HACETTEPE UNIVERSITY DEPARTMENT OF COMPUTER ENGINEERING
BBM204

Prepared by Burak YILMAZ
Number : 21627868
E-Mail : burak040898@gmail.com
Subject : Assignment 1

Introduction

- In this assignment we are expected to analyze different sorting algorithms additionally binary search, compare their asymptotic complexities in terms of time performance (not the memory usage) with experimental running times also we should compare experimental running time with theoretical running time. Considering these empirical data’s, we can interpret under which conditions these theoretical data’s and complexities are valid.

Background

First of all, what is an algorithm?

- Algorithm is a step by step method of solving a problem. It is commonly used for data processing calculation and other related computer and mathematical operations. In this assignment, we focused on different kind of sorting algorithms and one search algorithm called binary search algorithm.

What about sorting algorithms?

- Sorting algorithms are algorithms that puts the elements of an array or list etc. in a certain order. Efficient sorting algorithms are very important because we usually need sorted data’s in our applications or for our operations while we are writing code also nowadays for a good programmer there are 2 important topics to consider one is memory usage and the other one is time. Our programs should use less memory and has to be executed as fast as possible to satisfy our demands.

Problem Definition

When we are talking about analysis of algorithms we basically saying that is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required. However, the main concern of the analysis of algorithms is the required time or performance not the size. Analysis of algorithms is mainly used to predict performance and compare algorithms that are developed for the same task

Generally, we perform the following types of analysis;

- Worst-case is the maximum number of steps taken on any instance of size **N**.
- Best-case the minimum number of steps taken on any instance of size **N**.
- Average case is the average number of steps taken on any instance of size **N**.

Usually worst case is considered since it gives an upper bound on the performance that can be expected. Additionally, the average case is usually harder to determine (its usually more data dependent), and is often the same as the worst case I should mention that the Time performance depends on lots of things like hardware, operating system, processors, etc. However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm and ignoring the other details.

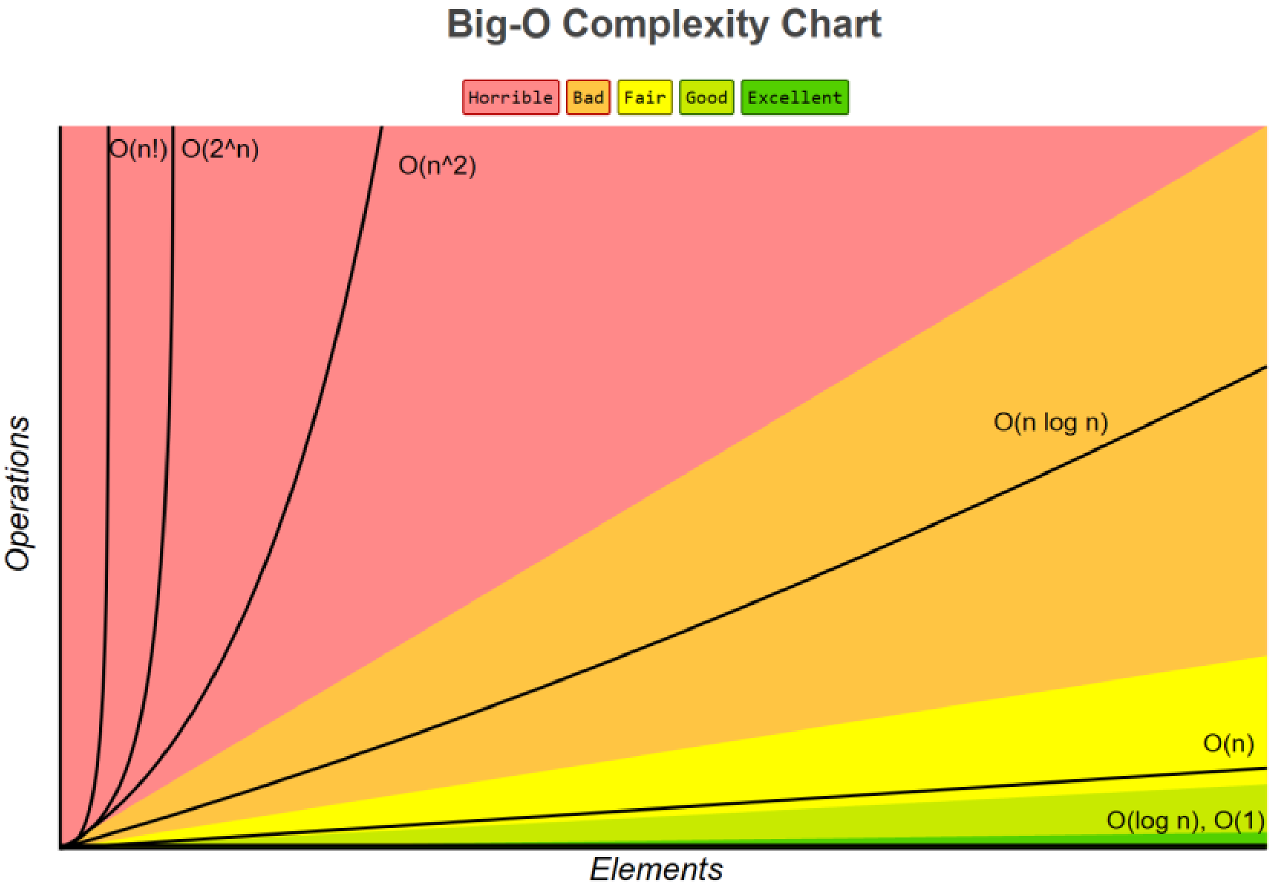
Measuring the Time

For measuring the time, I create a class called Stopwatch. While measuring the time complexity of the given algorithm I create a stopwatch object, when it occurs System time save into a variable, with the Elapsedtime function I measure the time passes from the creation of the object till the algorithm finishes. It can be nanoseconds or milliseconds.

Main Goal,Findings and Steps While Doing Experiment

- The main objective of this assignment is to show the relationship between the running time of implementations with the theoretical asymptotic complexities.
- While doing that experiment I encountered with noise but to reduce to noise and provide the accuracy of the running time I repeat the executions for each sorting algorithms for each input 300 times so my results are very close to theoretical data. Of course, some of the factors affects the result but their effects are less enough to be ignored. I am going to mention more below why some of the results are distinct from the other results. I plot three graphs for each sorting algorithm. These graphs represent worst case, average case and best case. You can easily detect which graph is the corresponding analysis of each sorting algorithm by looking the label on the right.
- I followed some steps while doing this experiment here are the steps:
- 1-I generated **N** random integer numbers
- 2-I run the different algorithms on randomly generated integer numbers.
- 3-I used Milliseconds for Merge, Radix, Insertion sorts and Nanoseconds for binary search and Insertion sort and I recorded the execution times. I will explain why I used different types of time when I am comparing the results of the algorithms.
- 4-I repeated the steps 1 to 3. **N** can vary between 1000-10000 for Merge, Selection, Insertion sorts and for Radix **N** can vary between 5000-50000 and for binary search It vary between 2000-20000 integers.
- 5-Finally I plot the graphs for each analysis of algorithms here you can find below.

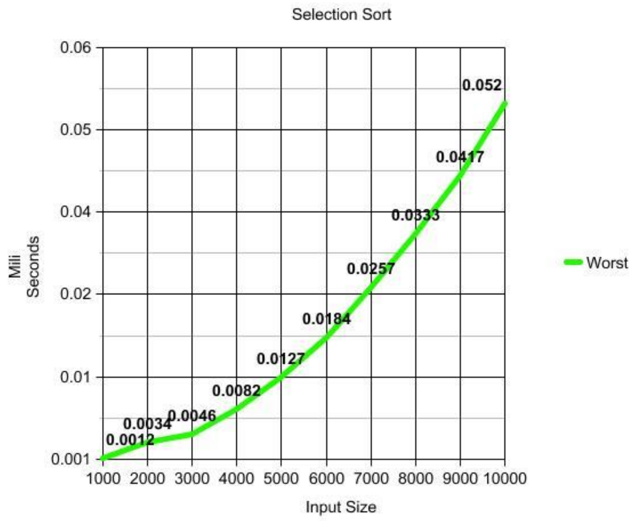
Below you can see the Big-O Complexity Chart with theoretical time complexities. With comparing the plots that I reached you can see the accuracy of the experiment.

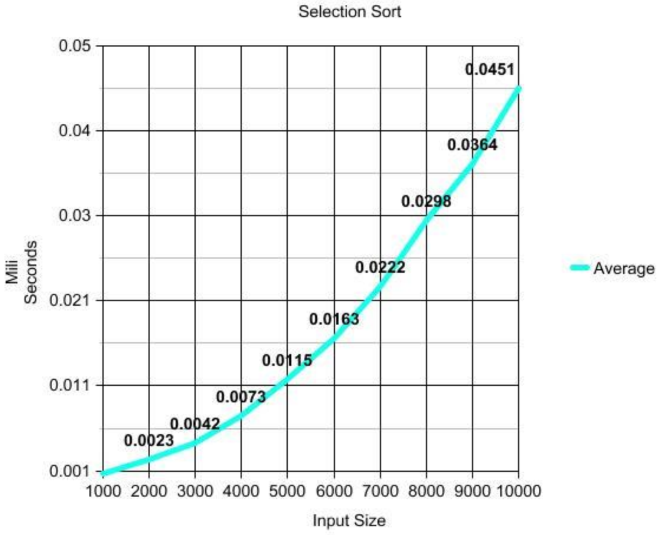


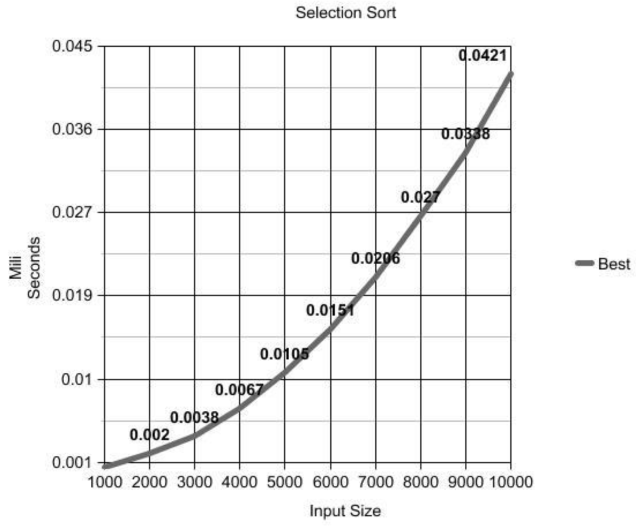
Selection Sort Algorithm Analysis

- The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray. Selection sort uses **(N– 1) + (N– 2) + ... + 1 + 0 ~ N^2 / 2** compares and **N** exchanges in each case (worst, best, average).
- The theoretical time complexities of selection sort for the best, worst and average case are **O(n^2).**
- The empirical time complexities of selection sort for the best, worst and average case that I reached after the experiment are **O(n^2)** also.
- Here are the graphs for 10 different input sizes on the X axis and on the Y axis the corresponding milliseconds for those input sizes.

- Finally, I should mention that for the best case analysis I used sorted array input, for the worst case analysis I used reversed sorted array and for the average case analysis I used randomly ordered array.



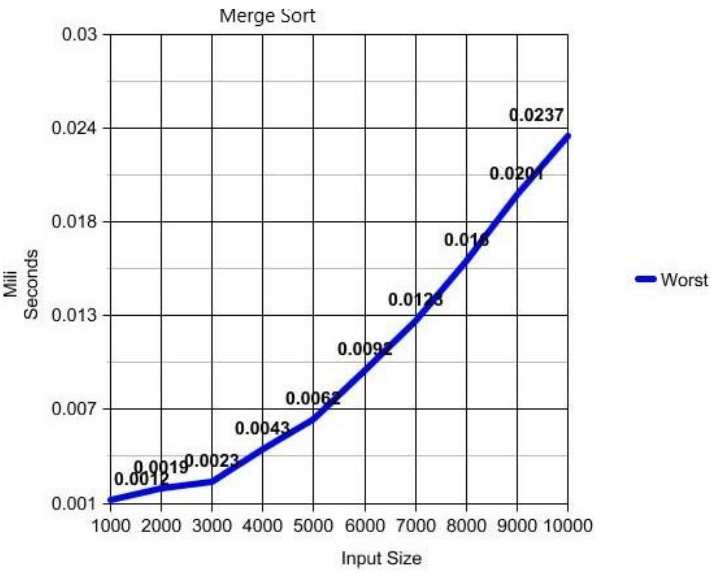


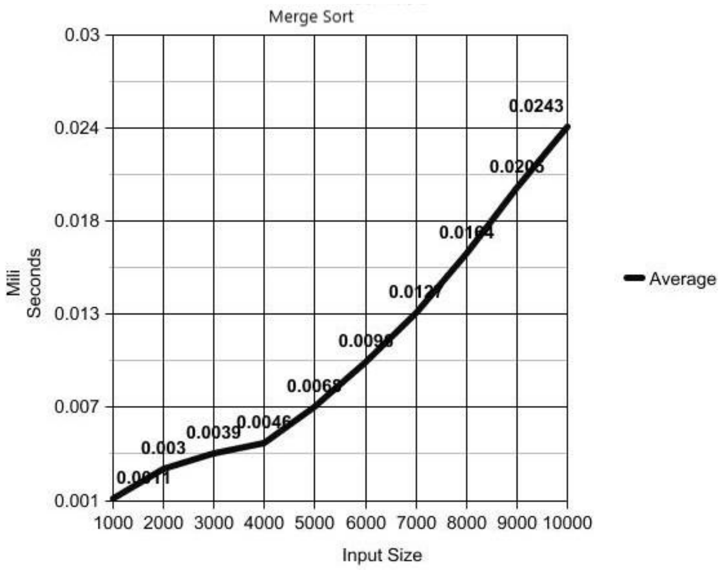


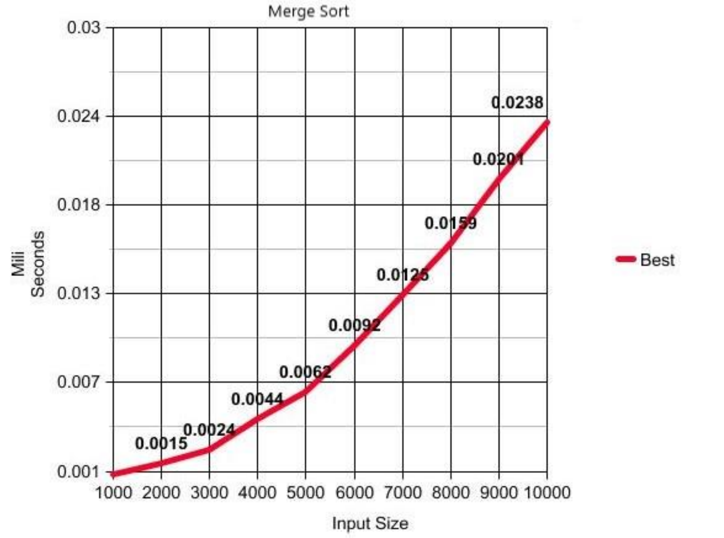
- The curves are almost smooth. There can be small increases or decreases this might have happened because of hardware or the disk usage at the current execution time etc. I tried to avoid the noise by repeating the same steps 300 times and take the average time of repetitions. Also for the best, worst and average case never changes for Selection sort because of the algorithm.
- Briefly in selection sort has to find minimum element in each iteration with comparing one element with the others and if that element smaller than the value hold by selection sort It changes the minimum to that value. So that can happen with two nested loops. So that means **O(n^2)** time complexity for best, average and worst case. Our results are also satisfying these theoretical data's.
- In Addition Selection sort also has **O(1)** space complexity.

Merge Sort Algorithm Analysis

- Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. Mergesort uses at most **N lg N** compares in each case (worst, best, average).
- The theoretical time complexities of Merge sort for the best, worst and average case are **O(n log(n)).**
- The empirical time complexities of Merge sort for the best, worst and average case that I reached after the experiment are **O(n log(n))** also.
- Here are the graphs for 10 different input sizes on the X axis and on the Y axis the corresponding milliseconds for those input sizes.
- Finally, I should mention that for the best case analysis I used sorted array input, for the worst case analysis I used reversed sorted array and for the average case analysis I used randomly ordered array.



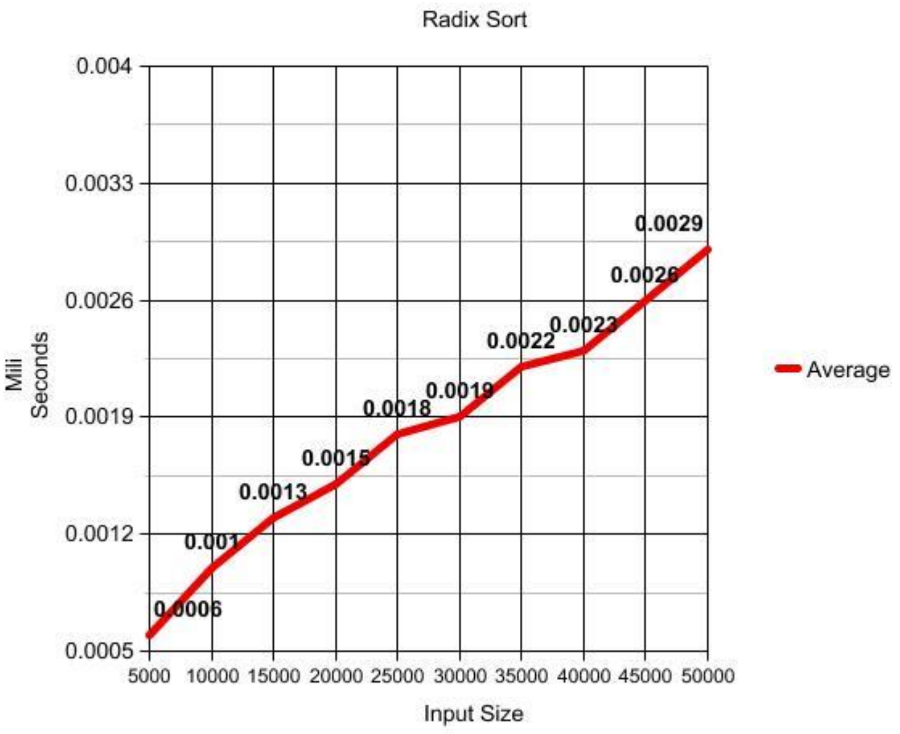


- 

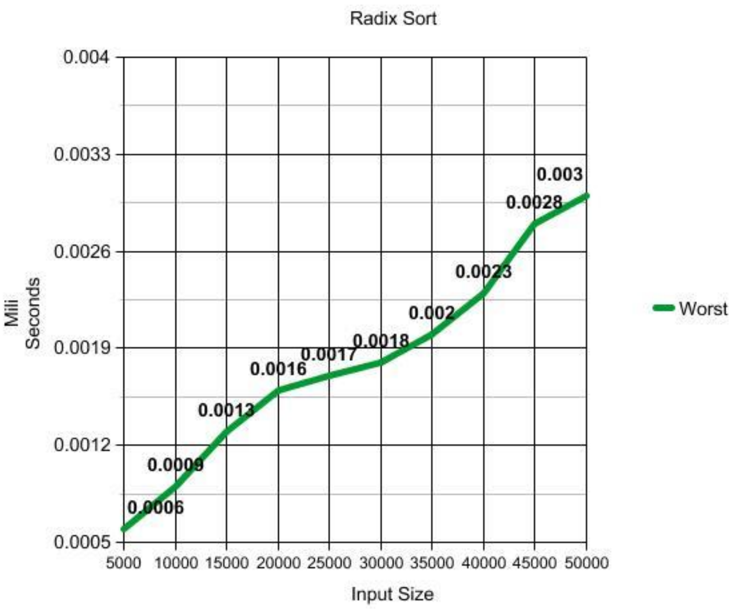
- The curves are almost smooth and satisfies the **NlogN** function mostly. There can be small rises and falls this might have happened because of hardware or the disk usage etc. at the current execution time. I tried to avoid the noise by repeating the same steps 300 times and take the average time of repetitions. Also for the best, worst and average case never changes for Merge sort because of the algorithm.
- Briefly in Merge sort it has to divide the whole array into two parts recursively until the recursive condition satisfies after that it merge the two halves of the array. So that means **O(n log(n))** time complexity(**logN** part comes from dividing the array into two halves each time **N** part comes from its comparisons that are made at each stage) for best, average and worst case. Our results are also satisfying these theoretical data's.
- In Addition, Merge sort also has **O(n)** space complexity this is the disadvantage of Merge sort.
- Finally, I should mention that for the best case analysis I used sorted array input, for the worst case analysis I used reversed sorted array and for the average case analysis I used randomly ordered array.

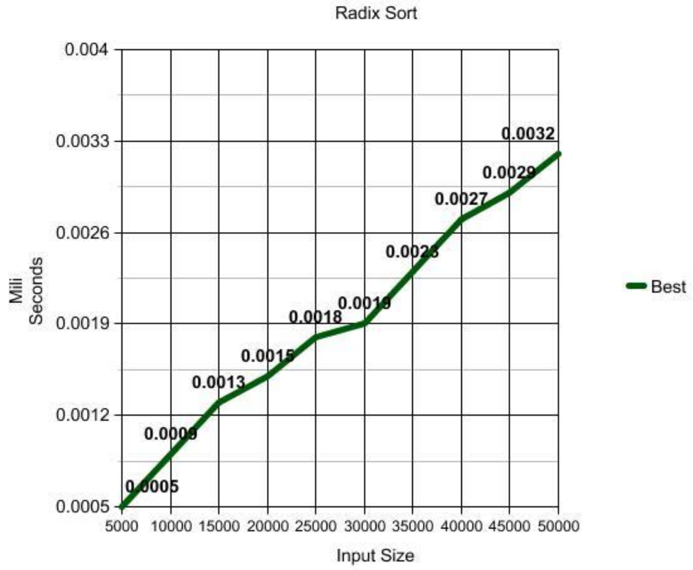
Radix Sort Algorithm Analysis

- Radix sort is one of the sorting algorithms used to sort a list of integer numbers in an order. In radix sort algorithm, list of integer numbers will be sorted based on the digits of numbers. Sorting is performed from least significant digit to the most significant digit. Moreover, Radix Sort is not based on key comparison sorting algorithm. It can differ with the other algorithms that I mentioned in this report. Radix sort algorithm requires number of passes which are equal to the number of digits present in the largest number among the list of numbers. For example, if the largest number is a 5-digit number then that list is sorted with 5 passes.
- The theoretical time complexities of Radix sort for the best, worst and average case are **O(kN)** (k is the number of digits of maximum value).
- The empirical time complexities of Radix sort for the best, worst and average case that I reached after the experiment are **O(kN)** also.
- Here are the graphs for 10 different input sizes on the X axis and on the Y axis the corresponding milliseconds for those input sizes. Also, Radix sorts inputs are starting from 5000 to 50000 because it is faster than the other algorithms so if I use small input sizes I may encounter with 0.0 milliseconds and it is not useful to plot the graphs or making comparisons.



-

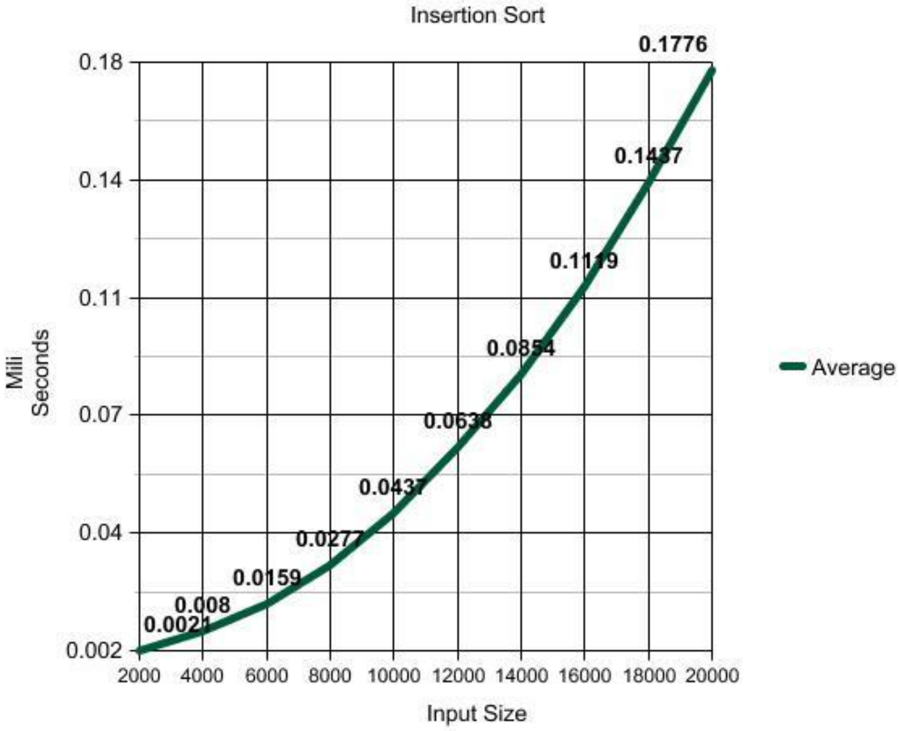


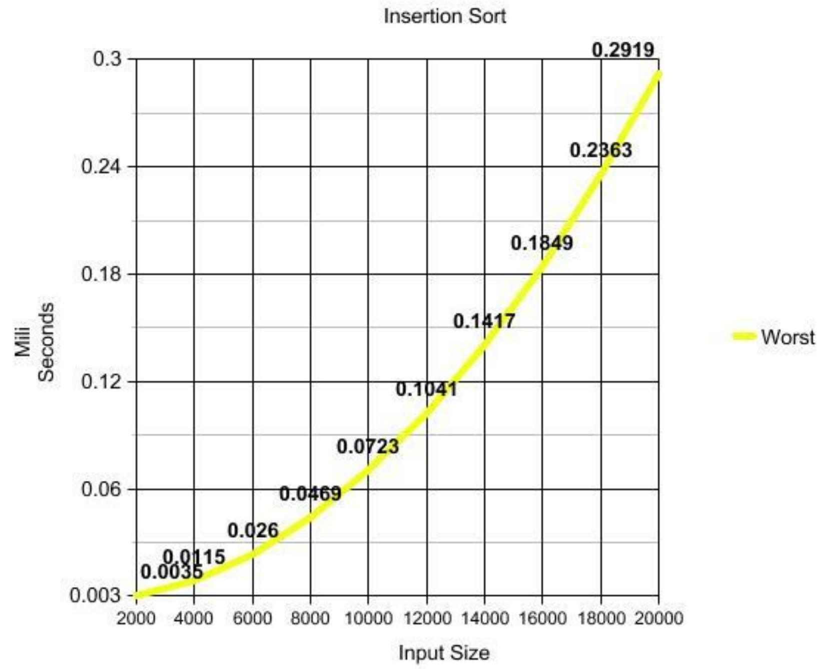


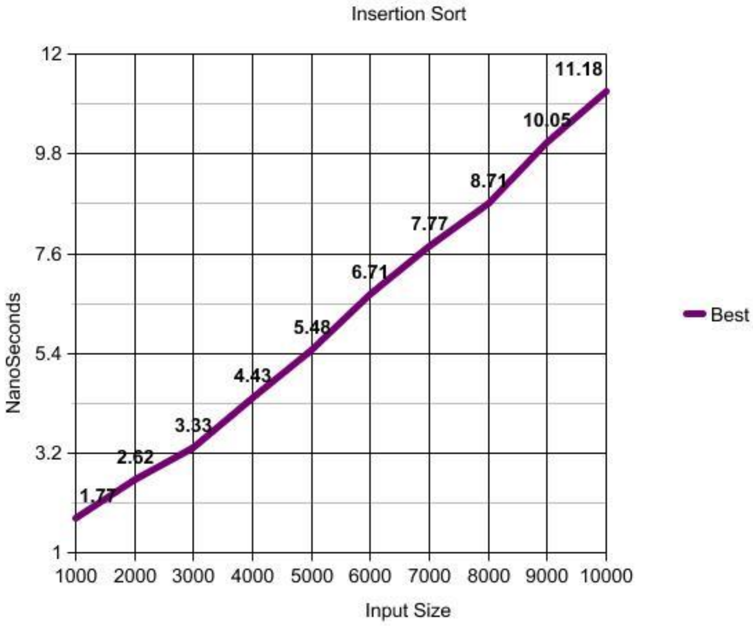
- The curves are almost smooth and satisfies the **N(linear)** function mostly. There can be small rises and falls this might have happened because of operating system or the disk usage etc. at the current execution time. I tried to avoid the noise by repeating the same steps 300 times and take the average time of that repetitions. Also, for the best, worst and average case never changes for Radix sort because of the algorithm.
- Briefly in Radix sort it works by sorting each digit from least significant digit to most significant digit. So, in base 10, radix sort would sort by the digits in the 1's place, then the 10’s place, and so on... So that means **O(kN)** time complexity for best, average and worst case. Our results are also satisfying these theoretical data’s.
- In Addition Radix sort also has **O(n+k)** space complexity this is the disadvantage of Radix sort.

Insertion Sort Algorithm Analysis

- Insertion sort is an in-place comparison-based sorting algorithm. 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's as its average and worst case complexity are of **O(n^2)**, where N is the number of items but the best case for Insertion sort is **O(n)**. Best case for Insertion sort If the array is in ascending order, insertion sort makes **N-1** compares and 0 exchanges. If the array is in descending order (and no duplicates), insertion sort makes **~ ½ N^2** compares and **~ ½ N^2** exchanges and that is the worst case.
- The theoretical time complexities of Insertion sort for the worst and average case are **O(n^2)** and for the best case is **O(n).**
- The empirical time complexities of Insertion sort for the worst and average case that I reached after the experiment are **O(n^2)** and for the best case **O(n)** also.
- Here are the graphs for 10 different input sizes on the X axis and on the Y axis the corresponding milliseconds for those input sizes but just only worst and average case. For the best case I used nanoseconds and input sizes starting from 2000 to 20000.Because Insertion sort best case is fast and I got 0.0 milliseconds when I used milliseconds.
- Finally, I should mention that for the best case analysis I used sorted array input, for the worst case analysis I used reversed sorted array and for the average case analysis I used randomly ordered array.







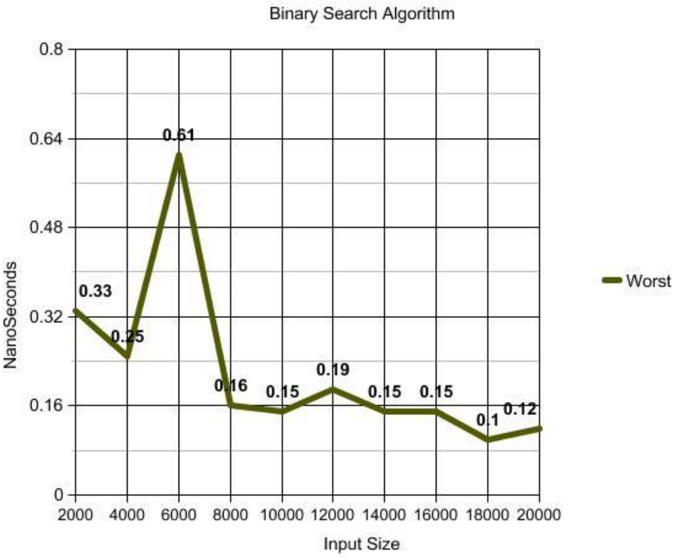
- The curves are almost smooth. There can be small rises or falls this might have happened because of hardware or the disk usage at the current time etc. I tried to avoid the noise by repeating the same steps 300 times and take the average time of repetitions. Worst and average case have **O(n^2)** time complexity and best case has **O(n)** time complexity.
- Briefly in Insertion sort assume that the first element is already sorted and then pick the next element compare with all the elements in sorted sub-list after that shift all the elements in sorted sub-list that is greater than the value to be sorted. Insert the value. Repeat until the list is sorted. This can be done in nested loops so that means time complexity is **O(n^2)** but in best case you break the loop in one comparison for each element because there is no greater value in sub-list. So in

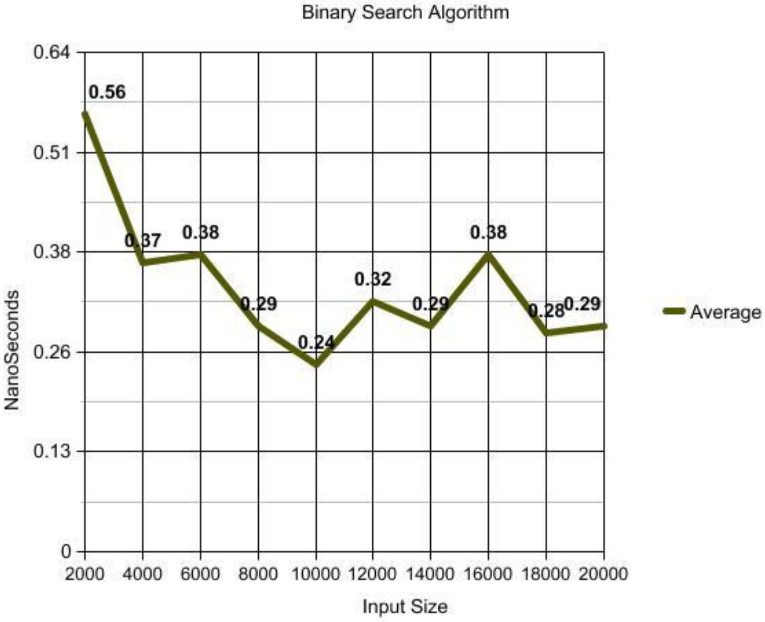
best case you make at most **N-1** comparisons that corresponds to **O(n)** time complexity.

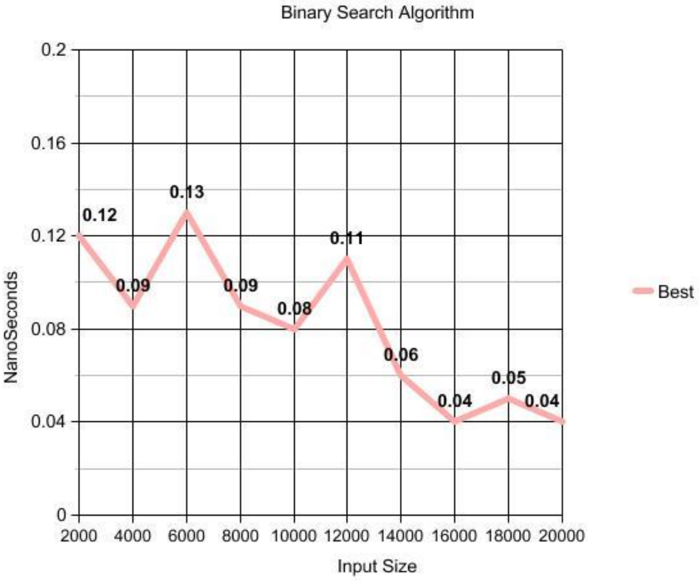
- In Addition Insertion sort also has **O(1)** space complexity.

Binary Search Algorithm Analysis

- Binary Search is applied on the sorted array or list of large size. It's time complexity of **O(log n)** makes it very fast as compared to other sorting algorithms. The only limitation is that the array or list of elements must be sorted for the binary search algorithm to work on it. Binary search uses **log N** compares to search any array of size **N.**
- The theoretical time complexities of Binary Search for the worst and average case are **O(logN)** and for the best case is **O(1)** because It searches the middle element of the array so at the first compare it finds the searched value.
- The empirical time complexities of Binary search for the worst and average case that I reached after the experiment is similar to **O(logN)** and for the best case **O(1)** but considering my experiment results there are few differences.
- The problem is sometimes on the first try my results occurs very high than the other experiment results of without any reason and after first try it becomes normal and when the input size is getting bigger time complexity sometimes falls and gives better results. I searched the problem that I encounter on the internet and also, I consult my teachers about that problem and I found some answers but neither on the internet nor from my teachers I could not find any mutual solution or exact solution. One of the reason can be arise from cache memory, the operating system or processor they said, and the other reason is can be disk usage at current execution time. I try to avoid the noise but could not achieve %100 accurate. I repeated the steps 1000 times for each input size and take the average of the result and got more smooth results except the problem that I mentioned above.
- Finally, I should mention that for the binary search we need sorted array so for all cases (worst, best, average) I used sorted array. Implementing radix sorting algorithm to sort on the array and then apply the binary search.
- Here are the graphs for 10 different input sizes on the X axis and on the Y axis the corresponding nanoseconds for those input sizes. Also, Binary Search algorithm inputs are starting from 2000 to 20000.I used nanoseconds in binary search because when I used milliseconds even if I used 40000000 input size I got 0.0 result so I changed it to nanoseconds to get smooth results.


Binary Search Algorithm
NanoSeconds
Input Size
0
0.16
0.32
0.48
0.64
0.8
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000
0.33
0.25
0.61
0.16
0.15
0.19
0.15
0.15
0.1
0.12
Worst


Binary Search Algorithm
NanoSeconds
Input Size
0
0.13
0.26
0.38
0.51
0.64
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000
0.56
0.37
0.38
0.29
0.24
0.32
0.29
0.38
0.28
0.29
Average


Binary Search Algorithm
NanoSeconds
Input Size
0
0.04
0.08
0.12
0.16
0.2
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000
0.12
0.09
0.13
0.09
0.08
0.11
0.06
0.04
0.05
0.04
Best

- Briefly in Binary Search Algorithm start with an array sorted in descending
- In each step pick the middle element of the array called middle and compare it with the searching key. If element values are equal, then return index of middle. If key is greater than middle, then key must be in left subarray. If middle is greater than key, then key must be in the right subarray.
- Repeat those steps on new subarray.
- In Addition Binary Search Algorithm also has **O(1)** space complexity.

Comments and Notes

As a result of those algorithm analysis we can reach some conclusions.

• The speed of the algorithms are in average case given in that order

- Radix sort > Merge Sort > Insertion Sort = Selection Sort

• The speed of the algorithms are in worst case given in that order

- Radix sort > Merge Sort > Insertion Sort = Selection Sort

• The speed of the algorithms are in best case given in that order

- Insertion Sort >= Radix Sort > Merge Sort > Selection Sort

• For different kind of data sets we can use different algorithms to optimize our code

• For example, for larger data sets it is more suitable to use radix sort or merge sort rather than selection or insertion sort or if you know the data you are

going to sort and if the values are almost sorted we can use insertion sort.

• For smaller data sets we can use any of them because the difference is

almost impossible to distinguish.

• If the data is reversed sorted you may use Radix Sort or Merge Sort again to sort in other way these algorithms are going to work much more faster than Selection sort and Insertion Sort.

For searching an element in a sorted array binary search is very efficient way to choose.

• If the element is at the middle it takes constant time to find it **O(1).**

• If the element that you are looking for is not in the array it takes at most **logN** compares. It corresponds to worst case.

• Average case is also similar to worst case. It can be the middle element last element or somewhere in the array so it has **logN** time complexity also.

- My laptop features are , 2.5GHZ, Intel Core i7 ,12 GB DDR4 ,1 TB ,16 GB SSD. All algorithms are done in 9.5 minutes...

- The most challenging parts are getting results for Insertion sort best case and getting presice results for Binary search algorithms.I explained their challenging parts above.

- As a last conclusion, there are many other sorting algorithms but it is computer engineers task to choose which algorithm is the best, efficient and faster to apply the data set that we have.