Analysis of Algorithms

BLG 335E

Project 1 Report

Çağla Mıdıklı
150200011
midikli20@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering

Date of submission: 21.11.2023

1. Implementation

1.0.1. Structures

I defined a structure named ’data’ to hold two types of data obtained from reading a file together. This structure is holding cities and their populations. To accommodate these structure entities, I have defined an array. However, due to its uncertain length, I have designated it as a vector.

1.0.2. Swap

The swap function is used to exchange two elements in an array.

1.0.3. Printarray

This function is used to write the obtained result to the target file.

1.0.4. Printarraytxt

This function prints the currently active portion of the array and the selected pivot at each step of the QuickSort algorithm to the text file.

1.0.5. Partition

In the partition step, the last element is chosen as the pivot, and the other elements are rearranged such that those smaller than the pivot come before it, and the others come after it as the algorithm progresses from the beginning.

1.0.6. Option

In the option part, we determine the pivot for QuickSort based on information obtained from a comment line. Lastly, we replace the pivot with the last element of the array so that the partition process operates based on our newly identified pivot.

1.0.7. Insertionsort

When the number of elements in the array is less than k, the insertion sort is employed, sorting the elements from smallest to largest.

1.0.8. Quicksort

Quicksort divides itself recursively into sub-arrays according to three different pivot selection methods until the base case is reached.

1.0.9. Hybridsort

When the value of k in Hybridsort is greater than 1, it operates similarly to a regular quicksort up to the k value. When the number of elements in subarrays equals or less than k, it employs insertion sort to sort that specific subarray

1.1. Implementation of QuickSort with Different Pivoting Strategies

1.1.1. Implementation Details

Partition: This function takes an array arr and two indices left.index and right.index, then partitions the array based on the pivot element (chosen as the rightmost element). Elements smaller than the pivot are moved to the left, and larger elements are moved to the right.

Option: This function selects a pivot based on the option provided (opt). If the option is ’r’, it selects a random element as the pivot. If the option is ’m’, it selects the median of three randomly chosen elements. The selected pivot is then moved to the end of the array.

QuickSort: This is the main quicksort function. It takes an array arr, left and right indices, an option for pivot selection , a parameter k, and a boolean flag v. If k is 1, it calls the option function to choose the pivot. It also prints the array if the v flag is true. It then recursively applies quicksort to the left and right partitions.

1.1.2. Related Recurrence Relation

The recurrence relation is defined as T(n) = T(n - k + 1) + T(k)+ O(n) for this part, where k is associated with the magnitude of the number chosen as the pivot and its position in the final sorted sequence. If the array is split in half, the recurrence relation becomes T(n) = T(n/2) + O(n)(average and best case scenario). If I additionally consider the worst-case scenario where the array is consistently split in a ratio of n-1:1 during each function call, the recurrence relation becomes T(n)=T(n-1) + O(n)(worst case).

The sizes of the two newly created arrays after the split operation significantly impact the length of recursion steps. The more similar the sizes of these arrays, the shorter the recursion steps become. Therefore, when we choose the median as the pivot, our chances of finding the result optimally increase. ‘

1.1.3. Time and Space Complexity

O(n log n) is the average time complexity of the Quicksort algorithm. The partition ratio of the array, which is correlated with the number of recursive calls, determines the base of the logarithm. The problem size is reduced logarithmically with each division step, yielding an overall complexity of n log n. On the other hand, the complexity may reach O(n*n) in the worst-case situation (n-1:1). Thus, our odds of obtaining the best outcome rise when we select the median as the pivot.

The space complexity is related to the length of recursive calls in Quicksort. In the best case, when the array is evenly divided into two subarrays at each level, the space complexity is O(log n). However, in the worst case, where each level almost entirely divides one element from the other, if the length of the recursive calls approaches n, the space complexity becomes O(n). Shortly, this parts time complexity is O(nlog)n and space complexity is O(logn). Log(n) line which is space complexity is O(1).

	Population1	**Population2**	**Population3**	**Population4**
Last Element	305828990	5955532820	2383328206	10127404
Random Element	9814818	8054297	9411106	9675454
Median of 3	8928915	8585322	9488677	10588553

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

1.1.4. Results

According to the table, the element that requires the most time is the last element, and using random or median yields more optimal results. The fact that Population 2 is in increasing order and Population 3 is in decreasing order creates the worst-case scenario for QuickSort because it affects performance when the last element is chosen as the pivot, resulting in T(n) = T(n-1) + n. However, in cases where the pivot is randomly chosen or selected as the median, the algorithm is less affected by the initial order of the array. For example, the time complexity of Population 2 with the last element pivot strategy is significantly higher compared to the median and random pivot strategies, which exhibit lower and more comparable times. The mixed data in Population 4 yielded similar results for the three pivot selection strategies. The strategy of selecting the median of three random elements increases the likelihood of more balanced partitions in the array.

1.2. Hybrid Implementation of Quicksort and Insertion Sort

1.2.1. Implementation Details

HybridSort: The average time complexity of insertion sort is O(n*n). However, for small arrays, insertion sort can be more optimal. Therefore, in the hybridSort algorithm, when the number of elements in subarrays is less than or equal to k, insertion sort is utilized. The selection of the pivot is still achieved through the option function, and the array is partitioned into two subarrays in the array partition function. If verbose is active, it prints each step to the text file. If the difference between the last index and the starting index is less than or equal to k, the insertion sort function is invoked for that subarray.

1.2.2. Related Recurrence Relation

For insertion sort part relation is T(n) = T(n-1) + n. For quicksort part relation is T(n) = T(n - k + 1) + T(k)+ O(n) If the difference between the last index and the starting index is less than or equal to given value, the insertion sort function is invoked for that subarray and recurrence is ended. T(n) = T(n - k + 1) + T(k)+ O(n) if (n-k+1) > insertion-value k > insertion-value (insertion value is commandline input) T(n) = T(n-1) + n other situation

1.2.3. Time and Space Complexity

Insertion Sort's worst-case and average time complexity is O(n*n), whereas its best-case time complexity is O(n). Insertion Sort has an O(1) space complexity and doesn't require more memory. The larger the value of k, the less space is used in terms of space complexity. Hybrid algorithms average case time complexity is nlogn. Space complexity is log(n). Log(n) line which is space complexity is O(1).

Threshold (k)	1	10	20	50	100
Population4	10023734480	2449981978	1439959861	617932643	312222438

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

Threshold (k)	500	1000	20000	1000000
Population4	186493515	208857157	3239157234	3259144834

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

1.2.4. Results

The minimum time was measured when the value of k was equal to 500. In general, quicksort is often preferred for larger arrays because its time complexity is O(n log n), while the average time complexity of insertion sort is O(n*n). However, for small array groups, insertion sort may be faster up to a certain size. In this case, we tested this idea, and our algorithm worked most efficiently when k was 500. Beyond 500, the array became too large, causing the runtime to increase instead of decreasing. If we had conducted this experiment with data other than population 4, the results might have been different. This is because the data in population 4 is given in random order, and whether the data is in ascending or descending order can significantly affect the complexity, especially in insertion sort.