I

Rajshahi University of Engineering & Technology
Department of Computer Science of Engineering

EXPERIMENT NO: 01
NAME OF EXPERIMENT: Complexity Analysis of Bubble Sort, Selection Sort, and Insertion Sort Algorithm and searching algorithm (Linear search and Binary search)

SUBMITTED TO:

(REZOAN TOUFIQ)
(Assistant Professor)
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
RAJSHAHI UNIVERSITY OF ENGINEERING & TECHNOLOGY

SUBMITTED BY:

NAME: Md Tanbeer Jubaer
ROLL NO.: 1903071
GROUP:
DATE OF EXP.: 11/14/2022
DATE OF SUB. :18/14/2022
SERIES: 19

MACHINE CONFIGURATION:

Intel(R) Core(TM) i7-
8750H CPU @ 2.20GHz
2.21 GHz, 16 GB of RAM
1 TB of Hard disk

BUBBLE SORT

It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

Procedure bubble_sort (a: an array of element)

```
procedure bubble_sort( A : list of sortable items ) defined as:
 do
   swapped := false
   for each i in 0 to length( A ) - 1 do:
      if A[ i ] > A[ i + 1 ] then
         swap( A[ i ], A[ i + 1 ] )
         swapped := true
      end if
   end for
 while swapped
end procedure
```

Complexity:

Average and worst case time complexity: n^2
Best case time complexity: n when array is already sorted.
Worst case: when the array is reverse sorted.
The expected and best cases are identical, so bubble sort is $\Omega(N^2)$ and $\Theta(N^2)$.

SELECTION SORT

The algorithm works as follows:

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat the steps above for remainder of the list (starting at the second position)

Procedure selection_sort (a: an array of element)

```
procedure selection_sort( A : list of sortable items ) defined as:
 for i ← 0 to n-2 do
   min ← i
   for j ← (i + 1) to n-1 do
      if A[j] < A[min]
      min ← j
   swap A[i] and A[min]
 end procedure
```

Complexity:;

Best, average and worst case time complexity: n^2 which is independent of distribution of data.

Insertion Sort:

In abstract terms, every iteration of an insertion sort removes an element from the input data, inserting it at the correct position in the already sorted list, until no elements are left in the input. The choice of which element to remove from the input is arbitrary and can be made using almost any choice algorithm.

procedure insertion_sort(A : list of sortable items) defined as:

```
insertionSort(array A)
   for i = 1 to length[A]-1 do
```

III

```
begin
  value = A[i]
  j = i-1
  while j >= 0 and A[j] > value do
  begin
     A[j + 1] = A[j]
     j = j-1
  end
 A[j+1] = value
end
```

Complexity:;
Average and worst case time complexity: n^2
Best case time complexity: n when array is already sorted.
Worst case: when the array is reverse sorted.
The expected and best cases are identical, so bubble sort is Ω (N^2) and Θ (N^2).

Linear Search
Suppose an array A with elements indexed 1 to n is to be searched for a value x. The following pseudo code performs a forward search, returning n + 1 if the value is not found:

```
Linear_Search(A,x):
   Set i to 1.
   Repeat this loop:
      If i > n, then exit the loop.
      If A[i] = x, then exit the loop.
      Set i to i + 1.
   Return i.
```

Complexity:;
In linear search, best-case complexity is O(1) where the element is found at the first index. Worst-case complexity is O(n) where the element is found at the last index or element is not present in the array.

Binary Search

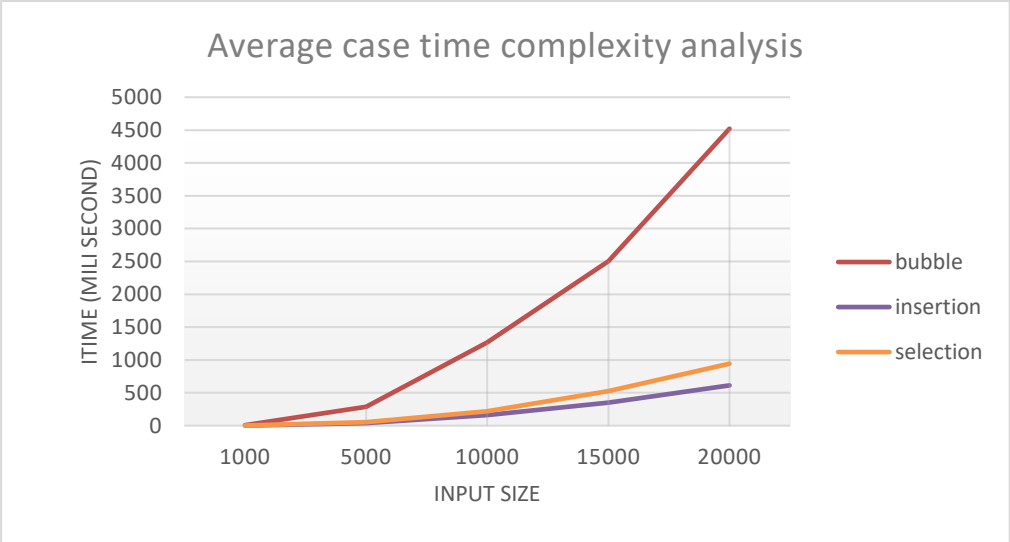
```
BinarySearch(A[0..N-1], value) {
  low = 0
  high = N - 1
  while (low <= high) { mid =
    (low + high) / 2 if (A[mid] >
    value)
    high = mid - 1
    else if (A[mid] < value) low =
    mid + 1
    else
     return mid // found
  }
  return -1 // not found
}
```

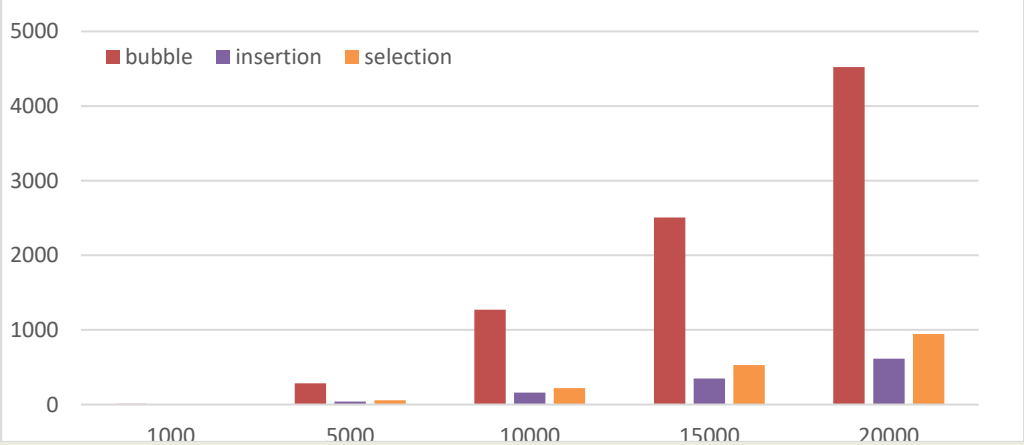
Complexity::

In binary search, best-case complexity is O(1) where the element is found at the middle index. The worst-case complexity is O(log2n).

DATA TABLE AND GRAPH:

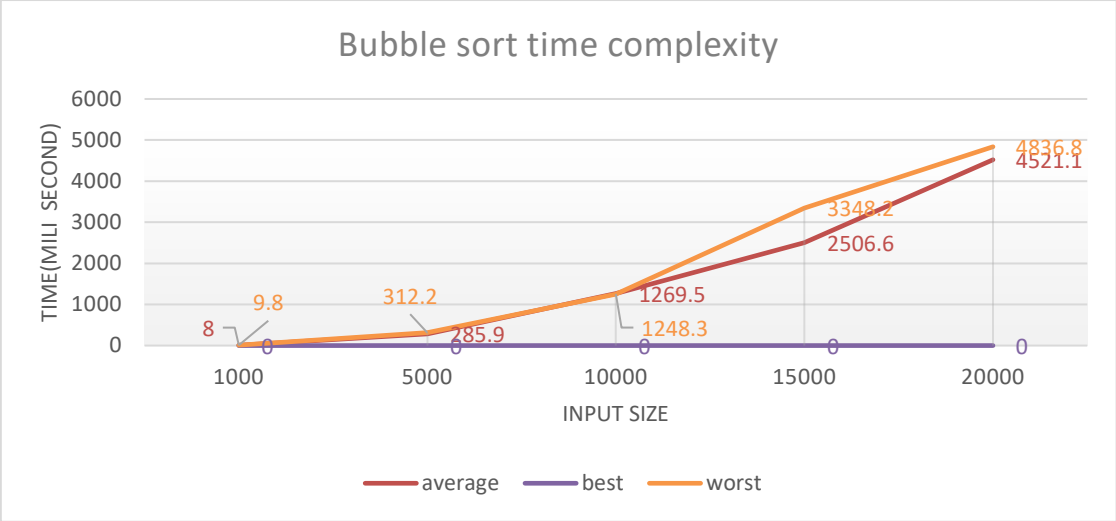
	Required Time (milli seconds)		
No. of Data	**Bubble sort**	**Insertion sort**	**Selection sort**
1000	8.015	1.008	2.03
5000	285.953	38.41	55.855
10000	1269.534	161.641	220.217
15000	2506.678	350.466	528.513
20000	4521.108	613.493	943.978

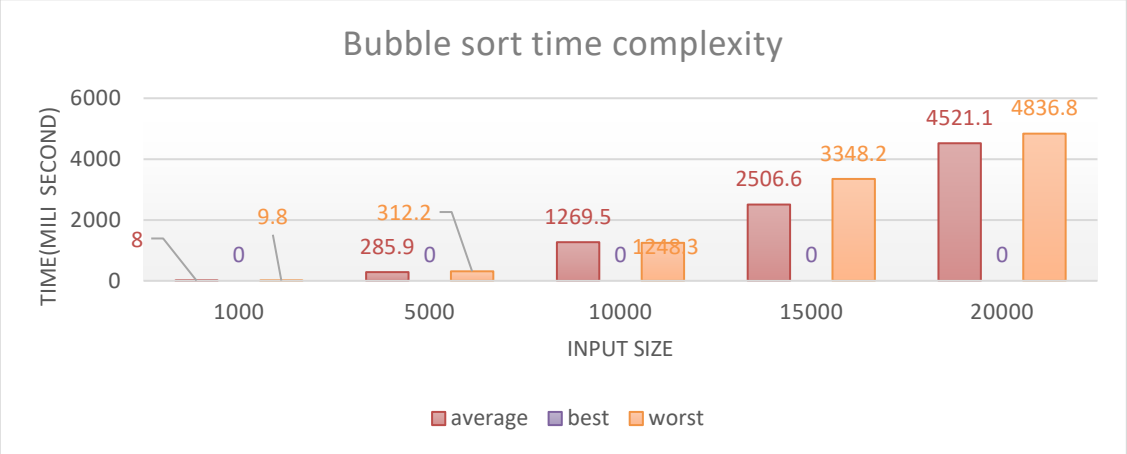




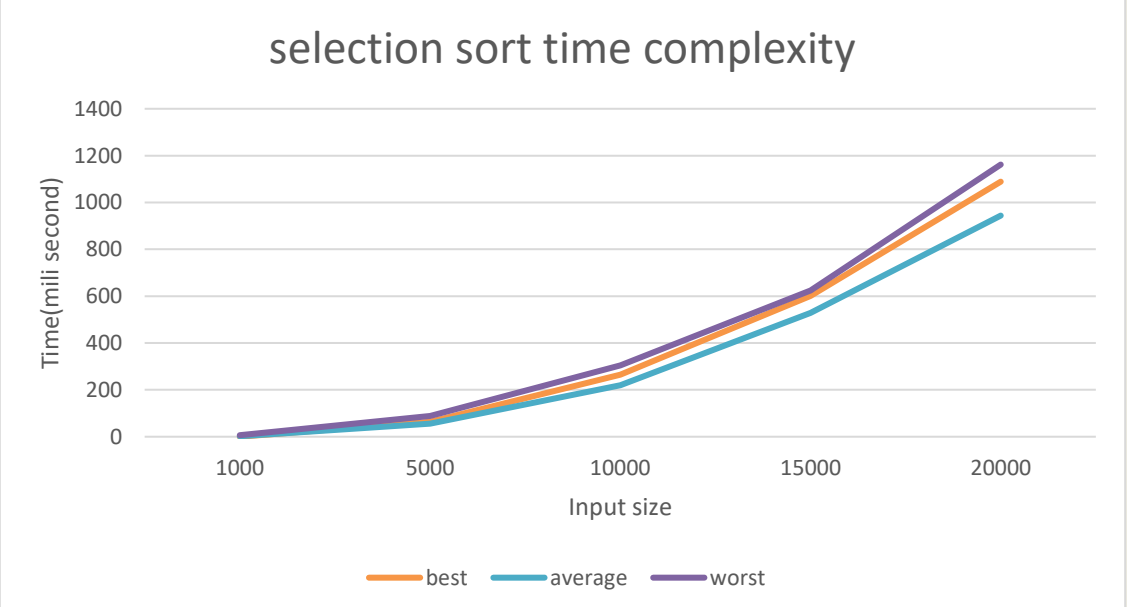
	Required Time in Bubble sort (milli seconds)		
No. of Data	**Best case**	**Average case**	**Worst case**
1000	0	8	9.8
5000	0	285.9	312.2
10000	0	1269.5	1248.3
15000	0	2506.6	3348.2
20000	0	4521.1	4836.8

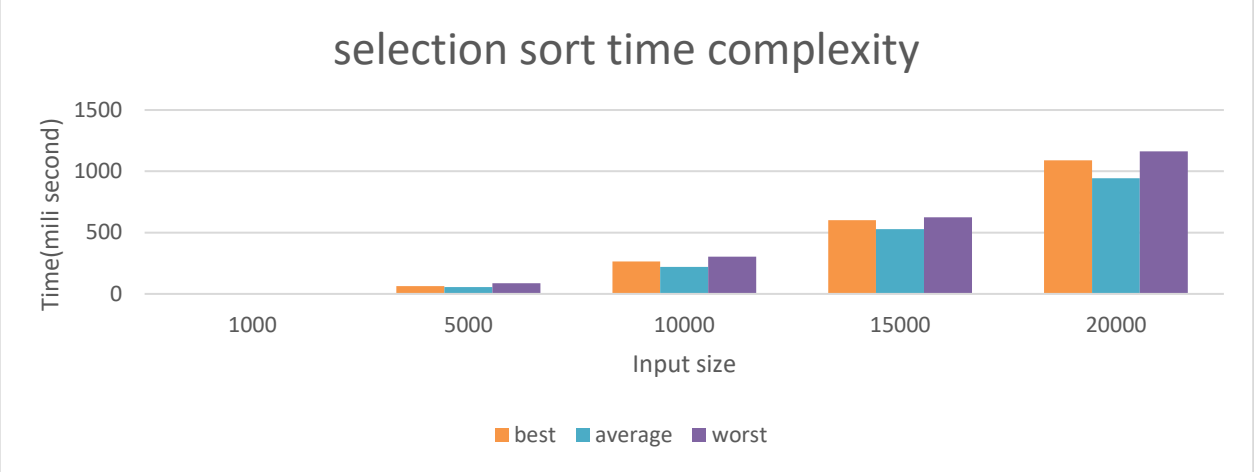
V



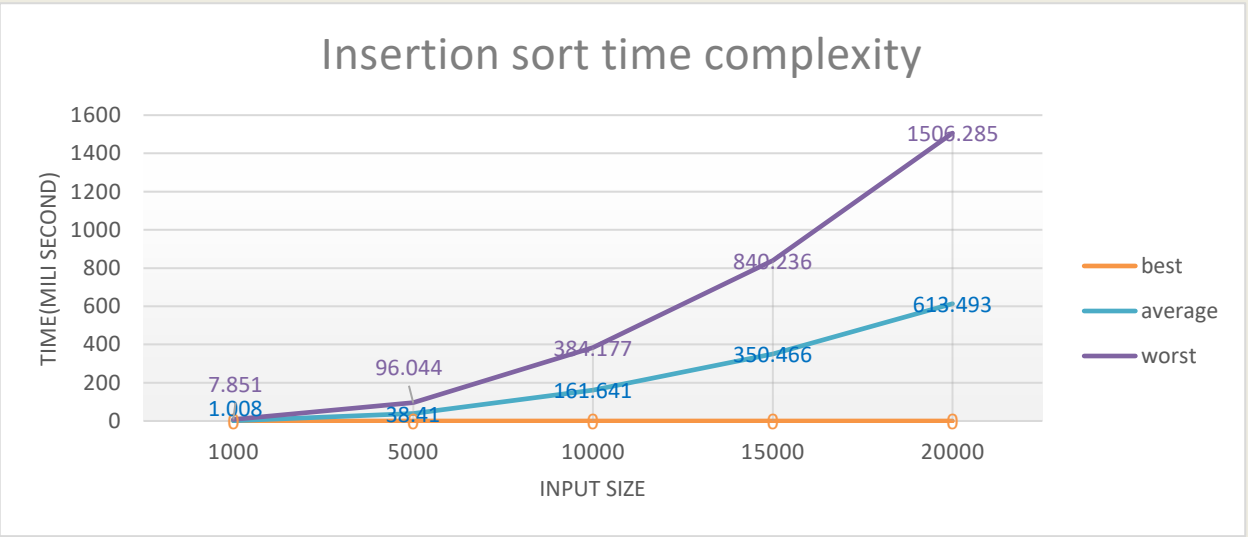


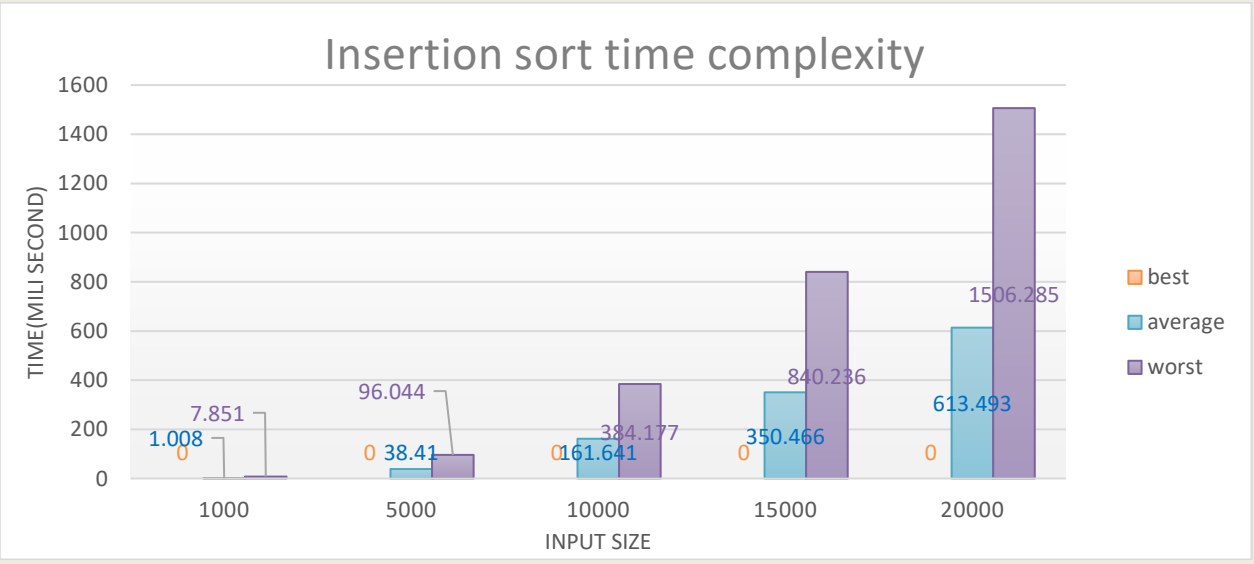
	Required Time in Selection sort (milli seconds)		
No. of Data	**Best case**	**Average case**	**Worst case**
1000	0	2.03	6.542
5000	64.151	55.855	88.036
10000	264.284	220.217	304.42
15000	600.913	528.513	624.64
20000	1088.859	943.978	1161.856



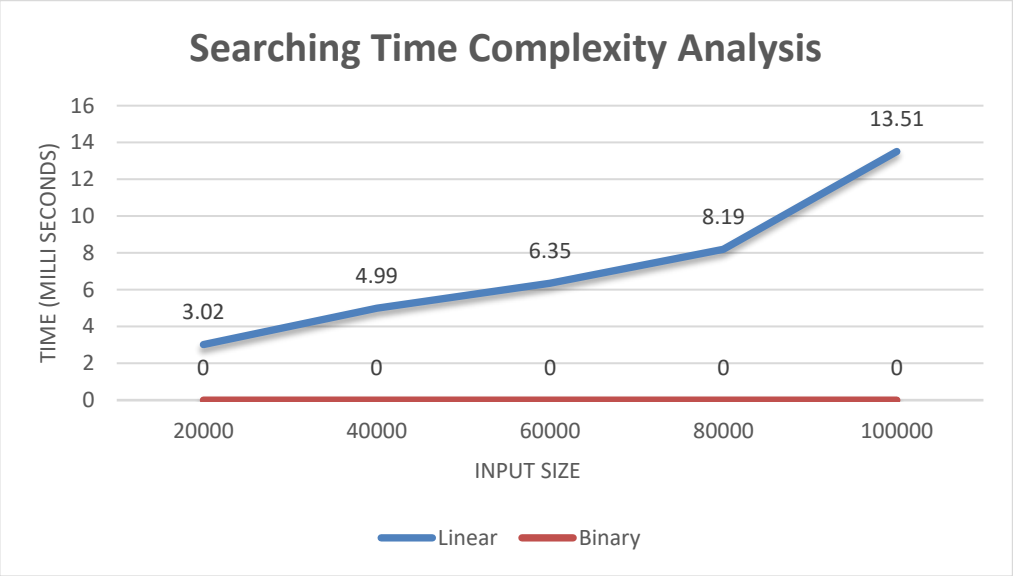


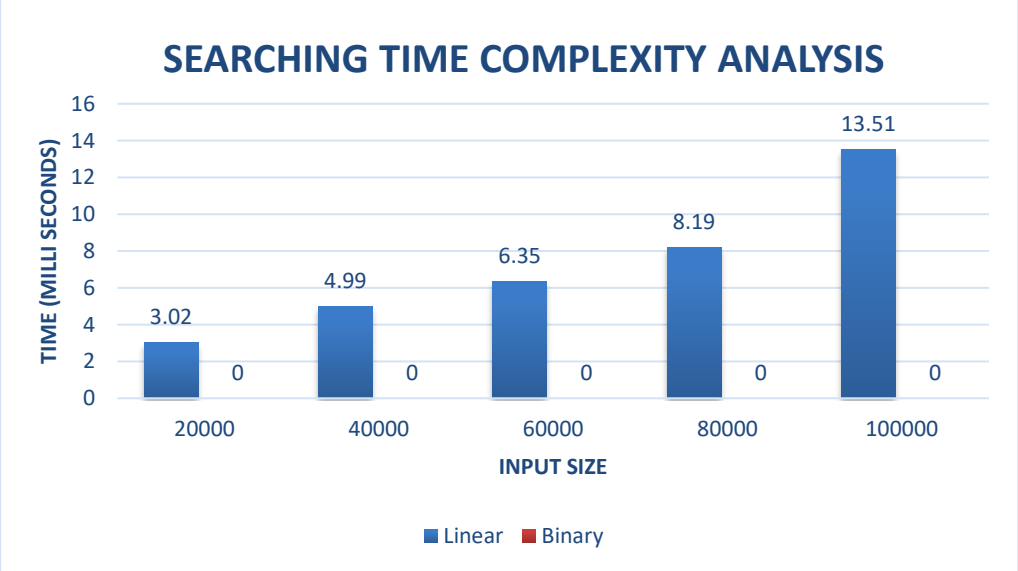
	Required Time in Insertion sort (milli seconds)		
No. of Data	**Best case**	**Average case**	**Worst case**
1000	0	1.008	7.851
5000	0	38.41	96.044
10000	0	161.641	384.177
15000	0	350.466	840.236
20000	0	613.493	1506.285





	Required Time in Searching Algorithms(milli seconds)		
No. of Data	**Linear search**	**Binary Search**	
20000	3.02	0	0
40000	4.99	0	0
60000	6.35	0	0
80000	8.19	0	0
100000	13.51	0	0





DISCUSSION:

Today we have analyzed the time complexity of some searching and sorting algorithms. We have used time function to calculate the time between the required function to be executed. We have measured the time in milli seconds. If we see the graphical output and bar graph for average case complexity, we can see that the average case for bubble sort is the worst. Clearly selection sort and insertion sort are better in terms of time complexity. Between insertion sort and selection sort insertion sort is slightly better. Though the average case complexity for both of them is O(n^2) theoretically. But in real time we can observe that insertion sort is better. Now if we talk about the complexity of bubble sort for best average and worst case, we can see the difference from fig 3 and 4. 0 micro second was taken by bubble sort in its best case and it is alright I think because the best case, complexity is O(n). But the average case and worst case are closer. So, I think its not a better choice for the average case condition. A huge difference comes in selection sort. Its best case, average case worst case all are same theoretically but in fig 5 and 6 we can see that the average case runs more efficiently. And its not a better choice if the data is in the best case(sorted). Finally, in insertion sort we can see a better output in every term. Its average case is average not only theoretically but also practically, its best case is just like bubble sort but its worst case is more efficient than bubble sort. So, after analyzing all we can say insertion sort is best among them. Now for searching algorithm we see that the binary search is far better than linear search. In terms of output we have observed in our machine we see that when the input size increases the time taken by linear search increases visually. But it takes no effect on binary search. Compared to linear search it is tensed to zero. So binary search is very efficient. But the disadvantage of it is that the data needs to be sorted.

Questions ?::

1. In our lab we were asked to use file system and random integers. Why do we need them?
2. Can we use pseudo random numbers to make this more efficient?

Answer ::

- **I think** in order to analyze the complexity for different algorithms we needed the file system. We have used random integers in order to form an array. Now suppose we have formed an array of size 5 to check bubble sort's complexity and there is a probability that the numbers that are generated are already sorted. So, this time bubble sort will run in its best case. After that suppose we want to check insertion sort and we have generated a list of random numbers again but there is a probability that the numbers that we have generated are reversely sorted so if we check this for insertion sort it will run in its worst case. So, clearly there is a chance of unfairness while computing. I think that's why we were asked to write down the list in a text file and every time we want to check we can just read the data from file and analyze different algorithm. This time there will be no chance of unfairness because we are analyzing with the same data from the file.

- **I think we can use pseudo** random numbers to avoid the file input output system. Because we know from the discrete mathematics course that pseudo random numbers always maintain a sequence for some specific values of a, m, x and c. No matter if we change the input size we will always find the same sequence of data. So, to check for different algorithms we don't need to maintain the file system. And if we want to change the sequence of data list we just need to change the values of a m or c. So, I think it will be efficient in terms of memory because we will not need the extra space for text file.