STEP computer ACADEMY

PROGRAMMING C

Lesson No. 10

Programming C

Contents

1. Linear search

Within the present lesson we are going to talk about search and sort algorithms. You might have already faced the necessity to arrange your array or quickly find some data within it. Well, today we are going to try to help you to automatize these processes.

At first we will study the simplest data search method — linear search.

This algorithm compares each array element with the search key. Our experimental array is not arranged and a situation may occur when a value in search will be the first within an array. But in general a program realizing a linear search compares half of the elements with the search key.

```
#include <iostream>

using namespace std;

int LinearSearch (int array[], int size, int key){
   for(int i=0;i<size;i++)
       if(array[i] == key)
           return i;
   return -1;
}

void main()
{
   const int arraySize=100;
   int a[arraySize], searchKey, element;
   for(int x=0;x<arraySize;x++)
           a[x]=2*x;

   //The next string displays a message
   //Input a search key:
   cout<<"Please, enter the key:  ";
```

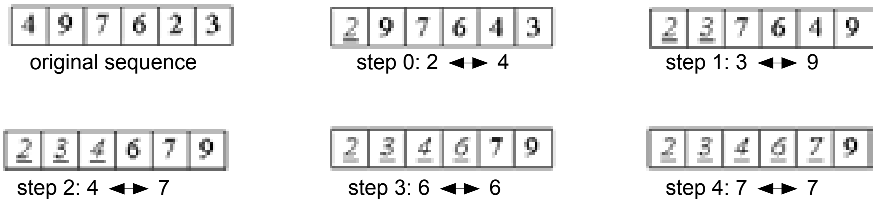
```
    cin>>searchKey;
    element=LinearSearch(a, arraySize, searchKey);

    if(element!=-1)
      // The next string displays a message
      //Element value found
      cout<<"\nThe key was found in element "<<element<<'\n';

    // The next string displays a message
    //Value not found
    else
      cout<<"\nValue not found ";

}
```

Note that a linear search algorithm is a reliable mechanism that perfectly operates only with the small or disordered arrays.

2. Selection sort

This method idea implies creating a sorted sequence by means of adding to it one element after the other in the correct order.

Now we will try to build a ready sequence starting from the left array end. The algorithm consists of n successive steps starting from zero and ending with (n–1). At the i-step we choose the least of the elements a[i] ... a[n] and switch their places with a[i]. Step sequence upon n=5 is shown at the picture below.



Regardless of the number of the current step i, the sequence a[0]...a[i] is an arranged one. Therefore, at the step (n–1) the whole sequence except a[n] it proves sorted and a[n] occupies its fair last position: all smaller elements have been mover left-wise. Let’s study an example realizing this method:

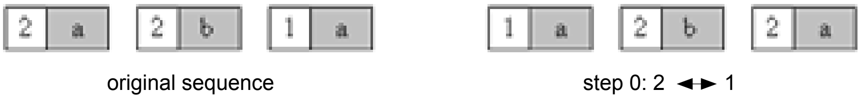
```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
template <class T>
void selectSort(T a[], long size) {
    long i, j, k;
    T x;

    for(i=0;i<size;i++) {            // i - number of current step
       k=i;
       x=a[i];
       // the smallest elements selection loop
       for(j=i+1;j<size;j++)
          if(a[j]<x){
            k=j;
            x=a[j];
            // k - the smallest element index
          }
       a[k]=a[i];
       a[i]=x;          // swaps places of the smallest and a[i]
   }
}

void main(){
        srand(time(NULL));
        const long SIZE=10;
        int ar[SIZE];

        // before sorting
        for(int i=0;i<SIZE;i++){
           ar[i]=rand()%100;
           cout<<ar[i]<<"\t";
        }
        cout<<"\n\n";
        selectSort(ar,SIZE);

        // after sorting
        for(int i=0;i<SIZE;i++){
           cout<<ar[i]<<"\t";
        }
        cout<<"\n\n";
}
```

Main method principles

1. 1. in order to find the smallest element out of n+1 under consideration and algorithm makes n-number of matches. Therefore, since number of switches is always less than number of matches, sort time increases in relation to number of elements.

2. Algorithm does not employ extra memory: all operations are run «at place».

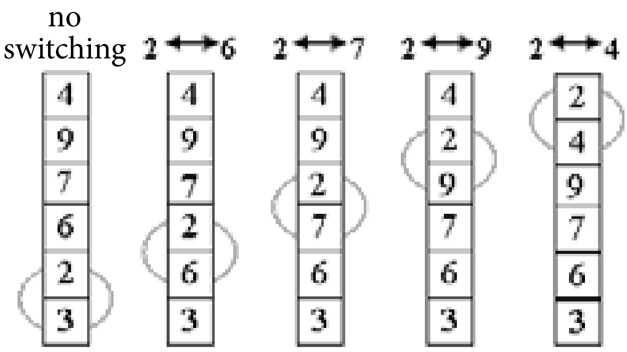
Let's find out how stable this method is? Review the sequence of three elements with two fields in each of them, sorting is exercised on the first one. We can find its sort results right after the step 0, since there are no more switches to be done. An order of the keys 2a, 2b was changed into 2b, 2a.



Therefore, input sequence is almost arranged and there is going to be the same number of matches, it means the algorithm behaves not in the most optimal way. However, this kind of sort can be used for small-sized arrays.

3. Bubble sort

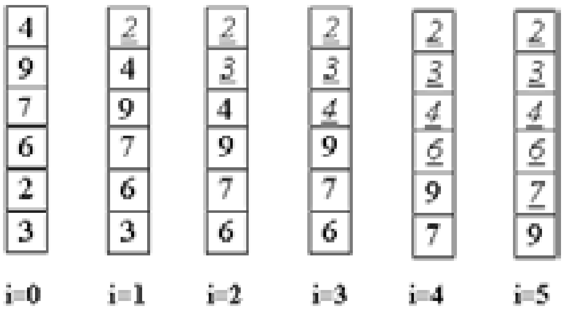
The method idea implies the following: sort step is involves passing through an array bottom-up. Pairs of the neighboring elements are considered alongside. If some pair elements are in a wrong order we switch their places. To realize it let us place an array upside down from zero element to the last one. After zero passage through the array there is the «lightest» element on the top of the array, here is an analogy with the bubble. The next passage is exercised till the top second element, thus the second value element lifts to the correct position.



Zero passage, matched pairs are marked

Let's undertake the passages along the decreasing lower array part till there is only one element remains. Sort stops thereon, since the sequence is arranged in ascending order:

Code example:



passage number

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
template <class T>
void bubbleSort(T a[], long size){
    long i, j;
    T x;
    for(i=0;i<size;i++){              // i - passage number
       for(j=size-1;j>i;j--){       // internal passage loop
          if(a[j-1]>a[j]){
            x=a[j-1];
            a[j-1]=a[j];
            a[j]=x;
          }
       }
    }
}

void main(){
    srand(time(NULL));
    const long SIZE=10;
    int ar[SIZE];

    // before sort
    for(int i=0;i<SIZE;i++){
       ar[i]=rand()%100;
       cout<<ar[i]<<"\t";
    }
    cout<<"\n\n";
    bubbleSort(ar,SIZE);

    // after sort
    for(int i=0;i<SIZE;i++){
       cout<<ar[i]<<"\t";
    }
    cout<<"\n\n";
}
```

Main method principles.

An average number of matches and switches have a squared order of growth, hence we can conclude that bubble algorithm is very slow and low-efficient. Nonetheless, it has a big plus: it is simple and can be enhanced in any desired way. That is actually what we are going to do right now.

Let us study a situation when there was no switch within a passage. It means that all pairs are positions in the correct order and the array is sorted. And there is no sense to continue. Therefore, the first optimization step is memorizing whether there was any switch at the particular passage. If no the algorithm terminates its operation.

We continue optimization process with not only memorizing the switch fact, but also last switch index k. in fact: all pairs of neighboring elements with the indexes smaller than k are arranged in a proper order. Next passages can be terminated at the index k instead of moving further till the preset upper limit i.

Principally other algorithm enhancement can be carried out from the following observation. Though a light «bubble» lifts bottom-up per one passage, «heavy» bubbles lower with minimal speed: one per iteration. So that the array 2 3 4 5 6 1 is to be sorted per 1 passage, and sort of the sequence 6 1 2 3 4 5 will require 5 passages.

To avoid this effect we can change the direction of the passages following one by one. An obtained algorithm is sometimes called «shaker-sort».

Therefore, we have found out that we can optimize the bubble sort according our desire. Knock yourself out!!!

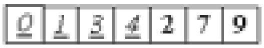
```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
template <class T>
void shakerSort(T a[], long size) {
  long j, k=size-1;
 long lb=1, ub=size-1; // boundaries of non-sorted array part
  T x;
  do{
      // bottom-up passage
      for(j=ub;j>0;j--){
         if(a[j-1]>a[j]){
           x=a[j-1];
           a[j-1]=a[j];
           a[j]=x;
           k=j;
         }
      }
      lb = k+1;
      // upside down passage
      for(j=1;j<=ub;j++){
         if(a[j-1]>a[j]){
           x=a[j-1];
           a[j-1]=a[j];
           a[j]=x;
           k=j;
         }
       }
      ub=k-1;
  }while (lb<ub);
}
void main(){
   srand(time(NULL));
   const long SIZE=10;
   int ar[SIZE];

   // before sort
   for(int i=0;i<SIZE;i++){
      ar[i]=rand()%100;
      cout<<ar[i]<<"\t";
   }
   cout<<"\n\n";
   shakerSort(ar,SIZE);
   // after sort
   for(int i=0;i<SIZE;i++){
      cout<<ar[i]<<"\t";
   }
   cout<<"\n\n";
}
```

4. Insertion sort

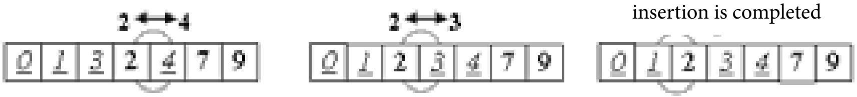
Simple insertion sort is somehow similar to the methods stated in the previous paragraphs of our lesson. Analogically the passages along the array parts are made, and analogically in its beginning there will «grow» a sorted sequence.

But in bubble or selection sort we can strictly claim that at the first step i elements a[0]...a[i] are in the correct positions and are not going to be moved. This assertion would be quite weak speaking about insertion sort: sequence a[0]...a[i] is arranged. Wherein, new elements will be inserted in the course of the algorithm operation.

Let's study the algorithm from its first step i. As we said before the sequence up to this moment is divided into two parts: arranged a[0]...a[i] and disordered a[i+1]...a[n]. At the next (i+1) step of the algorithm we take a[i+1] and insert it to a necessary array part. Search of an appropriate place for the next array element of the input sequence is carried out by means of sequential matches with the element that is before a matched one. Depending on the matching result the element either remains at its current place, or (insert is terminated), or they switch places and the process repeats.



Current sequence. Part a[0]..a[2] is already arranged.



insert of number 2 into a sorted sequence. Matched pairs are marked

Thus, during insertion we «sift» the element x to the top of the array and stop only if there is an element less than x or the beginning of the sequence is reached.

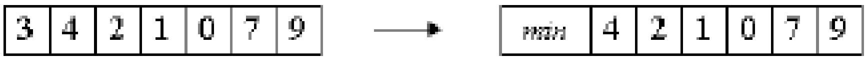
Realizing the method

```
#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;

template <class T>
void insertSort(T a[], long size) {
        T x;
        long i, j;
        // iteration loop, i - number of iteration
        for(i=0;i<size;i++){
           x=a[i];

           // searching an element place within a ready sequence
           for(j=i-1;j>=0&&a[j]>x;j--)
             // move an element rightwards until we reach
             a[j+1]=a[j];
             // the place found, insert element
             a[j+1] = x;
        }
}

void main(){
        srand(time(NULL));
        const long SIZE=10;
        int ar[SIZE];
```

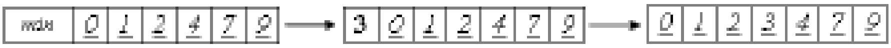
```
         // before sort
        for(int i=0;i<SIZE;i++){
                ar[i]=rand()%100;
                cout<<ar[i]<<"\t";
        }
        cout<<"\n\n";
        shakerSort(ar,SIZE);

        // after sort
        for(int i=0;i<SIZE;i++){
                cout<<ar[i]<<"\t";
        }
        cout<<"\n\n";
}
```

Method principles

A good sort indicator is rather natural behavior: almost sorted array will be quickly sorted. This together with the algorithm stability makes this method a good choice in the relevant situation. However, the algorithm can be enhanced. Note that at each step of the internal loop 2 conditions are checked. We can unite them into one placing them into the beginning of the array a special «sentry element». It should be designedly smaller of the other element array.



Then upon j=0 will be a priori true a[0] <=x. The loop stands to the zero element that was actually a goal of the condition j>=0. Therefore, sorting is exercised in a proper way, and within internal cycle it stands to one smaller match. However, sorted array is incomplete, for the first number disappeared from there. To complete the sort this number should be returned back, and then to insert into a sorted sequence a[1]...a[n].



```
#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;

template <class T>
void setMin(T a[],long size){
        T min=a[0];
        for(int i=1;i<size;i++)
                if(a[i]<min)
                        min=a[i];
        a[0]=min;
}

template <class T>
void insertSortGuarded(T a[], long size) {
        T x;
        long i, j;
        // save old first element
        T backup = a[0];
        // change for a minimal
        setMin(a,size);

        // to sort an array
        for(i=1;i<size;i++){
           x = a[i];

           for(j=i-1;a[j]>x;j--)
              a[j+1]=a[j];

           a[j+1] = x;
        }

        // to insert backup to the correct place
        for(j=1;j<size&&a[j]<backup;j++)
           a[j-1]=a[j];

        // element insertion
        a[j-1] = backup;
}

void main(){
        srand(time(NULL));
        const long SIZE=10;
        int ar[SIZE];

        // before sort
        for(int i=0;i<SIZE;i++){
                ar[i]=rand()%100;
                cout<<ar[i]<<"\t";
        }
        cout<<"\n\n";
        insertSortGuarded(ar,SIZE);

        // after sort
        for(int i=0;i<SIZE;i++){
                cout<<ar[i]<<"\t";
        }
        cout<<"\n\n";
}
```

5. Home assignment

1. Given an array of numbers размерностью 10 элементов. Write a function that sorts an array in ascending or descending order, depending on the third function parameter. If it is equal to 1, sort is carried out in a descending order if 0 then it is a ascending order. First two 2 function parameters is an array and its size, the third parameter is default equal to 1.

2. Given an array of random numbers in the range from –20 to +20. It is necessary to find positions of the last negative elements (the leftmost negative element and и rightmost negative element) and sort elements that are between them.

3. Given an array of 20 integer numbers with the values from 1 to 20.

We should:

- Write a function that scatters array elements in random way;
- To create a random number from the same range and find a position of this random number in the array;
- To sort array elements that are positioned on the left side from the found position in a descending order, array elements that is on the right from the found position in a ascending order.