(14) Negative weight cycle → Bellman Ford Algorithm.

→ To check the presence of negative weight cycle using Bellman Ford Algorithm.

TC → O(V*E)
SC → O(V)

code →

```cpp
class Solution {
public:
    int isNegativeWeightCycle(int n, vector<vector<int>>edges){
        vector<int>dis(n,INT_MAX);
        // initially, dist to src is 0
        dis[0] = 0;
        // relax n-1 times
        for(int i=0;i<n-1;i++)
        {
            for(auto edge:edges)
            {
                int src = edge[0];
                int dest = edge[1];
                int wt = edge[2];
                if(dis[src]!=INT_MAX) // to avoid integer overflow
                    dis[dest] = min(dis[dest],dis[src]+wt);
            }
        }
        // final relaxation
        for(auto edge:edges)
        {
            int src = edge[0];
            int dest = edge[1];
            int wt = edge [2];
            if(dis[src]!=INT_MAX && dis[dest]>dis[src]+wt)
              return 1;
        }
        return 0;
    }
};
```

(15) Floyd Warshall Algorithm

→ All source shortest path & -ve edges allowed.

→ Since its all source shortest path we need to run the loop for all nodes, considering it as intermediatery vertex.

→ $cost[i][j] = min(cost[i][j], cost[i][k] + cost[k][j])$

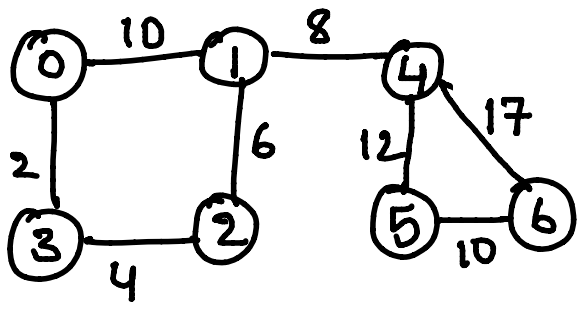
TC → $O(N^3)$ SC → $O(N^2)$

Code →

```
class Solution {
  public:
    void shortest_distance(vector<vector<int>>&matrix){
        int V = matrix.size();
        vector<vector<int>>costs(matrix.size(),vector<int>(matrix.size()));

        for(int i=0;i<V;i++)
            for(int j=0;j<V;j++)
                costs[i][j] = matrix[i][j];

        for(int k=0;k<V;k++)
            for(int i=0;i<V;i++)
                for(int j=0;j<V;j++){
                    // if intermediate is not -1 then
                    if(costs[i][k]!=-1 && costs[k][j]!=-1){
                        if(costs[i][j]==-1)
                            costs[i][j] = costs[i][k]+costs[k][j];
                         else
                            costs[i][j] = min(costs[i][j], costs[i][k]+costs[k][j]);
                    }
                }

        for(int i=0;i<V;i++)
            for(int j=0;j<V;j++)
                matrix[i][j] = costs[i][j];

    }
};
```

(16) Prim's Algorithm → Minimum spanning Tree (MST)

Eg



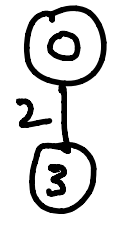
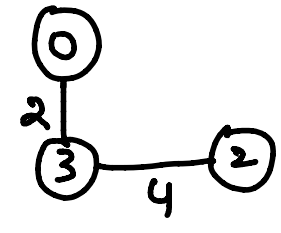
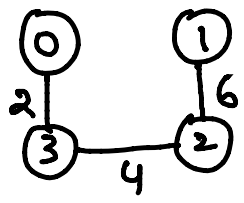
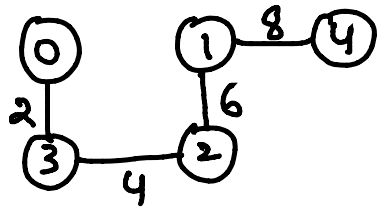
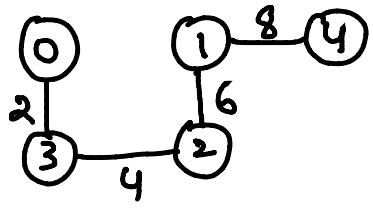
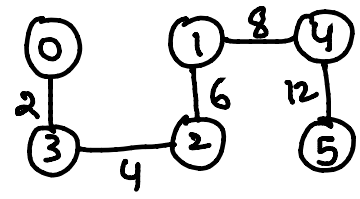
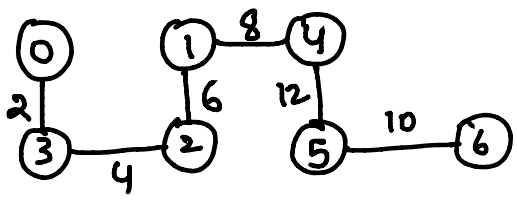
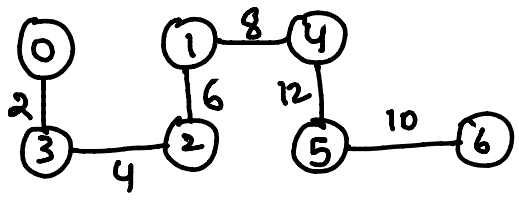
Vis = { }

pq (node, weight)

↑ returns node with lowest cost/weight.

* To find MST, just push node along with its weight.

→

pq	Vis	
(0,0)	{ }	
(1,10) (3,2)	{0}	
(1,10) (2,4)	{0,3}	
(1,10) (1,6)	{0,3,2}	
(1,10) (4,8)	{0,3,2,1}	
(1,10) (5,12) (6,17)	{0,3,2,1,4}	
(5,12) (6,17)	{0,3,2,1,4}	
(6,17) (6,10)	{0,3,2,1,4,5}	
(6,17)	{0,3,2,1,4,5,6}	
	{0,3,2,1,4,5,6}	

TC → O(V + ElogV)

SC → O(V)

Code →

```cpp
class Solution
{
    public:
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[])
    {
        int minCost = 0;
        vector<int>costs(V,INT_MAX);
        costs[0] = 0;
        vector<bool>vis(V, false);
        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
        pq.push({0,0}); // {cost, Node}

        while(!pq.empty())
        {
            pair<int,int> p = pq.top();
            int currNode = p.second;
            int currCost = p.first;
            pq.pop();

            if(vis[currNode])    continue;

            minCost += currCost;

            vis[currNode] = true;
            costs[currNode] = currCost;

            for(int i=0;i<adj[currNode].size();i++)
            {
                int neighbourNode = adj[currNode][i][0];
                int neighbourNodeCost = adj[currNode][i][1];
                if(vis[neighbourNode])  continue;
                pq.push({neighbourNodeCost, neighbourNode});
            }
        }
        return minCost;
    }
};
```

(17) Min cost to connect all points

→ Create graph with each node containing Wt & Node value

$Wt = abs(x_i - x) + abs(y_i - y)$

→ Perform Prims algo.

$Tc \rightarrow O(V + E\log V)$

$Sc \rightarrow O(V)$

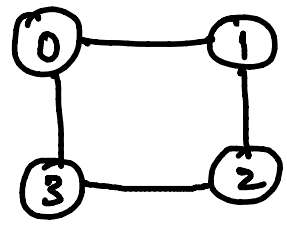
code →

```cpp

class Solution {
public:
    int minCostConnectPoints(vector<vector<int>>& points) {

        int n = points.size();
        vector<vector<pair<int, int>>> graph(n);

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (i == j) continue;
                graph[i].push_back({abs(points[i][0] - points[j][0]) + abs(points[i][1] - points[j][1]), j});
            }
        }

        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
        vector<bool> vis(n, false);
        pq.push({0, 0}); // {cost, Node}

        int ans = 0;
        while (!pq.empty())
        {
            pair<int,int> p = pq.top();
            int currNode = p.second;
            int currCost = p.first;
            pq.pop();

            if (vis[currNode]) continue;
            ans += currCost;
            vis[currNode] = true;

            for(int i=0;i<graph[currNode].size();i++)
            {
                int neighbourNode = graph[currNode][i].second;
                int neighbourNodeCost = graph[currNode][i].first;
                if(vis[neighbourNode])  continue;
                pq.push({neighbourNodeCost, neighbourNode});
            }

        }
        return ans;
    }
};

```

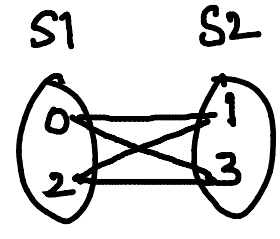
(18) Is Graph Bipartite

Bipartite graph is undirected graph, such that all vertices can be divided into 2 sets, S1 & S2 and no two vertices present in same set share an edge.

Eg $n = 4$



then



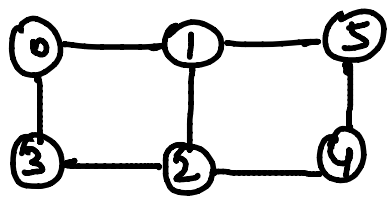
∴ the graph is bipartite.

⇒ for graph to be bipartite,

- it needs to be undirected acyclic graph (or)
- it needs to be even length cyclic graph

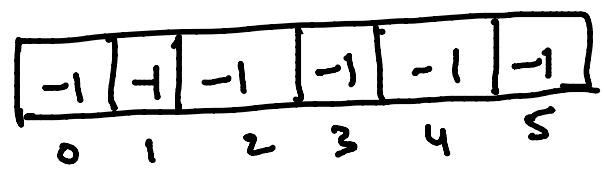
→ we generally denote sets by coloring it, color = 0, 1.
↓ ↓
S1 S2

Eg $n = 6$



vis = { } S1 = { } S2 = { }

initially color



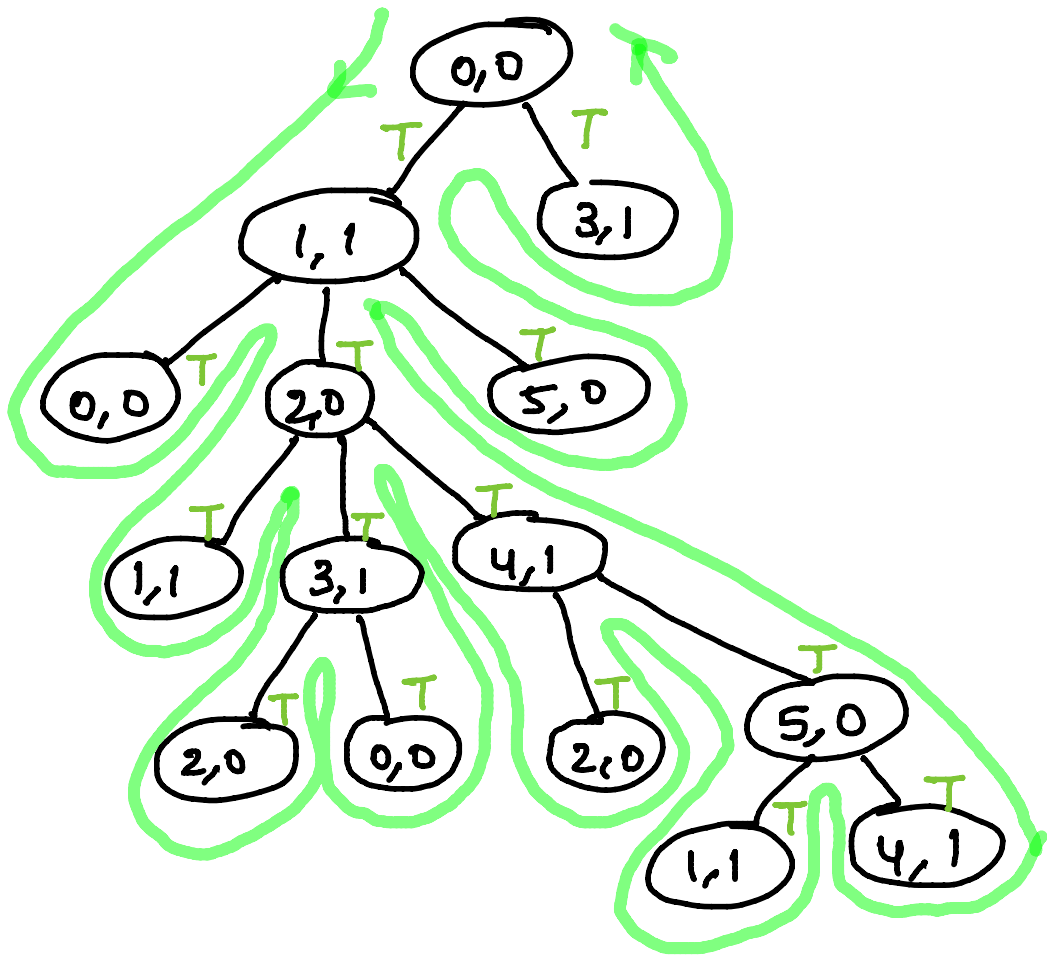
→ at each vertex, check if its visited or not.

→ if visited then check if its present in the intended set or not.

→ if yes then return True, else false

→ return AND of all the boolean values.

⇒



Code →

```cpp
class Solution {
public:

    bool isBipartite(vector<vector<int>>& graph) {

        int n= graph.size();
        vector<int>colors(n,-1);

        for(int curr=0; curr<n ; curr++){
            // if already colored then skip
            if(colors[curr]!=-1)    continue;
            // check for even length cycle
            if(hasEvenLengthCycle(graph, curr, 0, colors)==false)  return false;
        }
        return true;
    }

    bool hasEvenLengthCycle(vector<vector<int>>& graph,int curr,int color,vector<int>&colors)
    {
        if(colors[curr]!=-1)
            return colors[curr]==color;

        // if not colored then color it
        colors[curr] = color;

        // check for neighbours
        for(int neigh: graph[curr])
        {
            if(hasEvenLengthCycle(graph, neigh, 1-color, colors)==false)
                // 1- color will handle both changing colors 0 to 1 and 1 to 0
                return false;
        }
        return true;
    }

};
```

(19) Possible Bipartition →

→ create a graph using dislikes array.

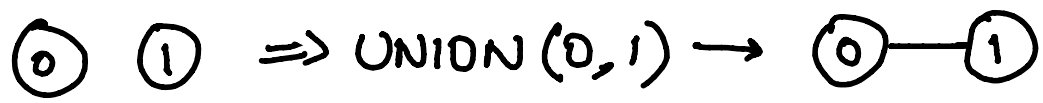
→ use previous problem's approach to solve it.

Code →

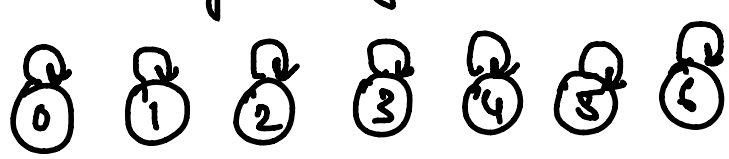
TC → O(V+E) SC → O(V+E)

```cpp
class Solution {
public:

    bool dfs(vector<int> graph[], int curr, vector<int>& color){

        // if not colored then color
        if(color[curr] == -1)
            color[curr] = 1;

        // process the neighbours and check their colors
        for(auto neigh : graph[curr])
        {
            if(color[neigh] == -1)
            {
                color[neigh] = 1 - color[curr];
                if(dfs(graph, neigh, color)==false) return false;
            }
            else if(color[neigh] == color[curr]) return false;
        }
        return true;
    }

    bool possibleBipartition(int n, vector<vector<int>>& dislikes) {
        vector<int> color(n+1, -1);
        vector<int> graph[n+1];

        // populating the graph
        for(auto edge : dislikes){
            graph[edge[0]].push_back(edge[1]);
            graph[edge[1]].push_back(edge[0]);
        }

        for(int i=1; i<=n; i++){
            if(color[i] == -1)
                if(!dfs(graph, i, color)) return false;
        }

        return true;
    }
};
```

(20) Disjoint set → UNION & FIND./getParent
↳ helps in finding parent of component
↳ helps in UNION of component/vertices.

Eg  ⇒ UNION(0,1) →

Eg n = 7 initially every component is parent of itself



parent =

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

now getParent(2) = 2, getParent(3) = 3.

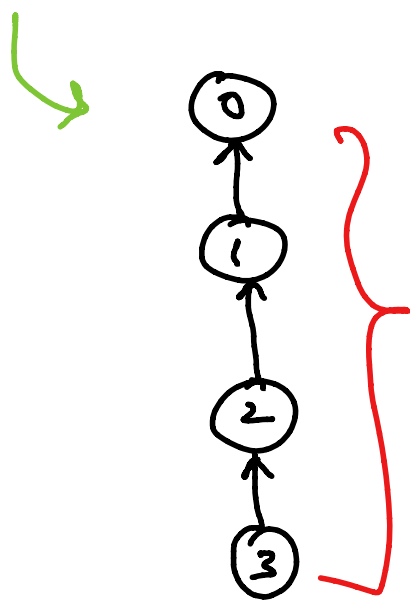
& if UNION(0,1) ⇒  & parent[1] = 0

now getParent(1) = 0

& UNION(1,2) ⇒ 

UNION(2,3) ⇒ 

& getParent(3) = 0



This increases the recursive calls and the tree is unbalanced so we'll use rank array to store min. height tree for node.

n = 7 initially every component is parent of itself

parent =

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

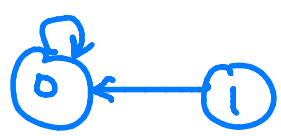
rank =

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

⟹ UNION (0,1) ⇒ then find(0) & find(1) & 0 != 1 ∴ diff components.

as they are diff components find rank & rank[0] = rank[1] = 0

∴ select either 0 or 1 & make it as root & inc the rank by 1

⟹ 

parent =

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

rank =

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

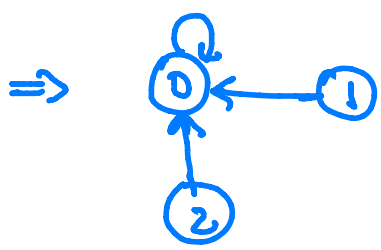
⟹ UNION (1,2) ⇒ parent(1) = 0 & parent(2) = 2

now rank[0] = 1 & rank[2] = 0

as rank[0] > rank[2] ,

vertex 0 should be the parent

& donot update rank if they are unequal.

⟹ 

parent =

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

rank =

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

Code →

```cpp
class DisjSet {
    int *rank, *parent, n;

    public:
    DisjSet(int n)
    {
        rank = new int[n];
        parent = new int[n];
        this->n = n;
        makeSet();
    }

    void makeSet()
    {
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    int find(int x)
    {
        // if x is not parent of itself then
        // find parent recursively
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    void Union(int x, int y)
    {
        int xset = find(x);
        int yset = find(y);

        // if set of x and y are same then return
        if (xset == yset)   return;

        // place the elements in small rank
        if (rank[xset] < rank[yset]) {
            parent[xset] = yset;
        }
        else if (rank[xset] > rank[yset]) {
            parent[yset] = xset;
        }
        // if same rank then increment it
        else {
            parent[yset] = xset;
            rank[xset] = rank[xset] + 1;
        }
    }
};

```

(21) Kruskal's Algorithm →

→ This is used to find minimum spanning tree.

→ can be implemented using Disjoint set.

→ sort all the edges in ↑ order of weight.

→ pick smallest edge & check if it contributes to cycle in graph

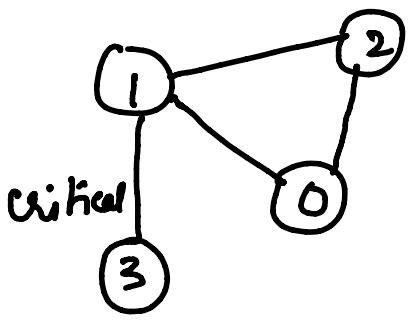
→ if yes then discard else include.

code →

```cpp
class Graph {
    vector<vector<int> > edgelist;
    int V;

public:
    Graph(int V) { this->V = V; }

    void addEdge(int x, int y, int w)
    {
        edgelist.push_back({ w, x, y });
    }

    void kruskals_mst()
    {
        // 1. Sort all edges
        sort(edgelist.begin(), edgelist.end());

        // Initialize the DSU - DisjointSet
        DSU s(V);
        int ans = 0;
        for (auto edge : edgelist) {
            int w = edge[0];
            int x = edge[1];
            int y = edge[2];
            // take that edge in MST if it does form a cycle
            if (s.find(x) != s.find(y)) {
                s.union(x, y);
                ans += w;
                cout << x << " -- " << y << " == " << w
                     << endl;
            }
        }
        cout << "Minimum Cost Spanning Tree: " << ans;
    }
};
```

(22) Critical Connection in a Network →

Eg: n = 4 edges = [[0,1], [1,2], [2,0], [1,3]]



→ Critical connection is a connection, when removed from graph, would result in breaking graph into different components.

Here if [1,3] is removed then graph becomes disconnected.

Approach 1

→ Remove one edge each time

→ Perform dfs

→ If all vertices are not visited then

→ Removed edge is a critical connection.

Approach 2

(discovery time for vertex) (min time for vertex to be discovered.)

→ initialise distime array & mintime array with -1.

→ perform dfs from one node

→ if neighbour == parent then continue

→ else if neighbour is already visited then

mintime[curr] = min(mintime[curr], distime[neigh])

→ while returning mintime[curr] = min(mintime[curr], mintime[neigh])

& at any point if distime[curr] < mintime[neigh]

This indicates critical connection

code →

```cpp
class Solution {
public:

    vector<vector<int>> criticalConnections(int n, vector<vector<int>>& connections) {
        vector<int> graph[n];
        for(vector<int> edge: connections){
            int u = edge[0];
            int v = edge[1];
            graph[u].push_back(v);
            graph[v].push_back(u);
        }
        return findCriticalConnections(n, graph);
    }

    vector<vector<int>> findCriticalConnections(int n, vector<int> graph[]){
        vector<int> disTime(n,-1);
        vector<int> lowTime(n,-1);
        int time = 0;
        vector<vector<int>> answer;
        tarjansDFS(graph, 0, -1, disTime, lowTime, time, answer);
        return answer;
    }

    void tarjansDFS(vector<int> graph[], int curr, int parent, vector<int>&disTime,
    vector<int> &lowTime, int &time, vector<vector<int>> &answer){

        disTime[curr] = time;
        lowTime[curr] = time;
        time += 1;

        for(int neigh: graph[curr]){
            if(neigh == parent) continue;

            if(disTime[neigh]!=-1){
                lowTime[curr] = min(lowTime[curr], disTime[neigh]);
                continue;
            }

            tarjansDFS(graph, neigh, curr, disTime, lowTime, time, answer);
            lowTime[curr] = min(lowTime[curr], lowTime[neigh]);

            if(disTime[curr] < lowTime[neigh]){
                vector<int> temp;
                temp.push_back(curr);
                temp.push_back(neigh);
                answer.push_back(temp);
            }
        }
        return;
    }

};
```

Find the rest on

https://linktr.ee/KarunKarthik

Follow Karun Karthik For More Amazing Content !

Dynamic Programming - 1

- Karun Karthik

Contents

Introduction

Dynamic programming is a technique to solve problems by breaking it down into a colle ction of sub-problems, solving each of those sub-problems just once and storing these solutions inside the cache memory in case the same problem occurs the next time.

Dynamic Programming is mainly an optimization over plain recursion .
Wherever we see a recursive solution that has repeated calls for same inputs, we can o ptimize it using Dynamic Programming.
This simple optimization reduces the time complexities from exponential to polynomial.

There are two different ways to store our values so that they can be reused at a later i nstance. They are as follows:
1. Memoization or the Top Down Approach.
2. Tabulation or the Bottom Up approach.

In Memoization we start from the extreme state and compute result by using values th at can reach the destination state i.e the base state.

In Tabulation we start from the base state and then compute results all the way till the extreme state.

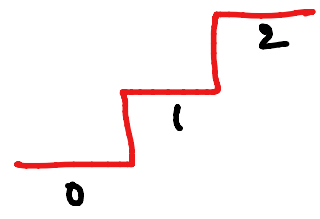
Note: To store the intermediate results we can use Array, Matrix, Hashmap etc., all we need is data storage and retrieval with a specific key.

How to find the use case of Dynamic Programming?

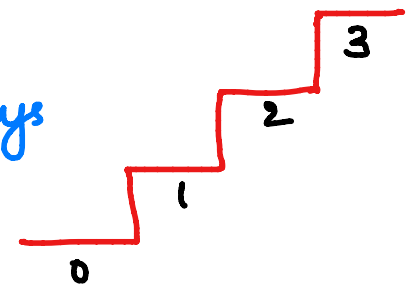
You can use DP if the problem can be,
1. Divided into sub-problems
2. Solved using a recursive solution
3. Containing repetitive sub-problems

① Climbing Stairs → Given a value 'N', find the number of ways to reach N & jumps possible are ONE or TWO.

Eg n = 2 ⇒ $0 \xrightarrow{1} 1 \xrightarrow{1} 2$; $0 \xrightarrow{2} 2$ } for N = 2 we have 2 ways



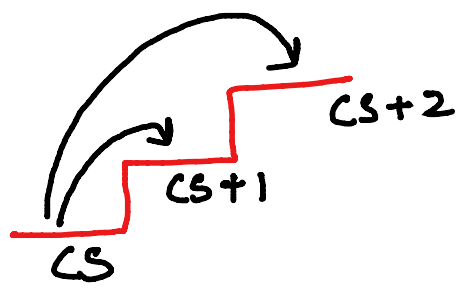
n = 3 ⇒ $0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{1} 3$; $0 \xrightarrow{1} 1 \xrightarrow{2} 3$; $0 \xrightarrow{2} 2 \xrightarrow{1} 3$ } for N = 3 we have 3 ways

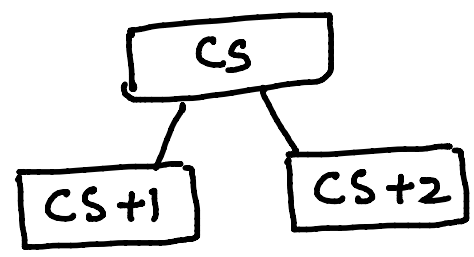


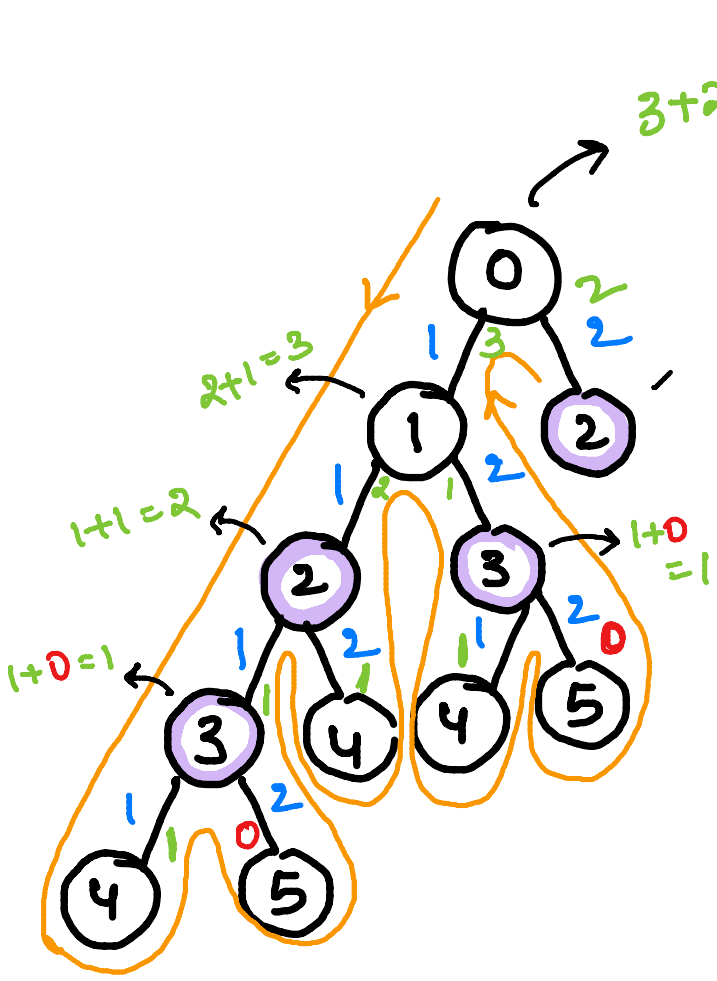
n = 4

↳ for every stair we have 2 cases ie

⇒







5 ways

$0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{1} 3 \xrightarrow{1} 4$

$0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{2} 4$

$0 \xrightarrow{1} 1 \xrightarrow{2} 3 \xrightarrow{1} 4$

$0 \xrightarrow{2} 2 \xrightarrow{1} 3 \xrightarrow{1} 4$

$0 \xrightarrow{2} 2 \xrightarrow{2} 4$

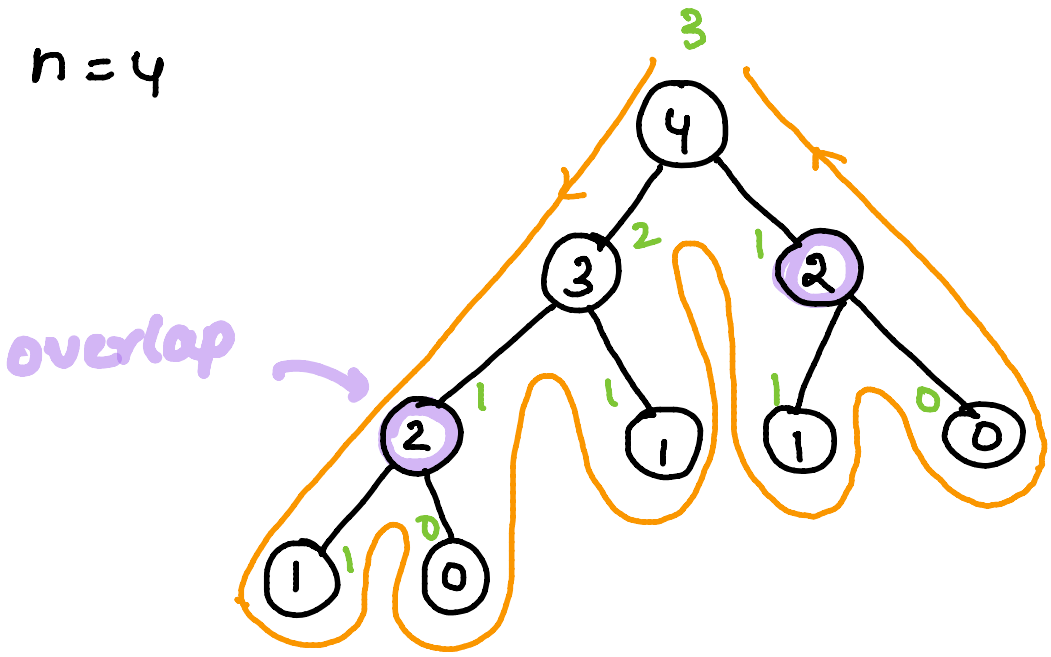
★ Here we can see for ②, ③ the subproblem is being done multiple times, we can solve using dp.

Code →

```cpp
class Solution {
public:
    int totalWays(int currentStair, int targetStair, unordered_map<int,int> &memo){

        if(currentStair==targetStair){
            return 1;
        }

        if(currentStair > targetStair){
            return 0;
        }

        int currentKey = currentStair;

        if(memo.find(currentKey)!=memo.end()){
            return memo[currentKey];
        }

        int oneStep = totalWays(currentStair+1, targetStair, memo);
        int twoStep = totalWays(currentStair+2, targetStair, memo);

        memo[currentKey] = oneStep+twoStep;

        return oneStep+twoStep;

    }

    int climbStairs(int n) {
        unordered_map<int,int> memo;
        return totalWays(0,n,memo);
    }
};
```

② Fibonacci Number → $f(n) = f(n-1) + f(n-2)$ & $f(0) = 0$, $f(1) = 1$, $n \geq 0$.

Eg → $n = 4$



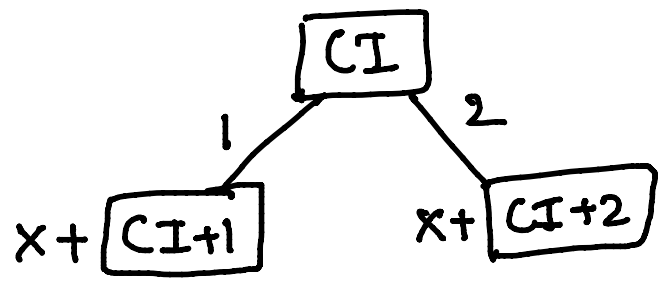
Code →

```cpp
class Solution {
public:
    int helper(int n, unordered_map<int,int>&memo){

        if(n<=1){
            return n;
        }

        int currentKey = n;

        if(memo.find(currentKey)!=memo.end()){
            return memo[currentKey];
        }


        int a = helper(n-1,memo);
        int b = helper(n-2,memo);

        memo[currentKey] = a+b;
        return memo[currentKey];
    }


    int fib(int n) {
        unordered_map<int,int>memo;
        return helper(n,memo);
    }
};
```

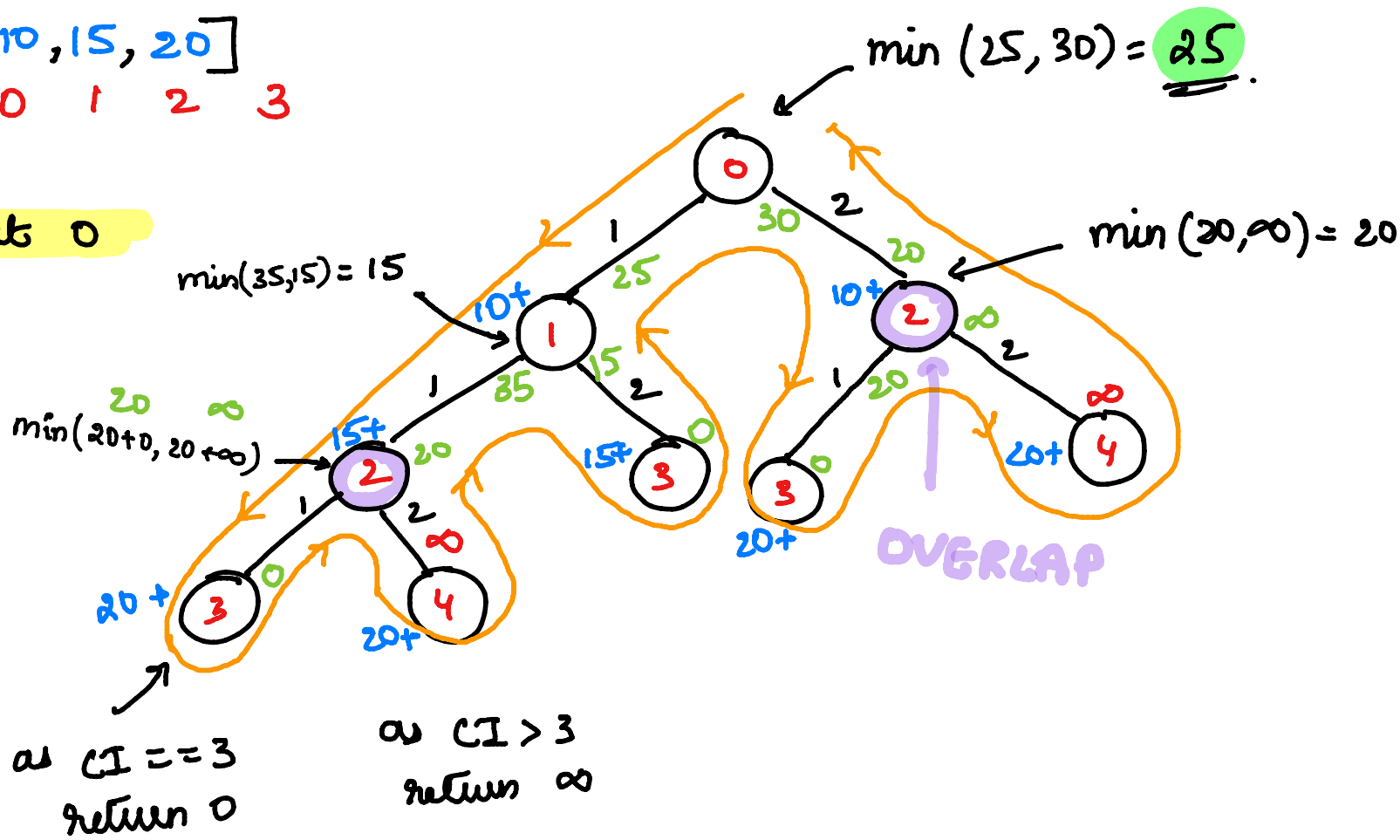
③ Min Cost Climbing Stairs →

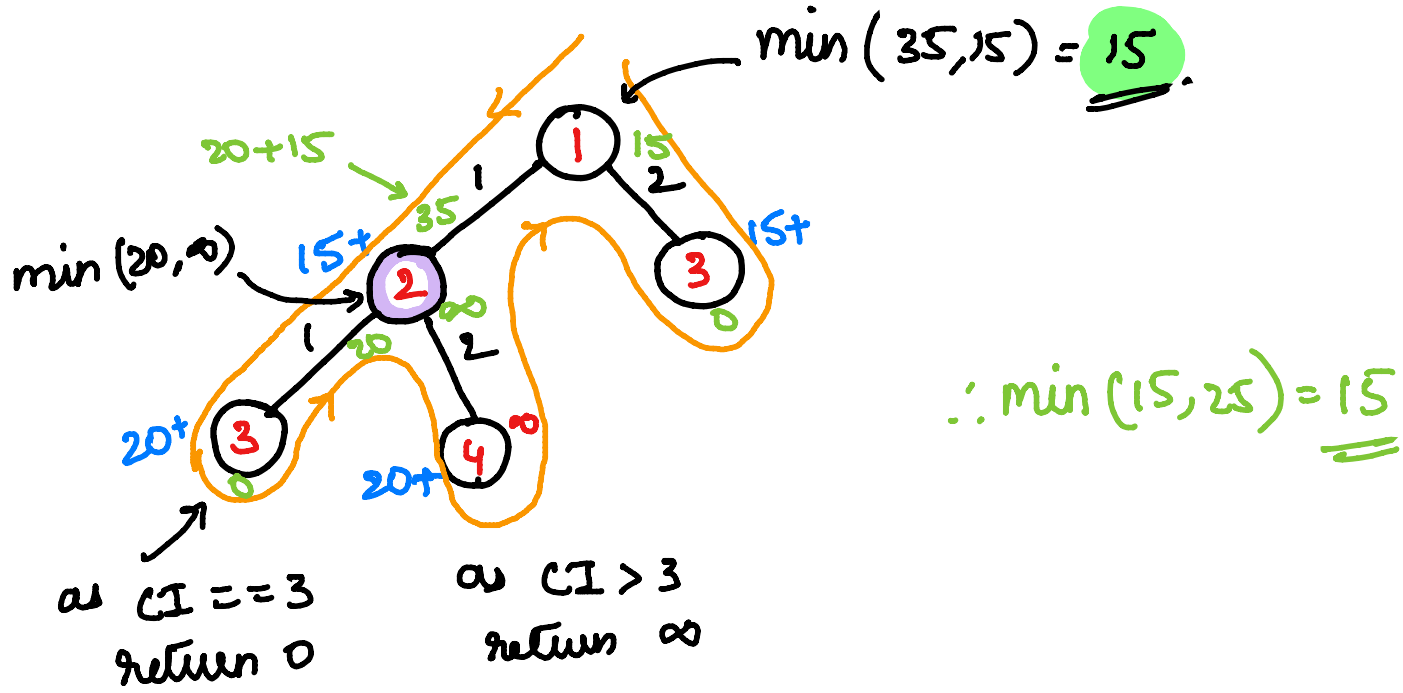
Given costs array, find min cost to reach the end, starting from 0 or 1 & making 1 or 2 jumps.

$\therefore$ costs = [_ _ X _ _ _]
CI



Eg cost = [10, 15, 20]
0 1 2 3





$\therefore$ min (15, 25) = 15

Code →

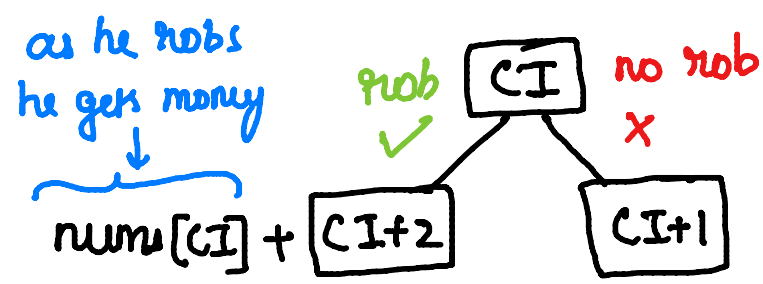
```cpp
class Solution {
public:

    int minCost(vector<int>&cost, int currentIndex, unordered_map<int,int> &m){

        if(currentIndex == cost.size()){
            return 0;
        }

        if(currentIndex > cost.size()){
            return 1000;    // large values, serves as INFINITY
        }

        if(m.find(currentIndex)!=m.end()){
            return m[currentIndex];
        }

        int oneJump = cost[currentIndex] + minCost(cost,currentIndex+1, m);
        int twoJump = cost[currentIndex] + minCost(cost,currentIndex+2, m);

        m[currentIndex] = min(oneJump, twoJump);
        return m[currentIndex];
    }

    int minCostClimbingStairs(vector<int>& cost) {
        unordered_map<int,int> m;
        return min( minCost(cost,0,m), minCost(cost,1,m));
    }
};
```

④ House Robber → Given an array of no. representing money, find max amount, that can be robbed without choosing the adjacent houses.

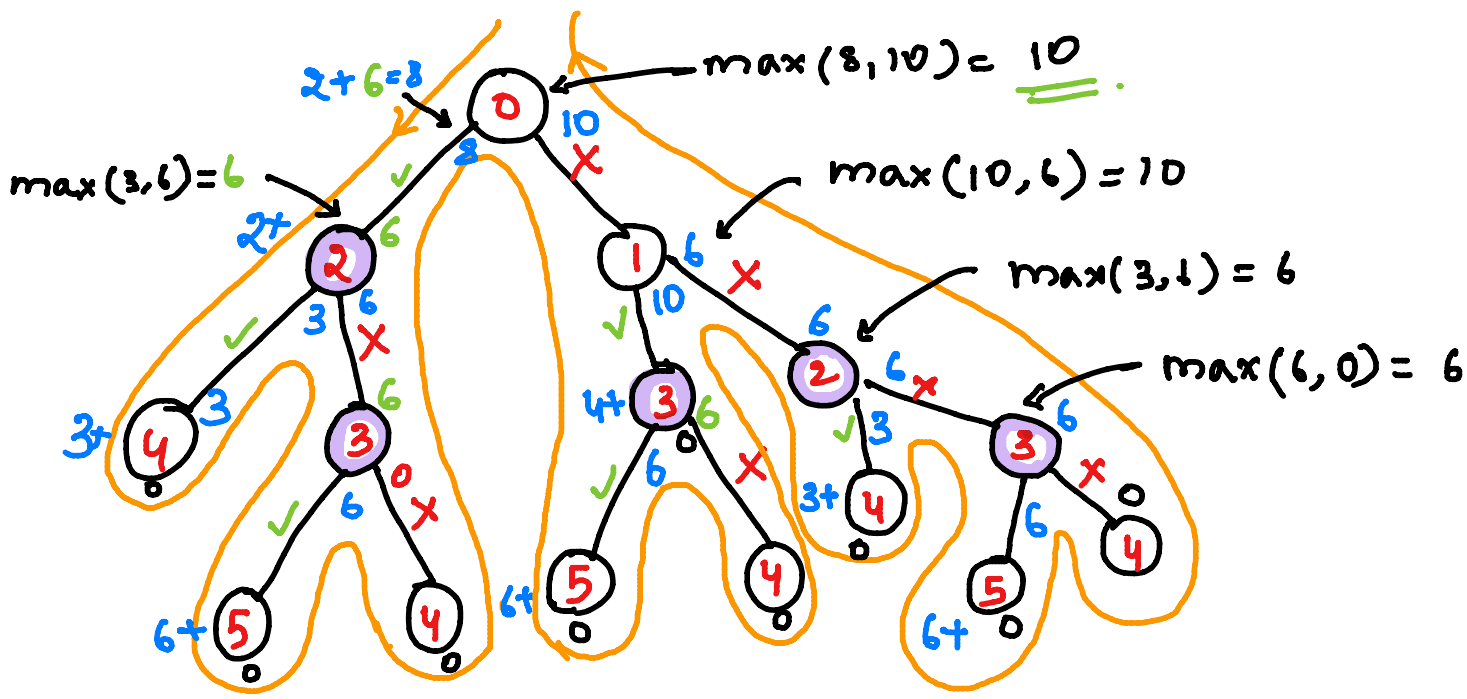
Eg nums = [2, 7, 9, 3, 1] ⇒ max amount = 2 + 9 + 1 = 12.
0 1 2 3 4 (0 → 2 → 4)

→ If robber robs house, then amount = nums[CI] & CI = CI + 2 (to avoid adjacent)
else CI = CI + 1

ie nums = [_ _ CI _ _ _ _]



Eg [2, 4, 3, 6]
0 1 2 3



as CI > 3
return 0

∴ at every node find max(left, right) & add it's value to the nums[CI] if selected, else continue.

code →

```cpp
class Solution {
public:

    int helper(vector<int>&nums, int currentIndex, unordered_map<int,int>&m){

        if(currentIndex >= nums.size()){
            return 0;
        }

        int currentKey = currentIndex;

        if(m.find(currentKey)!=m.end()){
            return m[currentKey];
        }

        int rob = nums[currentKey] + helper(nums, currentIndex+2, m);
        int noRob = helper(nums, currentIndex+1, m);

        m[currentIndex] = max(rob, noRob);

        return m[currentIndex];
    }

    int rob(vector<int>& nums) {
        unordered_map<int,int> m;
        return helper(nums,0,m);
    }
};
```

⑤ House Robber - II →

In this problem, the approach will be similar to previous one, but the houses are in circle, which means that

* if we start from 1st house, then we can't rob the last house.
* if we start from 2nd house, then we can rob the last house.
* and return max value between 1st house & 2nd house
* if only 1 house is present, then rob it directly.

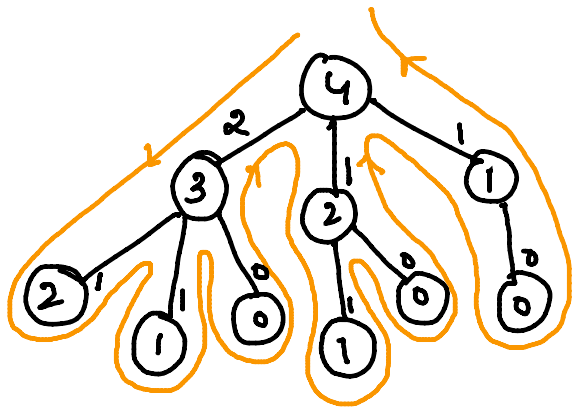
Code →

```cpp
class Solution {
public:

    int helper(vector<int>&nums, int currentIndex, int lastIndex, unordered_map<int,int>&m){

        if(currentIndex > lastIndex){
            return 0;
        }

        int currentKey = currentIndex;

        if(m.find(currentKey)!=m.end()){
            return m[currentKey];
        }

        int rob = nums[currentKey] + helper(nums, currentIndex+2, lastIndex, m);
        int noRob = helper(nums, currentIndex+1, lastIndex, m);

        m[currentIndex] = max(rob, noRob);

        return m[currentIndex];

    }

    int rob(vector<int>& nums) {

        int n = nums.size();
        if(n==1)    return nums[0];

        unordered_map<int,int> memo1,memo2;
        // we can start robbing from 2 houses
        int firstHouse = helper(nums, 0, n-2, memo1);
        int secondHouse = helper(nums, 1, n-1, memo2);
        return max(firstHouse, secondHouse);
    }
};
```

(6) N-th Tribonacci → given n, find T_n

$T_{n+3} = T_n + T_{n+1} + T_{n+2}$ ∀ $n \geq 0$ $T_0 = 0$, $T_1 = 1$, $T_2 = 1$.

Eg $n = 4$



4 (2+1+1)

code →

```cpp
class Solution {
public:

    int helper(int n, unordered_map<int,int> &m){
        if(n<=1){
            return n;
        }

        if(n==2){
            return 1;
        }

        int currentNum = n;

        if(m.find(currentNum)!=m.end()){
            return m[currentNum];
        }

        int a = helper(n-1,m);
        int b = helper(n-2,m);
        int c = helper(n-3,m);

        m[currentNum] = a+b+c;

        return m[currentNum];
    }

    int tribonacci(int n) {
        unordered_map<int,int>m;
        return helper(n,m);
    }
};
```

(7) 0 - 1 Knapsack Problem → find max profit such that the weight of all items <= capacity.

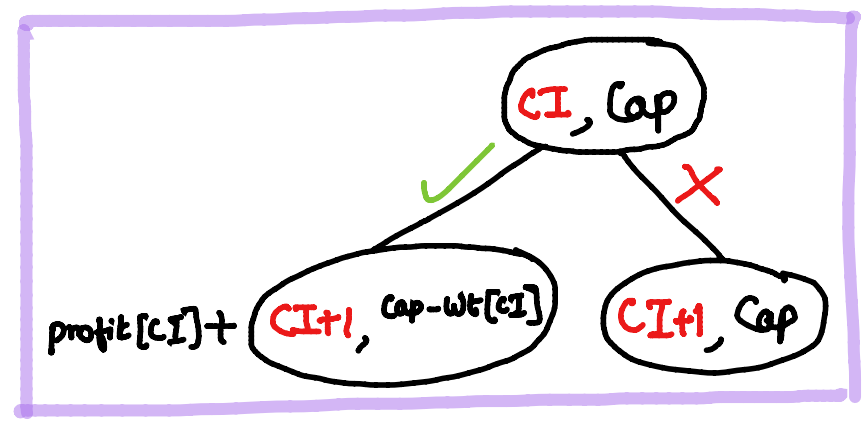
```
          0  1  2  3
wt      = [3, 4, 6, 5]
profits = [2, 3, 1, 4]
capacity = 8
```

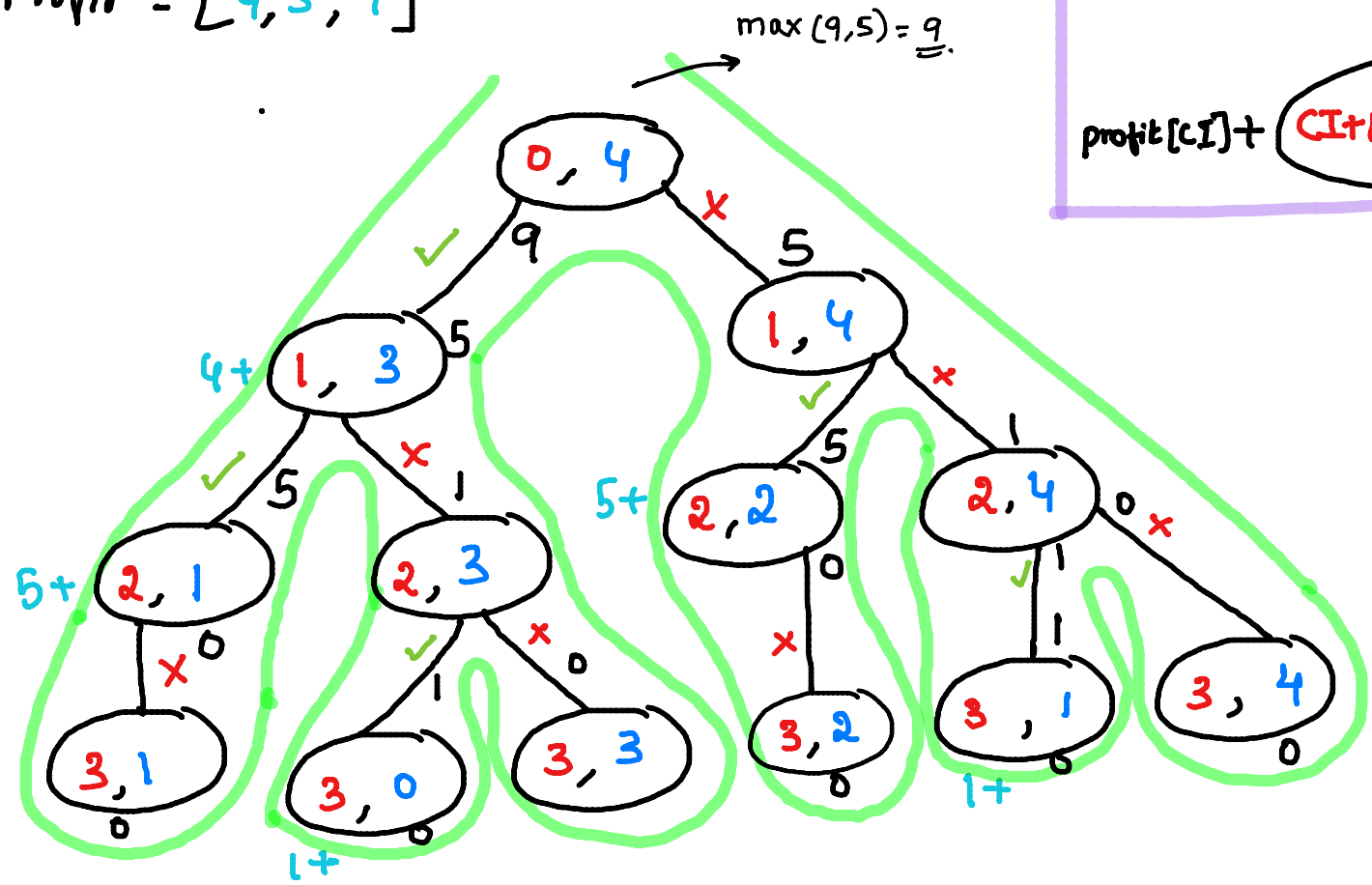
→ if we select 0 & 3,
then total weights = wt[0] + wt[3]
= 3 + 5 = 8.

& profits are 2 + 4 = 6 That's max profit possible

Eg

```
          0  1  2
Wt     = [1, 2, 3]      capacity = 4
Profit = [4, 5, 1]
```





∴ at every step,

→ if selecting an index then reduce capacity by wt[CI]
& add profit[CI] to result

→ if not selecting, increment CI by 1

→ find max(left, right)

code →

```cpp
class Solution
{
    public:

    int helper(int W, int wt[], int val[], int n, int curr,
                                unordered_map<string,int> &memo){
        if(curr==n) return 0;

        // Instead of Matrix we can use strings as unique keys
        string currKey = to_string(curr)+"_"+to_string(W);

        if(memo.find(currKey)!=memo.end()) return memo[currKey];

        int currWt = wt[curr];
        int currVal = val[curr];

        int selected = 0;
        if(currWt<=W){
            selected = currVal + helper(W-currWt, wt, val, n, curr+1, memo);
        }

        int notSelected = helper(W, wt, val, n, curr+1, memo);

        memo[currKey] = max(selected, notSelected);
        return memo[currKey];
    }


    int knapSack(int W, int wt[], int val[], int n)
    {
       unordered_map<string,int> memo;
       return helper(W, wt, val, n, 0, memo);
    }
};
```

(8) Partition Equal Subset Sum →

given an array, find if it can be divided into two subsets whose sum is equal.

Eg nums = [1, 5, 11, 5] can be divided into S1 = {1, 5, 5} & S2 = {11}
& sum of S1 == sum of S2 ∴ returns True.

∴ initially find sum of elements in array.

1) if sum is odd then return False

2) if sum is even, then proceed.

→ find a subset whose value == sum/2
which means that the other subset will have value == sum/2.

→ lets say ts = sum/2 (ts is target sum)

→ At every index, we have 2 choices

1) if we select then ts = ts − nums[CI] (CI: current index)
&
CI = CI + 1

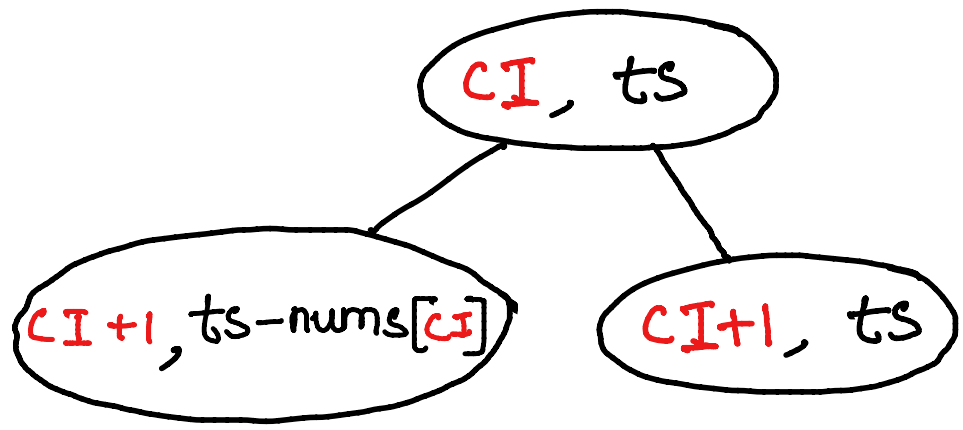
2) if we donot select then ts = ts (i.e remains same)
CI = CI + 1

3) return OR of left & right branch.

⇒ nums = [1, 5, 11, 5] (0, 1, 2, 3)

sum = 22

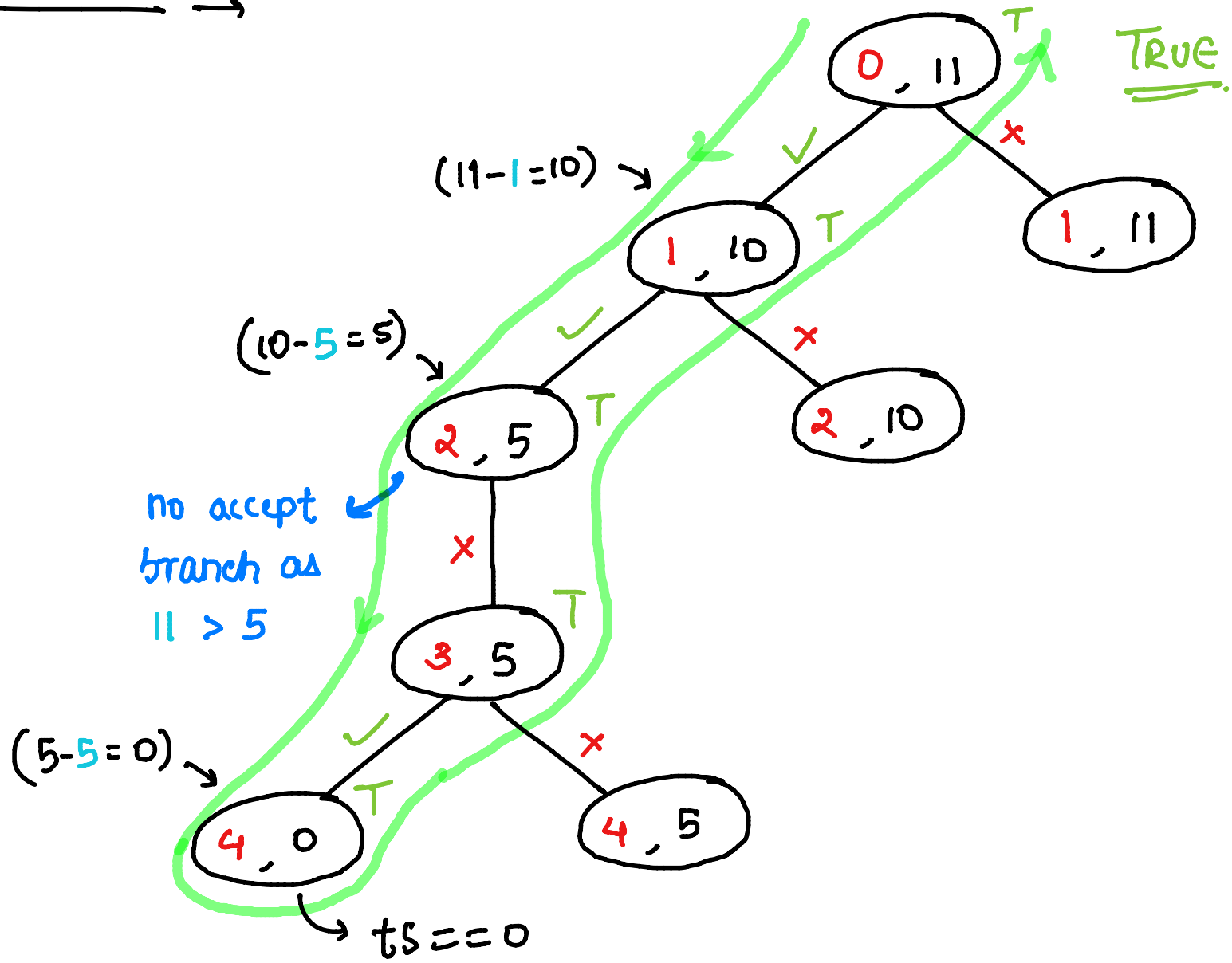
ts = 11



Here sum % 2 == 0

∴ dividing into 2 subsets is possible.

Explanation →



⇒ that subset is found

& return True

as we are using OR, one True branch is sufficient

Code →

```cpp
class Solution {
public:

    bool isPossible(int targetSum,int currentIndex, vector<int>&nums,
                                    unordered_map<string, bool> &memo){

        if(targetSum == 0)
            return true;

        if(currentIndex >= nums.size())
            return false;

        string currentKey = to_string(currentIndex)+"_"+to_string(targetSum);

        if(memo.find(currentKey)!=memo.end()){
            return memo[currentKey];
        }

        bool possible = false;

        if(nums[currentIndex]<=targetSum)
            possible = isPossible(targetSum-nums[currentIndex], currentIndex+1, nums, memo);

        // if already Possible then return True directly
        if(possible){
            memo[currentKey] = possible;
            return true;
        }

        bool notPossible = isPossible(targetSum, currentIndex+1, nums, memo);

        memo[currentKey] = possible||notPossible;
        return memo[currentKey];
    }

    bool canPartition(vector<int>& nums) {

        int total = 0;
        for(auto it:nums) total+= it;

        if(total%2!=0)  return false;

        unordered_map<string, bool> memo;
        return isPossible(total/2,0, nums,memo);
    }
};
```

(9) Target Sum →

given an array & target, find the number of ways to reach target by using + or - before each element in array.

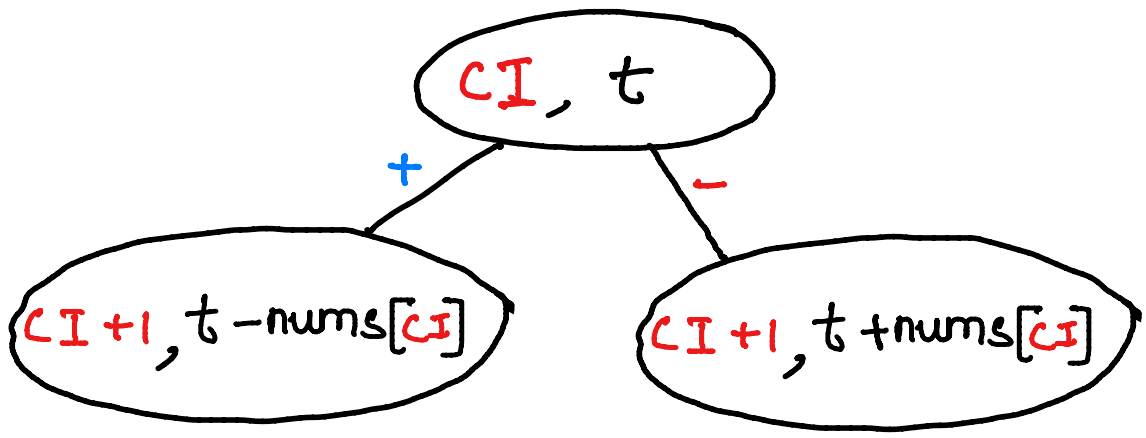
Eg

Input: nums = [1,1,1,1,1], target = 3
Output: 5
Explanation: There are 5 ways to assign symbols to make the sum of nums be target 3.
-1 + 1 + 1 + 1 + 1 = 3
+1 - 1 + 1 + 1 + 1 = 3
+1 + 1 - 1 + 1 + 1 = 3
+1 + 1 + 1 - 1 + 1 = 3
+1 + 1 + 1 + 1 - 1 = 3

→ at every index we can use + or - sign

if + then $t = t - (+nums[CI]) \Rightarrow t - nums[CI]$

if - then $t = t - (-nums[CI]) \Rightarrow t + nums[CI]$



→ at every node, return the sum of values from left & right. Because we need to find the total number of ways.

code →

```cpp
class Solution {
public:
    int totalWays(int currentIndex, vector<int>&nums, int target, unordered_map<string,int> &memo){

        if(target==0 and currentIndex==nums.size()){
            return 1;
        }

        if(currentIndex>=nums.size() and target!=0){
            return 0;
        }

        string key = to_string(currentIndex)+"_"+to_string(target);

        if(memo.find(key)!=memo.end()){
            return memo[key];
        }

        int plus = totalWays(currentIndex+1, nums, target-nums[currentIndex],memo);

        int minus = totalWays(currentIndex+1, nums, target+nums[currentIndex],memo);

        memo[key] = plus+minus;

        return plus+minus;
    }

    int findTargetSumWays(vector<int>& nums, int target) {
        unordered_map<string,int> memo;
        return totalWays(0,nums,target,memo);
    }
};
```

(10) Count number of subsets with given difference →

→ This is similar to Target Sum.

given the difference between two subsets, and an array find no. of subsets with the difference.

Approach →

Let say $S1 - S2 =$ difference (given) — (1)

we can calculate sum of every element, say sum

& it can be said that for 2 subsets S1 & S2

$S1 + S2 =$ Sum. — (2)

Now (1) + (2) $\Rightarrow$ $2(S1) =$ difference + sum

$S1 =$ (difference + sum)/2.

→ Implement Target sum with target value = S1.

(11) Delete and Earn →

You are given an integer array `nums`. You want to maximize the number of points you get by performing the following operation any number of times:

- Pick any `nums[i]` and delete it to earn `nums[i]` points. Afterwards, you must delete **every** element equal to `nums[i] - 1` and **every** element equal to `nums[i] + 1`.

Return *the **maximum number of points** you can earn by applying the above operation some number of times.*

Eg nums = [2, 2, 3, 3, 3, 4]

→ if we start deleting 2, then result = 2+2 = 4

then nums = [3, 3, 3, 4]

& we need to delete all 2+1 & 2-1 ⇒ nums = [4]

→ if we delete 4, then result = 4+4 = 8.

& nums = [ ]

(or)

→ if we start deleting 3, then result = 3+3+3 = 9.

then nums = [2, 2, 4]

& we need to delete all 3-1 & 3+1 ⇒ nums = [ ]

∴ result = 9.

(or)

→ if we start deleting 4, then result = 4

then nums = [2, 2, 3, 3, 3]

& we need to delete all 4+1 & 4-1 ⇒ nums = [2, 2]

→ if we delete 2, then result = 4+4 = 8.

& nums = [ ]

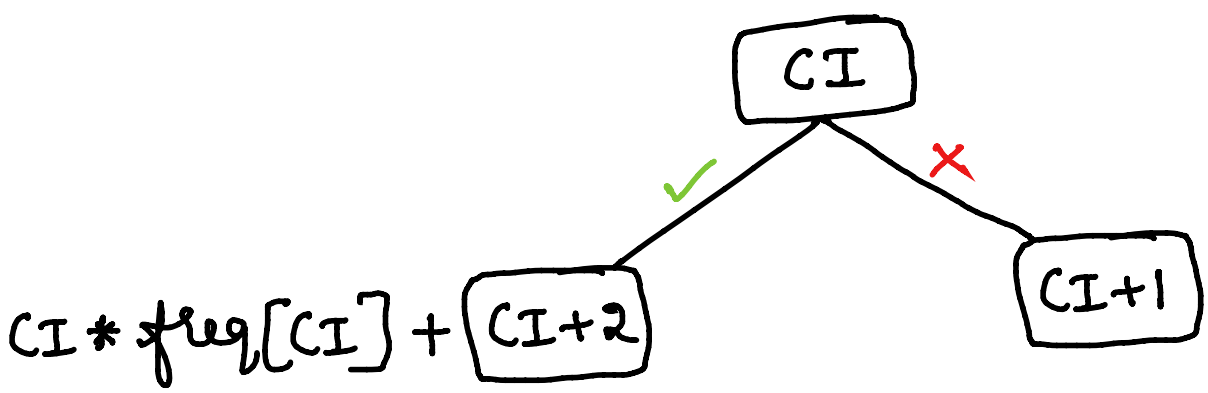
→ we can store frequency of each element and use the similar approach

nums = [2, 2, 3, 3, 3, 4]

freq =

0	1	2	3	4
0	0	2	3	1

↑
CI



Code →

```cpp
class Solution {
public:

    int maxPoints(vector<int>& freq, int currentIndex, unordered_map<int,int>&memo){

        if(currentIndex >= freq.size()) return 0;

        int key = currentIndex;

        if(memo.find(key) != memo.end()) return memo[key];

        int Delete = currentIndex*freq[currentIndex] + maxPoints(freq, currentIndex+2, memo);
        int NotDelete = maxPoints(freq, currentIndex+1, memo);

        memo[key] = max(Delete, NotDelete);

        return memo[key];
    }

    int deleteAndEarn(vector<int>& nums) {

        int maxi = *max_element(nums.begin(), nums.end());
        vector<int> freq(maxi+1, 0);

        for(auto i: nums)   freq[i]++;

        unordered_map<int,int> memo;

        return maxPoints(freq, 0, memo);
    }
};
```

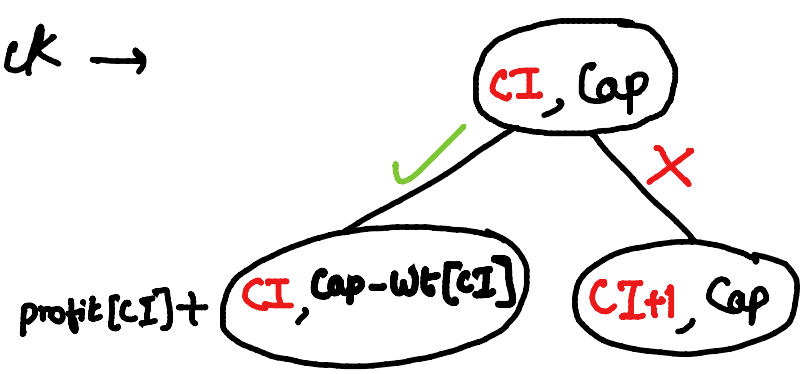
(12) Unbounded Knapsack → Similar to 0-1 knapsack but allows us to choose an item more than once

Eg
wt = [2, 1] (0, 1)
values = [1, 1]
capacity = 3

if bounded knapsack then profit = 2 (2, 1) (1+1)

if unbounded knapsack then profit = 3 (1, 1, 1) (1+1+1)

for unbounded knapsack →



Code →

```cpp
class Solution{
public:
    int helper(int W, int wt[], int val[], int N, int curr, vector<vector<int>>&memo){

        if(W==0)    return 0;
        if(curr==N) return 0;

        if(memo[curr][W]!=-1)   return memo[curr][W];

        int currWt = wt[curr];
        int currVal = val[curr];

        int selected = 0;
        if(currWt<=W){
            selected = currVal + helper(W-currWt, wt, val, N, curr, memo);
        }

        int notSelected = helper(W, wt, val, N, curr+1, memo);

        memo[curr][W] = max(selected, notSelected);
        return memo[curr][W];
    }

    int knapSack(int N, int W, int val[], int wt[])
    {
        vector<vector<int>> memo( N , vector<int> (W+1, -1));
        return helper(W, wt, val, N, 0, memo);
    }
};
```

(13) Coin Change II → (Similar to unbounded knapsack)

given an array of coins & amount, find total number of ways / combinations to make up that amount.

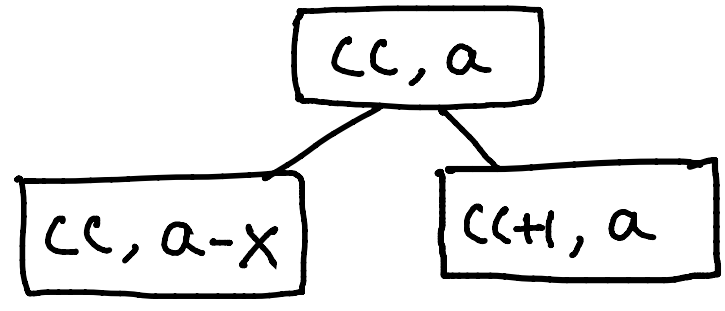
Eg coins = [1, 2, 5]

amount = 5

4 ways
- 1+1+1+1+1
- 1+1+1+2
- 1+2+2
- 5

coins = [_ x _ _ _] x = coins[cc]

current coin: cc, a → amount



code →

```cpp
class Solution {
public:

    int totalWays(int currentIndex, vector<int>& coins, int amount,vector<vector<int>>&memo){
        if(amount == 0) return 1;   // amount==0 means that target is reached so return 1
        if(currentIndex >= coins.size()) return 0;  //if index is out of bounds then return 0

        if(memo[currentIndex][amount]!=-1) return memo[currentIndex][amount];

        int consider = 0;
        if(coins[currentIndex]<=amount){
            consider = totalWays(currentIndex,coins,amount-coins[currentIndex],memo);
        }
        int notConsider = totalWays(currentIndex+1,coins,amount,memo);

        memo[currentIndex][amount] = consider+notConsider;
        return memo[currentIndex][amount];
    }

    int change(int amount,vector<int>& coins) {
        vector<vector<int>>memo(coins.size()+1,vector<int>(amount+1,-1));
        return totalWays(0,coins,amount,memo);
    }
};
```

(14) Coin Change → (Similar to unbounded knapsack)

Given an array of coins & amount, find fewest number of coins to make up that amount, return -1 if its not possible

Eg coins = [1, 2, 5]

amount = 11

for 11 ⇒ 1+....+1 (11 times) = 11

1+2+2+2+2+2 = 11

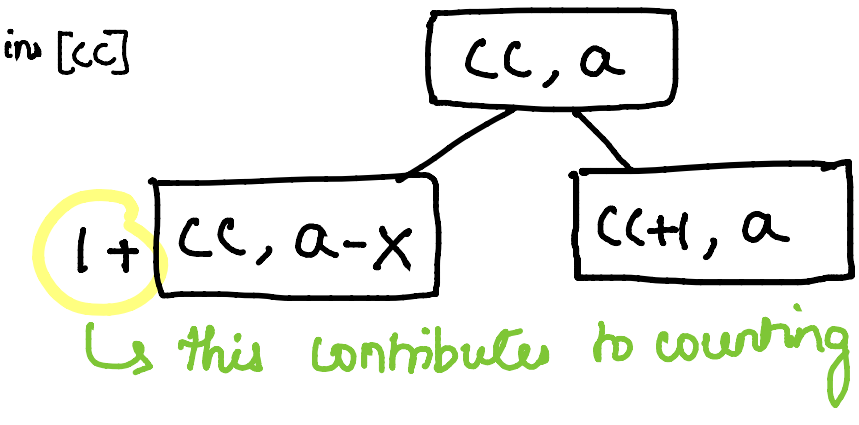
1+5+5 = 11

out of all the ways last has min coins.

coins = [_ x _ _ _]

x = coins[cc]

cc, a → current coin, amount



↳ this contributes to counting coins.

code →

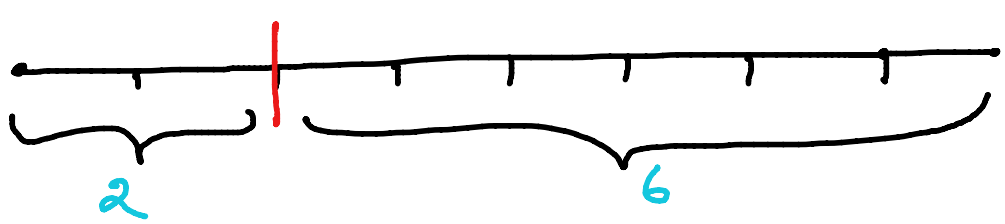
```cpp
class Solution {
public:

    int minimumCoins(int currentIndex, vector<int>& coins, int amount,vector<vector<int>>&memo){
        if(amount == 0)    return 0;
        if(currentIndex >= coins.size())    return 100000;  //Any Max Value outside boundary

        if(memo[currentIndex][amount]!=-1)  return memo[currentIndex][amount];

        int consider = 100000;
        if(coins[currentIndex]<=amount){
            consider = 1 + minimumCoins(currentIndex, coins, amount-coins[currentIndex], memo);
        }

        int notConsider = minimumCoins(currentIndex+1, coins, amount, memo);

        memo[currentIndex][amount]= min(consider,notConsider);
        return memo[currentIndex][amount];
    }

    int coinChange(vector<int>& coins, int amount) {

        vector<vector<int>>memo(coins.size()+1,vector<int>(amount+1,-1));
        int ans = minimumCoins(0, coins, amount, memo);

        return (ans==100000)? -1 : ans;
    }
};
```

(15) Rod Cutting →

Given a rod of length N and array of prices. find the max value that can be obtained by cutting rod.

Eg N = 8 prices = [1, 5, 8, 9, 10, 17, 17, 20]
(1 2 3 4 5 6 7 8 / 0 1 2 3 4 5 6 7)

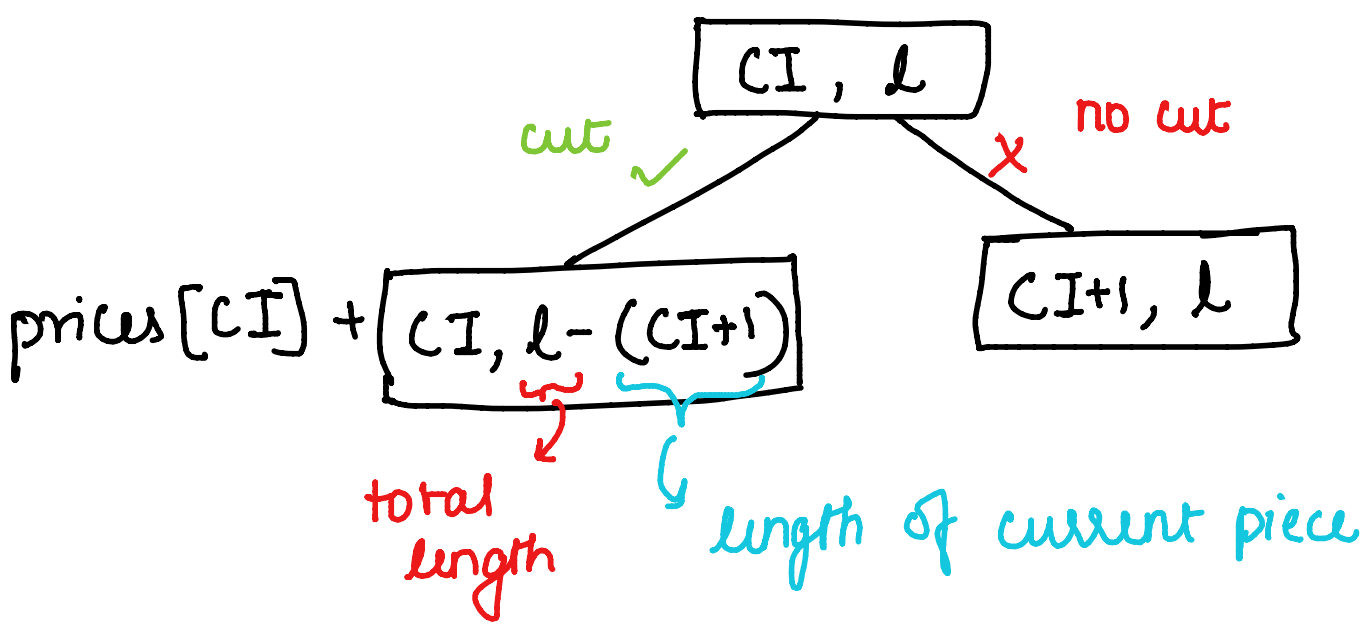
* price of a piece is prices[CI], whose length is CI+1



if we cut our rod into 2 pieces of length 2, 6 we get max value of 5+17 is 22.

→ there might be other ways, but this particular configuration returns max value.

* At any instance length of current piece is CI+1



code →

```cpp

class Solution{
  public:
    int maxProfit(int price[],int currentIndex, int n, vector<vector<int>>&memo){
        if(n==0)   return 0;
        if(currentIndex>=n)   return 0;

        if(memo[currentIndex][n]!=-1)   return memo[currentIndex][n];

        int selected = 0;
        if(currentIndex+1<=n){
            selected = price[currentIndex]+maxProfit(price, currentIndex, n-(currentIndex+1), memo);
        }
        int notSelected = maxProfit(price, currentIndex+1, n, memo);

        memo[currentIndex][n] = max(selected, notSelected);
        return memo[currentIndex][n];
    }
    int cutRod(int price[], int n) {
        vector<vector<int>> memo(n+1, vector<int>(n+1,-1));
        return maxProfit(price,0,n,memo);
    }
};
```

Find the rest on

https://linktr.ee/KarunKarthik

Follow Karun Karthik For More Amazing Content !