

Рекурсия, как прием программирования

В одном из фантастических рассказов Станислава Лема, автора "Соляриса"

"... Вычислительной машине поручили решить некую задачу. Машина была достаточно умна и ленива, и она решила сделать вычислительную машину, которой можно поручить решить эту задачу..."

Программирование. Рекурсия

Рекурсия – прием, при котором нечто выражается через такое же.

Примеры:

матем. факториал $N! = N \cdot (N-1)!$ $5! = 5 \cdot 4!$

Список чисел = элемент, Список чисел $2, 3, 4 = 2, \quad 3, 4$

Программа = (оператор, программа)

запись_числа = (запись_числа цифра) 3.1415926


Сумма элементов массива =

сложить(первый элемент, Сумма оставшихся)

$\text{Max}(a[0]..a[n]) = \text{Max}(a[0], \text{Max}(a[1]..a[n]))$

Рекурсия как математический прием часто применяется
в математике
в дискретной математике
в программировании

Рекурсия применяется
для построения и описания данных
для формулирования методов решения задач
для построения алгоритмов и программ



Важно: должен быть предусмотрен вариант, в котором рекурсия не нужна. Например, $1! = 1$, список из одного числа, число из одной цифры, суммирование элементов массива, в котором один элемент.

Наиболее успешные результаты получаются, если рекурсия применяется как для определения данных, так и для решения задач с этими данными

Рекурсивное определение целого числа (| обозначает "или")

(цифра) ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 //нерекурсивная часть

(число) ::= (цифра) | //нерекурсивная часть

(цифра)(число) | (число) (цифра) //рекурсивная часть

Рекурсивное определение списка элементов (пусть элементами будут числа)

(Список) ::= (элемент) | //нерекурсивная часть,

(Список)(запятая)(элемент) | //рекурсивная часть

(элемент)(запятая)(Список) //рекурсивная часть

Если есть некоторый список, то можно получить новый список, приписав элемент слева или справа

Пример

Список 23, 35, 74, 2, 11 - это

элемент 23 запятая Список 35, 74, 2, 11

Список 35, 74, 2, 11 – это

элемент 35 запятая Список 74, 2, 11

Список 74, 2, 11 – это

Элемент 74 запятая Список 2, 11

Список 2, 11 - это

элемент 2 запятая Список 11

Список 11 – это элемент, т.е. частный случай списка

Определение факториала:

$N! ::= 1$ если $N == 0$ или $N == 1$
 $::= N * (N-1)!$ если $N > 1$

Вычисление 5!

$$5! = (N > 1) \ 5 * 4!,$$

$$4! = (N > 1) \ 4 * 3!,$$

$$3! = (N > 1) \ 3 * 2!$$

$$2! = (N > 1) \ 2 * 1!$$

$$1! = 1$$

$$2! = 2 * 1$$

$$3! = 3 * 2 * 1$$

$$4! = 4 * 3 * 2 * 1$$

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

Итак,

Рекурсия – это процесс определения чего-то в терминах самого себя.

В программировании рекурсия обеспечивается возможностью для подпрограммы (процедуры или функции) вызвать саму себя.


```

program Factorial;
  var   i:integer;
  function fact_r (n: integer):longint;
    {n! - рекурсивный метод}
    var   f: longint;
    begin
      f:=1;
      if (n>1) then f:=n*fact_r(n-1);
      fact_r:=f;
    end;
  function fact_i (n: integer):longint;
    {n! - итерационный метод}
    var   f: longint; i: integer;
    begin
      f:=1;
      for i:=2 to n do f:=f*i;
      fact_i:=f;
    end;
begin {головная программа}
  for i:=1 to 15 do
    writeln (i,'!= ', fact_r (i),' ',fact_i (i));
end.

```

$$n! = n*(n-1)!$$

Окно вывода

1! = 1 1

2! = 2 2

3! = 6 6

4! = 24 24

5! = 120 120

6! = 720 720

7! = 5040 5040

8! = 40320 40320

9! = 362880 362880

10! = 3628800 3628800

11! = 39916800 39916800

12! = 479001600 479001600

13! = 1932053504 1932053504

14! = 1278945280 1278945280

15! = 2004310016 2004310016

Longint (4 байта) :

-2 147 483 648...+2 147 483 647

13! = 6 227 020 800

Результат не
поместился в
разрядной
сетке
Неверно!

Рекурсия при вычислении суммы элементов массива

$$S = \sum_{i=0}^{N-1} A[i] = A[0] + \left(\sum_{i=1}^{N-1} A[i] \right)$$

```
program Summa;  
  const n=100;  
  type  
    tarr=array[1..n] of integer;  
  var  
    A:tarr;  
    i,m:integer;
```

```
function arr_sum_r (var B:tarr; first, last:integer): integer;  
  {Сумма элементов массива с номерами от first до last -  
  рекурсивный метод}  
  var  
    res: integer;  
  begin  
    res:=0; {если first > last, метод вернет 0}  
    if first<=last {если есть, что суммировать}  
      then if first=last  
        then res:=B[first] {нерекурсивная часть}  
        else {рекурсивная часть:}  
          res:=B[first]+ arr_sum_r (B, first+1,last);  
      arr_sum_r:=res;  
  end;
```

```
function arr_sum_i (var B:tarr; first, last:integer):integer;  
  {Сумма элементов массива с номерами от first до last -  
  итерационный метод}
```

```
  var  
    res,i: integer;  
  begin  
    res:=0; {если first > last, метод вернет 0}  
    for i:=first to last do res:=res+B[i];  
    arr_sum_i:=res;  
  end;
```

```
Begin {головная программа}  
  m:=10; {В массиве будет 10 элементов}  
  writeln('Массив заполнен числами:');  
  for i:=1 to m do  
    begin  
      A[i]:=i+1; {заполнение массива}  
      write(' ',A[i]); {вывод значения элемента}  
    end;  
  writeln;  
  writeln ('Sum= ',arr_sum_r (A,1,m), ' = ', arr_sum_i (A,1,m));  
end.
```

Окно вывода

Массив заполнен числами:

2 3 4 5 6 7 8 9 10 11

Sum= 65 = 65

Рекурсия при вычислении минимума

$$m = \min(A[0], A[1], \dots, A[N - 1]) = \min(A[0], \min(A[1], \dots, A[N - 1]))$$

```
program Minimum;  
  const n=100;  
  type  
    tarr = array[1..n] of integer;  
  var  
    A:tarr;  
    i, m: integer;
```

function **arr_min_r** (var B:tarr; **first**, last:integer):integer;
*{Нахождение минимума среди элементов массива с
номерами от first до last - рекурсивный метод}*

var

res, tempmin: integer;

begin

res:=0; *{если first > last, метод вернет 0}*

if first<=last *{если есть, что проверять}*

then if first=last

then res:=B[first] *{нерекурсивная часть}*

else *{рекурсивная часть:}*

begin

tempmin:=**arr_min_r**(B,**first+1**,last);

if B[first]<tempmin

then res:=B[first]

else res:=tempmin

end;

arr_min_r:=res;

end;

function arr_min_i(var B:tarr; first, last:integer):integer
{Нахождение минимума среди элементов массива с номерами от first до last - итерационный метод}

var

res,i: integer;

begin

res:=0; *{если first > last, метод вернет 0}*

if first<=last *{если есть, что проверять}*

then

begin

res:=B[first];

for i:=first+1 to last do

if B[i]<res then res:=B[i]

end;

arr_min_i:=res;

end;

```
begin
  m:=8; {В массиве будет 10 элементов}
  A[1]:=10; A[2]:=3; A[3]:=5; A[4]:=1;
  A[5]:=6; A[6]:=4; A[7]:=2; A[8]:=8;
  writeln('Массив заполнен числами:');
  for i:=1 to m do
    write(' ', A[i]); {вывод значения элемента}
  writeln;
  writeln ('Min= ',arr_min_r(A,1,m),
           ' = ', arr_min_i(A,1,m));
end.
```

Окно вывода

```
Массив заполнен числами:
 10  3  5  1  6  4  2  8
Min = 1 = 1
```

Идея быстрой сортировки

Быстрая сортировка ([англ. quicksort](#)), часто называемая qsort по имени реализации в стандартной библиотеке языка [Си](#) — широко известный [алгоритм сортировки](#), разработанный английским информатиком [Чарльзом Хоаром](#) во время его работы в [МГУ](#) в [1960 году](#). Один из самых быстрых известных универсальных алгоритмов сортировки массивов (в среднем $O(n \log n)$ обменов при упорядочении n элементов); из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «Пузырьковая сортировка» и «Шейкерная сортировка»), известного, в том числе, своей низкой эффективностью. **Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы. Любопытный факт: улучшение самого неэффективного прямого метода сортировки дало в результате один из наиболее эффективных улучшенных методов.**

Общая идея алгоритма состоит в следующем:

Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива или же число, вычисленное на основе значений элементов (от выбора этого числа существенно зависит эффективность алгоритма). Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: «меньшие опорного», «равные» и «большие».

Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

На практике массив обычно делят не на три, а на две части - например, «меньшие опорного» и «равные и большие». Такой подход в общем случае эффективнее, так как упрощает алгоритм разделения

Хоар разработал этот метод применительно к машинному переводу; словарь хранился на магнитной ленте, и сортировка слов обрабатываемого текста позволяла получить их переводы за один прогон ленты, без перемотки её назад. Алгоритм был придуман Хоаром во время его пребывания в Советском Союзе, где он обучался в Московском университете компьютерному переводу и занимался разработкой русско-английского разговорника.

Выбор опорного элемента:

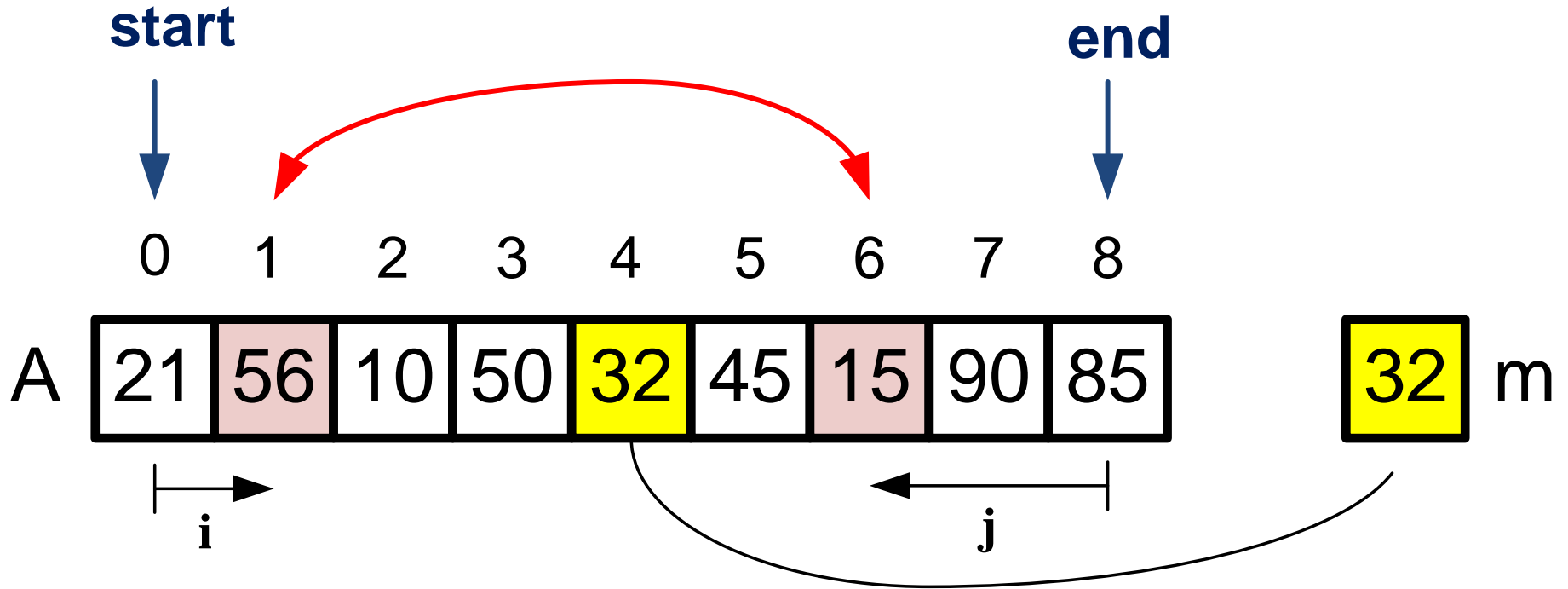
С точки зрения корректности алгоритма выбор опорного элемента безразличен. С точки зрения повышения эффективности алгоритма выгоднее всего выбирать медиану; но без дополнительных сведений о сортируемых данных её обычно невозможно получить.

Для справки:

Медиана — возможное значение признака, которое делит ранжированную совокупность на две равные части: 50 % «нижних» единиц ряда данных будут иметь значение признака не больше, чем медиана, а «верхние» 50 % — значения признака не меньше, чем медиана.

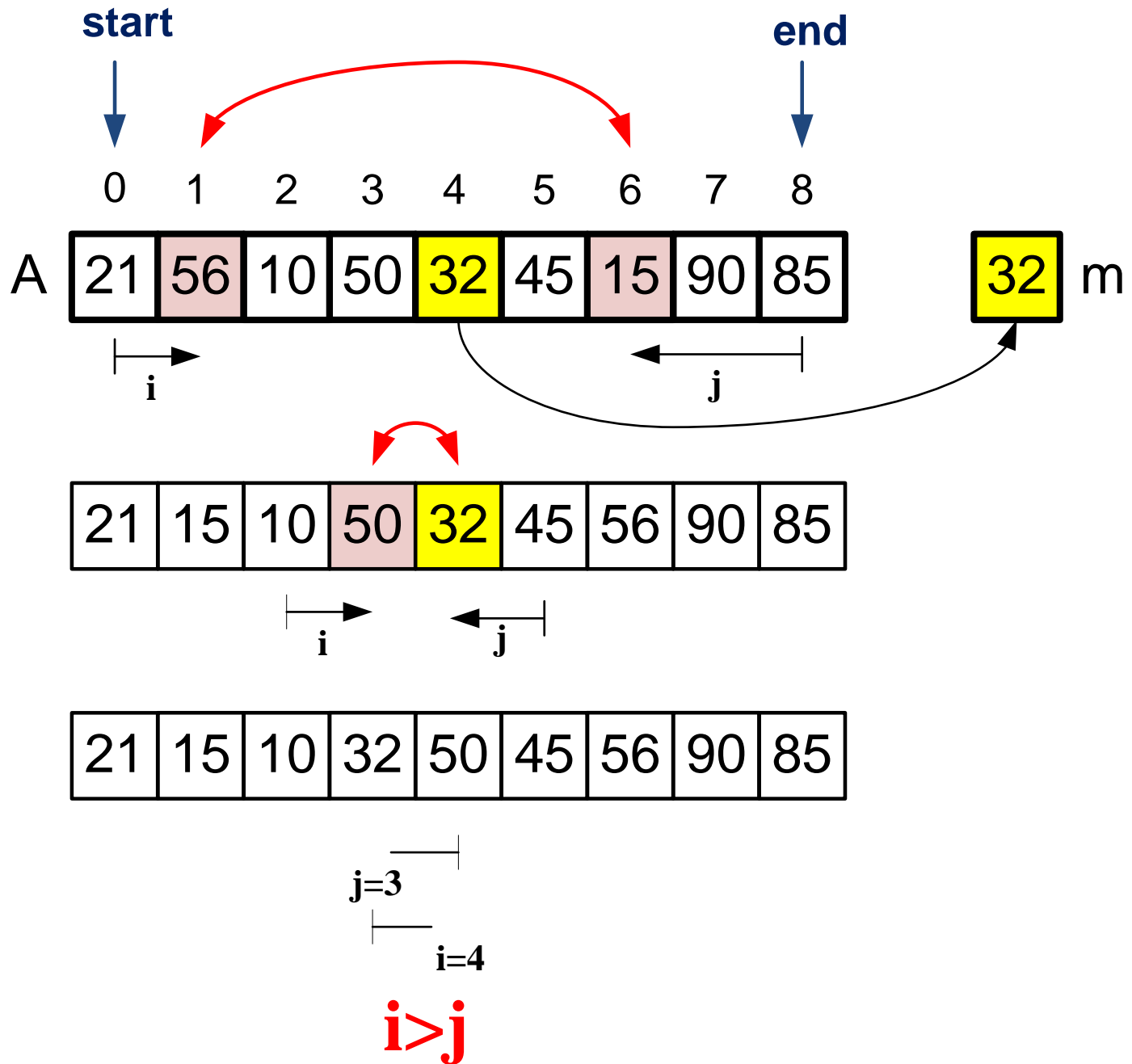
Известные стратегии: **выбирать постоянно один и тот же элемент**, например, **средний** или последний по положению; выбирать элемент со случайно выбранным индексом.

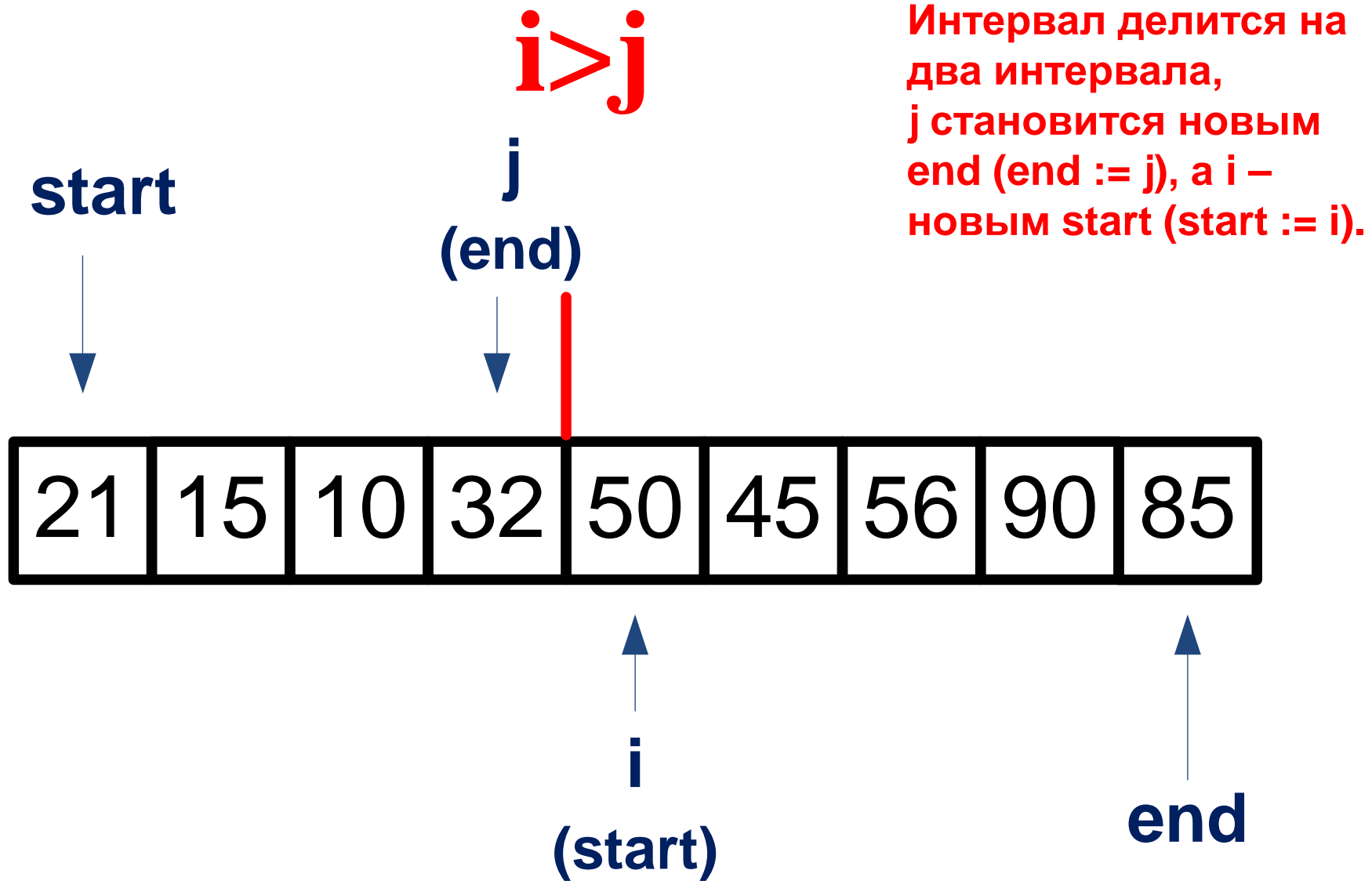
Часто хороший результат даёт выбор в качестве опорного элемента среднего арифметического между минимальным и максимальным элементами массива, особенно для целых чисел (в этом случае опорный элемент не обязан быть элементом сортируемого массива).

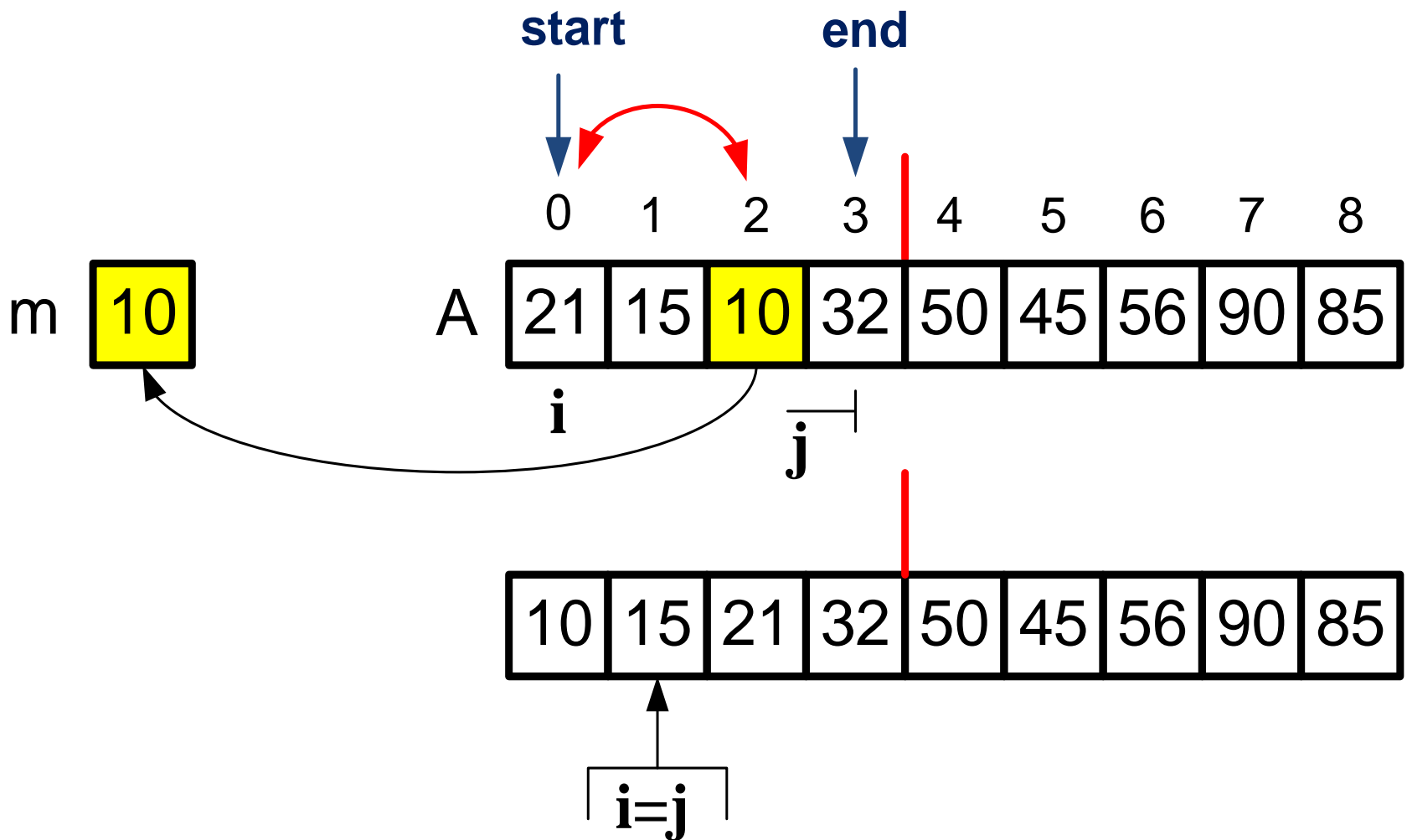


while $A[i] < m$ $i := i + 1$;

while $A[j] > m$ $j := j - 1$;



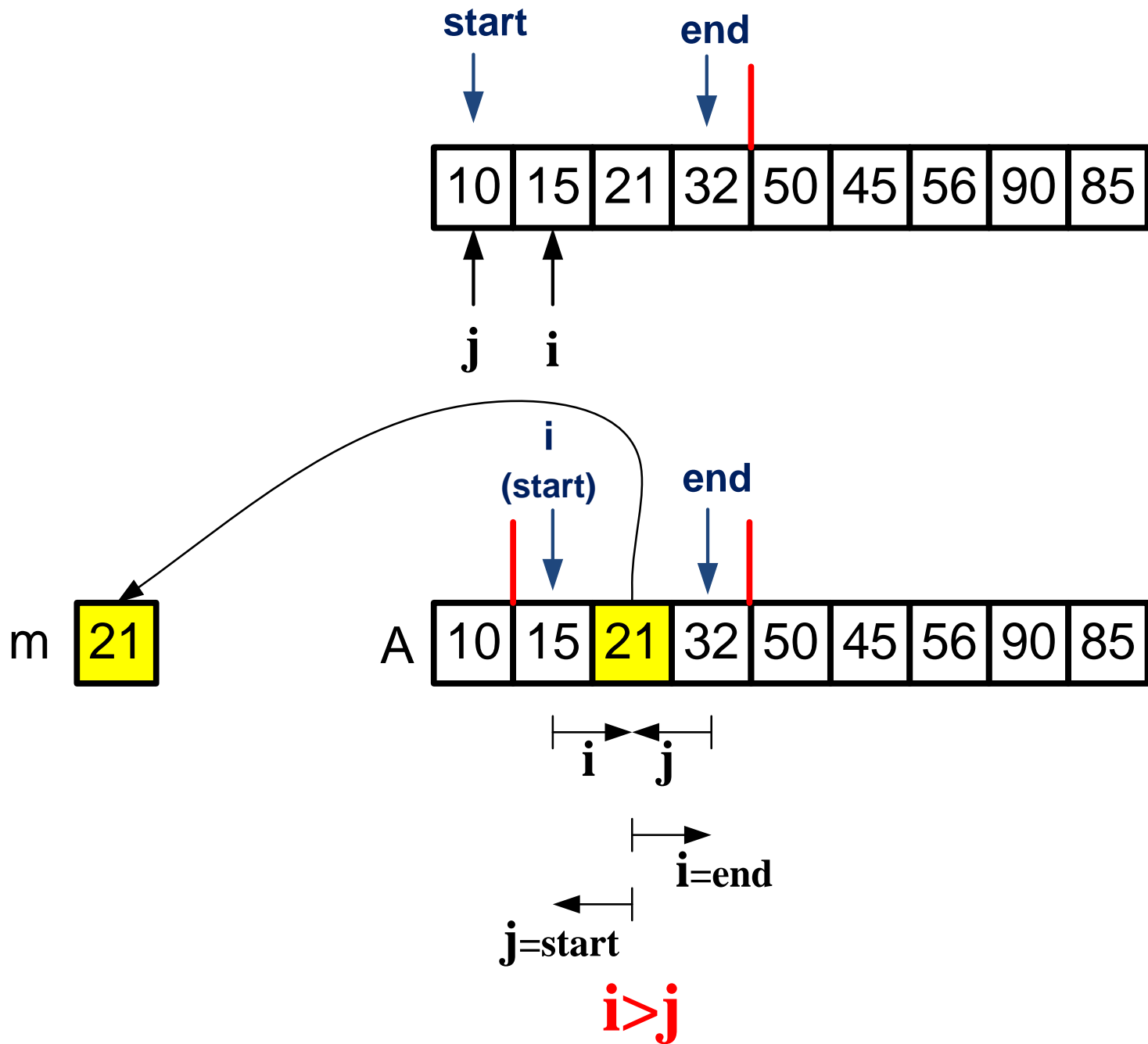




Левая часть массива упорядочена,
но как программа об этом узнает?

$A[i] > m \rightarrow i$ не увеличивается

$A[j] > m \rightarrow j = j - 1$



**Признак того, массив (часть массива)
отсортирован (дальнейшее деление
невозможно):**

1) $i > j$;

2) $j = \text{start}$;

3) $i = \text{end}$.

**Работу метода по упорядочиванию
правой части массива промоделировать
самостоятельно.**

```
program QuikSort;  
  const n=100;  
  type  
    tarr=array[0..n-1] of integer;  
  var  
    A,B: tarr;  
    i,m: integer;
```



```

procedure q_sort (var A:tarr; first, last:integer);
  {Быстрая сортировка методом Хоара}
  var i,j,k,b: integer;
    swapNum: integer; {число перестановок}
    m:integer; {опорный элемент}
begin
  i:=first; j:=last;
  swapNum:=0; {число перестановок}
  k:=(i+j+1)div 2;
  m:=A[k]; {опорный элемент}
  repeat
    while (A[ i ]<m) do i:=i+1;
    while (A[ j ]>m) do j:=j-1;
    if (i<=j) then
      begin
        b:=A[ i ]; A[ i ]:=A[ j ]; A[ j ]:=b; {перестановка}
        swapNum:=swapNum+1;
        i:=i+1; j:=j-1;
      end
    until (i>j);
    if (first<j) then q_sort (A, first, j); {рекурсивный вызов}
    if (i<last) then q_sort (A,i,last); {рекурсивный вызов}
    writeln ('Число перестановок:', swapNum);
end;

```

Дополнительные
операторы добавлены
для проведения
исследований

```

procedure sort_vial_1(var A: tarr; n: integer);
  {Сортировка первым методом пузырька}
var
  j, m, b: integer;
  flag:boolean;
  swapNum: integer; {число перестановок}
begin
  for m := n-1 downto 1 do
    begin
      flag:=true;
      for j:=0 to m-1 do
        if A[j]>A[j+1]
          then {перестановка}
            begin
              b:=A[j];
              A[j]:=A[j+1];
              A[j+1]:=b;
              flag:=false;
              swapNum:=swapNum+1;
            end;
          if flag then break;
        end;
      Writeln('Число перестановок:', swapNum);
    end;
end;

```

Дополнительные
операторы добавлены
для проведения
исследований

```
procedure putArr(var A: tarr; n:integer);
```

```
{Вывод массива в одну строку}
```

```
var i:integer;
```

```
begin
```

```
for i:=0 to n-1 do write(A[i]+' ');
```

```
writeln;
```

```
end;
```

```
begin {головная программа}
```

```
m:=9; {В глобальных массивах A и B будет по 9 элементов;  
массивы A и B – одинаковые:}
```

```
A[0]:=25; A[1]:=31; A[2]:=28; A[3]:=42;
```

```
A[4]:=46; A[5]:=39; A[6]:=25; A[7]:=35; A[8]:=20;
```

```
B[0]:=25; B[1]:=31; B[2]:=28; B[3]:=42;
```

```
B[4]:=46; B[5]:=39; B[6]:=25; B[7]:=35; B[8]:=20;
```

```
writeln('До сортировки:');
```

```
putArr(A,m); writeln;
```

```
{сортировка методом пузырька}
```

```
writeln('Метод пузырька'); sort_vial_1(A,m);
```

```
writeln ('После сортировки методом пузырька:');
```

```
putArr(A,m); writeln;
```

```
{быстрая сортировка методом Хоара}
```

```
writeln('Метод быстрой сортировки Хоара'); q_sort (B,0,m-1);
```

```
writeln('После сортировки методом Хоара:');
```

```
putArr(B,m); writeln end.
```

Окно вывода

До сортировки:

25 31 28 42 46 39 25 35 20

Метод пузырька

Число перестановок:19

После сортировки методом пузырька:

20 25 25 28 31 35 39 42 46

Метод быстрой сортировки Хоара

Число перестановок:1

Число перестановок:1

Число перестановок:1

Число перестановок:1

Число перестановок:2

Число перестановок:2

Число перестановок:1

Число перестановок:1

После сортировки методом Хоара:

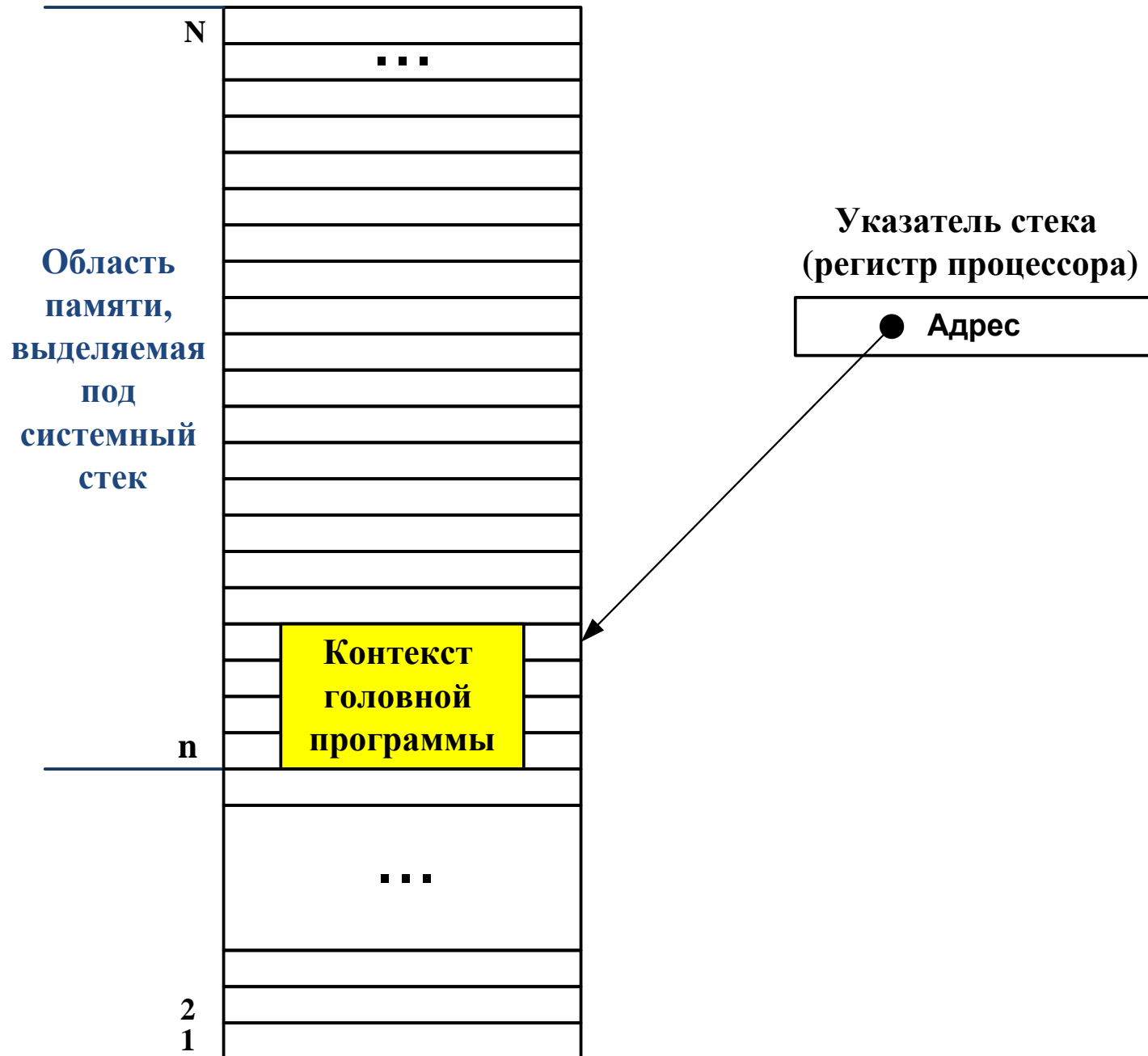
20 25 25 28 31 35 39 42 46

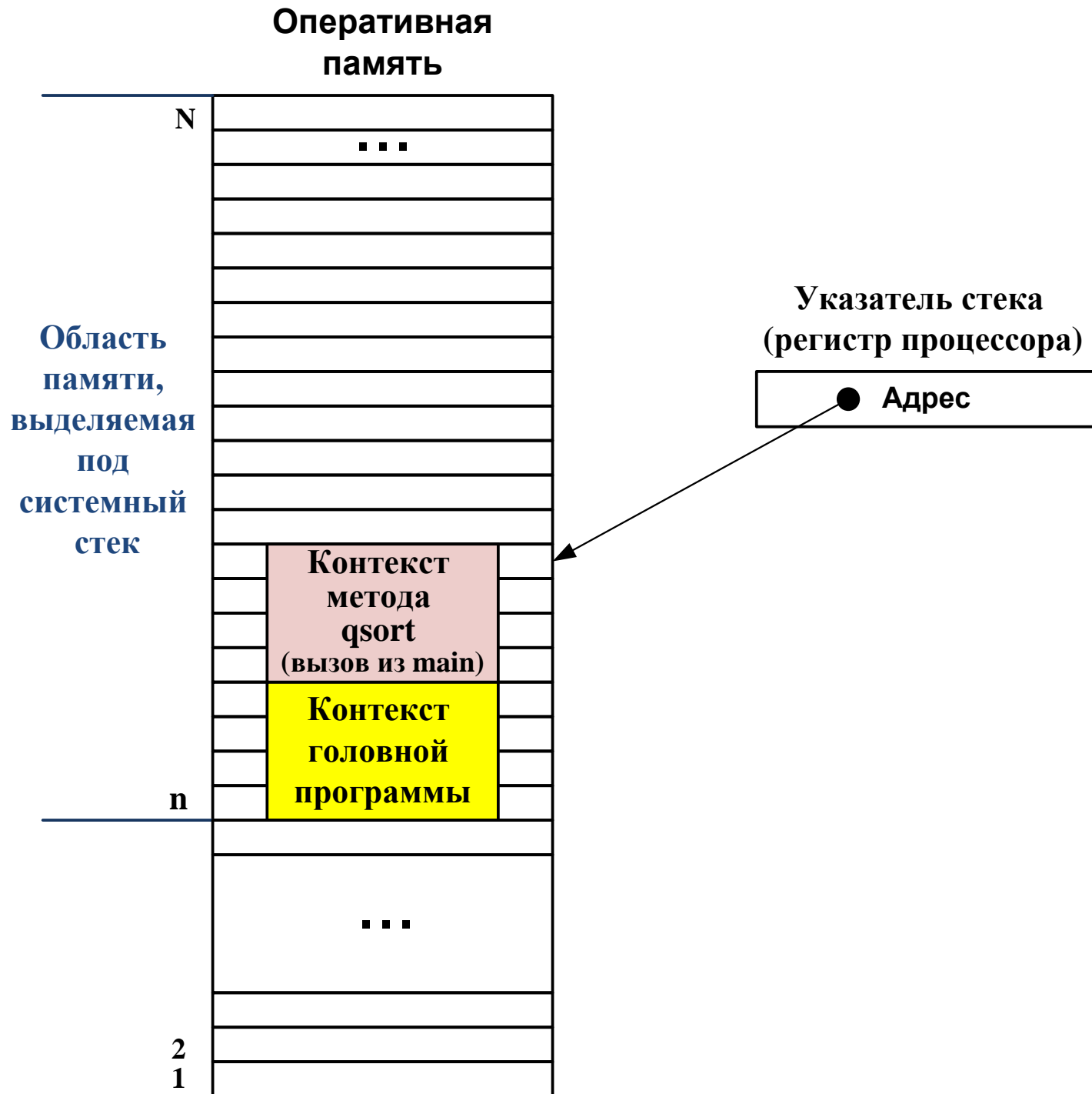
Метод Пузырька на тестовом примере дал 19 перестановок, а метод Хоара – 10 перестановок!

К недостаткам метода быстрой сортировки Хаара можно отнести возможность переполнения системного стека при большой глубине рекурсивных вызовов процедуры `q_sort` (когда исходный массив велик и плохо упорядочен).

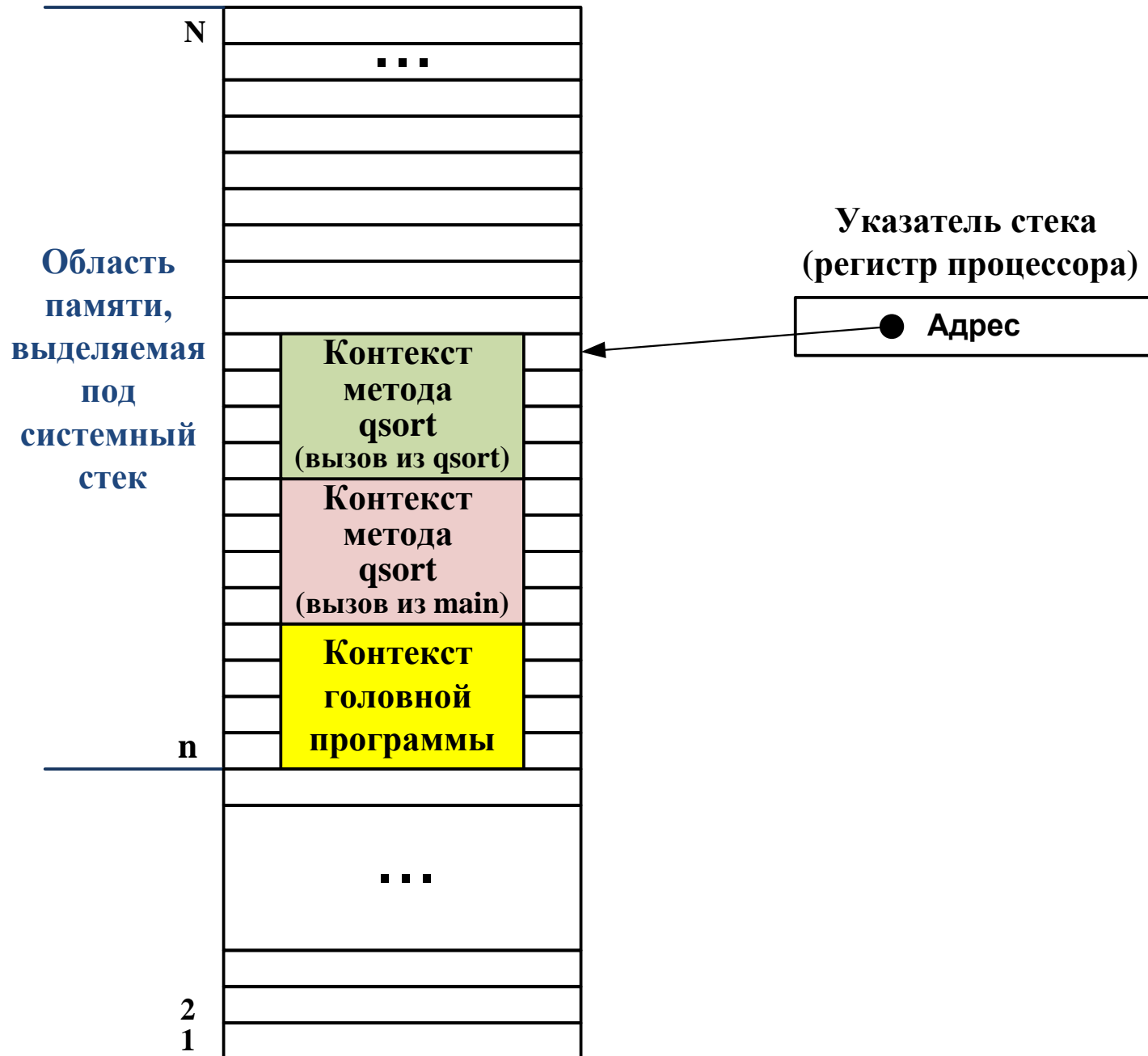
Подобный недостаток присущ и всем другим программам, использующим рекурсивные вызовы подпрограмм.

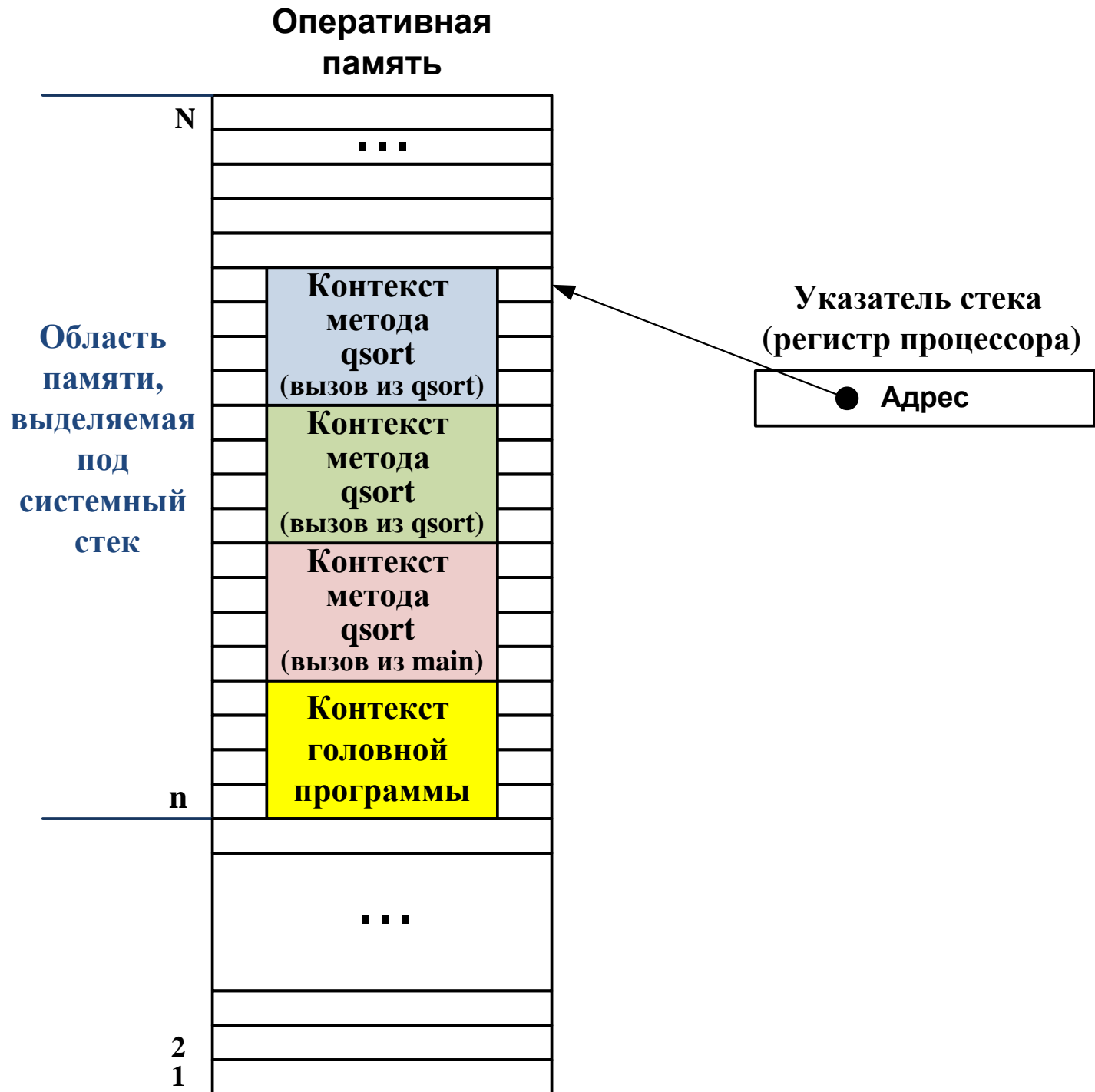
Оперативная память





Оперативная память







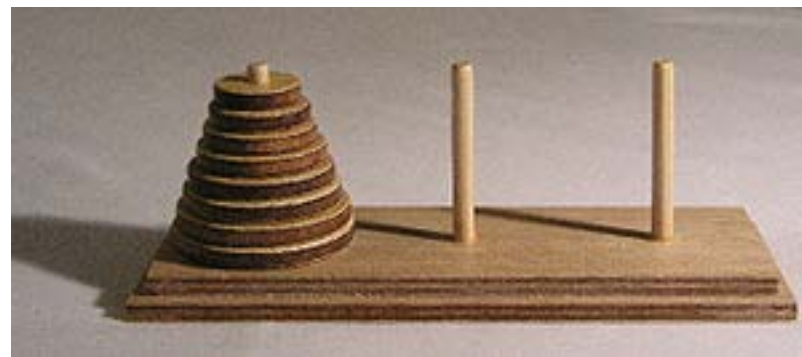
Когда метод вызывает сам себя, новым локальным переменным и параметрам выделяется память в стеке, и код метода выполняется с этими новыми переменными от начала стека. Рекурсивный вызов не делает новой копии метода. Обновляются только параметры. Когда каждый рекурсивный вызов выполняет возврат, старые локальные переменные и параметры удаляются из стека, и выполнение возобновляется в точке вызова внутри метода.

Рекурсивные версии многих подпрограмм могут выполняться немного медленнее, чем итерационный эквивалент, из-за добавления дополнительных вызовов функций. Частые рекурсивные обращения к методу могут вызывать переполнение стека. Поскольку память для параметров и локальных переменных находится в стеке, и каждый новый вызов создает новую копию этих переменных, возможно, что стек может быть исчерпан. Если это происходит, исполнительная система Java вызовет исключение.

Главное преимущество рекурсивных методов состоит в том, что их можно использовать для создания более ясных и простых версий некоторых алгоритмов (по сравнению с их итерационными "родственниками"). Например, алгоритм быстрой сортировки весьма трудно реализовать итерационным способом. Некоторые проблемы, особенно имеющие отношение к искусственному интеллекту, кажется, удобно решать рекурсивно. Наконец, некоторым людям кажется, что рекурсивное мышление легче, чем итеративное.

Задача о ханойских башнях

Легенда гласит, что в Великом храме города Бенарас, под собором, отмечающим середину мира, находится бронзовый диск, на котором укреплены 3 алмазных стержня, высотой в один локоть и толщиной с пчелу.

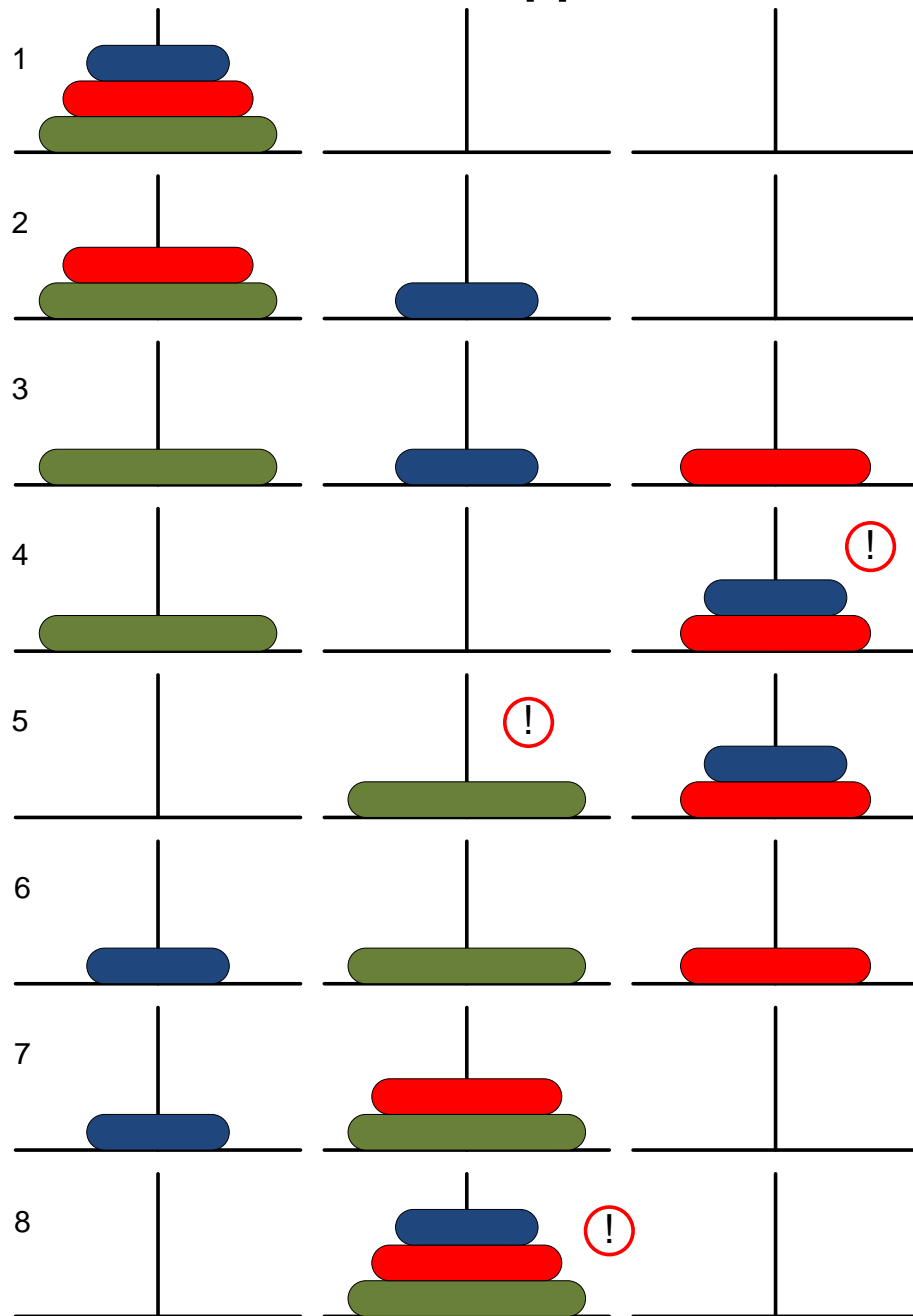


Давным-давно, в самом начале времён, монахи этого монастыря провинились перед богом Брахмой. Разгневанный, Брахма воздвиг три высоких стержня и на один из них возложил 64 диска, сделанных из чистого золота. Причем так, что каждый меньший диск лежит на большем.

Как только все 64 диска будут переложены со стержня, на который Брахма сложил их при создании мира, на другой стержень, башня вместе с храмом обратятся в пыль и под громовые раскаты погибнет мир.

За один раз можно переносить только одно кольцо, причём нельзя класть большее кольцо на меньшее.

Задача о ханойских башнях



Перекладывание стека (пирамидки) из 3 дисков — это:

1. Перекладывание стека (пирамидки) из 2 дисков на вспомогательную ось.

2. Перекладывание 3-го диска на нужную ось.

3. Перекладывание стека (пирамидки) из 2 дисков на нужную ось.

Алгоритмическая сложность

Если мы перемещаем стек из одного диска — то нам нужно 1 действие.

Если стек из двух — то $1 * 2$ (переместить дважды стек из одного диска) + 1 (перемещаем последний диск)

Если из трех $((1 * 2) + 1) * 2 + 1$

Из пяти: $(((((1 * 2) + 1) * 2 + 1) * 2 + 1) * 2 + 1)$

Итак, добавление одного диска увеличивает в 2 раза количество перемещений.

В итоге, количество перекладываний равно $2^n - 1$,
где n — число дисков.

То есть, если нам захочется странного, например записать решение ханойской башни для 64 дисков, то никаких современных носителей информации не хватит.

Число перемещений дисков, которые должны совершить монахи, равно 18 446 744 073 709 551 615. Если бы монахи, работая день и ночь, делали каждую секунду одно перемещение диска, их работа продолжалась бы 584 миллиарда лет.

Задача: **move_many** (**n**, s1, sw, sk) – переложить n дисков со стержня s1 (начальный) на стержень sk (конечный) с помощью стержня sw (вспомогательный):

```
if n = 1 then {если диск один}
    move_one {переместить его с s1 на sk – нерекурсивный
                                                    случай}

else {иначе:}
    begin
        move_many (n - 1, s1, sk, sw); {переместить n-1 дисков
        с начального стержня на вспомогательный с помощью
        конечного}
        move_one; {переместить оставшийся диск с начального
                    стержня на конечный}
        move_many (n-1, sw, s1, sk) {переместить n-1 дисков
        с вспомогательного стержня на конечный с помощью
        начального}
    end;
```

Задача для n дисков сведена к задаче с меньшим числом дисков!

Program Hanoi_towers;

const maxDisks = 64;

type

st = (left, middle, right); *{стержень}*

natur = 1..maxDisks; *{количество дисков}*

var

m: natur; *{m – число дисков в пирамидке}*

```

procedure move_many(n: natur; s1, sw, sk: st);
  {перемещение n дисков с s1(начальный стержень)
   на sk (конечный стержень) с помощью
   sw (вспомогательного стержня)}
  procedure move_one;
    {перемещение одного диска с s1 на sk}
    procedure print (s: st);
      {вывод названия стержня s}
      begin
        case s of
          left: write (' лев. ');
          middle: write (' средн. ');
          right: write (' прав. ');
        end;
      end; {print}

  begin {move_one}
    write(' снять диск с ');
    print(s1);
    write(' надеть на ');
    print(sk);
    writeln
  end; {move_one}

```

```
begin {move_many}  
  if n = 1 then {если диск один - переместить с s1 на sk:}  
    move_one  
  else {иначе:}  
    begin  
      move_many(n - 1, s1, sk, sw); {переместить n-1 дисков  
        с начального стержня на вспомогательный с помощью  
        конечного}  
      move_one; {переместить оставшийся диск с начального  
        стержня на конечный}  
      move_many(n-1, sw, s1, sk) {переместить n-1 дисков  
        с вспомогательного стержня на конечный с помощью  
        начального}  
    end  
  end;  
end; {move_many}
```

```
begin {головная программа}  
  writeln ('Введите число дисков');  
  read(m); {число дисков}  
  writeln ('для ', m:3, ' дисков следует произвести ',  
    'следующие действия:');  
  move_many(m, left, right, middle); {переместить m дисков  
    с левого стержня на средний с помощью правого}  
  readln end.
```

Окно вывода

Введите число дисков

3

для 3 дисков следует произвести следующие действия:

снять диск с лев. надеть на средн.

снять диск с лев. надеть на прав.

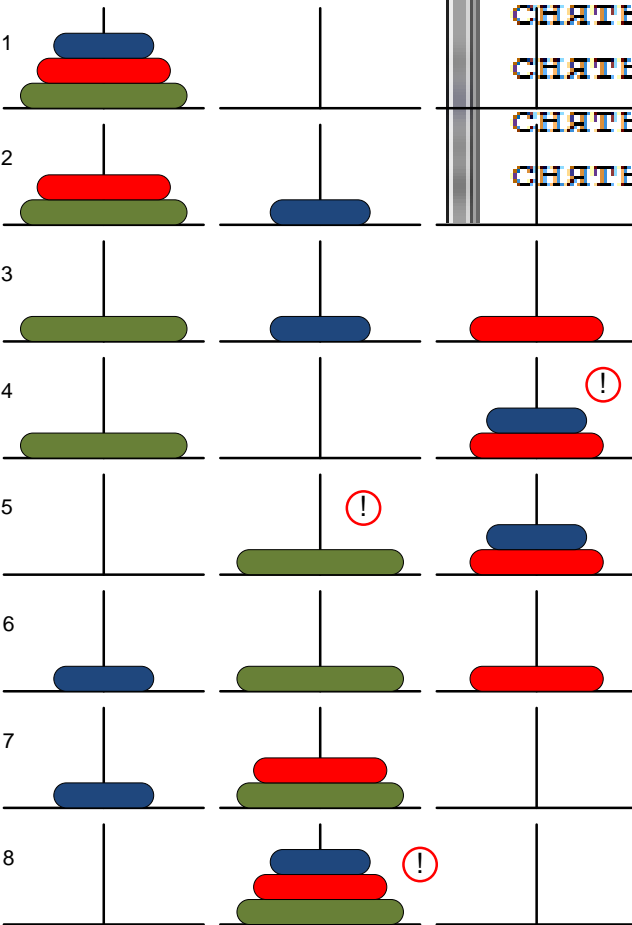
снять диск с средн. надеть на прав.

снять диск с лев. надеть на средн.

снять диск с прав. надеть на лев.

снять диск с прав. надеть на средн.

снять диск с лев. надеть на средн.



Окно вывода

Введите число дисков

4

для 4 дисков следует произвести следующие действия:

снять диск с лев. надеть на прав.

снять диск с лев. надеть на средн.

снять диск с прав. надеть на средн.

снять диск с лев. надеть на прав.

снять диск с средн. надеть на лев.

снять диск с средн. надеть на прав.

снять диск с лев. надеть на прав.

снять диск с лев. надеть на средн.

снять диск с прав. надеть на средн.

снять диск с прав. надеть на лев.

снять диск с средн. надеть на лев.

снять диск с прав. надеть на средн.

снять диск с лев. надеть на прав.

снять диск с лев. надеть на средн.

снять диск с прав. надеть на средн.

для 5 дисков следует произвести следующие действия:

снять диск с лев. надеть на средн.
снять диск с лев. надеть на прав.
снять диск с средн. надеть на прав.
снять диск с лев. надеть на средн.
снять диск с прав. надеть на лев.
снять диск с прав. надеть на средн.
снять диск с лев. надеть на средн.
снять диск с лев. надеть на прав.
снять диск с средн. надеть на прав.
снять диск с средн. надеть на лев.
снять диск с прав. надеть на лев.
снять диск с средн. надеть на прав.
снять диск с лев. надеть на средн.
снять диск с лев. надеть на прав.
снять диск с средн. надеть на прав.
снять диск с лев. надеть на средн.
снять диск с прав. надеть на лев.
снять диск с прав. надеть на средн.
снять диск с лев. надеть на средн.
снять диск с лев. надеть на прав.
снять диск с средн. надеть на прав.
снять диск с лев. надеть на средн.
снять диск с прав. надеть на лев.
снять диск с прав. надеть на средн.
снять диск с лев. надеть на средн.

Выводы.

- 1. Рекурсия – это выражение чего-то через то же самое.**
- 2. Рекурсия применяется при определении данных (рассмотрим позже) и для обработки данных.**
- 3. Рекурсия при обработке данных выражается в вызове подпрограммой самой себя.**
- 4. В подпрограммах, использующих рекурсию, нужно обязательно предусмотреть нерекурсивный случай.**
- 5. Иногда рекурсия позволяет повысить эффективность обработки данных (алгоритм быстрой сортировки Хаара).**
- 6. Иногда без рекурсии невозможно даже записать решение задачи, т.к. потребуется слишком много операторов (задача о Ханойских башнях: рекурсия здесь не влияет на трудоемкость алгоритма (число операций), но позволяет компактно записать (и вообще записать;-) решение задачи) .**
- 7. Если рекурсивные алгоритмы не дают перечисленных в пунктах 5 и 6 преимуществ, то лучше использовать итерационные алгоритмы, чтобы избежать переполнения системного стека при большой глубине рекурсивных вызовов.**