Programming Assignment 3

Gurpreet Singh-150259 . Nikita Awasthi-150453

March 2017

Question 1

The number of observations for each part are 1000.

For random arrays, we use a constant array [0, 1, 2 ... n-1] (for $length = n$) and randomly shuffle it using the function $random\_shuffle$ (in C++ algorithms library)

Part A

	$n = 10^2$	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$
Average running time of QuickSort	5.76μs	82.432μs	1.086431ms	13.79606ms	163.201154ms
Average running time of MergeSort	12.3μs	178.59μs	1.99737ms	24.15282ms	293.733225ms
Average number of comparisons in QuickSort	644.946	11002.594	155504.578	2017616.713	2523452.18
Average number of comparisons in MergeSort	542.25	8708.132	120452.184	1536355.955	2114345.23
No of times MergeSort had lesser no of comparisons than QuickSort	992	1000	1000	1000	1000

Table 1: MergeSort and QuickSort Analysis

The data given in the table clearly shows that on average QuickSort gives better time complexity as compared to MergeSort. The number of comparisons in Mergesort are almost always less than the number in Quicksort.

Despite of the fewer number of comparisons, MergeSort gives worse time because of the extra merging of half arrays to complete the sorted array. Though the QuickSort algorithm has worst case time complexity of $O(n^2)$, it essentially shows an average time complexity of $O(nlogn)$.

Part B

Table 2: Quicksort Analysis

	$n = 10^2$	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$
Average running time of QuickSort	5.76 micro s	0.082432 ms	1.086431 ms	0.01379606 s	0.163201154 s
Average number of comparisons in QuickSort	644.946	11002.594	155504.578	2017616.713	2523452.18
Percentage of cases when running time of QuicSsort exceeds average by 5%	11.9	20.4	13.5	10.6	3.2
Percentage of cases when running time of QuickSort exceeds average by 10%	0	17.0	6.8	2.7	0.4
Percentage of cases when running time of QuickSort exceeds average by 20%	0	6.4	4.1	0.1	0.1
Percentage of cases when running time of QuickSort exceeds average by 30%	0	4.1	3.2	0	0
Percentage of cases when running time of QuickSort exceeds average by 40%	0	2.0	2.0	0	0
Percentage of cases when running time of QuickSort exceeds average by 50%	0	1.3	1.2	0	0
Percentage of cases when running time of QuickSort exceeds average by 100%	0	0	0	0	0

The table shows that the average case running time for QuickSort is $O(nlogn)$ for all practical purposes. The data in the table shows that for most cases the running time remains close to the average value particularly when the number of array elements is large. For a small number of elements, generalisations may not be true always. However, the data for larger values clearly shows that except for a few worst case, QuickSort has a run time of $O(nlogn)$.

Note: We have considered count of cases where the running time is greater than 5%, 10% and so on

Question 2

Part A

The Op operation can be seen to be performed with a time complexity of $O(nlog_3 n)$. This can be seen through the fact that the array is divided into three parts and the function is recursively called for each part. The time complexity for the Op operation can be calculated using the relation:

Each recurssion call with length of array as len will be $T(len) = len + 3T(len/3)$. The second term is obtained due to the successive three recursive calls of the function Op with length of array as $len/3$ (approximately for large len). The first term, whereas is obtained due to the four loops that are used in copying the data from the main array to the temp array and vice versa.

Since we are using one extra (temporary) array of the same size (n), we can say we are using $O(n)$ auxiliary space for the Op operation.

Time Analysis:

$$\begin{aligned} T(n) &= n + 3T(n/3) \\ &= n + 3 * (n/3 + 3 * (n/9 + \ldots)) \\ &= \underbrace{n + n + \ldots + n}_{log_3 n \text{ times}} \\ &= nlog_3 n \end{aligned} \tag{1}$$

To perform the query operation, we do some preprocessing and use $O(n)$ time and $O(n)$ auxiliary space for an array sum whose ith element stores the sum of all the elements up to the ith index in the transformed array. This essentially reduces the time to compute a query to $O(1)$ time complexity which then only requires us to look up the values of the i and j indices provided in each input.

Execution time for Op: $O(nlog_3 n)$
Preprocessing time for queries : $O(log_3 n)$

Extra space: $O(n)$ for the temporary array and the sum array
Query time : $O(1)$

Part B

This part uses a concept similar to that of a (virtual) three child segment tree, however we do not store the sums in a node, however we still try to compute the sum in a range in $O(log_3 n)$ time. This is achieved by using recursion method like in a segment tree, and compute the sums for each range in $O(1)$ using logic of AP (since the actual elements are from 0 to $n-1$).

When using a three child segment tree, we say that at level, at most 9 nodes will be accessed, and hence for r layers, time will be at most $9rt$ where t_0 is the time taken at one node. Since in our case, $t_0 = O(1)$, and $r = O(log_3 n)$. Therefore, we say each query operation takes $O(log_3 n)$

Extra space: $O(1)$
Query time: $O(log_3 n)$
Total time: $O(qlog_3 n)$