National University of Computer and Emerging Sciences
Lahore Campus

D & A of Algorithms (CS2009)

Sessional-I Exam

Date: Sep 22 2025

Course Instructor(s)

Dr. MB, Dr. SK, Dr. MB, Dr. MAQ, AA, UH, SK

Total Time (Hrs): 1
Total Marks: 15
Total Questions: 3

_______________ Roll No   _______________ Section   _______________ Student Signature

Instructions: Answer in the space provided. Do not attach rough sheets with this exam.

CLO 2: **Analyze** the time and space complexity of different algorithms by using standard asymptotic notations for recursive and non-recursive algorithms.

Question 1: [2+ 2 + 3 = 7 Marks]

(a) $f(n) = 30*2^n + 15*4^n + 3*16^n$

Which of the following statements are true about f(N)

i. $f(n) = O(4^{2n})$

ii. $f(n) = O(8^{n+2})$

iii. $f(n) = O(4^{n+4})$

iv. $f(n) = O(2^{4n})$

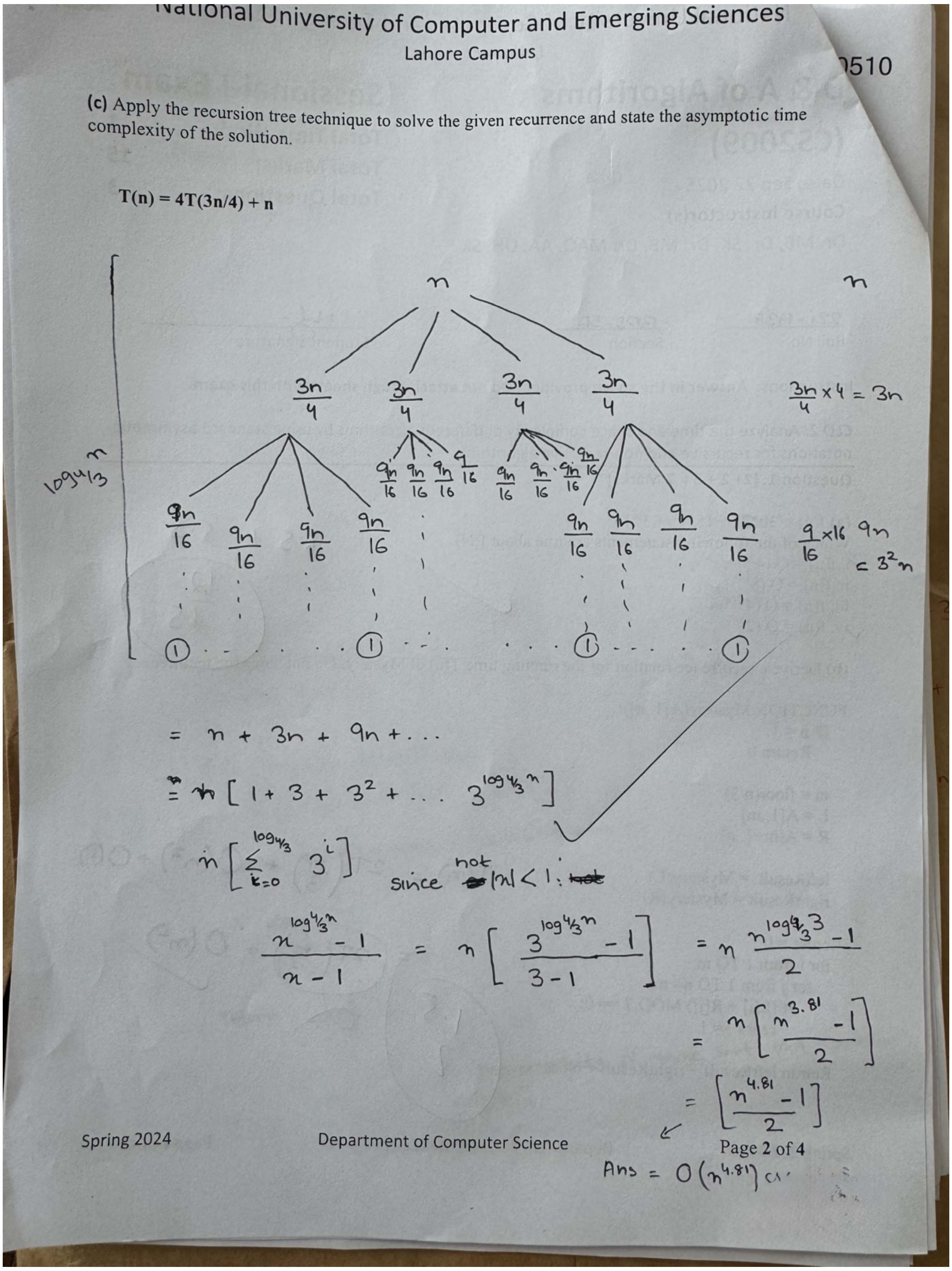
(b) Derive a recurrence relation for the running time T(n) of Mystery. Do not solve the recurrence.

```
FUNCTION Mystery(A[1..n]):
   IF n ≤ 1:
      Return 0

   m = floor(n/3)
   L = A[1..m]
   R = A[m+1..n].

   leftResult  = Mystery(L)
   rightResult = Mystery(R)

   cross = 0
   for i from 1 TO m:
      for j from 1 TO n - m:
         If (L[i] + R[j]) MOD 7 == 0:
            cross += 1

   Retrun leftResult + rightResult + cross
```

Solution: $T(n) = T(n/3) + T(2n/3) + O(n^2)$

(c) Apply the recursion tree technique to solve the given recurrence and state the asymptotic time complexity of the solution.

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



National University of Computer and Emerging Sciences
Lahore Campus

510

(c) Apply the recursion tree technique to solve the given recurrence and state the asymptotic time complexity of the solution.

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

n — n

$\frac{3n}{4}$ $\frac{3n}{4}$ $\frac{3n}{4}$ $\frac{3n}{4}$ — $\frac{3n}{4} \times 4 = 3n$

$\frac{9n}{16}$ $\frac{9n}{16}$ $\frac{9n}{16}$ $\frac{9n}{16}$ … — $\frac{9}{16} \times 16 \; 9n = 3^2 n$

$\log_{4/3} n$

1 … 1 … 1 … 1

$= n + 3n + 9n + \dots$

$= n \left[1 + 3 + 3^2 + \dots 3^{\log_{4/3} n} \right]$

$n \left[\sum_{i=0}^{\log_{4/3}} 3^i \right]$ since not $|x| < 1$ ~~not~~

$\frac{x^{\log_{4/3} n} - 1}{x - 1} = n \left[\frac{3^{\log_{4/3} n} - 1}{3 - 1} \right] = n \, \frac{n^{\log_{4/3} 3} - 1}{2}$

$= n \left[\frac{n^{3.81} - 1}{2} \right]$

$= \left[\frac{n^{4.81} - 1}{2} \right]$

Spring 2024 Department of Computer Science Page 2 of 4

Ans $= O(n^{4.81})$

CLO #1: Design algorithms using different algorithms design techniques i.e. Brute Force, Divide and Conquer, Dynamic Programming, Greedy Algorithms and apply them to solve problems in the domain of the program

Question 2: [2 Marks] Quick Sort's efficiency depends heavily on how the pivot is chosen. Describe a specific input and pivot-selection strategy that causes Quick Sort to run in $\Theta(n^2)$ time.

When the array is sorted, you always select the first or the last element of the array as the pivot.

Question 3: [2+4 = 6 Marks] Recall the problem of finding the number of inversions. We are given a sequence of n numbers $a_1, \ldots, a_n$, which we assume are all distinct, and we define an inversion to be a pair $i < j$ such that $a_i > a_j$. We motivated the problem of counting inversions as a good measure of the sortedness of an array. However, one might feel that this measure is too sensitive. Let's call a pair a significant inversion **if** $i < j$ **and** $a_i > 2a_j$.

(a) Give an $O(n^2)$ algorithm to count the number of significant inversions between two orderings.[2 Marks]

```
function count_significant_inversions_bruteforce(A[0..n-1]):

   count ← 0

   for i ← 0 to n-2:

      for j ← i+1 to n-1:

         if A[i] > 2 * A[j]:

            count ← count + 1

   return count
```

(b) Give an O(n log n) algorithm to count the number of significant inversions between two orderings.[4 Marks]

```
function countSignificantInversions(A, left, right):
    if left >= right:
        return 0

    mid = (left + right) // 2
    count = 0
    count += countSignificantInversions(A, left, mid)
    count += countSignificantInversions(A, mid+1, right)

    # Count cross inversions
    j = mid + 1
    for i from left to mid:
        while j <= right and A[i] > 2 * A[j]:
            count += (mid – i + 1))
            j += 1


    # Merge step (to keep array sorted)
    merge(A, left, mid, right)

    return count


def merge(A, left, mid, right):
    # Copy halves
    L = A[left:mid + 1]
    R = A[mid + 1:right + 1]

    i = 0
    j = 0
    k = left

    # Merge sorted halves back into A
    while i < len(L) and j < len(R):
        if L[i] <= R[j]:
            A[k] = L[i]
            i += 1
        else:
            A[k] = R[j]
            j += 1
        k += 1

    # Copy remaining elements (if any)
    while i < len(L):
        A[k] = L[i]
```

```
    i += 1
    k += 1

while j < len(R):
    A[k] = R[j]
    j += 1
    k += 1
```