DAA Experiment-1-B

(Batch-A/A1)

Name	Ansari Mohammed Shanouf Valijan
UID Number	2021300004
Class	SY B.Tech Computer Engineering(Div-A)
Experiment Number	1-B
Date of Performance	02-02-23
Date of Submission	08-02-23

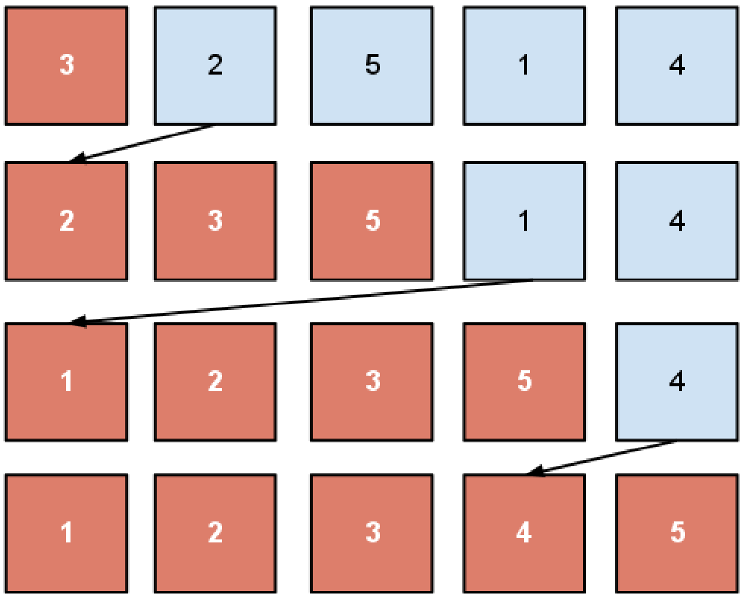
Aim:
To find the running time of an algorithm.

Problem Definition and Assumptions:
For this experiment, you need to implement two sorting algorithms namely Insertion and Selection sort methods. Compare these algorithms based on time and space complexity. Time required by sorting algorithms can be calculated using high_resolution_clock::now() under namespace std::chrono. You have to generate 1,00,000 integer numbers using C/C++ Rand function and save them in a text file. Both the sorting algorithms uses these 1,00,000 integer numbers as input as follows. Each sorting algorithm sorts a block of 100 integers numbers with array index numbers A[0..99], A[0..199], A[0..299],..., A[0..99999]. You need to use high_resolution_clock::now() function to find the time required for 100, 200, 300.... 100000 integer numbers. Finally, compare two algorithms namely Insertion and Selection by plotting the time required to sort 100000 integers using LibreOffice Calc/MS Excel. The x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot represents the tunning time to sort 1000 blocks of 100,200,300,...,100000 integer numbers. Note – You have to use C/C++ file processing functions for reading and writing randomly generated 100000 integer numbers.

Theory:
Insertion and selection sort are two of the important sorting algorithms that are helpful in many cases. Below, these sorting algorithms are explained in brief accompanied by their conceptual view.

Insertion Sort-

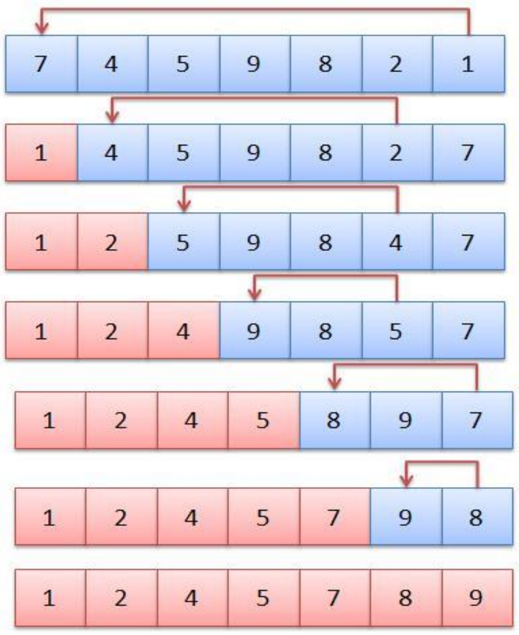
Insertion sort works similar to the sorting of playing cards. It is assumed that the first element in the list is already sorted. Further, as the array is traversed, the elements are moved to their correct locations with respect to the first element. Below is a conceptual view of insertion sort that explains its working-



In the above image, we have considered the sorting of 5 integers. Here, we assume that 3 is already sorted. Further, we move to 2, which is less than 3; hence it is placed before 3. Then we move to 5, which at present is at the proper location. Next, we have 1, which is moved to the first location as it is the smallest of all the previous numbers, which are all shifted one index position ahead. Then, the last element 4 is placed at its correct position. In this way, insertion sort can be implemented. One thing to note is that insertion sort requires adjacent swaps or shifting of elements rather than distant swaps.

Selection Sort-

In this algorithm, we continuous try to find the minimum element of the unsorted array. Once found, it is placed in the sorted section of the array. This algorithm requires distant swapping as the minimum element could be present at any location. Selection sort makes a lot more comparisons than insertion sort and hence, can be comparatively slow. This is demonstrated in the implementation part of this experiment.



The above image shows the working of selection sort. As can be seen, in each step, the minimum element from the unsorted section of the array is swapped with the first element in the unsorted array. The element, then becomes the last element of the sorted array. This process is repeated till the entire array is sorted.

Algorithms:

[A] For insertion sort-

I. Start iterating the concerned array from the second element.
II. Compare the current element with the previous element.
III. If the two elements are found in proper order, move to the next element.
IV. If the two elements are not in proper order, shift the previous elements till the proper location of the current element is found.
V. Repeat steps II to IV till the end of the array.

[B] For selection sort-

I. Start iterating the array from start to end and find the index value of the minimum element.
II. Swap it with the first element of the unsorted array.
III. Repeat till there are no elements left in the unsorted section of the array.

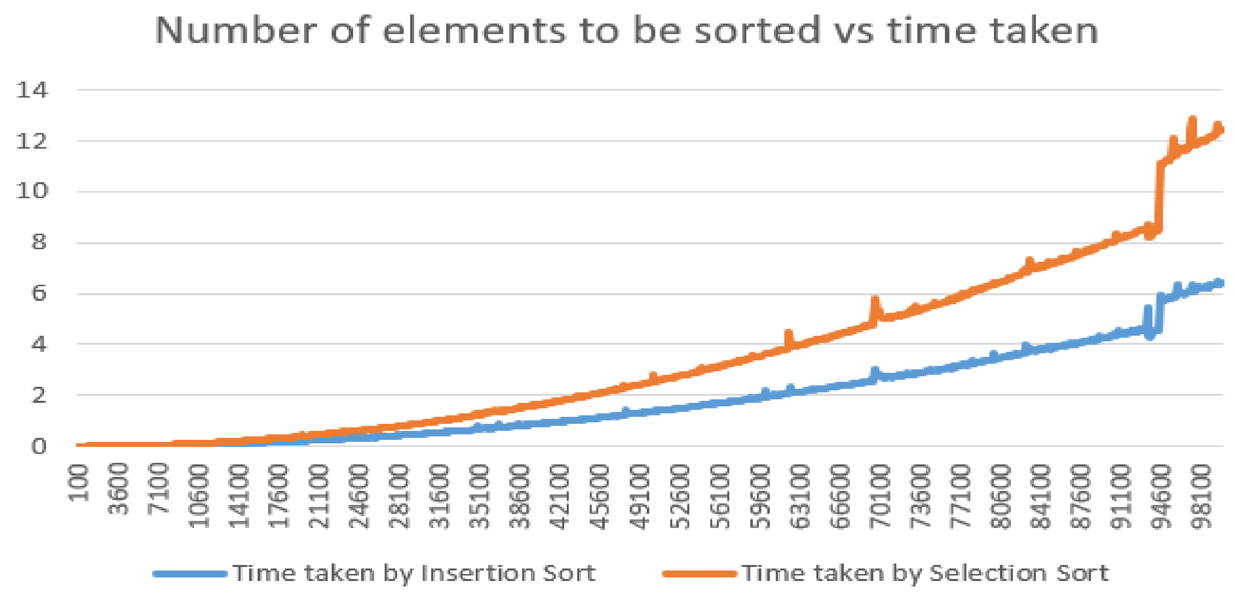
Program:

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

```c
//file for handling the input and output
FILE* data=NULL;
FILE* output=NULL;

//functions for sorting
void swap(int arr[], int ix, int iy){
    int temp=arr[ix];
    arr[ix]=arr[iy];
    arr[iy]=temp;
}

void insertionSort(int arr[], int size)
{
    int currentElem,j;
    for(int i=1; i<size; i++){
        currentElem=arr[i];
        j=i-1;
        while(j>=0 && arr[j]>currentElem){
            arr[j+1]=arr[j];
            j--;
        }
        arr[j+1]=currentElem;
    }
}

void selectionSort(int arr[], int size)
{
    int minIndex;
    for (int i=0; i<size-1; i++){
        minIndex=i;
        for (int j=i+1; j<size; j++)
            if(arr[j]<arr[minIndex])
                minIndex=j;
        if(minIndex!=i)
            swap(arr,minIndex,i);
    }
}

//for bringing the values into the array(for sorting)
void arrayLoading(int arr[],int endpoint){
    data=fopen("data.txt","r");
    for(int i=0; i<endpoint; i++){
        fscanf(data,"%d",&arr[i]);
    }
    fclose(data);
}

void main(){
    srand(time(NULL));
    clock_t t;
    int tempArr[100000];

    //intiating the data file
    data=fopen("data.txt","w");
    for(int i=0; i<100000; i++){
        fprintf(data,"%d\n",rand());
    }
    fclose(data);

    //sorting and printing out the time taken
    output=fopen("output.txt","w");
    double timeI, timeS;
    for(int i=100; i<=100000; i+=100){
        arrayLoading(tempArr,i);
        t=clock();
        insertionSort(tempArr,i);
        t=clock()-t;
        timeI=((double)t)/CLOCKS_PER_SEC;
        arrayLoading(tempArr,i);
        t=clock();
        selectionSort(tempArr,i);
        t=clock()-t;
        timeS=((double)t)/CLOCKS_PER_SEC;
        fprintf(output,"%d\t\t%lf\t\t%lf\n",i,timeI,timeS);
        printf("Block count -----> %d\n",i);
    }
    fclose(output);
}
```

Implementation:

For determining the running time of an algorithm, the ‘time.h’ header file was used. Firstly, 100000 random integers were generated and stored in a file named ‘data.txt’. These integers were brought into the array for sorting. The information regarding the number of elements taking part in sorting process, time taken by insertion and selection sort algorithms to complete the task was recorded in a separate file named ‘output.txt’. Further, the output obtained was brought into MS Excel for the purpose of graph plotting. Below is the graph that was obtained-



In the above graph, the time taken is shown in seconds. The x-axis consists of the number of elements/integers that took part in sorting process.

From the above graph, it is very clear that insertion sort takes less time to sort the same configuration of elements as sorted by selection sort. The difference in the time taken by these two algorithms to complete the task is negligible initially, when the number of elements is less; however, the difference is significant as the number of elements increases to about 35000 to 40000.

Conclusion:
By performing this experiment, I was able to understand the process of determining the time an algorithm implementation takes to perform the required task. Using this, I was able to compare insertion and selection sort algorithms which led me to the conclusion that insertion sort is generally a faster sorting algorithm than selection sort.