1. Graph Coloring using Backtracking (2 Colors)

Problem Statement: We are given a graph defined by the adjacency matrix:

- A is connected to B, C.
- B is connected to A, D.
- C is connected to A, D.
- D is connected to B, C.

This forms a cycle structure: **A — B — D — C — A**. We must color it using 2 colors: **Red (R)** and **Black (B)**.

Step-by-Step State Space Tree Process:

The Backtracking algorithm uses a **Depth First Search (DFS)** approach. We assign a color to a vertex, check for validity (safety), and if valid, move to the next vertex. If we reach a point where no color can be assigned safely, we backtrack.

Constraints: No two adjacent vertices can have the same color.

Step 1: Color Vertex A

- We start with vertex A.
- Option 1: Color **A = Red**. (Valid, as it has no colored neighbors yet).

Step 2: Color Vertex B (Neighbor of A)

- Neighbors of B: A (Colored Red), D (Uncolored).
- Try **B = Red**: Conflict with A (Red). **Backtrack.**
- Try **B = Black**: Valid.
- **Current State:** A=Red, B=Black.

Step 3: Color Vertex C (Neighbor of A and D)

- Neighbors of C: A (Colored Red), D (Uncolored).
- Try **C = Red**: Conflict with A (Red). **Backtrack.**
- Try **C = Black**: Valid.
- **Current State:** A=Red, B=Black, C=Black.

Step 4: Color Vertex D (Neighbor of B and C)

- Neighbors of D: B (Colored Black), C (Colored Black).
- Try **D = Red**: Valid (Neighbors B and C are Black).
  - **Solution Found: {A: Red, B: Black, C: Black, D: Red}**.

Verification:

- A(R) - B(B) $\rightarrow$ OK.
- A(R) - C(B) $\rightarrow$ OK.
- B(B) - D(R) $\rightarrow$ OK.
- C(B) - D(R) $\rightarrow$ OK.

State Space Diagram: The tree explores choices level by level (Vertex A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D).

Plaintext

```
                (Root)
               /      \
           A=R          A=B
          /   \        (Symmetric branch)
       B=R     B=B
       (X)    /   \
          C=R      C=B
          (X)     /   \
               D=R     D=B
              (SOL)   (X-Conflict with B,C)
```

(Note: (X) denotes a bounded/pruned node due to color conflict)

---

2. Proof: Nodes in Sum of Subsets State Space Tree

Theorem: The full state space tree for finding a sum of subsets of n elements using backtracking will have $2^n - 1$ internal nodes (excluding leaf nodes).

Proof:

1. **Tree Structure:** The state space tree for the "Sum of Subsets" problem is a **binary tree**. At each level i (representing item i), there are two branches:
   - **Left Branch ($x_i = 1$):** Include the element w_i in the subset.
   - **Right Branch ($x_i = 0$):** Exclude the element w_i from the subset.
2. **Depth and Levels:**

- **Level 0:** Root node (1 node).
- **Level 1:** 2 nodes (Include/Exclude item 1).
- ...
- **Level n:** These are the leaf nodes representing the final decision for the n-th item.

3. **Counting Leaves:** Since every level doubles the number of nodes from the previous level, the number of leaf nodes at level n is:

$$L = 2^n$$

4. **Total Nodes in a Full Binary Tree:** The total number of nodes N in a perfect binary tree of depth n is given by the geometric series sum:

$$N = 2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$$

5. **Calculating Internal Nodes:** The question asks for the number of nodes **excluding the leaf nodes**.

$$\text{Internal Nodes} = \text{Total Nodes} - \text{Leaf Nodes}$$

$$\text{Internal Nodes} = (2^{n+1} - 1) - 2^n$$

We know that $2^{n+1} = 2 \cdot 2^n$.

$$\text{Internal Nodes} = (2 \cdot 2^n - 1) - 2^n$$

$$\text{Internal Nodes} = 2^n(2 - 1) - 1$$

$$\text{Internal Nodes} = 2^n - 1$$

Conclusion: Thus, the number of nodes in the state space tree excluding the leaves is $2^n - 1$.

---

3. Drawbacks of Branch and Bound Method

The Branch and Bound (B&B) method is a powerful search technique for optimization problems, but it has significant limitations:

1. **Exponential Time Complexity:** In the worst-case scenario, B&B may explore almost all nodes in the state space tree, leading to exponential time complexity ($O(2^n)$), similar to

exhaustive search. It does not guarantee polynomial time.

2. **High Memory Consumption:**
   - Unlike Backtracking (which uses Depth First Search and consumes linear stack space), Branch and Bound often uses **Breadth First Search (BFS)** or **Least Cost Search**.
   - These strategies require maintaining a priority queue of live nodes. As the tree grows, the number of live nodes can become huge, causing memory overflow.
3. **Complexity of Bounding Functions:** Designing an efficient bounding function (to calculate upper/lower bounds) is difficult. A weak bound will not prune the tree effectively, while a complex bound calculation might slow down the processing of each node.
4. **Not Suitable for Decision Problems:** B&B is specifically designed for **Optimization Problems** (Minimization/Maximization). It is not typically used for decision problems (like finding *any* solution to N-Queens) where Backtracking is preferred.

---

4. Sum of Subsets Problem (Algorithm & Solution)

Problem Statement: Given a set of non-negative integers and a value `Sum`, find all subsets of the given set that add up to exactly `Sum`.

Algorithm (Backtracking):

C

```c
Algorithm SumOfSubsets(s, k, r)
// s: current sum of selected elements
// k: index of the current element being considered
// r: sum of remaining elements in the set
{
    // Generate left child (Include w[k])
    x[k] = 1;
    if (s + w[k] == Sum) {
        Print subset defined by x[1...k]; // Solution found
    }
    else if (s + w[k] + w[k+1] <= Sum) {
        // Pruning: Check if adding next smallest element exceeds Sum
        SumOfSubsets(s + w[k], k + 1, r - w[k]);
    }

    // Generate right child (Exclude w[k])
    // Pruning Condition: Even if we take all remaining (r - w[k]),
    // can we reach the Sum? AND is s + w[k+1] safe?
```

```
    if ((s + r - w[k] >= Sum) && (s + w[k+1] <= Sum)) {
        x[k] = 0;
        SumOfSubsets(s, k + 1, r - w[k]);
    }
}
```

Solving the Instance: Input: `Set = {2, 3, 5, 6, 8, 10}`, `Target Sum = 10`. Sorted Input: `{2, 3, 5, 6, 8, 10}`.

Trace:

1. **Start:** Root (Sum=0).
2. **Path 1 (Include 2):** Rem sum = 8.
   - Include 3 $\rightarrow$ Sum=5. Rem=5.
     - Include 5 $\rightarrow$ Sum=10. **Solution 1: {2, 3, 5}**.
   - Exclude 3 $\rightarrow$ Sum=2.
     - Include 5 $\rightarrow$ Sum=7.
       - Include 6 (Sum=13 > 10) $\rightarrow$ Backtrack.
       - Exclude 6 $\rightarrow$ Sum=7.
         - Include 8 (Sum=15 > 10) $\rightarrow$ Backtrack.
     - Exclude 5 $\rightarrow$ Sum=2.
       - Include 8 $\rightarrow$ Sum=10. **Solution 2: {2, 8}**.
3. **Path 2 (Exclude 2):** Sum=0.
   - Try starting with 3... (3+5=8... cannot reach 10 without 2 unless 3+something else. 3+6=9. 3+8 > 10).
   - Try starting with 10.
     - Include 10 $\rightarrow$ Sum=10. **Solution 3: {10}**.

Final Subsets: `{2, 3, 5}`, `{2, 8}`, `{10}`.

Time Complexity: The state space tree is binary. In the worst case, we generate 2^n nodes. Complexity: $O(2^n)$.

---

5. 0/1 Knapsack using LC Branch and Bound

Given Data:

- Capacity (M) = 7.
- Items:

Object	Weight (w_i)	Value (p_i)	Ratio (p_i/w_i)
O1	5	6	1.2
O2	4	5	1.25
O3	3	4	1.33

Export to Sheets

LC Branch and Bound Strategy: LC (Least Cost) B&B uses a ranking function to decide which node to explore next. For Maximization (Knapsack), we convert it to a minimization problem by inverting profits (Cost = -Profit) or using "Upper Bound" of profit as the priority. We explore the node with the **Highest Upper Bound** (most promising node).

Cost Function (Upper Bound) Calculation: For a node, the Upper Bound is calculated by filling the remaining capacity with the best available items (using Fractional Greedy approach). $UB = \text{Current Profit} + \text{Fractional Profit of remaining items}.$

Step-by-Step Execution:

1. **Preprocessing:** Sort by Ratio (Descending).
   - Order: O3 (1.33), O2 (1.25), O1 (1.2).
   - Sorted Items: $I_1(3, 4)$, $I_2(4, 5)$, $I_3(5, 6)$.
2. **Root Node (A):**
   - Capacity: 7.
   - Greedy Fill:
     - Take I_1 (Weight 3, Val 4). Rem Cap = 4.
     - Take I_2 (Weight 4, Val 5). Rem Cap = 0.
     - Full.
   - $UB(A) = 4 + 5 = 9.$
3. **Branching from Root (Item I_1):**
   - **Node B (Include I_1):**
     - Weight so far = 3, Profit = 4, Rem Cap = 4

    - Weight so far = 3. Profit = 4. Rem Cap = 4.
    - Calculate UB: Can we fit I_2? Yes. $4 + 5 = 9$.
    - $UB(B) = 9$.
  - **Node C (Exclude I_1):**
    - Weight = 0. Profit = 0. Rem Cap = 7.
    - Calculate UB:
      - Take I_2 (Wt 4, Val 5). Rem Cap = 3.
      - Take fraction of I_3: $(3/5) \times 6 = 3.6$.
      - Total: $5 + 3.6 = 8.6$.
    - $UB(C) = 8.6$.
4. **Selection:** Node B (UB=9) > Node C (UB=8.6). **Explore B.**
5. **Branching from B (Item I_2):**
  - **Node D (Include I_2):**
    - Current: $\{I_1, I_2\}$. Weight = $3 + 4 = 7$. Profit = $4 + 5 = 9$.
    - Rem Cap = 0. Cannot take I_3.
    - This is a leaf/feasible solution. **Profit = 9**.
  - **Node E (Exclude I_2):**
    - Current: $\{I_1\}$. Weight = 3. Profit = 4. Rem Cap = 4.
    - Calculate UB:
      - Take fraction of I_3: $(4/5) \times 6 = 4.8$.
      - Total: $4 + 4.8 = 8.8$.
    - $UB(E) = 8.8$.
6. **Pruning:**
  - We found a solution at Node D with Profit 9.
  - Node E has an Upper Bound of 8.8. Since $8.8 < 9$, Node E can never beat our current best. **Prune E**.
  - Node C has an Upper Bound of 8.6. Since $8.6 < 9$, **Prune C**.

Final Solution: Items: I_1 and I_2 (Original O3 and O2). Total Value = 9. Total Weight = 7.

---

6. Graph Coloring (3 Colors) - Recursive Backtracking

Given Graph (Adjacency Matrix Analysis): Nodes: A, B, C, D, E, F, G. Connections:

- A: B, C
- B: A, D, E
- C: A, F, G
- D: B
- E: B
- F: C
- G: C

Algorithm Logic: Function `GraphColoring(vertex_index)`:

1. If all vertices colored, print solution.
2. For color $c = 1$ to 3 (R, G, B):
   - If `IsSafe(vertex, c)`:
     - Assign color c to vertex.
     - Recursively call `GraphColoring(vertex_index + 1)`.
     - If recursion returns true, return true.
     - (Backtrack) Reset color of vertex.

Solution Trace (One valid assignment):

1. **Vertex A:** Assign **Red**.
2. **Vertex B:** (Neighbor of A). Cannot be Red. Assign **Green**.
3. **Vertex C:** (Neighbor of A). Cannot be Red. Assign **Green** (Valid, C is not connected to B).
4. **Vertex D:** (Neighbor of B). Cannot be Green. Assign **Red**.
5. **Vertex E:** (Neighbor of B). Cannot be Green. Assign **Red** (or Blue).
6. **Vertex F:** (Neighbor of C). Cannot be Green. Assign **Red**.
7. **Vertex G:** (Neighbor of C). Cannot be Green. Assign **Red**.

Result Configuration:

- A: Red
- B: Green

- C: Green
- D: Red
- E: Red
- F: Red
- G: Red

(Note: Blue was not even strictly needed for this specific graph structure, as it is a set of trees rooted at A, but 3 colors were available).

---

7. Note on LC Branch and Bound

LC (Least Cost) Branch and Bound: LC Branch and Bound is a variation of the Branch and Bound technique used to solve optimization problems more efficiently than standard FIFO or LIFO Branch and Bound.

Key Characteristics:

1. **Search Strategy:** Instead of exploring the tree in a rigid Breadth-First (FIFO) or Depth-First (LIFO) manner, LC B&B selects the "most promising" node to expand next.
2. **Cost Function ($\hat{c}$):** It assigns a cost or rank to every live node.
   - For Minimization: The node with the **lowest** cost/bound is selected (Least Cost).
   - For Maximization: The node with the **highest** upper bound is selected.
3. **Priority Queue:** A min-priority queue (or max-priority queue) is used to store live nodes, ordered by their cost function.
4. **Efficiency:** By expanding the most promising nodes first, LC B&B often reaches the optimal solution faster and prunes larger sections of the tree compared to FIFO B&B.

---

8. Drawbacks of Branch and Bound Method

(Refer to Answer 3. The points are identical).

1. **Exponential Worst-Case Time Complexity.**
2. **Large Memory Requirement** (Maintenance of the Priority Queue).
3. **Difficulty in formulating strong bounding functions.**

---

9. m-Coloring Algorithm & Complexity

Problem: Given a graph with n vertices and an adjacency matrix G, color the vertices using at most m colors such that no adjacent vertices share the same color.

Recursive Backtracking Algorithm:

C

```c
Algorithm mColoring(k)
// k is the index of the vertex being colored
{
    repeat
    {
        // Generate a legal color for vertex k
        NextValue(k);

        if (x[k] == 0) return; // No color possible, backtrack

        if (k == n)
        {
            Print(x); // Solution found
        }
        else
        {
            mColoring(k + 1); // Proceed to next vertex
        }
    } until (false);
}

Algorithm NextValue(k)
{
    repeat
    {
        x[k] = (x[k] + 1) mod (m + 1); // Try next color
        if (x[k] == 0) return; // All colors exhausted

        // Check for conflict with neighbors
        for (j = 1; j <= n; j++)
        {
            // If edge exists (G[k][j]) and neighbor j has same color x[k]
            if (G[k][j] != 0 and x[k] == x[j])
                break; // Conflict found
        }
        if (j == n + 1) return; // No conflict found, color is valid
    } until (false);
}
```

Time Complexity:

- For each vertex, there are m color choices.
- We have n vertices.
- The size of the state space tree is the total number of nodes in a full m-ary tree of depth n.
- Total nodes $\approx m^n$.
- **Time Complexity:** $O(m^n)$.

---

10. Comparison: Backtracking vs. Branch and Bound

Feature	Backtracking	Branch and Bound
1. Search Technique	Uses **Depth First Search (DFS)** traversal.	Uses **Breadth First Search (BFS)** or **Least Cost Search**.
2. State Space Tree	Explores the tree depth-wise. Moves to a child, then deeper, before backtracking.	Explores level-by-level (FIFO) or jumps to the most promising node (LC).
3. Problem Type	Used for **Decision Problems** (Does a solution exist?) and Enumeration (Find all solutions).	Used primarily for **Optimization Problems** (Minimize Cost / Maximize Profit).
4. Solution Approach	Finds **all** feasible solutions or the first one it encounters.	Finds the **optimal** solution by pruning sub-optimal paths.
5. Pruning Mechanism	Prunes using **Bounding Functions** (Feasibility/Constraint checks only).	Prunes using **Lower/Upper Bounds** (Cost estimation) + Feasibility.
6. Complexity	Space Complexity is low: $O(n)$ (stack space).	Space Complexity is high: **Exponential** (queue stores all live nodes).
7. Node Traversal	Recursive nature; easier to implement.	Iterative nature; requires a Priority Queue.
8. Example Problems	N-Queens, Graph Coloring, Hamiltonian Cycle.	Traveling Salesperson (TSP), 0/1 Knapsack, Job Scheduling.

Export to Sheets

---

11. Sum of Subsets: Finding First Solution

Problem: Set $A = \{5, 10, 15, 20, 25\}$. Target Sum = 30. Find the **first** solution using

Backtracking and show the space tree.

Sorted Set: $\{5, 10, 15, 20, 25\}$.

Execution Trace (DFS):

1. **Root (Sum=0)**
2. **Level 1 (Item 5):**
   - **Include 5:** (Current Sum = 5). Remaining Needed = 25.
3. **Level 2 (Item 10):**
   - **Include 10:** (Current Sum = $5 + 10 = 15$). Remaining Needed = 15.
4. **Level 3 (Item 15):**
   - **Include 15:** (Current Sum = $15 + 15 = 30$).
   - **Target Reached!**

First Solution Found: {5, 10, 15}.

State Space Tree (Partial):

Plaintext

```
                (Root: Sum 0)
               /       \
            (Inc 5)   (Exc 5)
             Sum=5       ...
            /      \
        (Inc 10) (Exc 10)
         Sum=15     ...
        /       \
    (Inc 15)   (Exc 15)
     Sum=30     ...
    (SOL)
```

The algorithm stops immediately after finding the first valid leaf node `{5, 10, 15}` and does not explore the branch `{5, 25}` or `{10, 20}` unless requested to find *all* solutions.