это задачи полного или ограниченного перебора. Например, это вышеописанная задача перебора всех букв без повторения, которая сводится к вложенным друг в друга циклам и может быть записана только в рекурсивном виде (используя стандартный стек вызовов процедур), или с помощью собственного стека (имитирующего необходимую часть работы стека вызовов).

Примером таких, изначально рекурсивных алгоритмов с переборами, может служить, например, следующая задача:

Дана горсть монет (все по 3-5 штук). Определить все способы набрать 50 рублей. В каком из них наибольшее количество монет?

То есть, задан двухмерный массив, в первой строке, которого находятся номиналы монет, во второй и количество. Найти все комбинации количеств монет, которые надо взять, чтобы набрать 50 рублей. Например,

1 коп	5 коп	10 коп	50 коп	1 руб	2 руб	5 руб	10 руб
5	3	5	3	5	3	5	3

Номиналы можно хранить или в виде целых значений – копеек, 1 руб=100 коп.: либо в рублях 1 коп = 0,01 руб. Точнее будет в копейках, так как 0,01 – в двоичной системе счисления бесконечная и округляется/обрезается при хранении в конечной памяти и точность вычисления результата при большом количестве мелких монет может оказаться недостаточной для правильного решения задачи.

Некоторые возможные комбинации:

50 руб = 3 х 10 руб + 4 х 5 руб

1 коп	5 коп	10 коп	50 коп	1 руб	2 руб	5 руб	10 руб
0	0	0	0	0	0	4	3

50 руб = 3 х 10 руб + 3 х 5 руб + 2 х 2руб + 1 х 1 руб

0	0	0	0	1	2	3	3

50 руб = 2 х 10 руб + 4 х 5 руб + 2 х 2 руб + 4х 1 руб + 3 х 50 коп + 3 х 10 коп +3 х5 коп + 5 х 1 коп

5	3	3	3	4	2	4	2

Последняя – самая большая по количеству необходимых монет.

Еще пример: Дано N цифр (могут быть одинаковыми), определить, сколькими способами можно получить число 20, и перечислить их все.

Большинство **олимпиадных задач** по программированию требуют написания рекурсивного алгоритма с разумно выбранными ограничениями и оценочными функциями, чтобы **избавится от полного перебора** и сократить тем самым время решения.

Задание на лабораторную работу №4 не настолько сложно. Надо всего лишь записать ранее изученные нами алгоритмы (сумма, количество, произведение, минимум, максимум, проверка условия) в рекурсивном виде.
Например, если **сумму вычислять как сумму сумм ее половин** (почти половин при **не**четном количестве слагаемых):

N=5: $\sum_{i=1}^{5} A_i = \sum_{i=1}^{3} A_i + \sum_{i=4}^{5} A_i$ (это НЕ *дерево вызовов*)

$$\sum_{i=1}^{3} A_i = \sum_{i=1}^{2} A_i + \sum_{i=3}^{3} A_i; \quad \sum_{i=4}^{5} A_i = \sum_{i=4}^{4} A_i + \sum_{i=5}^{5} A_i$$

$$\sum_{i=1}^{2} A_i = \sum_{i=1}^{1} A_i + \sum_{i=2}^{2} A_i; \quad \sum_{i=3}^{3} A_i = A_3; \quad \sum_{i=4}^{4} A_i = A_4; \quad \sum_{i=5}^{5} A_i = A_5$$

$$\sum_{i=1}^{1} A_i = A_1; \quad \sum_{i=2}^{2} A_i = A_2$$

рекурсивные вызовы надо заканчивать при достижении простейшего случая, в частности: Для суммы – это сумма из одного слагаемого $\sum_{i=K}^{K} A_i = A_K$; для произведения: $\prod_{i=K}^{K} A_i = A_K$

Для конъюнкции $\bigcap_{i=K}^{K} uslovie(A_i) = uslovie(A_K) = \begin{cases} true, & uslovie(A_K) = true \\ false, & uslovie(A_K) = false \end{cases}$

Для дизъюнкции $\bigcup_{i=K}^{K} uslovie(A_i) = uslovie(A_K) = \begin{cases} true, & uslovie(A_K) = true \\ false, & uslovie(A_K) = false \end{cases}$

Для количества: $\underset{i=K}{\overset{K}{Count}}(uslovie(A_i)) = \begin{cases} 1, & uslovie(A_K) = true \\ 0, & uslovie(A_K) = false \end{cases}$

Для максимума $Max(A_k, A_k) = A_K$ или $Max(A_i, A_k) = \begin{cases} A_i, & A_i \geq A_K \\ A_k, & A_i < A_K \end{cases}$

Пример рекурсивного поиска суммы. Для заданного одномерного **статического** вещественного массива A из N элементов написать рекурсивную процедуру поиска суммы ее элементов.

а) $\sum_{i=1}^{N} A_i = A_1 + \sum_{i=2}^{N} A_i$	б) $\sum_{i=1}^{N} A_i = \sum_{i=1}^{K} A_i + \sum_{i=K+1}^{N} A_i$ k – индекс среднего элемента массива

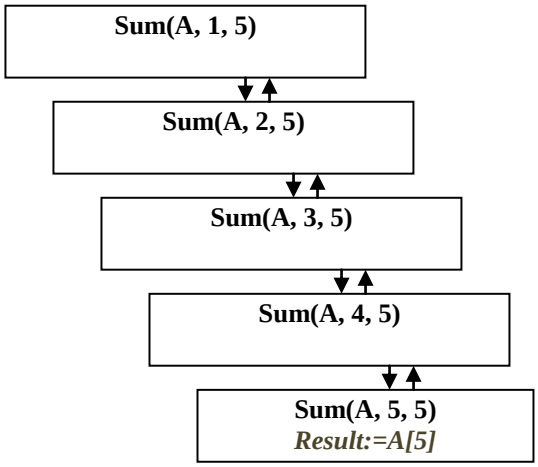
type mas = array [1..20] of real;
Function **Sum**(A: mas; i,N: byte): real;
Begin
  If i=N Then Result:=A[i]
  Else Result:=A[i]+**Sum**(A, i+1, N)
end;
 	
Вызов S:=**Sum(A, 1, 5)** с 1-го по 5-ый
	type mas = array [1..20] of real;
	Function **Sum**(A: mas; i1,i2: byte): real;
	Var k: byte;
	Begin
	  If i1=i2 Then Result:=A[i1]
	  Else Result:=**Sum**(A, i1, i1+(i2-i1) div 2)
	  + **Sum**(A, i1+(i2-i1) div 2 +1, i2)
	end; Вызов с 1-го по 5-ый: S:=**Sum(A, 1, 5)**

А) Линейная рекурсия	**Б) Каскадная** рекурсия

Для **5-ми** элементов:

Пусть A = (4, 2, -3, 1, -2)

S:=Sum(A, 1, 5) – Первый вызов



Глубина рекурсивных вызовов = 5

Объем рекурсии = 4

Для **8-ми** элементов будет:

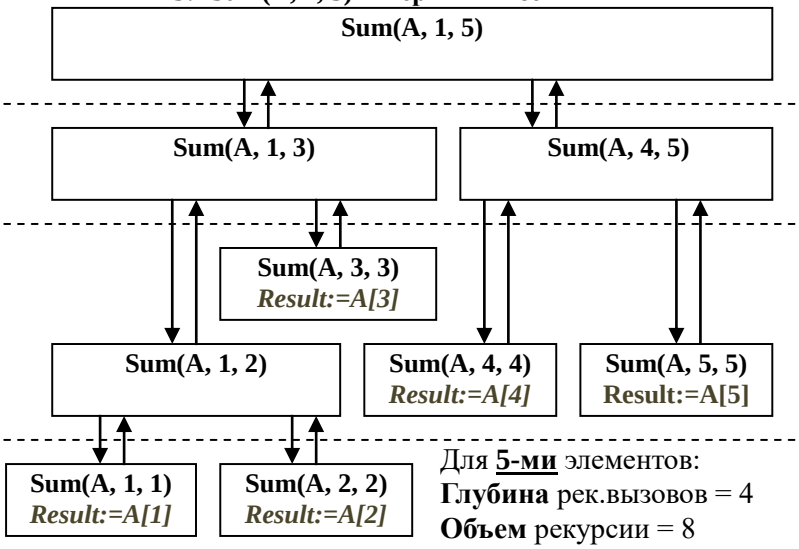
Глубина вызовов = **8**

Объем рекурсии = 7

Рост дерева в глубину

Для **5-ми** элементов: **Пусть A = (4, 2, -3, 1, -2)**

S:=Sum(A, 1, 5) – Первый вызов



Для **5-ми** элементов:

Глубина рек.вызовов = 4

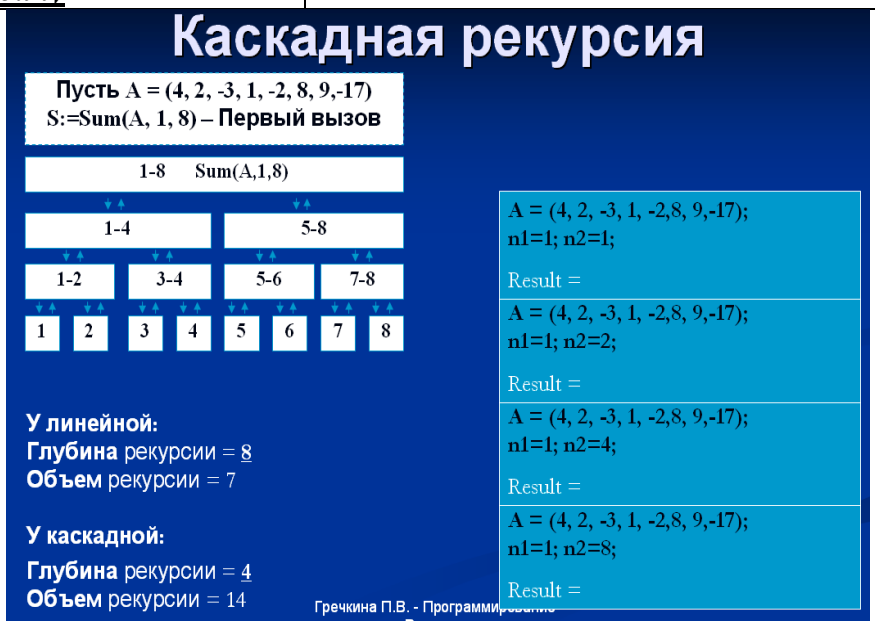
Объем рекурсии = 8

Для **8-ми** элементов будет:

Глубина рек.вызовов = 4

Объем рекурсии = **14** = 8+4+2

Рост дерева в ширину



Прямой и обратный ход рекурсии для рассмотренного выше примера ***линейной*** рекурсии для поиска суммы значений элементов одномерного массива:

Дерево вызовов	Прямой ход	Стек вызовов	Обратный ход
Для **5-ми** элементов: **Пусть A = (4, 2, -3, 1, -2)** **S:=Sum(A, 1, 5) – Первый вызов** **Sum(A, 1, 5)** ***Result:=A[1]+Sum(A, 2, 5)*** **Sum(A, 2, 5)** ***Result:=A[2]+Sum(A, 3, 5)*** **Sum(A, 3, 5)** ***Result:=A[3]+Sum(A, 4, 5)*** **Sum(A, 4, 5)** ***Result:=A[4]+Sum(A, 5, 5)*** **Sum(A, 5, 5)** ***Result:=A[5]***		**Result=?**	**Result=A[1]+-2=2**
		A=(4, 2, -3, 1, -2)	
	i ← i+1 =2	**i=1**	
		N=5	
		Result=?	**Result=A[2]+-4=-2**
		A=(4, 2, -3, 1, -2)	
		i=2	
	i ← i+1 =3	**N=5**	
		Result=?	**Result=A[3]+-1=-4**
		A=(4, 2, -3, 1, -2)	
		i=3	
	i ← i+1 =4	**N=5**	
		Result=?	**Result=A[4]+-2=-1**
		A=(4, 2, -3, 1, -2)	
		i=4	
	i ← i+1 =5	**N=5**	
		Result=?	**Result=A[5]=-2**
		A=(4, 2, -3, 1, -2)	
		i=5	
	Result=A[5]=-2	**N=5**	

Вычисление новых границ отрезка происходит ДО очередного рекурсивного вызова – прямой ход.

Результат вычисляется/передается ПОСЛЕ рекурсивного вызова – обратный ход.

Для сокращения количества параметров – в частности, чтобы каждый раз не передавать массив, можно сделать внешнюю процедуру со всеми параметрами и убрать «лишние» (неуправляющие) параметры у самой рекурсивной внутренней процедуры – тем самым уменьшить объем требуемой памяти в стеке вызовов при каждом вызове:

```
Type mas = array [1..20] of real; // Пример для статического массива
Function Sum(const A: mas; N: byte): real;
        Function Recursia(A: mas; i, N: byte): real; // A описана во внешней функции;
                                                     // здесь только управляющие параметры i и N
        Begin
          If i=N Then Result:=A[i]
          Else Result:=A[i]+ Recursia (i+1, N)
        end;
Begin
  Sum:=Recursia(1, N);
end;
```

Вызов в программе внешней процедуры
s:=**Sum(A, N)**

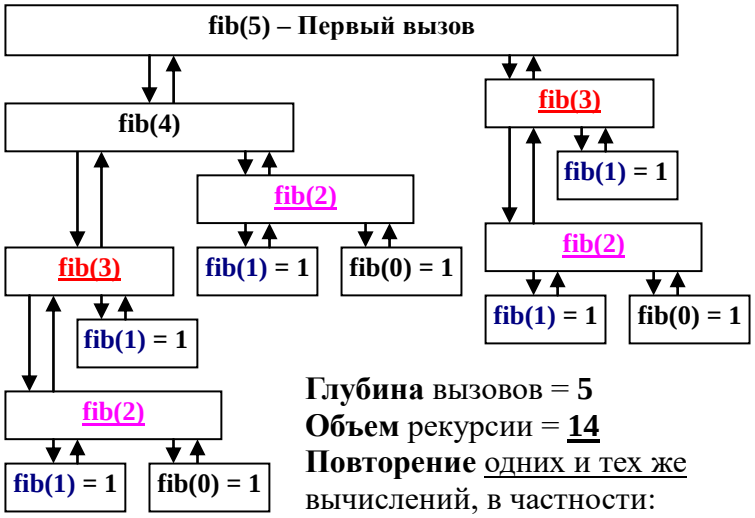
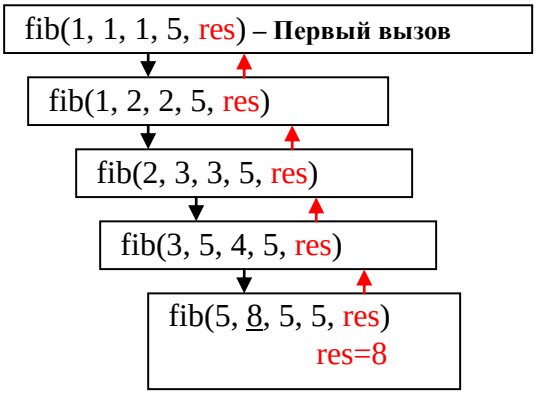
Так же для уменьшения размера стекового кадра желательно использовать как можно меньше локальных переменных в рекурсивной функции.

Пример. Найти N-ый член ряда Фибоначчи, если известно, что 0-ой и 1-ый его члены равны 1, а каждый следующий – сумма двух предыдущих:

N	0	1	2	3	4	5	6	7	8	9	…
Fib(N)	1	1	2 = 1+1	3 = 1+2	5 = 2+3	8 = 3+5	13 = 5+8	21 = 8+13	34	55	…

Каскадная рекурсия	**Линейная** рекурсия

Function fib(N: byte): cardinal;
Begin
  If N<2 then Result:=1
  Else Result:=fib(N-1)+fib(N-2);
End;
 	
Вызов
Res:=fib(**5**);
	Procedure fib(n1, n2, i, N: byte;
	var res: cardinal);
	Begin
	  If i>=N then Res:=n2
	  Else fib(n2, **n1+n2, i+1**, N, res);
	End;
	
	Вызов
	fib(1, 1, 1, **5**, res);

 **Глубина** вызовов = **5** **Объем** рекурсии = **14** **Повторение** одних и тех же вычислений, в частности: fib(3) – два, fib(2) и fib(0) по три, fib(1) – пять раз	 **Глубина** рекурсивных вызовов = **5** **Объем** рекурсии = **4**

	Линейная рекурсия	**Каскадная рекурсия**
Достоинства	Меньший объем рекурсии → быстрее	Меньшая глубина рекурсии → меньший расход памяти
Недостатки	Бо́льшая глубина рекурсии → больший расход памяти	Бо́льший объем рекурсии → медленнее Возможны *повторения* одних и тех же вычислений (как в методе Фибоначчи) → неэффективный расход ресурсов (проблема решается использованием динамической базы: один раз вычислив fib(3), запоминаем это решение → увеличение расхода памяти)

Взаимнорекурсивные функции
Прямая и косвенная рекурсия

$$F(x) = \begin{cases} x, & \text{если } x<1 \\ x * G(x-1), & \text{иначе} \end{cases}$$

$$G(x) = \begin{cases} 1, & \text{если } x<1 \\ x + F(x-1), & \text{иначе} \end{cases}$$

```
Function G( x: Integer): Integer; FORWARD;
```

```
Function F( x: Integer): Integer;
Begin
  If   x<1 then
     F := x
  Else
     F := x * G(x-1);
End;
```

```
Function G( x: Integer): Integer;
Begin
  If   x<1 then
     G := 1
  Else
     G := x + F(x-1);
End;
```

Гречкина П.В. - Программирование-Рекурсия

Для предварительного описания процедур используется заголовок со словом *forward*. Кроме того, при размещении описаний в модуле (*unit*), предварительные описания просто размещаются в разделе Interface без слова *forward*.