Minimum Spanning Tree (MST)

1. **What is a Spanning Tree?**
    - A spanning tree for a connected graph includes all vertices but uses only enough edges to form a tree (no cycles).
    - The edges in a spanning tree are a subset of the graph's edges.
2. **Minimum Spanning Tree (MST)**
    - An MST is a spanning tree with the smallest possible total weight for its edges.

Applications of MST

1. **Network Design**: Connecting locations (e.g., people, buildings) in the cheapest way without cycles.
2. **Optimizing Routes**: MST can help find the least costly routes between cities, which minimizes travel cost.
3. **set of circuit equations** for an electric network
4. **Communication Links**: If nodes represent cities and edges are possible connections, spanning trees show the minimal way (fewest links) to connect all cities.

MST Algorithms

Two main algorithms to find MST:

1. **Kruskal's Algorithm** (edge-based)
2. **Prim's Algorithm** (vertex-based)

Kruskal's Algorithm (Brief)

1. **Approach**: A "greedy" approach, choosing the smallest available edge at each step without forming cycles.
2. **Steps**:
    - Sort all edges by weight.
    - Starting with the smallest edge, add it if it doesn't create a cycle.
    - Stop once you've included n - 1 edges for a graph with n vertices.
3. **Running Time**:
    - Sorting edges takes O(eloge).
    - Adding edges without cycles takes O(n).
    - Total time complexity: O(eloge).

Prim's

- Prim's Algorithm is used to find the Minimum Spanning Tree (MST) of a graph
- It starts from one vertex and grows the MST by adding one edge at a time.

Prim's algorithm is a **greedy algorithm** used to find the **Minimum Spanning Tree (MST)** of a graph. It starts with a single vertex and continuously adds the smallest edge that connects a vertex already in the MST to a vertex outside the MST.

Key Idea:

- The algorithm starts from any vertex and expands the tree by adding the smallest edge connecting the current MST to a new vertex.
- It maintains an array near[] to keep track of the nearest vertex in the MST for each vertex not yet included in the MST.
- It updates this array as the tree expands to ensure the smallest edge is always chosen next.

Steps:

1. **Initialize the MST**:
    - Start with an edge of minimum cost from the graph and include its two vertices in the MST.

- Initialize the near[] array to track the nearest vertex in the MST for each vertex.

2. **Select the Next Edge**:
   - From the near[] array, find the smallest edge (j, near[j]) where j is a vertex not in the MST.
   - Add this edge to the MST and update the total cost.
3. **Update the near[] Array**:
   - For each vertex not in the MST, update its nearest vertex in the MST if a smaller-cost edge is found.
4. **Repeat**:
   - Continue the process until all vertices are included in the MST.

Time Complexity:

- The basic implementation has a time complexity of **$O(n^2)$**, where n is the number of vertices.
- Using advanced data structures like heaps or red-black trees can reduce the complexity to **$O((n + E) \log n)$**, where E is the number of edges.

Algorithm Prim(E, cost, n, t)
Input:
- E: Set of edges in graph G
- cost: Adjacency matrix with cost[i, j] indicating the cost of edge (i, j)
- n: Number of vertices

Output:
- t: Set of edges in the Minimum Spanning Tree (MST)

```
1. Let (k, l) be an edge of minimum cost from E
2. mincost := cost[k, l]
3. t[1, 1] := k; t[1, 2] := l
4. Initialize near[] array:
   for i from 1 to n do
      if cost[i, l] < cost[i, k] then
         near[i] := l
      else
         near[i] := k
5. Set near[k] := 0 and near[l] := 0 (vertices k and l are now part of the MST)

6. for i from 2 to n-1 do
   Find vertex j such that near[j] ≠ 0 and cost[j, near[j]] is minimum
   Add edge (j, near[j]) to t
   mincost := mincost + cost[j, near[j]]
   Set near[j] := 0 (include j in MST)

   for each vertex k from 1 to n do
      if near[k] ≠ 0 and cost[k, near[k]] > cost[k, j] then
         Update near[k] := j

7. return mincost
```

Kruskal's

Objective: Kruskal's algorithm is a greedy algorithm used to find a **Minimum Cost Spanning Tree (MST)** for a given graph. An MST is a subset of the graph's edges that connects all the vertices together without forming cycles and has the minimum possible total edge cost.

Kruskal's Algorithm (Simple Version)

Here is the cleaned-up and simplified version of Kruskal's algorithm:

Algorithm: Kruskal's Minimum Cost Spanning Tree

Input:

- E: Set of edges in the graph G.
- cost[u, v]: Cost of edge between vertices u and v.
- n: Number of vertices in the graph.

Output:

- t: Set of edges in the minimum cost spanning tree.
- mincost: Total minimum cost of the spanning tree.

Steps:

1. **Initialize**:
    - Construct a **Min-Heap** from all edges based on their costs.
    - Create a parent array for union-find operations. Set each vertex as its own parent (parent[i] = -1 for all i from 1 to n).
    - Set i = 0 (to count edges added to MST) and mincost = 0 (to keep track of the total minimum cost).
2. **While Loop**:
    - Repeat the following steps until the MST has n - 1 edges or the heap is empty:
        1. **Extract the Minimum Edge**:
            - Remove the edge (u, v) with the smallest cost from the heap.
        2. **Find the Set of Each Vertex**:
            - Use the Find function to check the set (or parent) of vertices u and v.
        3. **Check for Cycle**:
            - If u and v are in different sets (Find(u) != Find(v)):
                - Add edge (u, v) to the MST.
                - Increment the edge count i.
                - Add the edge's cost to mincost.
                - **Union** the sets of u and v (connect them).
            - If u and v are in the same set, discard this edge (to avoid forming a cycle).
3. **Check Completion**:
    - If i < n - 1, then the graph is not connected (no spanning tree possible).
    - Otherwise, return the mincost as the total minimum cost of the spanning tree.

Algorithm Kruskal(E, cost, n, t)

```
1. Construct a Min-Heap of all edges based on their costs.
2. Initialize parent[i] = -1 for all i from 1 to n.
3. i = 0, mincost = 0.0
4. while (i < n - 1) and (heap is not empty):
   a. (u, v) = delete minimum cost edge from heap
   b. j = Find(u)
   c. k = Find(v)
   d. if (j != k):
      i = i + 1
      t[i, 1] = u
```

```
        t[i, 2] = v
        mincost = mincost + cost[u, v]
        Union(j, k)
5. if (i < n - 1):
     write "No spanning tree"
6. else:
     return mincost
```

Data Structures Used

1. **Sorting the Edges**:
   - Sorting all the edges initially takes **O(E log E)** time.
2. **Disjoint Set Union (DSU)**:
   - The DSU helps in efficiently determining whether two vertices belong to the same connected component (using **Find**).
   - It also merges two sets when a new edge is added (using **Union**).
   - Both **Find** and **Union** operations have nearly constant time complexity using the **Union-Find with Path Compression** technique.

Time Complexity

- **O(E log E)**: Sorting or creating a heap of edges.
- **O(E log V)**: Union-find operations while processing edges.
- **Total Complexity**: **O(E log V)**, suitable for sparse graphs.

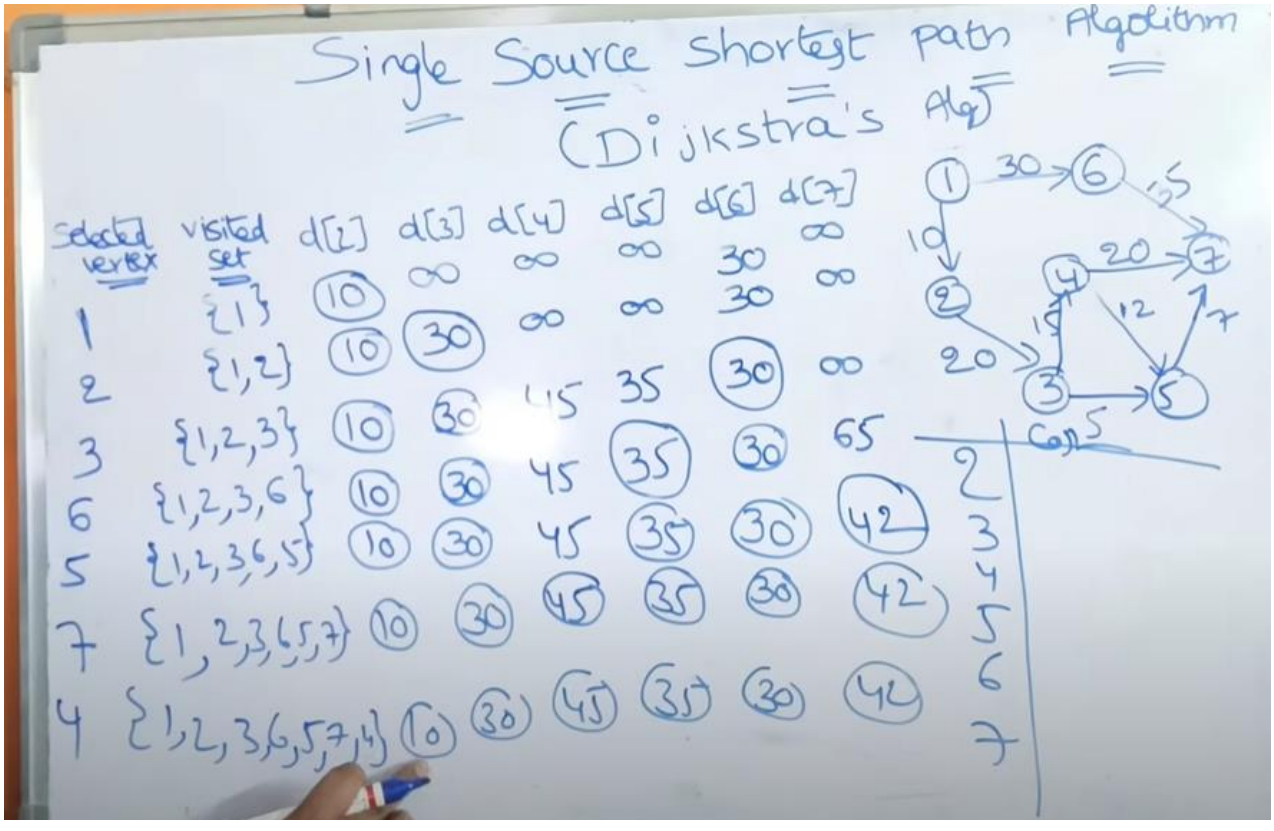
Why Kruskal's Algorithm is Effective:

- Kruskal's algorithm is best for **sparse graphs** (graphs with fewer edges).
- It efficiently finds the MST by processing edges in increasing order of cost and avoids cycles by checking connected components using the DSU.

Summary

- Kruskal's algorithm starts by sorting edges and picks the smallest edge while avoiding cycles using a union-find data structure.
- It builds the MST by connecting unconnected parts until all vertices are included, ensuring the minimum cost.

SINGLE SOURCE SHORTEST PATH



The **goal** is to find the shortest path from a starting city (source node) to every other city in the roadmap.

Key Concepts:

- **Source Node** (v): The starting point from where the shortest paths are calculated.
- **Distance Array** (dist): Stores the shortest known distance from the source node to every other node.
- **Visited Set** (S): Keeps track of nodes whose shortest paths have already been found.

Algorithm Explanation (Dijkstra's Algorithm):

1. **Initialization:**
    - Set the distance of the source node (v) to itself as **0** (dist[v] = 0).
    - Set the distance of all other nodes to a very large number (∞), indicating that they are initially unreachable.
    - Mark all nodes as **unvisited** (not in set S).
2. **Iterative Process:**
    - While there are unvisited nodes:
        - Select the **unvisited node** with the **smallest known distance** (u).
        - Mark this node (u) as **visited** (add it to set S).
        - For each **neighbor** (w) of node u that is still unvisited:
            - Check if going from the source to w **through u** offers a shorter path than previously known.
            - If yes, **update** the shortest known distance to node w.
3. **Completion:**
    - The process repeats until all nodes have been visited or the shortest paths to all reachable nodes are found.
    - The final distance array (dist) holds the shortest distances from the source node to every other node.

Algorithm Dijkstra(u, cost, dist, n)

1. Initialize:
  - For each node i from 1 to n:
    S[i] = false // No node has been visited yet
    dist[i] = cost[u, i] // Distance from source to i
  - Set S[u] = true // Mark the source node as visited
  - Set dist[u] = 0 // Distance to source is zero

2. For num = 2 to n: // Repeat for all nodes except the source
  - Choose the node u not in S with the smallest dist[u]
  - Mark u as visited (S[u] = true)

  - For each neighbor w of u that is not visited:
    - If dist[w] > dist[u] + cost[u, w]:
      - Update dist[w] = dist[u] + cost[u, w] // Found a shorter path

4. End

Optimal Merge Patterns

Concept: When you have multiple sorted files and you want to merge them into a single sorted file, the process of merging takes time. The goal of **Optimal Merge Patterns** is to find a way to merge these files such that the total merging time is minimized.

Why does the merging time vary? Merging two sorted files of sizes n and m takes about n + m steps. So, the time depends on the sizes of the files being merged. If we merge larger files first, it may take more time. Thus, we need to plan the merging order to minimize the total time.

Example

Imagine you have three files:

- File A with 30 records
- File B with 20 records
- File C with 10 records

There are two possible ways to merge:

1. Merge File A (30) and File B (20) first:
    - This takes 30 + 20 = 50 steps.
    - Then merge the result (50 records) with File C (10):
    - This takes 50 + 10 = 60 steps.
    - **Total steps**: 50 + 60 = 110 steps
2. Merge File B (20) and File C (10) first:
    - This takes 20 + 10 = 30 steps.
    - Then merge the result (30 records) with File A (30):
    - This takes 30 + 30 = 60 steps.
    - **Total steps**: 30 + 60 = 90 steps

Clearly, the second option is better as it takes fewer steps (90 vs. 110).

Greedy Approach

To find the optimal way to merge files:

1. **Pick the two smallest files** and merge them first.
2. **Repeat** this process until all files are merged into one.

Example: If you have five files with sizes: 5, 10, 15, 20, 25

- First merge 5 and 10 → Resulting size = 15
- Then merge 15 (result) and 15 → Resulting size = 30
- Then merge 20 and 25 → Resulting size = 45
- Finally, merge 30 and 45 → Resulting size = 75

By always merging the smallest pairs first, you ensure that the total merging time is minimized.

Find an optimal binary merge pattern for ten files whose lengths are 4, 2, 9, 7, 13, 3 and 5

(or)

Obtain a set of optimal Huffman codes for the messages (M1, M2, …, M7) with relative frequencies (q1,q2,…,q7) = (4, 2, 9, 7, 13, 3, 5). Draw the decode tree for this set of codes

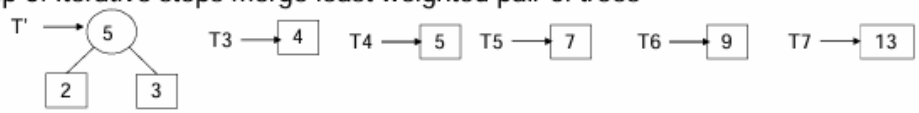
Solution:

Step 1: Sort the files with respect of its length.

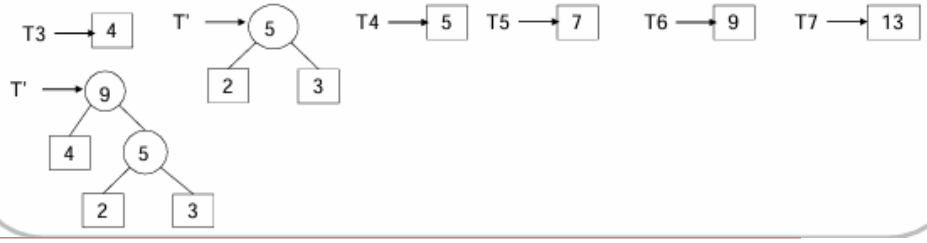
(i.e.) 2, 3, 4, 5, 7, 9, 13.

Step 2: Create a Forest:

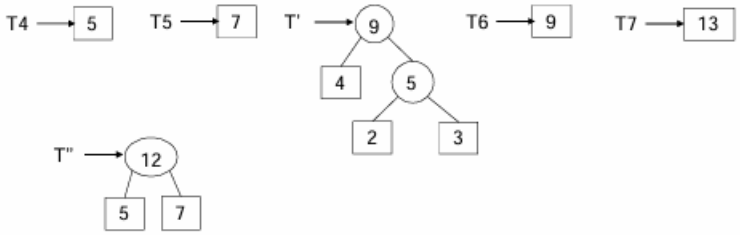
Step 3: Iterative steps merge least weighted pair of trees



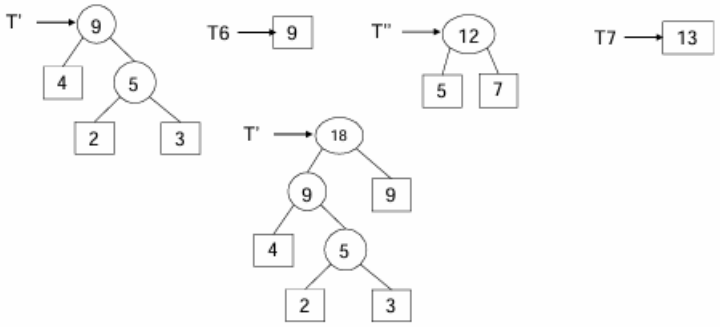
Step 4: Iterative steps merge least weighted pair of trees



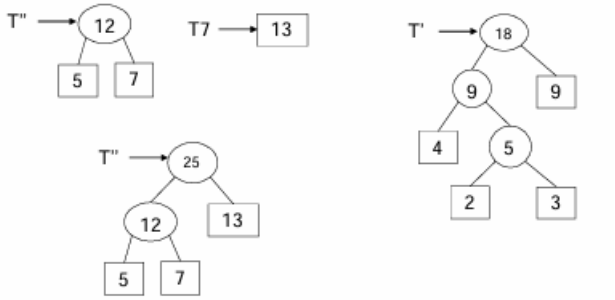
Step 5: Iterative steps merge least weighted pair of trees



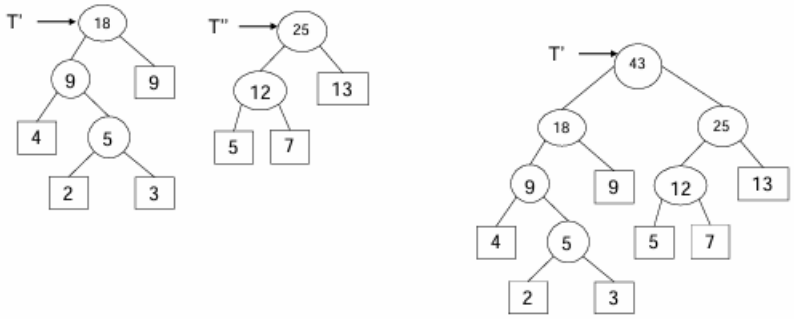
Step 6: Iterative steps merge least weighted pair of trees



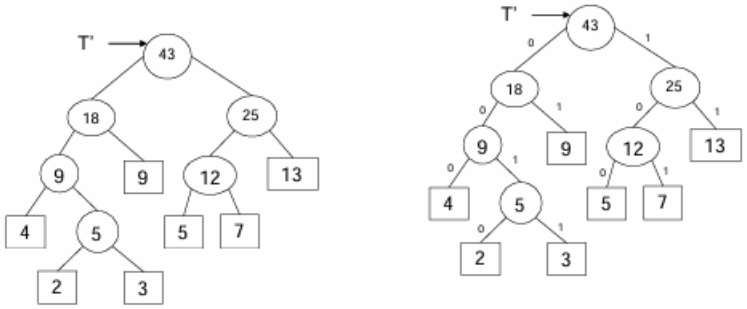
Step 7: Iterative steps merge least weighted pair of trees



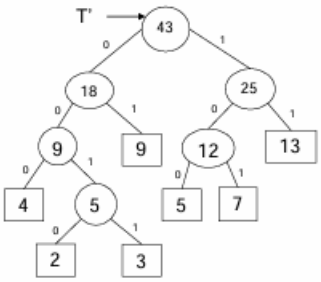
Step 8: Iterative steps merge least weighted pair of trees



Step 9: Labeled edges left branch 0 and right branch 1

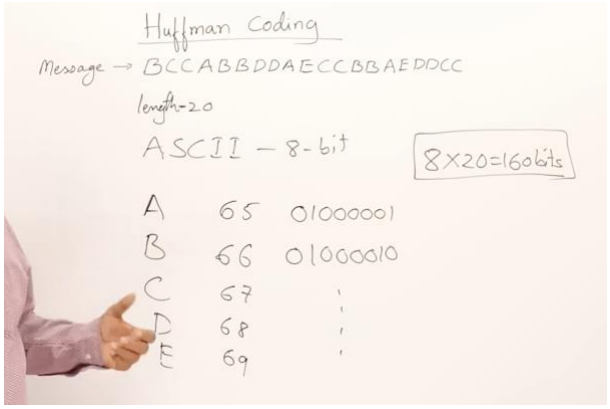


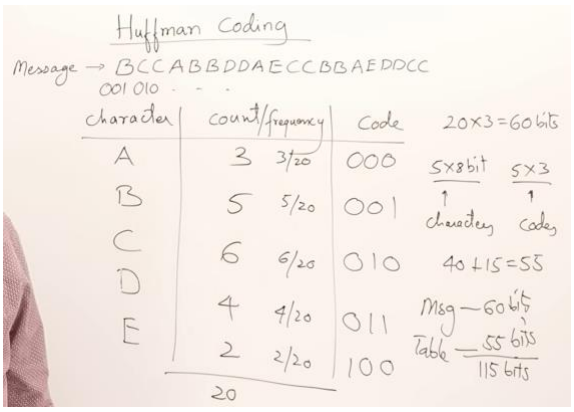
Step 10: Huffman Coding

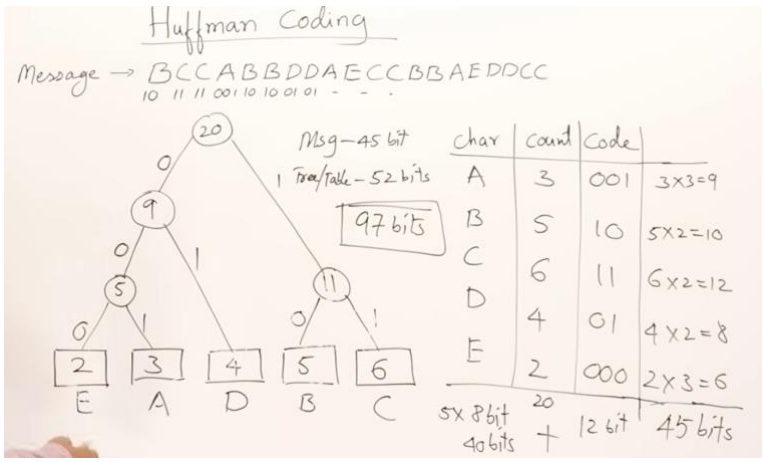


File	Size	Huffman Code
1	4	000
2	2	0010
3	9	01
4	7	101
5	13	11
6	3	0011
7	5	100

Huffman coding







Huffman Coding: An Overview

Huffman coding is an efficient method used in data compression to reduce the size of data by encoding it based on the frequency of characters. It uses a binary tree to assign shorter codes to more frequent characters, optimizing the transmission and storage of data.

Problem Description

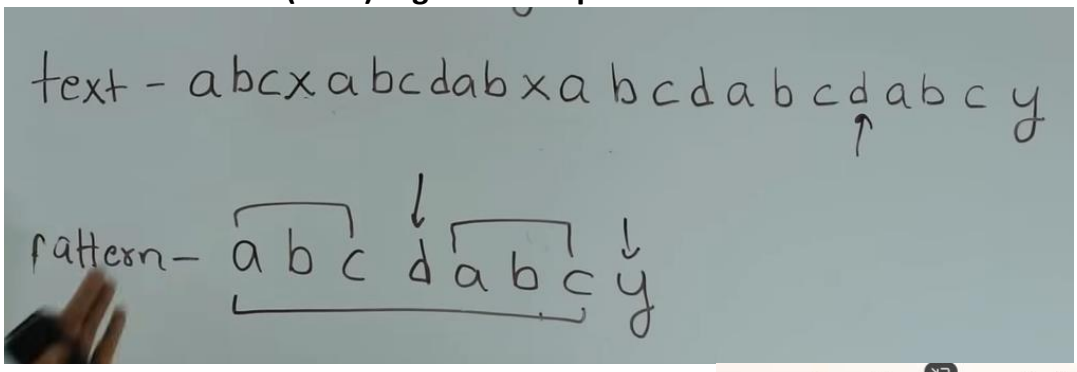
In telecommunications, data is often represented as binary sequences (0s and 1s). The challenge is to minimize the size of these sequences while maintaining accurate representation, especially when different messages have varying frequencies. The **goal** is to use fewer bits for frequently occurring characters and more bits for less frequent characters.

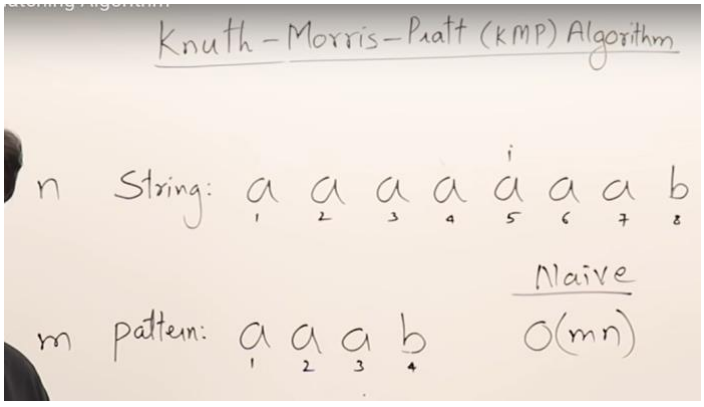
This problem can be solved using a **greedy approach** by constructing an optimal binary tree, known as the Huffman tree.

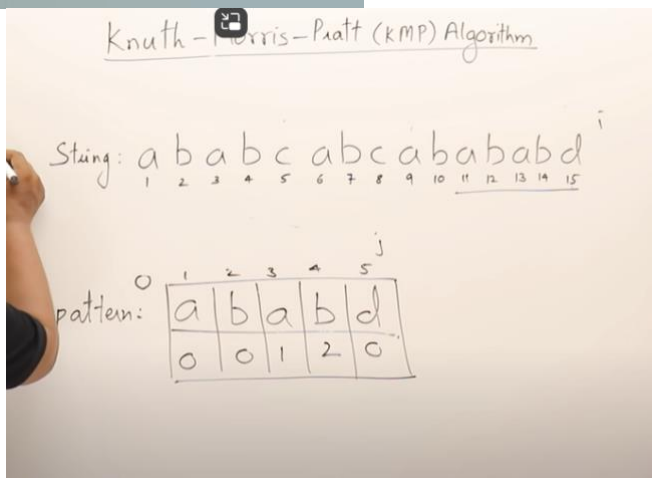
Steps to Construct Huffman Tree

1. **Frequency Table Creation**:
   - Count the frequency of each character in the message.
2. **Forest Initialization**:
   - Create a list where each character is a separate binary tree node with its frequency as the weight.
3. **Tree Merging**:
   - Select two nodes with the smallest frequencies.
   - Merge them into a new binary tree node, with the combined frequency as its weight.
   - Repeat until only one node (the root) remains, forming the complete Huffman tree.
4. **Code Assignment**:
   - Traverse the Huffman tree.
   - Assign 0 to the left branch and 1 to the right branch recursively to generate the Huffman codes for each character.

Knuth-Morris-Pratt (KMP) Algorithm: Explanation







The **Knuth-Morris-Pratt (KMP)** algorithm is an efficient string searching technique used to find the occurrence of a pattern in a given text. It avoids redundant comparisons by using information about the pattern itself.

Key Concepts:

1. **Pattern Matching:**
   - Given a **text** T of length n and a **pattern** P of length m, the goal is to find all occurrences of P in T.
2. **Prefix Function (LPS Array):**

- The **LPS (Longest Prefix Suffix)** array is used to store the longest prefix which is also a suffix for each substring of the pattern. This helps us to skip unnecessary comparisons in the text.

Algorithm Steps:

1. **Preprocessing Phase:**
   - Construct the **LPS array** for the pattern.
2. **Searching Phase:**
   - Use the LPS array to search for the pattern in the text without going back in the text.

Algorithm: Knuth-Morris-Pratt (KMP)

Algorithm KMP(T, P):
Input: Text T of length n, Pattern P of length m
Output: Starting indices of all occurrences of P in T

1. Construct the LPS array for the pattern P.
   a. Initialize LPS array: lps[] of size m.
   b. ComputeLPS(P, lps)

2. Set i = 0 (index for T), j = 0 (index for P)

3. While i < n:
   a. If T[i] == P[j]:
      i++
      j++
   b. If j == m:
      - Pattern found at index i - j
      - Set j = lps[j - 1] (to continue searching for other occurrences)

   c. Else if i < n and T[i] != P[j]:
      - If j != 0, set j = lps[j - 1]
      - Else, i++

4. End of algorithm

Computing the LPS Array

Algorithm ComputeLPS(P, lps):
Input: Pattern P of length m
Output: LPS array of size m

1. Initialize length = 0, lps[0] = 0, i = 1

2. While i < m:
   a. If P[i] == P[length]:
      length++
      lps[i] = length
      i++
   b. Else:
      If length != 0:
         length = lps[length - 1]

Else:
  lps[i] = 0
  i++

3. End of algorithm

Text (T): "ABBCADABCB"
Pattern (P): "ABC"
Step 1: Compute the LPS (Longest Prefix Suffix) Array
The **LPS** array helps us skip unnecessary comparisons.

Index	0	1	2
P	A	B	C
LPS	0	0	0

Explanation:

- For i = 0, there's no proper prefix or suffix, so LPS[0] = 0.
- For i = 1, the prefix "A" and suffix "B" do not match, so LPS[1] = 0.
- For i = 2, the prefix "AB" and suffix "BC" do not match, so LPS[2] = 0.

LPS Array: [0, 0, 0]
Step 2: Search Using the KMP Algorithm
We use the LPS array to search the pattern in the text.

Text (T)	A	B	B	C	A	D	A	B	C	B
Pattern (P)	A	B	C							
Matching indexes	0	1	2							

Detailed Steps:

1. **i = 0, j = 0**:
   - T[0] = P[0] = A → Match! Increment i and j.
2. **i = 1, j = 1**:
   - T[1] = P[1] = B → Match! Increment i and j.
3. **i = 2, j = 2**:
   - T[2] = B, P[2] = C → Mismatch. Reset j to lps[j - 1] = lps[1] = 0.
4. **i = 2, j = 0**:
   - T[2] = B, P[0] = A → Mismatch. Increment i to 3.
5. **i = 3, j = 0**:
   - T[3] = C, P[0] = A → Mismatch. Increment i to 4.
6. **i = 4, j = 0**:
   - T[4] = A, P[0] = A → Match! Increment i and j.
7. **i = 5, j = 1**:
   - T[5] = D, P[1] = B → Mismatch. Reset j to lps[j - 1] = lps[0] = 0.
8. **i = 5, j = 0**:
   - T[5] = D, P[0] = A → Mismatch. Increment i to 6.
9. **i = 6, j = 0**:
   - T[6] = A, P[0] = A → Match! Increment i and j.
10. **i = 7, j = 1**:
    - T[7] = B, P[1] = B → Match! Increment i and j.
11. **i = 8, j = 2**:
    - T[8] = C, P[2] = C → Match! Increment i and j.
12. **i = 9, j = 3**:
    - j has reached the length of the pattern (j = m), so we found a match at index i - j = 6.

Reset j using LPS: j = lps[j - 1] = lps[2] = 0.

13. **i = 9, j = 0**:
    - T[9] = B, P[0] = A → Mismatch. Increment i to 10.

Result:

The pattern "ABC" is found at **index 6** in the text "ABBCADABCB".

Summary:

- The LPS array helped us avoid redundant comparisons by resetting j correctly after mismatches.
- The pattern "ABC" was matched at index 6 in the text.
- This small example shows how the KMP algorithm efficiently finds the pattern with fewer comparisons than a naive approach.

The **time complexity** of this algorithm is O(n + m), where n is the length of the text and m is the length of the pattern.

BruteForce

```
function bruteForcePatternMatch(T, P):
   n = length(T)
   m = length(P)

   for i from 0 to n - m:
      j = 0
      while j < m and T[i + j] == P[j]:
         j = j + 1
      if j == m:
         return i  // Pattern found at position i
   return -1  // Pattern not found
```

UNIT - 4

Control Abstraction in Dynamic Programming

In **Dynamic Programming (DP)**, problems are solved by breaking them into smaller subproblems, using two main approaches:

1. **Forward Approach**:
   - Start solving from the **beginning** towards the **end**.
   - However, the solution is typically built **backwards**, starting from the end result and using recurrence relations.
2. **Backward Approach**:
   - Start solving from the **end** towards the **beginning**.
   - Here, the solution is often built **forwards**, using initial values to find the optimal solution.

Steps to Solve a DP Problem:

1. **Find Recurrence Relations**: Define formulas to solve the problem using smaller subproblems.
2. **Use a Multistage Graph**: Visualize the problem in stages, where each stage represents a decision or subproblem.

Summary: Choose a direction (forward or backward), define recurrence relations, and use them to find the optimal solution step by step.

```
Algorithm OBST(p, q, n)
Input:
    - p[1..n]: probabilities for keys
    - q[0..n]: probabilities for dummy keys
    - n: number of keys
Output:
    - c[0..n, 0..n]: cost of optimal binary search trees
    - r[0..n, 0..n]: root of optimal subtrees

1. Initialize:
    for i ← 0 to n do
        w[i, i] ← q[i]
        c[i, i] ← 0
        r[i, i] ← 0
    end for

2. Compute optimal trees for larger subtrees:
    for m ← 1 to n do         // m = number of keys in the subtree
        for i ← 0 to n - m do
            j ← i + m
            w[i, j] ← w[i, j-1] + p[j] + q[j]
            c[i, j] ← ∞

            for k ← i to j do
                cost ← c[i, k-1] + c[k+1, j] + w[i, j]
                if cost < c[i, j] then
                    c[i, j] ← cost
                    r[i, j] ← k
                end if
            end for
        end for
    end for

3. Output:
    return c[0, n], r[0, n]
```

Reliability Design using Dynamic programming

Design a 3 stage system with device types D_1, D_2, D_3. The costs are 30, 15, 20. The cost of the system is to be no more than 105. The reliability of each device type is 0.9, 0.8, 0.5

D_i	C_i	r_i	u_i
D_1	30	0.9	
D_2	15	0.8	
D_3	20	0.5	

Sol) $C_1=30, C_2=15, C_3=20, C=105,$
$r_1=0.9, r_2=0.8, r_3=0.5$

To calculate upper bound of each device (or) stage

$$u_i = \text{floor}\left(C + C_i - \sum_{j=1}^{n} C_j\right) / C_i$$

$u_1 = \text{floor}(105+30-(C_1+C_2+C_3))/30 \Rightarrow (135-(30+15+20))/30 \Rightarrow 70/30 \Rightarrow 2$

$u_2 = \text{floor}(105+15-(65))/30 \Rightarrow 55/15 \Rightarrow 3$

$u_3 = \text{floor}(105+20-(65))/20 \Rightarrow 60/20 \Rightarrow 3$

Reliability Design using Dynamic programming

S_i^j $i \rightarrow$ Stage no, $j \rightarrow$ no. of copies

D_i	C_i	r_i	u_i
D_1	30	0.9	2
D_2	15	0.8	3
D_3	20	0.5	3

Initially $S^0 = \{(r, c)\}$
$= \{(1, 0)\}$

Stage 1 (or) Consider D_1:-

$u_1 = 2, \quad S^1 = S_1^1 \cup S_2^1$

Consider $S_1^1 \quad m_1 = 1$

$1-(1-r_1)^{m_1} \Rightarrow 1-(1-0.9)^1 \Rightarrow 1-0.1 \Rightarrow 0.9$

$= (0.9, 30) \Rightarrow (0.9*1, 30+0) \Rightarrow (0.9, 30)$

Consider $S_2^1 \quad m_1 = 2$

$1-(1-r_1)^{m_1} \Rightarrow 1-(1-0.9)^2 \Rightarrow 1-(0.1)^2 \Rightarrow 1-0.01 \Rightarrow 0.99 \Rightarrow (0.99, 60)$

$(0.99*1, 60+0) = (0.99, 60)$

Reliability Design using

Stage 2 (a) Consider D_2 $\therefore u_2 = 3$

$S^2 = S_1^2 \cup S_2^2 \cup S_3^2$

$S_1^2 \Rightarrow m_2 = 1$

$1-(1-r_2)^{m_2} \Rightarrow 1-(1-0.8)^1 \Rightarrow 1-0.2 = 0.8$

$(0.8, 15)$

$S_1^2 \Rightarrow \{(0.8*0.9, 15+30), (0.8*0.99, 15+60)\}$

$\Rightarrow \{(0.72, 45), (0.792, 75)\}$

$S_2^2 \Rightarrow m_2 = 2$

$1-(1-r_2)^{m_2} \Rightarrow 1-(1-0.8)^2 \Rightarrow 1-0.04 \Rightarrow 0.96$

$(0.96, 30)$

$S^2 = \{(0.96*0.9, 30+30), (0.96*0.99, 30+60)\}$

$\{(0.864, 60), (0.9504, 90)\}$

$S^2 = S_1^2 \cup S_2^2 \cup S_3^2$

$= \{(0.72, 45), (0.792, 75), (0.864, 60), (0.8928, 75), (0.9504, 90)\}$

$S^1 = \{(0.9, 30), (0.99, 60)\}$

$S_3^2 \Rightarrow m_2 = 3$

$1-(1-r_2)^{m_2} \Rightarrow 1-(1-0.8)^3$

$\Rightarrow 1-0.2^3 \Rightarrow 1-0.008 \Rightarrow 0.992$

$(0.992, 45)$

$S_3^2 = \{(0.992*0.9, 45+30), (0.992*0.99, 45+60)\}$

$S_3^2 = \{(0.8928, 75), (0.9848, 105)\}$

Reliability Design using

Stage 3 (a) Consider D_3 $\therefore u_3 = 3$

$S^3 = S_1^3 \cup S_2^3 \cup S_3^3$

$S_1^3 \therefore m_3 = 1$

$1-(1-r_3)^{m_3} \Rightarrow 1-(1-0.5)^1 \Rightarrow 0.5$

$(0.5, 20)$

$\Rightarrow \{(0.5*0.72, 20+45), (0.5*0.864, 20+60), (0.5*0.8928, 20+75)\}$

$= \{(0.36, 65), (0.432, 80), (0.4464, 95)\}$

$S_2^3 \Rightarrow m_3 = 2$

$1-(1-r_3)^{m_3} \Rightarrow 1-0.5^2 = 1-0.25 = 0.75$

$(0.75, 40)$

$= \{(0.75*0.72, 40+45), (0.75*0.864, 40+60), (0.75*0.8928, 40+75)\}$

$= \{(0.54, 85), (0.648, 100)\}$

$S^1 = \{(0.9, 30), (0.99, 60)\}$

$S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

$S^3 = \{(0.36, 65), (0.432, 80), (0.54, 85), (0.648, 100)\}$

$S_3^3 = m_3 = 3$

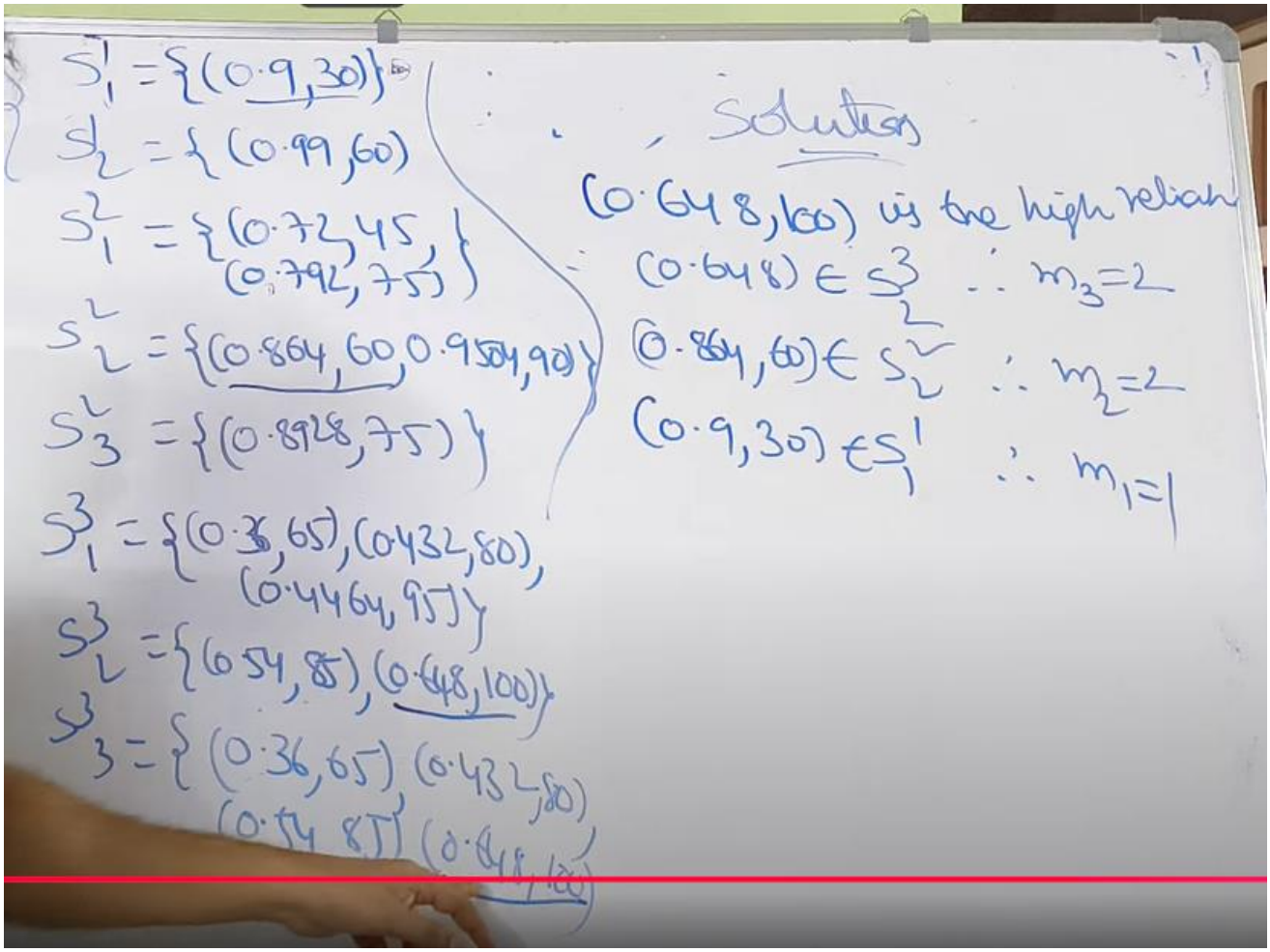
$1-(1-r_3)^{m_3} = 1-0.5^3$

$= 1-0.125 = (0.875, 60)$

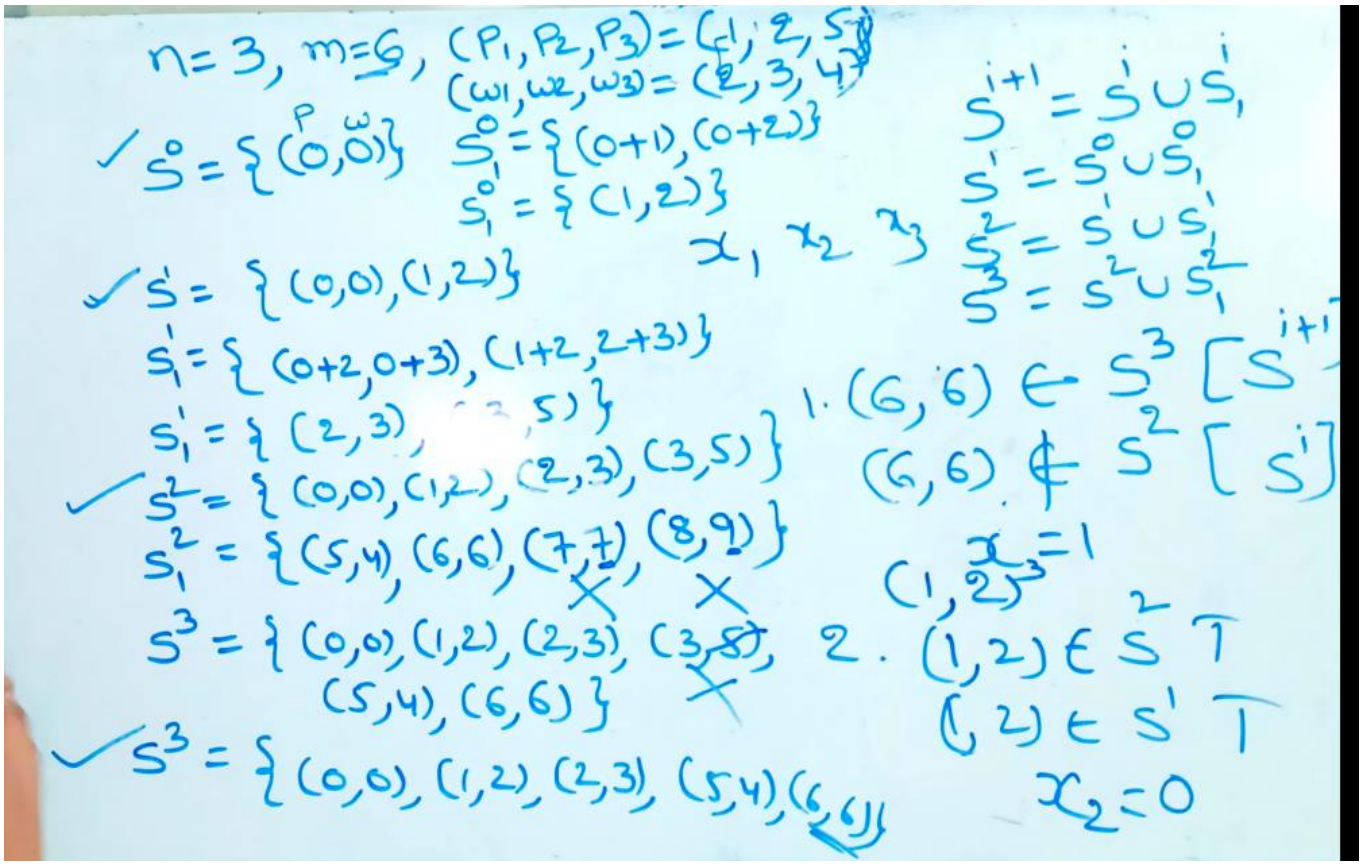
$= \{(0.875*0.72, 60+45), (0.875*0.864, 60+60), (0.875*0.8928, 75+60)\}$

$= \{(0.63, 105)\}$

$S^3 = \{(0.36, 65), (0.432, 80), (0.4464, 95), (0.54, 85), (0.63, 105), (0.648, 100)\}$



0/1 knapsack



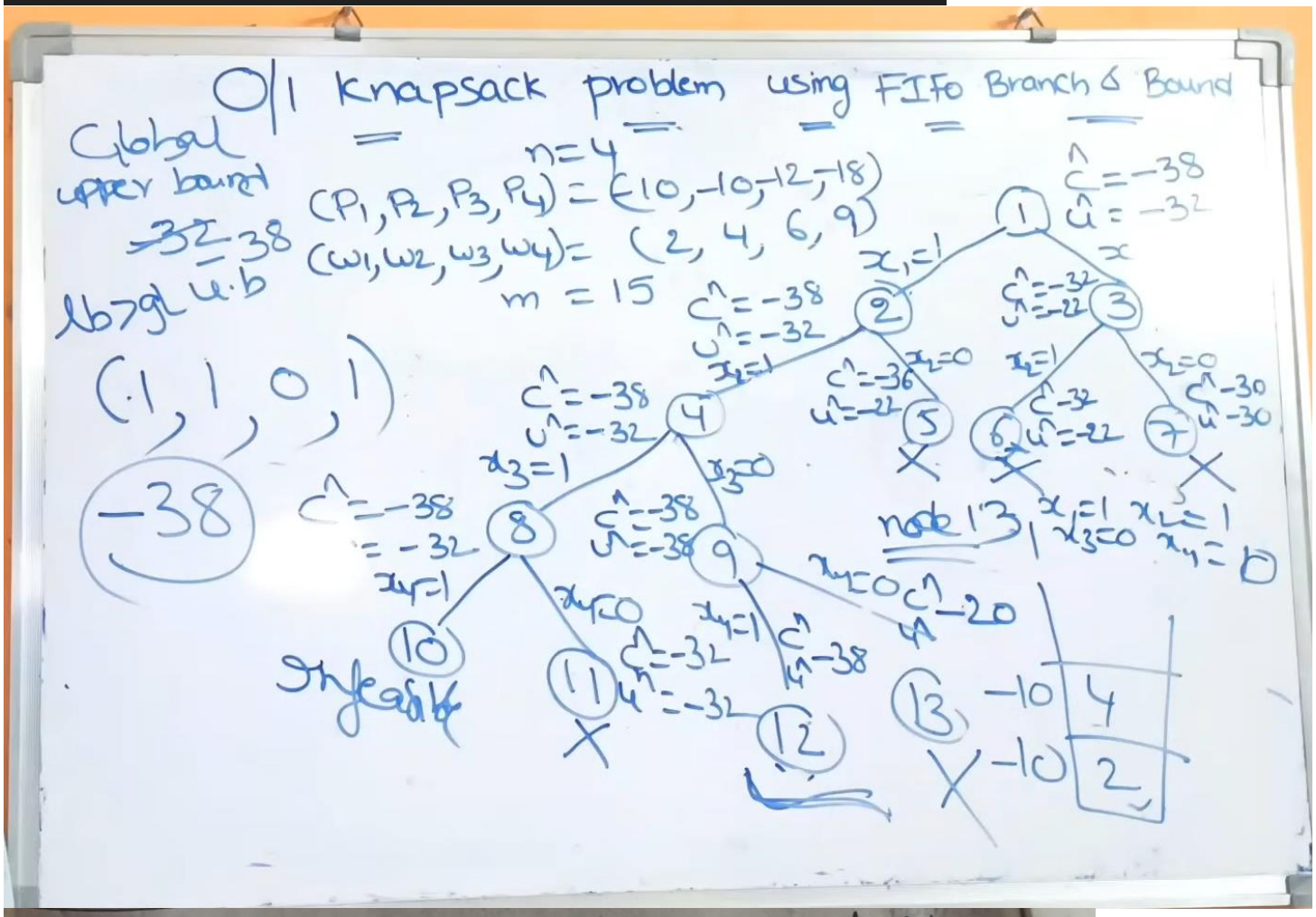
Refer notes

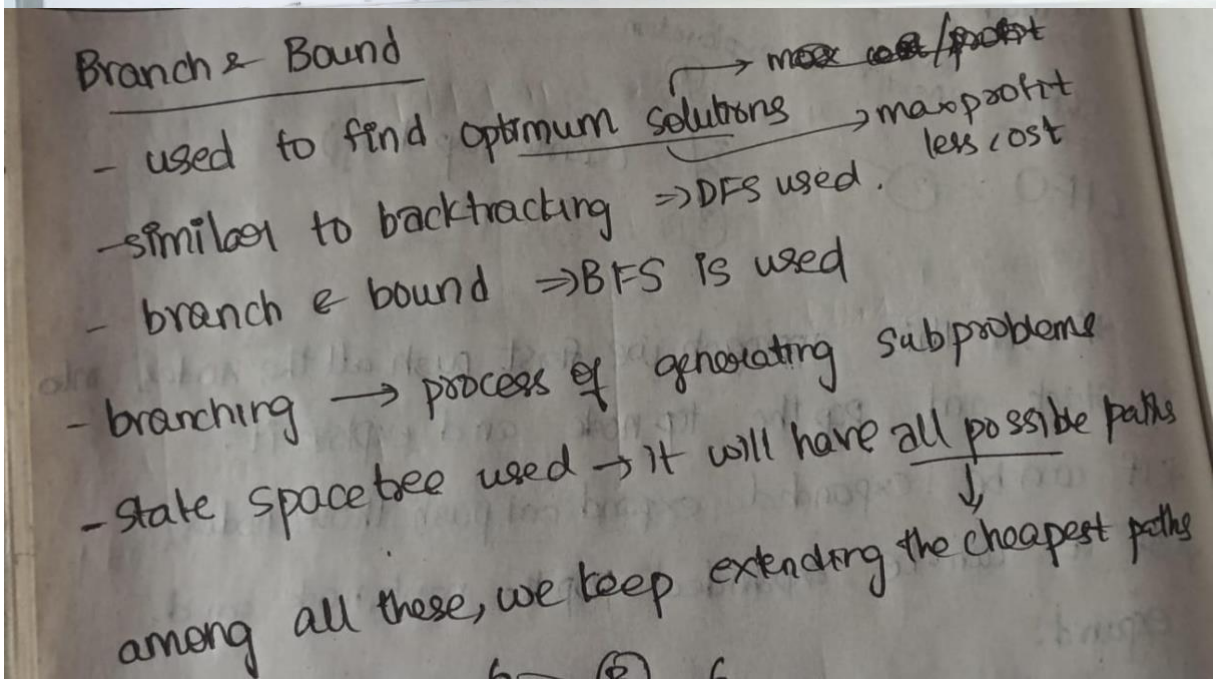
```java
import java.util.*;
public class NQueensColumns {
    public List<List<Integer>> solveNQueens(int n) {
        List<List<Integer>> solutions = new ArrayList<>();
        List<Integer> current = new ArrayList<>();
        solve(row:0, n, new HashSet<>(), new HashSet<>(), new HashSet<>(), current, solutions);
        return solutions;
    }
    private void solve(int row, int n, Set<Integer> cols, Set<Integer> posDiag, Set<Integer> negDiag,
            List<Integer> current, List<List<Integer>> solutions) {
        if (row == n) {
            // Add the current solution (list of column numbers)
            solutions.add(new ArrayList<>(current));
            return;
        }
        for (int col = 0; col < n; col++) {
            if (cols.contains(col) || posDiag.contains(row + col) || negDiag.contains(row - col)) {
                continue;
            }
            // Place the queen in this column
            current.add(col);
            cols.add(col);
            posDiag.add(row + col);
            negDiag.add(row - col);
            // Recur to the next row
            solve(row + 1, n, cols, posDiag, negDiag, current, solutions);
            // Backtrack
            current.remove(current.size() - 1);
            cols.remove(col);
            posDiag.remove(row + col);
            negDiag.remove(row - col);
        }
    }

    Run | Debug
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print(s:"Enter the number of queens (n): ");
        int n = sc.nextInt();

        NQueensColumns nQueens = new NQueensColumns();
        List<List<Integer>> results = nQueens.solveNQueens(n);

        if (results.isEmpty()) {
            System.out.println(x:"No solution exists.");
        } else {
            System.out.println(x:"Solutions (columns of queens in each row):");
            for (List<Integer> solution : results) {
                System.out.println(solution);
            }
        }
    }
}
```

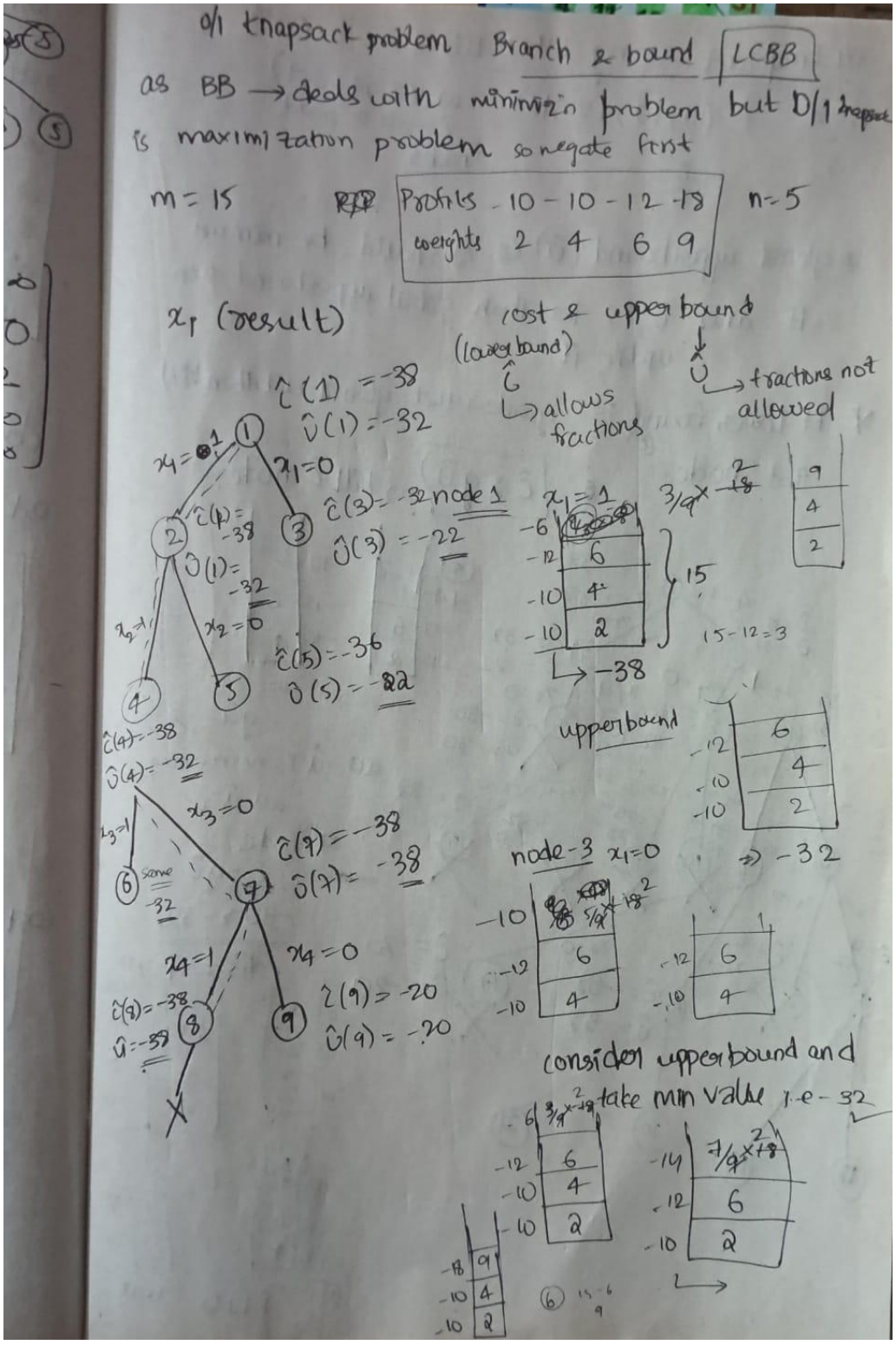
Travelling sales person

Summary

Approach	Time Complexity	Space Complexity
Brute Force	$O(n!)$	$O(n)$
Dynamic Programming (Held-Karp)	$O(n^2 \cdot 2^n)$	$O(n \cdot 2^n)$
Nearest Neighbor (Greedy)	$O(n^2)$	$O(n)$
Christofides Algorithm	$O(n^3)$	$O(n^2)$







A **game tree** is a **graphical representation** of all possible moves in a game, structured like a tree:

- **Nodes:** Represent game states.
- **Edges:** Represent moves between states.
- **Leaf Nodes:** End states of the game (e.g., win, loss, or draw).

Key Points:

1. **Tree Size:**
   - Small games like **Tic Tac Toe** can have their full tree constructed (225,000 leaf nodes).
   - Large games like **chess** have massive trees, making it impossible to construct fully.
2. **Heuristics:**
   - For large games, only a part of the tree is explored.
   - **Heuristic functions** estimate the quality of moves.

3. **Minimax Algorithm:**
    - Used to find the best moves.
    - Alternates between:
        - **Maximizer:** Tries to maximize the score.
        - **Minimizer:** Opponent trying to minimize the score.

Examples:

1. **Tic Tac Toe:**
    - Simple game where the entire tree can be built.
    - Perfect play is achievable using algorithms like Minimax.
2. **Stone-Pile Game:**
    - Alice and Bob remove stones from a pile with specific rules.
    - A game tree shows all possible outcomes based on their moves.

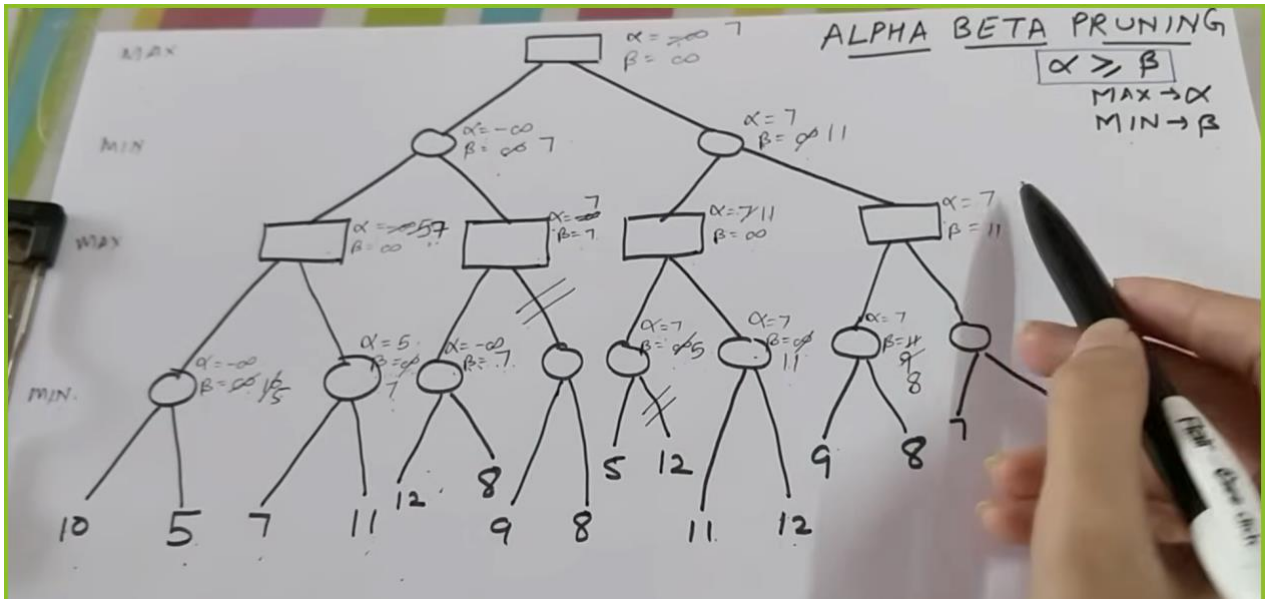
Conclusion:
Game trees help analyze games by simulating moves and predicting outcomes. They are crucial for AI but often limited by the size of the game. Algorithms like Minimax and heuristics make intelligent play possible even for complex games.

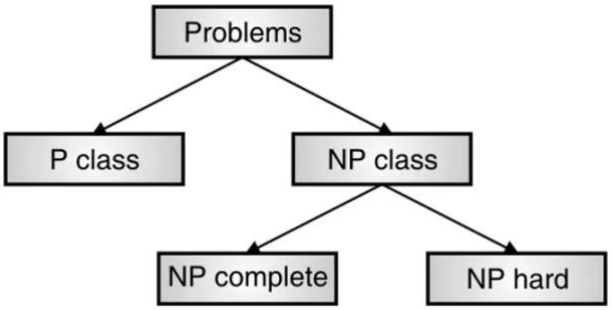
Alpha-Beta Pruning

Alpha-Beta Pruning is an optimization technique for the **Minimax algorithm** used in game theory to reduce the number of nodes evaluated in a game tree.
Key Points:

1. **Minimax Algorithm**: It explores all possible moves to find the best outcome, with a time complexity of O(B^D), where B is the branching factor and D is the depth of the tree.
2. **Alpha-Beta Pruning**: It "cuts off" or prunes branches that are not needed to make the final decision, reducing the number of nodes explored.
3. **Alpha (α\alphaα)**: The best value that the **Max** player can guarantee at that point.
4. **Beta (β\betaβ)**: The best value that the **Min** player can guarantee at that point.
5. **Pruning**: If at any point α≥β\alpha \geq \betaα≥β, further exploration of the branch is stopped because it won't affect the final decision.
6. **Time Complexity**: In the best case, it reduces time complexity to O(B^{D/2}), making the search process much faster.

This method helps in making the decision process more efficient by ignoring paths that don't influence the final result.





Sr. No.	P Problems	NP Problems
1.	P problems are set of problems which can be solved in polynomial time by deterministic algorithms.	NP problems are problems which can be solved in nondeterministic polynomial time.
2.	P Problems can be solved and verified in polynomial time	The solution to NP problems cannot be obtained in polynomial time, but if the solution is given, it can be verified in polynomial time.
3.	P problems are a subset of NP problems	NP Problems are a superset of P problems

Sr. No.	P Problems	NP Problems
4.	All P problems are deterministic in nature	All the NP problems are non-deterministic in nature
5.	**Example:** Selection sort, Linear search	**Example:** TSP, Knapsack problem