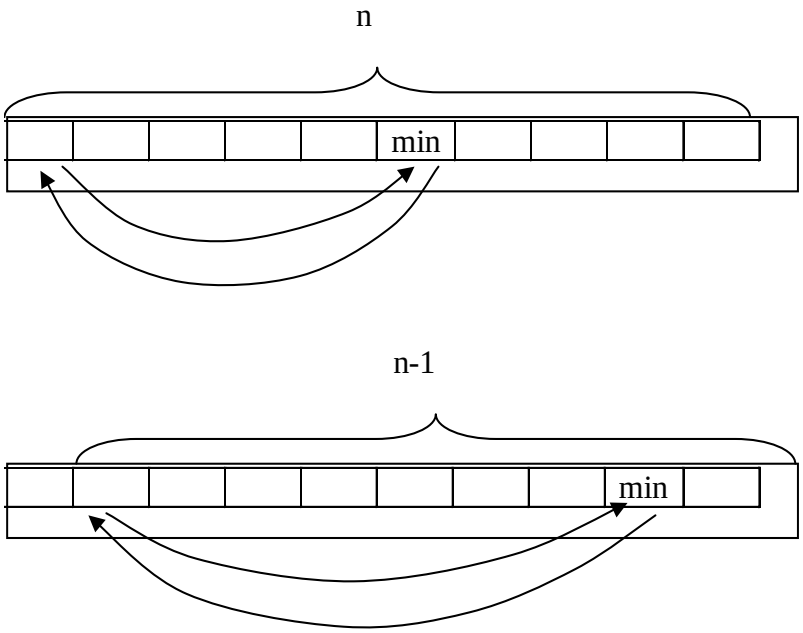
Сортировки

..1. Сортировка выбором

Один из самых простых алгоритмов (выбором – так как он работает по принципу выбора наименьшего элемента из оставшихся):

1) отыскивается наименьший элемент массива и меняется местами с первым
2) из оставшихся элементов снова находится наименьший и меняется местами со вторым

…



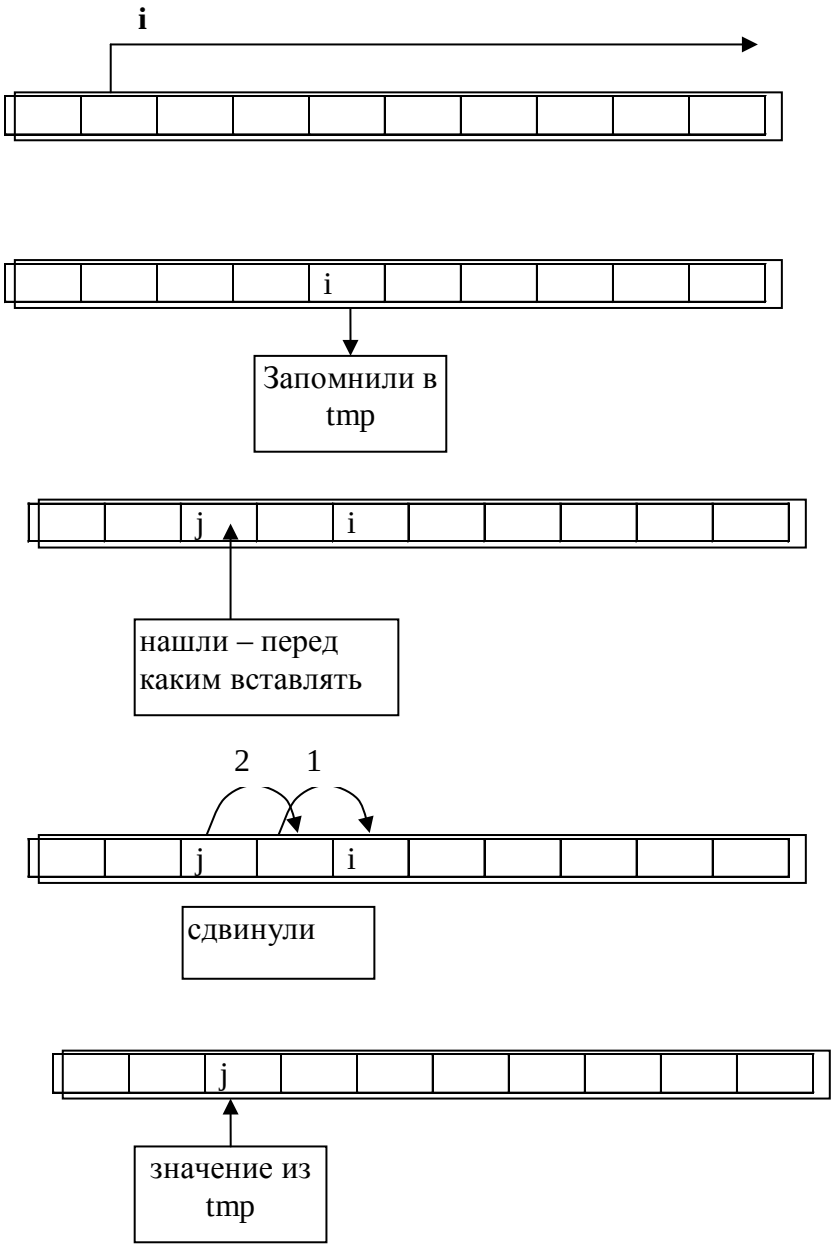

```
int ar[]={7,2,-5,6,0,3,2,5,3,-9,99,100};
int n = sizeof(ar)/sizeof(int);
for(int i=0; i<n-1; i++)  // n-1, так как последний
                          автоматически окажется на своем месте
{
     //Поиск минимального из оставшихся значений
     int min = i; //здесь будет индекс элемента с минимальным
                    значением
     for(int j=i+1; j<n; j++)//       {
          if(ar[j]<ar[min])       min=j;
     }
     //Обмен местами текущего с минимальным
     int tmp = ar[min];
     ar[min] = ar[i];
     ar[i] = tmp;
}
```

..2. Сортировка вставками

Вместо перестановки местами используется сдвиг

1) все предыдущие элементы отсортированы

2) для каждого очередного элемента находится место в уже отсортированном промежутке – перед каким вставлять
3) для освобождения места для вставки часть массива сдвигается вправо, начиная с самого правого элемента
4) текущий элемент вставляется в освободившееся место среди отсортированных




```
int ar[]={7,2,5,6,1,3,2,5,3,9,-1};
int n = sizeof(ar)/sizeof(int);

for(int i=1; i<n; i++)//Считаем, что нулевой элемент
                      "упорядочен"
{
     int tmp = ar[i];//запомнили значение текущего
     //нашли "куда" вставлять
     for(int j =0; j<i; j++)
     {
```

```
            if(ar[j]>tmp) break;//вставлять перед j-тым
        }
        //сдвинули от i до j на одну позицию вправо
        for(int k = i; k>j; k--)//начинаем сдвигать "с справого",
                    так как i-тый свободен
        {
                ar[k] = ar[k-1];
        }
        ar[j] = tmp; //вставили сохраненное значение на освободившееся место
```

..3. Пузырьковая сортировка

Обычно осваивают раньше других:

1) проход по данным с обменом местами соседних элементов, нарушающих заданный порядок => наверху оказывается наибольший или внизу наименьший
2) повтор по оставшейся неупорядоченной части массива

Замечание: так как элементы массива при каждом проходе все время упорядочиваются =>

1) на каждой итерации перестановок становится меньше
2) необязательно продолжать итерации до конца, а можно завести флажок – если на очередной итерации не произошло ни одной перестановки, все элементы уже стоят на своих местах => выход

```
    //Сортировка всплывающий пузырек - за одну итерацию - наверх
                    наибольший
    {
    //Цикл сортировки методом "всплывающего пузырька" в
    //порядке возрастания
        int ar[]={-5,3,-9,1,-11,0,22};
        int n = sizeof(ar)/sizeof(int);
        for(int i=0; i<n-1; i++)
        {
            bool b = false;//флажок - была ли хотя бы одна
                    перестановка
            for(int j=0; j<n-i-1; j++)
            {
                if(ar[j] > ar[j+1])
                {
                    //Переставляем местами элементы:
                    int Tmp=ar[j];
                    ar[j]=ar[j+1];
                    ar[j+1]=Tmp;
                    b=true;
                }
            }

            if(b==false)
                break;//для данных значений за 4 итерации все уже будет
                    упорядочено => продолжать не имеет смысла
            stop
        }

    }
```

```
{//за одну итерацию - вниз наименьший
//Цикл сортировки методом "всплывающего пузырька" в
//порядке возрастания
     int ar[]={7,2,5,6,1,3,2,5,3,9,1};
     int n = sizeof(ar)/sizeof(int);
     for(int i=1; i<n; i++)
     {
          bool b = false;//флажок - была ли хотя бы одна
                         перестановка
          for(int j=n-1; j>=i; j--)
          {
               if(ar[j-1] > ar[j])
               {
                    //Переставляем местами элементы:
                    int Tmp=ar[j];
                    ar[j]=ar[j-1];
                    ar[j-1]=Tmp;
                    b=true;
               }
          }

          if(b==false)
               break;
          stop
     }

}
```

..4. Сортировка Шелла

Основная идея: сначала сравниваются удаленные элементы, а не смежные. Это приводит к устранению большей части неупорядоченности => сокращает последующую работу. Интервал между элементами постепенно сокращается до единицы, тогда сортировка фактически превращается в метод перестановки соседних элементов (но к этому моменту они практически уже все упорядочены!).
Три вложенных цикла:
i – управляет интервалом между сравниваемыми элементами, уменьшая его в /n-раз при каждом проходе, пока он не станет равным 0.
j – сравнивает каждую пару элементов, разделенных на величину интервала (i).
k – переставляет каждую неупорядоченную пару.

```
{
     int ar[]={7,2,5,6,1,3,2,5,3,9,1};
     int n = sizeof(ar)/sizeof(int);
     for(int i=n/2; i>0; i/=2)
          for(int j=i; j<n; j++)
               for(int k=j-i; k>=0 && ar[k]>ar[k+i]; k-=i)
               {
                    int tmp = ar[k];
                    ar[k] = ar[k+i];
                    ar[k+i] = tmp;
```

```
                }
    }
```

..5. Алгоритм QuickSort

Нерекурсивный вариант – Давыдов 244, рекурсивный (Хоара)- Сэджвик 300

Самым быстрым алгоритмом сортировки массивов является алгоритм QuickSort. Реализован по принципу: «разделяй и властвуй!»

Он является рекурсивным, что создает некоторые трудности при его осмыслении.

Смысл алгоритма: последовательность разбивается на подпоследовательности, каждая из которых сортируется независимо.

1) произвольно выбираем некоторый пограничный (разделяющий) элемент и считаем, что он «стоит на своем месте»
2) просматриваем массив с левого конца и находим элемент, значение которого больше, чем значение пограничного
3) просматриваем массив с правого конца и находим элемент, значение которого меньше, чем значение пограничного
4) очевидно, что оба элемента стоят не на своем месте => меняем их местами
5) продолжаем дальше в том же духе, пока не убедимся, что слева не осталось ни одного элемента, значение которого больше пограничного

```
void QuickSort (int *ar, int l, int r)//l – левая граница, r - правая

{
    //if((r-l)<2) return;//если меньше двух элементов, не имеет
                         смысла => выходим
    int mid = ar[(l + r) / 2];  //пограничный элемент. Может быть
                         выбран в диапазоне [l,r] случайным образом. Я
                         задаю центральный элемент

    int i = l, j = r;           // Левая и правая границы интервала
    do                                  // Цикл, сжимающий интервал
    {
                // Поиск элементов, нарушающих порядок (слева и
                   справа)
        while (ar[i] < mid) i++;    //ищем слева большее значение.
                         Как только дойдем до mid, условие станет
                         false!!! => выйдем из цикла
        while (mid < ar[j] ) j--;   // ищем справа меньшее
                         значение.
                         Как только дойдем до mid, условие станет
                         false!!! => выйдем из цикла

        if (i <= j)                             // Если нашли оба
        {   // то производим обмен
            int temp = ar[i];
            ar[i++] = ar[j];//i сдвигаем на следующий
            ar[j--] = temp;//j сдвигаем на следующий
        }//Если i==j, то нужно просто сдвинуть i и j, но выделять
                         это сравнение отдельно дороже???
    }while (i </*=*/ j);          // Пока индексы не пересеклись

    if (l < j)                  //Если левая часть не упорядочена,
        QuickSort (ar, l, j);           // то занимаемся ею
    if (i < r)                  //Если правая часть не упорядочена,
        QuickSort (ar, i, r);           // то занимаемся ею
```

```
} //иначе выход из рекурсии – все упорядочено!

void main()
{
     {//QuickSort
          int ar[] = {1, 5, 2, 10, 4, 8, 11, -25, -1, 9};
          int N = sizeof(ar)/sizeof(int);
          QuickSort(ar,0,N-1);
          stop

     }
```

Функция QuickSort реализует рекурсивный алгоритм быстрой сортировки, а функция `main` иллюстрирует его применение для массива целых из 21 элемента. Алгоритм считается одним из самых эффективных в смысле количества необходимых операций для выполнения работы. Идея его состоит в том, что меняются местами элементы массива, стоящие слева и справа от выбранного «центрального» (`mid`) элемента массива, если они нарушают порядок последовательности. Интервал, в котором выбирается центральный элемент, постепенно сжимается, «расправляясь» сначала с левой половиной массива, затем с правой.

Производительность:
В самом плохом варианте - N*N/2??? (N в степени N) сравнений
В самом хорошем (в отсортированной последовательности) – (N+1) *N/2

..6. Простой поиск перебором

```
int* SimpleSearch(int* ar, int what, int size)
{
     for(int i = 0; i<size; i++) if(ar[i]==what) break;
     return & ar[i];
}
```

..7. Бинарный поиск

Работает с отсортированными последовательностями!!!
Алгоритм основан на принципе «разделяй и властвуй!»:

1) последовательность разделяется на две части
2) определяется – к какой из подпоследовательностей относится искомое значение
3) то же самое повторяется для подпоследовательностей

Существуют два варианта реализации: рекурсивный и нерекурсивный

```
int* BinarySearch(int *ar, int l, int r, int what)
{
      if(l>r) return 0;//признак - не найден
      int i = (l+r)/2;//центральный в заданном диапазоне
      if(ar[i]==what) return &ar[i];//средний и есть искомый
      if(what <ar[i]) return BinarySearch(ar,l,i-1,what);//ищем слева
      else return BinarySearch(ar,i+1,r,what);//ищем справа


}
{//QuickSort
           int ar[] = {55, 15, 2, 0, -4, 8, 11, -25, -1, 9};
           int N = sizeof(ar)/sizeof(int);
           QuickSort(ar,0,N-1);
      int* p = BinarySearch(ar,0,N-1,-25);

      }
```