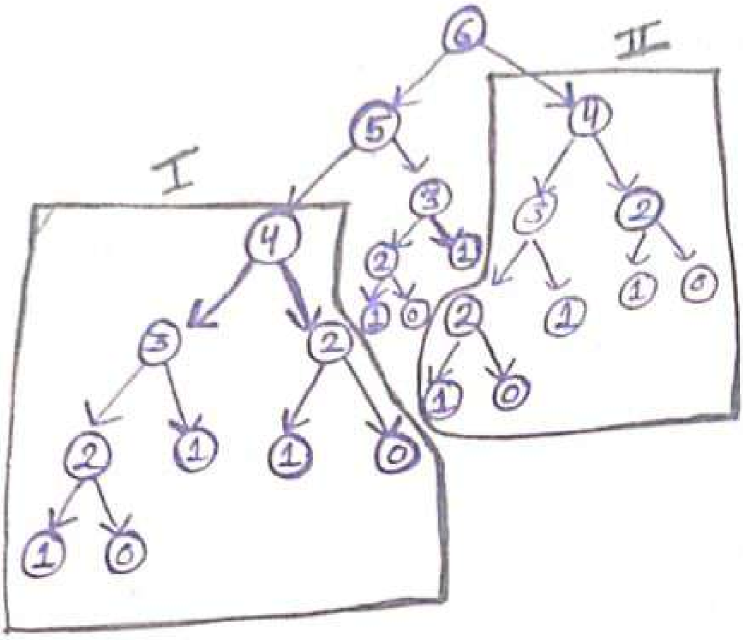
DP Foundations

- DP

Exa : Fibonacci recursive relation

$$F(i) = F(i-1) + F(i-2)$$
$$F(i) = i \quad (i \le 1)$$



→ It is identified that two different calls were made for $F(4)$, leading to redundancy and room for optimization.

→ DP will ensure that if $F(i)$ is fixed and calculated, the value is simply reuse instead of making another call.

- Important Theory Terms in DP :-

1. Overlapping subproblems

→ It occurs when the same subproblem is computed multiple times in the recursion tree.

→ For Exa : In the fibonacci sequence calculation, the call for $F(4)$ is made twice

→ DP addresses this by ~~[illegible]~~ ensuring that a subproblem is solved only once, and its result is ~~reduce~~ reused instead of recalculating it.

→ If a problem has no overlapping subproblems, DP will not reduce the complexity compared to a normal recursive or backtracking approach.

2. Optimal substructure

→ It means that the optimal solution to a problem depends on the optimal solutions of its smaller, same-type subproblems.

→ For Exa: calculating $F(i)$ depends on the values of $F(i-1)$ and $F(i-2)$.

3. Top-Down Approach : Recursive Approach

4. Bottom-Up Approach : Iterative Approach

- Two types of Problems :-

1. Counting : Finding the total no. of ways to achieve a certain goal.
Exa : Finding the no. of ways to make a sum X using a subset of an array.

2. Optimisation: Finding min or max
Exa : Find the min no. of elements needed to make a sum X using a subset of an array.

- Framework to solve a DP problem :-

1. Form
→ Detect its form (type). In DP there are 5 forms.
If the problem eventually leads to choosing which element can be take and which can't.

Element – Take / Not Take

Very similar to LCM. Form1 is simply direct backtracking.

2. State Formulation : The parameters used in the recursive function are called states.
→ The first parameter often tracks the index/element being considered (e.g. i)
→ The remaining states comes from the restriction in the problem, such as sum-so-far.

Exa : DP (i, sum_so_far) = (min no of ways to make total sum = X if [i -- N-1] & arr-sum = sum)

3. Transition

DP (i, sum taken)

Take ↙ Add ⊕ ↘ Not take

DP (i+1, sum taken + a[i]) DP (i+1, sum taken)

4. Time Limit check

T.C $\simeq$ (No. of states) $\times$ (1 + Average no of transitions)

For Exa: DP(i, sum) when N = array size
x = target sum

No. of states = $N \times X$

Average no of transition = 2 (Take / Not take)

Overall T.C $\simeq O(N * X (1+2)) = O(NX)$

Idea behind Formula

It is related to Topological ordering algorithm on a DAG formed by the dependencies b/w states.

$O(V + E)$

V → No. of states

E → No. of transitions.

```cpp
int dp[1001][1001];
int rec(int i,int sum){
    // pruning - 2
    if(sum>x)return 0;
    // basecase - 1
    if(i==n){
        if(sum==x)return 1;
        else return 0;
    }
    // cachecheck - 4
    if(dp[i][sum]!=-1){
        return dp[i][sum];
    }
    // transition - 2
    int ans = rec(i+1,sum+arr[i]) + rec(i+1,sum);
    // save and return. - 3
    return dp[i][sum] = ans;
}

void solve(){
    cin>>n>>x;
    for(int i=0;i<n;i++)cin>>arr[i];
    memset(dp,-1,sizeof(dp));
    cout<<rec(0,0)<<endl;
}
```

Sol 2 :-

- Form 1
- State : Min no. of element needed from i to N to make sum $= X$

$DP(i, sum)$ = min # of element needed for $(i..n)$ to make sum X.

- Transition

$$DP(i, sum)$$

Take ↙ ↘ Don't take

$1 + DP(i+1, sum + a[i])$ $\quad$ $0 + DP(i+1, sum)$

- TLE check

$\# S = N \cdot X$

$\# T = 2$

$\Rightarrow O(N \cdot X) = T.C$

Base case

	Accept	Reject
Count	1	0
Min	0	∞
Max	0	$-\infty$

```cpp
int dp[1001][1001];
int rec(int i,int sum){
    // pruning - 2
    if(sum>x)return 1e9;
    // basecase - 1
    if(i==n){
        if(sum==x){
            return 0;
        }else{
            return 1e9;
        }
    }
    // cachecheck - 4
    if(dp[i][sum] != -1)return dp[i][sum];
    // transition - 2
    int ans = min(1+rec(i+1,sum+arr[i]),rec(i+1,sum));
    // save and return. - 3
    return dp[i][sum] = ans;

}
```