Monsoon 2022 | Graduate Algorithms: End-sem | *Passing: 25 points*
Date: **13 December** | *Duration: 180 mins*

Avoid detailed pseudocode; instead, explain in words.
Common algorithms like binary search, sorting, graph algorithms done in course can be used directly.

Part-A (40 points)

Write answers to Part-A on the **question paper** itself.

Q1 [2x14=28] State True/False for these questions.

1. Vertex cover cannot be approximated to any constant greater than 1. T
2. The Hamiltonian Path problem can be solved in $O(n!)$ time (by checking all sequences of vertices for a possible path) but not any faster. F
3. There is a dynamic programming algorithm for the 0-1 Knapsack problem that runs in polynomial time. F
4. Determining the chromatic number of a graph currently requires exponential time. T
5. Given a graph G and an integer k, determining if G has an independent set of size k can be done in polynomial time. F
6. There is a polynomial-time reduction from the SCHEDULE problem (the homework-scheduling decision problem given as an HQ) to 3SAT and vice-versa. T
7. Since 2SAT can be reduced to 3SAT, there is unlikely to be a polynomial-time algorithm for 2SAT. F
8. All the strongly connected components of a directed graph can be identified by running the depth-first search algorithm just once. T
9. All NP-hard problems are solvable in exponential time. F
10. If there can be no polynomial-time algorithm to identify a subset of any given set whose sum matches a specified target, then the simpler problem of determining if such a subset exists cannot be solved in polynomial-time. T
11. Suppose that some NP problem A is reducible to B. Then, B is NP-hard. F
12. Suppose that some problem A is reducible to an NP problem B. Then, A is also in NP. T
13. Suppose that some NP-complete problem A is reducible to B. Then, B is NP-hard. T
14. The longest path in a directed acyclic graph can be approximated to any constant greater than 1. T

Q2 [3 points] Given a directed unweighted graph G with n vertices and m edges, and an integer k, consider the problem of deciding if the longest simple path in G has exactly k edges. Choose the best/tightest option.

1. Can be solved in polynomial time.
2. Is an NP-complete problem.
3. Is an NP problem but not NP-complete.
4. Is an NP-hard problem but not NP-complete. True
5. None of the above.

Q3 [2 points] Given an undirected graph G, with n vertices and m edges, consider the decision problem of determining if the vertices of G can be colored (legally) by n colors (i.e., no adjacent pair of vertices have the same color). This problem is (choose the best/tighest option):

1. Solvable in $O(1)$ time. True
2. Can be solved in polynomial time, but not in constant time.
3. Neither in P nor NP-complete, but solvable in exponential time.
4. Solvable in $O(n+m)$ time, but not in $O(1)$ time.
5. NP-Complete.

Q4 [2+3=5 points] Study these two algorithms, and then state True/False for the facts below.

```
def ReduceIStoVC(G,k): return (G,n–k)
def ApproxIS(G): V' = ApproxVC(G); return the vertices in G that are not in V'
```

In the above, `ApproxVC` is some 2-relative-approximation algorithm for the vertex cover problem.

1. `ReduceIStoVC` is a correct reduction from the Independent Set problem to the Vertex Cover problem. T
2. `ApproxIS` is a correct 2-approximation algorithm for the Largest Independent Set problem. F

Q5 [2 points] Suppose we want to obtain a 1000/999-relative approximation algorithm to the solution of an integer Knapsack instance and we run the scaling-based approximation algorithm that we studied. Write the formula to compute the new value, say v'_i, of the i-th item. Answer in terms of these variables: n is the number of items, $v_{\min}$ is the smallest value, $v_{\max}$ is the largest value, W is the Knapsack weight limit, v_i is the value and w_i is the weight of the i-th item.
Ans: $\lfloor 1000 * n * v_i / v_{\max} \rfloor$

Part-B (60 points)

15% if you write **"I don't know"** for any (sub)question.

Q6 [5] The algorithm below employs a Disjoint-Set data structure. Analyse the time-complexity of this algorithm using the *path-compression + union-by-rank* implementation of Disjoint-Sets in terms of n (the number of vertices) and m (the number of edges); separately specify the complexities of the major steps.

```
def MST(G):
 1.      for v in V:
 2.          MakeSet(v)
 3.      E = list of edges in G
 4.      Sort E according to increasing weight
 5.      T = empty set of edges // O(1) complexity
 6.      for e=(u,v) in E: // according to the sorted order
 7.          if Find(u) = Find(v): continue
 8.          Add e to T // O(1) complexity
 9.          Union(u,v)
10.      return T
```

The path-compression + union-by-rank implementation has the property: m operations on n elements incur a total cost of $O(m\alpha(m.n))$.
The above algorithm implements disjoint set on n ($|V|$) elements, and the number of operations are n (Makeset), $O(m)$ (Union), $2m$ (Find). Thus, the total complexity of the Disjoint-Set steps is $O(m\alpha(3m, n))$. *It is okay to write* $O(\alpha(3m, n))$ *as the (average) cost of each step.*
There is only one other operation: sorting E. That takes $O(m \log m)$ steps. So, total complexity is $O(m \log m)$ since $\alpha(3m, n) = o(\log m)$.

Q7 [3 points] Suppose I am trying to find the shortest distances from a source vertex s on a graph G using the Shimbel-Ford-Bellman dynamic programming algorithm done in class. While filling the memo $OPT[t, k]$ (where k is an integer and t is a vertex) in the increasing order of $k = 0, 1, 2 \ldots$, I observe that for one particular k, $OPT[u, k-1] = OPT[u, k]$ for all vertices u. Suppose $dist(s, u)$ denotes the shortest distance between s and u. Which of the following is true and why?
(a) $OPT[u, k] < dist(s, u)$ (b) $OPT[u, k] \leq dist(s, u)$ (c) $OPT[u, k] = dist(s, u)$
(d) $OPT[u, k] \geq dist(s, u)$ (e) $OPT[u, k] > dist(s, u)$ (f) None (answer depends on G)

The basic idea is the recurrence formula for $OPT[j, \cdot]$ depends only on $OPT[j-1, \cdot]$: $OPT[j, u] = \max_v\{OPT[j-1, u], OPT[j-1, v] + w(v, u)\}$ So, the values for any j (i.e., the j-th row in the memo storing the OPT values) can be filled entirely from the values for $j-1$ (the $(j-1$-th row). Since the $k-1$ and k-th rows are identical, this would mean the $(k+1)$-th row would be the same as them, and so will be all the next rows, including the $(n-1)$-th row. The $(n-1)$th row stores the distances $dist(s, u)$. So $dist(s, u) = OPT[n-1, u] = OPT[k, u]$.

Q8 [1+5=6 points] You are given a sequence $A[1...n]$ of integers and a number k. The objective is to partition the sequence into k non-empty subsequences that are *not necessarily contiguous*. The *value* of a subsequence is the sum of the integers in that subsequence and the quality of a partitioning is defined as the minimum value among all the corresponding subsequences. The partitioning should be done in a manner that maximizes the quality. For example, given $A = [1, 6, -1, 8, 0, 7, 3, 9, 8, 8, 3, 4, 9, 9, 8, 4, 8, 4, 8, 2]$ and the integer k = 3, the quality is 35 corresponding to the partition:$(1, 6, -1, 8, 0, 3, 3, 9, 8), (8, 7, 4, 9, 8), (9, 4, 8, 4, 8, 2)$.

(a) Define a decision problem SEQPART corresponding to this optimization problem. (b) Prove that SEQPART is NP-hard with the help of the NP-complete problem PARTITION; describe the reduction algorithm and briefly explain why it is correct (there is no need to state any lemma and its proof).

(a) SEQPART: Given input $A[1 \ldots n]$, integer k, and an integer t, is there a partitioning of A into k non-empty subsequences whose quality is at least t.
(b) Consider this reduction: `def Reduce(A): return (A,2,floor(sum(A)/2))` from SEQPART to PARTITION.

Q9 [5+1=6 points] We will be looking at the longest increasing subsequence problem. Suppose that we have solved an instance $A[0...n]$ (in which a sentinel $A[0] = -\infty$ has been added) by running DP to compute $LIS(i)$ = length of the longest increasing subsequence of $A[i...n]$ that starts with $A[i]$. All the values of $LIS(i)$ for $i = 0 \ldots n$ are available in a table $L[0...n]$. You are now asked to design an algorithm to compute the number of longest increasing subsequences, again using DP. For that you promptly defined the subproblem $NLIS(i)$ = number of longest increasing subsequences of $A[i...n]$ that start with $A[i]$. Explain the next step of DP as asked below. Feel free to use the table L if needed.

(a) Write a *recursive formula* (not a code) to compute $NLIS(i)$. You can use the 0-1 valued function $IF(cond)$ to evalute if some condition *cond* holds or not; for example, $IF(3 = 5-2)$ will return 1 while $IF($abc is a substring of cbcbcb$)$ will return 0. (b) Write the base case to compute $NLIS$.

$NLIS(i) = \sum_{j>i} IF(A[i] < A[j]) * IF(L[i] = 1 + L[j]) * NLIS(j)$,
$NLIS(n) = 1$

Q10 [6 points] A matching of a graph is a set E' of edges such that no two edges in E' share a common vertex. For example, if G is a square shaped graph $A-B-C-D-A$, then a few of its matchings are $\{A-B\}, \{A-B, C-D\}, \{A-D, B-C\}$ in which the last two are maximal; a matching is maximal if its size cannot be increased further by adding any new edge, so the matching $\{B-C\}$ is not maximal. There are simple polynomial-time algorithms to compute a maximal matching of any graph.

Consider the following algorithm to find the smallest vertex cover of a connected undirected graph G: *Construct a maximal matching M of G and return the endpoints of the edges in M.* Derive the approximation ratio of this algorithm. *Hint: You have seen this algorithm before.*

There are basically three steps: $APPROX = 2|E'|$, $OPT \geq |E'|$ since no two edges in E' have a common vertex, and so cannot be covered using a single vertex, and finally, $APPROX \leq 2 * OPT$.

Q11 [10 points] Imagine that we are given an array S of strings, each having at most 100 characters. (a) Describe an efficient divide-and-conquer algorithm to return the length of the longest common prefix of the strings in S. (b) Discuss its time-complexity. Your algorithm should not return anything else and you cannot use any additional subroutine, global variables, etc.

```
def LCPS(A[1...n]):
    if n = 1: return A

    r1 = LCPS(A[1...n/2])
    r2 = LCPS(A[n/2+1 ... n])
    r = length of longest common prefix of A[1] and A[n] // one from 1st and one from 2nd half
    // or... r = LCPS(A[1],A[n]) -- this requires a base case of n=2
    return min(r1,r2,r)
```

T(n) = 2T(n/2) + O(1) = O(n)

OR

Q12 [5+1+4=10 points] You are given an arithmetic expression comprising of "+" and "×"; for example, $S = 1 + 3 \times 2 \times 0 + 1 \times 6 + 7$. You have to determine the largest value possible by placing brackets at appropriate places. (a) Write an efficient recursive backtracking algorithm for the same by completing the outline below. Its input would be an array $V[1...n]$ of integers, and an array $G[1...(n-1)]$ of signs; so, the above S would be represented as $V = [1, 3, 2, 0, 1, 6, 7]$ and $G = [+, \times, \times, +, \times, +]$. Analyse its time-complexity by (b) stating a recurrence, and then (c) proving it by making a suitable guess.

```
def OPT(V,G):
    n = |V| // |G| must be n-1
    // base case(s)
    if n=1: return V[1]

    // recursive calls
    m = 0              (initialization, change it to +/- infinity or anything else as required)
    // fill here ...

    return m
```

Solution:

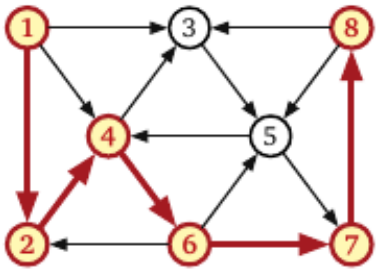
```
Typo: Question paper used S instead of V.
def OPT(V,G):
    n = |V| // |G| must be n-1
    // base case(s)
    if n=1: return V[1]

    // recursive calls
    m = -infinity
    for i=1... (n-1): // position of sign dividing outermost brackets
        if G[i] = +: r = OPT(V[1...i], G[1...i-1]) + OPT(V[i+1 ... n], G[i+1...n])
        if G[i] = *: r = OPT(V[1 ... i], G[1...i-1]) * OPT(V[i+1 ... n], G[i+1...n])
        m = max(m,r)

    return m
```

Define $T(n)$ as the complexity of OPT on an n-length V. Then,
$T(n) = \sum_{i=1}^{n-1} T(i) + T(n-i) + O(n) \leq 2\sum_{i=1}^{n-1} T(i) + cn$. Guess that $T(n) \leq 4^n$. LHS is $\leq 2\sum_{i=1}^{n-1} 4^i + cn = 2\frac{4^n-1}{4-1} - 2 + cn = \frac{2}{3}4^n - \frac{2}{3} - 2 + cn = 4^n - (\frac{1}{3}4^n + \frac{8}{3} - cn) \leq 4^n$ since $\frac{1}{3}4^n + \frac{8}{3} - cn > 0$ for large n.

Q13 [12 points] Let G be a directed graph in which every vertex v has a distinct integer label $l(v)$. We are interested in finding the length of the longest loop-free directed path in G whose vertex labels define an increasing sequence. For example, given the following graph as input, your algorithm should return the integer 5, which is the length of the increasing path $1 \to 2 \to 4 \to 6 \to 7 \to 8$.



Give an efficient dynamic programming algorithm for this problem. Analyse its time-complexity as well.

Define $OPT(s,t,k)$ = length of the longest increasing path from s to t using at most k edges.

- $OPT(s,s,k) = 0$
- $OPT(s,t,1) = -\infty$ if $l(s) > l(t)$ or no edge from s to t
- $OPT(s,t,1) = 1$ if $l(s) < l(t)$ and there is an edge $s \to t$
- $OPT(s,t,k) = \max\{OPT(s,t,k-1), 1 + \max_{v:v\to t,\ l(v)<l(t)} OPT(s,v,k-1)\}$
- This has complexity $O(n^4)$.

Another idea: $OPT(s,t,k)$ = length of the longest increasing path from s to t using intermediate vertices only from $\{1...k\}$ $= \max OPT(s,t,k-1)$, if $l(s) < l(k) < l(t)$: $OPT(s,k,k-1) + OPT(k,t,k-1)$. This has complexity $O(n^3)$.
To solve the original problem, add dummy s^* and t^* with $l(s^*) = -\infty$, $l(t^*) = \infty$ and compute $OPT(s*,t*,n)$.

Q14 [2+10=12 points] Given an unweighted undirected graph G and an edge e in the graph, consider the problem ALLCYCLE that determines if there is a simple cycle in G passing through the specified edge e and visiting every vertex. Now, consider the following algorithm.

```
def AlgoA(G):
    Construct G' in the following manner:
        Start with G=<V,E>
        Take two new vertices, say x and y
        Add edges between x and all vertices of V
        Add edges between y and all vertices of V
        Add an edge between x and y - denote the edge as e'
    Retutn (G',e')
```

(a) Is `AlgoA` a correct many-one polynomial-time reduction from the undirected Hamiltonian cycle problem (HC) to the ALLCYCLE problem? If yes, then state the correctness lemma and briefly explain its proof. If no, then give a counter-example that demonstrates that the reduction is incorrect. Incorrect.

(b) Describe a many-one polynomial-time reduction from ALLCYCLE to HC; prove that it is correct and analyse its complexity.

```
def Reduce(G,e):
    construct H which is a copy of G
    add a new vertex v0 to H
    let e=(u,v)
    remove e from H
    add edges in H: (u,v0), (v0,v)
    return H
```

If G has a Hamcycle passing through e, then H has a Hamcycle (passing through $u - v0 - v$).
If H has a Hamcycle, then it must visit v0, and that can happen either as $\ldots u - v0 - v \ldots$ or $\ldots v - v0 - u - \ldots$. The sequence of vertices on that cycle, except v0, is then a Hamcycle in G.