Практическая работа 10

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

Сортировка выбором. На каждом шаге сортировки из последовательности выбирается минимальный элемент и переносится в конец выходной последовательности. Дальше вступают в силу детали процесса, но характерным остается наличие двух независимых частей – неупорядоченной (оставшихся элементов) и упорядоченной. При исключении выбранного элемента из массива на его место может быть записано «очень большое число», исключающее его повторный выбор. Выбранный элемент может удалятся путем сдвига оставшейся части, минимальный элемент может меняться местами с «очередным». Трудоемкость алгоритма – **n×n/2**.

Следующий пример – один из многочисленных вариантов «мирного сосуществования» упорядоченной и неупорядоченной частей в одном массиве. Упорядоченная часть находится слева, и ее размерность соответствует числу выполненных шагов внешнего цикла. Неупорядоченная часть расположена справа, поэтому поиск минимума с запоминанием индекса минимального элемента происходит в интервале от **i** до конца массива.

```
//-------------------------------------------------25-05.cpp
//---- Сортировка выбором
void sort(int in[], int n){
for ( int i=0; i < n-1; i++){                  // Для очередного i
        for ( int j=i+1, k=i; j<n; j++)        // k - индекс минимального
        if (in[j] < in[k]) k=j;                // в диапазоне i..n-1
        int c=in[k]; in[k] = in[i]; in[i] = c; // Три стакана для очередного
        }}                                     // и минимального
```

В сортировке выбором контекст выбора минимального элемента обычно заметен «невооруженным глазом». Но в следующем варианте он совмещен с процессом обмена и потому не виден: минимальный элемент сразу же перемещается на очередную позицию.

```
//-------------------------------------------------25-06.cpp
//---- " Законспирированная" сортировка выбором
 void sort(int in[], int n){
 for ( int i=0; i < n-1; i++)                  // Для очередного i
 for ( int j=i+1, k=i; j<n; j++)               // Для всех оставшихся
        if (in[j] < in[ i]) {                  // в диапазоне i..n-1
     int c=in[i]; in[i] = in[j]; in[j] = c;    // сразу же менять с очередным
     }}                                  // Выбор совмещен с обменом
```

Сортировка вставками. Основная идея алгоритма: имеется упорядоченная часть, в которую очередной элемент помещается так, что упорядоченность сохраняется (включение с сохранением порядка). Технические детали: можно проводить линейный поиск от начала упорядоченной части до первого, больше данного, с конца – до первого, меньше данного (трудоемкость алгоритма по операциям сравнения – **n×n/4**), использовать двоичный поиск места в упорядоченной части (трудоемкость алгоритма – **n×log(n)**). Сама процедура вставки включает в себя перемещение элементов массива (не учтенное в приведенной трудоемкости). В следующем примере последовательность действий по вставке очередного элемента в упорядоченную часть «разложена по полочкам» в виде последовательности четырех действий, связанных переменными.

```
//-------------------------------------------------25-07.cpp
//---- Простая вставка
 void sort(int in[], int n){
 for ( int i=1; i < n; i++) {          // Для очередного i
      int v=in[i];                     // Делай 1 : сохранить очередной
      for (int k=0; k<i; k++)          // Делай 2 : поиск места вставки
            if(in[k]>v) break;         // перед первым, большим v
```

```
for(int j=i-1; j>=k; j--)      // Делай 3: сдвиг на 1 вправо
in[j+1]=in[j];                 // от очередного до найденного
in[k]=v;                       // Делай 4 : вставка очередного на место
}}                             // первого, большего него
```

В сортировке выбором нет характерных программных контекстов, «ответственных» за вставку: характер программы определяется циклом поиска места вставки, который корректно работает только на упорядоченных данных. Таким образом, получается замкнутый круг для логического анализа, разрываемый только доказательством методом математической индукции: вставка на **i**-м шаге выполняется корректно в упорядоченных данных, подготовленных аналогичным **i–1**-м шагом, и т.д. до 0.

Вставка погружением. Очередной элемент «погружается» путем ряда обменов с предыдущим до требуемой позиции в уже упорядоченную часть массива, пока «не достигнет дна» либо пока не встретит элемент, меньше себя. Наличие контекста «трех стаканов» делает его подозрительно похожим на обменную сортировку, но это не так.

```
//-------------------------------------------------25-08.cpp
//----- Вставка погружением, подозрительно похожая на обмен
 void sort(int in[],int n) {
 for ( int i=1; i<n; i++)      // Пока не достигли " дна" или меньшего себя
        for ( int k=i; k !=0 && in[k] < in[k-1]; k--){
    int c=in[k]; in[k]=in[k-1]; in[k-1]=c;
    }}
```

Сортировка Шелла. Существенными в сортировках вставками являются затраты на обмены или сдвиги элементов. Для их уменьшения желательно сначала производить погружение с большим шагом, сразу определяя элемент «по месту», а затем делать точную «подгонку». Так поступает **сортировка Шелла**: исходный массив разбивается на **m** частей, в каждую из которых попадают элементы с шагом **m**, начиная от **0, 1, ..., m–1** соответственно, то есть

```
0 , m  ,  2m  ,  3m  ,...
1 , m+1, 2m+1, 3m+1,...
2 , m+2, 2m+2, 3m+2,...
```

Каждая часть сортируется отдельно с использованием алгоритма вставок или обмена. Затем выбирается меньший шаг, и алгоритм повторяется. Шаг удобно выбрать равным степеням 2, например, 64, 32, 16, 8, 4, 2, 1. Последняя сортировка выполняется с шагом 1. Несмотря на увеличение числа циклов, суммарное число перестановок будет меньшим. Принцип сортировки Шелла можно применить и во всех обменных сортировках.

Замечание. Сортировка Шелла требует четырех вложенных циклов: по шагу сортировки (по уменьшающимся степеням 2 –

m=64, 32, 16 ...), по группам (по индексу первого элемента в диапазоне **k=0...m–1**), а затем два цикла обычной сортировки погружением для элементов группы, начинающейся с **k** с шагом **m**. Для двух последних циклов нужно взять базовый алгоритм, заменив шаг 1 на **m** и поменяв границы сортировки.

Обменная сортировка «пузырьком». Обзор вариантов обменной сортировки начнем с горячо любимой автором (с методической точки зрения), но с наименее эффективной простой сортировки обменом, или сортировки **методом «пузырька»**. Суть ее заключается в следующем: производятся попарное сравнение соседних элементов 0–1, 1–2 ... и перестановка, если пара расположена не в порядке возрастания. Просмотр повторяется до тех пор, пока при пробегании массива от начала до конца перестановок больше не будет.

```
//-----------------------------------------------------25-09.cpp
//------Сортировка методом "пузырька"
 void sort(int A[], int n){
 int i,found;                                  // Количество сравнений
 do {                                          // Повторять просмотр...
     found =0;
     for (i=0; i<n-1; i++)
            if (A[i] > A[i+1]) {               // Сравнить соседей
         int cc = A[i]; A[i]=A[i+1]; A[i+1]=cc;
         found++;                              // Переставить соседей
         }
     } while(found !=0); }                     //.пока есть перестановки
```

Оценить трудоемкость алгоритма можно через среднее количество сравнений, которое равно **(n×n–n)/2**.

Обменные сортировки имеют ряд особенностей. Прежде всего, они чувствительны к степени исходной упорядоченности массива. Полностью упорядоченный массив будет просмотрен ими один раз, в то время как выбор или вставка будут «изображать бурную деятельность». Кроме того, основное свойство, на котором основана их оптимизация, непосредственно не наблюдаемо в тексте программы: ему не соответствует никакой программный контекст, и оно выводится из наблюдения за последовательным выполнением ряда шагов цикла: элемент с большим значением «захватывается» рядом последовательных обменов и «всплывает» к концу массива, пока не встретит элемент, больше себя. С этим последним процесс продолжается.

Шейкер-сортировка учитывает тот факт, что от последней перестановки до конца массива будут находиться уже упорядоченные данные, например:

```
шаг n                          5  7  10  9  8  12  14
                               5  7  *****  8  12  14
последняя перестановка         5  7  9   *****  12  14
                               5  7  9   8  10 12  14
шаг n+1                                  -----------
                                         упорядоченная часть
```

Это свойство так же не очевидно, как и предыдущее, то есть не наблюдается непосредственно в программных контекстах. Но исходя из него, просмотр имеет смысл делать не до конца массива, а до последней перестановки, выполненной на предыдущем просмотре. Для этой цели в программе обменной сортировки необходимо запоминать индекс переставляемой пары, который по завершении внутреннего цикла просмотра и будет индексом последней перестановки. Кроме того, необходима переменная – граница упорядоченной части, которая должна при переходе к следующему шагу получать значение пресловутого индекса последней перестановки. Условие окончания – граница сместится к началу массива.

```
//-------------------------------------------------25-10.cpp
//------ Однонаправленная Шейкер-сортировка
 void sort(int A[], int n){
 int i,b,b1;                  // b граница отсортированной части
 for (b=n-1; b!=0; b=b1) {    // Пока граница не сместится к правому краю
     b1=0;                    // b1 место последней перестановки
     for (i=0; i<b; i++)      // Просмотр массива
         if (A[i] > A[i+1]) {  // Перестановка с запоминанием места
         int cc = A[i]; A[i]=A[i+1]; A[i+1]=cc;
         b1=i;
         }}}
```

Если же просмотр делать попеременно в двух направлениях и фиксировать нижнюю и верхнюю границы неупорядоченной части, то получим классическую Шейкер-сортировку.

Сортировка подсчетом. Особняком стоящая сортировка, требующая обязательного выходного массива, поскольку элементы в нем размещаются не подряд. Идея алгоритма: число элементов, меньше текущего, определяет его позицию (индекс) в выходном массиве. Наличие переменной-счетчика и использование его в качестве индекса в выходном массиве являются хорошо заметными программными контекстами. Трудоемкость алгоритма – **n×n/2**.

```
//-------------------------------------------------25-11.cpp
//----- Сортировка подсчетом (неполная)
 void sort(int in[],int out[],int n)
 { int i,j ,cnt;
 for (i=0; i< n; i++) {
     for ( cnt=0,j=0; j<n; j++)
         if (in[j] < in[i]) cnt++;       // Счетчик элементов, больших текущего
     out[cnt]=in[i];                     // Определяет его место в выходном
     }}                                  // массиве
```

Этот фрагмент некорректно работает, если в массиве имеются равные элементы. Объясните поведение программы в такой ситуации и предложите решение проблемы.

Сортировки рекурсивным разделением. Сортировки разделяют массив на две части относительно некоторого значения, называемого **медианой**. Медианой может быть выбрано любое «среднее» значение, например, среднее арифметическое. Сами части не упорядочены, но обладают таким свойством, что элементы в левой части меньше медианы, а в правой – больше. Благодаря такому свойству эти части можно сортировать независимо друг от друга. Для этого нужно вызвать ту же самую функцию сортировки, но уже по отношению не к массиву, а к его частям. Функции, вызывающие сами себя, называются рекурсивными и рассмотрены в разделе 3.4. Рекурсивный вызов продолжается до тех пор, пока очередная часть массива не станет содержать единственный элемент:

```
//---- Схема сортировки рекурсивным разделением
void sort(int in[], int a, int b){
int i;
if (a>=b) return;
        // Разделить массив в интервале a..b
        // на две части a..i-1 и i..b
        // относительно значения v по принципу <v, >=v
sort(in,a,i-1);   sort(in,i,b);}
```

Технический момент: разделение лучше всего производить в отдельном массиве (пример разделения приведен в разделе 1.2), после чего разделенные части перенести обратно. Кроме того, нужно следить, чтобы разделяемые части содержали хотя бы один элемент.

«Быстрая» сортировка умудряется произвести разделение в одном массиве с использованием оригинального алгоритма на основе обмена. Сравнение элементов производится с концов массива (**i=a, j=b**) к середине (**i++** или **j--**), причем «укорочение» происходит только с одной из сторон. После каждой перестановки меняется тот конец, с которого выполняется «укорочение». В результате этого массив разделяется на две части относительно значения первого элемента **in[a]**, который и становится медианой.

```
//-----------------------------------------------------25-13.cpp
//-------"Быстрая" сортировка
 void sort(int in[], int a, int b){
 int i,j,mode;
 if (a>=b) return;                    // Размер части =0
 for (i=a, j=b, mode=1; i < j; mode >0 ? j-- : i++)
       if (in[i] > in[j]){            // Перестановка концевой пары
             int c = in[i]; in[i] = in[j]; in[j]=c;
```

```
            mode = -mode;       // со сменой сокращаемого конца
           }
sort(in,a,i-1); sort(in,i+1,b);}
```

Очевидно, что медиана делит массив на две неравные части. Алгоритм разделения можно выполнить итерационно, применяя его к той части массива, которая содержит его середину (по аналогии с двоичным поиском). Тогда в каждом шаге итерации медиана будет сдвигаться к середине массива.

Сортировка слиянием. Алгоритм слияния упорядоченных последовательностей рассмотрен в разделе 1.2. На практике слияние эффективно при работе с данными большого объема в последовательных файлах, где принцип слияния последовательно читаемых данных «без заглядывания вперед» выглядит естественно.

Простое однократное слияние базируется на других алгоритмах сортировки. Массив разбивается на **n** частей, каждая из них сортируется независимо, а затем отсортированные части объединяются слиянием. Реально такое слияние используется, если массив целиком не помещается в памяти. В данной простой модели одномерный массив разделяется на 10 частей – используется двумерный массив из 10 строк по 10 элементов. Затем каждая строка сортируется отдельно. Алгоритм слияния использует стандартные контексты: выбирается строка, в которой первый элемент минимальный (минимальный из очередных), он-то и «сливается» в выходную последовательность. Исключение его производится сдвигом содержимого строки к началу, причем в конец добавляется «очень большое число», играющее роль «затычки» при окончании этой последовательности.

```
//-----------------------------------------------------25-14.cpp
//------- Простое однократное слияние
 void sort(int a[], int n); // Любая сортировка одномерного массива
 #define N 4               // Количество массивов
 void big_sort(int A[], int n){
 int B[N][10];
 int i,j,m = n/N;                                  // Размерность массивов
 for (i=0; i<n; i++) B[i/m][i%m]=A[i];             // Распределение
 for (i=0; i<N; i++) sort(B[i],10);                // Сортировка частей
 for (i=0; i<n; i++){                              // Слияние
        for ( int k=0, j=0; j<N; j++)          // Индекс строки с минимальным
              if (B[j][0] < B[k][0]) k=j;         // B[k][0]
        A[i] = B[k][0];                           // Слияние элемента
        for (j=1; j<m; j++) B[k][j-1]=B[k] [j]; // Сдвиг сливаемой строки
        B[k][m-1]=10000;                          // Запись ограничителя
        }}
```

Циклическое слияние. Оригинальный алгоритм «сортировки без сортировки» базируется на том факте, что при слиянии двух

упорядоченных последовательностей длиной **s** длина результирующей – в 2 раза больше. Главный цикл включает в себя разделение последовательности на 2 части и их обратное слияние в одну. Первоначально они неупорядочены, тем не менее, можно считать, что в них имеются группы упорядоченных элементов длиной **s=1.** Каждое слияние увеличивает размер группы вдвое, то есть размер группы меняется **s=2, 4, 8...** Поэтому «собака зарыта» в способе слияния: оно не может выйти за пределы очередной группы, пока обе сливаемые группы не закончились. Это значит, переход к следующей паре осуществляется «скачком» (рис. 2.7).

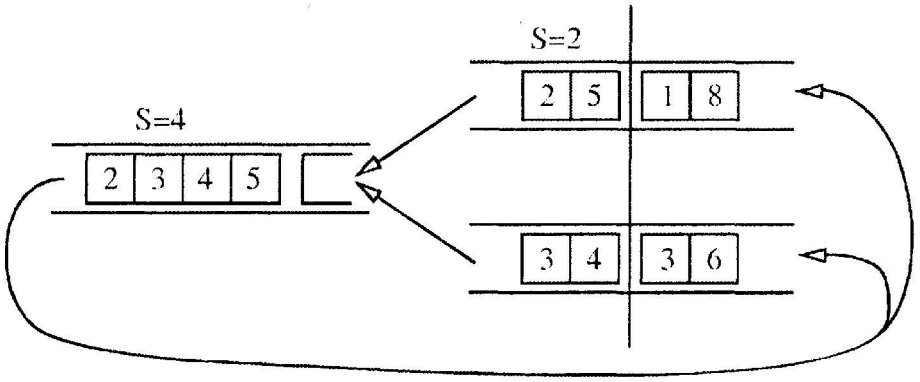

Рис. 2.7

В приведенной программе для простоты размерность массива должна быть равна степени 2, чтобы группы были всегда полными. Внешний цикл организован формально: переменная **s** принимает значения степени 2. В теле цикла сначала производится разделение массива на две части, а затем – их слияние. Для успешного проектирования слияния важно правильно выбрать индексы с учетом независимости и относительности «движений» по отдельным массивам. Поэтому их здесь целых четыре на три массива. Индекс **i** в выходном массиве увеличивается в заголовке цикла. Это значит, что за один шаг цикла один элемент из входных последовательностей переносится в выходную. Движение по группам разложено на две составляющие: **k** – общий индекс начала обеих групп, а **i1, i2** – относительные индексы внутри групп. Здесь же отрабатывается «скачок» к следующей паре групп: при условии, что обе группы закончились (**i1==s && i2==s**), обнуляются относительные индексы в группах, а индекс начала увеличивается на длину группы. В процессе слияния отрабатываются четыре возможные ситуации: завершение первой или второй группы и выбор минимального из пары очередных элементов групп – в противном случае.

```
//-----------------------------------------------------25-15.cpp
//------ Циклическое двухпутевое слияние ( n равно степени 2)
 void sort(int A[], int n){
 int B1[100],B2[100];
 int i,i1,i2,s,a1,a2,a,k;
 for (s=1; s!=n; s*=2){                    // Размер группы кратен 2
      for (i=0; i<n/2; i++)                // Разделить пополам
            { B1[i]=A[i]; B2[i]=A[i+n/2]; }
      i1=i2=0;
      for (i=0,k=0; i<n; i++){      // Слияние с переходом " скачком"
            if (i1==s && i2==s)             // при достижении границ
                  k+=s,i1=0,i2=0;           // обеих групп
            if (i1==s) A[i]=B2[k+i2++];
            else                      // 4 условия слияния по окончании
            if (i2==s) A[i]=B1[k+i1++];
            else                      // групп и по сравнению
            if (B1[k+i1 ] < B2[k+i2 ]) A[i]=B1[k+i1++];
            else A[i]=B2[k+i2++];
      }}}
```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Алгоритм сортировки реализовать в виде функции, возвращающей в качестве результата характеристику трудоемкости алгоритма (например, количество сравнений). Если имеется базовый алгоритм сортировки (для Шелла – «пузырек», для Шейкер – «пузырек», для вставки с двоичным поиском – погружение), то аналогично оформить базовый алгоритм и сравнить эффективность.

1. Сортировка вставками. Место помещения очередного элемента в отсортированную часть определить с помощью двоичного поиска. Двоичный поиск оформить в виде отдельной функции.

2. Сортировка Шелла. Частичную сортировку с заданным шагом, начиная с заданного элемента, оформить в виде функции. Алгоритм частичной сортировки – вставка погружением.

3. Сортировка разделением. Способ разделения: вычислить среднее арифметическое всех элементов массива и относительно этого значения разбить массив на две части (с использованием вспомогательных массивов).

4. Шейкер-сортировка. Движение в прямом и обратном направлениях реализовать в виде одного цикла, используя параметр – направление движения (**+1/–1**) и меняя местами нижнюю и верхнюю границы просмотра.

5. «Быстрая» сортировка с итерационным циклом вычисления медианы. Для заданного интервала массива, в котором производится разделение, найти медиану обычным способом. Затем выбрать ту часть интервала между границей и медианой, где находится середина исходного интервала, и процесс повторить.

6. Сортировка циклическим слиянием с использованием одного выходного и двух входных массивов. Для упрощения алгоритма и разграничения сливаемых групп в последовательности в качестве разделителя добавить «очень большое значение» (**MAXINT**).

7. Сортировка разделением. Медиана – среднее между минимальным и максимальным значениями элементов массива. Относительно этого значения разбить массив на две части (с использованием вспомогательных массивов).

8. Простое однократное слияние. Разделить массив на **n** частей и отсортировать их произвольным методом. Отсортированный массив получить однократным слиянием упорядоченных частей. Для извлечения очередных элементов из упорядоченных массивов использовать массив из **n** индексов (по одному на каждый массив).

9. Сортировка подсчетом. Выходной массив заполнить значениями «–1». Затем для каждого элемента определить его место в выходном массиве подсчетом количества элементов, строго меньших, чем данный. Естественно, что все одинаковые элементы попадают на одну позицию, за которой следует ряд значений «–1». После этого оставшиеся в выходном массиве позиции со значением «–1» заполнить копией предыдущего значения.

10. Сортировка выбором. Выбрать минимальный элемент в массиве и запомнить его. Затем удалить, а все последующие за ним элементы сдвинуть на один влево. Сам элемент занести на освободившуюся последнюю позицию.

11. Сортировка вставками. Извлечь из массива очередной элемент. Затем от начала массива найти первый элемент, больший, чем данный. Все элементы, от найденного до очередного сдвинуть на один вправо, и на освободившееся место поместить очередной элемент. (Поиск места включения от начала упорядоченной части.)

12. Сортировка выбором. Выбрать минимальный элемент в массиве, перенести в выходной массив на очередную позицию и заменить во входном на «очень большое значение» (**MAXINT**).

13. Сортировка Шелла. Частичную сортировку с заданным шагом, начиная с заданного элемента, оформить в виде функции. Алгоритм частичной сортировки – обменная (методом «пузырька»).

14. Сортировка выбором. Выбрать минимальный элемент в массиве, перенести в выходной массив на очередную позицию. Во входном массиве все элементы от следующего за текущим до конца сдвинуть на один влево.

15. Сортировка «хитрая». Из массива однократным просмотром выбрать последовательность элементов, находящихся в порядке

возрастания, перенести в выходной массив и заменить во входном на «–1». Затем оставшиеся элементы включить в полученную упорядоченную последовательность методом погружения.

16. Оптимизированный двоичный поиск. В процессе деления выбрать не середину интервала, а значение, вычисленное из предположения о линейном возрастании значений элементов массива в текущем интервале поиска. Сравнить эффективности разработанного и базового алгоритмов на массивах с резко неравномерным возрастанием значений (например, 1, 2, 2, 3, 4, 25).

ТЕСТОВЫЕ ЗАДАНИЯ И ПРОГРАММНЫЕ ЗАГОТОВКИ

По тексту программы определите алгоритм сортировки, «смысл» отдельных переменных и назначение циклов.

```
//-----------------------------------------------------25-16.cpp
//------------------------------------------------ 1
 void F1(int in[],int n)
 { int i,j,k,c;
 for (i=1; i<n; i++){
     for (k=i; k !=0; k--)
         { if (in[k] > in[k-1]) break;
         c=in[k]; in[k]=in[k-1]; in[k-1]=c;
         }
     }}
 //------------------------------------------------ 2
 void F2(int in[],int out[],int n)
 { int i,j ,cnt;
 for (i=0; i< n; i++) {
     for ( cnt=0,j=0; j<n; j++)
         if (in[j] > in[i]) cnt++;
     else
         if (in[j]==in[i] && j>i) cnt++;
     out[cnt]=in[i];
     } }
 //------------------------------------------------ 3
 void F3(int in[],int n)
 { int a,b,dd,i,lasta,lastb,swap;
 for (a=lasta=0, b=lastb=n, dd=1; a < b; dd = !dd, a=lasta, b=lastb){
       if (dd){
             for (i=a,lastb=a; i<b; i++)
                   if (in[i] > in[i+1]){
                         lastb = i;
                         swap = in[i]; in[i]=in[i+1]; in[i+1]=swap;
                   }
       }
       else {
            for (i=b,lasta=b; i>a; i--)
                  if (in[i-1] > in[i]){
                  lasta = i;
                  swap = in[i]; in[i]=in[i-1]; in[i-1]=swap;
                  }}}}
```

```
//------------------------------------------------ 4
int find(int out[],int n, int val);
// Двоичный или линейный поиск расположения значения val
// в массиве out[n]
void F4(int in[], int n){
int i,j,k;
for (i=1; i<n; i++)
    { int c; c = in[i]; k = find(in,i,c);
    for (j=i; j!=k; j--) in[j] = in[j-1];
    in[k] = c; }
}
//------------------------------------------------ 5
void F5(int in[], int n){
int i,j,c,k;
for (i=0; i < n-1; i++){
    for (j=i+1,c=in[i],k=i; j<n; j++)
            if (in[j] > c) { c = in[j]; k=j; }
    in[k] = in[i]; in[i] = c;
    }}
//------------------------------------------------ ---6
void F6(int A[], int n){
int i,found;
do { found =0;
        for (i=0; i<n-1; i++)
        if (A[i] > A[i+1])
                { int cc; cc = A[i]; A[i]=A[i+1]; A[i+1]=cc;
                found++;
                }
        } while(found !=0); }
//------------------------------------- -----------7
void sort(int a[], int n); //Любая сортировка одномерного массива
#define MAXINT 1000
int A[100], B[10][10];
void F7(){
int i,j;
for (i=0; i<100; i++) B[i/10][i%10]=A[i];
for (i=0; i<10; i++) sort(B[i],10);
for (i=0; i<100; i++){
        int k;
        for (k=0, j=0; j<10; j++)
                if (B[j][0] < B[k][0]) k=j;
        A[i] = B[k][0];
        for (j=1; j<10; j++) B[k][j-1]=B[ k] [j];
        B[k][9]=MAXINT;
        }}
//-------------------------------------------------8
void F8(int in[], int a, int b){
int i,j,mode;
if (a >=b) return;
for (i=a, j=b, mode=1; i < j; mode >0 ? j-- : i++)
      if (in[i] > in[j]){
              int c = in[i]; in[i] = in[j]; in[j]=c;
              mode = -mode;
      }
F8(in,a,i-1); F8(in,i+1,b); }
//-------------------------------------------------9
void F9(int A[], int n){
```

```
int i,b,b1;
for (b=n-1; b!=0; b=b1) {
        b1=0;
        for (i=0; i<b; i++)
             if (A[i] > A[i+1]) {
                  int cc = A[i]; A[i]=A[i+1]; A[i+1]=cc;
                  b1=i;
            }}}
//-----------------------------------------------10
void F10(int A[], int B1[], int B2[], int n){
int i,i1,i2,s,a1,a2,a,k;
for (s=1; s!=n; s*=2){
      for (i=0; i<n/2; i++)
              { B1[i]=A[i]; B2[i]=A[i+n/2]; }
       i1=i2=0; a1=a2=MAXINT;
       for (i=0,k=0; i<n; i++)
                {
                if (a1==MAXINT && a2==MAXINT && i1==s && i2==s)
                          k+=s,i1=0,i2=0;
                if (a1==MAXINT && i1!=s) a1=B1[k+i1],i1++;
                if (a2==MAXINT && i2!=s) a2=B2[k+i2],i2++;
                if (a1<a2)a=a1,a1=MAXINT;
                else a=a2,a2=MAXINT;
                A[i]=a;
                }}}
```