Analysis of Algorithms

BLG 335E

Project 1 Report

Abdullah Jafar Mansour Shamout
shamout21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 21.11.2023

1. Implementation

1.1. Implementation of QuickSort with Different Pivoting Strategies

1.1.1. Implementation Details

To Implement different pivoting strategies, at question 1.2 of the project, I wrote 7 functions. 3 quicksort functions that use different pivoting strategies. Each quicksort function calls on to its relevant partitioning function so that makes 6 total functions. Also I wrote a helpful swap function that I used to swap the elements withing the array. Since the Project question wasn't clear enough, I did two implementations. At question 1.2 I did the codes for each function separately as I understood at that time. Then when I reached the second part of the project where we needed to take input as arguments from the command line, I merged the functions into a single one where I check for the strategy within the function. Each code can be seen in its relevant file. For the 1.2 part of the project, I built on top of my naive sort algorithm. For the last_element_quicksort function, its the same as the naive algorithm since they use the same partitioning but I just called it differently here as requested. I did my code there according to the psuedocode taught in class. For the randomized_partition_quicksort, there I call on its relevant partitioning algorithm where using srand(time(NULL)); I use a seed to generate a random number that later I module by the range needed to get a valid index to pick a random valid pivot. After picking the pivot index, I switch it with the most right value and call on last_element_partition algorithm since from there it is the same and this is more efficient. For the median_3_partition algorithm, again I utilize a seed to generate 3 random numbers that I module to fit within my range. Afterwords, to find the median between the three numbers, I sort them using a simple bubble sort methodology, but since the values are only 3 and its fixed, its technically an O(1) complexity. After sorting them, I pick the value of the array at index 1 which will be the median. Then I swap it with the most right value and call on last_element_partition algorithm since from there it is the same and this is more efficient. As for the code that I wrote at part 2 and 3, there Instead of calling the function needed, I would have a single quicksort algorithm that has a switch block that checks for which partitioning to make. After making the partitioning, I do recursive calls back to the functions with the array but with its left and right parts till it is sorted fully. The general logic behind my quicksort algorithm is to pick an index, partition the array to it accordingly and swap its relevant value with the right most value. After doing that, starting from the left of the given array, I have two indexes that start from there, 1 of them increments till its at the right of the index, the other increments if the value at the other index is less than or equal to the pivot value, if it is then it swaps its value with the other pointer. After finishing the loop I return the pivot value to its correct index and return its index to recursively call the function till the index left is not < the

right index of the array.

1.1.2. Related Recurrence Relation

The recurrence relation for this algorithm considers three scenarios: the best case, worst case, and average case.

1.1.2.1. Last Element

Best Case: In the best-case scenario, each partition consistently splits the array in half, indicating that the pivot is the median. This results in the recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Here, $2T\left(\frac{n}{2}\right)$ represents two quicksort calls, each working on half the elements, and $O(n)$ denotes the cost of the partition function, which iterates over all elements.

Worst Case: For the worst-case scenario, each partition results in a split with $n-1$ elements on the left and 1 element on the right. This implies that the pivot is always the largest or smallest element, leading to the recurrence relation:

$$T(n) = T(n-1) + T(0) + O(n)$$

Here, $T(n-1)$ corresponds to the quicksort call on the $n-1$ elements of the array, $T(0)$ represents the direct return from the quicksort call on the right subarray with only one element, and $O(n)$ is the cost of the partition function.

Average Case: In the average case, the partition consistently results in a-to-b splits, where a is the fraction of the array in the left subarray, and b is the fraction in the right subarray. This fraction varies, such as 7-to-3 or 4-to-6, leading to the recurrence relation:

$$T(n) = T\left(\frac{4n}{10}\right) + T\left(\frac{6n}{10}\right) + O(n)$$

Here, $T\left(\frac{4n}{10}\right)$ corresponds to the quicksort call on the $\frac{4n}{10}$ elements of the left subarray, $T\left(\frac{6n}{10}\right)$ corresponds to the quicksort call on the $\frac{6n}{10}$ elements of the right subarray, and $O(n)$ represents the cost of the partition function.

1.1.2.2. random

Best Case:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Worst Case:

$$T(n) = T(n-1) + T(0) + O(n)$$

Average Case:

$$T(n) = T\left(\frac{6n}{10}\right) + T\left(\frac{4n}{10}\right) + O(n)$$

1.1.2.3. median of 3

Best Case:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Average Case and Worst Case:

$$T(n) = T\left(\frac{6n}{10}\right) + T\left(\frac{4n}{10}\right) + O(n)$$

1.1.3. Time and Space Complexity

for all of them its like below except for median of 3, the worst case there is nlogn making it also the average

Best Case:

$$T(n) = \Theta(n \log n)$$

Worst Case:

$$T(n) = \Theta(n^2)$$

Average Case:

$$T(n) = \Theta(n \log n)$$

Run your code with the configurations given in Table 1.1. Provide the execution time in nanoseconds (ns).

	Population1	**Population2**	**Population3**	**Population4**
Last Element	239720313 ns	5490568937 ns	2209000149 ns	19218411 ns
Random Element	183584756 ns	4571602696 ns	2388212138 ns	15915801 ns
Median of 3	180151525 ns	4492940365 ns	1420907266 ns	9896403 ns

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

1.1.4. Discussion

For the Last Element Pivoting Strategy:

- For `Population1.csv`, the data is almost sorted in descending order. Resulting in a worst-case time complexity of $T(n) = \Theta(n^2)$.

- In `Population2.csv`,data is already sorted in ascending order, the pivot is consistently the largest element. This causes a left subarray of size $n - 1$, making the worst-case time complexity $T(n) = \Theta(n^2)$.

- `Population3.csv`, similar to `Population1.csv` but fully sorted in descending order, guarantees that the pivot will be either the largest or smallest element, resulting in a worst-case scenario for the last element pivoting strategy.

- For `Population4.csv`, data is randomized and unsorted data, the pivot is random, leading to an average-case time complexity of $T(n) = \Theta(n \log n)$. `Population2.csv` takes the most time, as every pivot is the largest element. `Population1.csv` takes less time than `Population3.csv` due to its partial sorting, but both are faster than `Population2.csv`. Randomization in `Population4.csv` results in the least time consumption.

For the Random Element Pivoting Strategy:

For all four datasets, the randomization in choosing the pivot point makes it highly likely that the pivot is neither the largest nor the smallest element. This leads to fractional splits and an average-case time complexity of $T(n) = \Theta(n \log n)$. Therefore, for `Population1`, `Population2`, and `Population3`, the time is much less than that of the last element strategy. However, for `Population4`, since the data was already in random order, the time is almost the same.

For the Median of 3 Strategy:

In all four datasets, choosing three random elements and taking their mean equalizes the worst-case timing complexity with the average case and increases the chances of best-case behavior. This strategy is a bit faster than the random element strategy and is the fastest among all three strategies.

1.2. Hybrid Implementation of Quicksort and Insertion Sort

1.2.1. Implementation Details

For implementing this I only added an extra condition K that checks whether right - left + 1 <= K. if it is then we continue with insertion sort. if its not I continue with the sent strategy. I do this check at the start to make sure it is given priority. But before it I also check for if K = 1 to do naive sort

1.2.2. Related Recurrence Relation

Best Case: In the best-case scenario, our strategy involves dividing the elements in a way that consistently minimizes the work. This leads to the recurrence relation $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$.

Worst Case: The worst-case scenario, though still present, is less likely to occur due to our strategy utilizing insertion sort and randomization. The recurrence relation for the worst case is $T(n) = T(n-1) + T(0) + O(n)$.

Average Case: Our strategy is designed to encounter the average case more frequently. In this scenario, the recurrence relation is $T(n) = T\left(\frac{6n}{10}\right) + T\left(\frac{4n}{10}\right) + O(n)$.

1.2.3. Time and Space Complexity

Space Complexity:

$$O(n)$$ (No auxiliary space used)

Worst Case Time Complexity:

$$T(n) = \Theta(n^2)$$ (When $K \geq n$, only one call to insertion sort)

Average Case Time Complexity:

$$T(n) = \Theta(n \log(n/k))$$ (Recursion tree with cost at each level being n)

Best Case Time Complexity:

$$T(n) = \Theta(n \log(n/k))$$ (Recursion tree with cost at each level being n)

Run your code with different values of k. Record the execution time in nanoseconds (ns) on Table 1.3. Provide the execution time in nanoseconds (ns).

Threshold (k)	5	10	25	50	75
Population4	13373600 ns	12131699 ns	10416700	11439300	12221097

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

Threshold (k)	100	250	500	1000	5000
Population4	14778300	18689501	27913300	48125003	187987239

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

1.2.4. Discussion

I used random sorting for this part eventhough it doesnt affect it much since population 4 is already randomized. We can see a pattern that we find a minimum time between 25 and 50 such that the data sorting time steadily decreases from K=5 till K=25 and steadily increases from there onwards. This shows that insertion sort hybridisim is effective on low sized subarrays instead of going to the end with quicksort.