Алгоритмы

Алгоритмы

Алгоритм (algorithm) – это набор действий (система правил), определяющих порядок выполнения шагов для решения поставленной задачи.

Понятие алгоритма дополняется следующими свойствами:

1. Дискретность
2. Конечность
3. Корректность
4. Массовость
5. Определенность (детерминированность)
6. Производительность (эффективность)

Дискретность

Алгоритм состоит из последовательности отдельных шагов - элементарных действий, выполнение которых не представляет сложности.

Каждое действие, предусмотренное алгоритмом, исполняется только после того, как закончилось исполнение предыдущего.

(именно благодаря этому свойству алгоритм может быть реализован на ЭВМ).

Конечность

Алгоритм всегда должен заканчиваться после конечного числа шагов.

Например, алгоритм Евклида (нахождение наибольшего общего делителя двух целых чисел):

- найти остаток от деления (O = A%B);
- проверить его на равенство нулю (O == 0);
- если не равен, выполнить замену и повторить (A = B; B = O).

Если некоторая процедура (процесс) обладает всеми свойствами алгоритма, кроме конечности, то она называется методом.

Корректность

Алгоритм имеет некоторое количество входных (исходных) данных (величин), которые берутся из конкретного множества объектов.

Алгоритм имеет одну или несколько выходных величин (результат), которые имеют прямую зависимость от входных величин.

Т.е. при одних и тех же входных данных результат один и тот же (например: решение интеграла методом трапеций при заданных функции, диапазоне и точности, дает одно и тоже значение.

Массовость

Алгоритм можно быть предназначен не только для решения одной конкретной задачи, а и для решения любой задачи из некоторого класса однотипных задач при всех допустимых значениях исходных данных.

Например, алгоритм сортировки массива целых чисел размерностью 23 элемента (значения набора элементов могут быть одни и те же, но порядок их следования на входе – различным; при этом результат будет одним и тем же).

Определенность

Каждый шаг алгоритма должен быть строго определен (действия не должны быть двусмысленными)

Например, в алгоритме Евклида:

2) если остаток равен нулю (O == 0), то B наибольший общий делитель;

3) если не равен, выполнить замену (A = B;   B = O) и повторить первый шаг.

Существуют формальные языки описания алгоритмов во избежание неточного толкования действий, что приводит к механическому характеру действий.

Производительность

Все действия, которые необходимо выполнить алгоритму, должны быть достаточно простыми, чтобы их в принципе можно было выполнить точно и за конечный отрезок времени на бумаге.

Например, A=57, B=33.

1) 57 mod 33 = 24,
2) 33 mod 24 = 9,
3) 24 mod 9 = 6,
4) 9 mod 6 = 3,
5) 6 mod 3 = 0.

Применение алгоритмов

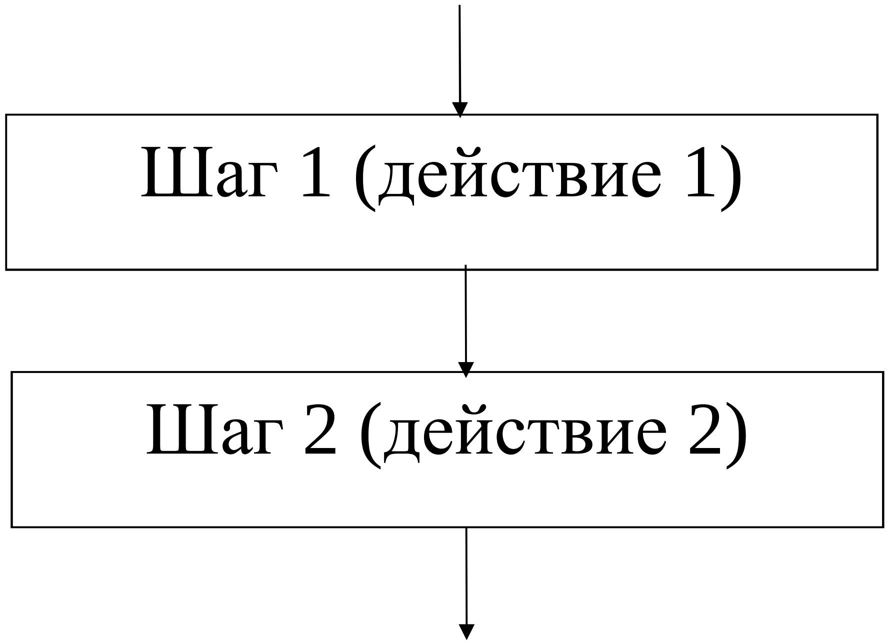
1. Задачи обработки данных (сортировка и поиск информации);
2. Задачи оптимизации (оптимальный маршрут);
3. Задачи защиты информации (криптография);
4. Вычислительные задачи (матрицы, интегралы, дифференциальные уравнения);
5. Задачи линейного программирования (предвыборная компания, нахождение минимальной стоимости авиабилетов для уменьшения количества свободных мест).

Алгоритмы

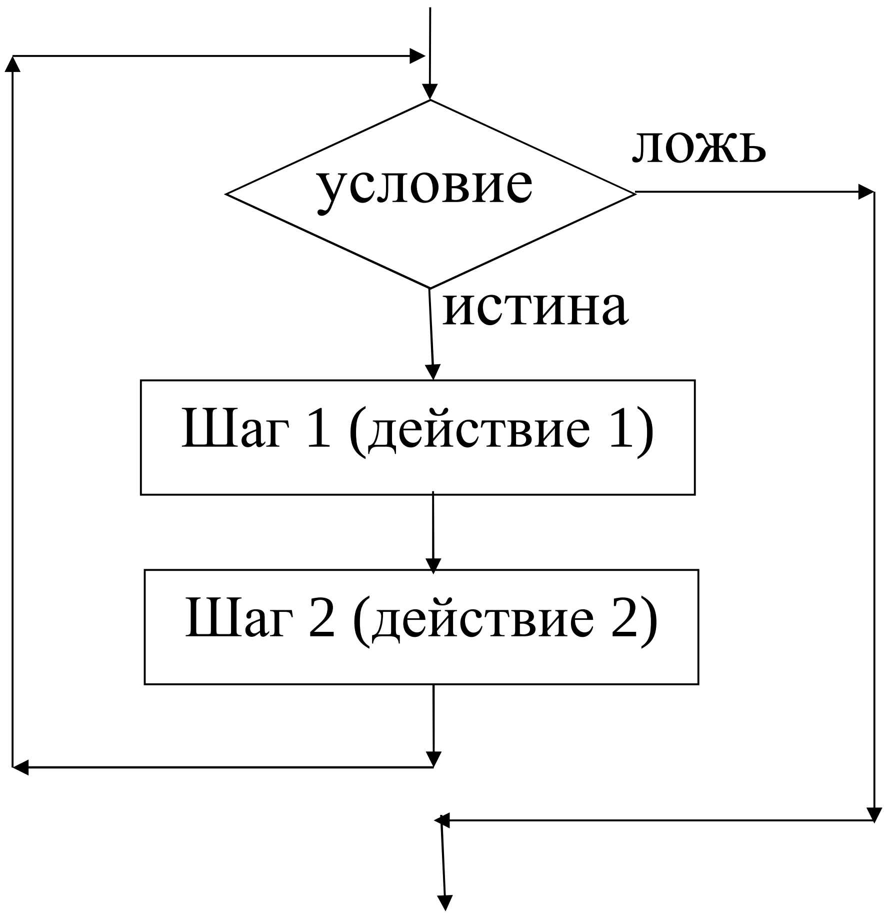
Виды алгоритмов

- Линейный
- Циклический
- Разветвляющийся
- Вспомогательный
- Рекурсивный

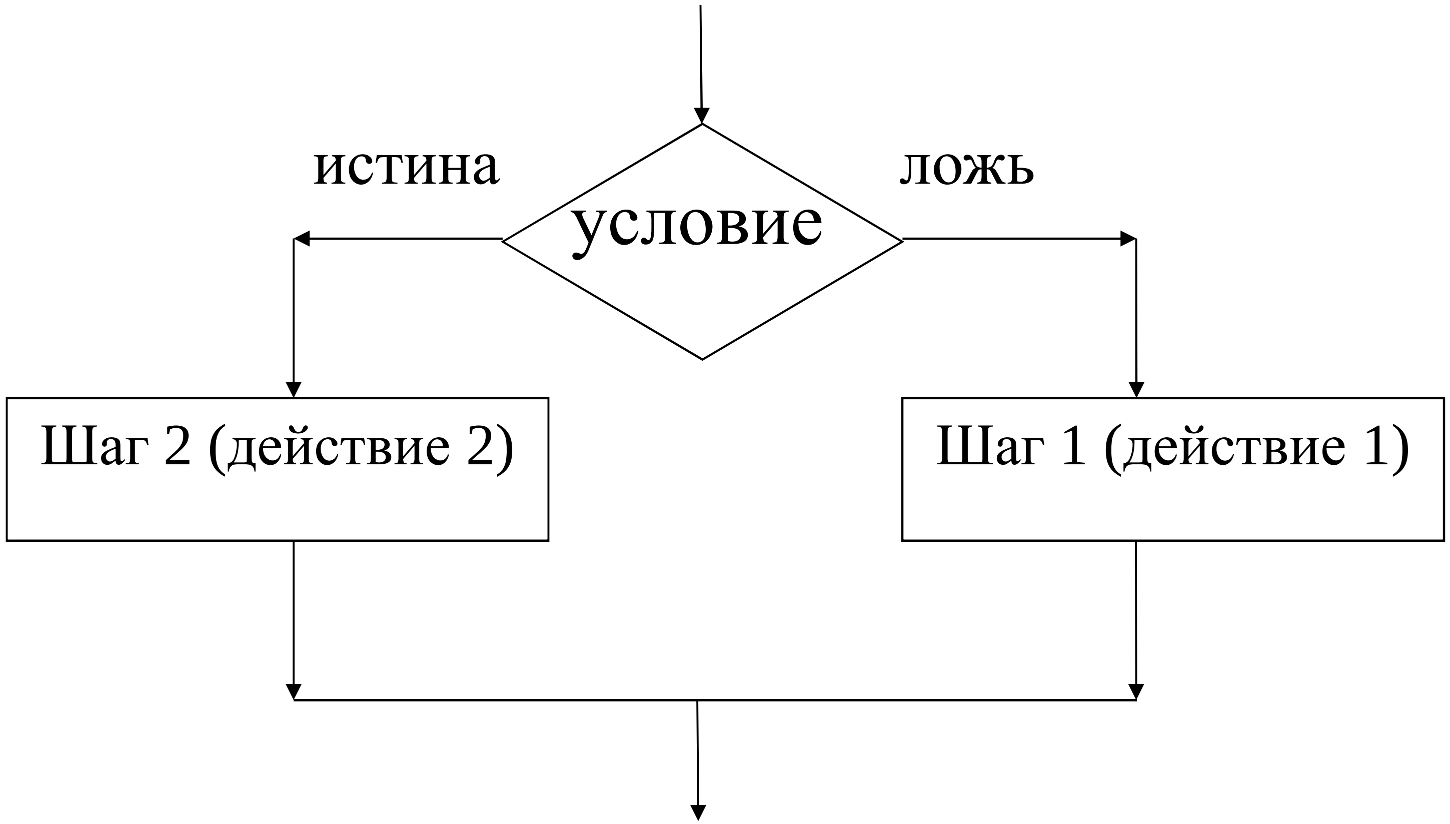
Линейный – все шаги алгоритма выполняются последовательно сверху вниз (нет ветвлений и повторов)



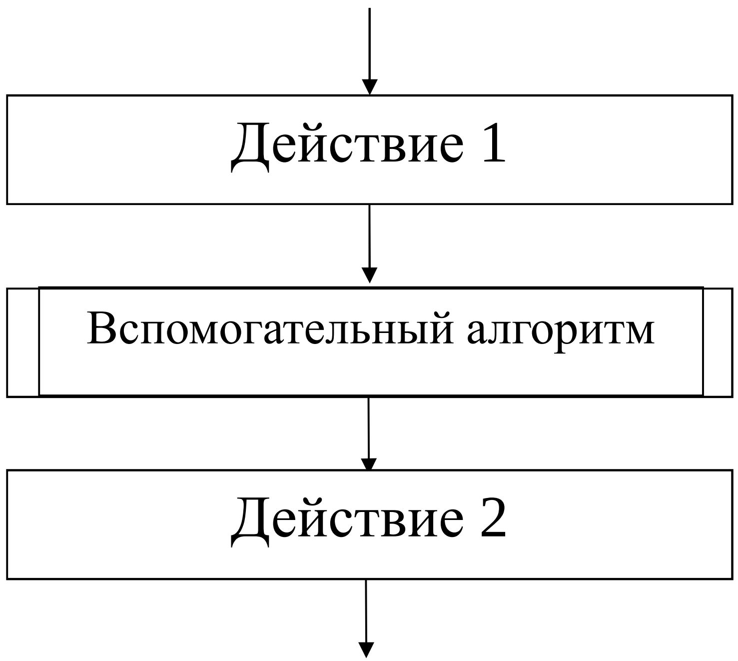
Циклический – часть шагов алгоритма повторяется некоторое количество раз;



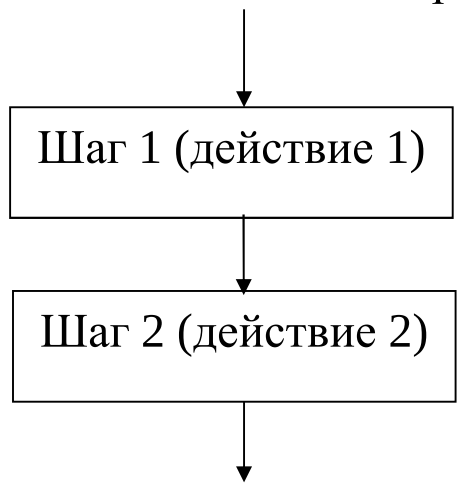
Разветвляющийся – в зависимости от выполнения некоторого условия выполняются разные шаги алгоритма



Вспомогательный – использование в качестве одного или более



Вспомогательный алгоритм



Рекурсивный – один из шагов алгоритма вызывает этот же алгоритм (еще раз вызывается сам алгоритм)

Рекурсивный алгоритм

РЕКУРСИВНЫЕ АЛГОРИТМЫ

Рекурсивный алгоритм – это алгоритм, который решает задачу путем решения нескольких более узких вариантов этой же задачи.

Рекурсивная функция – это функция, которая вызывает саму себя повторно с измененными входными параметрами.

Любое вычисление, которое предполагает выполнение циклов (повторных действий), можно реализовать посредством рекурсивных функций и наоборот.

Рекурсия используется для выражения сложных алгоритмов в компактной форме без потери производительности.

Особенности

1. Рекурсивная функция не может вызывать себя до бесконечности, т.е. должна иметь конечное количество вызовов.

2. Рекурсивная функция всегда должна иметь условие завершения, которое указывается первым действием.

3. Каждый последующий вызов осуществляется с уменьшением значения параметра функции.

Рекурсия

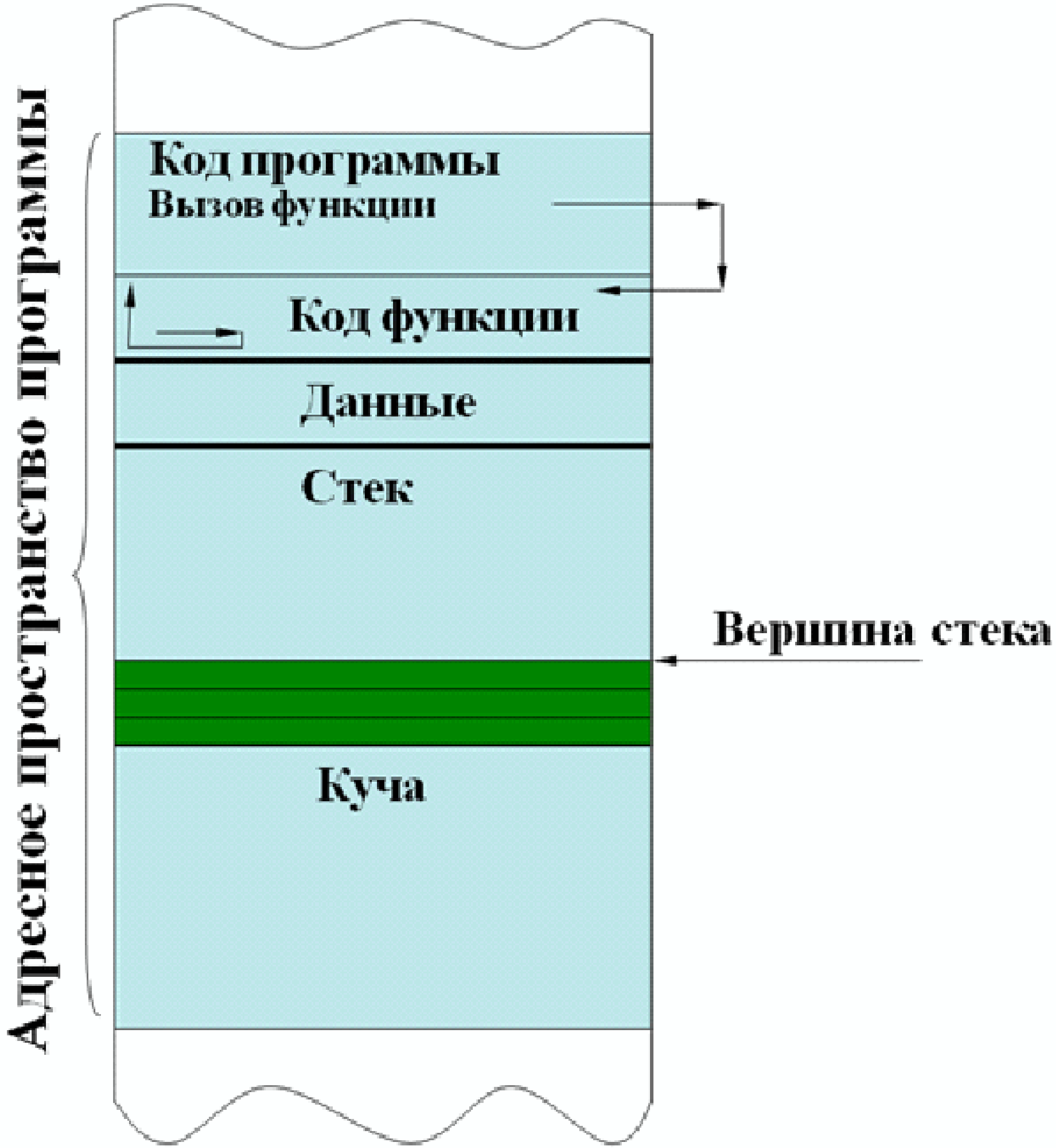
Вычисление факториала: $N! = N*(N-1)*(N-2)*...2*1$.

Factorial(*N*)
    if *N* = 1 **then**
        return 1
    temp $\leftarrow$ **call** Factorial(*N*-1)
    return *N* * *temp*

1) factorial(5) $\rightarrow$ 5*<u>**24**</u>=120
2)   factorial(4) $\rightarrow$ 4*<u>**6**</u>=24
3)   factorial(3) $\rightarrow$ 3*<u>**2**</u>=6
4)   factorial(2) $\rightarrow$ 2*<u>**1**</u>=2
5)   factorial(1) $\rightarrow$ 1

Представление рекурсии
в оперативной памяти

Рекурсия


Адресное пространство программы
Код программы
Вызов функции
Код функции
Данные
Стек
Вершина стека
Куча

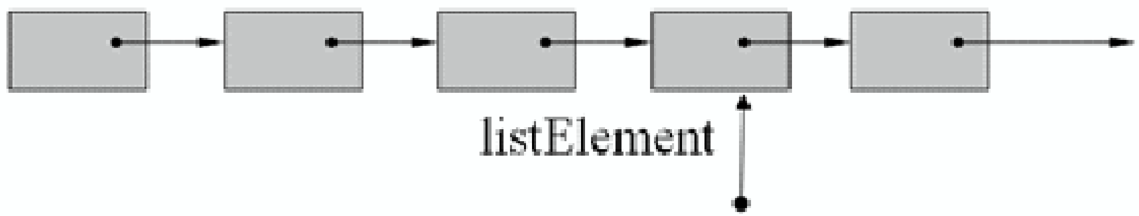
ПРИМЕРЫ

1) Рекурсивная функция реализующая алгоритм Евклида

```
Solve_Euclid(M, N)
    if  N = 0  then
        return M
    return call  Solve_Euclid (N, M % N)
```

Рекурсия

2) Рекурсивная функция для подсчета количества элементов однонаправленного списка



Count (*listElement*)

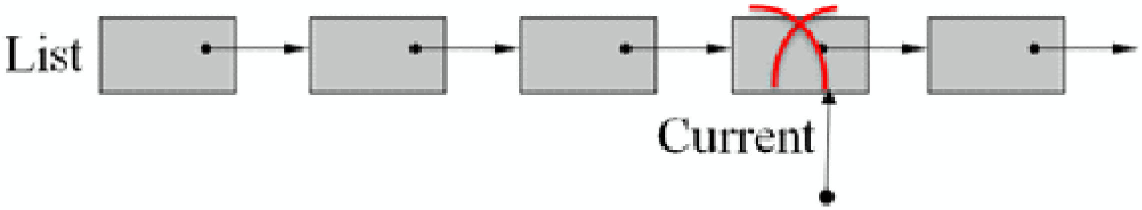
if *listElement* = NULL **then**

return 0

number ← **call** Count(*next*(listElement)) **return 1**

return 1 + *number*

3) Рекурсивная функция удаления элемента из однонаправленного списка



Delete (*value*)

List ← **call** Delete_Element (*List*, *value*)

Delete_Element (*Current*, *value*)

if *next*(Current) ≠ NULL **then**

next(Current) ← **call** Delete_Element (*next*(Current), *value*)

if *data*(Current) = *value* **then**

Current ← *next*(Current)

return *Current*

Задача «Ханойские башни»

Дано: три стержня и N дисков, которые имеют различный размер. Изначально все диски расположены на одном стержне в виде пирамиды (внизу наибольший, вверху – наименьший).

Необходимо переместить все диски на соседний стержень с учетом следующих правил:

1. За один шаг перемещается только один диск
2. Диск большего размера нельзя располагать над диском меньшего размера

348 столетий

(13 дисков, 1 диск – 1 секунда)

Hanoi (*N*, *direct*)

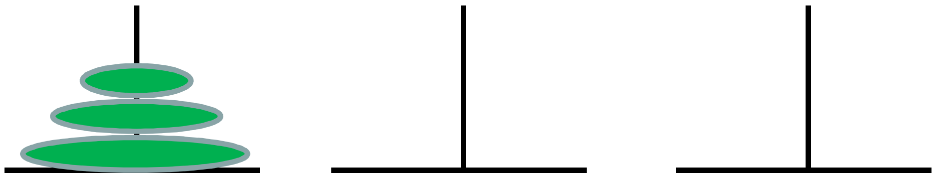
if *N* = 0 **then**

return

call Hanoi (*N*-1, -*direct*)

call Move (*N*, *direct*);

call Hanoi (*N*-1, -*direct*)



hanoi(3, +1)
hanoi(2, -1)
hanoi(1, +1)
hanoi(0, -1)
move(1, +1)
hanoi(0, -1)
move(2, -1)
hanoi(1, +1)
hanoi(0, -1)
move(1, +1)
hanoi(0, -1)
move(3, +1)
hanoi(2, -1)
hanoi(1, +1)
hanoi(0, -1)
move(1, +1)
hanoi(0, -1)

Рекурсивные алгоритмы для нелинейных структур данных

Алгоритмы обхода дерева

1) Последовательный (прямой)

(слева направо: обрабатывается левое поддерево текущего узла, затем сам узел, затем правое поддерево текущего узла)

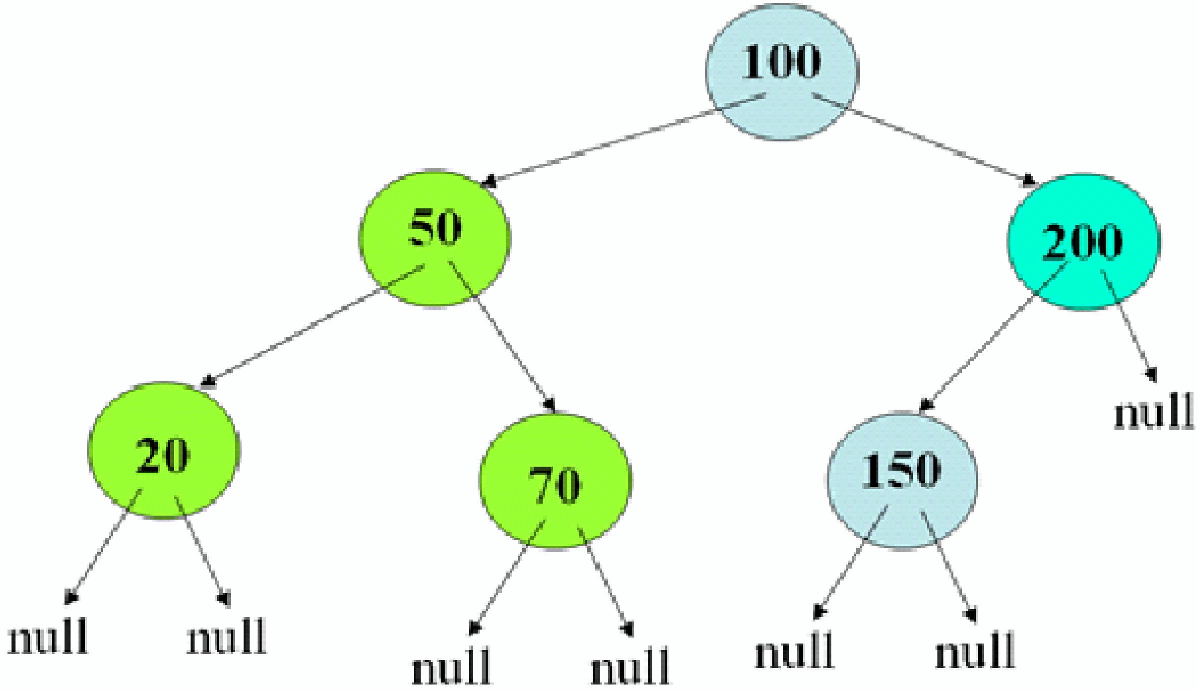
2) Параллельный (обратный)

(снизу вверх: обрабатывается левое поддерево текущего узла, затем правое поддерево текущего узла, затем сам узел)

3) В ширину

(сверху вниз: обрабатывается текущий узел, затем левое поддерево текущего узла, затем правое поддерево текущего узла)

Последовательный обход дерева



20 50 70 100

```
    // Алгоритм вывода узлов дерева
Print_Tree(Root)
    call  Print_Element_Tree(Root, 0)

    // алгоритм «Обход в ширину»
Print_Element_Tree(currentNode, level)
    if  currentNode  ≠  NULL   then
        for  i ← 0   to  level    do
            call  Print("    ")
        call  Print_Line( data(currentNode) )
        level   ←    level  + 1
        call Print_Element_Tree ( left(currentNode),  level)
        call Print_Element_Tree ( right(currentNode),  level)
    return   currentNode
```

Рекурсия

// алгоритм добавления нового узла

```
Insert_Node(Root, NewNode)
    if  Root = NULL  then
        Root ← NewNode
    else
        Root ← call Find_Position(Root, NewNode)
    return TRUE
```

// алгоритм поиска позиции нового узла

```
Find_Position(CurrentNode, NewNode)
  if data(CurrentNode) > data(NewNode)  then
      if  left(CurrentNode) = null  then
          left(CurrentNode) ← NewNode
      else
          left(CurrentNode) ← call Find_Position(left(CurrentNode), NewNode)
  else if  data(CurrentNode) < data(NewNode)  then
      if  right(CurrentNode) = null  then
             right (CurrentNode) ← NewNode
      else right(CurrentNode) ← call Find_Position(right(CurrentNode),
                                                               NewNode)
  return CurrentNode
```

Рекурсия

Результат формирования дерева из 9 узлов, значения которых сгенерированы случайным образом

```
Tree:
75
   22
      10
         21
            17
      67
   87
      83
      93
```

Алгоритм удаления узла

1. Найти удаляемый узел
2. Оценить имеет ли найденный узел левое и правое поддерево:

   - не имеет (ссылка обнуляется);

   - имеет либо левое, либо правое поддерево (ссылка переопределяется в соответствующее поддерево);

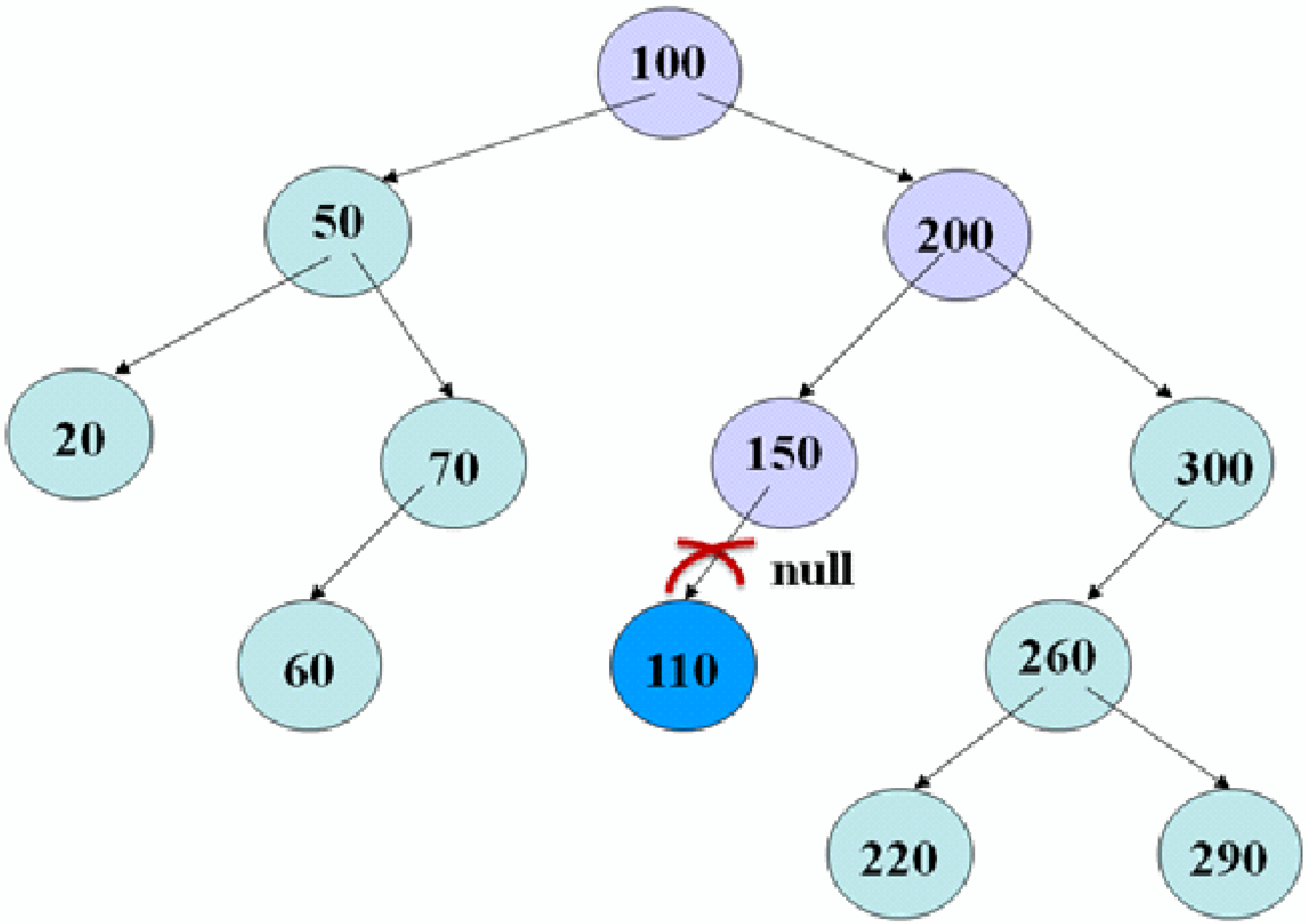
   - имеет оба поддерева (осуществляется поиск узла, стоящего на самом низшем уровне (лист) для замены удаляемого).

Существует два варианта поиска узла для замены:

а) самый крайний правый узел в левом поддереве;

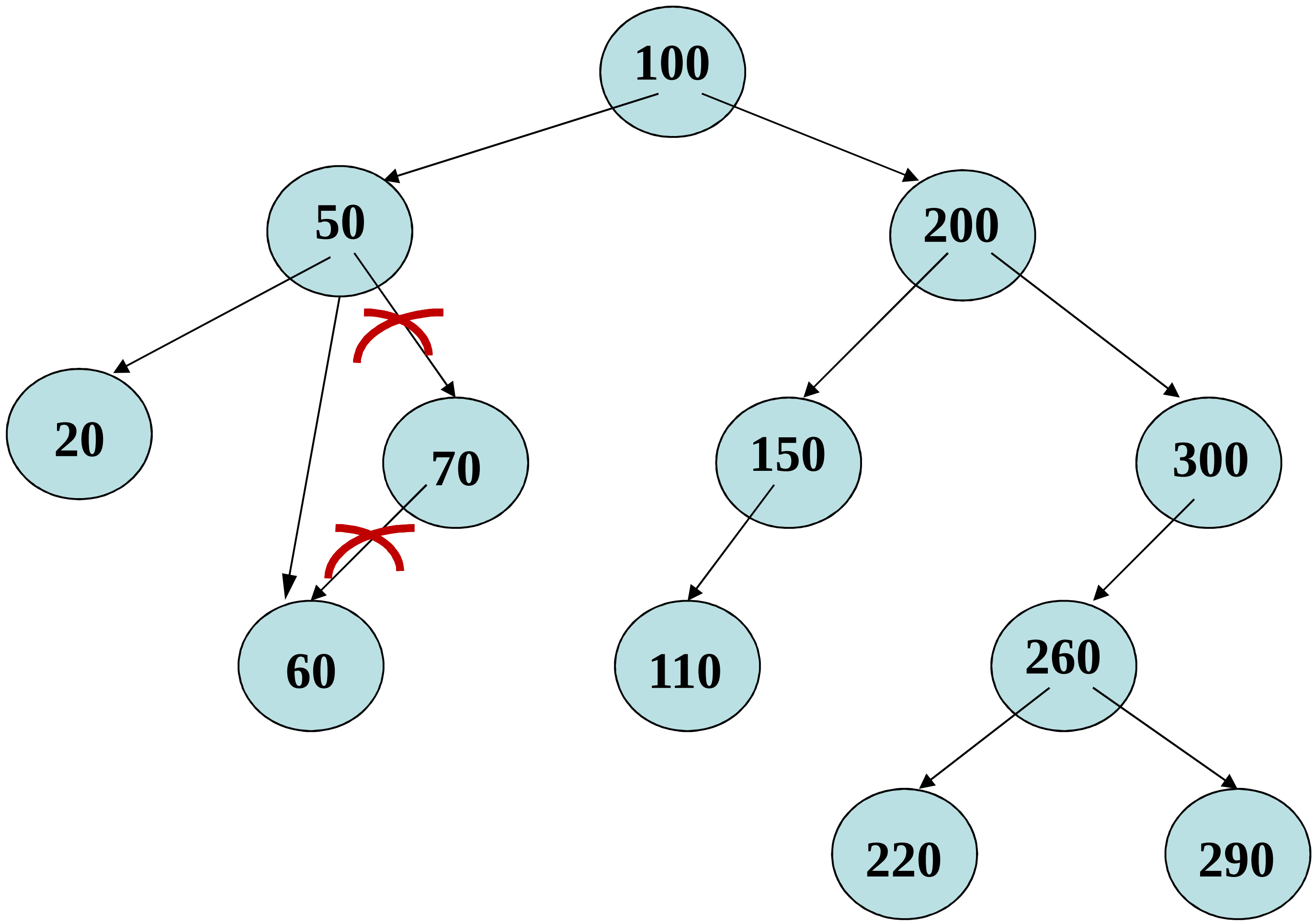
б) самый крайний левый узел в правом поддереве.

Удаление узла из дерева



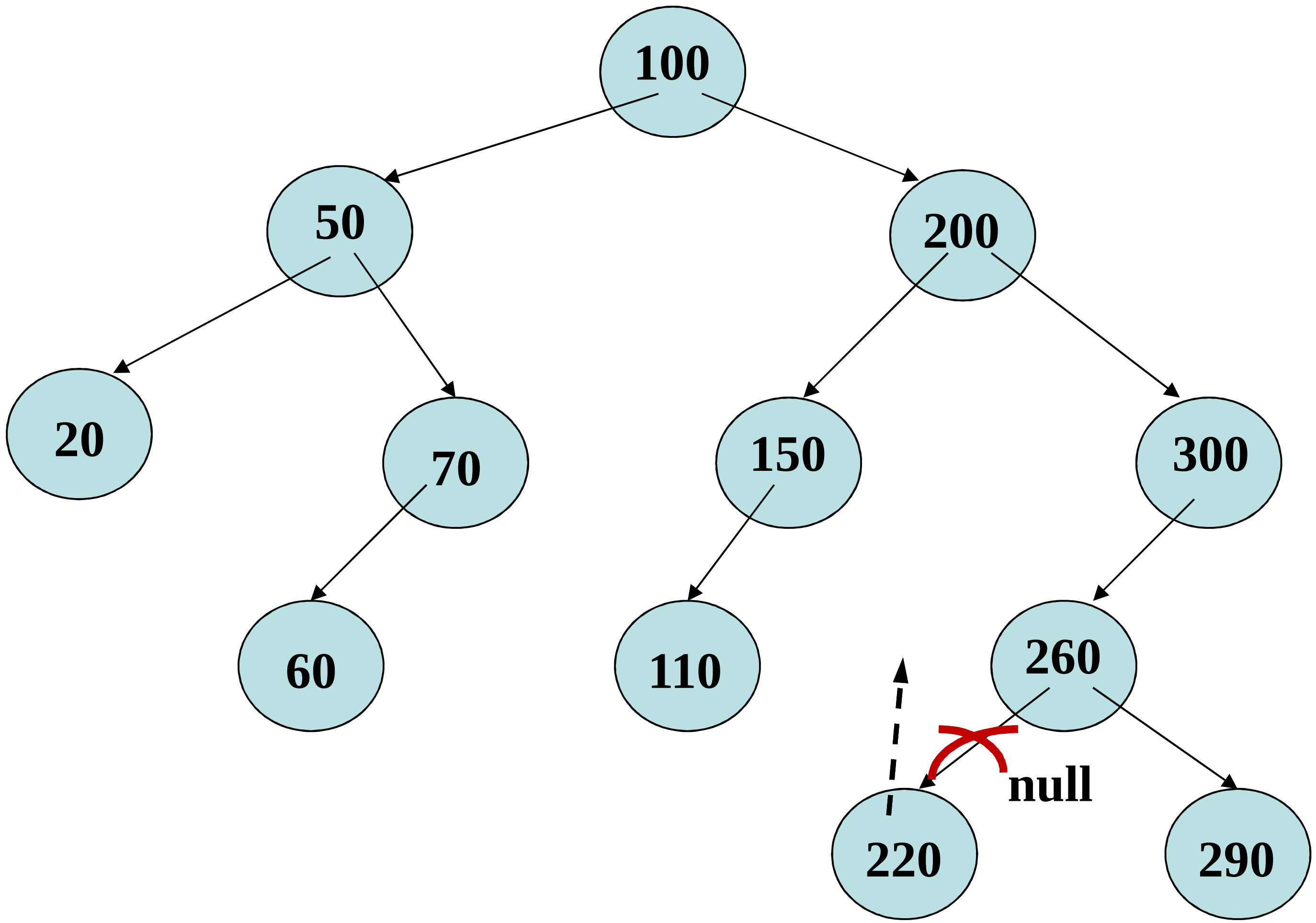
Допустим необходимо удалить узел со значением 110

Удаление узла из дерева



Допустим необходимо удалить узел со значением 70

Удаление узла из дерева



Допустим необходимо удалить узел со значением 200

Удаление узла из дерева

```
Delete(Root, value)                          // удаление из дерева
    Root ← call Find_Delete(Root, value)

Find_Delete(CurrentNode, value)              // алгоритм поиска узла для удаления
    if CurentNode ≠ NULL then
        if data(CurrentNode) = value then
            if left(CurrentNode) = NULL then
                CurrentNode ← right(CurrentNode)
                return CurrentNode
            else if right(CurrentNode) = NULL then
                    CurrentNode ← left(CurrentNode)
                    return CurrentNode
                else
                    right(CurrentNode) ← call Replace_Node(CurrentNode, right(CurrentNode))
                    return CurrentNode
        left(CurrentNode) ← call Find_Delete(left(CurrentNode), value)
        right(CurrentNode) ← call Find_Delete(right(CurrentNode), value)
    return CurrentNode
```

Удаление узла из дерева

// алгоритм поиска узла на замену удаляемому

Replace_Node(*DeleteNode*, *CurrentNode*)

if *left*(CurrentNode) $\neq$ NULL **then**

left(CurrentNode) $\leftarrow$ **call** Replace_Node(*DeleteNode*, *left*(CurrentNode))

else

data(DeleteNode) $\leftarrow$ *data*(CurrentNode)

CurrentNode $\leftarrow$ *right*(CurrentNode)

return *CurrentNode*

Удаление узла из дерева

Tree -> 10 node:

```
57   Удаляемый узел
  6
     46
        10
           41
        50
78
   59   Узел для замены
      63
         65
```

Tree for delete:

```
59
  6
     46
        10
           41
        50
  78
     63
        65
```

Enter for delete -> **57**

Динамическое программирование

Динамическое программирование – это подход к разработке алгоритмов, применяющих принцип «разделяй и властвуй» для задач с перекрывающимися подзадачами.

Виды динамического программирования:

1. Восходящее динамическое программирование

(для получения значения на i-м шаге требуется предварительное вычисление значений на предыдущих шагах)

2. Нисходящее динамическое программирование (значения, вычисленные на предыдущих шагах и необходимые для получения значения на i-м шаге, сохраняются и используются)

Последовательность чисел Фибоначчи

1) Восходящий

```
    numberElements ← call  Random(12)

Fibonacci (numberElements)
    call  Print(numberElements)              ► вывести количество элементов
    for  i ← 1  to  numberElements+1  do
        value ← call  Calculate(i)           ► вычислить текущий элемент
        call  Print(value)                   ► вывести полученное значение

Calculate(N)
    if  N = 0  then
        return 0
    if  N = 1  then
        return 1
    value_1  ← call  Calculate(N-1)
    value_2  ← call  Calculate(N-2)
    return  value_1 + value_2
```

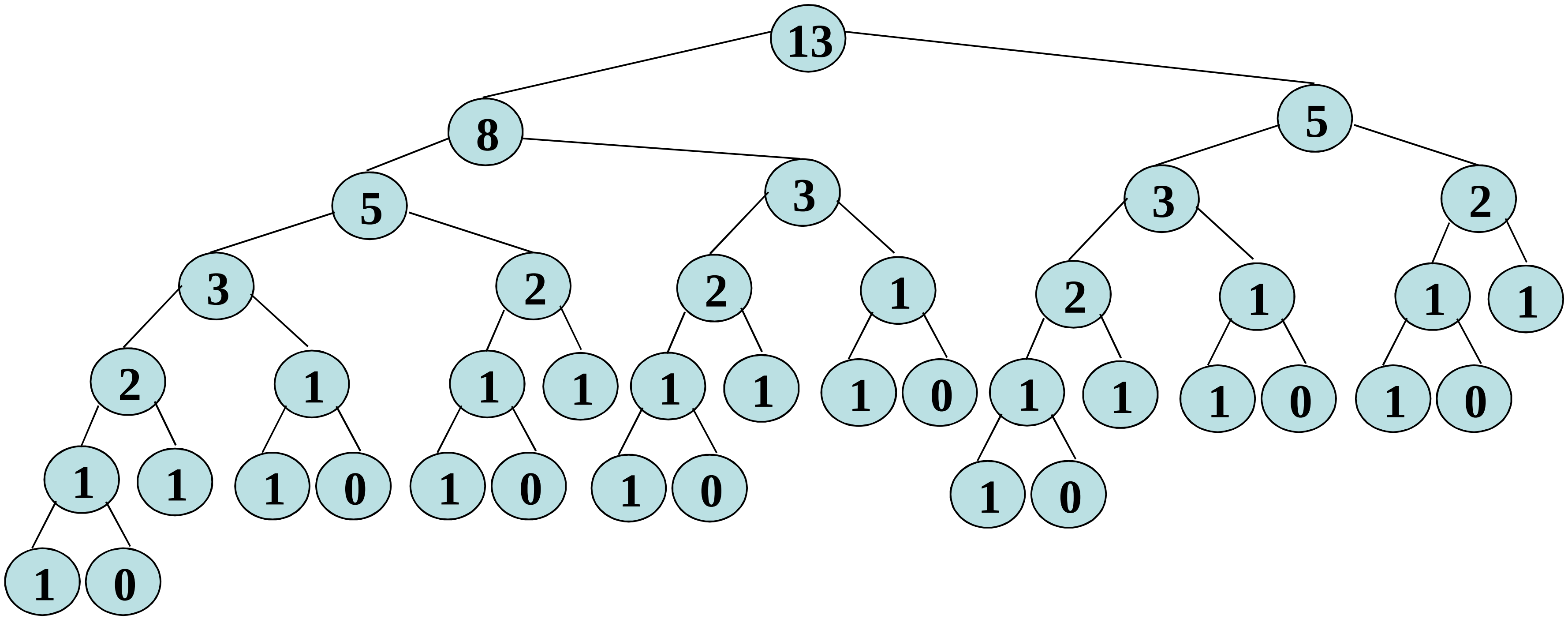
Рекурсия

N = 8

1 1 2 3 5 8 13 21

N = 9

1 1 2 3 5 8 13 21 34



2) Нисходящий

```
numberElements ← call Random(12)
► описать массив Array[] размерностью (numberElements+1)

Fibonacci (numberElements, Array)
    call Print(numberElements)              ► вывести количество элементов
    for i ← 1 to numberElements+1 do
        value ← call Calculate(i)           ► вычислить текущий элемент
        call Print(value)                   ► вывести полученное значение

Calculate(N)
    if Array[N] ≠ 0 then
        return Array[N]
    if N = 0 then
        return 0
    if N = 1 then
        Array[N] ← 1
    else
        value_1 ← call Calculate (N-1)
        value_2 ← call Calculate (N-2)
        Array[N] ← value_1 + value_2
    return Array[N]
```

Рекурсия

N = 7

1	1	2	3	5	8	13

N = 9

1	1	2	3	5	8	13	21	34