CS 202 - 01

Homework 1

Algorithm Efficiency and Sorting

Dilay Yigit

21602059

Question 1:

Part a:

- T(n) = 3T(n/3)+n, where T(1) = 1 and n is an exact power of 3.

T(n) = 3T(n/3) + n

= 3(3T(n/9) + n/3) + n

= 3[3(3T(n/27) + n/9) + n/3] + n

…

= $3^k T(\frac{n}{3^k}) + \sum_{i=0}^{k} 3^i \frac{n}{3^i}$

…

= $3^{(\log_3 n)} T(\frac{n}{3^{(\log_3 n)}}) + \sum_{i=0}^{k} n$

= $nT(1) + (n * \log_3 n)$

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

- T(n) = 3T(n/2)+1, where T(1) = 1 and n is an exact power of 2.

T(n) = 3T(n/2) + 1

= 3(3T(n/4) + 1) + 1

= 3[3(3T(n/8) + 1) + 1] + 1

…

= $3^k (T(\frac{n}{2^k}) + \sum_{i=0}^{k-1} 3^i)$

…

= $3^{\log_2 n} (T(\frac{n}{2^{\log_2 n}}) + \sum_{i=0}^{\log_2 n - 1} 3^i)$

= $3^{\log_2 n} (T(1) + 3^{\log_2 n} - 1)$

= $2 * n^{\log_2 3}$

= $\Theta(n^{\log_2 3})$

Part b:

- Tracing array [5, 6, 8, 4, 10, 2, 9, 1, 3, 7] by using bubble sort algorithm.

(*) means "is swapped".

- Pass 1

[5, 6, 8, 4, 10, 2, 9, 1, 3, 7] (Unsorted original array)

[**5, 6**, 8, 4, 10, 2, 9, 1, 3, 7]

[5, **6, 8**, 4, 10, 2, 9, 1, 3, 7]

[5, 6, **4, 8,** 10, 2, 9, 1, 3, 7] (*)

[5, 6, 4, **8, 10,** 2, 9, 1, 3, 7]

[5, 6, 4, 8, **2, 10,** 9, 1, 3, 7] (*)

[5, 6, 4, 8, 2, **9, 10,** 1, 3, 7] (*)

[5, 6, 4, 8, 2, 9, **1, 10,** 3, 7] (*)

[5, 6, 4, 8, 2, 9, 1, **3, 10,** 7] (*)

[5, 6, 4, 8, 2, 9, 1, 3, **7,|10**] (*)

- Pass 2

[**5, 6,** 4, 8, 2, 9, 1, 3, 7, 10]

[5, **4, 6,** 8, 2, 9, 1, 3, 7, 10] (*)

[5, 4, **6, 8,** 2, 9, 1, 3, 7, 10]

[5, 4, 6, **2, 8,** 9, 1, 3, 7, 10] (*)

[5, 4, 6, 2, **8, 9,** 1, 3, 7, 10]

[5, 4, 6, 2, 8, **1, 9,** 3, 7, 10] (*)

[5, 4, 6, 2, 8, 1, **3, 9,** 7, 10] (*)

[5, 4, 6, 2, 8, 1, 3, **7,|9,** 10] (*)

- Pass 3

[**4, 5,** 6, 2, 8, 1, 3, 7, 9, 10] (*)

[4, **5, 6,** 2, 8, 1, 3, 7, 9, 10]

[4, 5, **2, 6,** 8, 1, 3, 7, 9, 10] (*)

[4, 5, 2, **6, 8,** 1, 3, 7, 9, 10]

[4, 5, 2, 6, **1, 8,** 3, 7, 9, 10] (*)

[4, 5, 2, 6, 1, **3, 8,** 7, 9, 10] (*)

[4, 5, 2, 6, 1, 3, <u>**7, |8,**</u> 9, 10] (*)

- Pass 4

[<u>**4, 5,**</u> 2, 6, 1, 3, 7, 8, 9, 10]

[4, <u>**2, 5,**</u> 6, 1, 3, 7, 8, 9, 10] (*)

[4, 2, <u>**5, 6,**</u> 1, 3, 7, 8, 9, 10]

[4, 2, 5, <u>**1, 6,**</u> 3, 7, 8, 9, 10] (*)

[4, 2, 5, 1, <u>**3, 6,**</u> 7, 8, 9, 10] (*)

[4, 2, 5, 1, 3, <u>**6, |7,**</u> 8, 9, 10]

- Pass 5

[<u>**2, 4,**</u> 5, 1, 3, 6, 7, 8, 9, 10] (*)

[2, <u>**4, 5,**</u> 1, 3, 6, 7, 8, 9, 10]

[2, 4, <u>**1, 5,**</u> 3, 6, 7, 8, 9, 10] (*)

[2, 4, 1, <u>**3, 5,**</u> 6, 7, 8, 9, 10] (*)

[2, 4, 1, 3, <u>**5, |6,**</u> 7, 8, 9, 10]

- Pass 6

[<u>**2, 4**</u>, 1, 3, 5, 6, 7, 8, 9, 10]

[2, <u>**1, 4,**</u> 3, 5, 6, 7, 8, 9, 10] (*)

[2, 1, <u>**3, 4,**</u> 5, 6, 7, 8, 9, 10] (*)

[2, 1, 3, <u>**4, |5,**</u> 6, 7, 8, 9, 10]

- Pass 7

[<u>**1, 2,**</u> 3, 4, 5, 6, 7, 8, 9, 10] (*)

[1, <u>**2, 3,**</u> 4, 5, 6, 7, 8, 9, 10]

[1, 2, <u>**3, 4,**</u> 5, 6, 7, 8, 9, 10] (Sorted array)

- Tracing array [5, 6, 8, 4, 10, 2, 9, 1, 3, 7] by using selection sort algorithm.

<u>Underlined</u> numbers are biggest numbers in the array.

[5, 6, 8, 4, <u>10</u>, 2, 9, 1, 3, 7] (Unsorted original array)

[5, 6, 8, 4, 7, 2, <u>9,</u> 1, 3, **|10]**

[5, 6, <u>8,</u> 4, 7, 2, 3, 1, **|9, 10]**

[5, 6, 1, 4, <u>7,</u> 2, 3, **|8, 9, 10]**

[5, <u>6,</u> 1, 4, 3, 2, **|7, 8, 9, 10]**

[<u>5,</u> 2, 1, 4, 3, **|6, 7, 8, 9, 10]**

[3, 2, 1, <u>4,</u> **|5, 6, 7, 8, 9, 10]**

[<u>3,</u> 2, 1, **|4, 5, 6, 7, 8, 9, 10]**

[1, <u>2,</u> **|3, 4, 5, 6, 7, 8, 9, 10]**

[1, <u>2,</u> **|3, 4, 5, 6, 7, 8, 9, 10]**

[<u>1,</u> **|2, 3, 4, 5, 6, 7, 8, 9, 10]**

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (Sorted array)

Part c:

- The recurrence relation of quick sort algorithm for the worst case:

Worst Case: Pivot is always the largest or smallest number in the array.

T(0) = 0 and T(1) = 0

T(n) = T(n-1) + n

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

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

...

= $T(n-k) + \sum_{i=2}^{k} i$ (because T(0) = T(1) = 0)

...

= $\Theta(n^2)$

Question 2:

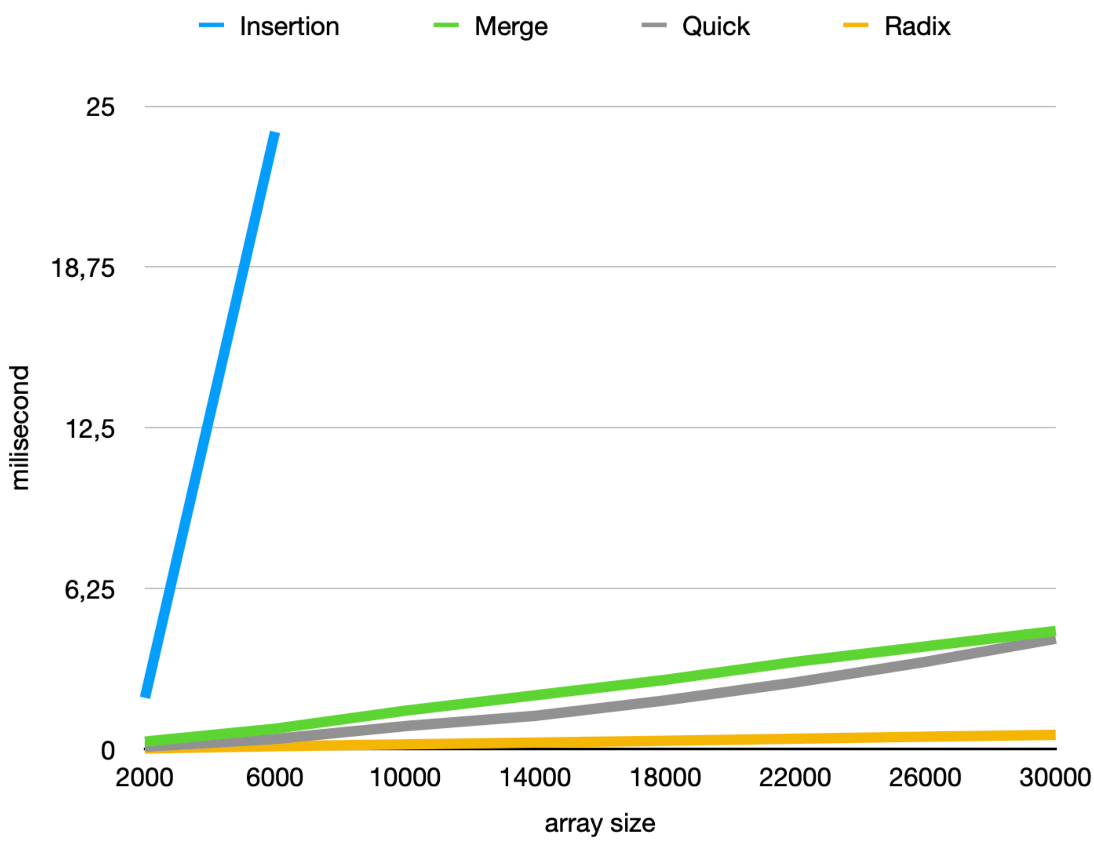
```
---------------- Insertion Sort ----------------
Original array is: 7 3 6 12 13 4 1 9 15 0 11 14 2 8 10 5
Sorted array is: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
---------------- Merge Sort ----------------
Original array is: 7 3 6 12 13 4 1 9 15 0 11 14 2 8 10 5
Sorted array is: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
---------------- Quick Sort ----------------
Original array is: 7 3 6 12 13 4 1 9 15 0 11 14 2 8 10 5
Sorted array is: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
---------------- Radix Sort ----------------
Original array is: 7 3 6 12 13 4 1 9 15 0 11 14 2 8 10 5
Sorted array is: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Part a — Time Analysis of Insertion Sort
-------------------------------------------------
Array Size      Time Elapsed    compCount       moveCount
2000            6.47923         1012061         1014060
6000            57.6113         9039833         9045832
10000           156.802         24656316                24666315
14000           309.51          48743273                48757272
18000           519.758         81440490                81458489
22000           767.976         120996268               121018267
26000           1067.05         167818914               167844913
30000           1421.82         223250989               223280988

Part b — Time Analysis of Merge Sort
-------------------------------------------------
Array Size      Time Elapsed    compCount       moveCount
2000            0.642655                19419           43904
6000            2.02911         67732           151616
10000           3.48126         120400          267232
14000           4.92945         175375          387232
18000           6.51234         231963          510464
22000           7.99881         290012          638464
26000           9.53255         348709          766464
30000           10.992          408426          894464

Part c — Time Analysis of Quick Sort
-------------------------------------------------
Array Size      Time Elapsed    compCount       moveCount
2000            0.39681         25404           35871
6000            1.32271         98545           101601
10000           2.43021         210970          186603
14000           3.84146         365185          305262
18000           5.34553         539621          382470
22000           6.96907         749995          462396
26000           8.95768         998458          672498
30000           10.3126         1211206         561393

Part d — Time Analysis of Radix Sort
-------------------------------------------------
Array Size      Time Elapsed    compCount       moveCount
2000            0.152193                0               0
6000            0.483102                0               0
10000           0.79771         0               0
14000           1.08976         0               0
18000           1.40551         0               0
22000           1.69108         0               0
26000           2.01966         0               0
30000           2.30284         0               0
[dilay.yigit@dijkstra CS202]$
```



Question 3:

Empirical result is the graph above. This result is similar to theoretical result because, when we consider the average cases, the time complexity of insertion sort is O(N^2). As we can see in the graph, elapsed time of insertion sort grows exponentially. For that reason, using insertion sort for large array sizes is not quite effective. On the other hand, the time complexity of both merge sort and quick sort is O(N log N) for average cases. In the graph, these two algorithm's growth rates are very similar. Both of sorting functions use divide and conquer method. So that, it is not inefficient like insertion sort algorithm. Finally, the time complexity of radix sort is O(N) for average cases. It is the most efficient algorithm to use for larger array sizes. When we compare theoretical time complexities and the empirical result, we can conclude that results are very similar. There is only one difference which is due to the fact that we create random arrays every time we run the program, quick sort and merge sort is not completely same although time complexities are the same. But this difference is normal because we only consider average cases when we interpret results.

If we applied the sorting algorithms to an array of increasing numbers instead of randomly generated numbers, the result would almost be same in terms of elapsed time. When we consider the best cases of these four sorting algorithms, only insertion sort has a different time complexity in best case which is O(N). The difference between best and average case of insertion sort is quite significant. On the other hand, the time complexity of quick, merge and radix sorting algorithms for the best case is the same as the time complexity of average cases. Also, in terms of the data moves, in the best cases, there will not be any data moves because the array is already sorted. In conclusion, the best case of insertion sort is the same as the average/best case of radix sort. However, it is not logical to select sorting method by considering the best case. For that reason, radix sort should be used for large array sizes.