Lecture-12: Subset Sum Equals to Target.

Problem

given an array arr[] with N positive integers. We need to find that if there exists a subset / subsequence with sum equal to 'K'. If yes return true / false.

Eg)

arr - [1, 2, 3, 4]

K = 4

Ans) True

Steps to form Recursive Solution.

S-1) Express the Problem in terms of Indexes.

We can ~~int~~ initially say f(n-1, target) which means that we need to find whether there exists a subsequence in the array from index 0 to (n-1), whose sum is equal to target. Generalizing.

f(ind, target) → check whether a subsequence exist in the array from index 0 to ind, whose sum is equal to index.

Base Case:

- if target == 0; it means we have already found a subsequence from the previous step, we can return true.
- if ind == 0; we need to check if arr[ind] == target if yes, return true else return false.

S-2) Try All possible Stuff.

- We will use the 'pick/not pick' technique. i.e.

(a) Include the current element in the subsequence.

We first try to find a subsequence without considering the current element, and make a recursive. f(n-1, target).

(b) Include the element in the subsequence.

Simple, this time we include the element arr[ind] in our subsequence and make recursive call to. f(n-1, target - arr[ind])

Note: only when target $\geq$ arr[ind].

3-3) Return (taken / not taken)

Among the taken / not taken we just need to find one 'true' and then return true for our function

Memoisation

- As usual, we create a ~~dp[n][k+1]~~ dp[n][k+1]. The size of the input array is 'n'; so index always lies between '0' to 'n-1'. The target can take any value between '0' to 'k'
- initialize with '-1'.
- follow as usual.

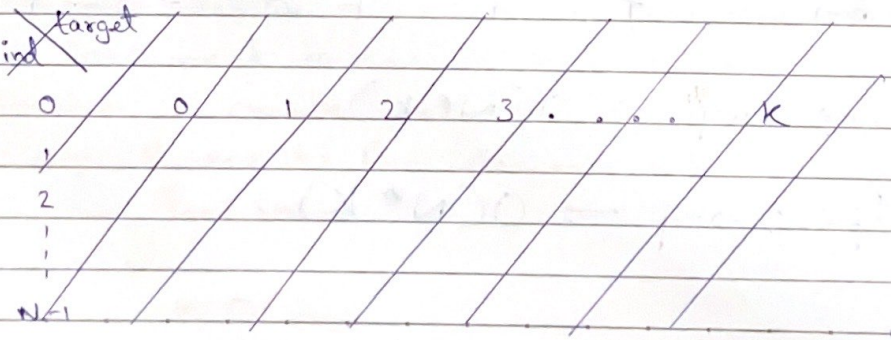
Code

```
bool f(int ind, int target, vector<int> &arr, vector<vector<int>>
                                                    &dp){
    if(ind==0)
        return arr[0]==target;
    if(target==0)
        return true;
    if(dp[ind][target]!=-1)
        return dp[ind][target];

    bool notpick = f(ind-1, target, arr, dp);
    bool pick = false;
    if(arr[ind]<=target)
        pick = f(ind-1, target-arr[ind], arr, dp);

    return dp[ind][target]= pick || notpick.
}
```

Time Comp — $O(N * K)$
Space Comp — $O(N * K) + O(N)$.

Tabulation.

Create the same dp[][] as in memoization. We can set it's type to bool and set all to false.



ind \ target	0	1	2	3	...	K
0	T	F	F	F	...	F
1	F	F	F	F	...	F
2	!	!	!	!		!
!	!	!	!	!		!
!	!	!	!	!		!
N-1	F	F	F	F		F

As always we first initialize the base condition from recursion :-

① If target == 0, ind can take any value from 0 to n-1 Thus we set the first column as true.

② The first row dp[0][] indicates that only the first element of the array is considered, therefore for the target value equal to arr[0], only call with that target will be true.
So set dp[0][arr[0]] = True.

③ Then we will set two nested loops for traversal the dp[][] as discussed previously.

④ Return dp[n-1][K] as answer.

ind \ target	0	1	2 ← if this was arr[0].	—	—	—	K
0	T	F	T	-	-		F
1	T	f	f	-	-	-	f
2	T	f	f	-	-	-	f
!	!	!	!	-	-	-	f
!	!	!	!	-	-	-	f
n-1	T	f	f	-	-	-	f

Time Comp → O(N * K)

Space Comp → O(N * K)

Code

```
bool f(int n, target, vector<int> &arr) {
    vector<vector<bool>> dp(n, vector<bool>(k+1, false));

    for(int i=0; i<n; i++)
        dp[i][0] = true;
    if(arr[0] <= target)
        dp[0][arr[0]] = True;
    for(int i=1; i<ind; i++) {
        for(int j=1; j<=target; j++){
            bool notpick = dp[i-1][j];
            bool pick = false;
            if(arr[i] <= target)
                pick = dp[i-1][target-arr[0]];

            dp[i][j] = pick || not pick;
        }
    }
    return dp[n-1][k];
}.
```

Space Optimization

$$dp[ind][target] = dp[ind-1][target] \;||\; dp[ind-1][target-arr[ind]]$$

Thus to calculate this we only need the prev. row values. Hence we don't need to main an entire array.

Note: Whenever we create a new row (say curr), we need to explicitly set it's first element to be true. acc. to base condition.

Space Comp → O(K).

Lecture 13 :- Partition Array into two ~~ways~~ Arrays to Minimise Sum Difference.

This problem all depends on prev. Lecture. From previous Lecture in Tabulation Code. If you try to understand. Then In tabulation the last row tells me whether is it possible to have subset sums from (0 → Target).

So all we have to do ~~to~~ is iterate over the last row, check if dp[n-1][S1] == true, then automatically S2 = ~~total~~ Sum - S1, and then keep a track of mini = min(mini, abs(S2-S1));

The problem was to simply return the minimum diff when partitioned. Ex [1, 2, 3, 4].

[1, 4] = 5 , [2, 3] = 5

Thus min Diff = 5-5 = 0

Time Comp → O(N * Target).

Space → O(N * Target)

PTO

Code

```
int minimumDiff(vector<int>& arr) {

    int totalSum = 0
    for(auto &p : arr)
        totalSum += p;
    vector<vector<bool>> dp(arr.size, vector<bool>(totalSum+1, false));

    for(int ind = 0; ind < arr.size(); ind++)
        dp[ind][0] = true;
    if(arr[0] <= totalSum)
        dp[0][arr[0]] = true;

    for(int ind = 1; ind < arr.size(); ind++) {
        for(int target = 1; target <= totalSum; target++) {
            bool notpick = dp[ind-1][target];
            bool pick = false.
            if(arr[ind] <= target)
                pick = dp[ind-1][target-arr[ind]];
            dp[ind][target] = pick || notpick;
        }
    }

    int ans = 1e9;
    for(int S1 = 0; S1 <= totalSum; S1++) {
        if(dp[arr.size()-1][S1] == true) {
            int S2 = totalSum - S1;
            ans = min(ans, abs(S2-S1));
        }
    }

    return ans;
}
```

Lecture 14: Count Subsets with Sum K.

The problem is exactly same as Lecture 12, the only difference is that here we have to calculate total no. of subsets equal to target.

I/p arr = [1, 2, 2, 3] ; K = 3.

O/p 3 {[1,2], [1,2], [3]}

Code (Memoization)

```
int f(int ind, int target, vector<int>& arr,
                    vector<vector<int>>& dp)
{
    if (target == 0)
        return 1;
    if (ind == 0)
        return arr[ind] == target;
    if (dp[ind][target] != -1)
        return dp[ind][target];

    int notpick = f(ind-1, target, arr, dp);
    int pick = 0;
    if (arr[ind] <= target)
        pick = f(ind-1, target-arr[ind], arr, dp);

    return dp[ind][target] = pick + notpick;
}
```

Tabulation → (i) Base Case
(ii) Look At Changing Variables/parameters with loops
(iii) Copy Recursion

Code

```
int f(int K ; vector<int>& arr) {

    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(K+1, 0));

    for(int i=0 ; i<n; i++)
        dp[i][0] = 1;

    if (arr[0] <= k)
        dp[0][arr[0]] = 1;

    for(int i=1 ; i<n ; i++) {
        for(int j = 1; j <= target ; j++) {
            int notpick = dp[i-1][j];
            int pick = 0 ;
            if(arr[i] <= j)
                pick = dp[i-1][j - arr[i]];

            dp[i][j] = pick + notpick;
        }
    }

    return dp[n-1][k];
}.
```

Lecture 15: 0/1 Knapsack

It's a classic knapsack problem. In which There are 'N' items, ~~items~~ Value Array, weight of each ~~Array~~ item and 'W' of bag. We have to calculate maximum value.

V	5	4	8	6
wt	1	2	3	4

N = 4 , W = 5

Ans → 13

This problem is also another DP on subsequences question. where we have the option to pick (or) notpick.

for Memoization

1. Express in terms of indexes - 'n' and 'wt'
2. Do All possible stuff - pick (or) not pick.
3. Take maximum of all.

Declare a DP[][] of N rows & W+1 colums.

Thus, the recurrence

f(ind, W) → Maximum value of items from index 0 to ind, with capacity W.

Base Case

if ind == 0, we can pick it if it's weight is less than (or) equal remaining weight of knapsack. Otherwise we Return 0.

Code

```
int f(ind ind, int W, vector<int>& wt, vector<int>& val,
                    vector<vector<int>>& dp) {

    if (ind == 0) {
        if (wt[ind] <= W)
            return val[ind];
        else
            return 0;
    }
    if (dp[ind][W] != -1) return dp[ind][W];
    int notpick = 0 + f(ind-1, W, wt, val, dp);
    int pick = INT_MIN;
    if (wt[ind] <= W)
        pick = val[ind] + f(ind-1, W-wt[ind], wt, val, dp);

    return dp[ind][W] = max(pick, notpick);
}
```

Time Comp - $O(N \cdot W)$.

Space Comp - $O(N \cdot W) + O(N)$

$O(N \cdot W)$ ↙ dp[][] ↘ recursion stack space.

Tabulation.

- At ind == 0 we are considering the first element, if the capacity of the knapsack is greater than the weight of the first item, we return val[0] as answer
- Use nested loop for traversing over ind & W.
- Copy the Recurrence.

```
int f (int n, int W, vector<int>& val, vector<int>& wt)
{
    vector<vector<int>>dp(n, vector<int>(W+1, 0));

    for(int i=wt[0]; i<=W; i++)
        dp[0][i] = val[0];

    for(int ind=1; ind<n; ind++) {
        for(int cap=0; cap<=W; cap++) {

            int notpick = 0 + dp[ind-1][cap];
            int pick = INT_MIN;
            if(wt[ind]<=cap)
                pick = val[ind] + dp[ind-1][cap-wt[ind]];

            dp[ind][cap]= max(pick, notpick);
        }
    }
    return dp[n-1][W];
}
```

Time Comp - O(NW)
Space Comp - O(NW)

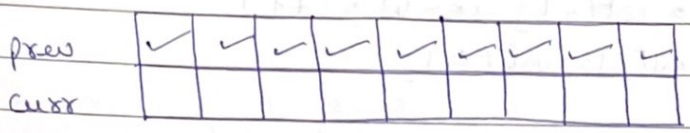
Space optimization.

look At the relation.

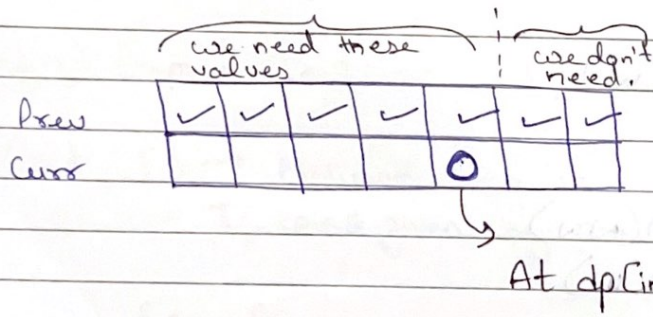
dp[ind][cap] = max(dp[ind-1][cap], dp[ind-1][cap-wt[ind]]);

Like always we only need a previous row to calculate current state. Thus we don't need to store full array. But we can do it in One Row/Array.

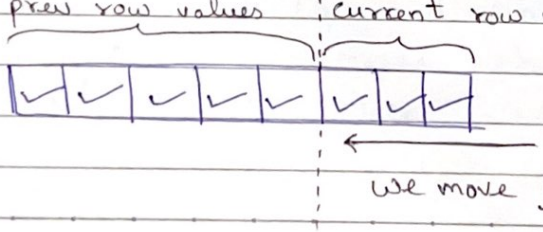
Notice, when we fill in 2-row method (curr & prev).



we initialize first row and using its values we calculate next row.
we can see from recurrence dp[ind-1][cap] & dp[ind-1][cap-wt[ind]].
We can say that if we are at column CAP, we only need values in left side and none in right side. as shown, because (CAP-wt[ind]) will always be less than CAP.



Since we don't value of Right, we start calculating from Right and we will overwrite on prev. ~~to~~ row from Right itself.



Code

```
int f(int n, int W, vector<int> & wt, vector<int> val)
{
    vector<int> prev(W, 0);

    for(int i = wt[0]; i<=W; i++)
        prev[i] = val[0];

    for(int ind = 1; ind < n; ind++) {
        for(int cap = W; cap >= 0; cap--) {
            int notpick = 0 + prev[cap];
            int pick = INT_MIN;
            if(wt[ind] <= cap)
                pick = val[ind] + prev[cap - wt[ind]];

            prev[cap] = max(pick, notpick);
        }
    }
    return prev[W];
}
```

Time Comp - O(NW).
Space Comp - O(W).

Lecture 16 - Minimum Coins (Leetcode: Coin Change)

Problem Link: We are given a target Sum and a 'N' sized array which represents unique denominations for coins. We have tell minimum no. of coins required to fulfill target amount and if it's not possible then return -1

Eg:-
arr = [1, 3, 2]
tar = 7

Ans = 3 → [3, 3, 1]

It's a typical Subsequence Problem where the main idea is to pick/not pick. But, here we can pick an element ∞ amount of times. Thus when we can pick an element we won't move to the next element. Instead we would max out on that and then move on.

Step-1) Express In terms Indexes.

f(ind, T) → Minimum coins required to form target T, coins given by arr[0...ind].

Step-2) Base Case.

If we are at ind==0 with a target (T), Then we need to understand, that we first have to check if the Amount And denomination at [0] is divisible COMPLETELY. If yes. Return T/coins[0] otherwise return INT_MAX. Eg.
arr[0]=4, T=12, since T%.arr[0]==0, we return T/arr[0]

Step-3) Try out all possible Stuff.

NOTPICK → The target Sum won't be affected and we will move to the next denomination. Recursive call is made to f(ind-1, T) we add 0 to our answer since we didn't pick it

Pick → We add '1' to our answer, since we are picking it. Subtract ~~arr[ind]~~ arr[ind] from Target Sum and call on the same denomination till target Sum less than the arr[ind] denomination.

Step 4) Return the Minimum of Take And Not take.

Memoization: By creating DP array of size [N][Amount+1].

Time Comp → O(N*T).

Space Comp → O(N*T) x O(N).

```
int MinimumCoins (vector<int>& arr, int T) {
    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(T+1, -1));
    int ans = f(n-1, arr, T, dp);
    if (ans >= 1e8)
        return -1;
    Return ans;
}
```

PTO

```
int f(int ind, vector<int>& arr, int T, vector<vector<int>>&dp)
{
    if (ind == 0) {
        if (T % arr[0] == 0)
            return T / arr[0];
        Else
            return 1e8;
    }
    if (dp[ind][T] != -1)
        return dp[ind][T];

    int NOTPICK = 0 + f(ind-1, arr, T, dp);
    int PICK = 1e8;
    if (T >= arr[ind])
        pick = 1 + (ind, arr, T - arr[ind], dp);

    return dp[ind][T] = min(PICK, NOTPICK);
}
```

Tabulation.

As Always try to first convert the recursive Base Cases i.e
At ind == 0 ; we are considering the first element if T % arr[ind] == 0, we initialize it to T / arr[ind] otherwise 1e8.
Traverse the 2D dp[][] with same recurrence as above.

PTO

Code

```
int f(vector<int>& arr, int T) {
    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(T+1, 0));

    for(int i=0; i<=T; i++) {
        if(i % arr[0] == 0)
            dp[0][i] = i / arr[0];
        else
            dp[0][i] = 1e8;
    }

    for(int i=1; i<n; i++) {
        for(int j=0; j<=T; j++) {

            int notpick = dp[i-1][j];
            int pick = 1e8;
            if(j >= arr[i])
                pick = 1 + dp[i][j-arr[i]];
            dp[i][j] = min(pick, notpick);
        }
    }

    return dp[n][T] >= 1e8 ? -1 : dp[n][T];
}
```

Time → O(N * T)

Space → O(N * T)

Space Optimization.

for Space optimization. use prev in place of dp[ind-1] and during initialization and curr during updation and return prev.

#Lecture 17: Coin Change 2

It's exactly the same problem as last Lecture. But this time we only need to calculate Total no. of ways, to reach target.

N = 3 Arr = {1, 2, 3} , target = 4

Ans → 4
{1, 1, 2}
{1, 3}
{1, 1, 1, 1}
{2, 2}

```
int f(vector<int>& arr, int T) {

    vector<vector<int>> dp(arr.size(), vector<int>(T+1, 0));
    for(int i = 0; i <= T; i++) {
        if (i % arr[0] == 0)
            dp[0][i] = i / arr[0];
    }
    for(int i=1; i<arr.size(); i++) {
        for(int j=0; j<=T; j++) {
            int notpick = dp[i-1][j];
            int pick = 0;
            if(arr[i] <= j)
                pick = dp[i][j-arr[i]];
            dp[i][j] = pick + notpick;
        }
    }
    int n = arr.size();
    return dp[n-1][T];
}
```

Lecture 18 : Unbounded Knapsack.

The Same problem as Knapsack, but here we have unlimited supplies WITH LIMITED Knapsack weight. We have to find Max profit

N = 3 W = 10
wt = { 2, 4, 6 }
val = { 5, 11, 13 }

Ans ⇒ 27
{ 11, 11, 5 }

It's the Same questions Logic as done in previous questions that when we pick the element we don't move to the next, instead we stay there and maximize our profit.

Memoization.

```
int f(vector<int>& wt, vector<int>& val, vector<vector<int>>& dp,
                                          int ind, int W){
    if(ind == 0)
        return (int)(W/wt[0]) * val[0];
    if(dp[ind][W] != -1)
        return dp[ind][W];

    int notpick = 0 + f(wt, val, dp, ind-1, W);
    int pick = 0;
    if(wt[ind] <= W)
        pick = val[ind] + f(wt, val, dp, ind, W-wt[ind]);

    return dp[ind][W] = max(notpick, pick);
}
```

Tabulation.

```
int f(vector<int>& wt, vector<int>& val, int W, int n) {

    vector<vector<int>> dp(n, vector<int>(W+1, 0));

    for(int i=0; i<=W; i++)
        dp[0][i] = (int)(i/wt[0]) * val[0];

    for(int i=1; i<n; i++) {
        for(int j=0; j<=W; j++) {

            int notpick = dp[i-1][j];
            int pick = 0;
            if(wt[i] <= j)
                pick = dp[i][j-wt[i]];
            dp[i][j] = max(pick, notpick);
        }
    }

    return dp[n-1][W];
```

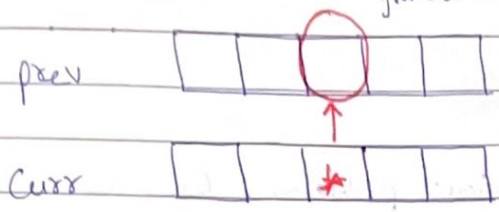
Space Optimization.

Observe At the recurrence.

```
dp[i][j] = max(dp[i-1][j], dp[i][j-wt[i]]);
```

Notice the values required: ~~dp[ind-1][cap dp[i]~~
dp[i-1][j] & dp[i][j-wt[i]], we can say that if we are at a column 'j', we will only require value from red box from prev and values will be from curr. Thus we don't need to store an entire array.

Only the same column from a row Above.




```
int F(vector<int>& wt, vector<int>& val, int W, int n) {

    vector<int> curr(W+1, 0);

    for(int i=0; i<=W; i++)
        curr[i] = ((int) W/wt[0]) * val[0];

    for(int i=1; i<n; i++) {
        for(int j=0; j<=W; j++) {

            int notpick = curr[j];
            int pick = INT_MIN;
            if(wt[i]<=j)
                pick = val[i] + curr[j-wt[i]];
            curr[j] = max(notpick, pick);
        }
    }

    return curr[W];
}
```

Lecture 19 : Rod Cutting Problem

This problem is similiar to previous problems. We are given a rod length 'n' and an array which tells the price of each piece. And each piece size is equal to their index no.

Eg. n = 5 , arr = [2, 5, 7, 8, 10].
Ans = 12

Possible partitions are.
[1, 1, 1, 1, 1] → max_cost (2+2+2+2+2) = 10.
[1, 1, 1, 2] → (2+2+2+5) = 11
⋮
[1, 2, 2] → [2+5+5] = 12

Same Concept of pick/not pick where we pick an element and don't move to the next since we want to max out on that

Base Case :- Try to understand, at ind == 0, it means the rod is being cut of length '1'. That means 'N' (whatever length left before reaching ind==0) Multiplied by arr[0] will give max cost at ind == 0.

Rest it is the same code for Memoization. using a dp[][] of size (n) & (n+1), where n → current index.
(n+1) → rod length left.

Code

Lecture-12: Subset Sum Equals to Target.

Problem

given an array arr[] with N positive integers. We need to find that if there exists a subset / subsequence with sum equal to 'K'. If yes return true / false.

Eg)

arr - [1, 2, 3, 4]

K = 4

Ans) True

Steps to form Recursive Solution.

S-1) Express the Problem in terms of Indexes.

We can ~~int~~ initially say f(n-1, target) which means that we need to find whether there exists a subsequence in the array from index 0 to (n-1), whose sum is equal to target. Generalizing.

f(ind, target) → check whether a subsequence exist in the array from index 0 to ind, whose sum is equal to index.

Base Case:

- if target == 0; it means we have already found a subsequence from the previous step, we can return true.
- if ind == 0; we need to check if arr[ind] == target if yes, return true else return false.

S-2) Try All possible Stuff.

- We will use the 'pick/not pick' technique. i.e.

(a) Include the current element in the subsequence.

We first try to find a subsequence without considering the current element, and make a recursive. f(n-1, target).

(b) Include the element in the subsequence.

Simple, this time we include the element arr[ind] in our subsequence and make recursive call to. f(n-1, target - arr[ind])

Note: only when target $\geq$ arr[ind].

3-3) Return (taken / not taken)

Among the taken / not taken we just need to find one 'true' and then return true for our function

Memoisation

- As usual, we create a ~~dp[n][k+1]~~ dp[n][k+1]. The size of the input array is 'n'; so index always lies between '0' to 'n-1'. The target can take any value between '0' to 'k'
- initialize with '-1'.
- follow as usual.

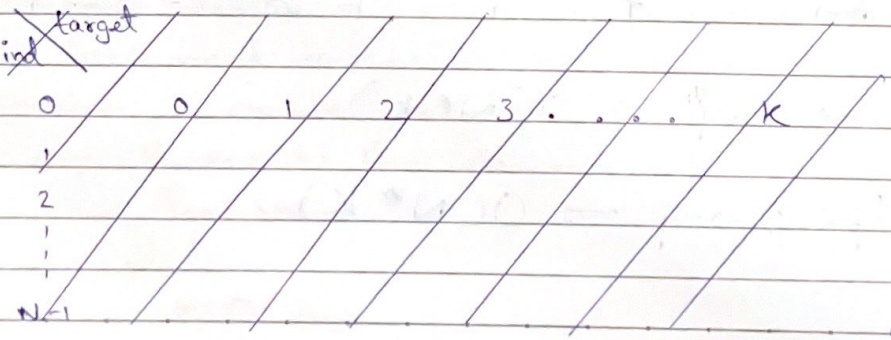
Code

```
bool f(int ind, int target, vector<int> &arr, vector<vector<int>>
                                                    &dp){
    if(ind==0)
        return arr[0]==target;
    if(target==0)
        return true;
    if(dp[ind][target]!=-1)
        return dp[ind][target];

    bool notpick = f(ind-1, target, arr, dp);
    bool pick = false;
    if(arr[ind]<=target)
        pick = f(ind-1, target-arr[ind], arr, dp);

    return dp[ind][target]= pick || notpick.
}
```

Time Comp — $O(N * K)$
Space Comp — $O(N * K) + O(N)$.

Tabulation.

Create the same dp[][] as in memoization. We can set it's type to bool and set all to false.



ind \ target	0	1	2	3	...	K
0	T	F	F	F	...	F
1	F	F	F	F	...	F
2	!	!	!	!		!
!	!	!	!	!		!
!	!	!	!	!		!
N-1	F	F	F	F		F

As always we first initialize the base condition from recursion :-

① If target == 0, ind can take any value from 0 to n-1 Thus we set the first column as true.

② The first row dp[0][] indicates that only the first element of the array is considered, therefore for the target value equal to arr[0], only call with that target will be true.
So set dp[0][arr[0]] = True.

③ Then we will set two nested loops for traversal the dp[][] as discussed previously.

④ Return dp[n-1][K] as answer.

ind \ target	0	1	2 ← if this was arr[0].	—	—	—	K
0	T	F	T	-	-		F
1	T	f	f	-	-	-	f
2	T	f	f	-	-	-	f
!	!	!	!	-	-	-	f
!	!	!	!	-	-	-	f
n-1	T	f	f	-	-	-	f

Time Comp → O(N * K)

Space Comp → O(N * K)

Code

```
bool f(int n, target, vector<int> &arr) {
    vector<vector<bool>> dp(n, vector<bool>(k+1, false));

    for(int i=0; i<n; i++)
        dp[i][0] = true;
    if(arr[0] <= target)
        dp[0][arr[0]] = True;
    for(int i=1; i<ind; i++) {
        for(int j=1; j<=target; j++){
            bool notpick = dp[i-1][j];
            bool pick = false;
            if(arr[i] <= target)
                pick = dp[i-1][target-arr[0]];

            dp[i][j] = pick || not pick;
        }
    }
    return dp[n-1][k];
}.
```

Space Optimization

$$dp[ind][target] = dp[ind-1][target] \;||\; dp[ind-1][target-arr[ind]]$$

Thus to calculate this we only need the prev. row values. Hence we don't need to main an entire array.

Note: Whenever we create a new row (say curr), we need to explicitly set it's first element to be true. acc. to base condition.

Space Comp → O(K).

Lecture 13 :- Partition Array into two ~~ways~~ Arrays to Minimise Sum Difference.

This problem all depends on prev. Lecture. From previous Lecture in Tabulation Code. If you try to understand. Then In tabulation the last row tells me whether is it possible to have subset sums from (0 → Target).

So all we have to do ~~to~~ is iterate over the last row, check if dp[n-1][S1] == true, then automatically S2 = ~~total~~ Sum - S1, and then keep a track of mini = min(mini, abs(S2-S1));

The problem was to simply return the minimum diff when partitioned. Ex [1, 2, 3, 4].

[1, 4] = 5 , [2, 3] = 5

Thus min Diff = 5-5 = 0

Time Comp → O(N * Target).

Space → O(N * Target)

PTO

Code

```
int minimumDiff(vector<int>& arr) {

    int totalSum = 0
    for(auto &p : arr)
        totalSum += p;
    vector<vector<bool>> dp(arr.size, vector<bool>(totalSum+1, false));

    for(int ind = 0; ind < arr.size(); ind++)
        dp[ind][0] = true;
    if(arr[0] <= totalSum)
        dp[0][arr[0]] = true;

    for(int ind = 1; ind < arr.size(); ind++) {
        for(int target = 1; target <= totalSum; target++) {
            bool notpick = dp[ind-1][target];
            bool pick = false.
            if(arr[ind] <= target)
                pick = dp[ind-1][target-arr[ind]];
            dp[ind][target] = pick || notpick;
        }
    }

    int ans = 1e9;
    for(int S1 = 0; S1 <= totalSum; S1++) {
        if(dp[arr.size()-1][S1] == true) {
            int S2 = totalSum - S1;
            ans = min(ans, abs(S2-S1));
        }
    }

    return ans;
}
```

Lecture 14: Count Subsets with Sum K.

The problem is exactly same as Lecture 12, the only difference is that here we have to calculate total no. of subsets equal to target.

I/p arr = [1, 2, 2, 3] ; K = 3.

O/p 3 {[1,2], [1,2], [3]}

Code (Memoization)

```
int f(int ind, int target, vector<int>& arr,
                    vector<vector<int>>& dp)
{
    if (target == 0)
        return 1;
    if (ind == 0)
        return arr[ind] == target;
    if (dp[ind][target] != -1)
        return dp[ind][target];

    int notpick = f(ind-1, target, arr, dp);
    int pick = 0;
    if (arr[ind] <= target)
        pick = f(ind-1, target-arr[ind], arr, dp);

    return dp[ind][target] = pick + notpick;
}
```

Tabulation → (i) Base Case
(ii) Look At Changing Variables/parameters with loops
(iii) Copy Recursion

Code

```
int f(int K ; vector<int>& arr) {

    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(K+1, 0));

    for(int i=0 ; i<n; i++)
        dp[i][0] = 1;

    if (arr[0] <= k)
        dp[0][arr[0]] = 1;

    for(int i=1 ; i<n ; i++) {
        for(int j = 1; j <= target ; j++) {
            int notpick = dp[i-1][j];
            int pick = 0 ;
            if(arr[i] <= j)
                pick = dp[i-1][j - arr[i]];

            dp[i][j] = pick + notpick;
        }
    }

    return dp[n-1][k];
}.
```

Lecture 15: 0/1 Knapsack

It's a classic knapsack problem. In which There are 'N' items, ~~items~~ Value Array, weight of each ~~Array~~ item and 'W' of bag. We have to calculate maximum value.

V	5	4	8	6
wt	1	2	3	4

N = 4 , W = 5

Ans → 13

This problem is also another DP on subsequences question. where we have the option to pick (or) notpick.

for Memoization

1. Express in terms of indexes - 'n' and 'wt'
2. Do All possible stuff - pick (or) not pick.
3. Take maximum of all.

Declare a DP[][] of N rows & W+1 colums.

Thus, the recurrence

f(ind, W) → Maximum value of items from index 0 to ind, with capacity W.

Base Case

if ind == 0, we can pick it if it's weight is less than (or) equal remaining weight of knapsack. Otherwise we Return 0.

Code

```
int f(ind ind, int W, vector<int>& wt, vector<int>& val,
                    vector<vector<int>>& dp) {

    if (ind == 0) {
        if (wt[ind] <= W)
            return val[ind];
        else
            return 0;
    }
    if (dp[ind][W] != -1) return dp[ind][W];
    int notpick = 0 + f(ind-1, W, wt, val, dp);
    int pick = INT_MIN;
    if (wt[ind] <= W)
        pick = val[ind] + f(ind-1, W-wt[ind], wt, val, dp);

    return dp[ind][W] = max(pick, notpick);
}
```

Time Comp - $O(N \cdot W)$.

Space Comp - $O(N \cdot W) + O(N)$

$O(N \cdot W)$ ↙ dp[][] ↘ recursion stack space.

Tabulation.

- At ind == 0 we are considering the first element, if the capacity of the knapsack is greater than the weight of the first item, we return val[0] as answer
- Use nested loop for traversing over ind & W.
- Copy the Recurrence.

```
int f (int n, int W, vector<int>& val, vector<int>& wt)
{
    vector<vector<int>>dp(n, vector<int>(W+1, 0));

    for(int i=wt[0]; i<=W; i++)
        dp[0][i] = val[0];

    for(int ind=1; ind<n; ind++) {
        for(int cap=0; cap<=W; cap++) {

            int notpick = 0 + dp[ind-1][cap];
            int pick = INT_MIN;
            if(wt[ind]<=cap)
                pick = val[ind] + dp[ind-1][cap-wt[ind]];

            dp[ind][cap]= max(pick, notpick);
        }
    }
    return dp[n-1][W];
}
```

Time Comp - O(NW)
Space Comp - O(NW)

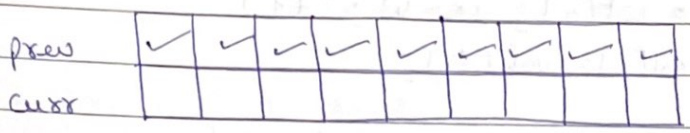
Space optimization.

look At the relation.

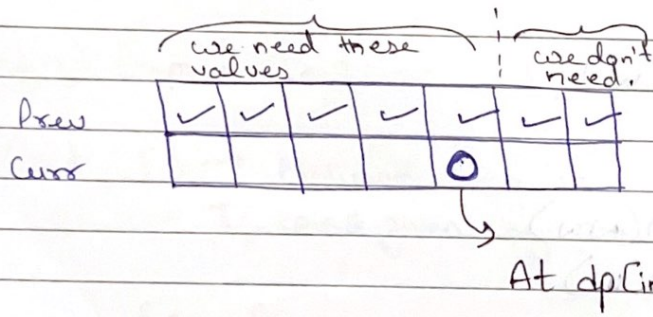
dp[ind][cap] = max(dp[ind-1][cap], dp[ind-1][cap-wt[ind]]);

Like always we only need a previous row to calculate current state. Thus we don't need to store full array. But we can do it in One Row/Array.

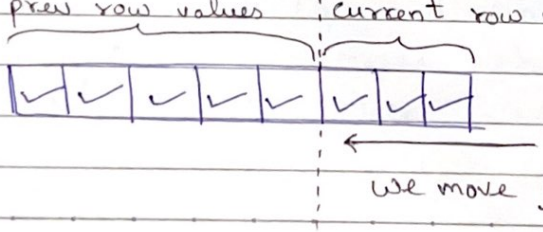
Notice, when we fill in 2-row method (curr & prev).



we initialize first row and using its values we calculate next row.
we can see from recurrence dp[ind-1][cap] & dp[ind-1][cap-wt[ind]].
We can say that if we are at column CAP, we only need values in left side and none in right side. as shown, because (CAP-wt[ind]) will always be less than CAP.



Since we don't value of Right, we start calculating from Right and we will overwrite on prev. ~~to~~ row from Right itself.



Code

```
int f(int n, int W, vector<int> & wt, vector<int> val)
{
    vector<int> prev(W, 0);

    for(int i = wt[0]; i<=W; i++)
        prev[i] = val[0];

    for(int ind = 1; ind < n; ind++) {
        for(int cap = W; cap >= 0; cap--) {
            int notpick = 0 + prev[cap];
            int pick = INT_MIN;
            if(wt[ind] <= cap)
                pick = val[ind] + prev[cap - wt[ind]];

            prev[cap] = max(pick, notpick);
        }
    }
    return prev[W];
}
```

Time Comp - O(NW).
Space Comp - O(W).

Lecture 16 - Minimum Coins (Leetcode: Coin Change)

Problem Link: We are given a target Sum and a 'N' sized array which represents unique denominations for coins. We have tell minimum no. of coins required to fulfill target amount and if it's not possible then return -1

Eg:-
arr = [1, 3, 2]
tar = 7

Ans = 3 → [3, 3, 1]

It's a typical Subsequence Problem where the main idea is to pick/not pick. But, here we can pick an element ∞ amount of times. Thus when we can pick an element we won't move to the next element. Instead we would max out on that and then move on.

Step-1) Express In terms Indexes.

f(ind, T) → Minimum coins required to form target T, coins given by arr[0...ind].

Step-2) Base Case.

If we are at ind==0 with a target (T), Then we need to understand, that we first have to check if the Amount And denomination at [0] is divisible COMPLETELY. If yes. Return T/coins[0] otherwise return INT_MAX. Eg.
arr[0]=4, T=12, since T%.arr[0]==0, we return T/arr[0]

Step-3) Try out all possible Stuff.

NOTPICK → The target Sum won't be affected and we will move to the next denomination. Recursive call is made to f(ind-1, T) we add 0 to our answer since we didn't pick it

Pick → We add '1' to our answer, since we are picking it. Subtract ~~arr[ind]~~ arr[ind] from Target Sum and call on the same denomination till target Sum less than the arr[ind] denomination.

Step 4) Return the Minimum of Take And Not take.

Memoization: By creating DP array of size [N][Amount+1].

Time Comp → O(N*T).

Space Comp → O(N*T) x O(N).

```
int MinimumCoins (vector<int>& arr, int T) {
    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(T+1, -1));
    int ans = f(n-1, arr, T, dp);
    if (ans >= 1e8)
        return -1;
    Return ans;
}
```

PTO

```
int f(int ind, vector<int>& arr, int T, vector<vector<int>>&dp)
{
    if (ind == 0) {
        if (T % arr[0] == 0)
            return T / arr[0];
        Else
            return 1e8;
    }
    if (dp[ind][T] != -1)
        return dp[ind][T];

    int NOTPICK = 0 + f(ind-1, arr, T, dp);
    int PICK = 1e8;
    if (T >= arr[ind])
        pick = 1 + (ind, arr, T - arr[ind], dp);

    return dp[ind][T] = min(PICK, NOTPICK);
}
```

Tabulation.

As Always try to first convert the recursive Base Cases i.e
At ind == 0 ; we are considering the first element if T % arr[ind] == 0, we initialize it to T / arr[ind] otherwise 1e8.
Traverse the 2D dp[][] with same recurrence as above.

PTO

Code

```
int f(vector<int>& arr, int T) {
    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(T+1, 0));

    for(int i=0; i<=T; i++) {
        if(i % arr[0] == 0)
            dp[0][i] = i / arr[0];
        else
            dp[0][i] = 1e8;
    }

    for(int i=1; i<n; i++) {
        for(int j=0; j<=T; j++) {

            int notpick = dp[i-1][j];
            int pick = 1e8;
            if(j >= arr[i])
                pick = 1 + dp[i][j-arr[i]];
            dp[i][j] = min(pick, notpick);
        }
    }

    return dp[n][T] >= 1e8 ? -1 : dp[n][T];
}
```

Time → O(N * T)

Space → O(N * T)

Space Optimization.

for Space optimization. use prev in place of dp[ind-1] and during initialization and curr during updation and return prev.

#Lecture 17: Coin Change 2

It's exactly the same problem as last Lecture. But this time we only need to calculate Total no. of ways, to reach target.

N = 3 Arr = {1, 2, 3} , target = 4

Ans → 4
{1, 1, 2}
{1, 3}
{1, 1, 1, 1}
{2, 2}

```
int f(vector<int>& arr, int T) {

    vector<vector<int>> dp(arr.size(), vector<int>(T+1, 0));
    for(int i = 0; i <= T; i++) {
        if (i % arr[0] == 0)
            dp[0][i] = i / arr[0];
    }
    for(int i=1; i<arr.size(); i++) {
        for(int j=0; j<=T; j++) {
            int notpick = dp[i-1][j];
            int pick = 0;
            if(arr[i] <= j)
                pick = dp[i][j-arr[i]];
            dp[i][j] = pick + notpick;
        }
    }
    int n = arr.size();
    return dp[n-1][T];
}
```

Lecture 18 : Unbounded Knapsack.

The Same problem as Knapsack, but here we have unlimited supplies WITH LIMITED Knapsack weight. We have to find Max profit

N = 3 W = 10
wt = { 2, 4, 6 }
val = { 5, 11, 13 }

Ans ⇒ 27
{ 11, 11, 5 }

It's the Same questions Logic as done in previous questions that when we pick the element we don't move to the next, instead we stay there and maximize our profit.

Memoization.

```
int f(vector<int>& wt, vector<int>& val, vector<vector<int>>& dp,
                                          int ind, int W){
    if(ind == 0)
        return (int)(W/wt[0]) * val[0];
    if(dp[ind][W] != -1)
        return dp[ind][W];

    int notpick = 0 + f(wt, val, dp, ind-1, W);
    int pick = 0;
    if(wt[ind] <= W)
        pick = val[ind] + f(wt, val, dp, ind, W-wt[ind]);

    return dp[ind][W] = max(notpick, pick);
}
```

Tabulation.

```
int f(vector<int>& wt, vector<int>& val, int W, int n) {

    vector<vector<int>> dp(n, vector<int>(W+1, 0));

    for(int i=0; i<=W; i++)
        dp[0][i] = (int)(i/wt[0]) * val[0];

    for(int i=1; i<n; i++) {
        for(int j=0; j<=W; j++) {

            int notpick = dp[i-1][j];
            int pick = 0;
            if(wt[i] <= j)
                pick = dp[i][j-wt[i]];
            dp[i][j] = max(pick, notpick);
        }
    }

    return dp[n-1][W];
```

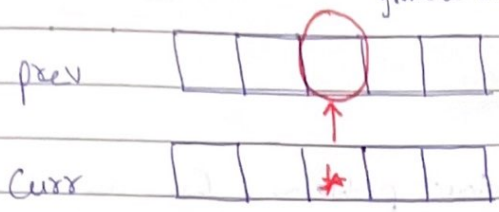
Space Optimization.

Observe At the recurrence.

```
dp[i][j] = max(dp[i-1][j], dp[i][j-wt[i]]);
```

Notice the values required: ~~dp[ind-1][cap dp[i]~~
dp[i-1][j] & dp[i][j-wt[i]], we can say that if we are at a column 'j', we will only require value from red box from prev and values will be from curr. Thus we don't need to store an entire array.

Only the same column from a row Above.




```
int F(vector<int>& wt, vector<int>& val, int W, int n) {

    vector<int> curr(W+1, 0);

    for(int i=0; i<=W; i++)
        curr[i] = ((int) W/wt[0]) * val[0];

    for(int i=1; i<n; i++) {
        for(int j=0; j<=W; j++) {

            int notpick = curr[j];
            int pick = INT_MIN;
            if(wt[i]<=j)
                pick = val[i] + curr[j-wt[i]];
            curr[j] = max(notpick, pick);
        }
    }

    return curr[W];
}
```

Lecture 19 : Rod Cutting Problem

This problem is similiar to previous problems. We are given a rod length 'n' and an array which tells the price of each piece. And each piece size is equal to their index no.

Eg. n = 5 , arr = [2, 5, 7, 8, 10].
Ans = 12

Possible partitions are.
[1, 1, 1, 1, 1] → max_cost (2+2+2+2+2) = 10.
[1, 1, 1, 2] → (2+2+2+5) = 11
⋮
[1, 2, 2] → [2+5+5] = 12

Same Concept of pick/not pick where we pick an element and don't move to the next since we want to max out on that

Base Case :- Try to understand, at ind == 0, it means the rod is being cut of length '1'. That means 'N' (whatever length left before reaching ind==0) Multiplied by arr[0] will give max cost at ind == 0.

Rest it is the same code for Memoization. using a dp[][] of size (n) & (n+1), where n → current index.
(n+1) → rod length left.

Code