Сортировка массивов

Алгоритмы сортировки одномерных массивов

Под **сортировкой** понимают процесс перестановки объектов данного массива в определенном порядке. Целью сортировки являются упорядочение массивов для облегчения последующего поиска элементов в данном массиве.
Рассмотрим основные алгоритмы сортировки по возрастанию числовых значений элементов массивов.

Алгоритмы сортировки одномерных массивов

Пусть есть последовательность $a_0, a_1 \dots a_n$ и функция сравнения, которая на любых двух элементах последовательности принимает одно из трех значений: меньше, больше или равно. Задача сортировки состоит в перестановке членов последовательности таким образом, чтобы выполнялось условие: $a_i <= a_{i+1}$, для всех i от 0 до n. Возможна ситуация, когда элементы состоят из нескольких полей:

```
struct element {  field x;  field y; }
```

Алгоритмы сортировки одномерных массивов

Если значение функции сравнения зависит только от поля x, то x называют ключом, по которому производится сортировка. На практике, в качестве x часто выступает число, а поле y хранит какие-либо данные, никак не влияющие на работу алгоритма.
Проблема сортировки породила множество различных алгоритмов, потому, что не существует некоторого "универсального", наилучшего алгоритма. Однако, имея приблизительные характеристики входных данных, можно подобрать метод, работающий оптимальным образом

Алгоритмы сортировки одномерных массивов

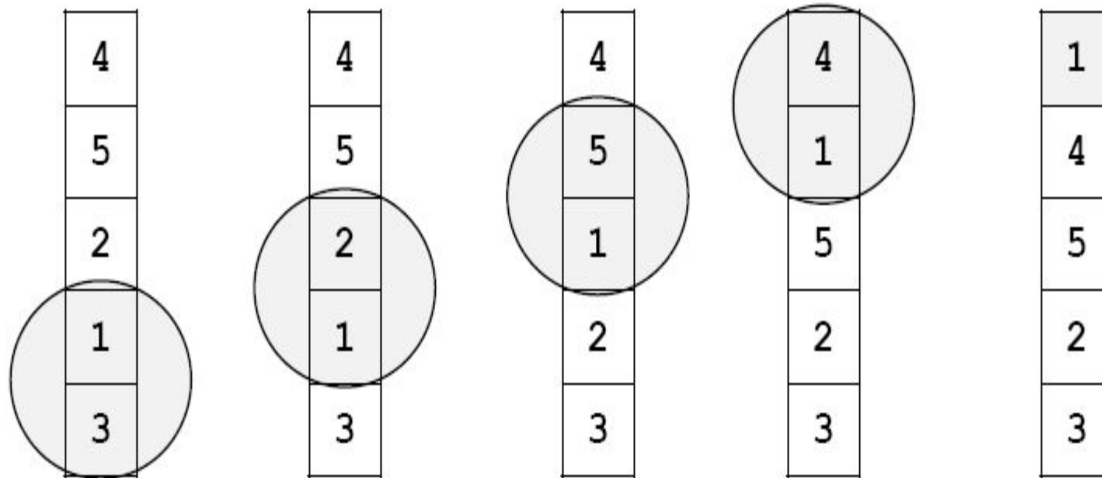
Для того, чтобы обоснованно сделать такой выбор, рассмотрим параметры, по которым будет производиться оценка алгоритмов.

- *Время* сортировки - основной параметр, характеризующий быстродействие алгоритма.
- *Память* - ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке используемой памяти не будет учитываться место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы.
- *Устойчивость* - устойчивая сортировка не меняет взаимного расположения равных элементов. Такое свойство может быть очень полезным, если они состоят из

Сортировка методом парных перестановок (методом «пузырька»)

- Самый простой вариант алгоритма сортировки массива основан на принципе сравнения и обмена пары соседних элементов. Процесс перестановок пар повторяется просмотром массива с начала до тех пор, пока не будут отсортированы все элементы, т.е. во время очередного просмотра не произойдет ни одной перестановки.

Сортировка методом парных перестановок (методом «пузырька»)



- Для подсчета количества перестановок целесообразно использовать счетчик – специальную переменную . Если при просмотре элементов массива значение счетчика перестановок осталось равным нулю, то это означает, что все элементы отсортированы

Сортировка методом парных перестановок (методом «пузырька»)

Пример кода программы

```
#include <stdio.h>
#include <stdlib.h>

void pr(int a[],int n)
{   int i;
for (i=0; i<n; i++) printf(" a[%d]=%d  ",i,a[i]);}

int main() {      int i,j,K;
  int n=7;     int a[n];
  for (i=0; i<n; i++) {a[i]=rand();}
  pr(a,n);
  printf("\n\n");
  do
  { j=0;
  for(i=0;i<n-1;i++)
  {
      if(a[i]>a[i+1])
      { K=a[i]; a[i]=a[i+1];a[i+1]=K;
        j++; }

        else a[i]=a[i]; }
   }
   while (j!=0) ;
    pr(a,n);
    printf("\n");
    return 0;
  }
```

Сортировка методом парных перестановок (методом «пузырька»)

Результат работы программы

```
a[0]=41   a[1]=18467   a[2]=6334   a[3]=26500   a[4]=19169   a[5]=15724   a[6]=11478

a[0]=41   a[1]=6334    a[2]=11478  a[3]=15724   a[4]=18467   a[5]=19169   a[6]=26500

Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.
```

Этот алгоритм всегда будет дела $\frac{(n^2-n)}{2}$ шагов, независимо от входных данных. Даже если массив отсортирован, $\frac{(n^2-n)}{2}$ авно он будет пройден раз. Более того, буду в очередной раз проверены уже отсортированные данные.

Сортировка является устойчивой. Не требует дополнительного объема памяти.

Сортировка методом парных перестановок (методом «пузырька»)

Пузырьковая сортировка имеет такую особенность: неупорядоченные элементы на "большом" конце массива занимают правильные положения за один проход, но неупорядоченные элементы в начале массива поднимаются на свои места очень медленно. Поэтому, вместо того чтобы постоянно просматривать массив в одном направлении, в последовательных проходах можно чередовать направления. Таким образом, элементы, сильно удаленные от своих положений, быстро станут на свои места. Данная версия пузырьковой сортировки носит название **шейкер-сортировки** (shaker sort сортировка перемешиванием, сортировка взбалтыванием, сортировка встряхиванием), поскольку действия, производимые ею с массивом, напоминают взбалтывание или встряхивание.

Сортировка модифицированным методом простого выбора

- Этот метод основывается на алгоритме поиска минимального элемента. В массиве А[1..n] отыскивается минимальный элемент, который ставится на первое место. Для того, чтобы не потерять элемент, стоящий на первом месте, этот элемент устанавливается на место минимального. Затем в усеченной последовательности, исключая первый элемент, отыскивается минимальный элемент и ставится на второе место и так далее *n*-1 раз пока не станет на свое место предпоследний *n*-1 элемент массива А, сдвинув максимальный элемент в самый конец.

Сортировка модифицированным методом простого выбора

- Рассмотрим алгоритмическое решение задачи на примере сортировки некоторого массива значений по возрастанию. Необходимо несколько раз выполнять операции поиска минимального элемента и его перестановку с другим элементом, то есть потребуется несколько раз просматривать элементы массива с этой целью. Количество просмотров элементов массива равно n-1, где n- количество элементов массива. Проектируемый алгоритм сортировки будет содержать цикл, в котором будет выполняться поиск минимального элемента и его перестановка с другим элементом

Схема метода

Через *i* обозначен счетчик (номер) просмотров элементов массива.

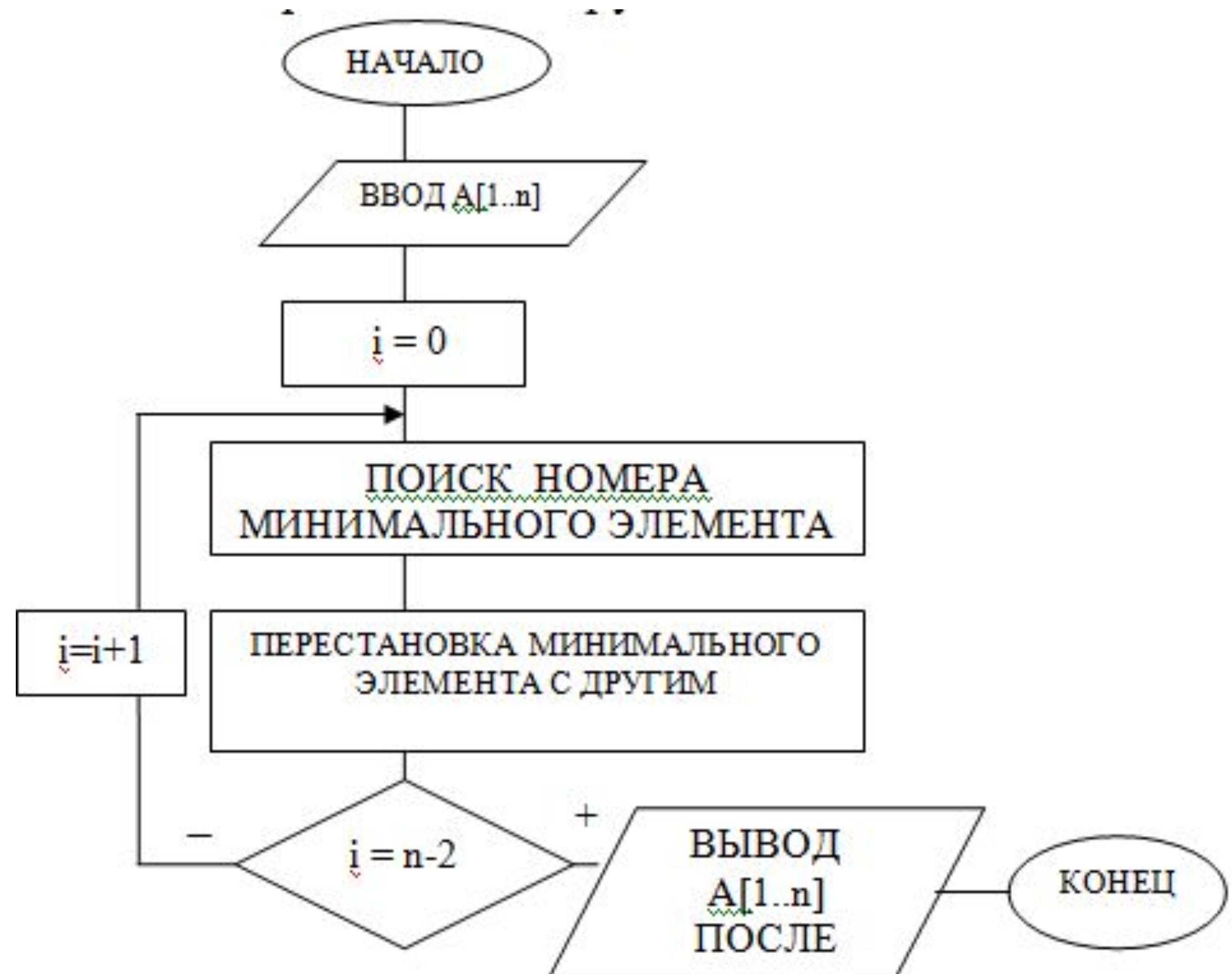


Схема метода

- Рассмотрим выполнение сортировки данным методом на конкретном примере. Пусть исходный массив содержит 5 элементов (2, 8, 1, 3, 7). Количество просмотров согласно модифицированному методу простого выбора будет равно 4. Покажем в таблице, как будет изменяться исходный массив на каждом просмотре.

Схема метода

Номер просмотра массива i	Исходный Массив	Минимальный Элемент		Переставляемый элемент		Массив после переста-новки
		Номер	Значе-ние	Номер	Значе-ние	
0	(2,8,1,3,7)	2	1	0	2	(1,8,2,3,7)
1	1,(8,2,3,7)	2	2	1	8	1,(2,8,3,7
2	1,2,(8,3,7)	3	3	2	8	1,2,(3,8,7)
3	1,2,3,(8,7)	4	7	3	8	1,2,3,7,8

Код программы

```
#include <stdio.h>
#include <stdlib.h>

void pr(int a[],int n)
{    int i;
for (i=0; i<n; i++) printf(" a[%d]=%d  ",i,a[i]);}

int main() {      int i,j,K;
  int n=7;     int a[n];
  for (i=0; i<n; i++) {a[i]=rand();}
  pr(a,n);
  printf("\n\n");
  for ( i = 0; i < n - 1; i++) {
/* устанавливаем начальное значение минимального индекса */
          int min_i = i;
     /* находим индекс минимального элемента */
     for (j = i + 1; j < n; j++) {
          if (a[j] < a[min_i]) {
               min_i = j;    }
     }
     /* меняем значения местами */
     int temp = a[i];
     a[i] = a[min_i];
     a[min_i] = temp; }
     pr(a,n);
     printf("\n");
     return 0;   }
```

Результат работы программы

```
 a[0]=41   a[1]=18467   a[2]=6334   a[3]=26500   a[4]=19169   a[5]=15724   a[6]=11478

 a[0]=41   a[1]=6334   a[2]=11478   a[3]=15724   a[4]=18467   a[5]=19169   a[6]=26500

Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.
```

Результат работы программы

Как и в пузырьковой сортировке, внешний цикл выполняется n-1 раз, а внутренний – в среднем n/2 раз. Сортировка методом простого выбора требует

$$\frac{(n^2-n)}{2}$$

сравнений. Это алгоритм порядка n^2, из-за чего он считается слишком медленным для сортировки большого количества элементов. Несмотря на то, что количество сравнений в пузырьковой сортировке и сортировке простым выбором одинаковое, в последней количество обменов в среднем случае намного меньше, чем в пузырьковой сортировке.

Результат работы программы

Покажем, почему данная реализация является неустойчивой.

Рассмотрим следующий массив из элементов, каждый из которых имеет два поля. Сортировка идет по первому полю.

Массив до сортировки:

{ (2, a), (2, b), (1, a) }

Уже после первой итерации внешнего цикла будем иметь отсортированную последовательность:

{ (1, a), (2, b), (2, a) }

Теперь заметим, что взаимное расположение элементов

(2, a) и (2, b) изменилось. Таким образом, рассматриваемая реализация является неустойчивой.

Результат работы программы

Существует также двунаправленный вариант сортировки методом выбора, в котором на каждом проходе отыскиваются и устанавливаются на свои места и минимальное, и максимальное значения.

Сортировка методом простого включения (вставками)

В этом методе задача формулируется так: есть часть массива, которая уже отсортирована, и требуется вставить остальные элементы массива в отсортированную часть, сохранив при этом упорядоченность. Для этого на каждом шаге алгоритма мы выбираем один из элементов входных данных и вставляем его на нужную позицию в уже отсортированной части массива, до тех пор пока весь набор входных данных не будет отсортирован.

Сортировка методом простого включения (вставками)

Метод выбора очередного элемента из исходного массива произволен, однако обычно (и с целью получения устойчивого алгоритма сортировки), элементы вставляются по порядку их появления во входном массиве.
Так как в процессе работы алгоритма могут меняться местами только соседние элементы, каждый обмен уменьшает число инверсий на единицу. Следовательно, количество обменов равно количеству инверсий в исходном массиве вне зависимости от реализации сортировки.

Сортировка методом простого включения (вставками)

Максимальное количество инверсий содержится в массиве, элементы которого отсортированы по невозрастанию. Число инверсий в таком массиве

$$\frac{n(n-1)}{2}.$$

Сортировка вставками наиболее эффективна когда массив уже частично отсортирован и когда элементов массива не много. Если же число элементов меньше 10, то данный алгоритм является лучшим.

Сортировка методом простого включения (вставками)

Рассмотрим на примере числовой последовательности процесс сортировки методом вставок. Клетка, выделенная темно-серым цветом – активный на данном шаге элемент, ему также соответствует i-ый номер. Светло-серые клетки это те элементы, значения которых сравниваются с i-ым элементом. Все, что закрашено белым – не затрагиваемая на шаге часть последовательности.

i = 2 | 8 | 5 | 0 | 3 | 7 | → | 5 | 8 | 0 | 3 | 7 |

i = 3 | 5 | 8 | 0 | 3 | 7 | → | 0 | 5 | 8 | 3 | 7 |

i = 4 | 0 | 5 | 8 | 3 | 7 | → | 0 | 3 | 5 | 8 | 7 |

i = 5 | 0 | 3 | 5 | 8 | 7 | → | 0 | 3 | 5 | 7 | 8 |

Код программы

```
#include <stdio.h>
#include <stdlib.h>

void pr(int a[],int n)
{    int i;
for (i=0; i<n; i++) printf(" a[%d]=%d  ",i,a[i]);}

int main() {  int i,j,temp;
  int n=7;     int a[n];
  for (i=0; i<n; i++) {a[i]=rand();}
  pr(a,n);
  printf("\n\n");
for (i = 1; i < n; i++)
{    temp = a[i];
     for (j = i - 1; j >= 0; j--)
     {
          if (a[j] < temp)
               break;
          a[j + 1] = a[j];
          a[j] = temp;
     }
}
     pr(a,n);
     printf("\n");
     return 0;   }
```

Результат работы

```
 a[0]=41   a[1]=18467   a[2]=6334   a[3]=26500   a[4]=19169   a[5]=15724   a[6]=11478

 a[0]=41   a[1]=6334   a[2]=11478   a[3]=15724   a[4]=18467   a[5]=19169   a[6]=26500

Process returned 0 (0x0)   execution time : 1.063 s
Press any key to continue.
```

Сортировка вставками

Это устойчивый алгоритм сортировки (не меняет порядок элементов, которые уже отсортированы);

может сортировать массив по мере его получения;

не требует временной памяти.

Сортировка вставками

Это устойчивый алгоритм сортировки (не меняет порядок элементов, которые уже отсортированы);

может сортировать массив по мере его получения;

не требует временной памяти.

Быстрая сортировка (*quick-sort*)

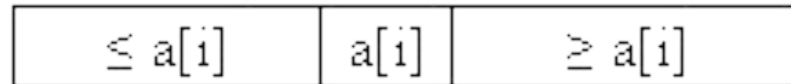
Быстрая сортировка является улучшенным вариантом алгоритма сортировки с помощью прямого обмена (пузырьковая сортировка). Она является наиболее широко применяемым и одним их самых эффективных алгоритмов. Разработана английским информатиком Чарльзом Хоаром во время его работы в Московском государственном университете в 1960 году.

Быстрая сортировка (*quick-sort*)

Общая схема алгоритма:

- из массива выбирается некоторый опорный элемент a[i]
- запускается процедура разделения массива, которая перемещает все элементы, меньшие, либо равные a[i], влево от него, а все элементы, большие, либо равные a[i] - вправо
- теперь массив состоит из двух подмножеств, причем левое меньше, либо равно правого,

Быстрая сортировка (*quick-sort*)



- для обоих подмассивов: если в подмассиве более двух элементов, рекурсивно запускаем для него ту же процедуру.
- В конце получится полностью отсортированная последовательность.

Быстрая сортировка (*quick-sort*)

Рассмотрим алгоритм подробнее.

- На входе массив a[0]...a[N] и опорный элемент *p*, по которому будет производиться разделение.
- Введем два указателя: *i* и *j*. В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.
- Будем двигать указатель *i* с шагом в 1 элемент по направлению к концу массива, пока не будет найден элемент a[i] >= p. Затем аналогичным образом начнем двигать указатель *j* от конца массива к началу, пока не будет найден a[j] <= p.
- Далее, если i <= j, меняем a[i] и a[j] местами и продолжаем двигать i, j по тем же правилам...
- Повторяем шаг 3, пока i <= j.

Быстрая сортировка (*quick-sort*)

Рассмотрим работу процедуры для массива a[0]...a[6] и опорного элемента p = a[3].

Теперь массив разделен на две части: все элементы левой меньше либо равны p, все элементы правой - больше, либо равны p. Разделение завершено.

Быстрая сортировка. Оценка сложности алгоритма

- Операция разделения массива на две части относительно опорного элемента занимает время O (n) . Поскольку все операции разделения, проделываемые на одной глубине рекурсии, обрабатывают разные части исходного массива, размер которого постоянен, суммарно на каждом уровне рекурсии потребуется также O (n) операций. Общая сложность алгоритма определяется лишь количеством разделений, то есть глубиной рекурсии. Глубина рекурсии, в свою очередь, зависит от сочетания входных данных и способа определения опорного элемента.

Быстрая сортировка. Оценка сложности алгоритма

- Лучший случай. В наиболее сбалансированном варианте при каждой операции разделения массив делится на две приблизительно одинаковые части, следовательно, максимальная глубина рекурсии, при которой размеры обрабатываемых подмассивов достигнут 1, составит $\log_2 n$, что даёт общую сложность алгоритма $O(n \cdot \log_2 n)$.
- Среднее. Средняя сложность алгоритма составляет $O(n \log n)$.

Быстрая сортировка. Оценка сложности алгоритма

- Худший случай. Реализуется если каждый раз в качестве центрального элемента выбирается максимум или минимум входной последовательности. В этом случае каждое разделение даёт два подмассива размерами 1 и $n - 1$ и при каждом рекурсивном вызове больший массив будет на 1 короче, чем в предыдущий раз.
- В этом случае потребуется $n - 1$ операций разделения, а общее время работы составит $O(n^2)$ операций, то есть сортировка будет выполняться за квадратичное время.
- Для больших значений n худший случай может привести к исчерпанию памяти (переполнению стека) во время работы программы.

Быстрая сортировка.

- Метод неустойчив. Поведение довольно естественно, если учесть, что при частичной упорядоченности повышаются шансы разделения массива на более равные части.
- Сортировка использует дополнительную память, так как приблизительная глубина рекурсии составляет O(log n), а данные о рекурсивных подвызовах каждый раз добавляются в стек.
- Улучшения алгоритма направлены, в основном, на устранение или смягчение вышеупомянутых недостатков, вследствие чего все их можно разделить на три группы: придание алгоритму устойчивости, устранение деградации производительности специальным выбором опорного элемента, и защита от переполнения стека вызовов из-за большой глубины

Рекурсия

- В языке Си функции могут ***вызывать сами себя*** непосредственно или косвенно, т.е. могут быть ***рекурсивными***. Если функция непосредственно вызывает саму себя – то это называется ***прямой рекурсией***, а если функция вызывает какую-либо другую функцию, которая либо сама, либо посредством другой функции вызывает исходную функцию – то это называется ***косвенной рекурсией***.

Рекурсия

- Каждая цепочка рекурсивных вызовов должна на каком-то шаге завершиться. Условие полного окончания работы рекурсивной функции должно находиться в самой функции (иначе произойдет зацикливание), а именно, любая рекурсивная функция должна содержать рекурсивный вызов внутри условного оператора.

Рекурсия

- Применять рекурсивные методы программирования стоит в тех задачах, где рекурсия использована в определении обрабатываемых данных. Многие задачи, решаемые при помощи рекурсии, более эффективно решаются либо с помощью итеративных алгоритмов либо с помощью подпрограмм.
- Например, вычисление факториала, которое мы рассматриваем ниже, удобно для объяснения рекурсии, однако не дает никакого выигрыша в программной реализации. Более того, рекурсивный алгоритм вычисления факториала работает медленнее итеративного алгоритма, за счет накладных расходов на вызов функции и возврат значений.

Рекурсия

Пример программы вычисляющей факториал числа

```
#include <stdio.h>
int main()
{   int f(int k); /*функция вычисления факториала*/
    int fact = f(5 );
    printf( "fact 5! =%d\n ",fact);
return 0;
}
/* Рекурсивная функция вычисления факториала*/
int f(int k)
{   if (k == 0) return 1; /*т.к. 0! == 1*/
    return k * f(k-1);
}
```

```
fact 5! =120

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

Рекурсия

- При первом вызове **f(5)** функция возвращает выражение **5*f(4)**, т.е. функция **f()** фактически не возвращает значение, а вызывает сама себя с другим значением. Рекурсивные вызовы будут продолжаться до тех пор, пока **k** не станет равным **0**. Будет создана следующая цепочка возвращаемых выражений:

  5*f(4), 4*f(3), 3*f(2), 2*f(1), 1*f(0)
- Вызов **f(0)** не провоцирует дальнейших вызовов, а возвращает значение 1, произведение **1*1** будет возвращено предыдущему вызову и т.д. до вызова **f(5)**, которому возвращается значение **120**. Тем самым будут организованы следующие умножения:

  1*1*2*3*4*5, а в общем случае
  1*1*2*3*4*5*…*(k-1)*k

Код алгоритма *quick-sort*

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   void qs (char *items, int left, int right)
5   {    int i, j;
6        char x, y;
7        i= left; j = right;
8        x= items [(left+right)/2];
9        do
10       {
11           while ((items[i]<x)&&(i<right)) i++;
12           while ((x<items [j])&&(j>left)) j--;
13
14           if(i<=j)
15           {    y=items[i];
16                items[i] = items [j];
17                items [j] = y;
18                i++; j--; }
19       }
20       while (i<=j);
21       if (left<j) qs (items, left, j);
22       if (i<right) qs(items, i, right);
23   }
```

Код алгоритма *quick-sort*

```
int main()
{
  int n,k;
  n=20;
  char str [n];

   for(k=0;k<n;k++)
  {   str[k]=(rand() % 25)+97; }

      for(k=0;k<n;k++)
  {       printf(" %c",str[k]);

      qs ( str, 0, n-1);
  printf(" \n");

       for(k=0;k<n;k++)
  {       printf(" %c",str[k]);

      return 0;}
```

```
q r j a t y d i m o f u g c l q u r c l
a c c d f g i j l l m o q q r r t u u y
```