HOMEWORK – 1

Introduction to Algorithms (CS 430)

1. Arranging the functions of n in non-decreasing order of their speed of growth.

Order by Growth in non-decreasing order

- 1
- 2^{100}
- lg lg n
- $\sqrt{log\ n}$
- lg^2 n
- log n^{100}
- lg n/2
- $n^{1/100}$
- $n^{0.5} + 1$
- ln n^2
- n lg_4 n
- 6n . lg n
- n . lg^2 n
- n^2 log n
- $\sqrt{n\text{^}3}$
- 4 x $n^{3/2}$
- n^3
- $2^{lg\ n}$
- $4^{lg\ n}$
- $2^{2\text{^}lg\ n}$
- n . 2^n
- 4^n
- e^n
- $2^{2\text{^}n}$
- $100^{100\text{^}100}$ n

Grouping together with functions that are of same order

Constants:

- 1
- 2^{100}

Log:

- lg^2 n
- log n^{100}

- $\lg n/2$
- $\lg^2 n$

Double Log:

- $\lg \lg n$

Poly Log:

- $\ln n^2$

Sub-poly:

- $n^{0.5} + 1$

Linearithmic:

- $n \lg_4 n$
- $6n \cdot \lg n$
- $n \cdot \lg^2 n$

Polynomial:

- $n^2 \log n$
- $\sqrt{n\text{^}3}$
- $4 \times n^{3/2}$
- n^3

Exponential (based on polynomial):

- $2^{\lg n}$
- $4^{\lg n}$
- $2^{2\text{^}\lg n}$

Exponential:

- $n \cdot 2^n$
- 4^n

Double Exponential:

- $2^{2\text{^}n}$

Super Exponential:

- $2^{2\text{^}n}$
- $100^{100\text{^}100}\ n$

2. Alternative version of the merge sort with three sub-arrays

a) Merging Three Sorted Arrays

- When merging three sorted arrays, you can follow a similar approach to merging two arrays. Here's how you can merge three sorted arrays efficiently:
    1. Compare the first elements of all three arrays.
    2. Select the smallest element among the three.
    3. Remove the smallest element from its array and add it to the merged array.
    4. Repeat steps 1-3 until all elements from all three arrays are merged.

The worst-case time complexity of merging three sorted arrays of size n would be O(n), as each comparison and element movement take constant time, and all elements need to be processed.

b) Alternative Merge Sort with Three-Way Split

```
1. MergeSortAlternative(arr):
2. if length of arr <= 1:
3. return arr
4. else:
5. divide arr into three sub-arrays: left, middle, right
6. left = MergeSortAlternative(left)
7. middle = MergeSortAlternative(middle)
8. right = MergeSortAlternative(right)
9. return MergeThreeArrays(left, middle, right)
```

Time Complexity Analysis

In this alternative version of Merge Sort, the time complexity can be analyzed using the divide-and-conquer recurrence relation:

- Let T(n) be the time complexity to sort an array of size n.
- In this case, we divide the array into three sub-arrays of size n/3 each.
- The time complexity can be expressed as: T(n) = 3 * T(n/3) + O(n).

By solving this recurrence relation, the time complexity of this alternative Merge Sort with three-way split can be determined.

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.

a) Inversions in array {1,4,2,9,6,3}

To determine the inversions of given array {1,4,2,9,6,3}

i) We need to identify all pairs of (i , j) such that i < j and A[i] > A[j]
ii) Compare each element with all other elements
iii) Now, we check the pairs where the first element > second element

Note: I have considered index value starting from 1.

Compare A[1] with next elements

A[1] > A[2] which is 1 < 4 no inversion
A[1] > A[3] which is 1 < 2 no inversion
A[1] > A[4] which is 1 < 9 no inversion
A[1] > A[5] which is 1 < 6 no inversion
A[1] > A[6] which is 1 < 3 no inversion

Result: As none of the elements are greater than the first element, no inversion is possible.

Compare A[2] with next elements

A[2] > A[3] which is 4 > 2 we get, inversion (2,3)
A[2] > A[4] which is 4 < 9 no inversion
A[2] > A[5] which is 4 < 6 no inversion
A[2] > A[6] which is 4 > 3 we get, inversion (2,6)

Result: Possible inversions (2,3) (2,6)

Compare A[3] with next elements

A[3] > A[4] which is 2 < 9 no inversion
A[3] > A[5] which is 2 < 6 no inversion
A[3] > A[6] which is 2 < 3 no inversion

Result: As none of the elements are greater than '2' no inversion is possible

Compare A[4] with next elements

A[4] > A[5] which is 9 > 6 we get, inversion (4,5)
A[4] > A[6] which is 9 > 3 we get, inversion (4,6)

Result: Possible inversions (4,5) (4,6)

Compare A[5] with A[6]

A[5] > A[6] which is 6 > 3 we get, inversion (5,6)

Result: Possible inversions (5,6)

***Therefore, the inversions in the array {1,4,2,9,6,3} are 5*: (2,3) (2,6) (4,5) (4,6) (5,6)**

b) Number of inversions in first half array a, and second half array b and inversions crossing two halves c.

To find the values of a, b, and c for the array {1,4,2,9,6,3}, we need to:

1. Determine the number of inversions within the first half {1,4,2}, which is a.
2. Determine the number of inversions within the second half {9,6,3}, which is b.
3. Determine the number of inversions that cross between the two halves, which is c.

Find inversions in first half (a):

A[1] > A[2] which is 1 < 4 no inversion
A[1] > A[3] which is 1 < 2 no inversion
A[2] > A[3] which is 4 > 2 we get, inversion (2,3)

Number of inversions in first half: **a = 1**

Find inversions in second half (b):

A[4] > A[5] which is 9 > 6 we get, inversion (4,5)
A[4] > A[6] which is 9 > 3 we get, inversion (4,6)
A[5] > A[6] which is 6 > 3 we get, inversion (5,6)

Number of inversions in second half: **b = 3**

Find inversions that crosses the halves (c):
An inversion crosses the halves if A[i] is in the first half and A[j] is in the second half, where i < j and A[i] > A[j].
Comparing elements from first half with elements from second half.
A[1] < A[4] , A[1] < A[5], A[1] < A[6] - no inversions
A[2] < A[4] , A[2] < A[5], A[2] > A[6] - we get inversion (2,6)
A[3] < A[4] , A[3] < A[5], A[3] < A[6] - no inversions
Number of inversions that crosses halves: c **= 1**

Observation:
Total no. of inversions = a + b+ c = 1 + 3 + 1 = 5

This matches the result from part (a), where the inversions in the array {1,4,2,9,6,3} were:(2,3) (2,6) (4,5) (4,6) (5,6)

c) Orderings inside both halves, which value among a,b,c won't change.

In the array {1,4,2,9,6,3} with the values a=1, b=3, and c=1 identified in parts a) and b), if you only change the orderings inside both halves, the value that won't change is c.

- **a:** The number of inversions within the first half {1,4,2} depends on the relative ordering of the elements within this half. If you change the ordering of these elements, the number of inversions a can change.
- **b:** Similarly, the number of inversions within the second half {9,6,3} depends on the relative ordering of the elements within this half. Changing the order of these elements can change the number of inversions b.
- **c:** The number of inversions that cross between the two halves depends on the relative ordering of elements in the first half compared to those in the second half. Since you are not changing which elements are in the first half and which are in the second half, but only the order within each half, the number of inversions that cross between the two halves (c) remains the same. Specifically, c is determined by the pairs where an element from the first half is greater than an element from the second half, which will not be affected by reordering within the halves themselves.

Therefore, value of c, which represents the number of inversions crossing between the two halves, won't change if you only change the orderings inside both halves.

d) Algorithm to Calculate Number of Inversions using Merge Sort

To calculate the number of inversions in an array A[1…n] of n distinct numbers, we can modify the merge sort algorithm. The idea is to count inversions while merging two sorted halves of the array. Here's how we can do it:

- Divide the array into two halves.
- Count inversions in the left half, right half, and while merging the two halves.
- Merge the two halves while counting cross-inversions.

Here is the algorithm: (pseudo-code)

```
function merge_and_count(A, temp, left, mid, right)
   i = left      // Starting index for left subarray
   j = mid + 1   // Starting index for right subarray
   k = left      // Starting index to be sorted
   inv_count = 0 // Number of inversions

   while i <= mid and j <= right
      if A[i] <= A[j]
         temp[k] = A[i]
         i = i + 1
      else
         temp[k] = A[j]
         inv_count = inv_count + (mid - i + 1)
         j = j + 1
      k = k + 1

   // Copy remaining elements of left subarray, if any
   while i <= mid
      temp[k] = A[i]
      i = i + 1
      k = k + 1

   // Copy remaining elements of right subarray, if any
   while j <= right
      temp[k] = A[j]
      j = j + 1
      k = k + 1

   // Copy the sorted subarray into Original array
   for i = left to right
      A[i] = temp[i]
   return inv_count

function merge_sort_and_count(A, temp, left, right)
   inv_count = 0
   if left < right
      mid = (left + right) // 2

      inv_count = inv_count + merge_sort_and_count(A, temp, left, mid)
      inv_count = inv_count + merge_sort_and_count(A, temp, mid + 1, right)
      inv_count = inv_count + merge_and_count(A, temp, left, mid, right)

   return inv_count

function count_inversions(A, n)
   temp = new array of size n
return merge_sort_and_count(A, temp, 0, n - 1)
```

e) Time Complexity Analysis

This merge sort algorithm has the same structure as the normal/regular merge sort, but has additional counting inversions.

1) Dividing array:
    - The array is divided into two halves, with Time complexity: O(log n)
2) Merging array:
    - Each merge involves comparing elements from two halves and merging them, with Time complexity: O(log n)
    - Total Time complexity for recursion tree is: O(log n)
    - Total Time complexity for merging across all levels is: O(n log n)
3) Counting the number of inversions:
    - But, for counting inversions during the merge does not add any additional asymptotic complexity, as it is performed suing O(n) merge.

Therefore, the overall ***Time Complexity of the modified merge sort algorithm is O(n log n).***

4. Let A[1 … n] and B[1 … n] be two sorted arrays

a) Algorithm to find the median of all 2n elements in arrays A and B?

To present an algorithm to find the median of all 2n elements in arrays A and B. We can use divide and conquer method which is inspired by the SELECT algorithm in Lecture 4 with worst case time complexity $\Theta(\lg n)$.

Algorithm:

1. **Base Case**
    - **If n = 1: return median(A[0], B[0])**
2. **Recursive Case:**
    - Find the median of A, denoted as A_m. Since A is sorted, A_m =A[n/2] (using integer division).
    - Find the median of B, denoted as B_m. Similarly, B_m = B [n/2]
    - Compare A_m and B_m:
        - If $A_m < B_m$, discard the first half of A and the second half of B, as the median must be in the remaining elements.
        - If $A_m > B_m$, discard the second half of A and the first half of B.
    - Recursively apply the same procedure to the reduced arrays.

Algorithm to find median of all 2n elements in arrays A and B:

```
def findMedian(A, B):
    n = len(A)
    if n == 1:
        return (A[0] + B[0]) / 2.0
    if n == 2:
        return (max(A[0], B[0]) + min(A[1], B[1])) / 2.0

    //Find the medians of A and B
    mid = n // 2
    A_m = A[mid]
    B_m = B[mid]

    if A_m < B_m:
        //The median must be in A[mid:] or B[:mid+1]
        return findMedian(A[mid:], B[:mid+1])
    else:
        //The median must be in A[:mid+1] or B[mid:]
        return findMedian(A[:mid+1], B[mid:])
```

b) Time Complexity Analysis

- *Base Case:* Θ(1) time.
- *Recursive Case:* In each recursive step, the problem size is halved, hence the depth of recursion is log n.
- In each step, comparisons and subarray selections take constant time, i.e., Θ(1).
- Thus, the overall time complexity is Θ(log n).