Analysis of Algorithms

BLG 335E

Project 1 Report

Erblina Nivokazi
nivokazi21@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

Explain your code by providing:

1.1.1. Implementation Details

To start by the main function, in main the arguments are taken from command line, and to ensure that there are 6 arguments, or 5 depending on the verbose condition, I implemented a condition for it, for the code to return an error so that the user provides the correct arguments. An std::stream logFile is created in main to initialize the logFile that will be used if the user enters verbose as an input. Then I took all these arguments and stored them in their respective variable containers. To add I also implemented some functions that read and write the data from csv.

For the **naive Quick Sort** I first started by implementing naive partition and then Quick Sort, these were fairly easy since it was direct implementation of the pseudocodes we've learned in class. For **the second part** I had to start modifying the partition function, and I started writing a new Quick Sort function for the pivoting strategies I had to use. I created two new functions, one for creating a random index and another for creating an index from the median of three, using these functions I determined which strategy would be used by looking at the condition, whether the strategy parameter provided was r, l or m.

The **random pivoting function**, $strategy = r$, chooses an index for the pivot based of random values between 0 and the number that the length of the array or list of numbers contains. The **median of three** function, $strategy = m$, that I implemented generates three random numbers with he same idea as the random function but then it chooses in between them the median of them. As per the last element pivot strategy I did not implement any function for it as I can just assign the index of it from taking the last index as parameter.

The quicksort function that takes different pivoting strategies take as parameter std::vector<int> &a which is a reference to a vector that would contain the populations data values, int first which contains the index of the first element of the array or subarray being sorted, int last would contain the last index of the array or subarray being sorted, char strategy contains the strategy that will be used in the selection of pivot, and std::fstream &logFile which is a parameter that I later implemented into all my quicksorts which serves to log each call and operation that is being called in the program.

The logFile is implemented in the main function and it is declared to be open when there is a verbose input provided. For the logFile I implemented it in the partition function since it is called by all the Quick Sort functions that I have in my program, thus whenever

partition is called the program writes into the log file. To achieve this log.txt I had to use logFile as a parameter in naiveQuickSort, quickSort and hybridQuickSort, and of course it's a parameter in partition as well where I check whether the log file is open to use (it is open if there a v verbose condition provided in the command line arguments), and if it is indeed open then it writes all the arrays and subarrays in it.

1.1.2. Related Recurrence Relation

1.1.2.1. The normal Quick Sort with last element as pivot

For the first implementation of the naive quicksort the recurrence relations can be divided in three case: best, average and worst.

In the **best case** the array gets divided into two equal subarrays and we get the recurrence relation of merge sort which is $T(n) = 2T(n/2) + O(n)$.

The **average case** is when the array is divided into non equal subarrays and the recurrence relation might resemble something like $T(n) = T(6n/10) + T(4n/10) + O(n)$, this is not bad it will still give a running time of O(nlogn) and it is the usual case with Quick Sort, and it is why it's still preferred over the other sorts since even in it's average case it performs in nlogn. So even if the recurrence relation is $T(n) = T(9999n/10000) + T(n/10000) + O(n)$, the time complexity will still be $O(nlogn)$.

The worst case happens when the array is already sorted, in our since we're always sorting in ascending order the worst case would occur when the array or list of numbers provided to us is already sorted in ascending order and our pivot is in either end of the list, resulting on either only i being incremented or j, thus giving a time complexity of O(n^2). The worst case recurrence relation would be $T(n) = T(n-1) + T(0) + O(n)$.

1.1.2.2. Quick Sort with Random Element as Pivot Strategy

In this case similar to the naive Quick Sort, or the Quick Sort with last element as pivot, there are three cases as well, the cases being, best case, average case and worst case. What changes in this implementation of Quick Sort is that the probability for the worst case to occur is minimized, what this means is that for the worst case to happen the random element would have to be the last or first element in an array that is already sorted in ascending order.

So as for the last element as pivot strategy, the Random Element as Pivot strategy has a worst case of $T(n) = T(n-1) + T(0) + O(n)$, but with much lesser chances of happening. The average case is something like $T(n) = T(6n/10) + T(4n/10) + O(n)$ (or something mixed like this), which is the most likely case. Lastly the best case is again $T(n) = 2T(n/2) + O(n)$, but it's not very likely to happen.

1.1.2.3. Quick Sort with Median of Three as Pivot Strategy

Since we are picking the middle element of three different values that we've randomly picked, now the worst case of $O(n^2)$ is considered to be eliminated, now the worst and average case are the same, $T(n) = T(6n/10) + T(4n/10) + O(n)$, and the best case again is $T(n) = 2T(n/2) + O(n)$.

1.1.3. Time and Space Complexity

1.1.3.1. Normal Quick Sort with last element as pivot

The time complexity for normal Quick Sort with last element as pivot is calculated with the recurrence relations of this strategy.

To start with the best case which is $T(n) = 2T(n/2) + O(n)$, calculating the time complexity from it could be done by using the master's theorem. From the relation we can see that $a = 2$, $b = 2$, and $k = 1$, according to the master's theorem if $\log_b(a)$ is equal to k then the time complexity is equal to $O(n^2 logn)$, so accordingly the time complexity for the best case is then equal to $O(nlogn)$

The average case that I picked for this Quick Sort is $T(n) = T(6n/10) + T(4n/10) + O(n)$, and to calculate it's time complexity a recurrence tree can be used, what the recurrence tree will show that even if the length of the tree in both sides is not the same the tree will still branch on two sides, thus avoiding the worst case of $O(n^2)$ and getting $O(nlogn)$ even if it's not as efficient as the best case, it's efficient enough.

The worst case is $T(n) = T(n-1) + T(0) + O(n)$ and to calculate its time complexity we have to use the iterating method:

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

$T(n) = T(n-2) + T(n-1) + 2O(n)$

$T(n) = T(n-3) + T(n-2) + T(n-1) + 3O(n)$

$T(n) = T(n-4) + T(n-3) + T(n-2) + T(n-1) + 4O(n)$

...

$T(n) = T(n-k) + T(n-k-1) + T(n-k-2) + ... + kO(n)$

Taking $kO(n)$ we can see that n is multiplied by the value of k, which value when used $n - k = 0$, so that we can obtain $T(0)$, gives us k = n.

So the time complexity in the worst case is $O(n^2)$

The space complexity for quick sort is O(n) because the sorting is done without using any other space except for the input that is already provided

1.1.3.2. Quick Sort with Random Element as Pivot Strategy

Same with the normal implementation of Quick Sort the time and space complexity here are the same values.

The time complexity in the **best case** is again $O(nlogn)$, in the **average case** it is again

$O(nlogn)$, as for the **worst case** similar like before $O(n^2)$ except that it's occurrence is now less likely to happen.
As for the space complexity again no additional space from the array of data provided is being used so it is again $O(n)$.

1.1.3.3. Quick Sort with Median of Three as Pivot Strategy

For Median of Three as Pivot Strategy the time complexity can only be $O(nlogn)$ since here the worst case of $O(n^2)$ is eliminated. So the **best case** is $O(nlogn)$, and the **average case** and **worst case** are $O(nlogn)$ but the power of n might change, it's not as efficient as the best case.
The space complexity is still $O(n)$

	Population1	**Population2**	**Population3**	**Population4**
Last Element	73918400 ns	1290497000 ns	641700300 ns	3188000 ns
Random Element	3071300 ns	2521300 ns	2376900 ns	3585600 ns
Median of 3	2697200 ns	3572700 ns	3228500 ns	3570700 ns

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

It's noticeable that the Last Element pivoting strategy is one of the worse pivoting strategies since it takes always much more time than the others to complete. As for the Random Element and Median of Three, their time complexities are sort of similar but their difference is still noticeable in these files, for most of the files Random Element pivoting strategy showed to be faster but it was not the case for all of them.

In **population1** the most time is taken by the Last Element strategy, then Random Element strategy, and the fastest sorting happened with the pivoting strategy of Median of 3.

In **population2** the most time is taken by the Last Element strategy, then Median of 3 strategy, and the fastest sorting happened with the pivoting strategy of Random Element.

In **population3** the most time is taken by the Last Element strategy, then Median of 3 strategy, and the fastest sorting happened with the pivoting strategy of Random Element.

In **population1** the most time is taken by the Random Element strategy, then Median of 3 strategy, and the fastest sorting happened with the pivoting strategy of Last Element. Here the data is already random and the selection of pivoting strategy will always result in almost the same time complexity.

Population2 is the slowest since each pivot is to be the largest element in the array. Population1 is much faster than population3, as it's not completely sorted in descending order, but both are faster than population2. The randomization in population4 results in an average time complexity, making it the fastest.

1.2. Hybrid Implementation of Quicksort and Insertion Sort

1.2.1. Implementation Details

Hybrid Quick Sort

As with normal Quick Sort, I started with the usual condition of checking whether the last element's index is bigger that the first element's index. Further I continued to make an if statement based on the value of k, or otherwise the threshold. Based on how many elements the array or subarray has, they get sorted with insertion sort if the value of elements is smaller than k, and if the value is greater than k then it is sorted with Quick Sort.

Insertion Sort

For the Insertion Sort part I implemented an insertion sort function by referring to the pseudocode provided to us during classes.

Quick Sort part

The Quick Sort part of this hybrid implementation is similar with the Quick Sort implemented that takes three different pivoting strategies, the onlye difference is that when recursively call Quick Sort again I call the hybrid function again again so that after some time when the array sizes get smaller, if their size gets smaller than k then they get sorted by Insertion Sort rather than Quick Sort.

1.2.2. Related Recurrence Relation

Te related recurrence relation in Hybrid Quick Sort is measured differently from the previous Quick Sort, since her the amount of splits that the array is getting depends on the value of k.

The **best case** for this implementation is when the split in the array happen at a good time where applying insertion sort proves to be more beneficial than Quick Sort, differing for the best case in Quick sort here the relation includes the splits too, thus getting $T(n) = 2T(n/2) + O(n) + kn$.

The **average case** for this implementation is when the splits into different fractions, that are non-equal. This would give a relation of $T(n) = T(6n/10) + T(4n/10) + O(n) + kn$.

As per the **worst case**, this would occur when the value of k is much bigger than the amount of inputs that we have, this would result in the program only using the insertion sort, and since the worst case to happen in Insertion Sort is $O(n^2)$, it would give a recurrence relation of $T(n) = T(n-1) + n$.

1.2.3. Time and Space Complexity

For the hybrid quick sort, the time complexity can be easily deducted by the related recurrence relations for each case.

For the **best case** the times complexity can be solved by using a recurrence tree,

from which we would get $T(n) = O(nlog(n/k)$.

For the **average case**, the same recurrence tree can be implemented and the complexity would result to be the same, of course not as efficient as the best case but in the same category, $O(nlog(n/k))$.

For the **worst case**, we can recall even from the insertion sort's worst case that it is going to be $O(n^2)$, but otherwise it can also be deducted from the previous related recurrence relation, $T(n) = T(n-1) + n$, through iteration method.

Threshold (k)	10	25	50	100	150
Population4	4010500 ns	4111700 ns	3493000 ns	3710000 ns	3457600 ns

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

Threshold (k)	200	500	1000	3000	6000
Population4	3558100 ns	3617500 ns	3674200 ns	3672300 ns	3813400 ns

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

Since the data in **population4** is already not sorted and is a very mixed array of vaues, so they are random, the time complexities do not change much. Even with insertion sort it does not reach the time complexity of $O(n^2)$, so for most inputs of k the time does not change in a drastic way it will sort in a fairly good time. But the time complexity does get worse when insertion sort is used instead of quick sort, but since the data provided in Population4 is not problematic it does not show to make a big difference here

Reference for Read and Write functions which were used for csv files:
`https://www.gormanalysis.com/blog/reading-and-writing-csv-files-with-cpp/`