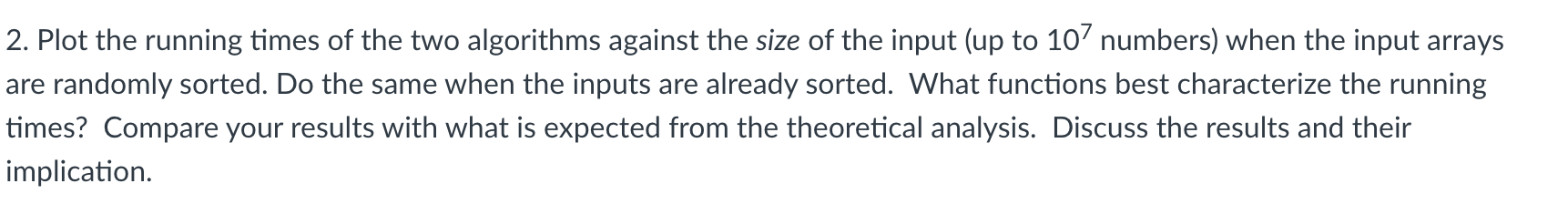
Assignment 2

CS 514 – Algorithms

Submitted By: Aman Pandita

Onid: [panditaa@oregonstate.edu](mailto:panditaa@oregonstate.edu)



***For Merge Sort vs Quick Sort when randomly sorted:***

A graph with a line and a line

Description automatically generated

*JUST AS AN EXTRA REPRESENTATION:*

A graph with a red line and green line

Description automatically generated

***For Merge Sort vs Quick Sort when already sorted:***

A graph with a line and a line

Description automatically generated

*JUST AS AN EXTRA REPRESENTATION:*

A graph with a red line and green line

Description automatically generated

***For Merge Sort vs Quick Sort when already sorted but the pivot was always selected as the first element (We only tested max of 104 input sizes):***

A graph with a green line and red dotted line

Description automatically generated

*JUST AS AN EXTRA REPRESENTATION:*

A graph with a green line and red dotted line

Description automatically generated

**Analyzing the results:**

**Merge Sort Expectation vs Experimental Data:**

|  |  |
| --- | --- |
| **Theoretical Expectation** | **Experimental Data** |
| *For Randomly Sorted Data:*  The time complexity of Merge Sort is expected to be **O(n log n)** for all cases of Best case, average case and worst Case. | The time required does not rise tenfold as the amount of the input does, but rather more than linearly. Given that it is greater than linear but less than quadratic, this fits with the n log n complexity. |
| *For already Sorted Data:*  As it has a O (n log n) complexity, it does not benefit from the already sorted data | When compared to the input that was sorted randomly, the time is a little bit faster, but not noticeably so. The passage of time continues to scale in a n log n manner. |

**Quick Sort Expectation vs Experimental Data:**

|  |  |
| --- | --- |
| **Theoretical Expectation** | **Experimental Data** |
| *For randomly Sorted Data:*  Quick Sort can degenerate to **O (n2)** in the worst situation but has an average-case time complexity of **O (n log n)**. However, we anticipate it to be close to **O (n log n)** for inputs that are sorted randomly. | Similar to Merge Sort, the length of time grows as the size does. This suggests that it is operating close to its **O (n log n)** average-case complexity. |
| *For already Sorted Data:*  As it has a **O (n log n)** complexity, it does not benefit from the already sorted data | The **worst-case** time complexity for previously sorted data would be **O (n2)** because the pivot selection and dividing strategy are not beneficial. |

**Complexity Analysis Quick sort:**

**Best case:** The best-case scenario for Quick Sort is when the pivot selected is the median, which causes the array to be split into two almost equal halves at each recursive call. Since the pivot is selected randomly, it’s possible to get the median, but not guaranteed.

**Derivation:**

* At each level of recursion, we are performing O(n) operations because we are iterating through each element once to split it among the three arrays: left\_arr, mid\_arr, and right\_arr.
* If we are lucky and always select the median as the pivot, the height of the recursive tree would be log(n)

Multiplying the work done at each level by the number of levels gives us:

T(n)=O(n)⋅O (log n)

T(n)=O (n log n)

**------------------------------------------------------------------------------------------------------------------------------------------Average Case:** The average case of Quick Sort, given that the pivot is chosen randomly, is generally quite good because, on average, the partitions will be reasonably balanced.

#### Derivation:

* Since the pivot is chosen randomly, the expected height of the recursive tree is O (log n)
* We are still doing O(n) work at each level for partitioning.
* We must also consider the probabilistic nature of pivot selection.

Summarizing this, we get an expected time complexity:

T(n)=O(n)⋅O (log n)

T(n)=O (n log n)

**------------------------------------------------------------------------------------------------------------------------------------------Worst Case:** The worst-case scenario for Quick Sort is when the smallest or largest element is always chosen as the pivot. However, because we're choosing the pivot at random, the chances of consistently choosing the smallest or largest element are low.

#### Derivation:

* If we unluckily always pick the smallest or largest element, it leads to skewed partitions, and the height of the recursive tree would be O(n).
* As always, partitioning takes O(n) time at each level.

Multiplying these together, we get:

T(n)=O(n)⋅O(n)

T(n)=O(n2)

|  |  |
| --- | --- |
| **Merge Sort** | **Quick Sort** |
| **Best case:** O (n log n)  **Average case:** O (n log n)  **Worst case:** O (n log n) | **Best case:** O (n log n)  **Average case:** O (n log n)  **Worst case:** O (n2) |

*Commentary of the analysis when inputs are randomly sorted:*

The fewer hidden constant variables make Quick Sort, on average, quicker. Its built-in partitioning also helps to conserve space.

In contrast to Quick Sort, Merge Sort has a constant time complexity of **O (n log n)** for all situations.   
Merge Sort would be preferred if stability was a priority.

Additionally, Quick Sort is more space-efficient because it takes up less room.

Furthermore, Quick Sort may better optimize efficiency by adjusting to different pivot selection techniques.

*Commentary of the analysis when inputs are already sorted:*

Merge Sort consistently performs well and retains its **O (n log n)** time complexity for inputs that have previously been sorted, making it a consistent and reliable option.

However, Quick Sort needs to be carefully considered. To prevent the **O(n2)** worst-case situation, the pivot selection mechanism must be improved. Quick Sort can also function effectively and attain **O (n log n)** time complexity with the appropriate optimization (e.g., picking a middle element or employing a median-of-three technique).

Without pivot optimization, Quick Sort runs the danger of reaching the worst-case time complexity for inputs that have already been sorted. Thus, unless the Quick Sort implementation is well optimized, Merge Sort could be a more dependable option in such situations.

Overall, Merge Sort is always going to be more stable and consistent no matter what the array orientation is. However, when Quick Sort is used, it performs comparatively well when the input is randomly sorted considering that the random selection of pivot is not randomly selected to the worst-case scenario that is making the time complexity O(n2)

A key finding was that when the input number is as high as 107, Quick sort used to reach maximum recursion depth, this was tackled by using a recursion depth limiter:

