October 19, 2025

# DP FOUNDATIONS

## * Dynamic Programming

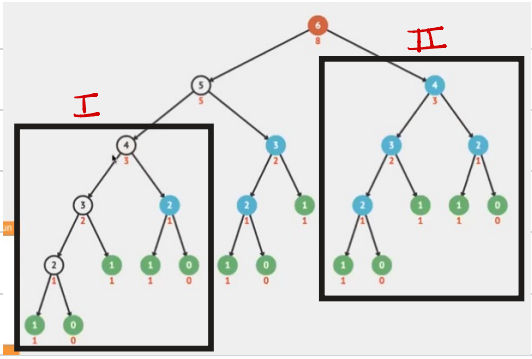
Let us understand this using an example :

Fibonacci Recursive relation :

$F(i) = F(i-1) + F(i-2)$

$F(i) = i \quad (i \leq 1)$

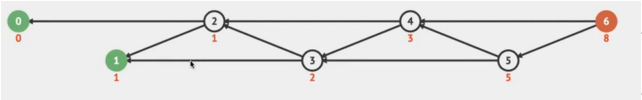
The Recursion tree for the same looks like :




In this recursive call, we identify that two different call was made for f(4) leading to redundancy and room for optimisation

DP, in this case, will ensure that if i is fixed and f(i) is already calculated then we can simply reuse that value instead of making another call.

Directed Acyclic Graph or DAG for our proposed optimisation using DP :



## * Important Theory Terms in DP -

### 1. Overlapping Subproblems

In our previous example, the f(6) recursive call had 2 separate calls for f(4) within the tree and that is the concept of overlapping subproblems or we can say f(6) had overlapping subproblems

This is something that we can possibly reduce and that's why DP can optimise problems better.

If there is no overlapping subproblems then the DP code will not help us in terms of reducing complexity from a normal recursion or backtracking

### 2. Optimal Substructure

If you are calculating some fibonacci, it will depend on smaller values of the same type and the optimal value of the same type. The bigger values will depend on the smaller values.

### 3. Top-Down

Recursive approach.

### 4. Bottom Up

Iterative approach

AlgoZenith

## * Two types of problems

1. Counting
2. Optimisation

Example :

Arr (N) = [1, 2, 3, 2, 5, 7]

Counting
- ① Find the # of ways to make sum = X using any (subset / subseq) of array. Every element - 1 time

Optimisation
- ② Find the min # of elements needed to make sum = X using only subset/subseq of array

## * Framework to solve a DP problem

1. Form, detect its form (type)

In DP, there are 5 forms.

a. If the problem eventually leads to choosing which element can be taken and which can't

Element - Take / Not Take

Very similar to LCCM. Form 1 is simply direct backtracking.

2. State Formulation, once the form is decided then we do state formulation.

What is state?

In a recursion, if we are trying to convert it into DP, the parameters that we use in our function is called states.

Example : In fibonacci, 'i' or index was the state.

For a particular state, the value will always be fixed.

DP ( i ) = ( Return )

(i ← State)

For DP state, the return val is simply of the type what is asked in the question. For example, for question 1 (counting), the return val will be of the type :

# of ways.

Subsets lies in form 1 and in form 1, we draw the diagram and for a particular element, i, we are deciding the level whether we will take it or not. (i came from the form)

In form 1, our DP fx^n call looks like :

DP( i, - - - - )

(i ← first parameter)

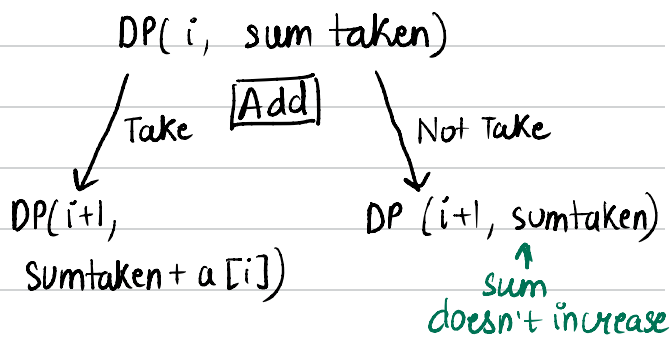
The rest of the states comes from restrictions in the problem. So, again in ques 1., why is all the subseq not a solution? Because not every one has a sum = X (restriction)

DP( i, sum_so_far )

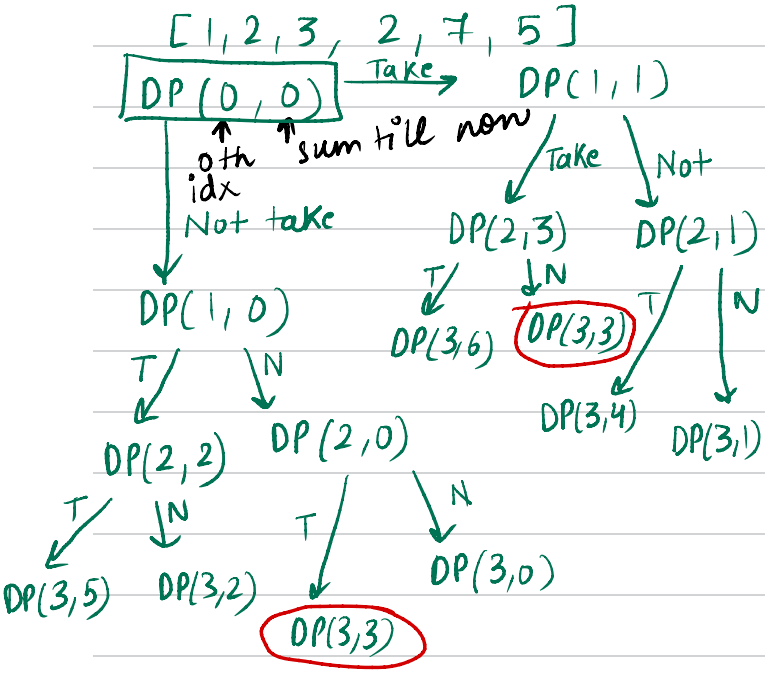
(sum_so_far → help us check sum = X)

DP(i, sum) = (# of ways to make total sum = X if [i--N-1] & cur-sum = sum)

3. Transition




Example: Solution 1.




So, we have traced till 3rd position

Let's take DP(3, 3)

[1, 2, 3, (2, 7, 5)]

Sum is 3 till now and let's say we had to make sum = 5. So, we just have add 2 to 3 to make it 5. How many ways to do this from i=3 to 5.

⇒ 1 (i=3)

Anything > 5 like DP(3,6) sum=6 here It will always return 0. So, this is called pruning. No need to make further tree.

DP(3, 4) sum = 4 so, we need 1

How many ways to make 1 from i=3 to 5? 0. No need to go further.

Similarly,

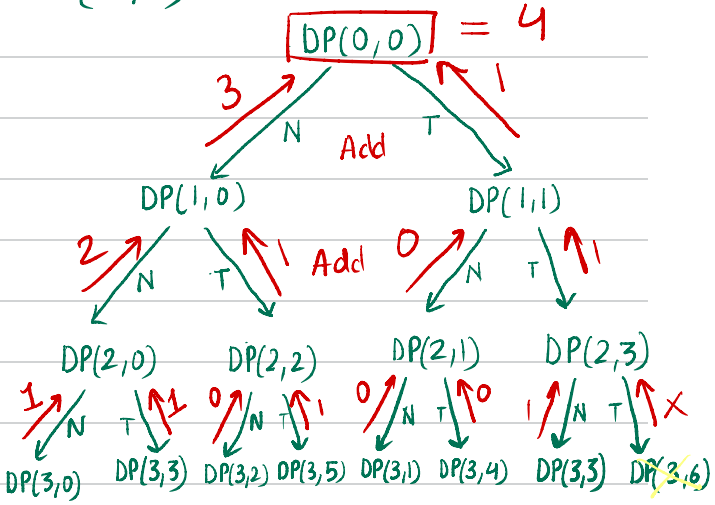
DP(3,1) = 0 (No 4)

DP(3,5) = 1

DP(3,2) = 0

DP(3,3) = 1

DP(3,0) = 1




For x = 5,

Ans = 4

## 4. Time Limit Check

T.C for above solution :

(#state) ( 1 + Avg time (#Transition))

N*X — sum : [0 -- x], i : [0 -- N)

1 cost for creating every state

2 in our ex.

$\Rightarrow (N * X (1+2))$

$\Rightarrow O(NX)$

Idea behind the formula :

In our DAG (graph), if we have to compute the values and there are some dependencies like in our prev problem, for one node, we needed to know the value of other 2 nodes. What problem are we solving here?

Topological Ordering algo

↓

Complexity : $O(V+E)$

$V(1 + \frac{E}{V})$

V — No. of States

$\frac{E}{V}$ — Avg. # of Transitions

```cpp
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
    // transition - 2
    int ans = rec(i+1,sum+arr[i]) + rec(i+1,sum);
    // save and return. - 3
    return ans;
}

void solve(){
    cin>>n>>x;
    for(int i=0;i<n;i++)cin>>arr[i];

    cout<<rec(0,0)<<endl;
}
```

Test Case 1

Input

6 5
1 2 3 2 7 5

Output

4

## Whole code using DP

```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;
}
```

## Solution 2>

- Form 1
- State

what is asked in ques? min. no. of ele needed from i to N to make sum = X.

$DP(i, sum)$ = (min # of ele needed for (i -- N) to make total sum = X given currently we have take sum from prev. elements)

form — i; restriction — sum

- Compute DP( i, sum)

How many min. no. of elements needed from i onwards.

AlgoZenith

DP(i, sum)

Take 'i' ↓ Don't take 'i'

1 + DP(i+1, sum+a[i]) ↑ including

0 + DP(i+1, sum) ↑ Not including the contribution

Example :

Took this also ↓

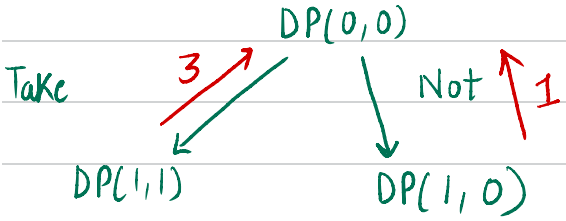
[ 1 [ 2 3 2 7 5 ]

4(5-1) sum=5

DP(1,1)

Min ele needed to make 4?

⇒ 2+1=3




```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;

}
```

- TL Check

$\#S = N \cdot X$

$\#T = 2$

$O(N \cdot X (1+2))$

$\Rightarrow O(N \cdot X)$

Base Case

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