Sardar Patel Institute of Technology

Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058, India
(Autonomous College Affiliated to University of Mumbai)

Experiment No.	**1b**
Name	**Varada Khadake**
UID No.	**2021300059**
Class & Division	**COMPS A BATCH D**

Aim: Experiment on finding the running time of an algorithm.

Algorithm:

```
1.Start
2.initialize array a with size=100000
3for i=0 to i<1000
4.call function getData(i+1,a)
5.initialize start and end values of clock() function
6.call function insertionSort(a,(i+1)*100)
7.print array
8. call function getData(i+1,a)
9.initialize start and end values of clock() function
10.call function selectionSort(a,(i+1)*100)
11.print array
12.end for
```

```
insertionSort() :
n=length(A)
for i=1 to n-1 do
  j=i
while j>0 and A[j-1]>A[j] do
  swap (A[j],A[j-1])
j=j-1
end while
end for
```

selectionSort():

```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for

end procedure
```

Observation/Theory:

Insertion sort:

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

Insertion sort has various advantages such as –

- Simple implementation
- Efficient for small data sets

- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Selection sort:
In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right

Selection sort is generally used when -

- A small array is to be sorted
- Swapping cost doesn't matter
- It is compulsory to check all elements

As observed from the graphs , the run time goes on increasing for selection sort algorithm as the number of elements in the array increases. While, for insertion Sort the run time increases in the median and again decreases. Mostly, for insertion sort the run time is below 10.

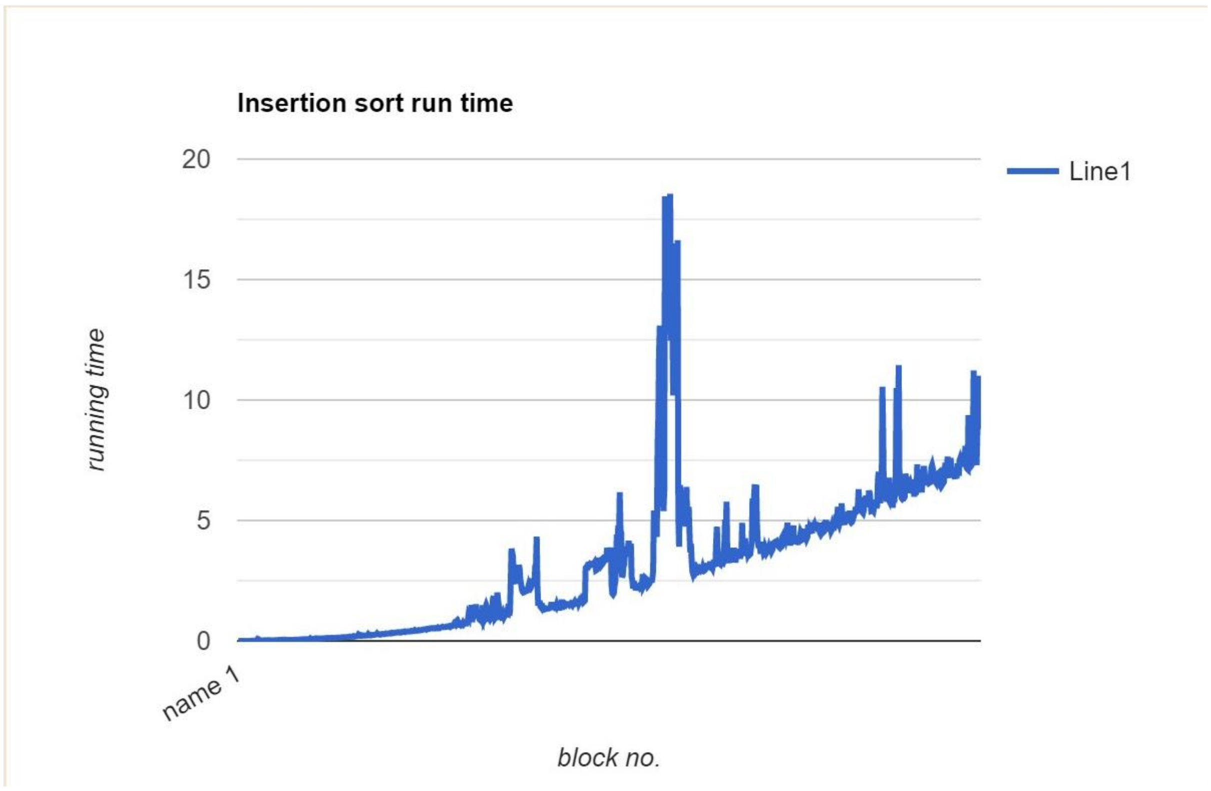
Time complexity:
Insertion sort: Best case-O(n)
Worst case-O(n^2)

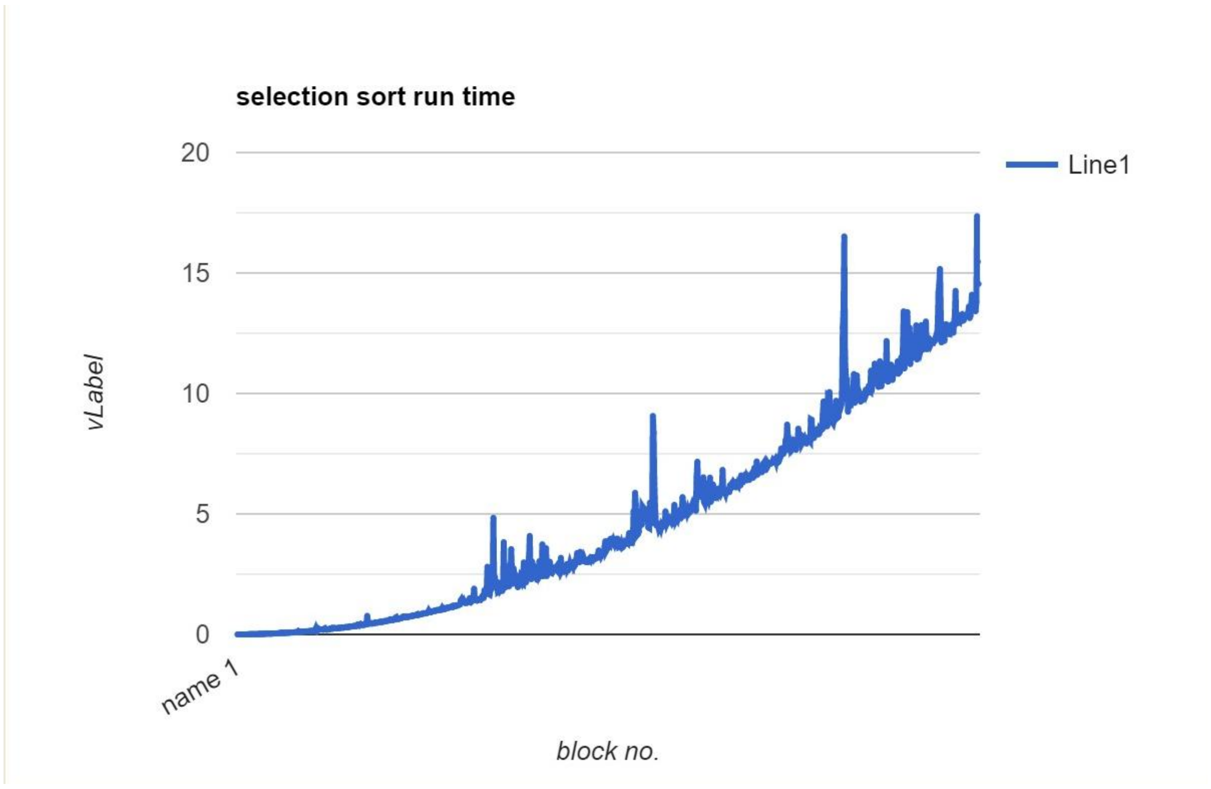
Selection sort: Best and worst case- O(n^2)

Code:

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

int main()
{
    void getData(int x, int arr[]), insertionSort(int arr[], int n), printArray(int arr[],
int n), selectionSort(int arr[], int n);
    clock_t start, end;
    double time_req;
    int a[100000];
    for (int i = 0; i < 1000; i++)
    {
        printf("insertion sort:");

        getData(i + 1, a);
        start = clock();
        insertionSort(a, (i + 1) * 100);
        end = clock();
        printf("\nset %d", i + 1);
        printArray(a, (i + 1) * 100);
        time_req = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("\t%.2lf", time_req);*/

        printf("selection sort:");
        getData(i + 1, a);
        start = clock();
        selectionSort(a, (i + 1) * 100);
        end = clock();
        printf("\nset %d", i + 1);
        printArray(a, (i + 1) * 100);
        time_req = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("\t%.2lf", time_req);
    }
    return 0;
}

void getData(int x, int arr[])
{
    for (int i = 0; i < x * 100; i++)
    {
        arr[i] = rand() % 100 + 1;
    }
}

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

void selectionSort(int arr[], int n)
{
    int i, j, min;
    for (i = 0; i < n - 1; i++)
    {

        min = i;
        for (j = i + 1; j < n; j++)
            if (arr[j] < arr[min])
                min = j;

        if (min != i)
        {
            int temp = arr[min];
            arr[min] = arr[i];
            arr[i] = temp;
        }
    }
}

void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("\t%d", arr[i]);
    }
}
```

Output:


Insertion sort run time
20
15
10
5
0
running time
Line1
name 1
block no.


selection sort run time
20
15
10
5
0
vLabel
Line1
name 1
block no.

Conclusion:

By performing the sorting of arrays of size 100,200,…..100000 , I could see the difference in run times as the number of elements changed in both the algorithms. This made it easy to compare the efficiency of both the algorithms. So,by observing the graphs ,I found insertion sort algorithm to be more efficient for array having more number of elements.