National University of Computer & Emerging Sciences, Karachi
Fall-2021 Department of Computer Science
Mid Term-2
22 November 2021, 10:30 AM – 12:30 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 **6 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: 120 minutes. **Max Marks: 17.5**

Question # 1 **[0.5*4 = 2 marks]**

Are these following statements True or False? Prove your answer by computing the values of n_0, c_1, c_2 or by contradiction.

a) $\mathbf{2^{2n} = O\ (2^n)}$
b) $\mathbf{n^2\ (\ 2n - 5) = \Theta\ (\ n^3)}$
c) $\mathbf{n^2 + 2n + 10\ = \Omega\ (n)}$
d) $\mathbf{\sqrt{1000n^2 + 100n} = \Theta(\ n^2)}$

Solution:

Marking Scheme: 0.5 for correct answer and solution, 0.25 for incorrect answer but good attempt. 0.25 for correct answer but wrong solution. 0 for incorrect and totally incorrect attempt

1. $2^{2n} = O\ (2^n)$

Answer: False

Assume there exist constants n_0, c_1

$$2^{2n} \leq c.2^n$$
$$then\ 2^{2n} = \ 2^n\ .2^n\ \leq c.2^n$$

$$2^n \leq c$$

But no constant is greater than all 2^n, and so the assumption leads to a contradiction.

2. $n^2\ (2n - 5) = \Theta\ (n^3)$

Answer: True

$c_1\ n^3 \leq 2n^3 - 5n^2 \leq c_2\ n^3$

First Find value of c_2 *Proving Big O Bound*

$2n^3 - 5n^2 \leq c_2\ n^3$

When we select c_2 = 2 above equation will hold for all n.

Now find value of c_1 *Proving Big* Ω *Bound*

$$c_1\ n^3 \leq 2n^3 - 5n^2$$

Divide n^3 *both sides so*

$$c_1 \leq 2 - \frac{5}{n}$$

When $n = 3$

$$c_1 \leq 2 - \frac{5}{3} \leq \frac{1}{3}$$

$$c_1 = \frac{1}{3}$$

So $\boldsymbol{n_0 = 3}$, $\boldsymbol{c_1 = \frac{1}{3}}$, $\boldsymbol{c_2 = 2}$

3. $n^2 + 2n + 10 = \Omega\ (n) \rightarrow$

Answer: True

there exist constants n_0, c_1 such that $n^2 + 2n + 10 \ \ \geq \Omega\ (n)$
$As\, n^2 \geq n,$

$so\ c = 1\ and\ n_0 = 1$

4. $\boldsymbol{\sqrt{1000n^2 + 100n} = \Theta(\ n^2)}$

Answer: False

If we solve square root, equation will be linear so $(n2)$ is not possible

$$c_1\ \ n^2\ \leq\ \sqrt{1000n^2 + 100n}\ \leq\ c_2\ \ n^2$$

$c_1\ \ n^2\ \leq\ \sqrt{1000n^2 + 100n} \rightarrow c_1\ \ n^2 \leq 31.6\ n + 10\ \sqrt{n} \rightarrow$ No constant multiplier can make $n^2 \leq n,$ and so the assumption leads to a contradiction.

Question # 2 **[1 + 1 = 2 marks]**

Compute the time complexity for both below mentioned algorithms separately. Show all the steps.

(a)

```
 public static void main(String[ ] args) {

for (int i=1 ; i<=n ; i++) {
          for (int j=1; j<n; j=j+i) {
                    System.out.println("*");
                    break;
                    }
           }
}
```

(b)

```
 public static void main(String[ ] args) {
 int n=100;
for (int i=n/2 ; i<=n ; i++) {
          for (int j=1; j<n; j=j*2) {
                    for (int k=1; k<=n; k=k*2) {
                        System.out.println("*");
                    }
           }
}}
```

Solution:
a)

1 for correct answer. 0.5 for either correct complexity of first loop or 2nd loop

first loop will run N times

second will break out after every first iteration. so it will run 1 time

so time complexity is O(n)

b)

1 for correct answer. 0.25 for first loop i.e. n/2. 0.5 more for either correct logn of 2nd or 3rd loop

first loop will run n/2 times

second and third loop as per above example will run logn times

so time complexity = n/2*logn*logn = $O(nLog^2n)$

<u>Question # 3</u> **[2.5 +1.5 = 4 marks]**

Prove by contradiction below given Lemma. Also give a small example
Given: Let **G** be a weighted graph, and **S** be a subset of its edges. Let **T** be the *MST* of **G**, when the edges of **S** are given weights lower than those of any other edge in the graph.
Prove that No matter how the edge weights in **S** are changed, the *MST* of **G** will always contain the edges in **T-S**
b) An edge in an undirected graph G is a bridge if removing it disconnects the graph. Design algorithm to find all the bridge edges.

3(a) Solution:

Proof: Consider changing the weights of the edges in S, one by one, from the weights in T to the new desired weight. At each such change, either the MST will not change, or the changed edge will leave the MST and some other edge will replace it. Therefore, the edges in $T - S$ will remain in the MST. □

Marking Scheme: **2.5 for correct answer. 1 to 1.25 points for showing T or S graph with higher values in S graph. 0.5 for just drawing MST tree from G**

Now any example can be given by taking any graph..

3(b) Solution :

Marking Scheme: 1.5 points for correct answer. 1 point for writing BFS, SFS or traverse without remove / add statements. 0-1 point according to answer

Algorithm for bridge edges :

A simple approach is to one by one remove all edges and see if removal of an edge causes disconnected graph. Following are steps of simple approach for connected graph.
1) For every edge (u, v), do following
…..a) Remove (u, v) from graph
…..b) See if the graph remains connected (We can either use BFS or DFS)
…..c) Add (u, v) back to the graph.
This will be done for all edges

<u>Question # 4</u> **[3 marks]**

Consider the following instance of the 0/1 knapsack problem

Item	1	2	3	4	5

Benefit	15	35	10	9	9
Weight	4	12	4	4	5

The maximum allowable total weight in the knapsack is W = 9.
Find an optimal solution for the above problem with the weights and benefits above using Dynamic Programming. Be sure to state both the value of the maximum benefit that you obtain as well as the item(s) that you need to obtain this benefit. Show all steps.

Solution:

Marking Scheme: 3 Points for correct answer. 2.5 points for V table and 0.5 points for keep table. Marks to be deducted if wrong values in there and steps are not shown at all.

			Weight											
vi	wi	Index	0	1	2	3	4	5	6	7	8	9	10	11
9	5	5	0	0	0	0	15	15	15	15	25	25	25	25↓
9	4	4	0	0	0	0	15	15	15	15	25	25	25	25↓
10	4	3	0	0	0	0	15	15	15	15	25	25	25	25
35	12	2	0	0	0	0	15	15	15	15↓	15	15	15	15
15	4	1	0	0	0	0	15	15	15	15	15	15	15	15
		0	0	0	0	0	0	0	0	0	0	0	0	0

The maximum benefit is 25
The selection array representing the indexes of items to select is: (The first element is from base case 0)
S = [0,1,0,1,0,0]

Another Solution

V	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	15	15	15	15	15	15
2	0	0	0	0	15	15	15	15	15	15
3	0	0	0	0	15	15	15	15	25	25
4	0	0	0	0	15	15	15	15	25	25
5	0	0	0	0	15	15	15	15	25	25

Max Benefit = 25
Keep Table

K	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1	1	1	1

2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0

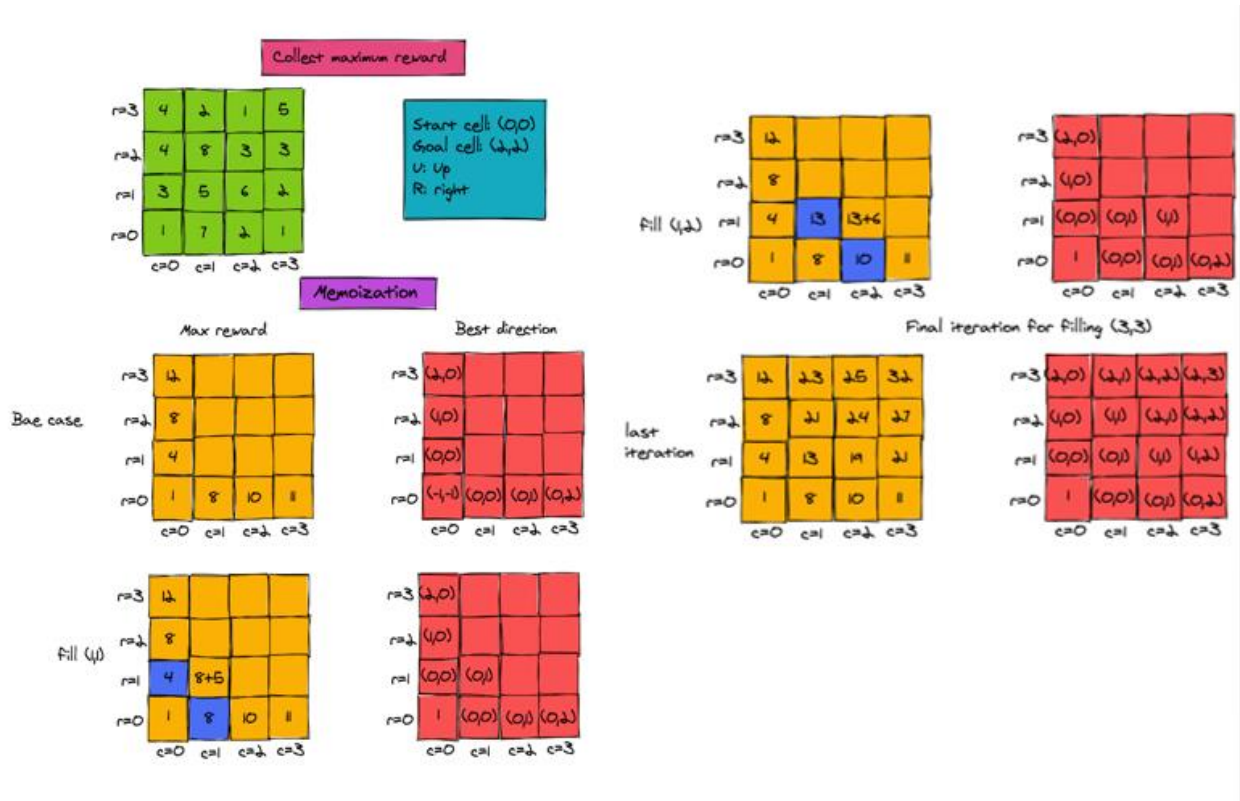
Keep [5,9] = 0
Keep [4,9] = 0
Keep [3,9] = 1; Item 3 is selected
9 – 4 = 5
Keep [2,5] = 0
Keep [1,5] = 1; Item 1 is selected
{1,3} are the items in the knapsack with total weight of 8

Question # 5 **[4 marks]**

Suppose we have an m x n grid ("m" rows and "n" columns), where each cell has a "reward" associated with it. Let's also assume that there's a robot placed at the starting location, and that it has to find its way to a "goal cell". While it is doing this, it will be judged by the path it chooses. We want to get to the "goal" via a path that collects the maximum reward. The only moves allowed are "up" and "right". Give an algorithm that utilizes dynamic programming to find the required path.

Collect Maximum Reward				
	Column 0	Column 1	Column 2	Column 3
Row 3	4	2	1	5
Row 2	4	8	3	3
Row 1	3	5	6	2
Row 0	1	7	2	1
Start Cell: (0, 0) Goal Cell: (3, 3) Possible Moves: U, R				

Solution:



Input: Matrix w of dimensions m * n containing reward for each cell,
Start cell coordinates: (0, 0)
Goal cell coordinates: (m, n)
Output: Path found and the accumulated reward on that path
// (r, c) denotes (row, column) coordinates
1. total = 0
2. (r, c) = (0,0)
3. while (r, c) != goal
 a. total = total + w[r, c]
 b. print (r, c) // print coordinates of the cell [2 Points until here]
 // check if we are in top row
 c. if (r == m) [2 Points rest accordingly]
 c = c+1 // go right. no other choice
 // check if we are in rightmost col
 d. if (c == n)
 r = r+1 // go up, no other choice
 // select either up or right move
 e. if w[r+1, c] > w[r, c+1]
 r = r+1 // move up
 else
 c = c+1 // move right
4. Print goal
5. return total // return accumulated reward

Question # 6 [1+1.5=2.5 marks]

You are given a list of ***n-1*** unsorted integers and these integers are in the range of ***1 to n***. There are no duplicates in the list. One of the integers is missing in the list.
a) Design algorithm to find the missing integer in **O(n)** time using linear sorting
b) Design algorithm to find the missing integer in **O(n)** time with out using linear sorting

Example **Input:** arr[] = {1, 2, 4, 6, 3, 7, 8} **Output:** 5

Solution:

a) For **part a**, you can apply linear sorting algorithm and then check difference of consecutive elements to find missing number. If this difference is greater than 1 then missing number is there and you may find that missing number by adding 1 in previous element.

[1 Point for correct description]
[0-0.5 points for some attempt]

b) Now for **part (b)**

[1.5 points] without creating temp array and sort it
[1 Point] with creating some temp sorted array and then apply logic
[0-1] for some attempt accordingly

```python
# getMissingNo takes list as argument
def getMissingNo(A):
    n = len(A)
    total = (n + 1)*(n + 2)/2
    sum_of_A = sum(A)
    return total - sum_of_A


# Driver program to test the above function
A = [1, 2, 4, 5, 6]
miss = getMissingNo(A)
print(miss)
```