Monsoon 2020 Graduate Algorithms: Mid-sem *Max: 80 points*
Due: **Friday 16 October** *Duration: 2880 mins*

Name:Debajyoti Bera **Roll No.:PHD14000ABCDABCDABCD**

Put answers in the question paper itself.
Attach scanned images of rough sheets.
You will be awarded 20% if you write "I don't know" for any question.
All questions require some explanation; justify briefly using 2-3 lines if not specified.

Q1 [3 points] Suppose $P(x)$ is a degree-3 polynomial and $Q(x)$ is a degree-4 polynomial. Define polynomial $R(x) = P(x)Q(x)$. Let ω_4 denote the 4-th root of unity. Then, is it true that

$$[P(0)Q(0) \quad P(\omega_4)Q(\omega_4) \quad P(\omega_4^2)Q(\omega_4^2) \quad P(\omega_4^3)Q(\omega_4^3)] = [R(0) \quad R(\omega_4) \quad R(\omega_4^2) \quad R(\omega_4^3)] \ ?$$

Explain your answer in 1-2 lines.
Ans:

Since R(x) = P(x)Q(x), for any number a, R(a) = P(a) * Q(a). This follows from the property of polynomials.

[This has nothing to do with roots of unity, complex numbers, etc. : -1 if those are mentioned]

Q2 [7 points] Consider the following recursive function, defined in terms of a fixed array $X[1 \dots n]$:

$$DP(i,j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ \max \left\{ \begin{array}{c} 2 \cdot [X[i] \neq X[j]] + DP(i+1, j-1) \\ 1 + DP(i+1, j) \\ 1 + DP(i, j-1) \end{array} \right\} & \text{otherwise} \end{cases}$$

What is the optimal time complexity and optimal space complexity to compute the value $DP(1, n)$ using dynamic programming (with memoization)? Briefly explain your answer using 1-2 lines. Here $[X[a] \neq X[b]] = 1$ if $X[a] \neq X[b]$, else $= 0$.
Time complexity:

3 We need to fill n*n entries, and to fill any entry, max of at most three values (each of which can be computed in O(1)) have to be computed. Hence time complexity is O(n*n)

Space complexity:

4 DP(i,j) depends only on DP(i+1,j-1), DP(i+1,j) and DP(i,j-1) [left, lower-left and lower]. The lower half of the array, below the diagonal, is 0 and the diagonal is 1. DP(1,n) is in the top right corner. Hence only the upper diagonal entries need to be filled. These can be filled starting from the last row, and moving up, filling each row from the left. Filling a row will require the values in the row below it.
Hence, only two rows are required. Optimal space complexity is O(n).

Q3 [10 points] Let X and Y be two strings of lengths m and n, respectively. Define $Q[i,j]$ for $i \leq m$ and $j \leq n$, respectively, in the following manner. Suppose in *some* optimal edit sequence from $X[1 \dots i] \mapsto Y[1 \dots j]$, $X[1 \dots \lfloor \frac{m}{4} \rfloor]$ is edited to $Y[1 \dots k]$. Then we define $Q[i,j] = k$; if this scenario cannot happen then $Q[i,j] = \infty$.

Derive a recursive expression for $Q[i,j]$ (along with suitable base cases) that allows us to compute $Q[m,n]$. You can assume that you have the entire edit distance table already available to you.
Ans:

Q[i,j] =

1 if i < floor(m/4) infinity

Q[i,j] is not well-defined for such i,j. Should be infinity as specified above.
if X[1...floor(m/4)] -> Y[1...j] then Q[i,j] must be j by defn.

1.5 if i = floor(m/4) j

Last symbol of X[1...i] is inserted. If X[1...(m/4)] is mapped to Y[1...k] in X[1...i] -> Y[1...j] then X[1...(m/4)] will be mapped to Y[1...k] also in X[1...(i-1)] -> Y[1...j]

2.5 if i > floor(m/4) and Edit(i,j) = Edit(i-1,j) + 1 Q[i-1,j]

Last symbol of X[1...i] is deleted. If X[1...(m/4)] is mapped to Y[1...k] in X[1...i] -> Y[1...j] then X[1...(m/4)] will be mapped to Y[1...k] also in X[1...i] -> Y[1...(j-1)]

2.5 if i > floor(m/4) and Edit(i,j) = Edit(i,j-1) + 1 Q[i,j-1]

2.5 if i > floor(m/4) and Edit(i,j) = Edit(i-1,j-1) or = Edit(i-1,j-1) + 1 Q[i-1,j-1]

Last symbol of X[1...i] is replaced/unchanged. If X[1...(m/4)] is mapped to Y[1...k] in X[1...i] -> Y[1...j] then X[1...(m/4)] will be also mapped to Y[1...k] in X[1...(i-1)] -> Y[1...(j-1)]

Q4 [10 points] Suppose you are given a directed graph $G = (V, E)$ and two vertices s and t. Construct a new directed graph $G' = (V', E')$ in the following manner. $V' = V \times \{0, 1, 2\}$, i.e., for every $u \in V$, there exists three vertices labeled $(u, 0), (u, 1), (u, 2)$ in V'. For every edge $u \to v$ in E, add edges $(u, 0) \to (v, 1)$, $(u, 1) \to (v, 2)$, and $(u, 2) \to (v, 0)$ in E'.

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 G'. 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. Algorithm + justification : 6, complexity : 2

Ans:

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.

Q5 [5 points] Define a directed graph G to be "nice" if it satisfies the property that either $u \rightsquigarrow v$ or $v \rightsquigarrow u$ (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 $(v_1, v_2, \dots)$. 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)
    Run A(H).
    If A(H) returns true:
        return true
    Else
        return false
```

Algorithm + Justification : 5
The algorithm must call A() and use its result.

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

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 $DFS(x)$ in G (before calling $DFS()$ 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 $u \rightsquigarrow v$ to denote that there is a path from u to v and $u \not\rightsquigarrow v$ to denote that there is no path from u to v.

(A) Suppose it holds that there is a path from either $u \rightsquigarrow v$ or $v \rightsquigarrow u$ 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)

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().

(B) Suppose it holds that for every vertex v there is some $u \neq v$ such that $v \not\rightsquigarrow u$. Then, there must be a pair of vertices (x, y) such that $x \not\rightsquigarrow y$ and $y \not\rightsquigarrow x$.

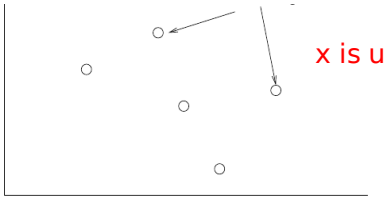
Proof of (B): Let $w \in V$ be the vertex such that $subtree(w)$ has the maximum number of vertices — if there are multiple such vertices, choose any one arbitrarily. (Now arrive at a contradiction involving w)

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.

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.

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 (x_i, y_i). A point (x_i, y_i) is said to be an upper-corner point if for every other point (x_j, y_j), either $x_i > x_j$ or $y_i > y_j$ (or both). For simplicity assume that no two points in P share the same X or Y coordinates.

Algorithm: 11
Complexity: 4



x is upper corner := there is no point that is top-right of x

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
    m = compute median of x coordinates of P
    L = points in P with x-coordinates < m
    R = points in P with x-coordinates >= m
    y-max = maximum y-coordinate of any point in R
    remove points in L whose y-coordinate is less than y-max
    n1 = Solve(L)
    n2 = Solve(R)
    return n1 + n2
```

Another way is to design Solve to return the list of UC points.

```
def SolveList(P):
    L,R : divide points in half according to x-coord
    L1 = SolveList(L)
    L2 = SolveList(R)
    m = number of points in L1 that have
        y-coord below the largest y-coord in L2
    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

Q8 [20 points] 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. For example, given the tree shown below, your algorithm should return the number -12.

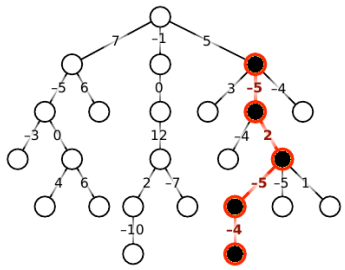


Figure 1: The minimum-weight downward path in this tree is shaded and has weight -12.

You can get partial marks (upto 8) if you solve the special case when all edge weights are negative.

Ans:

3 1) Define P(x) = minimum weight among paths from x to any descendant

7 2) If x is leaf, P(x) = 0

(incuding justification) If x is not a leaf, x has at least one children and best path from x must go via some children

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 }

1 3) Memo is T itself and P(x) is stored with the node x

1 4) Memo is filled by starting at root and going up the tree

2 5) To solve the original problem, return min { P(x) : x is a node in T }

2 6) Space complexity : O(n) where n is the number of nodes in T // only store P(x) with x

4 Time complexity : O(n).

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(n)

When all edge weights are negative, P(x) = min { w(x,y) + P(y) : y is a children of x }

1+3 + .5 + .5 + 1 + 1+1