

**T.C.**

**MARMARA UNIVERSITY**

**FACULTY of ENGINEERING**

**COMPUTER ENGINEERING DEPARTMENT**

CSE2246 Analysis of Algorithms

**PROJECT REPORT**

Title of the Project

***“Finding median of an unsorted list of n numbers. “***

April 21 2024

Group Members

Murat Tüzün 150119633

Aimen Daddi 150119879

Yusuf Kaya 150119059

Designing the Experiment

Part a) Deciding on reasonable inputs / generating reasonable sample inputs

We did our experiments on a total of 5 different txt files (total of 10 txt files) with 500,1.000,5.000,10.000,15.000 element numbers. In txt files starting with 'sh' (such as sh.500.txt), if the elements are from 1 to whatever number is written in the name of that file, for example, if the file name is sh.500.txt, this means that the numbers from 1 to 500 are shuffled in this txt file. For txt files with 'r', for example r500.txt, this means that we have written an ordered list from 1 to 500 in reverse order. We did our experiments with these two different file types, we did not prefer a list of random numbers because the probability of repeated numbers is very high and we thought that this would negatively affect the quality of the experiment we would do, instead we found it more meaningful to shuffle the list. The reason why we use reverse ordering is that a list has many (15,000) or few (500) elements and we chose reverse ordering to see how the algorithms we will use will give the results of the experiments we will do when these elements are written in reverse order.

Part b) Deciding on reasonable metrics (for complexity measurement).  
 In order to make a complexity measurement, we found it appropriate to measure the number of operations and the time taken to perform the operation (in seconds). When we count operations, we get a clear picture of how each algorithm manages its tasks and uses its resources, which is great for comparing their efficiency. This count also helps us look at the algorithm’s performance on a basic level that isn't tied to any specific type of computer or system, which means we can compare how different algorithms perform in a fair way, no matter where we run them. The time each algorithm takes to finish is also really useful because it shows us directly how long an algorithm will make us wait in real life Although the time can vary based on which computer used, so we used only one computer while doing these experiments.  
  
Expectations from the algorithms  
  
For the shuffled files, What we expected from the following algorithms:

**Insertion Sort:** This might struggle with larger files due to its O(n^2) complexity but should be okay for smaller ones.

**Merge Sort:** This should be consistently strong across all sizes because of its O(nlogn) efficiency.

**Quick Sort (first element as pivot):** Generally solid but might suffer if the first element isn't representative, especially as file size grows.

**Max-Heap:** This should be robust for selection tasks like finding the median, regardless of order, with good scalability.

**Quick Select (first element as pivot):** It might share vulnerabilities with quick sort if the first element is an extreme value.

**Quick Select (median-of-three pivot):** Should be more resilient than using the first element as pivot, good for shuffled data due to improved pivot selection.

**Quick Select (median-of-medians pivot):** Likely the best choice for shuffled data among the quick select variations, excellent for larger datasets.  
  
  
  
For the reversed files, What we expected from the following algorithms:

**Insertion Sort:** Performance will likely be poor as each element needs repositioning to the start, making it highly inefficient.

**Merge Sort:** Its divide-and-conquer method should mean that performance remains unaffected by the initial reverse order.

**Quick Sort (first element as pivot):** Likely to perform poorly as the pivot will create unbalanced partitions.

**Max-Heap:** The build and extract operations are unaffected by the initial order of elements, so performance should remain consistent.

**Quick Select (first element as pivot):** Could see significantly worse performance similar to quick sort due to poor pivot choice.

**Quick Select (median-of-three pivot):** Better than using the first element as pivot but might still be challenged if the median-of-three is not central.

**Quick Select (median-of-medians pivot):** This should perform optimally even in reversed datasets, thanks to its effective pivot selection strategy.

**Results for shuffled numbers**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **Insertion Sort** | **500** | 0.006 | 62179 |
| **Insertion Sort** | **1000** | 0.025 | 256071 |
| **Insertion Sort** | **5000** | 0.646 | 6207738 |
| **Insertion Sort** | **10000** | 2.659 | 24957562 |
| **Insertion Sort** | **15000** | 5.985 | 55770429 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **Merge Sort** | **500** | 0.001 | 500 |
| **Merge Sort** | **1000** | 0.002 | 1000 |
| **Merge Sort** | **5000** | 0.011 | 5000 |
| **Merge Sort** | **10000** | 0.025 | 10000 |
| **Merge Sort** | **15000** | 0.038 | 15000 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **Quick Sort** | **500** | 0.001 | 5955 |
| **Quick Sort** | **1000** | 0.002 | 14409 |
| **Quick Sort** | **5000** | 0.013 | 92004 |
| **Quick Sort** | **10000** | 0.027 | 190491 |
| **Quick Sort** | **15000** | 0.044 | 317142 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **Max-Heap** | **500** | 0.003 | 5893 |
| **Max-Heap** | **1000** | 0.003 | 12727 |
| **Max-Heap** | **5000** | 0.021 | 78808 |
| **Max-Heap** | **10000** | 0.042 | 170240 |
| **Max-Heap** | **15000** | 0.066 | 266273 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **Quick Select** | **500** | 0 | 1773 |
| **Quick Select** | **1000** | 0.0005 | 4495 |
| **Quick Select** | **5000** | 0.0039 | 38667 |
| **Quick Select** | **10000** | 0.0059 | 52658 |
| **Quick Select** | **15000** | 0.0119 | 113452 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **QS-MOT** | **500** | 0 | 1785 |
| **QS-MOT** | **1000** | 0 | 4784 |
| **QS-MOT** | **5000** | 0.002 | 15881 |
| **QS-MOT** | **10000** | 0.005 | 42077 |
| **QS-MOT** | **15000** | 0.009 | 78805 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **QS-MOM** | **500** | 0.0009 | 2686 |
| **QS-MOM** | **1000** | 0.0009 | 5111 |
| **QS-MOM** | **5000** | 0.0049 | 25920 |
| **QS-MOM** | **10000** | 0.0099 | 50583 |
| **QS-MOM** | **15000** | 0.139 | 75707 |

**Analysis :**

The Insertion Sort method showed quick results for the smallest file, aligning with the expected behavior due to its lower computational complexity at that scale. However, as we moved to larger files, its performance dipped dramatically, with both the time and number of operations required increasing significantly. This steep decline confirms the initial prediction about its inefficiency with large datasets, as it’s hindered by its quadratic time complexity, particularly with shuffled data.

Merge Sort proved its robustness and justified our initial predictions, delivering consistent and strong performance irrespective of the data size. Even as the file sizes increased, Merge Sort maintained a stable execution time, reinforcing its capability to handle large data sets efficiently thanks to its O(nlogn) time complexity.

Quick Sort displayed a remarkable efficiency for the smaller files, but this efficiency waned with larger files. While it still performed adequately, the uptick in execution time suggests that its performance is somewhat influenced by the dataset's size and complexity, even though it's typically known for its speed and O(nlogn) average time complexity.

Max Heap's execution times were impressive across various file sizes. The algorithm showed a slight increase in the number of operations for larger datasets, but the execution time remained minimal. This was in line with our expectations, demonstrating Max Heap's effectiveness and its capability to maintain performance as the dataset grows.  
  
When we looked at Quick Select, using the first element as a pivot turned out to be less efficient, especially for larger files. In contrast, the median-of-three and median-of-medians strategies displayed greater resilience, managing to keep both execution time and the number of operations low, even as file sizes increased. Notably, the median-of-medians approach excelled, displaying the best performance for the largest datasets.

In summary, the algorithms that used more sophisticated strategies to deal with shuffled data—specifically Merge Sort and Quick Select with the median-of-medians pivot—proved to be more efficient, particularly for larger files. Quick Sort and Max Heap also showed reliable performance, though they were slightly outpaced by the top performers. The results illustrate how pivotal the choice of algorithm is when dealing with shuffled data, especially as the scale of the data increases.

**Results for reversed numbers**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **Insertion Sort** | **500** | 0.011 | 125249 |
| **Insertion Sort** | **1000** | 0.057 | 500499 |
| **Insertion Sort** | **5000** | 1.307 | 12502499 |
| **Insertion Sort** | **10000** | 5.673 | 50004999 |
| **Insertion Sort** | **15000** | 11.564 | 112507499 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **Merge Sort** | **500** | 0.000 | 500 |
| **Merge Sort** | **1000** | 0.002 | 1000 |
| **Merge Sort** | **5000** | 0.009 | 5000 |
| **Merge Sort** | **10000** | 0.020 | 10000 |
| **Merge Sort** | **15000** | 0.033 | 15000 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **Quick Sort** | **500** | 0.014 | 125748 |
| **Quick Sort** | **1000** |  |  |
| **Quick Sort** | **5000** |  |  |
| **Quick Sort** | **10000** |  |  |
| **Quick Sort** | **15000** |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **Max-Heap** | **500** | 0.001 | 4373 |
| **Max-Heap** | **1000** | 0.002 | 9943 |
| **Max-Heap** | **5000** | 0.027 | 64310 |
| **Max-Heap** | **10000** | 0.051 | 140490 |
| **Max-Heap** | **15000** | 0.055 | 221950 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **Quick Select** | **500** | 0.025 | 187749 |
| **Quick Select** | **1000** |  |  |
| **Quick Select** | **5000** |  |  |
| **Quick Select** | **10000** |  |  |
| **Quick Select** | **15000** |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **QS-MOT** | **500** | 0.0 | 1504 |
| **QS-MOT** | **1000** | 0.0 | 3004 |
| **QS-MOT** | **5000** | 0.0009 | 15004 |
| **QS-MOT** | **10000** | 0.0029 | 30004 |
| **QS-MOT** | **15000** | 0.0049 | 45004 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Size** | **Time** | **Operations** |
| **QS-MOM** | **500** | 0.0009 | 2686 |
| **QS-MOM** | **1000** | 0.0009 | 5111 |
| **QS-MOM** | **5000** | 0.0049 | 25920 |
| **QS-MOM** | **10000** | 0.0099 | 50583 |
| **QS-MOM** | **15000** | 0.139 | 75707 |

**Analysis:**

Quick Sort and Quick Select Algorithms failed when we try to run them more than 500 element list, so their values are not written after r500.txt.

Upon reviewing the performance metrics for all the algorithms when applied to reversed .txt files, it appears that our expectations were met in some areas, while in others, the outcomes were surprising. Our initial expectation for algorithms such as Merge Sort and Max Heap was that they would perform consistently regardless of input order, which has been supported by the data.

For Insertion Sort, as predicted, we found that it performed poorly on larger files; the execution time and number of operations increased drastically with file size. This was expected due to its O(n^2 ) complexity, and it is evident that Insertion Sort is not suitable for large datasets with reversed ordering.

Merge Sort showed strong and consistent performance, with a relatively low and stable number of operations, and a predictable increase in execution time as file sizes grew. This aligns with the expectation that Merge Sort's O(nlogn) complexity makes it a robust choice for any file size.

The Quick Sort algorithm demonstrated significant performance issues with the reversed files. It failed to execute for lists larger than 500 elements. This was due to the choice of the first element as the pivot, which is the worst-case scenario for Quick Sort and leads to O(n^2) performance. Hence, the algorithm's failure was aligned with the expected worst-case scenario.

Max Heap's performance was also as expected, with the algorithm being fairly unaffected by the reverse order of the input. It scaled well with larger file sizes, though there was an increase in both execution time and operations count as file size increased.

For the Quick Select variations, the pivot strategy played a key role. The basic Quick Select, using the first element as the pivot, struggled with reversed files just as Quick Sort did. However, Quick Select with the median-of-three and median-of-medians (MOM) pivot strategies showed better resilience. Notably, the median-of-medians approach had a consistently low number of operations and did not fail even on larger files, demonstrating its effectiveness for finding the median in a reversed list.

To summarize, while Insertion Sort and Quick Sort (first element as pivot) failed to meet efficiency criteria for larger reversed datasets, Merge Sort and Quick Select with the median-of-medians pivot strategy outperformed others. Max Heap also showed a strong performance. The Quick Select median-of-medians pivot, in particular, stood out for its ability to handle reversed lists efficiently, confirming it as the superior algorithm among those tested for this specific task.

Metrics for all algorithms using shuffled txt files