Module 3. – Greedy Algorithms and Dynamic Programming

* The Greedy Approach.

→ In the greedy approach, the solution to the problem is built incrementally by choosing the locally optimal choice at each step.

→ The optimal choice is made based on the information available at that moment.

---

* Kruskal's algorithm for Minimum Spanning Tree.

→ Step 1 - Construct a minimum heap of e-edges

→ Step 2 - Take the edges one-by-one starting from the least weight. (cycle/loop is not created)

Best case → (n-1) edges
Worst case → e edges

---

* Spanning Tree.

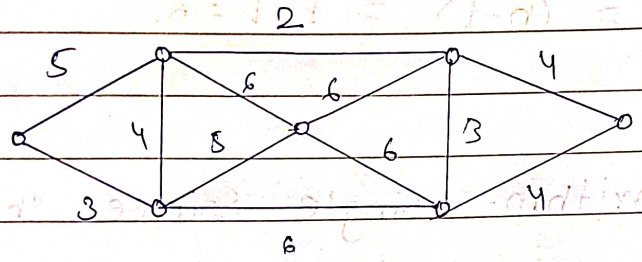
→ A spanning tree (S) should be a sub-graph of the graph G(V,E)

① S should contain all the vertices of G.

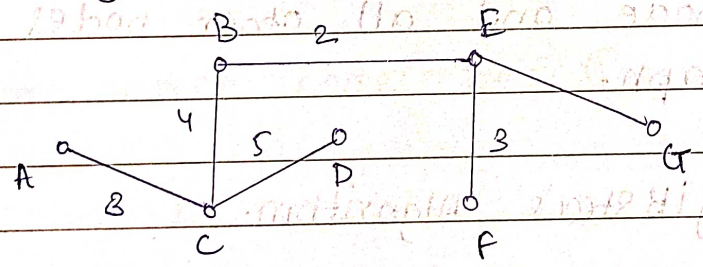
② S should contain (|V|-1) edges.

③ S should have no cycles or loops.

Example ①



→ Step ① → Arrange all edges to be in ascending order.



Consider minimum weight first

① → 2 (BE)

② → 3 (AC, EF)

③ → 4 (BC, FG, GE) {EFG form a loop so one out of FG and EG is not drawn}

④ → ⑤ : (CD, AB) ∵ ABC form a cycle so AB will not be drawn ∴

⑤ ∴ Total weight = 3+4+5+2+3+4 = 21

So edges = (n-1) = 7-1 = 6

* Djikstra's Algorithm. Single Source Shortest Path.

→ Djikstra's Algorithm is a popular algorithm used for finding the shortest path between a source node and all other nodes in a weighted graph.

Steps in Djikstra's Algorithm.

① Initialize all nodes with a distance value of ∞ and the source node to 0.

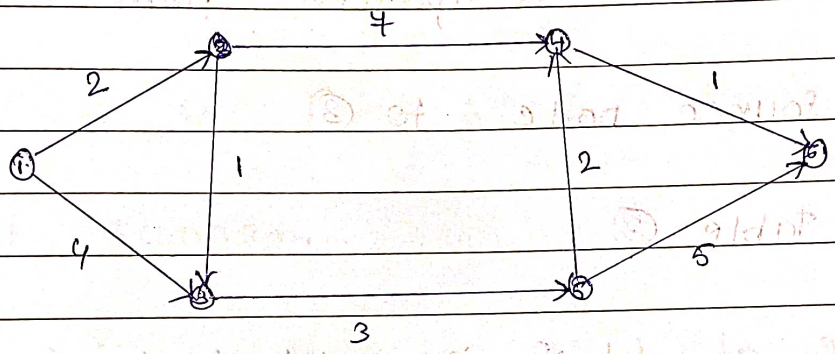
② Create a set of unvisited nodes and all all nodes to the graph.

③ While there are unvisited nodes, select the node with the smallest distance value and mark it visited

④ For each neighbour of the selected node that is still unvisited, calculate the distance to that neighbour by adding the weight of the edge b/w two nodes to the distance value of selected nodes.

⑤ Repeat this for all nodes

Example ①



→ Let the source node be ①

So table 1

1	0
2	2
3	4
4	∞
5	∞
6	∞

→ Change the source node to 2.

So Table 2-

Use formula for Relaxation

2	0
3	∞ 1
4	∞ 7
5	∞
6	∞

if (d[u] + c(u,v) < d[v])
d[v] = d[u] + c(u,v)

As (2+1) < 4 & 2+7 < ∞

∴ 1 → 3 is updated from 4 to 3
1 → 4 is updated from ∞ to 9.

→ Move source node to ③

So Table ③

3	
4	∞
5	3
6	∞

1→5 is updated from ∞ to 6.

As 2+1+3 < ∞

2 → 5 is updated from ∞ to 4

→ Move source node to 5.

So Table 5

5	0
1	6
2	4
3	3
4	2
6	5

2→6 updated from ∞ to 9.
2→4 updated from 7 to 6
(1→3→5)
1→4 updated from 9 to 8
1→6 updated from ∞ to 11.
3→6 updated from ∞ to 8.
3→4 updated from ∞ to 5

Move selected node to (4)

Table 4

4	
1	8
2	6
3	8
5	2
6	1

1 → 6 updated from 11 → 9
2 → 6 updated from 9 to 7
3 → 6 updated from 8 to 6.
5 → 6 updated from 5 to 3

Move to node 6.

Full scanned.

So minimum distance of 1 to 2, 3, 4, 5, 6.

Final Table

1	0
2	2
3	3
4	8
5	6
6	9

* Knapsack Problem.

→ The Knapsack problem is a classic optimization problem which involves packing a knapsack with a set of items, each with a weight and a value. in such a way that we can maximise the value while staying within the weight capacity.

There are two types of knapsack problems

① 0/1 Knapsack - Each item can either be included or excluded in the knapsack. We cannot include the fraction of an item.

The Recurrence Relation is:-

→ If $i=0$ and $j=0$ then max value is 0.

→ If $w_i > j$ then item cannot be included in the knapsack, the max value is reached by packing the first $i-1$ items.

→ If $w_i <= j$ - Then item can be included or excluded.

Here i is the items
j is the knapsack capacity.

(2) Fractional Knapsack - In this problem, fractions of weights and profits can be taken.

(1) 0/1 Knapsack Problem.

Example

$m = 8$ $P = \{1, 2, 5, 6\}$

$n = 4$ $W = \{2, 3, 4, 6\}$

Method (1) Table form.

Pi	Wi	0	1	2	3	4	5	6	7	8
1	2	0	0	1	1	1	1	1	1	1
2	3	0	0	1	2	2	3	3	3	3
5	4	0	0	1	2	5	5	6	7	7
6	5	0	0	1	2	5	6	6	7	8

Formula.

$$V[i, w] = \max \{ V[i-1, w], V[i-1, w-W[i]] + P[i] \}$$

Method ② - Sets Method.

→ Form sets in form (P, W)

$S^0 = \{(0,0)\}$

$S^0_1 = \{(1,2)\}$

$\therefore S^1 = S^0 \cup S^0_1 = \{(0,0), (1,2)\}$

$S^1_1 = \{(2,3), (3,5)\}$

$S^2 = \{(0,0), (1,2), (2,3), (3,5)\}$

$S^2_1 = \{(5,4), (6,6), (7,7), (8,9)\}$ (Weight can't be 9)

$S^3 = \{(0,0), (1,2), (2,3), (3,5), (5,4), (6,6), 7,7\}$

$S^3_1 = \{(6,5), (7,7), (8,8), (11,9), (12,11), (13,12)\}$

$S^4 = \{(0,0), (1,2), (2,3), (5,4), (6,6), (7,7), (6,5), (7,7), (8,8)\}$

(8,8) ∉ S_4

but (8,8) ∉ S_3 → ①

(8-6, 8-5) = (2,3)

(2,3) ∉ S_3

but (2,3) ∉ S_2 — ② 0.

(2,3) ∉ S_2

but (2,3) ∉ S_1 — ①

(2-2), (3-3) = (0,0)

---

* Job sequencing. (with deadlines)

→ Job sequencing is an optimisation problem that involves scheduling a set of jobs on a single machine with a fixed deadline for each job.

→ Algorithm for Job sequencing

① Sort all jobs in ~~non~~ decreasing order based on profit.

② Initialise a schedule, all slots empty.

a) Starting from a deadline, find the first available slot.

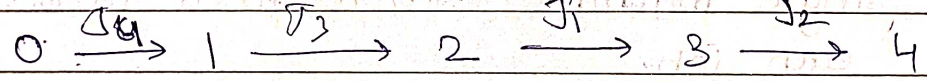
b) If slot exists, assign the job to that slot.

c) If not, skip the job.

④ Compute total profit.

Example

Jobs	J_1	J_2	J_3	J_4	J_5	J_6	J_7
profits	35	30	25	20	15	12	8
Deadlines	3	4	4	2	3	1	2



Job	slots assigned	profit
J_1	[2,3]	35
J_2	[2,3] [3,4]	35+30
J_3	[2,3] [3,4] [1,2]	35+30+25
J_4	[2,3] [3,4] [1,2]	35+30+25+20
J_5	[2,3] [3,4] [1,2]	35+30+25+~~20~~
J_6	[2,3] [3,4] [1,2] [0,1]	35+30+25+~~12~~
J_7	[2,3] [3,4] [1,2] [0,1]	35+30+25+12

100.

* Optimal Binary Search Tree (OBST)

→ OBST is a dynamic programming algorithm used to minimize the expected search cost by constructing a binary search tree.

→ The keys are stored in the internal nodes of the tree and the probability of accessing each key is given by probability distribution.

→ The OBST algorithm works by constructing a table of minimum expected search costs for all subtrees that can be formed from a given set of keys.

→ Algorithm with an example

	0	1	2	3	4
Q. Keys = {		10	20	30	40 }
p_i = {		4	2	6	3 } → Successful / Frequency
q_i = {					} → Unsuccessful

→ Step 1 - Sort all keys in ascending order

Keys = { 10, 20, 30, 40 }

Step 2 - Initialize a table.

	0	1	2	3	4
0	0	4	8^1	20^3	26^3
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

Step 3 - Fill all diagonal elements as 0 (since cost from (key, key) is 0)

Step 4 - Start calculating values (when $j-i=1$)

C[0,1]	C[1,2]	C[2,3]	C[3,4]
10	20	30	40
4	2	6	3

Step 5 → When $j - i = 2$.

a) $c[0,2]$

$$c[i,j] = \min_{i<k\leq j} \{ c[i,k-1] + c[k,j] \} + w(i,j)$$

2,3

$$\therefore c[0,2] = c[0,0] + c[1,2] + w(0,2)$$

$$0 + 2 + 6 = 8$$

or $c[0,1] + c[2,2] + w(0,2)$

$$= 4 + 0 + 6 = 10.$$

Minimum is 8.

b) $c[1,3]$

$$= c[1,1] + c[2,3] + w(1,3)$$

$$= 0 + 6 + 8 = 14$$

or $c[1,2] + [3,3] + w(1,3)$

$$2 + 8 = 10.$$

Similarly fill all the values in the table using the formula.

$$c[i,j] = \min_{i<k\leq j} \{ c[i,k-1] + c[k,j] \} + w(i,j$$

* Pseudocode for OBST (By Dynamic Programming)

```
function OBST ( Keys , probs )
    n = length ( keys )
    Sort ( keys )
    table = create Table (n+1)
    for i = 1 to n
        table [i][i] = probs [i]
    for l = 2 to real n+1
        j = i+l-1
        table [i][j] = ∞
        for k = i to j
            cost = expected cost ( keys, probs, i, j, k)
            if cost < table [i][j]
                table [i][j] = cost
    return table [1][n+1]

function expected cost ( keys, probs, i, j, k)
    cost = sum (probs [i:j+1])
    if k > i
        cost += OBST (keys [i:k-1], probs [i:k-1])
    if k < j
        cost += OBST ( keys [k+1
```

* Floyd - Warshall Algorithm.

→ The Floyd - Warshall Algorithm is a dynamic programming algorithm used to find the shortest path b/w all pairs of vertices in a weighted graph.

→ Algorithm of Floyd - Warshall Algorithm.

① Initialize a 2-D array 'dist' with size 'V x V'

② dist[i][i] = 0 (No self loop)

③ For each edge (u, v), dist[u][v] = weight[u][v]

④ check. for every vertex 'K' that if

dist[i][K] + dist[K][j] < dist[i][j]

for W(i, j)

* Pseudocode for Floyd - Warshall Algorithm.

```
function FloydWarshall (graph)

    dist = 2Darray (V, V)
    for i = 0 to V-1
        dist[i][i] = 0.
```

```
for K=1 to V
    for i = 1 to V
        for j=1 to V
            if dist[i][K] + dist[K][j] < dist[i][j]
                dist[i][j] = dist[i][K] + dist[K][j]

return dist.
```

---

* Longest common subsequence (LCS)

→ LCS involves finding the longest subsequence common to two sentences. (or strings)

→ Algorithm for LCS using DP.

① Initialize a 2-D array of size $(m+1) \times (n+1)$ (where m and n are the length of both strings)

② For each index 'i' from 0 to m and for each index 'j' from 0 to n set $L[i][j] = 0$.

③ For each index i from 1 to m and for each index j from 1 to n, check if the i^{th} character in m matches the j^{th} character in n.

m = "AARYA"

n = "AY"

j

④ If match is found set L[i][j] = L[i-1][j-1] + 1

⑤ If match is not found, set L[i][j] = max(L[i-1][j], L[i][j-1])

* Pseudocode for Longest common subsequence (LCS)

```
function LCS (X, Y)

    m = length(X)
    n = length(Y)

    ~~c[m][n]~~ c[m+1][n+1]

    for i = 0 to m
        c[i][0] = 0

    for j = 0 to n
        c[0][j] = 0

    for i = 0 to m-1
        for j = 0 to n-1
            if X[i] == Y[j]
                c[i][j] = c[i-1][j-1] + 1

            else
                c[i][j] = max(c[i][j-1], c[i-1][j])

    return c[m][n]
```

* Travelling Salesman Problem.

→ The Travelling Salesman Problem (TSP) is a classic optimization problem in computer science.

→ The goal is to find the shortest possible path that allows the salesman to visit each city and return to the starting city.

* Algorithm for TSP

① Create a 2-D array 'DP' of size $2^n \times n$, where n is the no. of cities in the TSP instance and 2^n represents all possible subsets of cities

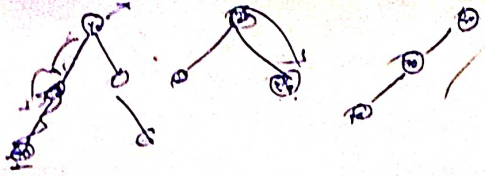
② Initialize all entries in 'DP' to ∞.

③ Set DP[{1}, 1] to 0, where {1} represents a set containing only the start city.

④ For each subset 'S' of cities of size greater than 1 and containing the start city.

- For each city j in 'S'
  → If j is the start city, move to next iteration

- For each city 'K' in 'S'.
  → If K=j move to next iteration.
  → Set DP[S,j] to the minimum of 'DP[S/{j}] + cost[K][j]'.



⑤ Set minDistance to ∞

⑥ For each j from 2 to n.

- Set minDistance to minimum of min Distance and DP[{1...n}, j] + cost[i][j]

⑦ Return minDistance.

* Pseudocode for TSP.

```
function TSP (cities)
    n = length (cities)
    subsets = generate_subsets (n)

    dp [2^n, n]

    for each subsets in subsets:
        for each city in subset:
            dp[subset, city] = ∞
    dp [{1}, 1] = 0.

    for m = 2 to n.
        for each subset in subsets
            for each city in subset
                if city is not start city
                dp dp[subset, city] = min_distance
                                (subset, city,
                                dp, cities)
```

```
minDistance = ∞
for each city in {2 . . . n}
    distance = dp[subset, city] + distance(city, 1, city)
    minDistance = min(minDistance, distance)
return minDistance
```