Оглавление

Линейный поиск

В данном уроке пойдет речь об алгоритмах поиска и сортировки. Вы, наверняка, уже успели столкнуться с необходимостью упорядочить свой массив или быстро найти в нем какие-то данные. Что же, сегодня мы попытаемся помочь вам автоматизировать данные процессы.

Для начала, мы рассмотрим наиболее простой из способов поиска данных - линейный поиск.

Данный алгоритм сравнивает каждый элемент массива с ключом, предоставленным для поиска. Наш экспериментальный массив не упорядочен и, может сложиться ситуация, при которой отыскиваемое значение окажется первым в массиве. Но, в общем и целом, программа, реализующая линейный поиск, сравнит с ключом поиска половину элементов массива.

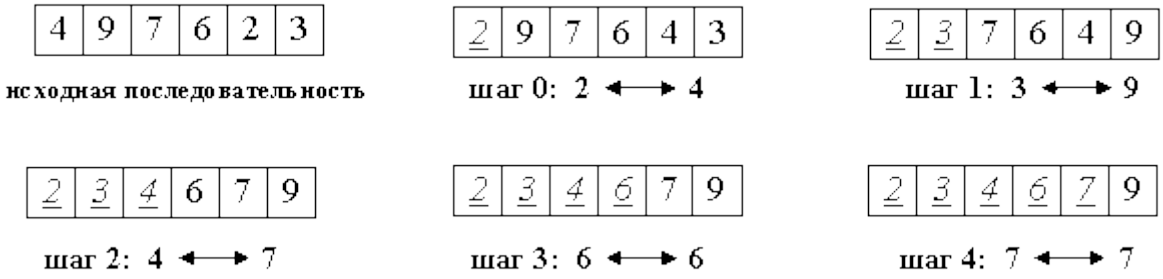
```
#include <iostream>


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;

   //Следующая строка выводит на экран сообщение
   //Введите ключ поиска:
   cout<<"Please, enter the key:  ";
   cin>>searchKey;
   element=LinearSearch(a, arraySize, searchKey);

   if(element!=-1)
     //Следующая строка выводит на экран сообщение
     //Найдено значение в элементе
     cout<<"\nThe key was found in element "<<element<<'\n';

   //Следующая строка выводит на экран сообщение
   //Значение не найдено
   else
     cout<<"\nValue not found ";

}
```

Заметим, что алгоритм линейного поиска отлично работает только для небольших или неупорядоченных массивов и является абсолютно надежным.

Сортировка выбором

Идея данного метода состоит в том, чтобы создавать отсортированную последовательность путем присоединения к ней одного элемента за другим в правильном порядке.

Сейчас, мы с вами попробуем построить готовую последовательность, начиная с левого конца массива. Алгоритм состоит из n последовательных шагов, начиная от нулевого и заканчивая (n-1). На i-м шаге выбираем наименьший из элементов a[i] ... a[n] и меняем его местами с a[i]. Последовательность шагов при n=5 изображена на рисунке ниже.



Вне зависимости от номера текущего шага i, последовательность a[0]...a[i] является упорядоченной. Таким образом, на шаге (n-1) вся последовательность, кроме a[n] оказывается отсортированной, а a[n] стоит на последнем месте по праву: все меньшие элементы уже ушли влево. Рассмотрим пример, реализующий данный метод:

```
#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 - номер текущего шага
       k=i;
       x=a[i];
       // цикл выбора наименьшего элемента
       for(j=i+1;j<size;j++)
          if(a[j]<x){
            k=j;
            x=a[j];
            // k - индекс наименьшего элемента
          }
       a[k]=a[i];
       a[i]=x;          // меняем местами наименьший с a[i]
```

```
        }
    }

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

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

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

Основные принципы метода

1. Для нахождения наименьшего элемента из n+1 рассматримаемых алгоритм совершает n сравнений. Таким образом, так как число обменов всегда будет меньше числа сравнений, время сортировки возрастает относительно количества элементов.

2. Алгоритм не использует дополнительной памяти: все операции происходят "на месте".

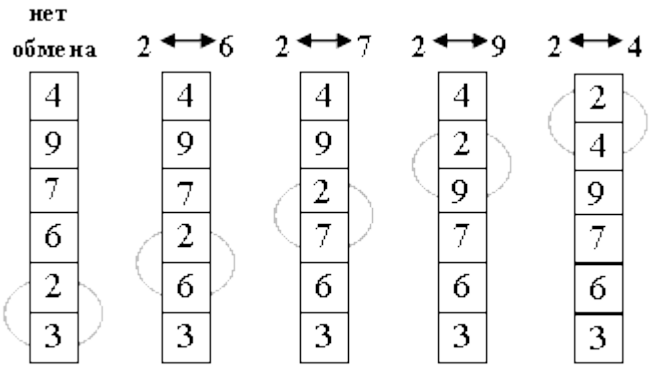
Давайте, определим, насколько устойчив данный метод? Рассмотрим последовательность из трех элементов, каждый из которых имеет два поля, а сортировка идет по первому из них. Результат ее сортировки можно увидеть уже после шага 0, так как больше обменов не будет. Порядок ключей 2a, 2b был изменен на 2b, 2a.

исходная последовательность шаг 0: 2 ⟷ 1

Таким образом, входная последовательность почти упорядочена, то сравнений будет столько же, значит алгоритм ведет себя не очень оптимально. Однако, такую сортировку можно использовать для массивов имеющих небольшие размеры.

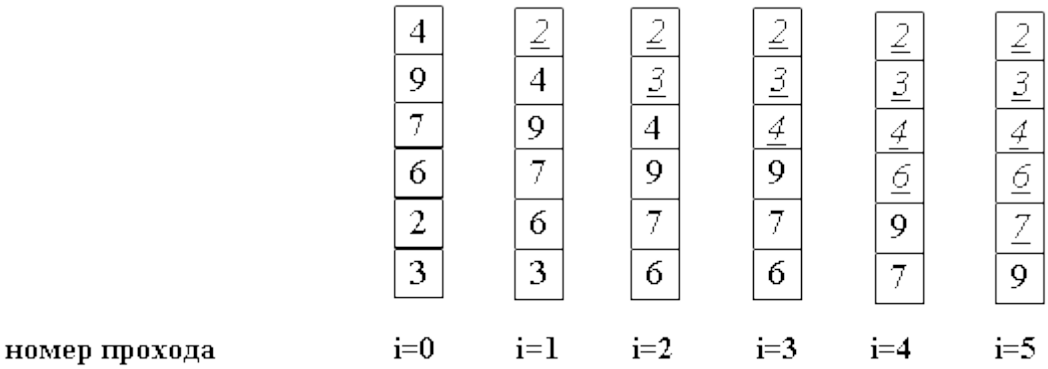
"Пузырьковая" сортировка

Идея метода состоит в следующем: шаг сортировки заключается в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами. Для реализации расположим массив сверху вниз, от нулевого элемента - к последнему. После нулевого прохода по массиву "вверху" оказывается самый "легкий" элемент - отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом второй по величине элемент поднимается на правильную позицию.



Нулевой проход, сравниваемые пары выделены

Делаем проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию.



Пример кода:

```
#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 - номер прохода
           for(j=size-1;j>i;j--){      // внутренний цикл прохода
              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];

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

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

Основные принципы метода.

Среднее число сравнений и обменов имеют квадратичный порядок роста, отсюда можно заключить, что алгоритм пузырька очень медлителен и малоэффективен. Тем не менее, у него есть громадный плюс: он прост и его можно по-всякому улучшать. Чем мы сейчас и займемся.

Рассмотрим ситуацию, когда на каком-либо из проходов не произошло ни одного обмена. Это значит, что все пары расположены в правильном порядке, так что массив уже отсортирован. И продолжать процесс не имеет смысла. Таким образом первый шаг оптимизации заключается в запоминании, производился ли на данном проходе какой-либо обмен. Если нет - алгоритм заканчивает работу.

Процесс оптимизации можно продолжить, запоминая не только сам факт обмена, но и индекс последнего обмена k. Действительно: все пары соседних элементов с индексами, меньшими k, уже расположены в нужном порядке. Дальнейшие проходы можно заканчивать на индексе k, вместо того чтобы двигаться до установленной заранее верхней границы i.

Качественно другое улучшение алгоритма можно получить из следующего наблюдения. Хотя "легкий" пузырек снизу поднимется наверх за один проход,

"тяжелые" пузырьки опускаются со минимальной скоростью: один шаг за итерацию. Так что массив 2 3 4 5 6 1 будет отсортирован за 1 проход, а сортировка последовательности 6 1 2 3 4 5 потребует 5 проходов.

Чтобы избежать подобного эффекта, можно менять направление следующих один за другим проходов. Получившийся алгоритм иногда называют "шейкер-сортировкой".

```
     #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;  //  границы  неотсортированной  части
массива
       T x;

       do{
           // проход снизу вверх
           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;

           // проход сверху вниз
           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];

        // до сортировки
        for(int i=0;i<SIZE;i++){
           ar[i]=rand()%100;
           cout<<ar[i]<<"\t";
```

```
        }
        cout<<"\n\n";
        shakerSort(ar,SIZE);

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

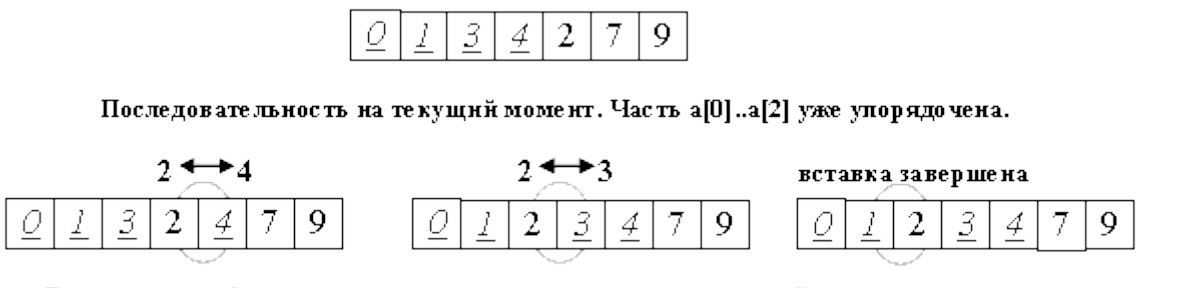
Таким образом, мы выяснили, что можем оптимизировать сортировку "пузырьком" на свой вкус. Дерзайте!!!

Сортировка вставками

Сортировка простыми вставками в чем-то похожа на методы изложенные в предыдущих разделах урока. Аналогичным образом делаются проходы по части массива, и аналогичным же образом в его начале "вырастает" отсортированная последовательность.

Однако в сортировке пузырьком или выбором можно было четко заявить, что на i-м шаге элементы a[0]...a[i] стоят на правильных местах и никуда более не переместятся. Здесь же подобное утверждение будет более слабым: последовательность a[0]...a[i] упорядочена. При этом по ходу алгоритма в нее будут вставляться все новые элементы.

Будем разбирать алгоритм, рассматривая его действия на i-м шаге. Как говорилось выше, последовательность к этому моменту разделена на две части: готовую a[0]...a[i] и неупорядоченную a[i+1]...a[n]. На следующем, (i+1)-м каждом шаге алгоритма берем a[i+1] и вставляем на нужное место в готовую часть массива. Поиск подходящего места для очередного элемента входной последовательности осуществляется путем последовательных сравнений с элементом, стоящим перед ним. В зависимости от результата сравнения элемент либо остается на текущем месте(вставка завершена), либо они меняются местами и процесс повторяется.



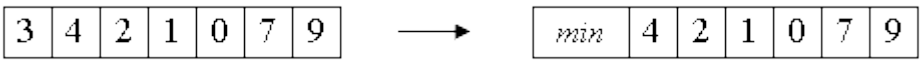
Таким образом, в процессе вставки мы "просеиваем" элемент x к началу массива, останавливаясь в случае, когда найден элемент, меньший x или достигнуто начало последовательности.

Реализация метода

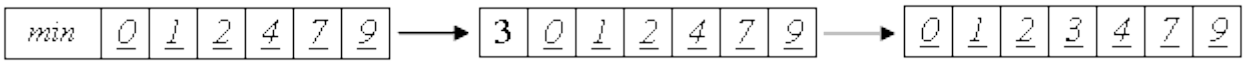
```
#include <iostream>

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

using namespace std;

template <class T>
void insertSort(T a[], long size) {
        T x;
        long i, j;
        // цикл проходов, i - номер прохода
        for(i=0;i<size;i++){
           x=a[i];

           // поиск места элемента в готовой последовательности
           for(j=i-1;j>=0&&a[j]>x;j--)
             // сдвигаем элемент направо, пока не дошли
             a[j+1]=a[j];
             // место найдено, вставить элемент
             a[j+1] = x;
        }
}

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

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

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

Принципы метода

Хорошим показателем сортировки является весьма естественное поведение: почти отсортированный массив будет досортирован очень быстро. Это, вкупе с устойчивостью алгоритма, делает метод хорошим выбором в соответствующих ситуациях. Однако, алгоритм можно слегка улучшить. Заметим, что на каждом шаге внутреннего цикла проверяются 2 условия. Можно объединить из в одно, поставив в начало массива специальный "сторожевой элемент". Он должен быть заведомо меньше всех остальных элементов массива.



Тогда при j=0 будет заведомо верно a[0] <=x. Цикл остановится на нулевом элементе, что и было целью условия j>=0. Таким образом, сортировка будет происходить правильным образом, а во внутреннем цикле станет на одно сравнение меньше. Однако, отсортированный массив будет не полон, так как из него исчезло первое число. Для окончания сортировки это число следует вернуть назад, а затем вставить в отсортированную последовательность 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;
        // сохранить старый первый элемент
        T backup = a[0];
        // заменить на минимальный
        setMin(a,size);

        // отсортировать массив
        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;
         }

         // вставить backup на правильное место
         for(j=1;j<size&&a[j]<backup;j++)
            a[j-1]=a[j];

         // вставка элемента
         a[j-1] = backup;
}

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

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

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

Домашнее задание

1. Дан массив чисел размерностью 10 элементов. Написать функцию, которая сортирует массив по возрастанию или по убыванию, в зависимости от третьего параметра функции. Если он равен 1, сортировка идет по убыванию, если 0, то по возрастанию. Первые 2 параметра функции - это массив и его размер, третий параметр по умолчанию равен 1.

2. Дан массив случайных чисел в диапазоне от -20 до +20. Необходимо найти позиции крайних отрицательных элементов (самого левого отрицательного элемента и самого правого отрицательного элемента) и отсортировать элементы, находящиеся между ними.

3. Дан массив из 20 целых чисел со значениями от 1 до 20.

Необходимо:

- написать функцию, разбрасывающую элементы массива произвольным образом;
- создать случайное число из того же диапазона и найти позицию этого случайного числа в массиве;
- отсортировать элементы массива, находящиеся слева от найденной позиции по убыванию, а элементы массива, находящиеся справа от найденной позиции по возрастанию.