**Winter 2024** **Algorithm Design and Analysis: Endsem (Sec-B)** **Max: 90 Marks**
**May 4, 2024** **180 minutes**

---

**Roll No:** ______________________________ **Name:** ____________________________________________________________

| Question | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Total |
|---|---|---|---|---|---|---|---|---|---|---|
| Marks | | | | | | | | | | |

**Instructions:** *Please read the questions carefully and write your answers in the space provided. Extra sheet will be provided* only *for roughwork. So use extra sheet to formalize your solution before you articulate it in the question paper. Meaningless blabber & untidy writing fetches negative marks.*

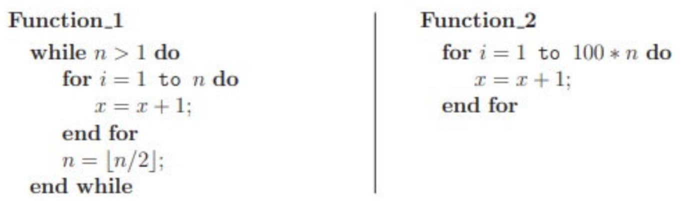
*All The Best!*

---

1. Answer the following questions. **(20 Marks)**

   (a) If $f(n)$ and $g(n)$ are two asymptotically positive functions, such that, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $g(n) \in$ ________ $(f(n))$. **(Ans: $g(n) \in \omega(f(n))$)**

   (b) In the following two functions, let $f(n)$ and $g(n)$ denote the number of times the line "$x = x + 1$" is executed in Function 1 and Function 2 respectively. Then $f(n) \in$ _______ $(g(n))$. **(Ans: $f(n) \in \theta(g(n))$ – 2 marks, $f(n) \in O(g(n))$ or $f(n) \in \Omega(g(n))$ – 1 mark)**



```
Function_1
  while $n > 1$ do
     for $i = 1$ to $n$ do
        $x = x + 1$;
     end for
     $n = \lfloor n/2 \rfloor$;
  end while
```

```
Function_2
  for $i = 1$ to $100 * n$ do
     $x = x + 1$;
  end for
```

Figure 1:

   (c) We (can / cannot) implement Belmanford's algorithm in a distributed setup where as Dijkstra's algorithm we (can / cannot). **(can, cannot)**

   (d) Given a flow network, the minimum cut of a graph is always equal to the maximum flow of the graph. (Yes / No) **(yes)**

   (e) Let $G$ be a flow network with a total capacity $C$. Then there is an algorithm that computes maximum flow of the graph whose running time is independent of $C$. (Yes / No) **(yes)**

   (f) The recurrence relation $T(n) = \sqrt{2}T(n/2) + \log(n)^2$ has a running time $\theta(\_\_)$. **(Ans $\theta(\sqrt{n})$)**

   (g) A problem that cannot be solved in polynomial time is always in the class NP. **(Ans No)**

   (h) A problem that can be solved in polynomial time is always in the class NP. **(Ans Yes)**

   (i) Let, $X$ and $Y$ be two problems, where $X \leq_p Y$. So, $Y$ is at least as hard as $X$. **(Ans Yes)**

   (j) Problem $X$ is in NP-complete and it is solvable in polynomial time iff P = NP. **(Ans Yes)**

2. **(10 Marks)** Given an undirected graph $G = (V, E)$ with $n$ vertices, $m$ edges, and two specified vertices $s$ and $t$. A shortest path between $s$ and $t$ is a path with minimum number of edges. Design an $O(n + m)$-time algorithm that computes the number of shortest paths between $s$ and $t$. Explain the running time of your algorithm. Your algorithm does not have to output all the shortest paths between $s$ and $t$, just the number of shortest paths between $s$ and $t$ suffices.

   (if your algorithm description goes past this page, then also it is okay, but please make sure to properly mention where you write the time complexity of your algorithm)

   (a) **A brief description of your algorithm:**

   The idea is a combination of BFS and Dynamic Programming.

   **Subproblem:** For every vertex $u \in V(G)$, define $Path(u)$ the number of shortest paths from $s$ to $u$ in $G$.

   **Recurrence:** The recurrence of the dynamic programming is as follows.

   **Base Case:** $\text{PATH}(s) = 0$.

   For every $u \in V(G) \setminus \{s\}$, $\text{PATH}(u) = \sum_{v \in \mathsf{Adj}(u): Level(v)+1=Level(u)} \text{PATH}(v)$.

   **Subproblem that solves the actual problem:** $\text{PATH}(t)$.

   **Algorithm Description:** The main steps of the algorithm are as follows.

   (i) Invoke BFS on $G$ starting from $s$.

   (ii) Let $s, v_1, v_2, \ldots, v_k, t, v_{k+1}, \ldots, v_{n-2}$ be the order of vertices obtained by the BFS traversal of the graph.

   (iii) Let $\text{Level}(u)$ denotes the distance of $u$ from $s$ as obtained from BFS traversal.

   (iv) Initialize $\text{NumPath}[s] = 0$.

   (v) For every $v_1, \ldots, v_k$ (in this order as appears in BFS traversal) set $NumPath[v_i] = \sum_{v_j \in Adj[v_i]: \text{Level}(v_i)=\text{Level}(v_j)+1} \text{NumPath}[v_j]$.

   (vi) $\text{NumPath}[t] = \sum_{v_j \in \text{Adj}[t]: \text{Level}(t)=\text{Level}(v_j)+1} \text{NumPath}[v_j]$.

   (vii) Return $\text{NumPath}[t]$.

   (If used dynamic programming, then **3 marks** for stating the recurrences, subproblem definitions, and subproblem that solves the actual problem and **5 Marks** for algorithm description)

   (If not used recurrence, but directly used the algorithm, then he should explain why his algorithm should work. **8 Marks** for the algorithm description and explanation.)

   (b) **Explanation of running time of your algorithm**.

   Observe that BFS traversal takes $O(n + m)$-time for graphs with $n$ vertices and $m$ edges. The subsequent steps need to scan the adjacency list for every vertex constant number times. This procedure also takes $O(n + m)$-time.

   Hence, the total time taken is $O(n + m)$.

   **(2 Marks)**

3. **(10 Marks)** Recall matrix chain multiplication. For given matrices $M_1 \in \Re^{i_0 \times i_1}, M_2 \in \Re^{i_1 \times i_2}, \ldots, M_n \in \Re^{i_{n-1} \times i_n}$, the goal is to quickly compute the multiplication of all the matrices, i.e., $M = M_1 \cdot M_2 \cdot \ldots \cdot M_n$. Following is an efficient algorithm that computes $M$.

   (a) Using DP compute the 'order' in which the matrices needs to be multiplied.

   (b) Compute $M$ using the above 'order'.

   Design an dynamic programming algorithm to compute the 'order' and the running time.

   (a) Define your subproblem.
   **(1 Marks for correct subproblem definition)** For every $1 \leq a < c \leq n$, let $m(a,c)$ is the minimum number of scalar multiplications needed to compute $M_a \cdot M_{a+1} \cdot \ldots \cdot M_c$
   For every $1 \leq a < c \leq n$, let $s(a,c)$ is the break point $b$ such that $M_a \cdot M_{a+1} \cdot \ldots \cdot M_c$ is optimally multiplied as $(M_a \cdot M_{a+1} \cdot \ldots \cdot M_b) \cdot (M_{b+1} \cdot \ldots \cdot M_c)$. Do not deduct marks if $s$ is not defined.

   (b) Write the recurrence of the subproblem and the subproblem that solves the final problem
   **(2 Marks for correct recurrence; deduct 1 marks if there are issues such as $\min\limits_{b|a<b<c}$ is not well defined)**

$$m(a,c) = \begin{cases} 0 & \text{if } a = c \\ \min\limits_{b|a<b<c} m(a,b) + m(b+1,c) + i_a \cdot i_b \cdot i_c & \text{else} \end{cases}$$

   The subproblem that solves the final problem is $m(1,n)$.

$$s(a,c) = \begin{cases} b; b \in arg\min\limits_{b|a<b<c} m(a,b) + m(b+1,c) + i_a \cdot i_b \cdot i_c & a < b < c \end{cases}$$

   Do not deduct marks if $s$ is not defined.

   (c) Write a pseudocode or briefly describe the algorithm in english along with its running time. **(3 Marks)** To compute the order, $s$ needs to traversed back from $s(1,n)$ to $s(i,i)$

```
MATRIX-CHAIN-ORDER(p)
$n = p.length - 1$
let $m[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables
for $i = 1$ to $n$
    $m[i,i] = 0$
for $l = 2$ to $n$          // $l$ is the chain length
    for $i = 1$ to $n - l + 1$
        $j = i + l - 1$
        $m[i,j] = \infty$
        for $k = i$ to $j - 1$
            $q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$
            if $q < m[i,j]$
                $m[i,j] = q$
                $s[i,j] = k$
return $m$ and $s$
```

Figure 2: Pseudo Code

   for every $i \in [n]$. **(Deduct 1 marks if traverse $s$ is not done.)**

(d) Consider 5 matrices defined as $M_1 \in \Re^{9\times 3}$, $M_2 \in \Re^{3\times 6}$, $M_3 \in \Re^{6\times 4}$, $M_4 \in \Re^{4\times 1}$, and $M_5 \in \Re^{1\times 9}$. Using a table filling method or recurrence tree compute the most efficient order of matrix chain multiplication and the total running time for the multiplication, i.e., computing $M$ from the given matrices.

**(4 marks for table or recurrence tree), deduct 0.5 marks if the table is not complete, deduct 0.5 if $s$ values are not inconsistent. In case of recursion tree, do not deduct marks for $m$ if at least one path is there, deduct 1 marks if order is not handled in the recursion tree.**

Table 1: Table $m$ and $s$

| | $M_1$ | $M_2$ | $M_3$ | $M_4$ | $M_5$ |
|---|---|---|---|---|---|
| $M_1$ | 0 | 162 (1) | 180 (1) | 69 (1) | 150 (4) |
| $M_2$ | | 0 | 72 (2) | 42 (2) | 69 (4) |
| $M_3$ | | | 0 | 24 (3) | 78 (4) |
| $M_4$ | | | | 0 | 36 (4) |
| $M_5$ | | | | | 0 |

**(Deduct 1 marks if order is not mentioned)** The matrix multiplication order is $(M_1 \cdot (M_2 \cdot (M_3 \cdot M_4))) \cdot M_5$.

4. **(10 points)** Consider a vast network of routers such that no router has the knowledge of the complete network. However, every router has the knowledge of its neighbours along with the latency (i.e., the time taken to send a message from itself to the neighbour).

   (i) Design a space efficient algorithm (i.e., does not require the knowledge of the complete network) to compute shortest path (i.e., a path with minimum sum of latency) from a router $s$ to every other router in the network. If your algorithm uses dynamic programming then define subproblem, recurrence of the subproblem and the description of the algorithm.

   **(2 marks for subproblem)** Subproblem: $C(i, v)$ is the minimum cost of $A - v$ path with at most $i$ edges.

   **(3 marks for reccurence and subproblems that solves the original problem. Deduct 1 marks if all the subproblems that solves the original problem are not properly mentioned or explained.)** Recurrence: $C(i, v) = \min(C(i-1, v), \min_{u\in V} C(i-1, u) + w_{uv})$

   Subproblems that solves the original problem: $C(j, B), C(j, C), \ldots, C(j, F)$ where $j$ is the longest path from $A$ to any vertex or $j = |E|$.

   (ii) What is the running time of your algorithm?

   **(1 marks)** $O(mn)$ or $O(n^3)$ where $n = |V|$ and $m = |E|$.

(iii) Consider the following network of routers. Using your above algorithm compute the shortest path from router $A$ to every other router in the network.

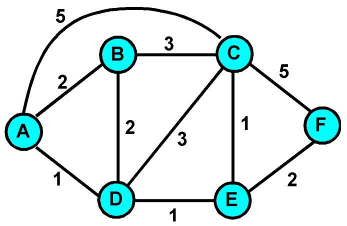


Figure 3:

**4 marks** Shortest distance from $A$ to $[B, C, D, E, F]$ is $[2, 3, 1, 2, 4]$.

5. **(5 Marks)** Define problems in class P and in class NP.

   **(2 marks for correct definition of P and 3 marks for correct definition of NP)**

   P: The class of problems for which there are algorithms that solves the problem in polynomial time. Example, sorting an array, finding $k^{th}$ smallest number etc.

   NP: The class of problems for which there are nonditerministic polynomial time algorithm that solves the problem, i.e., given a solution there is a polynomial time algorithm that can verify if the solution is correct or incorrect. Example, $k$ independent set, $k$ colouring etc.

6. **(5 Marks)** Recall that a problem $X$ is in NP-hard if $Y \leq_p X$ for every problem $Y$ in NP. Prove that if $X$ is polynomial time solvable then P = NP.

   **(5 marks)** If $X$ can be solved in polynomial time then for every problem $Y$ in NP can also be solved in polynomial time. This is due to fact that $Y \leq_p X$.

7. **(10 Marks)** SAT problem finds satisfiability instances in boolean expressions. In particular, the boolean expression for the 3SAT problem consists of clauses that have at most 3 literals, e.g. $(x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_3 \vee \bar{x}_4)$.

   (a) Given an undirected graph $G = (V, E)$, an independent set is a set of vertices $S \subseteq V$ such that for every pair of vertices $u, v \in S$, the edge $(u, v) \notin E$. Give a polynomial time reduction from a 3SAT boolean expression $\psi$ to a graph $G_\psi$, such that $\psi$ can be solved in polynomial time by solving the independent set problem in $G_\psi$ in polynomial time using a black box.

   **(3 marks)** The reductions are as follows,

   - Make a clique of literals from each clause.
   - Join every $x_i$ and $\overline{x}_i$ nodes.

   (b) Let $\psi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_3 \vee \bar{x}_4)$, construct $G_\psi$ using the above reduction rules.

   **(4 marks)** The reductions are as follows,

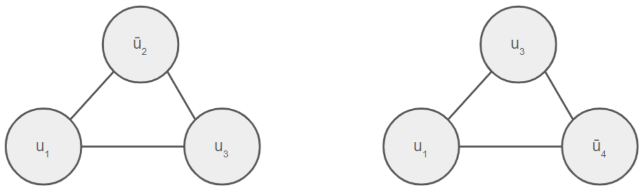


Figure 4:

(c) Show that the 3SAT is satisfiable if and only if the graph has an independent set of size 2. You can take a 3SAT solution as an instance and show its corresponding solution for the independent set problem and vice versa.

**(4 marks)** If $\psi$ is satisfiable then each triangle in the graph $G_\psi$ has at least one one node which evaluates to 1. Select one such node from each triangles and construct a set $S$. The set $S$ is an independent set. If there were an edge between two nodes $u, v \in S$, then the labels of $u$ and $v$ would have been a conflict. However, this is not possible since they both evaluate to 1.

If $S$ is an independent set then assign its corresponding variables to 1 in $\psi$. This satisfies $\psi$. Since $S$ of $G_\psi$ touches every clause of $\psi$ so every clauses are true hence, $\psi$ is also true.

8. **(10 Marks)** Consider the following flow network. The numbers denote the capacities of the edges. Illustrate the execution of Ford-Fulkerson's Algorithm to compute a maximum flow from $s$ to $t$ in this network. Your illustration must show step by step execution of the Ford-Fulkerson's Algorithm, particularly the changes in the residual graph, and the flow value in every edge at each step.

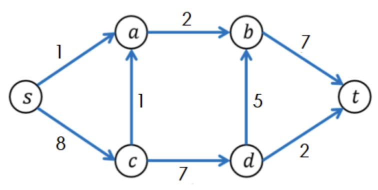


Figure 5: Flow network

9. **(10 Marks)** The CS department of a university has a flexible curriculum with a complicated set of graduation requirements. The department offers $n$ courses and there are $m$ requirements. A student has to satisfy all of the $m$ requirements to graduate. Each requirement specifies a subset $A$ of courses and the number of courses that must be taken from the subset. The subsets for different requirements may overlap but each course can be used to satisfy at most one requirement.
Design a poly$(n + m)$-time algorithm that determines whether a student can graduate.
**Example:** There are 5 courses $x_1, x_2, x_3, x_4, x_5$ and $m = 2$. The requirements are **(i)** at least two courses must be taken from $\{x_1, x_2, x_3\}$, and **(ii)** at least two courses must be taken from $\{x_3, x_4, x_5\}$. A student can graduate with courses $\{x_1, x_2, x_3, x_4\}$ but cannot graduate with courses $\{x_2, x_3, x_4\}$.

- **Formulation as Flow Network** (**4 Marks**)

  For every course, create a vertex. For every requirement, create a vertex. Create two additional vertices $s$ and $t$.

  In summary, the vertices $\{x_1, \ldots, x_n\}$ represent the courses and the vertices $\{y_1, \ldots, y_m\}$ represent the requirements.

  Suppose that the $j$-th requirement contains a subset of courses $D_j$ says that at least $r_j$ courses must be taken from $D_j$.

  For every $i \in \{1, \ldots, n\}$, create a directed edge $s \to x_i$ with capacity 1.

  For every $j \in \{1, \ldots, m\}$, create a directed edge $y_j \to t$ such that the capacity of the edge $y_j \to t$ is $r_j$.

  **(Also, consider the solutions, with number of requirements independent of $m$, say $k$ and the number of course from each requirements is $m$)**

Finally, for every $i \in \{1, \dots, n\}$, if the course corresponding to the vertex $x_i$ appears in $D_j$, then put a directed edge $x_i \to y_j$ with capacity 1.

- **Designing the algorithm with this flow network and justification.**

  (i) Construct the flow network as described above.

  (ii) Compute a maximum $s$-$t$-flow using Ford Fulkerson's Algorithm.

  (iii) If the value of the maximum flow is $\sum_{j=1}^{m} r_j$, then output "yes, a student can graduate".

  **In the other case, if maximum flow is $km$ then output 'yes, a student can graduate'**. Otherwise, output that a student cannot graduate.

  **Justification:** Note that in order to graduate, there must be a set of courses that must be taken so that each of those chosen courses satisfies exactly one requirement, and for every requirement $D_j$, exactly $r_j$ courses are taken.

  The capacity of $x_i \to y_j$ being 1 is justified since one chosen course can satisfy exactly one requirement, no more than that.

  (**4 Marks** total out of which 3 marks for the algorithm description and 1 mark for the justification.)

- **Running time of the algorithm.**

  The ford-Fulkerson's algorithm runs in time $O((|V| + |E|)|f^*|)$ such that $|f^*|$ is the value of maximum flow. Observe in this graph that the value of maximum flow is $n$ (i.e., total number of courses).

  The number of vertices is $n+m+2$ **(or $n+k+2$)** and the number of edges is $O(kn)$. Hence, it holds that the running time of the Ford-Fulkerson's algorithm is $O((mn+n+m)n)$ which is $O(n^2 \cdot m)$. **(Alternatively, $(n + k + kn)n$, which is $O(n^2k)$)**

  Finally, the construction of the graph takes $O(mn)$-time **(Alternatively, $O(nk)$)**.