Assignment N*1

Name: Harlee Ramos **Due Date:** 05/26/2024

Use pseudo-code in questions that need you to present your algorithms, but your pseudo-code can be "English-like" if the operation is obvious. Make sure that each line in your pseudo-code is numbered, and the indentation is correct.

1. Arrange the following functions of n (you may assume the domain of variable n is $\mathbb{N}$) so that each function is big-Oh of the next function. In other words, order them by their speed of growth in non-decreasing order. Group together those functions that are of the same order.

$6n \times \lg n$	$2^{\lg n}$	e^n	2^{100}	$\lg \lg n$
$\ln n^2$	2^{2^n}	n^3	$n^{\frac{1}{100}}$	$4 \times n^{\frac{3}{2}}$
$\log n^{100}$	4^n	$2^{2^{\lg n}}$	$\sqrt{n^3}$	$n^{0.5} + 1$
$n \log_4 n$	$\lg^2 n$	$100^{100^{100}} n$	$4^{\lg n}$	$n^2 \log n$
1	$n \cdot 2^n$	$\lg(n/2)$	$n \lg^2 n$	$\sqrt{\log n}$

Constant functions (they do not grow with n)

1. 1
2. 2^{100}

Logarithmic functions (they grow very slowly with n)

3. $lg\ lg\ n$
4. $lg(n/2)$
5. $\sqrt{log\ n}$
6. $lg^2 n$
7. $ln\ n^2 = 2\ ln\ n$
8. $log\ n^{100} = 100\ log\ n$

Polynomial functions (they grow moderately with n)

9. $n^{1/100}$
10. $\sqrt{n^3} = n^{1.5}$
11. $n^{0.5} + 1$
12. $6n\ lg\ n$
13. $nlog_4 n$
14. $nlg^2 n$ //This function it a polynomial-logarithmic hybrid
15. $2^{lgn} = n$
16. $4^{lgn} = 2^{2lgn} = n^2$ // #15 and #16 have the same time complexity
17. $2^{2lgn} = n^2$
18. $4n^{3/2}$
19. $n^2 log\ n$
20. n^3

Exponential functions (they grow rapidly with n)

21. e^n
22. 4^n
23. $n2^n$

Double exponential functions (they grow extremely rapidly with n)

24. 2^{2^n}

Triple exponential function (it grows even more rapidly with n)

25. $100^{100^{100}} n$

2. Consider an alternative version of the merge sort: instead of dividing an array into two sub-arrays evenly, we now divide the array into three sub-arrays evenly.
   a) How to merge three sorted arrays? What is the worst-case time complexity of your merge-three-arrays algorithm?

Alternative way:

```
1. MERGE_THREE(A, L1, L2, L3)
2. i = j = k = 0 // Initialize pointers for L1, L2, and L3
3. for m = 0 to length(A) - 1
4. if i < length(L1) and (j >= length(L2) or L1[i] <= L2[j]) and (k >= length(L3) or L1[i]
   <= L3[k])
5. A[m] = L1[i]
6. i = i + 1
7. else if j < length(L2) and (k >= length(L3) or L2[j] <= L3[k])
8. A[m] = L2[j]
9. j = j + 1
10. else
11. A[m] = L3[k]
12. k = k + 1
```

Steps to merge three sorted arrays:

1. Initialize three points for each array (using indexes i,j and k)
2. Compare the elements at the pointers and select the smallest element.
3. Add that smallest element to the sorted array.
4. Increment the pointer corresponding from which the smallest element was selected.
5. Repeat steps 2 to 4 until all elements from all three arrays have been combined into the final array.

Worst-case time complexity analysis

Perform the initialization operations, which require a constant time Θ(1), followed by the comparison and moving operations, whose steps correspond to the lengths of each array. For instance, if the length of array L1 is n, the length of array L2 is m, and the length of array L3 is p, and you combine a total of 3n array elements, you can express the time complexity as Θ(n+ m+ p). However, if you choose a naive approach that involves concatenating the three arrays into one, the worst-time complexity becomes $\Theta((n + m + p)\log(n + m + p))$. Note also the three lengths can be expressed just as n denoting a Θ(n) and Θ(nlog(n)) notation.

   b) Present pseudo code for the alternative version of the merge sort and analyze its time complexity.

Pseudocode of the TriMergeSort

```
TRIMERGE_SORT(A, p, r)
1. if p < r
2.     q1 = p + (r - p) / 3        // Calculate the 1st one-third point
3.     q2 = q1 + (r - p) / 3       // Calculate the 2nd one-third point
4.     TRIMERGE_SORT(A, p, q1)     // Recursively sort the left sub-array
5.     TRIMERGE_SORT(A, q1 + 1, q2)// Recursively sort the middle sub-array
6.     TRIMERGE_SORT(A, q2 + 1, r) // Recursively sort the right sub-array
7.     MERGE_THREE(A, p, q1, q2, r)// Merge the three sorted sub-arrays

MERGE_THREE(A, p, q1, q2, r)
1. n1 = q1 - p + 1
2. n2 = q2 - q1
3. n3 = r - q2
4. let L[1 .. n1], M[1 .. n2], R[1 .. n3] be new arrays
```

```
5.  for i = 0 to n1 - 1
6.      L[i] = A[p + i]
7.  for j = 0 to n2 - 1
8.      M[j] = A[q1 + 1 + j]
9.   for k = 0 to n3 - 1
10.   R[k] = A[q2 + 1 + k]
11.  i = j = k = 0
12.  for m = p to r
13.    if i < n1 and (j >= n2 or L[i] <= M[j]) and (k >= n3 or L[i] <= R[k])
14.      A[m] = L[i]
15.      i = i + 1
16.    else if j < n2 and (k >= n3 or M[j] <= R[k])
17.      A[m] = M[j]
18.      j = j + 1
19.    else
20.      A[m] = R[k]
21.      k = k + 1
```

Worst-case time complexity analysis

The operation of merging the arrays takes Θ(1), then the recursive approach of sorting will take T(n/3) for each array, them by replacing the values on the master theorem of the book

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

In this case a = 3, b= 3 and f(n)= Θ(1) and $n^{log_3 3} = n$

$$T(n) = 3T(n/3) + \Theta(n) \Longrightarrow \Theta(n \log n)$$

3. Let $A[1 \ldots n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then we say (i, j) is an *inversion* of A. Answer the following questions.

   a) What are the inversions in array $\{1,4,2,9,6,3\}$?

Array

Index	1	2	3	4	5	6
Element	1	4	2	9	6	3
Q of Inver.		2		2	1	

Pairs (i, j) where i < j and A[i] > A[j]:

4 > 2 True, then (4,2) → which represents indexes (2,3)
4 > 3 True, then (4,3) → which represents indexes (2,6)
9 > 6 True, then (9,6) → which represents indexes (4,5)
9 > 3 True, then (9,3) → which represents indexes (4,6)
6 > 3 True, then (6,3) → which represents indexes (5,6)

5 Inversions:

$\binom{4}{2},\binom{2}{3}$, $\binom{4}{2},\binom{3}{6}$, $\binom{9}{4},\binom{6}{5}$, $\binom{9}{4},\binom{3}{6}$, $\binom{6}{5},\binom{3}{6}$

b) Consider the same array as in question a). Denote the number of inversions within the first half-array (aka $\{1,4,2\}$) as a, then denote the number of inversions within the second half-array (aka $\{9,6,3\}$) as b, and denote the number of inversions crossing two halves as c. What are the values of a, b, c? What's your observation about values a, b, c and the answer of part a)?

Splited array:

First half: a

Index	1	2	3
Element	1	4	2

4 > 2 True, then (4,2) → which represents indexes (2,3)
1 Inversion : $\binom{4}{2},\binom{2}{3}$
a = 1

Second half: b

Index	1	2	3
Element	9	6	3

9 > 6 True, then (9,6) → which represents indexes (1,2)
9 > 3 True, then (9,3) → which represents indexes (1,3)
6 > 3 True, then (6,3) → which represents indexes (2,3)

3 Inversions : $\binom{9}{1},\binom{6}{2}$, $\binom{9}{1},\binom{3}{3}$, $\binom{6}{2},\binom{3}{3}$
b = 3.

Inversions crossing the two halves: c
9 > 2 True, then represents indexes (4,3)
1 Inversions
c = 1.
The total number of inversions (sum of all letters) 1 + 3 + 1 = 5.
Using this method, we can break down the total number of inversions in the entire array into the sum of inversions within each half of the array and inversions which cross the two halves. Algorithms that count inversions, such as those used in merge sort improvements, can take advantage of this behavior, as merging two halves also includes counting cross-halve inversions.

This method can result in an efficient O(nlogn) solution for counting inversions, which is much faster than the brute-force $O(n^2)$ approach.

c) Consider the array in part a) and the values a, b, c in part b). If I only change the orderings inside both halves, which value among a, b, c won't change?

Modifying the orderings within both halves (a and b) may alter the number of inversions within each half. Nevertheless, the count of inversions that cross the two halves (c) remains constant, as the components in one half are consistently compared to the elements in the other half, regardless of their internal order. Consequently, the value of c will remain unchanged.

d) Present an algorithm to calculate the number of inversions of $A[1 \dots n]$ with n distinct numbers by modifying merge sort. Hint: You can use the observations in parts a), b) and c).

```
1. MergeSortAndCount(A, b, a):
2. Base Case:  If b >= a, return (A, 0)
3. Recursive Case: Calculate the middle point:  c = (b + a) // 2
4. Recursively sort and count inversions in the left half:
5. (A, b_inv) = MergeSortAndCount(A, b, c)
6. Recursively sort and count inversions in the right half:
7. (A, a_inv) = MergeSortAndCount(A, c + 1, a)
8. Count and merge split inversions:
9. (A, c_inv) = MergeAndCountSplitInv(A, b, c, a)
10. Total inversions = b_inv + a_inv + c_inv
11. Return (A, Total_inv)

MergeAndCountSplitInv(A, b, c, a):
Create temporary arrays for the left (b to c) and right (c + 1 to a) subarrays
Initialize inversion count to 0
Merge the two halves while counting split inversions
Return the merged array and the count of split inversions
```

e) Analyze the time complexity your algorithm presented part d).

- **Division step** takes $\Theta(1)$ time.
- **Conquer step**, each conquer step (recursively sorting and counting) takes T(n/2), but it must be considered left and right representing 2 times T(n/2).
- **Merge step**, merging and counting split inversions take $\Theta(n)$
- **Time complexity:** $T(n) = 2T(n/2) + \Theta(n) \Longrightarrow \Theta(nlogn)$

4. Let $A[1 \ldots n]$ and $B[1 \ldots n]$ be two sorted arrays.
   a) Present an algorithm to find the median of all $2n$ elements in arrays A and B. To get full marks, your algorithm should have a worst-case time complexity $\Theta(\lg n)$. Hint: This algorithm can be inspired by the **Select** algorithm in Lecture 4.

```
1. //Initialize Pointers: Start with two pointers, one for each array. Set them initially to
   point to the first element of their respective arrays.
2. startA = 0
3. endA = n-1
4. startB = 0
5. endB = n-1
6. findMedian(A, B, startA, endA, startB, endB):
7. //Base Case: If the length of the array is n=1,
8.   If endA - startA == 0
9.     return the median as  median = (A[startA]+B[startB])/2
10.//Using recursive approach
11.//Base Case of two elements.
12.  If endA - startA == 1
13.//Calculate the new median
14.   median = (max(A[startA],B[startB])+min(A[startA],B[endB]))/2
15.//Calculate the middle index of both arrays
16.  middle_A = (startA+endA)/2
17.  middle_B = (startB+endB)/2
18.//Compare the middle elements in both arrays A[middle_index] and B[middle_index]:
19.//If A[middle_A] < B[middle_B], discard the first half of A and the second half of B, due to
   these values cannot be combine with the merged median.
20.//If B [middle_A] < A[middle_B], discard the first half of B and the second half of A.
21.  If A[middle_A] < B[middle_B
22.  findMedian(A, B, middleA, endA, startB, middleB)
23.   else
24.    findMedian(A, B, startA, middleA, middleB, endB)
25.//Call the function
26.    findMedian(A, B, startA, startA, startB, endB)
```

   b) Analyze the time complexity your algorithm presented part a).

Analysis of time complexity

1. In the base case when both arrays have a length size of n=1, takes 0(1).
2. On the recursive part of the algorithm (middle index calculation, comparison of middle indexes, discard of halves and recursive steps), will take T(n/2), because the divide and conquer approach stablished the reduction to n/2.

$$T(n) = T(n/2) + \Theta(1)$$

Applying theorem 4.1. Doing the relationship between the master theorem for Divide and Conquer, provided on page 94, screenshot from the textbook:

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

For this problem is the case 2. $f(n) = \Theta(1)$, $n^{log_b a} = 1$

$$T(n) = T(n^{log_b a} * logn) = \Theta(1 * logn) = \Theta(logn)$$

Then the time complexity for this approach is $\Theta(logn)$