Name: Vishal Shashikant Salvi.

Class: SE Comps

Batch: C

UID: 2019230069

Aim:

Experiments for time complexity analysis Insertion Sort and Selection Sort.

Theory:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

Insertion Sort

Algorithm

INSERTION SORT	Cost	Time
for i←2 to n do	C1	n
key← a[i]	C2	n-1
j←i-1	C3	n-1
while(j>0 && a[j]>key)	C4	$\sum_{i=2}^{n} tj$
a[j+1]←a[j]	C5	$\sum_{i=2}^{n} (tj-1)$

j←j-1	C6	$\sum_{i=2}^{n}(tj-1)$
end	-	
a[j+1]←key	C7	n-1
end	-	

$T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} tj + c_5 \sum_{i=2}^{n}(tj-1) + c_6 \sum_{i=2}^{n}(tj-1) + c_7(n-1)$

Best Case

$T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$

$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$

$= O(n)$

Worst Case / Average Case

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

$= O(n^2)$

Complexity of Insertion Sort. **Insertion sort** runs in O (n) O(n) O(n) **time** in its best case and runs in O (n 2) O(n^2) O(n2) in its worst and average cases.

Analysis of insertion sort

The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers. Moreover, INSERTION-SORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are. In general, the time taken by an algorithm grows

with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input. To do so, we need to define the terms "running time" and "size of input" more carefully.

The best notion for input size depends on the problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the number of items in the input-for example, the array size n for sorting. For many other problems, such as multiplying two integers, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation. Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph. We shall indicate which input size measure is being used with each problem we study.

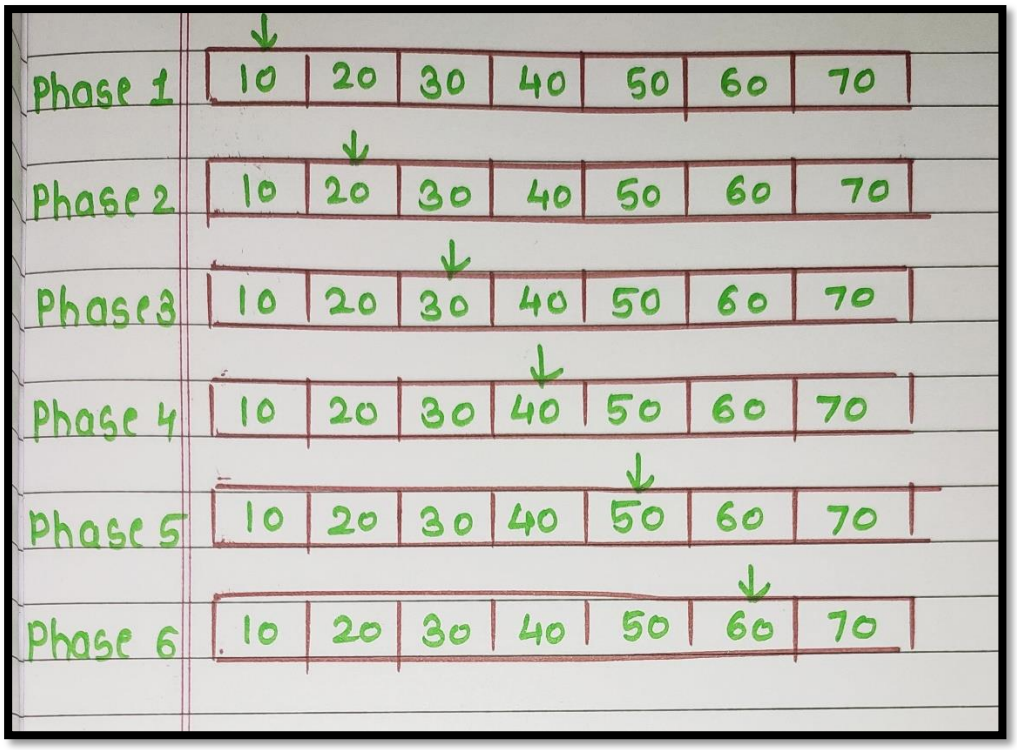
The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed. It is convenient to define the notion of step so that it is as machine independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the ith line takes time ci , where ci is a constant. This viewpoint is in keeping with the RAM model, and it also reflects how the pseudocode would be implemented on most actual computers.[4]

In the following discussion, our expression for the running time of INSERTION-SORT will evolve from a messy formula that uses all the statement costs ci to a much simpler notation that is more concise and more easily manipulated. This simpler notation will also make it easy to determine whether one algorithm is more efficient than another. We start

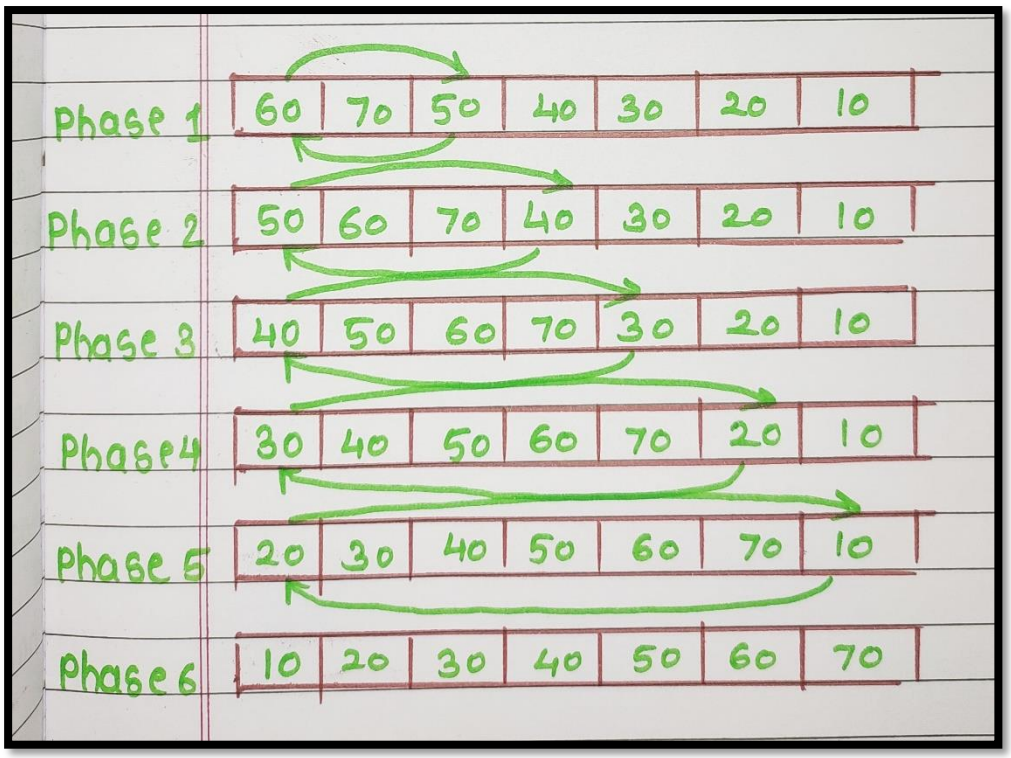
by presenting the INSERTION-SORT procedure with the time "cost" of each statement and the number of times each statement is executed. For each j = 2, 3, . . . , n, where n = length[A], we let tj be the number of times the while loop test in line 5 is executed for that value of j.

When a for or while loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body. We assume that comments are not executable statements, and so they take no time.

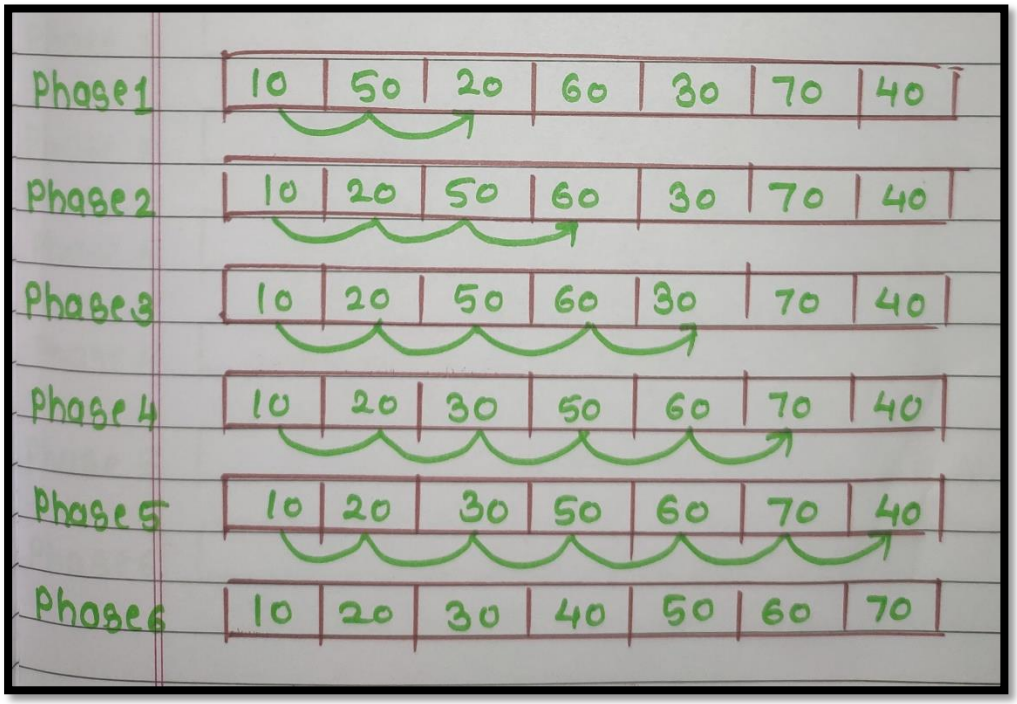
Best Case:



Worst Case:



Average Case:



Insertion Sort:

```
#include<stdio.h>

int array[20];
int size,count1=0,count2=0;

void printArr()
{
   printf("\nNew Phase:\n");
   for(int n=0;n<size;n++)
      {
         printf("%d ",array[n]);
      }
}

void insertionSort()
{
   int i,j,k,count1=0,count2=0;
      for(i=0;i<size-1;i++)
      {
         j=i+1;
         k=0;
         count1++;
      while(k<=i)
      {
         if(array[j]>= array[k])
         {
            k++;
         }
      else
```

```
        {
            int element=array[j];
            count1++;
        for(int t=i;t>=k;t--)
            {
                array[t+1]=array[t];
                count2++;
            }
        array[k]=element;
        break;
        }
    }
    printArr();
  }
  printf("\nNo. of Comparisons: %d",count1);
  printf("\nNo. of Shifting: %d",count2);
}

void main()
  {
    int count1=0,count2=0;
    printf("Enter the size of the array you want to perform insertion sort:\n");
    scanf("%d",&size);
    for(int n=0;n<size;n++)
    {
        printf("Enter the %d element in array:\n",n+1);
        scanf("%d",&array[n]);
    }
    insertionSort();
}
```

Output:

Best Case:

```
Enter the size of the array you want to perform insertion sort:
12
Enter the 1 element in array:
10
Enter the 2 element in array:
20
Enter the 3 element in array:
30
Enter the 4 element in array:
40
Enter the 5 element in array:
50
Enter the 6 element in array:
60
Enter the 7 element in array:
70
Enter the 8 element in array:
80
Enter the 9 element in array:
90
Enter the 10 element in array:
100
Enter the 11 element in array:
110
Enter the 12 element in array:
120
```

```
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
No. of Comparisons: 11
No. of Shifting: 0

...Program finished with exit code 19
Press ENTER to exit console.
```

Worst case:

```
Enter the size of the array you want to perform insertion sort:
12
Enter the 1 element in array:
120
Enter the 2 element in array:
110
Enter the 3 element in array:
100
Enter the 4 element in array:
90
Enter the 5 element in array:
80
Enter the 6 element in array:
70
Enter the 7 element in array:
60
Enter the 8 element in array:
50
Enter the 9 element in array:
40
Enter the 10 element in array:
30
Enter the 11 element in array:
20
Enter the 12 element in array:
10
```

```
New Phase:
110 120 100 90 80 70 60 50 40 30 20 10
New Phase:
100 110 120 90 80 70 60 50 40 30 20 10
New Phase:
90 100 110 120 80 70 60 50 40 30 20 10
New Phase:
80 90 100 110 120 70 60 50 40 30 20 10
New Phase:
70 80 90 100 110 120 60 50 40 30 20 10
New Phase:
60 70 80 90 100 110 120 50 40 30 20 10
New Phase:
50 60 70 80 90 100 110 120 40 30 20 10
New Phase:
40 50 60 70 80 90 100 110 120 30 20 10
New Phase:
30 40 50 60 70 80 90 100 110 120 20 10
New Phase:
20 30 40 50 60 70 80 90 100 110 120 10
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
No. of Comparisons: 22
No. of Shifting: 66

...Program finished with exit code 20
Press ENTER to exit console.
```

Average case:

```
Enter the size of the array you want to perform insertion sort:
12
Enter the 1 element in array:
10
Enter the 2 element in array:
70
Enter the 3 element in array:
20
Enter the 4 element in array:
80
Enter the 5 element in array:
30
Enter the 6 element in array:
90
Enter the 7 element in array:
40
Enter the 8 element in array:
100
Enter the 9 element in array:
50
Enter the 10 element in array:
110
Enter the 11 element in array:
60
Enter the 12 element in array:
120
```

```
New Phase:
10 70 20 80 30 90 40 100 50 110 60 120
New Phase:
10 20 70 80 30 90 40 100 50 110 60 120
New Phase:
10 20 70 80 30 90 40 100 50 110 60 120
New Phase:
10 20 30 70 80 90 40 100 50 110 60 120
New Phase:
10 20 30 70 80 90 40 100 50 110 60 120
New Phase:
10 20 30 40 70 80 90 100 50 110 60 120
New Phase:
10 20 30 40 70 80 90 100 50 110 60 120
New Phase:
10 20 30 40 50 70 80 90 100 110 60 120
New Phase:
10 20 30 40 50 70 80 90 100 110 60 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
No. of Comparisons: 16
No. of Shifting: 15

...Program finished with exit code 20
Press ENTER to exit console.
```

Selection Sort:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

Algorithm

SELECTION SORT	Cost	Time
for i←1 to n-1 do	C1	n
min←i	C2	n-1
for j←i+1 to n do	C3	$\sum_{i=1}^{n-1}(n-i+1)$
if(A[j] < A[min]) do	C4	$\sum_{i=1}^{n-1}(n-i)$
min←j	C5	$\sum_{i=1}^{n-1}(n-i)$
End	-	
End	-	
swap(a[i],a[min])	C6	n-1
End	-	

Best Case / Worst Case / Average Case

$T(n) = c_1 n + c_2(n-1) + c_3 \sum_{i=1}^{n-1}(n-i+1) + c_4 \sum_{i=1}^{n-1}(n-i) + c_5 \sum_{i=1}^{n-1}(n-i) + c_6(n-1)$

$= O(n^2)$

In computer science, **selection sort** is a **sorting algorithm**, specifically an in-place comparison **sort**. It has $O(n^2)$ **time complexity**, making it inefficient on large lists, and generally performs worse than the similar insertion **sort**.

Analysis of selection sort:

Selection sort loops over indices in the array; for each index, selection sort calls indexOfMinimum and swap. If the length of the array is n*n*n, there are n*n*n indices in the array.

Since each execution of the body of the loop runs two lines of code, you might think that 2 n2*n*2, n lines of code are executed by selection sort. But it's not true! Remember that indexOfMinimum and swap are functions: when either is called, some lines of code are executed.

How many lines of code are executed by a single call to swap? In the usual implementation, it's three lines, so that each call to swap takes constant time.

How many lines of code are executed by a single call to indexOfMinimum? We have to account for the loop inside indexOfMinimum. How many times does this loop execute in a given call to indexOfMinimum? It depends on the size of the subarray that it's iterating over. If the subarray is the whole array (as it is on the first step), the loop body runs n*n*n times. If the subarray is of size 6, then the loop body runs 6 times.

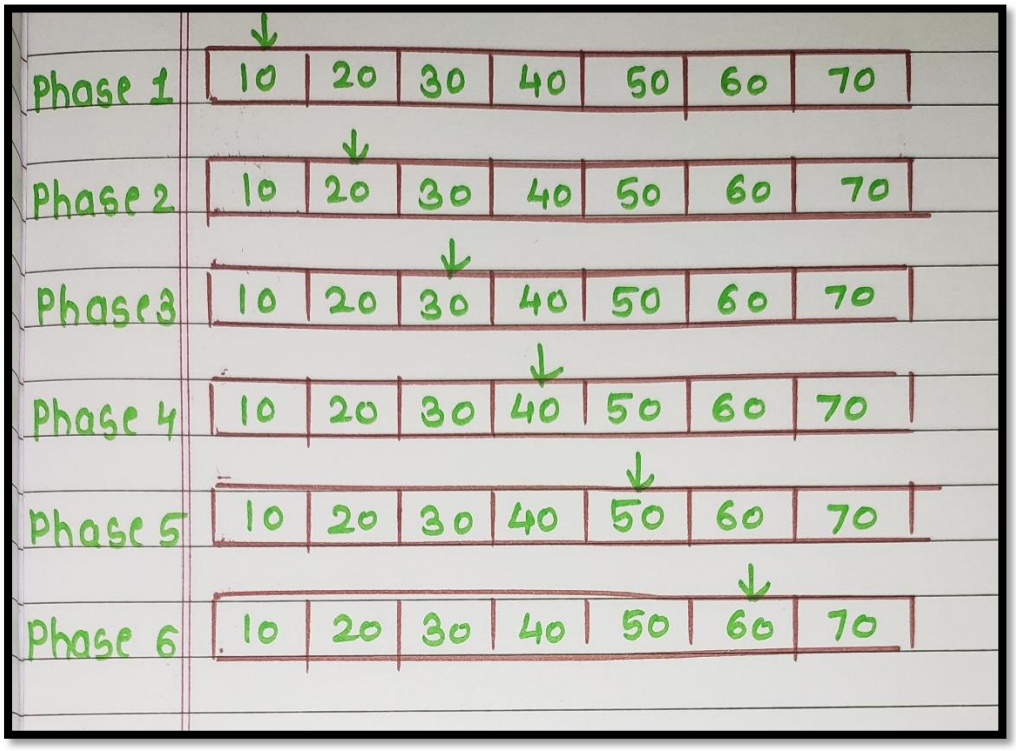
For example, let's say the whole array is of size 8 and think about how selection sort works.

1. In the first call of indexOfMinimum, it has to look at every value in the array, and so the loop body in indexOfMinimum runs 8 times.

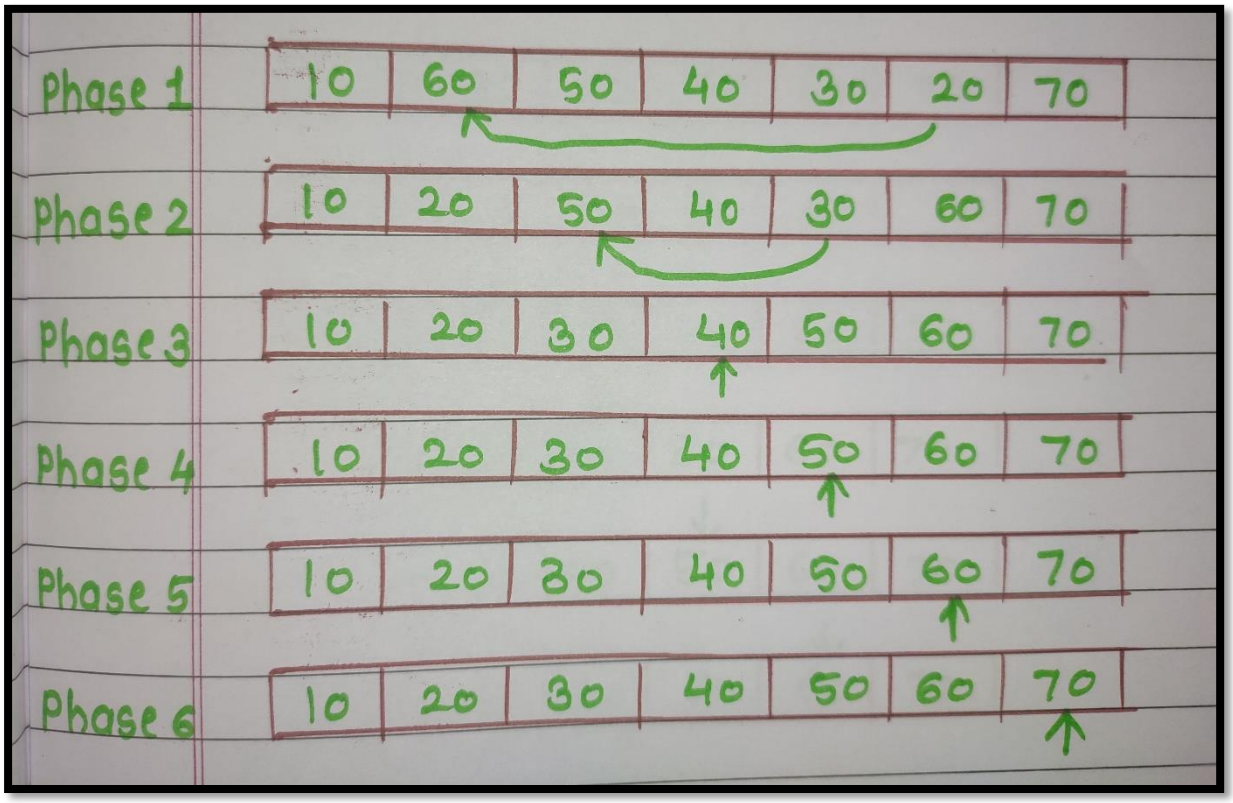
2. In the second call of indexOfMinimum, it has to look at every value in the subarray from indices 1 to 7, and so the loop body in indexOfMinimum runs 7 times.
3. In the third call, it looks at the subarray from indices 2 to 7; the loop body runs 6 times.
4. In the fourth call, it looks at the subarray from indices 3 to 7; the loop body runs 5 times.
5. …
6. In the eighth and final call of indexOfMinimum, the loop body runs just 1 time.

   If we total up the number of times the loop body of indexOfMinimum runs, we get 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 36 times.

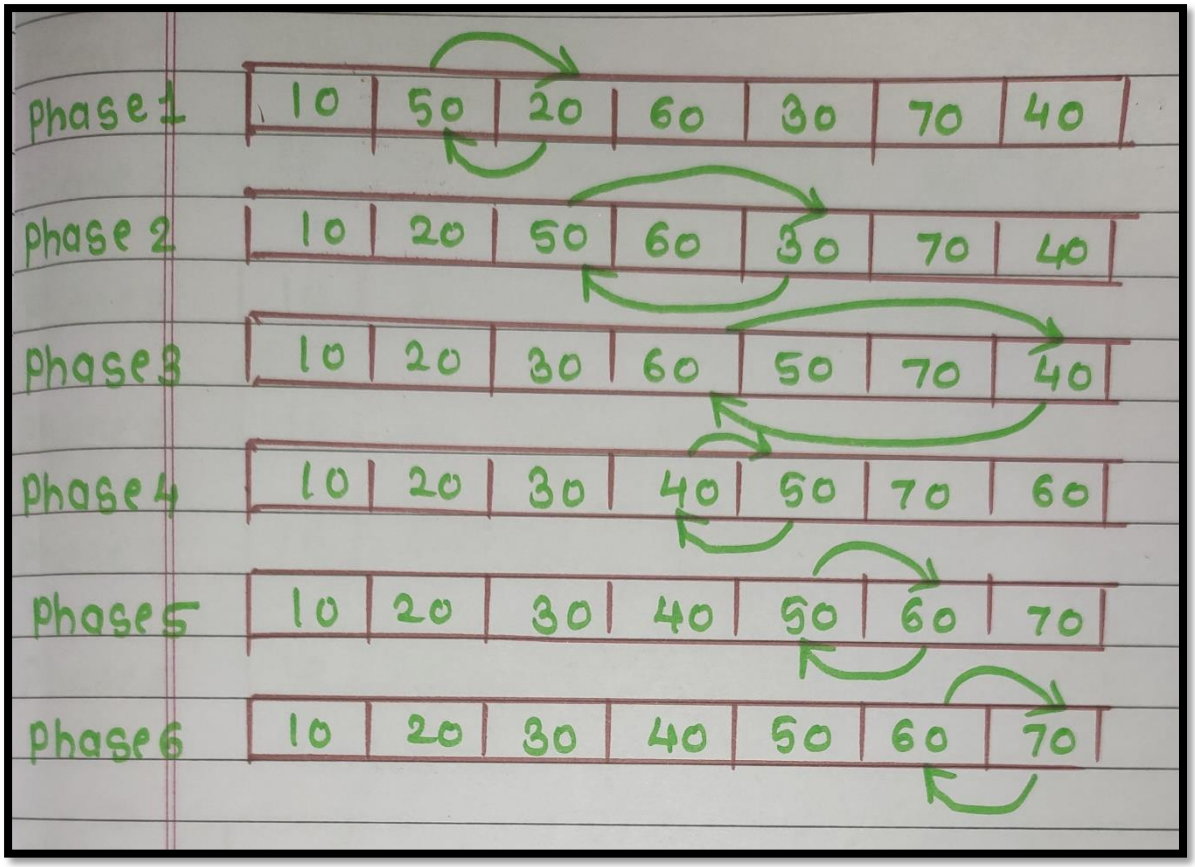
   Best Case:



Worst Case:



Average Case:



Selection Sort:

```
#include<stdio.h>

int array[20];
int size;
void printArr()
{
   printf("\nNew Phase:\n");
   for(int n=0;n<size;n++)
      {
         printf("%d ",array[n]);
      }
}

void selection_sort()
{
   int min,min_index,count1=0,count2=0;
   for(int i=0;i<size-1;i++)
      {
         min_index=i;
         min=array[i];
      for(int j=i;j<size;j++)
      {
         count1++;
         if(array[j]<=min)
         {
         min=array[j];
         min_index=j;
      }
   }
```

```
    int temp;
    temp=array[i];
    array[i]=min;
    array[min_index]=temp;
    count2++;
    printArr();
}



    printf("\nNo. of Comparisons: %d",count1);
    printf("\nNo. of Shifting: %d",count2);
}

void main()
{

    printf("Enter the size of the array you want to perform Selection sort:\n");
    scanf("%d",&size);
    for(int n=0;n<size;n++)
        {
            printf("Enter the %d element in array:\n",n+1);
            scanf("%d",&array[n]);
        }
selection_sort();
}
```

Output:

Best case:

```
Enter the size of the array you want to perform Selection sort:
12
Enter the 1 element in array:
10
Enter the 2 element in array:
20
Enter the 3 element in array:
30
Enter the 4 element in array:
40
Enter the 5 element in array:
50
Enter the 6 element in array:
60
Enter the 7 element in array:
70
Enter the 8 element in array:
80
Enter the 9 element in array:
90
Enter the 10 element in array:
100
Enter the 11 element in array:
110
Enter the 12 element in array:
120
```

```
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
No. of Comparisons: 77
No. of Shifting: 11

...Program finished with exit code 20
Press ENTER to exit console.
```

Worst case:

```
Enter the size of the array you want to perform Selection sort:
12
Enter the 1 element in array:
120
Enter the 2 element in array:
110
Enter the 3 element in array:
100
Enter the 4 element in array:
90
Enter the 5 element in array:
80
Enter the 6 element in array:
70
Enter the 7 element in array:
60
Enter the 8 element in array:
50
Enter the 9 element in array:
40
Enter the 10 element in array:
30
Enter the 11 element in array:
20
Enter the 12 element in array:
10
```

```
New Phase:
10 110 100 90 80 70 60 50 40 30 20 120
New Phase:
10 20 100 90 80 70 60 50 40 30 110 120
New Phase:
10 20 30 90 80 70 60 50 40 100 110 120
New Phase:
10 20 30 40 80 70 60 50 90 100 110 120
New Phase:
10 20 30 40 50 70 60 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
No. of Comparisons: 77
No. of Shifting: 11

...Program finished with exit code 20
Press ENTER to exit console.
```

Average case:

```
Enter the size of the array you want to perform Selection sort:
12
Enter the 1 element in array:
10
Enter the 2 element in array:
70
Enter the 3 element in array:
20
Enter the 4 element in array:
80
Enter the 5 element in array:
30
Enter the 6 element in array:
90
Enter the 7 element in array:
40
Enter the 8 element in array:
100
Enter the 9 element in array:
50
Enter the 10 element in array:
110
Enter the 11 element in array:
60
Enter the 12 element in array:
120
```

```
New Phase:
10 70 20 80 30 90 40 100 50 110 60 120
New Phase:
10 20 70 80 30 90 40 100 50 110 60 120
New Phase:
10 20 30 80 70 90 40 100 50 110 60 120
New Phase:
10 20 30 40 70 90 80 100 50 110 60 120
New Phase:
10 20 30 40 50 90 80 100 70 110 60 120
New Phase:
10 20 30 40 50 60 80 100 70 110 90 120
New Phase:
10 20 30 40 50 60 70 100 80 110 90 120
New Phase:
10 20 30 40 50 60 70 80 100 110 90 120
New Phase:
10 20 30 40 50 60 70 80 90 110 100 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
No. of Comparisons: 77
No. of Shifting: 11

...Program finished with exit code 20
Press ENTER to exit console.
```

Conclusion:

Thus, we studied algorithm of selection and insertion sort. We also came to know that the running time of selection sort in best, average and worst case are the same. On the other hand, performance of insertion sort is better during the best case as minimal comparisons are required.