Algorithm Final Project

Martha Akua Owusu Agyeman, 04072025

Kwame Frimpong Afriyie-Buabeng, 09752025

Tiffany Abui Degbotse, 74882025

March 26, 2024

Introduction

Sorting Algorithms are essential in the field of Computer Science, and Software Development. An example of such is the Quicksort Algorithm, that was introduced in 1961 by British computer scientist Tony Hoare. Since then, different variations of the Quicksort have been created, making it one of the fastest sorting algorthms, thus a popular choice in programming libraries. Unfortunately, unlike some sorting algorithms, the Quicksort is not adaptive- it does not adjust its performance based on the degree of how sorted the input is. The Quicksort algorithm has a best case of $\mathcal{O}(n \log n)$, which occurs when the pivot element chosen is the median element in the array. The worst case of $\mathcal{O}(n^2)$ occurs when the pivot element chosen is either the smallest, or largest element in the array, thus leading to imbalanced proportions , and poor performance.

In this paper, we will discuss how we can improve on the worst case behavior of Quicksort, which occurs when the array is almost sorted, already sorted, sorted in a reverse order, and many more. We propose a modified version to mitigate these shortcomings mentioned. However, when the array is nearly sorted, the way in which a pivot is selected could lead to a performance that is not optimal, thus resulting in the worst-case time complexity of $\mathcal{O}(n^2)$

To address the issues mentioned above, our modified Quicksort Algorithm introduces a new pivot selection strategy. Instead of always selecting the first or last element, we randomize, and select a pivot to increase our chances of balanced proportions. Furthermore, we incorporate a hybrid approach that works with the Tail Recursion Elimination, Median of three and the Randomized Pivot Selection. The Tail Recursion Elimination reduces the space complexity and improves the overall performance. It can be achieved either by using iteration, or converting the recursive calls into a loop. The Randomized Pivot Selection, as described earlier randomly selects a pivot position for comparison and partitioning. This helps avoid worst-case scenarios where the input is already sorted which can lead to poor partitioning.

Literature Review

A thorough desk study was carried out on the most classic sorting algorithms such as the Heapsort, Merge sort and Quicksort. From the various publishments covered, studies showed that most classic sorting algorithms are not adaptive(Brodal et al., 2004). Irrespective of the input, the time complexity is $\mathcal{O}(n \log n)$. For quicksort however, the worst-case time complexity, which occurs with an almost sorted or sorted input array, is $\mathcal{O}(n^2)$. Over the years there have been many variations of the quicksort algorithm to mitigate the worst-case behavior, some proven to be faster than the original.

A study on the Adaptiveness of Quicksort (Brodal et al., 2004) revealed that Quicksort exhibits adaptive behavior in practice and is dependent on the measure of presortedness, element swaps and pivot selection. Although none of the variations entirely eliminate the worst-case scenario of $\mathcal{O}(n^2)$ time complexity, the implementation of specific methods successfully reduce the time complexity if quicksort.

Research on the Worst-Case Efficient Sorting suggested the implementation of QuickMergeSort (Edelkamp et al, 2019). Further investigation into pivot selection methods, small subarray sorting, merging procedure optimization, in-place Mergesort implementations, and parallelization of QuickMergesort can improve its efficiency.

Other sorting algorithms like mergesort is used for cases where the array is nearly sorted. Adaptive pivot selection analyzes input data and selects the pivot based on its characteristics, resulting in an $\mathcal{O}(n \log n)$ time complexity. Injection sort for small subarrays uses insertion sort instead of quicksort, resulting in an $\mathcal{O}(n)$ time complexity (Estvill-Castro et al., 1992).

Alternatively, other approaches have been proposed. These include randomized pivot selection, which has an O(n log n) time complexity, median-of-three selection, which chooses the pivot as the median of three randomly selected elements, and three-way partitioning. A hybrid approach combines multiple strategies, such as randomized pivot selection and insertion sort for small subarrays, to handle different scenarios effectively. All these approaches aim to minimize the time complexity of quicksort.

Applications

QuickSort is used widely due to its efficiency. Some real-world applications of QuickSort include:

- In computer graphics, QuickSort is used for image rendering.
- In addition, it is used for data visualization.
- In numerical computations, QuickSort is used for matrix sorting.
- Sorting algorithms in programming languages and libraries often use QuickSort due to its efficiency and simplicity.
- Database systems use QuickSort to sort large datasets efficiently, improving query performance.
- QuickSort is used in search algorithms such as binary search, where sorted arrays are required for efficient searching.
- Many operating systems use QuickSort for tasks such as file system organization and memory management.
- Online platforms and e-commerce websites utilize QuickSort for sorting and organizing large volumes of data, such as product listings and search results.

Algorithm Description

Our approach to mitigating the worst case involved the implementation of two hybrid algorithms which were then tested. The first hybrid algorithm combines the Pivot randomization and Tail recursion approaches, reducing the time complexity for the worst case. The code efficiently sorts an integer array in ascending order by implementing the quicksort algorithm with tail recursion optimization. The method performs better on average across a variety of input conditions when random pivot selection is used.

The RandomTailRecursive method initiates the quicksort algorithm, taking an integer array as input and calling the quicksort method with initial parameters. The quicksort method is the core of the algorithm, taking three parameters: the array, the lower bound, and the upper bound of the array segment to be sorted. The partition method rearranges elements around a pivot element, randomly selecting a pivot index within the given range, and swapping elements smaller than or equal to the pivot.

Building on the efficiency of the first hybrid, a second algorithm which incorporates the Median of three approach is implemented This tail-recursive QuickSort algorithm uses pivot randomization and the median-of-three technique to improve performance and reduce worst-case scenarios. The algorithm randomly selects three indices (low, mid, high) from an array and shuffles them using the shuffle method to avoid worst-case behavior. The median index is chosen as the pivot index, aiming to select a pivot closer to the actual median of elements. The quicksort method uses this partitioning scheme in a tail-recursive manner, optimizing stack space usage while benefiting from pivot randomization and median-of-three pivot selection. This combination aims to achieve efficient and reliable sorting performance across various input scenarios.

Algorithm Psuedocode

Tail Recursion with Pivot Randomization Algorithm.

```
function QuicksortTailRecursive(arr)
    Quicksort(arr, 0, length(arr) - 1)
end function

function Quicksort(arr, low, high)
    while $low < high$ do
        $pivotIndex \leftarrow$ Partition(arr, low, high)
        if $pivotIndex - low < high - pivotIndex$ then
            Quicksort(arr, low, pivotIndex - 1)
            $low \leftarrow pivotIndex + 1$
        else
            Quicksort(arr, pivotIndex + 1, high)
            $high \leftarrow pivotIndex - 1$
        **end if**
    **end while**
**end function**

**function** PARTITION(arr, low, high)
    $pivotIndex \leftarrow \text{random}(low, high)$
    $pivot \leftarrow arr[pivotIndex]$
    SWAP(arr, pivotIndex, high)
    $i \leftarrow low - 1$
    **for** $j = low$ **to** $high - 1$ **do**
        **if** $arr[j] \leq pivot$ **then**
            $i \leftarrow i + 1$
            SWAP(arr, i, j)
        **end if**
    **end for**
    SWAP(arr, i + 1, high)
    **return** $i + 1$
**end function**

**function** SWAP(arr, i, j)
    $temp \leftarrow arr[i]$
    $arr[i] \leftarrow arr[j]$
    $arr[j] \leftarrow temp$
```

```
end function
```

Quicksort with Tail Recursion, Median of Three, and Pivot Randomization.

```
function QuicksortTailRecursive(arr)
    Quicksort(arr, 0, length(arr) - 1)
end function
function Quicksort(arr, low, high)
    while $low < high$ do
        $pivotIndex \leftarrow$ Partition(arr, low, high)
        if $pivotIndex - low < high - pivotIndex$ then
            Quicksort(arr, low, pivotIndex - 1)
            $low \leftarrow pivotIndex + 1$
        else
            Quicksort(arr, pivotIndex + 1, high)
            $high \leftarrow pivotIndex - 1$
        end if
    end while
end function
function Partition(arr, low, high)
    $indices \leftarrow [low, low + \frac{high-low}{2}, high]$
    Shuffle(indices)
    $pivotIndex \leftarrow indices[1]$
```

```
        SWAP(arr, pivotIndex, high)
        $pivot \leftarrow arr[high]$
        $i \leftarrow low - 1$
        for $j = low$ to $high - 1$ do
            if $arr[j] \leq pivot$ then
                $i \leftarrow i + 1$
                SWAP(arr, i, j)
            end if
        end for
        SWAP(arr, i + 1, high)
        return $i + 1$
    end function
    function SHUFFLE(arr)
        for $i = \text{length}(arr) - 1$ downto 1 do
            $j \leftarrow \text{random.nextInt}(i + 1)$
            SWAP(arr, i, j)
        end for
    end function
    function SWAP(arr, i, j)
        $temp \leftarrow arr[i]$
        $arr[i] \leftarrow arr[j]$
        $arr[j] \leftarrow temp$
    end function
```

Algorithm Implementation

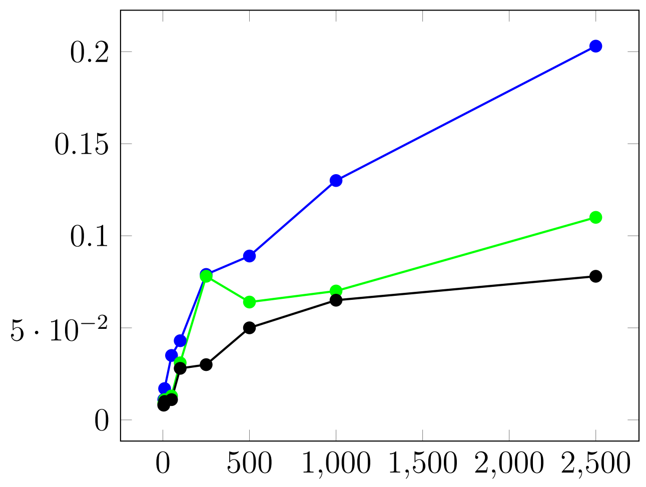
See .java files for Java implementation of the psuedocode above. A runtime counter is included to measure and print the running times of each algorithm alongside the successfully sorted array.

Algorithm Analysis

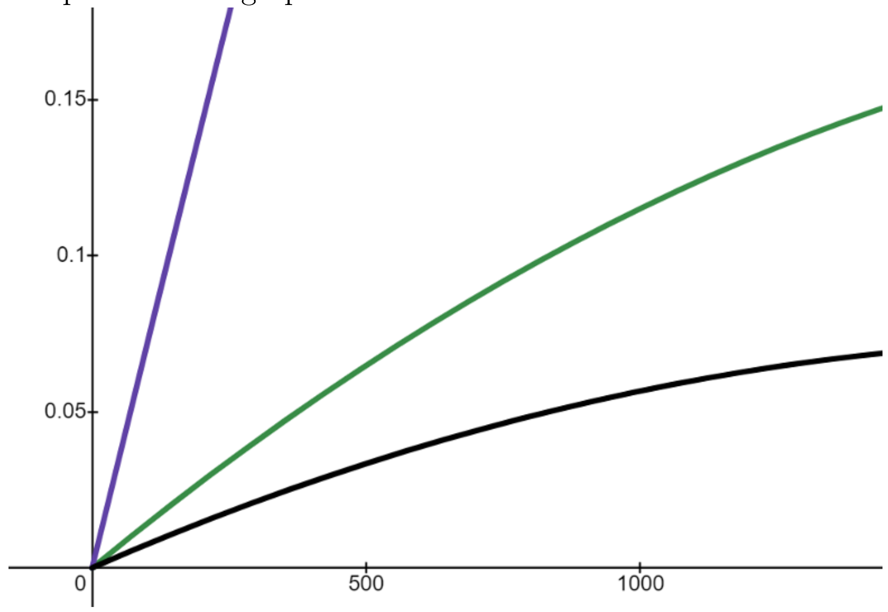
Array size N	**Quicksort**	**Adaptive 1**	**Adaptive 2**
10	0.017	0.011	0.010
50	0.035	0.013	0.011
100	0.043	0.031	0.028
250	0.079	0.078	0.030
500	0.080	0.064	0.050
1000	0.130	0.070	0.065
2500	0.203	0.110	0.078

Table 1: Table showing the runtimes of the algorithms using varying input sizes (in milliseconds).

Representation of running times



Above is a scattered plot of the various running times of the algorithms against the inputted array sizes. From here, we notice how the run time increases with the input size, but then the distinction lies in how as we improve the algorithm, the gradient keeps reducing as compared to the graphs above it.



Above is a graph of the respective time complexities of the Original QuickSort Algorithm(blue), The Recursive Tail Elimination and Pivot Randomization(green), and finally, median 3 hybrid with the Elimination and Pivot Randomization(black). From here we notice how the median 3 proves to be a more efficient algorithm, as it runs in less time than the other 2, as shown in the table above.

Given OC: our Algorithm and

ST: the standard algorithm,

Mean W-case-original= 0.085

Mean W-case-adaptive= 0.038857

$\frac{OC}{ST} = \frac{0.038857}{0.085}$

$= 0.46$

Given that the OC/ST¡1, we can assume that progress has been made in improving on the worst case of the algorithm

Discussion

Our project aimed to enhance the time complexity of the Quicksort algorithm for sorting sorted or nearly sorted list of elements. We began with the standard implementation and observed its performance as a baseline. Subsequently, we introduced tail recursion elimination optimization and randomizing pivot selection, which significantly improved efficiency. However, to further enhance performance, we implemented the ”median of three” pivot selection strategy combined with tail recursion elimination and randomization of the chosen elements.

Each optimization step led to a noticeable improvement in time complexity, as illustrated by the plotted graphs above. These visualizations depicted a clear downward trend in sorting time from the standard implementation to the final optimized version.Using the time complexities obtained from the original algorithm and our improved algorithm, we had OC/ST to be less than 1 which further proved that progress had been made in improving the time complexity of the worst-case scenario of the quicksort. The iterative process of algorithm optimization underscored the significance of thoughtful techniques in addressing performance bottlenecks and optimizing efficiency for real-world applications. Although Quicksort may not necessarily be adaptive, we have found a way to improve upon its efficiency in its worst-case scenario.

Conclusion

In summary, our project demonstrates the iterative optimization of the Quicksort algorithm, employing techniques such as tail recursion elimination and the "median of three" pivot selection. Through careful analysis and graphical representation, we show consistent improvements in time complexity. This underscores the significance of algorithmic design in enhancing efficiency, with potential applications across various domains.

References

Brodal, G. S. B., Fagerberg, R. F., & Moruz, G. M. (2004). On the adaptiveness of quicksort. *Basic Research in Computer Science.* https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=5825a3c309c6161c2b1b9c081021b58d0469d642

Edelkamp, S., & Weiß, A. (2019). Worst-case efficient sorting with quickmergesort. *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, 1–14.

Estivill-Castro, V., & Wood, D. (1992). A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, *24*, 441–476.

Iliopoulos, V. (2015). *The quicksort algorithm and related topics.* arXiv:1503.02504.

Khreisat, L. (2018). A survey of adaptive quicksort algorithms.

Quicksort—data structure and algorithm tutorials. (2014, January). GeeksforGeeks.