Analysis of Algorithms

BLG 335E

Project 1 Report

Kaan Karataş
karatask20@itu.edu.tr
150200081

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 26.11.2023

1. Implementation

1.1. Implementation of QuickSort with Different Pivoting Strategies

1.1.1. Introduction

Various approaches for the QuickSort algorithm, a sorting technique based on the divide-and-conquer principle is implemented. The algorithm recursively divides an array into two smaller segments by selecting a pivot element and arranging the elements so that those smaller than the pivot are on the left, and larger ones are on the right. The partitioning, handled in the ’partition’ method during each iteration, determines the pivot’s position. As the QuickSort algorithm is implemented in-place, the left segment spans from leftmost point to the pivot’s position - 1, and the right segment spans from pivot’s position + 1 to rightmost position. Following each partitioning, QuickSort recursively calls itself for these segments, adjusting their starting and ending points. Ultimately, the resultant data is a sorted array in ascending order based on the city populations.

1.1.2. Sumamry

The provided C++ code implements the quicksort algorithm for sorting a dataset of city populations by utilizing a custom CityData struct. The CityData struct represents cities with their names and populations. Firstly, the data from the CSV file is parsed and read by the ’read_csv’ method. After the parsing is complete, a vector of CityData structs is created. The ’swap_city_values’ function is used repeatedly and it is responsible for exchanging the values of two CityData instances used in Insertion Sort and Quick Sort methods.

The ’partition’ method is responsible for organizing elements in the dataset around the chosen pivot. The function ensures that elements smaller than the pivot reside on the left side, while larger ones occupy the right. The process begins by selecting the pivot index using the specified pivoting strategy. The chosen pivot is then relocated to the end of the current dataset. Subsequently, the method iterates through the dataset, moving elements smaller than or equal to the pivot to the left side of its final position. Finally, the pivot is accurately placed as a reference point for subsequent recursive calls within the Quicksort algorithm.

The ’choose_pivot’ method helps the partitioning process by determining the index of the pivot based on the chosen strategy. There are three different strategies: selecting the last element as the pivot (’l’), choosing a random element from the dataset (’r’), and opting for the average of three randomly selected elements (’m’). The last element strategy involves returning the index of the last element as the pivot, while the random element strategy generates a random index within the current dataset. The median of three strategy selects three random elements, calculates their average, and returns this average index as the pivot. And lastly, the ’quicksort’ function utilizes all afromentioned methods as the final algorithm.

The ’save_and_print_dataset’ function writes the sorted dataset to an output file, displaying relevant information about the sorting process, including the sorting duration with the respectvie pivoting strategy and the threshold. Lastly, the main function is where the entire sorting process happens by reading the data, handling command-line arguments, measuring sorting duration and every other method stated in the report.

1.1.3. Time and Space Complexity

The time complexity of the Quicksort algorithm is influenced by the choice of pivot strategy. In the worst case, when the pivot is consistently chosen poorly, the time complexity is $O(n^2)$, but on average, it is $O(n * logn)$. The space complexity is $O(logn)$ due to the recursive calls, and the algorithm is implemented in-place.

The time complexity of the quicksort algorithm is expressed by the recurrence relation:

$$T(n) = T(k) + T(n - k - 1) + O(n)$$

In contrast to merge sort, quicksort does not uniformly divide the array into two equal subarrays, as evident in the absence of the recurrence relation $2T(n/2)$. The non-uniform partitioning arises from the dynamic nature of quicksort, where the pivot selection and subsequent partitioning depend on the properties of the input data. Consequently, the recurrence relation reflects the variable nature of quicksort’s partitioning strategy and the resulting time complexity for sorting an array of size n.

1.1.4. Experiments With Different Pivoting Methods And Different Datasets

	Population1	**Population2**	**Population3**	**Population4**
Last Element	330 791 200	5 816 447 100	2 534 546 200	15 633 800
Random Element	10 839 300	8 704 200	11 017 000	12 257 600
Median of 3	11 317 200	7 251 200	11 450 400	12 407 000

Table 1.1: Comparison of different pivoting strategies on input data in nanoseconds.

The time taken for each sorting operation with various Quicksort pivoting strategies in nanoseconds(ns) is detailed in Table 1.1. The data reveals that opting for the last element as the pivot generally results in slower sorting compared to other methods, making it less reliable overall. However, it may exhibit improved performance in specific scenarios, such as when applied to the ’population4’ dataset. In this case, the alignment of the data order with a pattern allows the medians of the left and right segments to be appropriately positioned after partitioning. Nevertheless, beyond these specific conditions, choosing the last element tends to be consistently slower in most cases.

While the dataset ’population2’ is organized in ascending order, the dataset ’population3’ is arranged in a descending one. When the QuickSort algorithm selects the pivot as the last element, especially in datasets that are already sorted, the algorithm ends up with the worst-case scenario due to leaving one of the segments devoid of elements, concentrating the entire work on the other segment. Consequently, this situation leads to a time complexity of $O(n^2)$, emphasizing the inherent inefficiency of QuickSort under such circumstances. The algorithm, grappling with pre-sorted data, encounters a considerable performance setback, highlighting the significance of choosing a more strategic pivot for optimal sorting outcomes.

Utilizing a random pivot selection in QuickSort ensures that subarrays are reasonably balanced regardless of the input data, resulting in improved efficiency compared to the last element strategy. In this scenario, QuickSort achieves an average case time complexity of $O(n * logn)$. This is attributed to the fact that QuickSort undergoes logarithmic recursive calls, as the partitioning process remains balanced. At each partition, which incurs a linear time complexity due to comparing each element with the pivot, the overall average case complexity is calculated as $O(n * logn)$.

In terms of the best case scenario, the median-of-three strategy emerges as an enhancement over the random element strategy. This strategy involves selecting the median among three randomly chosen values as the pivot. Empirical measurements from Table 1.1 indicate that both strategies exhibit close performance.

The space required remains constant for local variables in each iteration of QuickSort, resulting in a space complexity of $O(1)$. However, since QuickSort undergoes logarithmic recursive calls for $logn$ times, the overall space complexity becomes $O(logn)$. Notably, global variables such as the chosen pivoting strategy and verbose, have been defined in main and the variables that do not necessitate repetitive creation in each iteration are handled accordingly to optimize efficiency. Importantly, these variables do not contribute to an increase in space complexity and are treated as $O(1)$.

1.2. Hybrid Implementation of Quicksort and Insertion Sort

1.2.1. Summary

In this second section of the homework, a distinctive hybrid approach has been employed, incorporating elements of both quicksort and insertion sort. This approach aims to balance the efficiency of quicksort and the simplicity of insertion sort. Quicksort, known for its effectiveness in larger datasets, is utilized for dividing the array into smaller segments, while insertion sort is applied when the size of the subarray, $(rigthmostpoint - leftmostpoint + 1)$ becomes sufficiently small, indicated by being smaller than the threshold inputted as a command line argument. This implementation capitalizes on the strengths of each algorithm, optimizing performance and adaptability to different scenarios.

1.2.2. Experiments With Different Threshold Values With Dataset Population4

Threshold (k)	0	10	20	50
Population4	12 555 200	12 662 700	13 714 800	17 897 300

Table 1.2: Comparison of different thresholds on the hybrid QuickSort algorithm on input data.

Threshold (k)	100	500	1000	20000
Population4	26 047 400	80 991 000	166 457 100	3 163 546 300

Table 1.3: Comparison of different thresholds on the hybrid QuickSort algorithm on input data.

The temporal efficiency of each sorting operation, measured in nanoseconds (ns), is presented in Tables 1.2 and 1.3, utilizing random pivoting as it often proves to be the most optimized strategy among the three. Notably, the observed measurements reveal an inverse relationship between sorting time and the threshold value. As the threshold surpasses 100, a substantial increase in sorting time becomes apparent. This phenomenon suggests that the threshold value significantly influences the efficiency of the sorting process, underscoring the need for a nuanced selection to achieve optimal performance.

1.2.3. Time and Space Complexity

The time complexity of the hybrid sorting approach, combining quicksort and insertion sort, is contingent on the chosen threshold value. In the case of random pivoting, the overall time complexity can be expressed by the recurrence relation:

$$T(n) = \begin{cases} T(k) + T(n-k-1) + O(n) & \text{if } n > \text{threshold} \\ O(n^2) & \text{if } n \leq \text{threshold} \end{cases}$$

Here, n represents the size of the array. When the array size exceeds the specified threshold, the hybrid approach uses quicksort with random pivoting, incurring a time complexity of $O(n * logn)$ in the average case. However, when the array size is equal to or less than the threshold, insertion sort is used, resulting in a time complexity of $O(n^2)$.

The observed inverse relationship between sorting time and the threshold value aligns with the theoretical understanding of the algorithm. As the threshold increases, the array is more likely to be sorted using quicksort, leading to improved average-case time complexity. Conversely, lower threshold values trigger more frequent use of insertion sort, resulting in a higher average-case time complexity.

The space complexity of the quicksort algorithm can be analyzed by considering the memory requirements at each level of recursion. At each recursive call, the algorithm uses a constant amount of space for local variables such as indices and pivot values. However, since quicksort is a recursive algorithm, the space complexity is influenced by the maximum depth of the recursion, which is logarithmic in the worst case.

In the worst case, when the algorithm makes unbalanced partitions at each step, the recursion tree can reach a height of log_2n where n is the number of elements in the array. Therefore, the overall space complexity of the quicksort algorithm is $O(logn)$.