Data Structures

# **Lecture No. 05**

### **Reading Material**

Data Structures and Algorithm Analysis in C++ Chapter. 7

7.1, 7.2

## ***Summary***

* Selection Sort
  + Selection Sort analysis
* Insertion Sort
  + Insertion Sort Analysis
* Summary

This is the sequel of the previous lecture in which we talked about the sort algorithms. There are three elementary sorting methods being discussed in this hand out. These are- selection sort, insertion sort and bubble sort. To begin with, we will talk about the selection sort algorithm.

**Selection Sort**

Suppose we have an array with different numbers. For sorting it in an ascending order, selection sorting will be applied on it in the following manner. We find the smallest number in the array and bring it to the position 1. We do the same process with the remaining part of the array and bring the smallest number among the remaining array to the next position. This process continues till the time all the positions of the array are filled with the elements. Thus the main idea of selection sort is that

* + find the smallest element
  + put it in the first position
  + find the next smallest element in the remaining elements
  + put it in the second position
  + …
  + And so on, until we get to the end of the array

Let’s understand this algorithm by considering an example with the help of figures.

Consider an array that has four numbers i.e. 19, 5, 7 and 12 respectively. Now we want to sort this array in ascending order. To sort the array, selection algorithm will be applied on it.

|  |
| --- |
|  |

The above pictorial representation explains the selection sort. It describes that at the start, we begin the search for the smallest number from the first position of the array i.e. from the index zero. We find that 5 is the smallest number and bring it to the first position i.e. index 0. Later, number 19 is put at the position that was occupied by number 5. Thus in a sense, we swap these numbers. Now 5, the smallest number in the array, has come at its final position i.e. index 0.

As index 0 has the proper number, so we start our search from position 2 i.e. index 1. Now we look at the remaining elements 19, 7, 12 and find the smallest number among them. Here 7 is the smallest so we change its position with 19 to bring 7 at its position. Thus 5 and 7 have come at their final positions. Now there are two elements are left behind i.e. 12 and 19. We search the smallest among these and find that 12 is the smallest. So we swap it with 19 to bring it at index 2 i.e. its final position. Now there is last element remaining and obviously it is at its position as there is no element to compare with it. The array is now in the sorted form with ascending numbers.

The point to remember in the selection search is that at the beginning, we start search for the smallest number from index 0 (i.e. first position). After it we start search from the index 1 (i.e. position 2). After each search one number gets its final position so we start the next search from the next position to it. Thus we do the multiple passes of the array to sort it. First, we pass through the n elements of the array and search the n-1 elements and then n-2. Thus at last, we come to the single and last element of the array.

Now let’s see the code of this algorithm in C++.

|  |
| --- |
| void selectionSort(int \*arr, int N)  {  int posmin, count, tmp ;  for (count=0;count<N;count++)  {  posmin = findIndexMin(arr, count, N) ;  tmp=arr[posmin] ;  arr[posmin]=arr[count] ;  arr[count]=tmp ;  }  }  int findIndexMin (int \*arr, int start, int N)  {  int posmin=start ;  int index ;  for(index=start; index < N; index++)  if (arr[index]<arr[posmin])  posmin=index ;  return posmin ;  } |

In the above code, we write the function *selectionSort*. This function takes an array of integers as *\*arr* and the size of array as *N*. There are local variables declared in function as *posmin, count* and *tmp*. Then there is a ‘*for* loop’ that starts from zero and goes to *N-1*. This is due to the fact that the index of array of N elements is from zero to N-1. The condition *count < N* indicates that loop will execute as long as *count* is less than *N* and will exit when count gets *N*. Now in the loop, we calculate the *posmin* with a function i.e. *findIndexMin*. This *findIndexMin* method is written below in the code. This routine or method works in the way that we pass to it the whole array i.e. *arr*, the value of *count* (what it is in that execution of loop) and size of the array i.e. *N*. This routine starts the search from the index equal to value of *count* and goes to Nth position, returning the position of the minimum (smallest) element. Now in the first routine, we get this position value in the variable *posmin*. Thus the code line:

posmin = findIndexMin(arr, count, N) ;

gets the position of the smallest number returned by the method findIndexMin in the variable *posmin*. After finding this position, we do the swapping of this *posmin* with the *count* position. Thus we put the value of position *posmin* in the *count* position.

The *findIndexMin* routine is such that it takes *arr, start* and *N* as arguments. It starts searching from the *start* position in the *for* loop, finds the position of the minimum value and returns that position.

This sorting algorithm is also known as the in-place sorting algorithm as there is no need of additional storage to carry out this sorting. The pictorial representation of the swap action of this algorithm will be as under:

We want to sort the following array that has five elements. We start the search from the first element and find that the smallest element is 5. As we have started our search from index 0 (that is the *count* in our above code) so we swap 5 with the value at index 0 i.e. 20. After this swap, 5comes at the position of 20 while 20 goes to the position of 5. Afterwards, we start the search from index 1. We find that 7 is the smallest number among the remaining elements. It swaps its position with 8. After this, in the remaining three elements, 8 is the smallest. This number 8 swaps its position with 20. At the end, 10 is the smallest number among 10 and 20. So there is no need of swapping as 10 has already got its position.

|  |
| --- |
|  |

**Selection Sort Analysis**

We have seen in the code for the algorithm that there are two loops. The loop in the *selectionSort* method passes the array to search the smallest element and the second loop that is in the findIndexMin method finds the position (index) of the smallest value. To find the first smallest element, we have to go through the N elements of the array. For the purpose of finding the second smallest element, we have to search the N-1 elements. During this search process, we have to find two elements for the second last smallest element. And obviously in the end, there is one element that is at its proper position, necessitating no search. We have to do all these searches. These are N, N-1, N-2 ……2, 1 for the first, second, third ……second last and last element respectively. Now if we want to find the total searches, the addition of all these searches together will help us get a sum total as given in the following equation.

Total searches = 1 + 2 + 3 + …….+ (N-2) + (N-1) + N

= N (N+1) / 2

= (N2 + N) / 2

Suppose if N is 10, then according to this formula, the total searches will be (100 + 10) / 2 i.e. 55. If N is 100, the total searches will be (10000 + 100) / 2 i.e. 5050. Similarly if N is 1 million, N2 is going to be very large as compared to N. Thus there we can ignore N and can say that the time (total searches) is proportional to N2. This means that larger the N, greater will be the performance of selection with respect to N2.

**Insertion Sort**

The main idea of insertion sort is

* + Start by considering the first two elements of the array data. If found out of order, swap them
  + Consider the third element; insert it into the proper position among the first three elements.
  + Consider the fourth element; insert it into the proper position among the first four elements.
  + … …

This algorithm is not something uncommon to the persons who know card playing. In the game of cards, a player gets 13 cards. He keeps them in the sorted order in his hand for his ease. A player looks at the first two cards, sorts them and keeps the smaller card first and then the second. Suppose that two cards were 9 and 8, the player swap them and keep 8 before 9. Now he takes the third card. Suppose, it is 10, then it is in its position. If this card is of number 2, the player will pick it up and put it on the start of the cards. Then he looks at the fourth card and inserts it in the first three cards (that he has sorted) at a proper place. He repeats the same process with all the cards and finally gets the cards in a sorted order. Thus in this algorithm, we keep the left part of the array sorted and take element from the right and insert it in the left part at its proper place. Due to this process of insertion, it is called insertion sorting.

Let’s consider the array that we have used in the selection sort and sort it now with the insertion sorting. The following figure shows the insertion sort of the array pictorially.

|  |
| --- |
|  |

The array consists of the elements 19, 12, 5 and 7. We take the first two numbers i.e. 19 and 12. As we see 12 is less than 19, so we swap their positions. Thus 12 comes at index 0 and 19 goes to index 1. Now we pick the third number i.e. 5. We have to find the position of this number by comparing it with the two already sorted numbers. These numbers are 12 and 19. We see that 5 is smaller than these two. So it should come before these two numbers. Thus the proper position of 5 is index 0. To insert it at index 0, we shift the numbers 12 and 19 before inserting 5 at index 0. Thus 5 has come at its position. Now we pick the number 7 and find its position between 5 and 12. To insert 7 after 5 and before 12, we have to shift the numbers 12 and 19 to the right. After this shifting, we put number 7 at its position. Now the whole array has been sorted so the process stops here.

Following is the code of the insertion sort in C++.

|  |
| --- |
| void insertionSort(int \*arr, int N)  {  int pos, count, val;  for(count=1; count < N; count++)  {  val = arr[count];  for(pos=count-1;pos >= 0; pos--){  if (arr[pos] > val){  arr[pos+1]=arr[pos];}  else { break;}  }  arr[pos+1] = val;  }  } |
|  |

In this sorting function, we start the process from index 0 and 1 and swap them. Afterwards, we go to the third position and put it into its proper position. While inserting a number in the sorted portion of the array, we have to shift the elements. This shifting is an additional overhead of this sorting algorithm. Due to this shifting, the sorting requires more time. This algorithm is also an in place algorithm as we don’t need any additional storage. At the most, we need some variables of local nature, normally used in programs.

From the above code, we see that the name of this sorting routine is *insertionSort*. It takes an array and its size as arguments. There are local variables *pos*, *count* and *val* declared in this function. After this there is a *for* loop that starts from the *count* having value equal to 1. Now we put the value of *count* index (i.e. arr[count]) in variable *val*. This value has to find its place in the left sorted portion of the array. To find that position we have to execute one more *for* loop. This loop starts from *count-1* and goes to down to zero. In the body of the loop we compare the value of *val* with the value at current position in the array i.e. arr[pos]. If *val* is less than the *arr[pos]* i.e. value at current index. It means that the *val* has to go to the left position of *arr[pos].So* we shift the *arr[pos]* to right to create place for the new value i.e. *val*. When the loop exits, we put this value *val* at arr[pos + 1]. Thus we have inserted the number in the array at its proper position.

Following is the step by step explanation for the insertion sort of the above example with same previous array.

First of all we take the first two elements 19 and 12. As 12 is less than 19, we do right shift on 19. And put 12 at its position i.e. index 0. Afterwards, we go to index 2. There is 5 at this position. As we see that 5 is less than the other elements on the left side of array, it has to come at the first position. To bring 5 to first position, the number 12 and 19 has to be shifted to right. After this shifting, the position of index 0 becomes empty and we put 5 there. Finally, there is number 7. The position of 7 will be between 5 and 12. For this, we have to shift 12 and 19 towards right so that the place for 7 could be created. We put 7 at that place. Thus the array is sorted now.

**Insertion Sort Analysis**

Let’s analyze that when the value of N increases. How much time for insertion sort increases? In the code of insertion sort, we have seen that there are outer and inner loops. Due to these two loops, we can understand that it is also like N2 algorithm. In the sort process, there may be a situation that every iteration inserts an element at the start of the array by shifting all sorted elements along. Now if we have to bring the second element to its position, there will be need of shifting the first element. This means that we have to shift one element. Similarly, for placing the third element at the start position (we are discussing the worst case scenario in which at every iteration the element has to go to the first position), we have to shift two elements. Thus we sum up all the shiftings, the total becomes 2 + 3 + 4 +……. + N-1 + N.

The summation can be written as follows.

Total = (2 + N ) (N -1) / 2

= O (N2)

From this expression, we see that when the value of N increases, the value of N2 will dominate. It will increase significantly with respect to N. Thus we see that insertion sort is also an N2 algorithm like selection sort.