Assignment No. 2

Course Title: Advanced Algorithm

Course No: CSE 511

Submitted to: Mohammad Mahmudul Alam (MLD)

Submitted by: Sumaiya Tarannum Noor

Section: 02

ID: 2425410650

Submission date: 6th December, 2025

Name : Sumaiya Tarannum Noor
ID : 2425410650
Date : 6/12/2025

Assignment-2

Answer to the Question No. 1

(a) Step-by-step dynamic programming explanation:

We use the classic DP approach where dp[i] = length of the longest increasing subsequence ending at index i.

Input : A = [10, 9, 2, 5, 3, 7, 101, 18]
indicies 0 → 7

1. Initialize dp = [1, 1, 1, 1, 1, 1, 1, 1]
(every element alone is an LIS of length 1).

2. Also keep a prev array initialized to -1 to reconstruct one LIS later:

prev = [-1, -1, -1, -1, -1, -1, +1, -1]

Now, iterate i from 0 to 7, and for each i check all j<i.

If A[j] < A[i] then A[i] can extend the LIS that ends at j, so consider dp[j] + 1.

Now, updates are shown:

- $i=0$ $(A[0]=10)$: no $j<0 \rightarrow dp[0]=1$
- $i=1$ $(A[1]=9)$: check $j=0$ $(10<9?$ no$) \rightarrow dp[1]=1$.
- $i=2$ $(A[2]=2)$: $j=0$ $(10<2?$ no$)$, $j=1$ $(9<2?$ no$)$
  $dp[2]=1$
- $i=3$ $(A[3]=5)$:
  = $j=0$ $(10<5?$ no$)$, $j=1$ $(9<5?$ no$)$, $j=2$
  $(2<5?$ yes$) \rightarrow$ candidate $dp[2]+1=2$.
  So, $dp[3]=2$, $prev[3]=2$
- $i=4$ $(A[4]=3)$:
  = $j=2$ $(2<3?$ yes$) \rightarrow dp[2]+1=2 \Rightarrow dp[4]=2$,
  $prev[4]=2$
  = other j's give no larger value.
- $i=5$ $(A[5]=7)$:
  = $j=2$ $(2<7 \rightarrow 2)$, $j=3$ $(5<7 \Rightarrow dp[3]+1=3)$,
  $j=4$ $(3<7 \rightarrow dp[4]+1=3)$.
  Best is 3 $\rightarrow dp[5]=3$, $prev[5]=3$
  (or 4 - either works)

- $i = 6$ ($A[6] = 101$):
  = many j with ($A[j] < 101$), best extension is from $j = 5$ with
  $dp[5] = 3 \Rightarrow dp[6] = 4$, $prev[6] = 5$.

- $i = 7$ ($A[7] = 18$):
  = best extension also from $j = 5$ ($dp[5] = 3$) $\Rightarrow$
  $dp[7] = 4$, $prev[7] = 5$

Final dp array:

$dp = [1, 1, 1, 2, 2, 3, 4, 4]$

Maximum value is 4 (so LIS length = 4).

Reconstruction (example for index 7): use prev chain: $7 \leftarrow 5 \leftarrow 3 \leftarrow 2 \Rightarrow$
sequence indices $[2, 3, 5, 7] \Rightarrow$ values $[2, 5, 7, 18]$.

For index 6 chain $6 \leftarrow 5 \leftarrow 3 \leftarrow 2 \Rightarrow$
$[2, 5, 7, 101]$. Another valid LIS is $[2, 3, 7, 18]$ if $prev[5]$ chosen as 4.

So LIS length = 4 (e.g. $[2, 3, 7, 18]$ or $[2, 5, 7, 101]$).

Answer to the Question No. 2

Code:

```
import bisect


def longest_increasing_subsequence_optimized(nums):
    if not nums:
        return 0, []



    tails = []
    prev = [-1] * len(nums)
    indices = []

    for i, num in enumerate(nums):
        idx = bisect.bisect_left(tails, num)
        if idx == len(tails):
            tails.append(num)
            indices.append(i)
        else:
            tails[idx] = num
            indices[idx] = i



        if idx > 0:
            prev[i] = indices[idx - 1]

    max_length = len(tails)
    last_index = indices[-1]
```

```
    lis = []
    while last_index != -1:
        lis.append(nums[last_index])
        last_index = prev[last_index]
    lis.reverse()

    return max_length, lis

nums = [10, 9, 2, 5, 3, 7, 101, 18]
length, subsequence = longest_increasing_subsequence_optimized(nums)
print(f"Optimized Length of LIS: {length}")
print(f"Optimized LIS: {subsequence}")
```

Output:

```
Optimized Length of LIS: 4
Optimized LIS: [2, 3, 7, 18]
```

(4) Time complexity

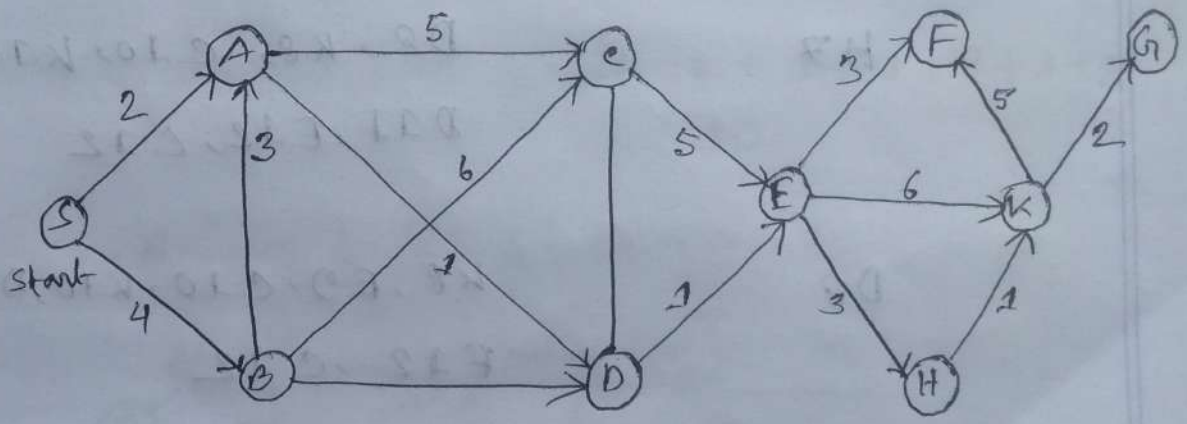
- Two nested loops → $O(n^2)$

Space complexity

- dp array + prev array → $O(n)$

Answer to the Question No. 2

Applying Dijkstra's Algorithm to find the shortest path from the start vertex to the goal vertex in the graph:



Current Node	Frontier
~~S0~~	S0, ~~A2, B4~~
S0	(A2), B4
A2	(D3), B4, C7
D3	B4, (E4), C7
B4	E4, C7, A7, C10, D11
E4	C7, A7, F7, (H7), C10, K10, D11

Current Node	Frontier
C7	A7, F7, H7, C10, K10, D11, D11, E12
A7	F7, H7, D8, C10, K10, D11, D11, E12, C12
F7	H7, D8, C10, K10, D11, D11, E12, C12
H7	D8, K8, C10, K10, D11, D11, E12, C12
D8	K8, E9, C10, K10, D11, D11, E12, C12
K8	E9, C10, K10, G10, D11, D11, E12, C12, F13
E9	C10, K10, G10, D11, D11, E12, F12, H12, F13, K15
C10	K10, G10, D11, D11, E12, C12, F13, D14, K15, E15

Current Node	Frontier
K 10	G10, D11, D11, E12, C12, G12, F13, D14, k15, E15, F15
(G10) Goal Achieved	D11, D11, E12, C12, G12, F13, D14, k15, E15, F15

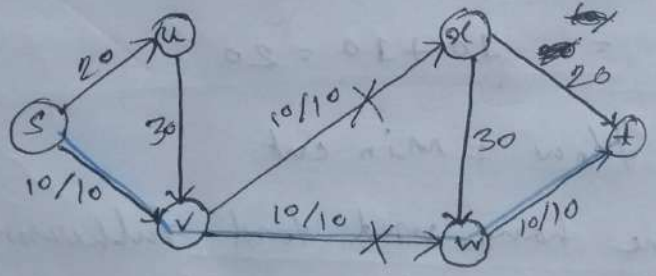
As we achieved the goal, All of these will remain in the frontier.

Path: S → A → D → E → H → k → G

So, the shortest path length = 2 + 1 + 1 + 3 + 1 + 2
= 10.

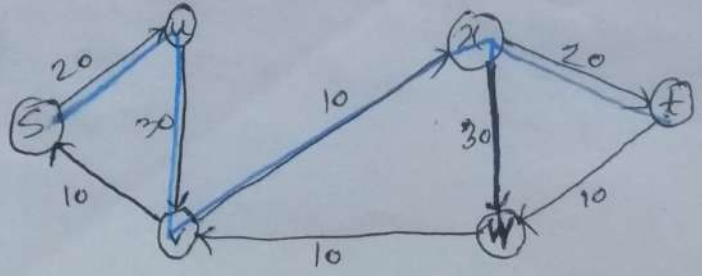
Answer to the Question No. 3

(a)



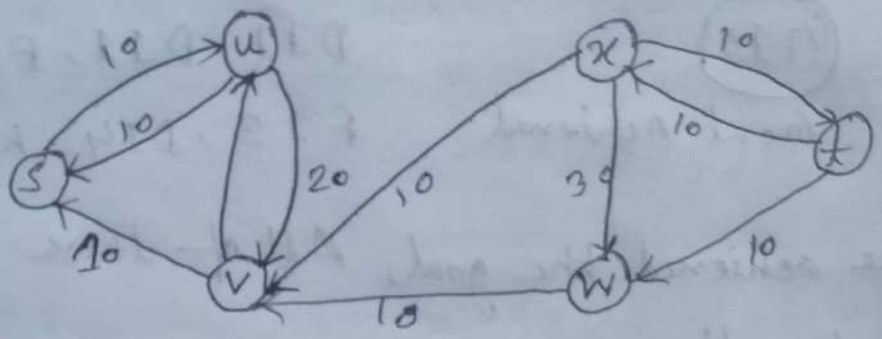
Path: S → V → W → t

Bottleneck : 10



Path : $s \to u \to v \to x \to t$

Battleneck : 10



There is no more ~~aug~~ augmenting path.

$S = \{s, u, v\}$ $\quad t = \{x, w, t\}$

Max - flow = 10 + 10 = 20

Min - cut = 10 + 10 = 20

So, max - flow = min cut

Therefore, we have used ford - fulkerson Algorithm to solve the problem.

(6) Efficiency of the Ford-Fulkerson Algorithm:

The Ford-Fulkerson method finds a maximum flow by repeatedly finding augmenting path in a residual graph and increasing the flow along those paths.

Efficiency:

* The efficiency depends on how augmenting paths are selected.
* When we use BFS to find the shortest (in edges) augmenting path, the algorithm becomes the Edmonds-Karp varient of Ford-Fulkerson.
* Using BFS avoids choosing extremely long or slow paths and gwarantees polynomial time.

Time Complexity:

When BFS is used:

* Each BFS takes $O(V+E)$ time
* It can perform at most $O(V \times E)$ augmentations.

Therefore,

Time complexity = $O(VE^2)$

Using BFS to find augmenting paths converts Ford-Fulkerson into the Edmonds-Karp algorithm.

It's time complexity is:

$O(VE^2)$

which is polynomial and ensures the algorithm always terminates with the correct maximum flow.

Answer to the Question No. 4

(a) Hamiltonian cycle and NP-completeness proof (from 3 SAT) —

Definition (Problem statement)

A Hamiltonian cycle in an undirected graph $G = (V, E)$ is a cycle that visits every vertex exactly once and returns to the start.

The Hamiltonian cycle (HC) decision problem asks: given G, does G contain a Hamiltonian cycle?

(i) $HC \in NP$.

A nondeterministic certificate is an ordering $v_1, v_2, \ldots, v_{|V|}$ of all vertices. We can verify in polynomial time that every consecutive pair (v_i, v_{i+1}) and $(v_{|V|}, v_1)$ is an edge and that all vertices are distinct. So, HC is in NP.

(ii) Reduction $3\text{-SAT} \leq_p$ Hamiltonian cycle (sketch of standard construction).

Let φ be a 3-CNF formula with variables $x_1, \ldots, x_n$ and clauses $c_1, \ldots, c_m$; each clause has three literals. We build a graph G such that G has a Hamiltonian cycle iff φ is satisfiable.

Construction (outline, polynomial size)

1. Variable gadget (choice structure)

For each variable x_i with k_i occurrences in φ, create a ladder (two parallel rows of vertices with rungs between them) made them $2k_i$ occurrence-vertices arranged so that a Hamiltonian cycle must traverse either top row entirely or the bottom row entirely through that gadget - but not both.

* Intuition: choosing the top row means set x_i = true; choosing the bottom row means set x_i = false.
* The gadget is built so there are exactly two ways (up to local symmetry) to pass through all its vertices in a Hamiltonian cycle: the cycle follows the top rail or the bottom rail.

2. Occurrence vertices and clause connections.

For every occurrence of a literal (say literal x_i in clause c_j or $\neg x_i$ in c_j) use one specific occurrence-vertex in the corresponding row of the variable gadget.

For each clause c_j create a clause vertex c_j. connect c_j to the three occurrence-vertices that corresponds to three literals of c_j. (so a clause vertex has degree 3.)

3. Global wiring.

Join variable gadgets into one connected structure with extra connector vertices and edges so a Hamiltonian cycle must pass through every gadget in some order.

Also ~~each~~ attach each clause vertex so it can be visited only by detouring to one of its three occurrence-vertices.

4. Size and time.

The number of vertices and edges is linear in the formula size (sum of variable occurrences and number of clauses), so the construction is polynomial-time.

Correctness (why the reduction works).

* ($\Rightarrow$) If φ is satisfiable $\Rightarrow$ G has a Hamiltonian cycle.

  Let an assignment satisfy φ. For each variable gadget choose the rail corresponding to the truth value (top for true, bottom for false). Follow those rails to visit all variable-gadget vertices. For each clause C_j pick one satisfied literal in that clause; from the cycle route use that clause's connection to the chosen occurrence-vertex to include the clause vertex c_j in the cycle by a short detour and return. Because every clause has a

satisfied literals, all clause vertices can be included without revisiting vertices. Thus we can construct a Hamiltonian cycle visiting every vertex exactly once.

* ($\Leftarrow$) If G has a Hamiltonian cycle $\Rightarrow$ φ is satisfiable.

In any Hamiltonian cycle, each variable gadget must be traversed in one of it's two canonical ways (top or bottom rail) – otherwise some gadget vertex would be missed or repeated. Interpret top = true, bottom = false. Each clause vertex c_j must also appear on the cycle; the only way for the cycle to reach c_j is via one of it's three incident occurrence – vertex that belongs to the chosen rail of it's variable gadgets. Therefore for each clause at least one of it's literal is true under the assignment determined by the rails. Hence the assignment satisfies φ.

The reduction is polynomial and preserves satisfiability, so $3\text{-SAT} \leq_p HC$. Since 3-SAT is NP-complete and $HC \in NP$, HC is NP-complete.

(b) Significance of NP-completeness and impact of NP-hardness

why NP-completeness matters (practical significance)

1. Classification & expectation

NP-completeness classifies decision problems that are as hard as any problem in NP. If a problem is NP-complete, we strongly suspect there is no polynomial time algorithm for all instances (unless $P = NP$).

this sets realistic expectations for algorithm designers and users: for large arbitrary inputs, exact polynomial algorithms probably do not exist.

2. Modeling and reduction use.

Many real-world optimization, planning, scheduling, routing and verification problems are NP-complete when expressed naturally.

Knowing a problem is NP-complete helps practitioners:

instead of searching for an elusive poly-time exact algorithm, they focus on alternatives (heuristic, special cases, approximations).

3. Guides algorithm design

NP-completeness motivates several successful research directions:

* Exact exponential algorithms with improved bases (e.g., $O(c^n)$ with small c for moderate-size instances.
* Approximation algorithms with provable guarantees where exact solution is infeasible

* Parameterized complexity (FPT):
find parameters k such that problem is efficient when k is small.

* Heuristics and SAT/ILP solvers:
highly optimized solvers that work well on practical instances despite worst-case hardness.

4. Cryptography and hardness assumptions.
Hardness of certain problems underpins cryptographic security. While NP-completeness is not directly the basis of most cryptosystems, the general idea of computational hardness is central.

Impact of NP-hardness on research and practice

* Algorithmic research focus shifts.
Rather than seeking impossible general polynomial solutions, researchers:

** Identify tractable special cases (planar graphs, bounded tree width, monotone formulas).

** Seek approximations ratios or PTAS when approximations are acceptable.

** Develop parameterized algorithms and kernelization.

** Improve practical solvers (SAT, CP, ILP) using clever engineering, learning and preprocessing.

Practice consequences

Industries adapt by:

** Using heuristic and meta heuristics (local search, genetic algorithms) that perform well in practice.

** Accepting near-optimal or good-enough solutions where optimality is infeasible.

** Applying problem reductions to well-solved frameworks (encode as SAT/ILP and use robust solvers).

Research Benefit from hardness proofs:

Reductions and NP-hardness proofs are useful tools to:

** Understand relative difficulty of problems.

** Transfer lower-bound / approximation impossibility results (if A reducible to B, hardness of A implies hardness of B).

** Guide effort: invest in special-case, approximation, or heuristic research when hardness is proven.

NP-completeness tells us which problems are unlikely to have efficient general exact algorithms. This reframes both theoretical research and practical engineering:
it drives the search for special cases, parameterized and approximate solutions, and high-performance heuristic that makes otherwise intractable problems solvable in real applications.

—x—
—o—