Q5 [5 points] Define a directed graph G to be “nice” if it satisfies the property that either u ⇒ v or v ⇒ u (or
both) for any two vertices u, v. Let A(H) be an algorithm to detect if a directed acyclic graph (DAG) H is nice
— it returns true if H is nice and false otherwise (A behaves weirdly if H is not a DAG). Fill in the blanks
below (do not remove the blue text) to complete the design of an algorithm that returns true if its input G is
nice and false otherwise — here G is not required to be acyclic.
Assume that the vertices in G are labeled using integers (v1, v2, . . .). The Kosaraju-Sharir (KS) algorithm can
be easily modified to return the component graph of the SCCs. Add 2-3 lines to explain what your algorithm is
doing and why it is correct.
def IsNice(G): ⇒⇒G is directed graph⇐⇐
Run Kosaraju-Sharir (KS) on G
H ← strongly connected component graph returned by KS
(. . . fill the rest — A() must be used)

2

Run BFS/DFS (any reachability algorithm) on G' to find if (s,0) has a path/walk to (t,1). Other alternatives
could be a path from (s,1) to (t,2), and a path from (s,2) to (t,0). It is not required to try all three approaches.
Size of G': |V'| = 3V, |E'| = 3E. Complexity = O(|V'|+|E'|) = O(|V|+|E|) in terms of the parameters of G.
Claim: There is a path from (s,0) to (t,1) in G' if and only if there is a path from s to t in G.
Proof of claim:
<= Suppose there is a path from s -> v1 -> v2 -> ... -> vk -> t where vi's belong to V, may be repeated and
k+1 = 1 (mod 3). Then the following would be a path in G': (s,0) -> (v1,1) -> (v2,2) -> ... (vi,i(mod 3))
-> (vk,k (mod 3)) -> (t,(k+1) (mod 3)) as per the rule for constructing edges in G'.
Since k+1=1 (mod 3), the path ends at (t,1).
=> Suppose there is a path from (s,0) to (t,1) in G'. By the rule of constructing edges in G', the path must be
like: (s,0) -> (v1,1) -> (v2,2) -> ... (vk, k mod (3)) -> (t,1) and therefore, k+1=1 (mod 3). Thus, the path length
k+1 leaves a remainder of 1 when divided by 3.
Algorithm + justification : 6, complexity : 2

Run A(H).

If A(H) returns true:
return true
Else
return false

By property of KS, H is a directed acyclic graph.
Claim: If H is nice, then G is nice.
Proof: Suppose H is nice. We have to show that G is nice, i.e., for any two vertices u and v in G, either u has a path to v or v has a path to u (or both).
Case : u and v belong to the same SCC of G. Then, by the property of SCC, u has a path to v.
Case : u belongs to SCC A and v belongs to SCC B which is different from A. Since H is nice, there is either
a path from A to B in H or a path from B to A in H. Now u has a path to and from every vertex in A and
similarly, v has a path to and from every vertex in B. Applying this argument for the SCC on the path
between u and v, either u has a path to v or v has a path to u (or both). // Formal proof will need a few
// more steps.
Claim: If G is nice, then H is nice.
Proof: Take any two vertices a and b in H, corresponding to SCC A and B in G. Take any vertex u in A
and v in B. Since G is nice, without loss of generality, assume that u has a path to v. Now tracing the
components that the vertices on this path lies in, we will see that there is a path from a to b in H as well.
// Formal proof will need a few more steps

The KNAPSACK Problem:

You are given a collection of objects, where each object X has:
- A value, $X.value$
- A weight, $X.weight$

You need to pack a knapsack with a maximum weight capacity of W. The goal is to select a subset of objects such that:
1. The total weight of the selected objects is at most W, and
2. The total value of the selected objects is maximized.

Assumptions:
- If the weights are floating-point numbers or large integers, the problem is NP-hard.
- However, if all weights and W are small positive integers, there exists an efficient solution using dynamic programming.

Tasks:
1. Design an efficient dynamic programming algorithm to solve the problem under the assumption that the weights and W are small positive integers.
2. Analyze the running time of your algorithm as a function of n (the number of objects) and W (the knapsack capacity).
3. Write the algorithm in a way that allows you to recover the actual set of selected objects, not just the optimal value.
4. Explain how to recover the optimal set of objects from the algorithm's output.

Let
D
[
i
] be the maximal value of any subset whose total weight is exactly
i
; and let
P
[
i
]
be the last element added to that set.
The algorithm can then be written as follows

```
Knapsack(C: collection of objects; W: maximal weight) {
for (i = 1 to W) {  /* Initialization */
D[i]=0;
}
for (X in C)
for (j = W downto X.weight+1) {
New = D[j-X.weight] + X.value;
/* Combine the set adding up to j-X.weight with X */
if (New > D[j]) {
D[j] = New;
P[j] = X;
} endif
} endfor endfor
return D,P.
}
FindBestSet(D,P) {
k = argmax_{i} D[i];  /* Index of the largest element of D */
return FBS1(k,P);
}
FBS1(k,P) { /* Using P, recover the best set whose weights add up to k */
if (k == 0) return emptyset
else return union( { P[k] }, FBS(k-P[k].weight,P)
```

Q6 [10 points] Let G = (V, E) be a directed graph. Define subtree(x) for any vertex x as the set of vertices
that are visited if we call DF S(x) in G (before calling DF S() on any other vertex). Either prove part (A) or
part (B) by completing one of the proofs below. Comment the part that you will not prove but in the part that
you will prove, do not remove the blue text. I use u $\rightarrow$ v to denote that there is a path from u to v and u $\rightarrow$ v to
denote that there is no path from u to v.
(A) Suppose it holds that there is a path from either u $\rightarrow$ v or v $\rightarrow$ u or both for every pair (u, v) of vertices.
Then, there must be some vertex x such that x has a path to every other vertex.
Proof of (A): We will do a proof by contradiction by assuming that there exists no x such that every vertex
is reachable from x. Choose a vertex w $\in$ V such that subtree(w) has the maximum number of vertices — if there
are multiple such vertices, choose any one arbitrarily. (Now arrive at a contradiction involving w)

(B) Suppose it holds that for every vertex v there is some u $\neq$ v such that v $\rightarrow$ u. Then, there must be a pair
of vertices (x, y) such that x $\rightarrow$ y and y $\rightarrow$ x.
Proof of (B): Let w $\in$ V be the vertex such that subtree(w) has the maximum number of vertices — if there
are multiple such vertices, choose any one arbitrarily. (Now arrive at a contradiction involving w)

Q7 [15 points] P is a set of n points in 2D plane given by their X and Y coordinates, i.e., the i-th point has
coordinates (xi, yi). A point (xi, yi) is said to be an upper-corner point if for every other point (xj , yj), either
xi > xj or yi > yj (or both). For simplicity assume that no two points in P share the same X or Y coordinates.

Describe and analyze a divide and conquer algorithm to compute the number of upper-corner points in P in
O(n log n) time. For example, your algorithm should return the integer 2 for the above picture. Include a brief
description of your algorithm, why it is correct and analyse its time complexity.
Ans:

def Solve(set of points P): // return number of UC points Another way is to design Solve to return the list of
m = compute median of x coordinates of P UC points.
L = points in P with x-coordinates < m def SolveList(P):
R = points in P with x-coordinates >= m L,R : divide points in half according to x-coord
y-max = maximum y-coordinate of any point in R L1 = SolveList(L)
remove points in L whose y-coordinate is less than y-max L2 = SolveList(R)
n1 = Solve(L) m = number of points in L1 that have
n2 = Solve(R) y-coord below the largest y-coord in L2
return n1 + n2 return |L1| + |L2| - m
Complexity: T(n) <= 2T(n/2) + O(n) = O(n log n)
Use linear time median + linear scans to implement Solve/SolveList

x is upper corner := there is no point that is

top-right of x Algorithm: 11
Complexity: 4
Let UC(P) be the set of upper corner points of P. Then, points in UC(R) belong to UC(P). But not all points in
UC(L) may belong to UC(P). In particular, q in L belongs to UC(P) iff there is no point in R with y-coord > q.y.
Based on the assumption, there is some y that is not reachable from w, i.e., y is not in subtree(w). Since y
is not reachable from w, and based on the statement of the claim, w must be reachable from y. So w is in
subtree(y). So, all the nodes in subtree(w) are in subtree(y). So, subtree(y) has more nodes than in subtree(w).
This contradicts the claim that w has the largest size of subtree().

For the sake of contradiction, assume that for every pair (x,y), either x has a path to y or y has a path to x
or both. Based on the statement of the lemma, there is some u that is not reachable from w. Thus,
w must be reachable from u. That is, w belongs to subtree(u) and so every node in subtree(w) belongs
to subtree(u). Thus subtree(u) has more nodes compared to subtree(w) which contradicts the choice of w
as the node with the largest subtree.

[10 points] Suppose you are given a directed graph G = (V, E) and two vertices s and t. Construct a new
directed graph G$\hat{}$ = (V $\hat{}$
, E$\hat{}$
) in the following manner. V $\hat{}$ = V × {0, 1, 2}, i.e., for every u ∈ V , there exists three

vertices labeled (u, 0),(u, 1),(u, 2) in V ﷺ

. For every edge u → v in E, add edges (u, 0) → (v, 1), (u, 1) → (v, 2),

and (u, 2) → (v, 0) in Eﷺ
.

We want to know if there is a path from s to t in G whose length when divided by 3 leaves a remainder of
1. Solve this problem by running an appropriate algorithm on Gﷺ

. Explain what algorithm will you run and why
is your approach towards solving the original problem a correct approach. Derive the time complexity of your
approach.
Ans: