National University of Computer & Emerging Sciences, Karachi
Fall-2021 Department of Computer Science
Mid Term-1
11th October 2021, 01:00 PM – 02:00 PM

Course Code: CS2009	**Course Name:** Design and Analysis of Algorithm
Instructor Name / Names: Dr. Muhammad Atif Tahir, Dr. Fahad Sherwani, Dr. Farrukh Saleem, Waheed Ahmed, Waqas Sheikh, Sohail Afzal	
Student Roll No:	**Section:**

Instructions:

- Return the question paper
- Read each question completely before answering it. There are **5 questions** on **2 pages**
- In case of any ambiguity, you may make assumption. But your assumption should not contradict any statement in the question paper

Time: 60 minutes. **Max Marks: 12.5**

Question # 1 **[0.5*3 = 1.5 marks]**

Solve the following recurrences using **Master's Method.** Give argument, if the recurrence cannot be solved using Master's Method. [See appendix for Master's method 4th case if required]

$\boldsymbol{a)\ T(n) = 6\,T\left(\frac{n}{\sqrt{n}}\right) + n + 30}$

$\boldsymbol{b)\ T(n) = 9\,T\left(\frac{n}{3}\right) + 3n^2 + 2^3\,n}$

$\boldsymbol{c)\ T(n) = 7T\left(\frac{n}{4}\right) + n^{log_4 7}\log n}$

Solution :

1) $6\,T\,(n/\sqrt{n}) + n + 30$

In this recurrence a= 6, d= 1, however b is not expressed as constant. Therefore, Master Theorem cannot applied.

2) $9\,T\,(n/3) + 3n^2 + 2^3\,n$

Recurrence can be written as $9\,T\,(n/3) + 3n^2 + 2^3\,n$

In this recurrence a= 9, b= 3, d= 2. So b^d is 3^2 = 9.
So a = b^d Master method Case 2 is applied and complexity is
$\boldsymbol{O(n^2 \log n)}$

3) $7\,T(n/4) + n^{\log_4 7} \log n$

In this recurrence a= 7, b= 4, k= 1. F(n) is polylogarithmic. $n^{\log_4 7}\ \log n$
So Master method Case 4 is applied and complexity is
$\boldsymbol{O(n^{\log_4 7}\ \log^2 n)}$

<u>Question # 2</u> **[1 + 2 = 3 marks]**

Compute the time complexity of the following recurrence relations by using **Iterative Substitution Method or Recurrence-Tree Method**. [See appendix for formulas if required]

$\boldsymbol{a)\ T(n) = 3T\left(\frac{2n}{3}\right) + n}$, **Assume $T(1) = 1$**

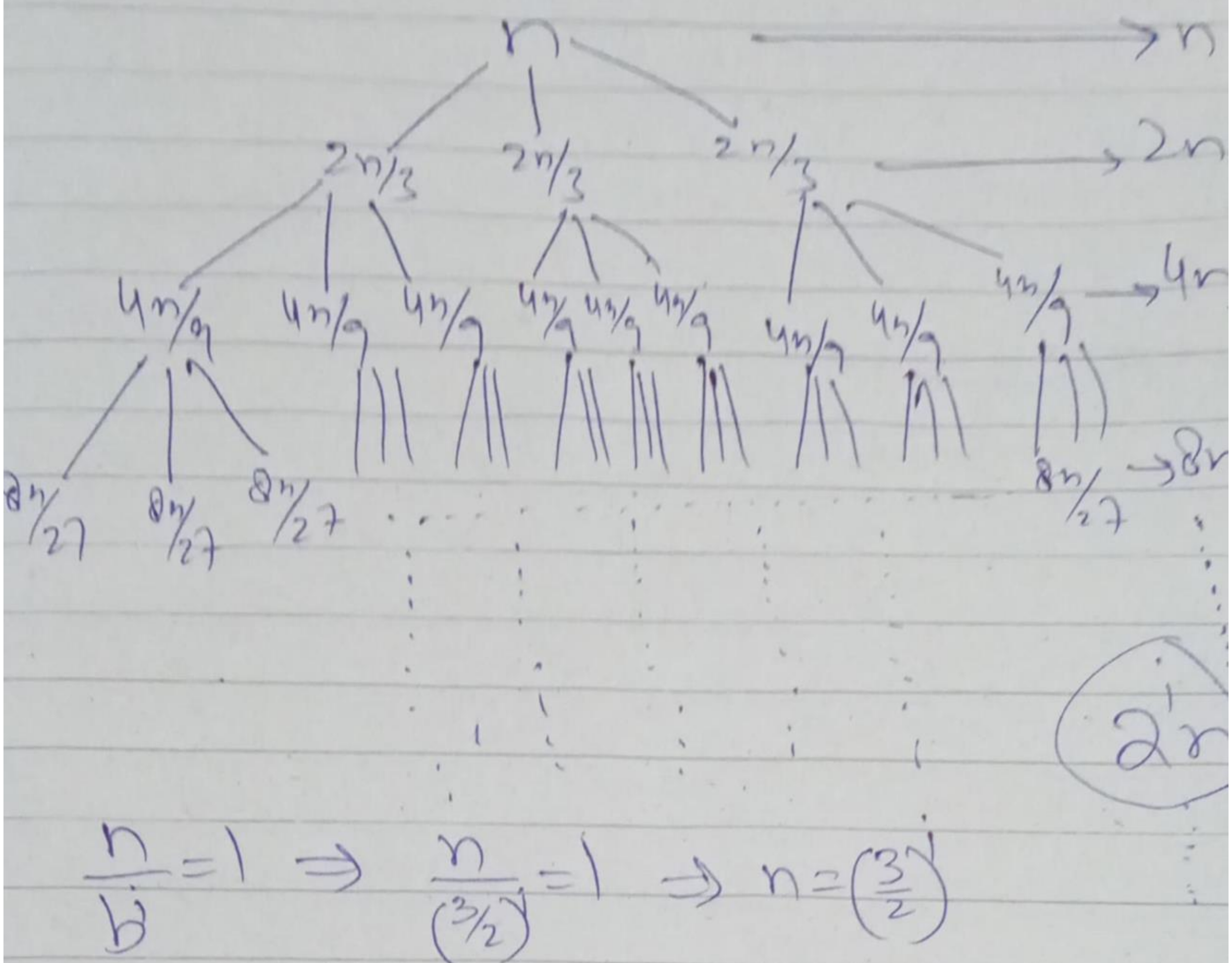
$\boldsymbol{b)\ T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + n^2}$, **Assume $T(1) = 1$**

<u>Solution:</u>

<u>Q2 part (a)</u>

Recurrence Tree

$$T(n) = 3T\left(\frac{2n}{3}\right) + n \quad , T(1) = 1$$



$$\frac{n}{b^i} = 1 \Rightarrow \frac{n}{\left(3/2\right)^i} = 1 \Rightarrow n = \left(\frac{3}{2}\right)^i$$

$$\Rightarrow i = \log_{3/2} n$$

$$\boxed{\sum_{i=0}^{\log_{3/2} n} 2^i n = n \sum_{i=0}^{\log_{3/2} n} 2^i}$$

$$\sum_{k=0}^{n} 2^k = 2^{k+1} - 1$$

$$= n\left(2^{(\log_{3/2} n + 1)} - 1\right)$$

$$= n\left(2^{\log_{3/2} n} \times 2 - 1\right)$$

Substitution

$$\boxed{T(n) = 3T\left(\frac{2n}{3}\right) + n \quad , T(1) = 1} \quad \text{①}$$

$$T\left(\frac{2n}{3}\right) = 3T\left(\frac{4n}{9}\right) + \frac{2n}{3} \quad \text{②}$$

Putting ② in ①, we get

$$T(n) = 3\left[3T(4n/9) + 2n/3\right] + n$$

$$T(n) = 9T\left(\frac{4n}{9}\right) + \frac{6n}{3} + n \Rightarrow \boxed{T(n) = 9T\left(\frac{4n}{9}\right) + 3n} \quad \text{③}$$

$$T\left(\frac{4n}{9}\right) = 3T\left(\frac{8n}{27}\right) + \frac{4n}{9} \quad \text{④}$$

Putting ④ in ③, we get

$$T(n) = 9\left[3T\left(\frac{8n}{27}\right) + \frac{4n}{9}\right] + 3n$$

$$\boxed{T(n) = 27T\left(\frac{8n}{27}\right) + 7n} \quad \text{⑤}$$

From ①, ③ & ⑤, pattern is

$$\boxed{T(n) = 3^i\, T\left(\frac{2^i n}{3^i}\right) + (2^i - 1)n} \quad \text{⑥}$$

$$\frac{2^i}{3^i} n = 1 \Rightarrow i = \log_{3/2} n$$

Putting values in ⑥, $T(n) = 3^{\log_{3/2} n}\, T(1) + \left(2^{\log_{3/2} n} - 1\right)n$

$T(n) = n^{\log_{3/2} 3}$

Q2 part (b)

$T(n) = T(n/4) + T(n/2) + n^2$

Ignore $T(n/4)$ as $T(n/2)$ will be dominant

$T(n) \Rightarrow T(n/2) + n^2 \rightarrow (1)$

$\Rightarrow T(n/2) = T\left(\frac{n}{4}\right) + \frac{n^2}{4}$

Plug $T(n/2)$ in eq (1)

$\Rightarrow T(n) = T\left(\frac{n}{4}\right) + \frac{n^2}{4} + n^2 \rightarrow (2)$

$\Rightarrow T(n/4) = T\left(\frac{n}{8}\right) + \frac{n^2}{16}$

Plug $T(n/4)$ in eq − (2)

$\Rightarrow T(n) = T\left(\frac{n}{8}\right) + \frac{n^2}{4} + \frac{n^2}{16} + n^2$

Generalizing $\rightarrow \frac{n}{2^k} \Rightarrow \frac{n}{2^k} = 1$ for base case $k = \log_2 n$

$n^2\left[1 + \frac{1}{4} + \frac{1}{16}\right] \Rightarrow n^2\left(\frac{1}{1 - 1/4}\right)$

$T(1) = 1$

$\Rightarrow 1 + \frac{4}{3} n^2 \Rightarrow O(n^2)$

Recurrence tree 2(b)

cn2

/ \

T(n/4) T(n/2)

If we further break down the expression T(n/4) and T(n/2),

we get following recursion tree.

cn2

/ \

c(n2)/16 c(n2)/4

/ \ / \

T(n/16) T(n/8) T(n/8) T(n/4)

Breaking down further gives us following

cn2

/ \

c(n2)/16 c(n2)/4

/ \ / \

c(n2)/256 c(n2)/64 c(n2)/64 c(n2)/16

/ \ / \ / \ / \

To know the value of T(n), we need to calculate sum of tree

nodes level by level. If we sum the above tree level by level,

we get the following series

T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...

The above series is geometrical progression with ratio 5/16.

To get an upper bound, we can sum the infinite series.

We get the sum as (n2)/(1 - 5/16) which is O(n2)

Question # 3 **[2 marks]**

Consider the given recurrence relation. You need to apply **Substitution Guess and Test method** on both guess one by one to find correct one.

$$\boldsymbol{T(n) = 8T\left(\frac{n}{2}\right) + n^2}$$

Guess 1 : $\boldsymbol{T(n) = O(n^2 logn)}$ **,** **Guess 2 :** $\boldsymbol{T(n) = O(n^3 logn)}$

Solution:

Solution for Guess 1:

Prove (Inductive Hypothesis): **T(n) <= cn²logn** for some constant c>0

Assume (Inductive Step): $T\left(\frac{n}{2}\right) \le c\,\frac{n^2}{4}\log(\frac{n}{2})$

$$T(n) = 8T(n/2) + n^2$$
$$T(n) \le 8 \cdot c\,\frac{n^2}{4}\log(\frac{n}{2}) + n^2$$
$$T(n) \le 2cn^2\,(logn - \log 2) + n^2$$
$$T(n) \le 2cn^2\,logn - 2cn^2 + n^2$$

Thus $T(n) \le 2cn^2\,logn - 2cn^2 + n^2 <= \mathbf{cn^2logn}$

WRONG: Since, $2cn^2 logn$ is always greater than $cn^2\,logn$, (while ignoring $2cn^2$ as it is asymptotically smaller)

Solution for Guess 2:

Prove (Inductive Hypothesis): **T(n) <= cn³logn** for some constant c>0

Assume (Inductive Step): $T\left(\frac{n}{2}\right) \le c\,\frac{n^3}{8}\log(\frac{n}{2})$

$$T(n) = 8T(n/2) + n^2$$
$$T(n) \le 8 \cdot c\,\frac{n^3}{8}\log(\frac{n}{2}) + n^2$$
$$T(n) \le cn^3\,(logn - \log 2) + n^2$$
$$T(n) \le cn^3\,logn - cn^3 + n^2$$

Thus $T(n) \le cn^3\,logn - cn^3 + n^2 <= \mathbf{cn^3logn}$

Note that the negative value will be higher (from $-cn^3$ than the n^2), therefore proved.

Question # 4 **[0.5+1.5=2 marks]**

Consider below given bubble sort algorithm :

```
BUBBLESORT(A)
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

a) Let *A*’ denote the output of BUBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that:

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n]$$

where *n* = *A.length*. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

b) Prove below given loop invariant property for inner loop (lines 2 to 4)

Loop Invariant Property of inner loop: At the start of each iteration, the position of the smallest element of A[i…….n] is at most *j*

Solution

(a)

We need to prove that A' contains the same elements as A, which is easily seen to be true because the only modification we make to A is swapping its elements, so the resulting array must contain a rearrangement of the elements in the original array.

(b)

The **for** loop in lines 2 through 4 maintains the following loop invariant: At the start of each iteration, the position of the smallest element of $A[i..n]$ is at most j. This is clearly true prior to the first iteration because the position of any element is at most $A.length$. To see that each iteration maintains the loop invariant, suppose that $j = k$ and the position of the smallest element of $A[i..n]$ is at most k. Then we compare $A[k]$ to $A[k-1]$. If $A[k] < A[k-1]$ then $A[k-1]$ is not the smallest element of $A[i..n]$, so when we swap $A[k]$ and $A[k-1]$ we know that the smallest element of $A[i..n]$ must occur in the first $k-1$ positions of the subarray, the maintaining the invariant. On the other hand, if $A[k] \geq A[k-1]$ then the smallest element can't be $A[k]$. Since we do nothing, we conclude that the smallest element has position at most $k-1$. Upon termination, the smallest element of $A[i..n]$ is in position i.

<u>Question # 5</u> **[4 marks]**

Given a sorted array, integer ***k*** and target ***t*** as input, the objective is to find ***k*** closest elements to ***t*** in the array

For example:

Input array = [17,18,20,25,30], ***k*** = 2 , ***t*** =16

Output = [17,18]

If the target is smaller than all the elements in the array then return first ***k*** elements, likewise, if target is greater than all the elements in the array then return the last ***k*** elements. The ordering of returned numbers should be maintained as in original array. Design algorithm for the above scenario that takes no longer than ***O(k + log n)*** time.

Solution:

Find the index where the target can be placed using binary search in O (log(n)) time and compare the elements around index in O(k) time.

The overall complexity is O(log(n) + k)

```
# Function to find the cross over point
# (the point before which elements are
# smaller than or equal to x and after
# which greater than x)
def findCrossOver(arr, low, high, x) :

    # Base cases
    if (arr[high] <= x) : # x is greater than all
        return high

    if (arr[low] > x) : # x is smaller than all
        return low

    # Find the middle point
    mid = (low + high) // 2 # low + (high - low)// 2

    # If x is same as middle element,
    # then return mid
    if (arr[mid] <= x and arr[mid + 1] > x) :
        return mid

    # If x is greater than arr[mid], then
    # either arr[mid + 1] is ceiling of x
    # or ceiling lies in arr[mid+1...high]
    if(arr[mid] < x) :
        return findCrossOver(arr, mid + 1, high, x)

    return findCrossOver(arr, low, mid - 1, x)
```

```python
# This function prints k closest elements to x
# in arr[]. n is the number of elements in arr[]
def printKclosest(arr, x, k, n) :

    # Find the crossover point
    l = findCrossOver(arr, 0, n - 1, x)
    r = l + 1 # Right index to search
    count = 0 # To keep track of count of
              # elements already printed

    # If x is present in arr[], then reduce
    # left index. Assumption: all elements
    # in arr[] are distinct
    if (arr[l] == x) :
        l -= 1

    # Compare elements on left and right of crossover
    # point to find the k closest elements
    while (l >= 0 and r < n and count < k) :

        if (x - arr[l] < arr[r] - x) :
            print(arr[l], end = " ")
            l -= 1
        else :
            print(arr[r], end = " ")
            r += 1
        count += 1

    # If there are no more elements on right
    # side, then print left elements
    while (count < k and l >= 0) :
        print(arr[l], end = " ")
        l -= 1
        count += 1

    # If there are no more elements on left
    # side, then print right elements
    while (count < k and r < n) :
        print(arr[r], end = " ")
        r += 1
        count += 1

# Driver Code
if __name__ == "__main__" :

    arr =[12, 16, 22, 30, 35, 39, 42,
              45, 48, 50, 53, 55, 56]

    n = len(arr)
    x = 35
    k = 4

    printKclosest(arr, x, 4, n)
```

Appendix

Masters Theorem 4th Case

$$\text{If } f(n) \in \Theta(n^{\log_b a} \log^k n) \text{ for some } k \geq 0 \text{ then}$$
$$T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$$

$\sum_{k=0}^{\infty} ar^k = \frac{a}{1-r}$ (if $r<1$)

$\sum_{k=0}^{n} 2^k = 2^{k+1} - 1$