Monsoon 2022 Graduate Algorithms: Mid-sem *Max: 90 points*
Due: **18 Oct** *Duration: 120 mins*

Name: **Roll No.:**

15% if you write **"I don't know"** for any (sub)question, with a cap of at most 10 marks.
Write answers to Q1-Q5 in the question paper itself.

Q1 [2+1+1+1=5 points,E] 1. Fill in the three blanks below to complete the algorithm that recursively computes size of the smallest vertex cover of an unweighted binary tree. The size of a cover is the number of vertices in that cover.

```
def VC(node r): // compute optimal vertex cover size for subtree rooted at r
    if r is null: return 0

    // vertex cover values that are computed are memoized
    if r.vc is not null: return r.vc // avoid recomputation of memoized values

    // compute optimal vertex cover size when r is part of the cover

    vc_with_r = [1 + VC(r.left) + VC(r.right)]
                (1 line recursive code to compute this value)

    // compute optimal vertex cover size when r is not part of the cover
    vc_wo_r = 0
    if r.left is not null:
        vc_wo_r += [1 + VC(r.left.left) + VC(r.left.right)]
                   (1 line recursive code for left subtree of r)
    if r.right is not null:
        vc_wo_r += [1 + VC(right.left) + VC(right.right)]
                   (1 line recursive code for right subtree of r)

    // optimal vertex cover size is the best of two cases
    vc = min (vc_with_r, vc_wo_r)
    // memoize the value
    r.vc = vc
    return r.vc
```

2. What is the asymptotic running time of `VC(root of tree T)`? Express the running time in terms of any/all of these: number of nodes n, smallest depth of any leaf s, largest depth of any leaf h, number of leaf nodes l. **Ans:** [O(n)]

Q2 [5 points,E] Solve the recurrence to derive an upper bound on $T(m, n)$.

$$T(m, n) \leq 2T(m/2, n/2) + 2mn, \quad T(1, n) = 6n, \quad T(m, 1) = 6m.$$

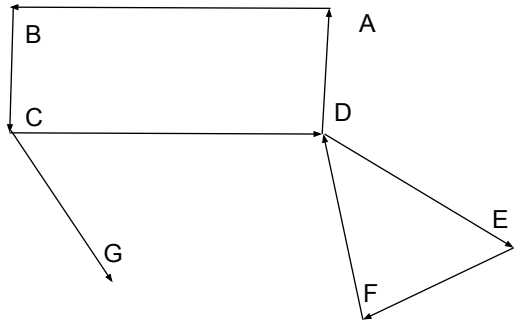
Use induction to prove that $T(m, n) \leq cmn$ holds for any $c \geq 6$.

Q3 [5 points,E] Below (left) you see the filled-memo (table) required to compute the edit distance from the string x="INTERIOR" to the string y="EXECUTE". The rows are indexed with $0 \dots 7$ (for EXECUTE), and the columns are indexed with $0 \dots 8$ (for INTERIOR). Fill the $h^8(\cdot,\cdot)$ entries in the right table.

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

	0	1	2	3	4	5	6	7	8
0	∞	∞	∞	∞	0	0	0	0	0
1	∞	∞	∞	∞	1	1	1	1	1
2	∞	∞	∞	∞	2	1	1	1	1
3	∞	∞	∞	∞	3	3	1,3	1,3	1,3
4	∞	∞	∞	∞	4	3	3	1,3	1,3
5	∞	∞	∞	∞	5	3,4	3	1,3	1,3
6	∞	∞	∞	∞	6	3,4,5	3,4	3	1,3
7	∞	∞	∞	∞	7	7	3,4,5,7	3,4	3

Q4 [3+2=5 points,E] (a) Perform a DFS on this graph, and for each node in it, write the pre/start-time and low-number of every vertex (write them beside each node). During DFS, if there are multiple vertices to select from, always select them in the alphabetical order.



A:1/1, B:2/1, C:3/1, D:4/1, E:5/4, F:6/4, G:7/7

(b) List all the strongly connected components in the above graph as would be returned by Tarjan's algorithm following the above DFS. For each component, list the vertices in the component and explicitly mark the root vertex in that component.

Strongly connected components = $\{A, B, C, D, E, F\}$, $\{G\}$

Q5 [5 points,E] Fill-in the blanks to design a backtracking algorithm that returns a subset of a set $X[1 \dots k]$ of integers whose sum is a target t, or returns `None` if no such subset exists. You cannot use any global variable or additional subroutines. Additional data structures are discouraged.

```
def SubsetSumConstruct(set X[1 ... k], target t):
    if t = 0: return empty-set {}
    if t < 0 or k=0: return None

    // fill-in
    S = SubsetSumConstruct(X[1...n-1],t) // explore solutions without Xn
    if S is not None: return S
    S = SubsetSumConstruct(X[1...n-1], t-Xn) // explore solutions with Xn
    if S is not None: return S U {Xn}
    return None
```

It is also possible to branch based on X1 instead of Xn.

Q6 [10 points,M] Consider an n-vertex graph $G = \langle V = \{v_1, \ldots, v_n\}, E\rangle$ where $w(u,v)$ denotes the positive weight of the edge $u \to v$ ($w(u,v) = \infty$ denotes that there is no edge between u and v). Recall that the Floyd-Warshall algorithm computes the function values $g(u,v,k)$ =(shortest-path) distance from $u \rightsquigarrow v$ using intermediate vertices only from $\{1,\ldots,k\}$. Now, consider the following problem on directed weighted graphs: $Pred(u,v,k) =$

$=$ predecessor of v on some shortest-path from u whose intermediate vertices must be from $\{1,\ldots,k\}$, or
$=$ NIL, if no such path exists.

Suppose you are given a 3D-array $g(u,v,k)$ filled-up for all $u \in V$, $v \in V$, and $k = 1 \ldots n$. Explain how you will fill-up the values $Pred(u,v,k)$ for all u,v,k by answering these points.

- **Base cases to be filled-up**
- $Pred(u,v,0) = v$ if $w(u,v) < \infty$ or if $g(u,v,0) < \infty$, = None otherwise. If $u = v$, $Pred(u,v,0)$ = None. $k = 1$ would be a complicated base case.
- **Recursive formula to fill any** $Pred(u,v,k)$
- $= Pred(u,v,k-1)$ if $g(u,v,k) = g(u,v,k-1)$, $= Pred(k,v,k-1)$ if $g(u,v,k) = g(u,k,k-1) + g(k,v,k-1)$
- **Write an algorithm that uses the** $Pred()$ **array to print the list of edges in some shortest-path from** $s \rightsquigarrow t$**. You can use multiple subroutines. Add brief explanations to your algorithm.**

```
def Print(s,t): return PrintPath(s,t,n);
def PrintPath(s,t,k): v = Pred(s,t,k); PrintPath(s,v,k); print ("v→t");
```

- **Analyse the time-complexity of your algorithm (use of a recurrence is optional)**
- $O(n)$ since any shortest path has at most $n-1$ edges

Q7 [5+4+6=15 points, M] We are given a stack of n pancakes of different sizes. You can visually find out which pancake is larger than which one (and even the smallest and the largest ones), however, you are only allowed to insert a spatula below some pancake and "flip" the entire lot from the top to that pancake (as shown below).

(a) Design an efficient recursive algorithm to sort the entire stack of pancakes from the smallest diameter (at top) to the largest diameter (at bottom); your algorithm should use at most $2n$ flips. Fill-in the pseudocode below. You can use the following $O(1)$ subroutines: $Flip(D,i)$ will flip the top-i pancakes, i.e., $D' = Flip(D = [8,1,6,9,3],3)$ will return $D' = [6,1,8,9,3]$, $Min(D)$ will return the index of the minimum of an array D, and similarly, $Max(D)$ will return the index of the maximum of D.

```
// should return a sorted version of D (smallest to largest)
def SortPancakes(diameters D[1...n]):
        // Add handling of base case(s)
        if n = 1, return D
        // Try! Adding a base case for n = 2 would make the algorithm slightly more efficient.

        // Add handling of recursive case(s)
        k=position/index of the largest value in D

        D1=flip(D,k) // now largest is at the top
        D2=flip(D1,n) // now largest is at the bottom of the input pancakes

        return SortPancakes(D2[1...n-1])
```

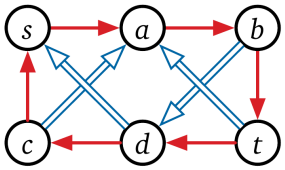
(b) Suppose that you had to that your algorithm is correct using induction. State the induction hypothesis and then prove the fact.

IH : SortPancakes(D[1...n]) sorts the top n pancakes from smallest to largest and does not involve/move any pancake below the n-th pancake. (Second part is important).
Base case: SortPanCakes(D) simply returns D when D has only 1 element. This D is correctly sorted, so the claim is true.
Proof of general case: Assume that SortPanCakes(D') correctly returns a sorted version of D' when D' has at most k-1 elements.
Now consider any D with k elements. Suppose its max, say M, appears in index i. Then, after the 1st flip, M will appear at the top, i.e., at index 1. After the second flip, M will move to the bottom, i.e., at index k. The recursive call to SortPanCakes will sort the topmost k-1 pancakes, and would not disturb the pancakes at indices k,k+1, ... Hence, after that recursive call, the top k pancakes will be sorted.

(c) Analyse the exact/precise number of "flips" used your algorithm in the worst-case in terms of n. Let $T(n)$ denote the number of flips used (in the worst-case) by `SortPancakes` on any array of size n. Explain the following.

- **Base case** $T(1) = 0$
- **Recurrence** $T(n) = T(n-1) + 2$
- **Solution** Derivation is easy. $T(n) = 2n - 2$.

Q8 [20 points,H] Suppose you are given a directed graph G where some edges are red and the remaining edges are blue. Describe an algorithm to find the shortest walk in G from one vertex s to another vertex t in which no three consecutive edges have the same colour. That is, if the walk contains two red edges in a row, the next edge must be blue, and if the walk contains two blue edges in a row, the next edge must be red. You must use graph reduction to solve this problem.

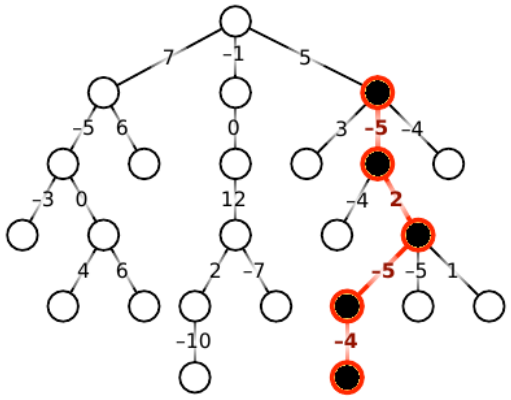


For the graph given above (single arrows are red and double arrows are blue), your algorithm should return the integer 7, because the shortest legal walk from s to t is $s \to a \to b \implies d \to c \implies a \to b \to t$.

Do not write a pseudocode. Instead, explain in the form of this template.

- **Description of graph H**
- Let n denote the number of nodes and m denote the number of edges in G. Add $8n+2$ vertices in the following manner: for every v, add 8 vertices $(v,R,R,R), (v,R,R,B), \ldots, (v,B,B,B)$. Add 2 more vertices s and t (these could have been skipped at the expense of solving multiple reachabilitity problems). The additional 3 attributes denote the colours of the last 3 edges used during any walk.
  Add $8+8+8m$ edges in the following manner: $s \to (s,\cdot,\cdot,\cdot)$, $(t,\cdot,\cdot,\cdot) \to t$, for $u \to v$ add $(u\cdot,b,c) \to (v,b,c,R)$ and for $u \implies v$ add $(u,\cdot,b,c) \to (v,b,c,B)$.
- **What problem P will you solve on H and what algorithm will you run to solve that problem**
- reachability from s to t
- **Briefly justify the correctness of this reduction, i.e, why P on H solves the original question**
- **Time complexity analysis in terms of n (number of vertices) and m (number of edges)**
- Since number of vertices and edges are $O(n)$ and $O(m)$, respectively, the complexity of BFS would be $O(n+m)$.

Q9 [20 points,H] Let T be a rooted tree with integer weights on its edges, which could be positive, negative, or zero. The weight of a path in T is the sum of the weights of its edges. Describe and analyze a dynamic programming algorithm to compute the minimum weight of any path from *a node (not necessarily the root) in T down to one of its descendants (the problem does not require the descendant to be a leaf)*. It is not necessary to compute the actual minimum-weight path; just its weight is sufficient. State your solution in terms of template shared in class. For example, given the tree shown below, your algorithm should return the number -12 (the minimum-weight downward path in this tree is shaded).



Write the answer in the form of this template.

- **Definition of subproblem**
- Define P(x) = minimum weight among paths from x to any descendant
- **Recurrence formula for subproblem, along with base case**
- If x is leaf, P(x) = 0
  If x is not a leaf, x has at least one children and best path from x must go via some children. So,
  P(x) = min of
  min { w(x,y) + P(y) : y is a children of x } // best path is from x to y to the best descendant of y
  min { w(x,y) : y is a children of x } // best path is from x to some children

  Another way of writing the above : P(x) = min w(x,y) + min P(y),0 : y is a children of x .
- **How to solve original problem using the subproblem**
- To solve the original problem, return min { P(x) : x is a node in T }
- **Memo for storing subproblem values and the order of filling them**
- Memo is T itself and P(x) is stored with the node x. Memo is filled by starting at leaf nodes and going up the tree.
- **Time complexity analysis**
- To compute P(x), number of operations is proportional to number of children of x.
  Total time to compute all P(x) : $O(\sum_x$ number of children of $x)$ = O(number of edges in T) = O(number of nodes in T).
- **Space complexity analysis**
- O(n) where n is the number of nodes in T // only store P(x) with x