Kristianstad University
SE-291 88 Kristianstad
+46 44-250 30 00
www.hkr.se

Seminar : Quick Sort, Insertion Sort, Binary Search

Course: Algorithms and Data Structures (DA256F)

Meenu Gupta

Content

1 Introduction

1.1 Background/idea

The Seminar is related to Sorting and Searching algorithms. It consists of two tasks:

Task 1: Writing Code to implement insertion sort and quick sort recursively as well as iteratively. Further quick sort is implemented using three different strategies to select pivot is:

a) First element
b) Median of three pivot
c) Random pivot. a queue using stacks & stack using queue.

Task 2: Writing Code to implement Binary Search using recursion.

1.2 Purpose

Purpose of the report:

- For Task 1 - To measure the running time for different algorithms using recursive and iterative approach for various input sizes.
- For Task 1- Testing out how running time changes using different implementation, pivot strategies and input sizes.
- For Task 2 - To measure the running time of Binary Search (Recursive) for various input sizes.
- Present the results in a table and graph for Task 1 & Task 2.
- Analyze the results for Task 1 & Task 2.

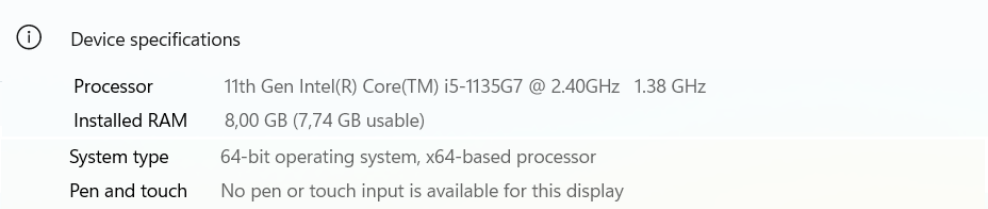
2 Method

- The code developed for all tasks is written in Python language.

- For sorting algorithm -Task 1 file containing unsorted random numbers between 0 and 99 (including duplicates) is used to run the test which was provided along with tasks by the instructor.

- For binary search algorithm - Task 2 random sorted array has been generated ranging from 1 to 1000000 without duplicates to get more exact time calculation, as in case of binary search it returns the first element in duplicate values.

- Device Specifications used for running test is Dell personal laptop:



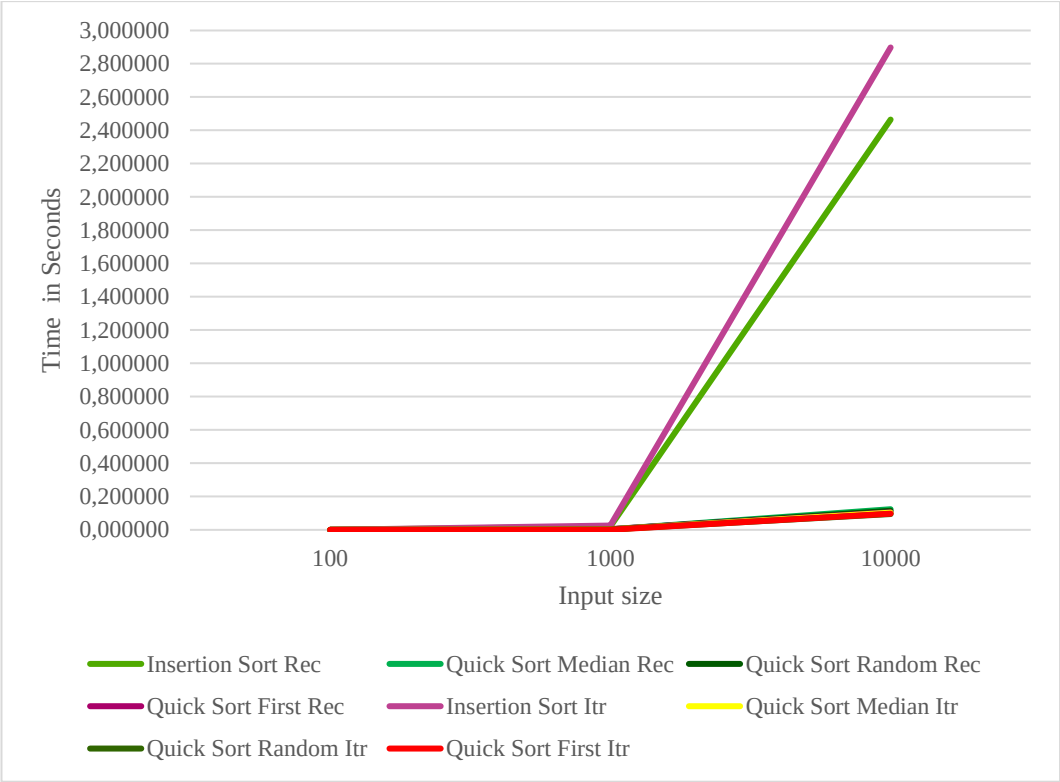
3 Results (Table & Graph)

3.1 Task 1 Results:

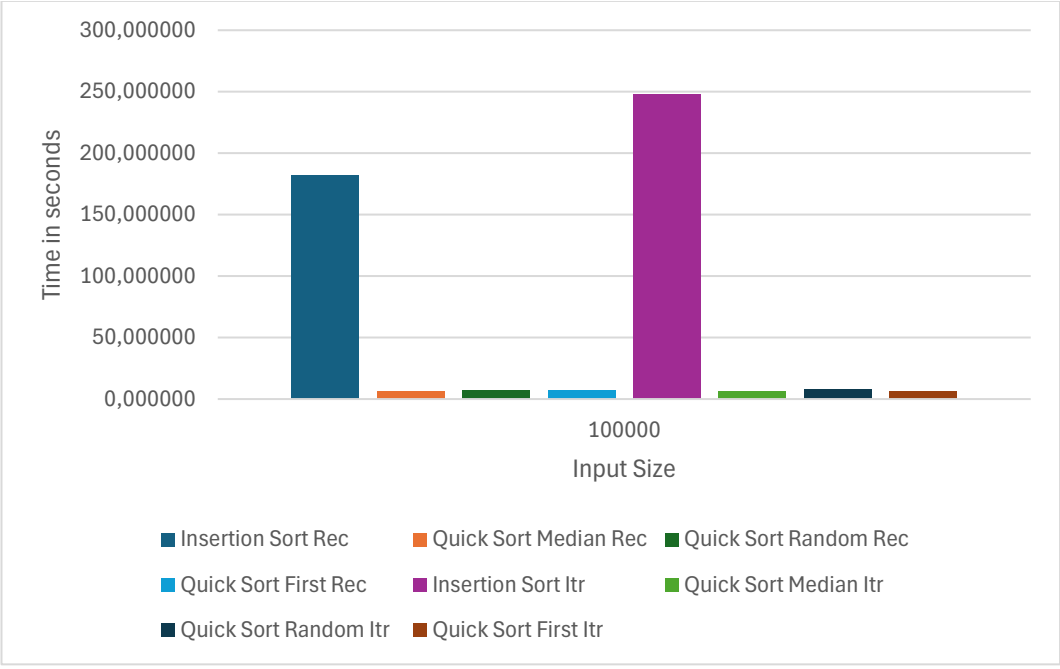
Results for Task 1 provides the reflection on sorting time for quick sort and insertion sort algorithm using recursive and iterative approaches. Also reflects the time measurement quick sort using different pivot strategy.

			Running Time in Seconds (Rounded up to 6 digits) for various input sizes				
Algorithm	**Pivot Strategy**	**Recursive(Rec)/Iterative(Itr)**	100	1000	10000	100000	1000000
Insertion Sort		Rec	0,000155	0,019174	2,464686	182,101877	
		Itr	0,000260	0,025152	2,897966	247,416214	
Quick Sort	Median of 3 pivot	Rec	0,000124	0,002071	0,123885	6,429296	757,316412
		Itr	0,000129	0,002765	0,103286	6,685808	856,84951
Quick Sort	Random pivot	Rec	0,000267	0,003556	0,114392	7,170452	613,119974
		Itr	0,000164	0,002668	0,095092	7,668866	748,413614
Quick Sort	first element pivot	Rec	0,000099	0,004101	0,105144	7,359506	687,479986
		Itr	0,000137	0,001798	0,097333	6,711055	880,047263

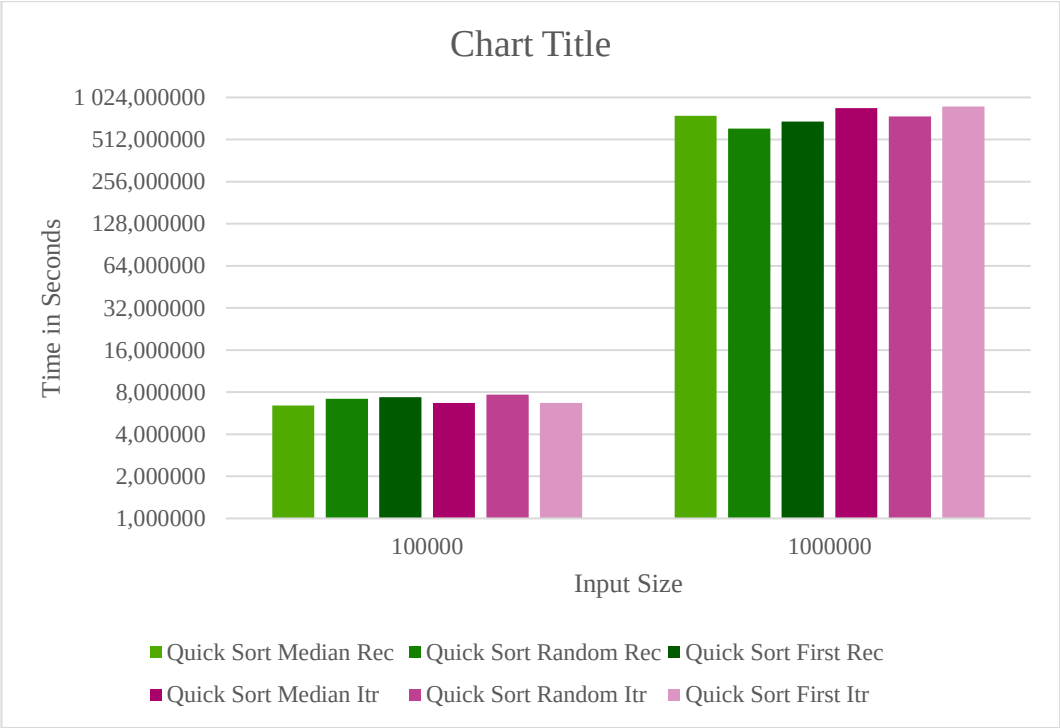
Table 1: Insertion Sort & Quick Sort Algorithm using recursive and iterative approach



Graph 1.1: Insertion sort & Quick sort Algorithm using recursive & iterative approach for input size <10001



Graph 1.2: Insertion sort & Quick sort Algorithm using recursive & iterative approach for input size 100000



Graph 1.3: Insertion sort & Quick sort Algorithm using recursive & iterative approach for input size 100 thousand & 1 million.

3.2 Task 1 Analysis:

Figure 1 below shows the time complexity for different algorithms:

Algorithm	Approach	Best Case	Average Case	Worst Case	Space Complexity
Insertion Sort	Iterative	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	Recursive	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$
Quick Sort	Iterative	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ to $O(n)$
Quick Sort	Recursive	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

Fig 1: Time complexity for Insertion sort & Quick sort Algorithm using recursive & iterative approach

As we can see from Fig.1 that theoretically the time complexity for algorithms doesn't change based on the recursive or iterative approach. So, we have run tests and measure time on various input sizes to analyze which approach is more efficient. Findings related to sorting algorithms are:

- Insertion sort performance as compared to quick sort for smaller arrays (n is less than 1001) is almost similar for both recursive and iterative approach as per results in Table 1. Results show that insertion sort recursive took less time as compared to insertion sort iterative. However, insertion sort iterative is better approach in terms of space complexity i.e. O (1). Insertion sort can be used for simplicity and to avoid stack overhead.

- Quick sort performs better as compared to insertion sort as per table 1, which proves that the time complexity for quick sort is O (n log n). As running time for insertion sort grows exponentially with data size. Graph 1.1 & 1.2 shows the time complexity for insertion sort grows exponentially up to $O(n^2)$.

- Quick sort performs best and maintains the time complexity to O (n log n) when median of three pivot strategy is used. So, partitioning should be balanced to avoid the worst time complexity for quick sort algorithm.

- Quick sort iterative approach takes less time as compared to recursive approach for input size less than 10001 for all pivot strategy and vice versa for input size greater than 10000 as per Table 1. So, it seems using iterative approach is good for small input sizes and recursive for large input sizes. However, the time difference is very little in seconds. Therefore, an iterative approach can be found beneficial even for large input sizes to avoid stack overflow.

- As per Table 1 & graph 1.2 & 1.3 quick sort is the clear winner for large datasets (input size greater than 100000). As for input size 1 million, we are unable to sort the data on Dell laptop using insertion sort algorithm.

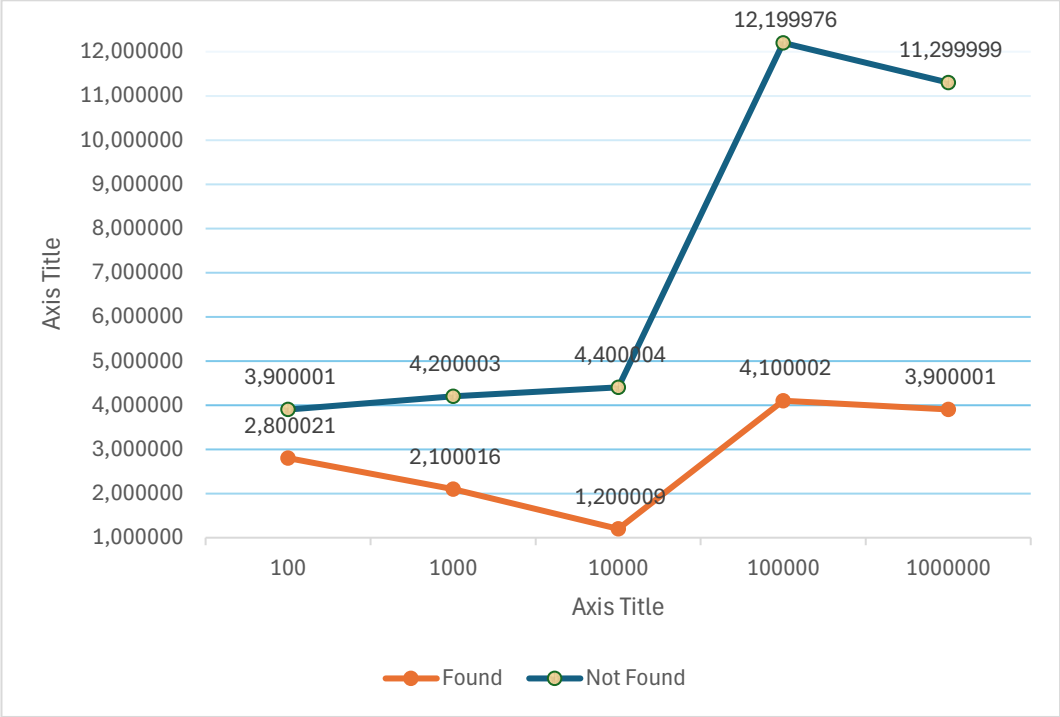
3.3 Task 2 Results:

Time complexity for Binary Search algorithm is O (log n). So, we have run tests and measure time on various input sizes. For every input size/array length binary

search has been used to find the middle element in sorted array and a value which does not exist in array. Results are shown in Table 2 and presented in Graph 2.

			Running Time in Microseconds (Rounded up to 6 digits) for various input sizes				
Algorithm	**Search Value**	**Found (True) / Not Found (False)**	100	1000	10000	100000	1000000
Binary Search	Array length/2	TRUE	2.800021	2.100016	1.200009	4.100002	3.900001
	Array length + 1	FALSE	3.900001	4.200003	4.400004	12.199976	11.299999

Table 2: Binary Search Algorithm using recursive approach



Graph 2: Binary Search Algorithm using recursive approach

3.4 Task 2 Analysis:

Result analysis for Task 2 shows that time taken by binary search recursive algorithm grows logarithmically. Doubling the size of input only adds a small constant to the total time for value found and not found. This proves the time complexity for Binary Search is O (log n). Even for massive input size, time remains manageable.

4 Conclusion:

Based on the testing of various sorting and searching algorithms using various input size, several key observations can be made:

- Task 1 – Quick sort exhibit better performance with a time complexity of **O (n log n)** in sorting large amount of data as compared to insertion sort which have **$O(n^2)$** time complexity.

- To maintain the time complexity of quick sort **O (n log n)**, it's important to choose pivot that divides the array into two roughly equal halves. As highly unbalanced partition can lead to worst time complexity i.e. **$O(n^2)$** for quick sort.

- Quick sort is preferred for large input size due to superior average case performance, while for small arrays (less than 1001) switching to simpler algorithms like Insertion sort is better for improved performance and avoiding quick sort overhead.

- Recursive approaches can be used for simplicity and clarity. However, an iterative approach is better to avoid stack overflow.

- Task 2 test proves that time complexity of Binary Search algorithm is O(log n). However, iterative approach can be used if want to avoid stack overflow.

So, while choosing an algorithm it's important to consider the size of data, type of data available, stack overhead and requirement of application. As selection algorithms can impact the performance of programs in terms of time complexity and space complexity. For real world scenarios, hybrid algorithms can provide best performance.

5 References :

[1] Weiss, Mark. A. (2012), Data structures and algorithm analysis in Java. 3rd edition Harlow, Essex: Pearson. (632 p).

[2] GeeksforGeeks. (n.d.). *GeeksforGeeks*. https://www.geeksforgeeks.org/