

ISTANBUL TECHNICAL UNIVERSITY

BLG335E

**ANALYSIS OF ALGORITHMS I
HOMEWORK 1**

HOMEWORK NO : 1

150190915 : Yasmeeen Abdelghany

INSTRUCTOR: Hazım Kemal Ekenel

Fall 2022

Contents

1	INTRODUCTION	1
2	PART 2.a	1
2.1	How2Run	1
3	PART 2.b	1
3.1	How2Run	1
4	PART 2.c	2
5	PART 2.d	3
5.1	Worst-Case Analysis	3
5.2	Best-Case Analysis	4
5.3	Average-Case Analysis	5
6	PART 2.e	5
7	References	6

1 INTRODUCTION

2 PART 2.a

2.1 How2Run

Write the following commands to compile and run the source code of my quicksort algorithm implementation.

```
g++ -Wall -Werror QS1.cpp -o hw1
```

```
./hw1
```

After that, you'll be prompted to enter the size and elements of the array to be sorted.

Below is a picture of the command line prompt after running the program.

A screenshot of a terminal window with a dark background. The window title is 'test@blg335e: ~/hostvolume'. The terminal shows the following text: 'test@blg335e:~/hostvolume\$ g++ -Wall -Werror QS1.cpp -o hw1', 'test@blg335e:~/hostvolume\$./hw1', 'Enter array size: 5', 'Enter array elements: 2 5 1 0 3', 'Sorted Array: 0 1 2 3 5', 'Number of swaps: 6', 'Number of partitions: 3', and 'Time Elapsed in seconds: 1.117e-06'.

```
test@blg335e:~/hostvolume$ g++ -Wall -Werror QS1.cpp -o hw1
test@blg335e:~/hostvolume$ ./hw1
Enter array size: 5
Enter array elements: 2 5 1 0 3
Sorted Array: 0 1 2 3 5
Number of swaps: 6
Number of partitions: 3
Time Elapsed in seconds: 1.117e-06
```

Figure 1:

3 PART 2.b

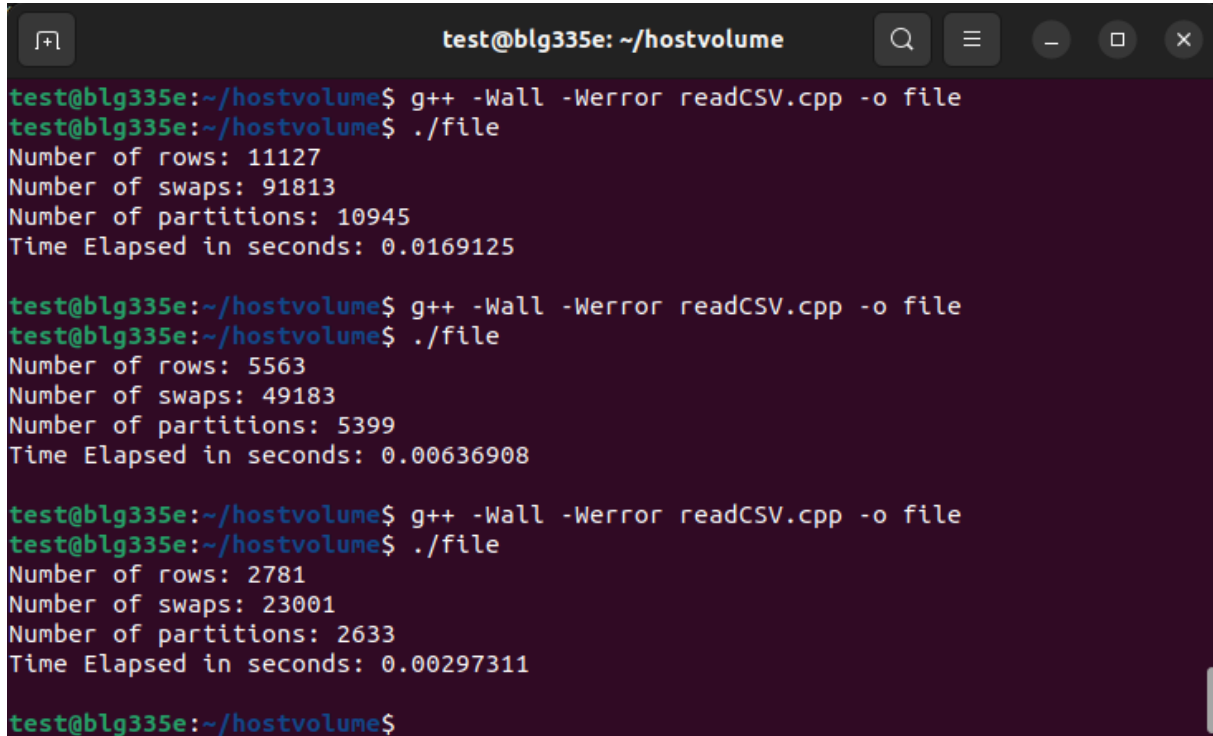
3.1 How2Run

Write the following commands to compile and run the source code of that reads data from a file and writes the sorted average ratings into another file. The file name is already written as a parameter within the source code.

```
g++ -Wall -Werror readCSV.cpp -o file
```

```
./file
```

After that, the sortedbooks.csv file will be created in the same directory you're in. Below is a picture of the command line after running the program. Note: to check out the statistics, please uncomment the relative code line in the source code.



```
test@blg335e: ~/hostvolume
test@blg335e:~/hostvolume$ g++ -Wall -Werror readCSV.cpp -o file
test@blg335e:~/hostvolume$ ./file
Number of rows: 11127
Number of swaps: 91813
Number of partitions: 10945
Time Elapsed in seconds: 0.0169125

test@blg335e:~/hostvolume$ g++ -Wall -Werror readCSV.cpp -o file
test@blg335e:~/hostvolume$ ./file
Number of rows: 5563
Number of swaps: 49183
Number of partitions: 5399
Time Elapsed in seconds: 0.00636908

test@blg335e:~/hostvolume$ g++ -Wall -Werror readCSV.cpp -o file
test@blg335e:~/hostvolume$ ./file
Number of rows: 2781
Number of swaps: 23001
Number of partitions: 2633
Time Elapsed in seconds: 0.00297311

test@blg335e:~/hostvolume$
```

Figure 2:

4 PART 2.c

As shown in Figure 2 of part 2.b, the first output shows the statistics after sorting the full data. The second and third outputs show the statistics after sorting half and a quarter of the data. The number of rows to be sorted can also be seen in the output.

When sorting only half of the data, the time taken is 10543 microseconds less.

When sorting only a quarter of the data, the time taken is 13939 microseconds less than the time taken to sort the whole data.

5 PART 2.d

Table 1: Asymptotic Upper, Lower and tight bounds

Case	Time Complexity
Worst Case	$O(n^2)$
Best Case	$O(n \cdot \log n)$
Average Case	$O(n \cdot \log n)$

```
41 void quickSort(int *Arr, int Low, int High)
42 {
43     if (Low < High)
44     {
45         int i = partition(Arr, Low, High);
46         partitions_count++;
47         quickSort(Arr, Low, i - 1);
48         quickSort(Arr, i + 1, High);
49     }
50 }
```

Figure 3:

The time complexity of partitioning for each recursive call on an n -element subarray is $O(n)$, for the function will always compare the n elements each time.

5.1 Worst-Case Analysis

In worst-case scenarios, the index i will always be 0 after each partition. For each quickSort function call, the right subarray will always have no elements while the left subarray will consist of all the elements from index 1 to High. Thus, the recurrence equation is $T(n) = O(n) + T(1) + T(n - 1)$. Using the recursive tree method to solve the recurrence equation, we can prove that the algorithm's upper bound is $O(n^2)$.

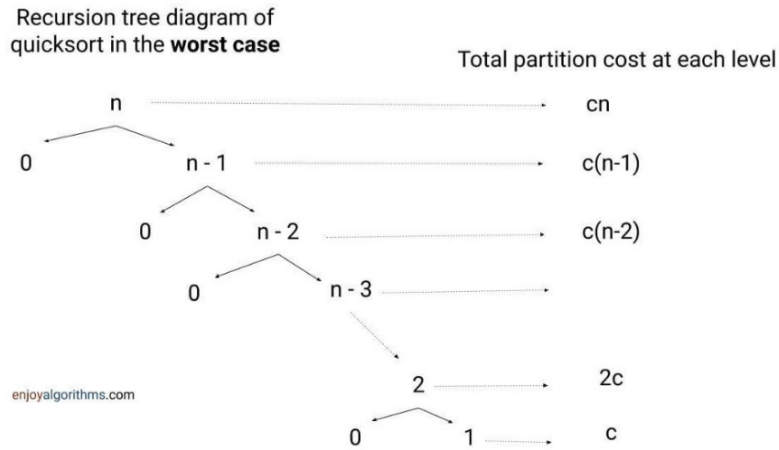


Figure 4: Recursive Tree Method

Summing up all the partition costs at each level we get: $cn + c(n-1) + c(n-2) + \dots + 2c + c$. Simplifying this result we get: $c(n-1 + n-2 + \dots + 2 + 1) = c((n*(n-1))/2)$. From this asymptotic notation, we can say that the worst-case time complexity for quickSort algorithm is indeed $O(n^2)$.

5.2 Best-Case Analysis

In best-case scenarios, the partition function will always pick the pivot index $i = n/2$. In other words, the partitions are always balanced; the left subarray will always consist of half of the elements of the complete array while the right one will consist of $n/2 - 1$ element. Thus the recurrence equation can be written as $T(n) = O(n) + T(n/2) + T(n/2 - 1) = O(n) + T(n/2) + T(n/2)$. Using the master method, $n^{\log_2 2} = F(n) = n^1$. Based on this theorem, the algorithm's tight bound is $\theta(n * \log_2 n)$. Also, using the recursive tree method to solve the recurrence equation, we can prove that the algorithm's tight bound is $\theta(n * \log_2 n)$.

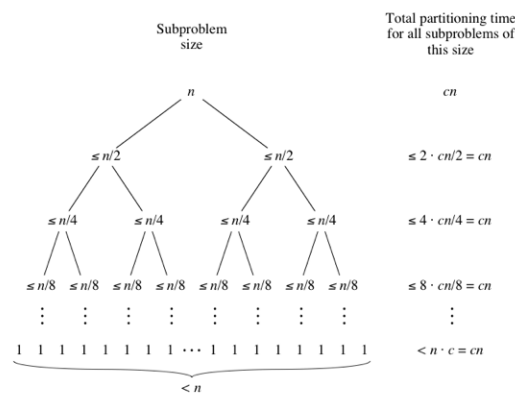


Figure 5: Recursive Tree Method

Summing up all the partition costs at each level we get: $cn + cn + \dots + cn$. Since the height of binary trees are $\log n$, we can say that the sum of cn from index $i = 1$ to $\log n$ is $cn * \log n$, which proves that the algorithm's tight bound is $\theta(n * \log n)$.

5.3 Average-Case Analysis

In average-case scenarios, however, partition functions generates a random pivot index each in recursive call. That is, throughout the recursive, if one side gets $x*n$ elements, the other side will get $(1-x)*n$ element. Again, the total partition cost at each level will sum up to cn . Just like in the proof of the best-case scenario, the algorithm's lower bound is $\Omega(n * \log n)$.

6 PART 2.e

An instance of the worst-case scenario is shown in the figure below. Worst-case scenarios happens when the array is sorted in the same or reversed order. If that's the case, the partition function will always pick the pivot index i to be the first element in the array, so one subarray will contain no element while the other subarray will contain $n - i$ elements. And since each partition recursive function compares all n elements, it ends up doing $n * O(n)$ comparisons, that is $O(n^2)$. The solution to this could be picking the pivot index at random or at the middle of the partition. This way worst-case scenarios are very unlikely to occur.

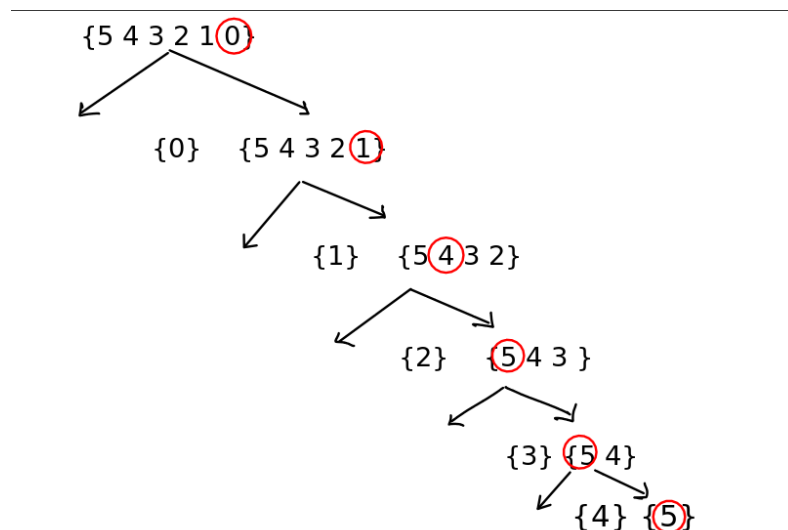


Figure 6: Schematic representation of worst-case scenario

7 References

- <https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort>
- <https://www.enjoyalgorithms.com/blog/quick-sort-algorithm>
- <https://www.geeksforgeeks.org/when-does-the-worst-case-of-quicksort-occur/>
- https://en.wikipedia.org/wiki/Quicksort#Worst-case_analysis
- <https://stackoverflow.com/questions/50199599/reduce-complexity-of-quicksort-method>