Title: Algorithm Efficiency and Sorting

Author: Zeynep Cankara

ID: 21703381

Section: 2

Assignment: 1

Description: File contains answers of questions 1, 2 and 3.

Question 1.a:

Show:

- $f(n) = 100n^3 + 8n^2 + 4n$ is $O(n^4)$

Condition:

- $0 \leq T(n) \leq cf(n)$, $\forall n \geq n_0$; There exists $\exists$ n_0, c positive constants.
- $0 \leq 100n^3 + 8n^2 + 4n \leq cn^4$ for all $n \geq n_0$

Choosing $c = 100$ and $n_0 = 2$ $\rightarrow$ $100n^3 + 8n^2 + 4n \leq 100n^4$ for all $n \geq 2$

Question 1.b:

Solve Recurrence Relation by using repeated substitution method

- $T(n) = 8T(\frac{n}{2}) + n^3$

$T(1), \ldots, T(100) = 1$

$$T(n) = 8\,(8\,T(\frac{n}{4}) + (\frac{n}{2})^3) + n^3$$

$$T(n) = 8(\,8(\,8T(\frac{n}{8}) + (\frac{n}{4})^3) + (\frac{n}{2})^3) + n^3$$

$$T(n) = T(\frac{n}{8})\prod_{i=1}^{3} 8 + \sum_{i=1}^{3} 8^i(\frac{n}{2^i})^3$$

Carry out $\log_2 n$ operations to reach the base case…

$$T(n) = T(\frac{n}{2^{log_2 n}}) \prod_{i=1}^{log_2 n} 8 \; + \sum_{i=1}^{log_2 n} n^3$$

$T(n) = T(1)\,8^{log_2 n} + n^3 log_2 n$

$T(n) = 2^{log_2 n^3} + n^3 log_2 n \;= n^3(1 + log_2 n)$

$T(n) = n^3 + n^3 log_2 n$

Ignore constants and low order terms…

$T(n) = \Theta(n^3 \cdot log_2 n)$

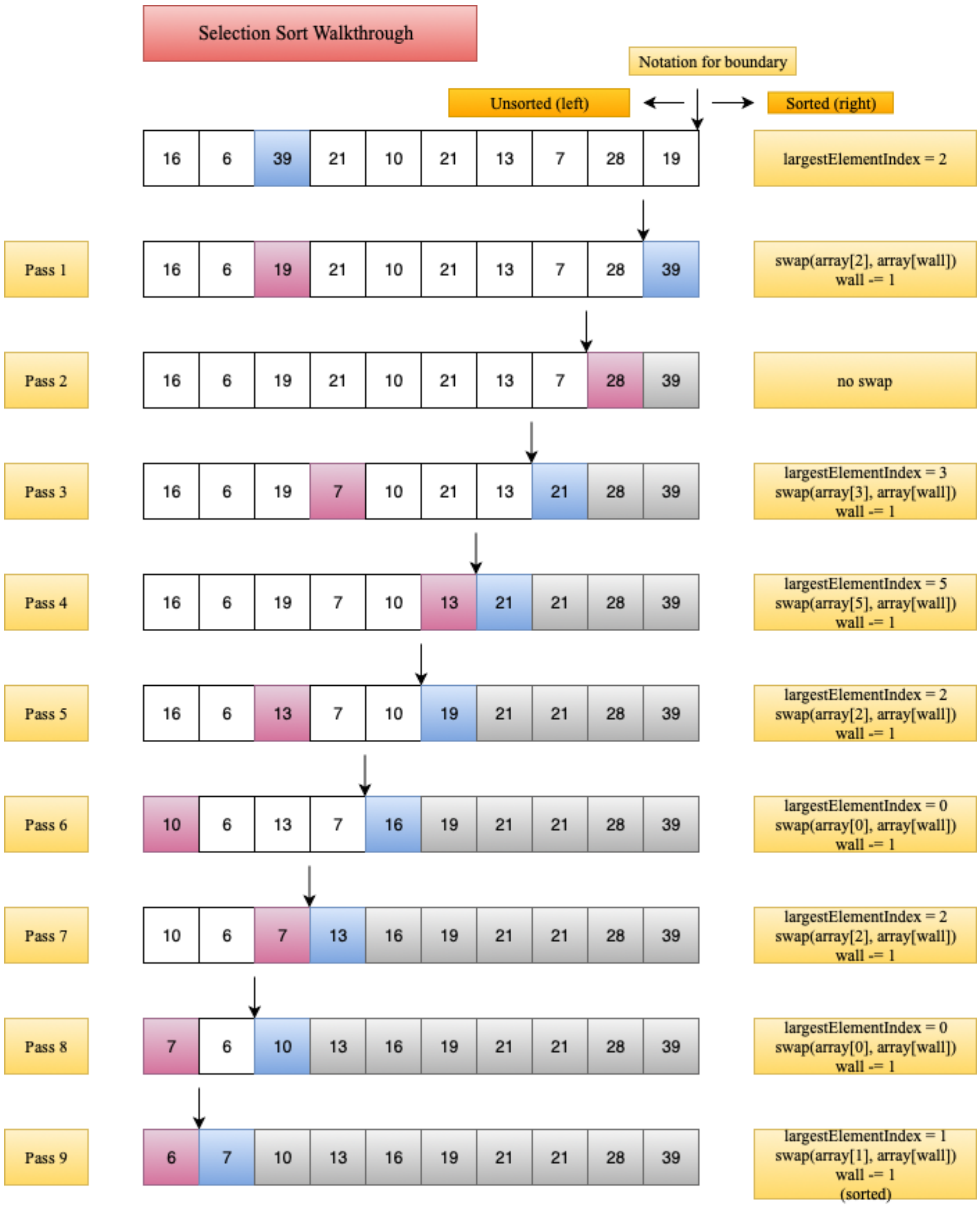
Question 1.c:

```
for (i = n; i > 0; i /= 2)   // the loop will execute log2n times
  for (j = 1; j < n; j++)  // (nested within a for-loop) the loop will execute n* log2n times
    for (k = 1; k < n; k += 2)  // (nested within 2 for-loops) the loop will execute n *n* log2n times
      sum += (i + j * k);  // 2 additionsΘ(1) , 1 multiplication Θ(1) and 1 assignment Θ(1) operation.
```

Total of $4n^2 \cdot log_2 n$ operations performed. Dropping the constants and low order terms.

Running Time Complexity: $\Theta(n^2 \cdot log_2 n)$

Question 1.d:



PROCEDURE

1. Unsorted/sorted boundary pointing to the last element.
2. Starting from the beginning of the array find the index which contains the largest element
3. swap the element with the element at the decision boundary.
4. decrement the decision boundary by one.
5. Repeat the procedure (n-1) times where n is the number of items in the array.


Insertion Sort Walkthrough
Notation for boundary
Sorted (left)
Unsorted (right)
16 6 39 21 10 21 13 7 28 19
key = 1. Move 6 to index 0
Pass 1
6 16 39 21 10 21 13 7 28 19
key = 2. No change
Pass 2
6 16 39 21 10 21 13 7 28 19
key = 3. Move 21 to index 2
Pass 3
6 16 21 39 10 21 13 7 28 19
key = 4. Move 10 to index 2.
Pass 4
6 10 16 21 39 21 13 7 28 19
key = 5. Move 21 to index 4.
Pass 5
6 10 16 21 21 39 13 7 28 19
key = 6. Move 13 to index 2.
Pass 6
6 10 13 16 21 21 39 7 28 19
key = 7. Move 7 to index 1.
Pass 7
6 7 10 13 16 21 21 39 28 19
key = 8. Move 28 to index 7.
Pass 8
6 7 10 13 16 21 21 28 39 19
key = 9. Move 19 to index 5.
Pass 9
6 7 10 13 16 19 21 21 28 39
Sorted!!!
PROCEDURE
1. Unsorted/sorted boundary pointing to the second element.
2. Starting from the index 1 choose element at the boundary as key
3.Iterate through the left (sorted) portion of the array.
4.When you encounter an element less than the key or when index becomes negative stop and do the insertion
5. Increment the boundary and choose the new item at boundary index. Repeat the procedure (n-1) times where n is the number of items in the array.
key
compared
not-compared

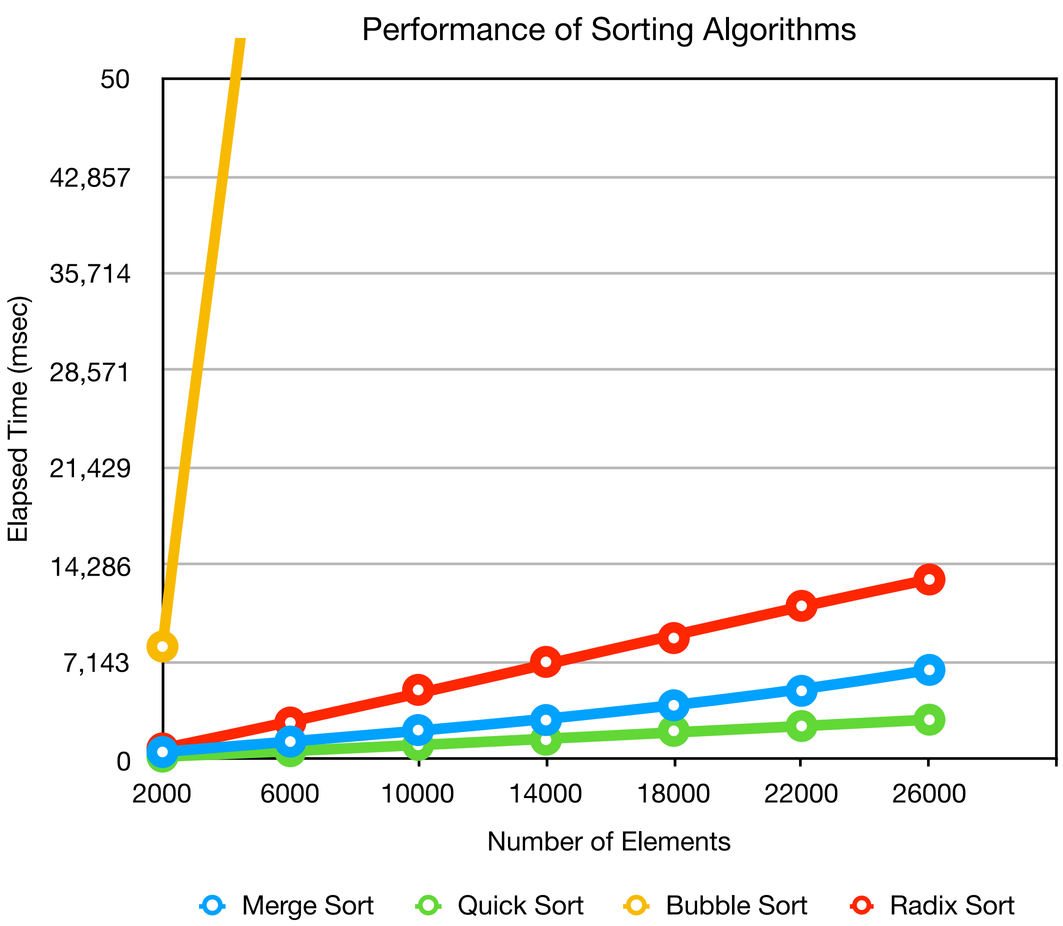
Question 2:

```
 Start the Performance Analysis
------------------------------------
Part c - Time analysis of Radix Sort
Array size   Time Elapsed
2000           0.812000 ms
6000           2.715000 ms
10000           5.114000 ms
14000           7.171000 ms
18000           8.921000 ms
22000           11.287000 ms
26000           13.227000 ms
30000           14.770000 ms
------------------------------------
Part c - Time analysis of Bubble Sort
Array size   Time Elapsed      compCount      moveCount
2000          8.295000 ms    1999000     2967621
6000          87.948000 ms    17997000     27159501
10000          255.525000 ms    49995000     74288394
14000          516.696000 ms    97993000     147199770
18000          890.076000 ms    161991000     246058329
22000          1324.135000 ms    241989000     361191102
26000          1829.886000 ms    337987000     508134375
30000          2488.002000 ms    449985000     677132907
------------------------------------
Part c - Time analysis of Quick Sort
Array size   Time Elapsed      compCount      moveCount
2000          0.209000 ms      22976       37868
6000          0.608000 ms      85328       134494
10000          1.070000 ms      158339      264376
14000          1.451000 ms      218533      342154
18000          2.102000 ms      305207      517801
22000          2.458000 ms      365771      575039
26000          2.923000 ms      438741      720080
30000          3.478000 ms      540169      816342
------------------------------------
Part c - Time analysis of Merge Sort
Array size   Time Elapsed      compCount      moveCount
2000           0.552000 ms      19425       43904
6000           1.317000 ms      67816       151616
10000           2.161000 ms      120571       267232
14000           2.898000 ms      175295       387232
18000           3.982000 ms      231999       510464
22000           5.043000 ms      290100       638464
26000           5.608000 ms      349220       766464
30000           6.581000 ms      408643       894464
```

Question 3:

Performance of Sorting Algorithms

Elapsed Time (msec)				
Array Size	Merge Sort	Quick Sort	Bubble Sort	Radix Sort
2000	0.5520	0.2090	8.2950	0.8120
6000	1.3170	0.6080	87.9480	2.7150
10000	2.1610	1.0700	255.5250	5.1140
14000	2.8980	1.4510	516.6960	7.1710
18000	3.9820	2.1020	890.0760	8.9210
22000	5.0430	2.4580	1324.1356	11.2870
26000	5.6080	2.9230	1829.8860	13.2270
30000	6.5810	3.4780	2480.0020	14.7700



Comments

- It can be seen from both graph and the table that quick sort performs slightly better than the merge sort when sorting an array which contains randomly generated.
- The theoretical result suggests that merge sort is an $O(nlog(n))$ in worst-case, best-case, average-case. On the other hand quick sort is an $O(nlog(n))$ in the average case only and $O(n^2)$ in the worst-case.
- Question: Why quick sort outperforms merge sort in the case of homework assignment.?
- Answer: Because in practice (real life) most frequently data distributed randomly so quick sort efficiency close to average case rather than worst case. Thus, merge sort is not an in-place algorithm and requires extra memory. Creation of the auxiliary memory space for array takes time as well.
- Radix Sort is an $O(nk)$ algorithm in the average case and the worst-case where ($n =$ number of elements) and ($k =$ number of digits). The radix sort outperforms bubble sort which is an $O(n^2)$ algorithm in the worst case and in the average case. Thus, radix sort underperforms both merge sort and quick sort which is perfectly aligned with the theoretical result.
- If we were to apply our sorting algorithms to array of <u>decreasing numbers</u> than it will likely that we will observe merge sort performing better than quick sort because now we lost our advantage of dividing the list into half with each recursive calls. Thus, we lost our logarithmic advantage, quick sort operates with worst case $O(n^2)$. Bubble Sort, merge sort and radix sort will still continue same with their worst case complexity. Radix sort can outperform quick sort which has $O(nk)$ complexity in the worst case. Bubble Sort can outperform quick sort since space complexity of bubble sort is $O(1)$. Whereas space complexity of quick sort is $O(log(n))$ due to swaps during partitions.