→ Deepak Manglani

Date.....
Page.....

Dynamic Programming

DP = Enhanced Recursion.

-> When to apply DP?

(i) Choices (2 or more) (ii) Optimal

-> Note :- Never start solving Dp problems directly by making table (Bottom-Up Manner).

-> How to start Writing DP code?

(i) Recursive Approach
(ii) Memoization
(iii) Bottom - Up (Tabulation)

Deepak Manglani

Dynamic Programming

(i) 0-1 Knapsack
(ii) Unbounded Knapsack
(iii) Fibonacci
(iv) LCS
(v) LIS
(vi) Kadane's Algorithm
(vii) Matrix Chain Multiplication
(viii) DP on trees
(ix) DP on Grid
(x) Others

} Parent Problems

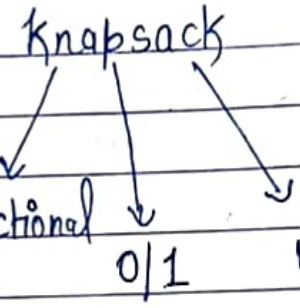
Date.....
Page.....

Knapsack Problem

-> What is Knapsack?

Knapsack is a kind of bag in which we have to store the items with maximum profit.

Types of Knapsack :-



-> Fractional KNapsack :-

It is a greedy problem, in which we need to store the maximum val/weight into the bag, i.e., we can store a part of an item by cutting it.

-> 0/1 Knapsack :-

In this problem, we use dp in this either we include an item into the bag or just exclude it.

-> Unbounded :- It is same as 0/1 but, it have an infinite supply of every item.

Date:.....
Page.....

(I) 0-1 Knapsack Problem

(i) Subset Sum
(ii) Equal Sum Partition
(iii) Count of Subset Sum
(iv) Minimum Subset Sum Diff
(v) Target Sum
(vi) No of Subset Sum given diff.

} Variations

-> **Problem :-** 0/1 Knapsack

Identification :- It is a DP problem :-
(i) Because of choices
(ii) Because of optimal ans
↓
Max, Min, Largest, et

I/P :-

wt[] :- 1 3 4 5
val[] :- 1 4 5 7
W :- 7 kg

O/P :- Max. Profit

DP Code :- Recursion → Memoization → Tabulation

How to Write Recursive Code. :-

Recursive Code

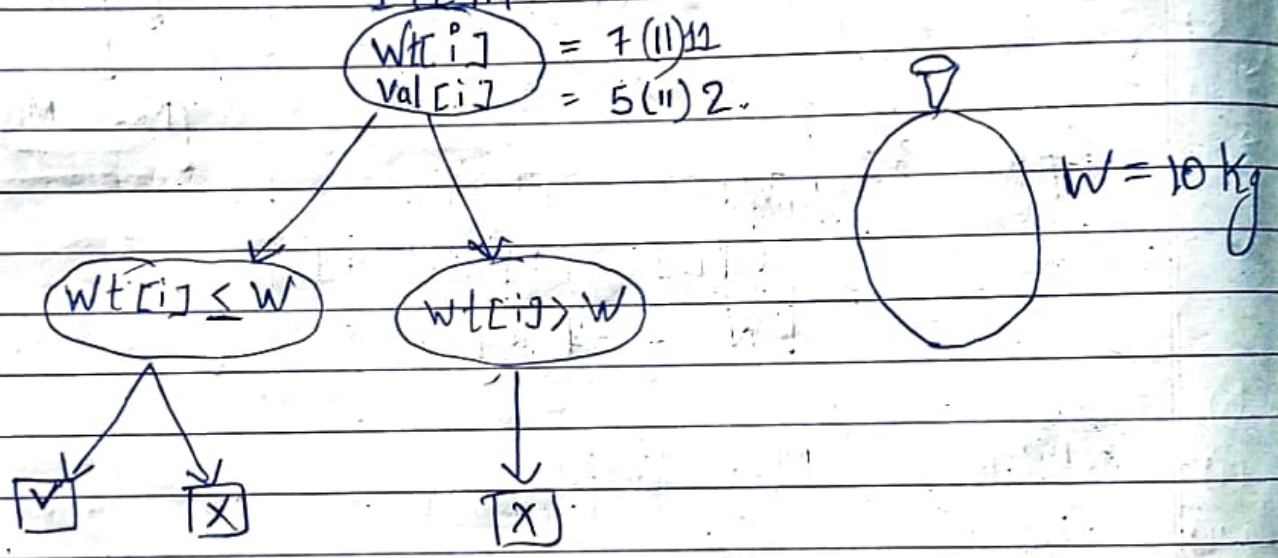
{
(i) Base Cond.
(ii) Choice Diagram.
}

(i) Base Condition :- Think of the smallest valid input

In this problem, it will be when we have an empty array or size of the bag is 0.

(ii) Choice Diagram :-

For an element/item :-



The Maximum of the two choices above in the left sub-tree will be our answer.

OR-

It can be from the right subtree.

Date.....
Page.....

Recursive Code :-

Max Profit)

```
int knapsack(int wt[], int val[], int W, int n)
{

    if (n == 0 || W == 0) return 0;
    if (wt[n-1] <= W)
    {
        return max(val[n-1] + knapsack(wt, val, W-wt[i], n-1
                , knapsack(wt, val, W, n-1));
    }
    else if (wt[n-1] > W)
    {
        return knapsack(wt, val, W, n-1);
    }

}
```

Date.....
Page.....

Memoize Code :-

We will add 4 lines to the recursive code to memoize it.
The basic idea is to make an array of changing variables in the recursive code

```
int dp[n+1][W+1]; memset(dp, -1, sizeof(dp));
int Knapsack (int wt[], int val[], int W, int n)
{
    if (n==0 || W==0) return 0;
    if (dp[n][W] != -1) return dp[n][W];
    if (wt[n-1] <= W)
    {
        return dp[n][W] = max( val[n-1] + Knapsack(W-wt[n-1], n-1)
                             , Knapsack(n-1) );
    }
    else if (wt[n-1] > W)
    {
        return dp[n][W] = Knapsack(wt, val, W, n-1);
    }
}
```

Date.....
Page.....

Tabulation Code (Bottom - Up Code) :-

The basic idea behind Tabulation is to remove the recursive calls, that we have seen in the Memoization technique into iterative form.

Tabulation can be performed in the following two steps :-

(i) Initialization (Basically these are the BC of recursion)

(ii) Changing Recursive functions into iterative one.

```
int dp[n+1][W+1];
```

-> Initialisation :-

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

-> Iterative Version

```
for(int i=1; i<n+1; i++)
for(int j=1; j<W+1; j++)
  if(wt[i-1] ≤ j)
      dp[i][j] = max(dp[i-1][j-wt[i-1]]+val[i-1],
                     dp[i-1][j]);
  else dp[i][j] = dp[i-1][j];
return dp[n][W];
```

Date.....
Page.....

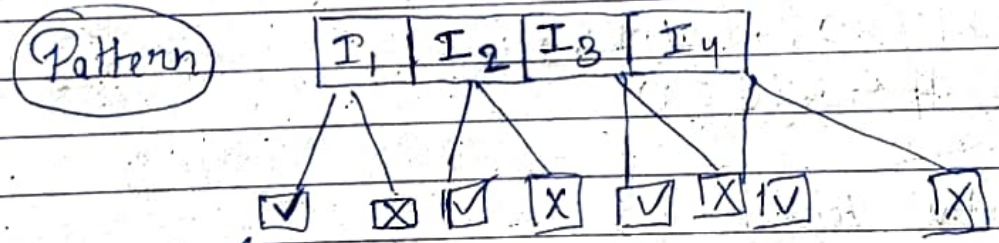
-> Revision of Knapsack and Introduction to its variations/ further problems

Q: How to Identify if a problem in dp can be solved using Knapsack?

Whenever we see a pattern where, we have a capacity and an item array to fill that capacity with optimal profit that problem falls under Knapsack

Sometimes, this can also happen that we have only 1 array rather than two (wt[] and val[]), then, we neglect the value array and consider that array as our weight array.

Example :-



Choices and a constraint as W that we can't take value more that W in our ans.

And, there is also the change in the initialization in its variation

Date.....
Page.....

(i) Subset - Sum Problem

Problem :-

Give an array and a sum, find whether by adding some elements of the array we get sum or not.

Example :- I/p arr : {2, 3, 7, 8, 10}

sum :- 11

O/P :- True.

I/P arr : {2, 3, 7, 8, 10}

sum :- 14

O/P :- False

Similarity with 0/1 Knapsack :-

As we can see here we have given a single array, rather than two that we have seen in Knapsack. We consider it as our weight array as discussed in the revision.

Also we have choices to include or not to include an array element onto our set.

And we have given Maximum capacity as sum analogous to (W in Knapsack)

Date.....
Page.....

→ Code Variations :-

(i) Initialisation :-

If array size is 0 and sum is 0.
So, Is it possible to get the sum?
Yes, it is always possible empty subset.
{}

If sum is greater that 0 but array is empty is it possible to get sum?
No.

So, we got our initialisation as :-

```
dp[n+1][sum+1];

for(int i=0; i<n+1; i++)
for(int j=0; j<sum+1; j++)
    if (i==0) dp[i][j] = false;
    if (j==0) dp[i][j] = true;
```

(ii) Main Code :-

wt[] → arr[]
W → sum
int (Max. Profit) O/P → O/P (T/F) bool
Basic Variation

Date.....
Page.....

There is no purpose of max function in boolean so, we use (||) operator.

```
if (arr[i-1] <= j)
    dp[i][j] = (dp[i-1][j - arr[i-1]] || dp[i-1][j]);
else
    dp[i][j] = dp[i-1][j];
```

Then, we have to :-

```
return dp[n][sum];
```

Deepak Mangaw

(ii) Equal Sum Partition

-> Problem :-

Given an array, can we partition it into two subsets such that if we take all the elements and put it into these subsets we will get equal sum in these subsets.

I/p :- arr :- {1, 5, 5, 11}

O/p :- True // since, {1,5,5} == {11}

I/p :- arr :- {1, 2, 3, 4}

O/p :- False.

Date.....
Page.....

→ Some Intuition and Observation :-

Since, we have to take all the elements of the array and partition it into two subsets of equal sum, The total sum of the array should never be odd.

Proof :-

Let, say sum of one subset is x

since, subsetsum(S_1) = subsetsum(S_2)

$$x = x$$

$$x + x = 2x \quad (\text{sum of the array})$$

It should always be divisible by 2.

Otherwise, we written false.

Also, as we can see it would be enough to show that if we get a subset of from the whole array of half sum $\left(\frac{sum}{2}\right)$.

We will get our answer.

Date.....
Page.....

Code :-

```
bool EqualSumPart(int arr[], int n, int sum)
{
    if (sum % 2) return false;
    if
    return SubsetSum(int arr[], int n, int sum/2);
}
```

(iii) Count of Subset Sum with a Given Sum

=> Problem :-

Given an array and a sum, find how many subsets of given sum are possible

Ex :-

I/P :- {1, 2, 3, 5, 8, 10}

O/P :- 3

Explanation :- {{2, 3, 5}, {10}, {2, 8}}

-> Similarity with Subset Sum :-

In Subset Sum problem, we only need to output if a subset of given sum is possible or not.

But here, in this problem we also need to return the count of the subset.

-> Code Variation :-

(i) Initialisation :-

```
for (int i=0; i<n+1; i++)
for (int j=0; j<sum+1; j++)

  if (i==0)  dp[i][j] = 0;
  if (j==0)  dp[i][j] = 1;
```

(ii) Main Code :-

```
if ( arr[i-1] > sum(j) )  dp[i][j] = dp[i-1][j]
else dp[i][j] = dp[i-1][j-arr[i-1]] + dp[i-1][j]
```

Date.....
Page.....

(iv) Minimum Subset Sum Difference :-

-> Problem :-

Given an array of integers, we have to partition it into two subsets of with minimum difference i.e.,

$$S_1 - S_2 = \text{minimum}$$

Deepak Manglani

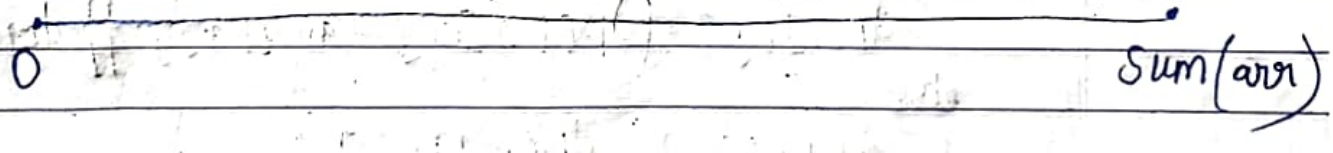
-> Intution and Observations :-

Given two subsets S_1 and S_2 to us :-
such that :-

$$S_1 - S_2 = \min(\text{diff})$$

Let's try to find a range of this difference if we have S_1 = array, S_2 = { }

then, there difference will be the sum of array i.e., range will be :-



Let's claim here that atleast one of the subset will always present in the first half completely.

This can be proved easily since,

$S_1 + S_2 = sum(arr)$

So, we can find out the difference as

$$(sum(arr) - S_1) - S_1 = diff$$

$$\boxed{diff = sum(arr) - 2 \times S_1}$$

-> Code :-

```
int dp[n+1][sum+1];
for(int i=0; i<n+1; i++)
for(int j=0; j<sum+1; j++)
    if(i==0) dp[i][j] = false;
    if(j==0) dp[i][j] = true;

for(int i=1; i<n+1; i++)
for(int j=1; j<sum+1; j++)

    if(arr[i-1] <= sum)
        dp[i][j] = (dp[i-1][j-arr[i-1]] || dp[i-1][j])
    else
        dp[i][j] = dp[i-1][j];
```

Date.....
Page.....

```
for(int i = sum/2 ; i >= 0 ; i--)
{
    if (dp[n][i] == true)
    {
        diff = sum - 2 x i ;
        break;
    }
}
return diff;
```

(iv) Count the number of subset with given Difference

-> Problem :-

Given an array and difference between the two subset sum, count the number of subset with this difference

Ex :-

I/P :- {1, 1, 2, 3}

O/P :- 3

Explanation :- {1, 2} and {1, 3} {1, 2} and {1, 1, 2} and {3}

Deepak Mangani

→ Intuitions and Observations :-

Let, the two subsets be s_1 and s_2 then,

$$s_1 - s_2 = \text{diff} \quad \longrightarrow (i)$$

Also, we know that

$$s_1 + s_2 = s(\text{sum}(arr)) \quad \longrightarrow (ii)$$

From adding eq. (i) and (ii) we get,

$$2s_1 = S + \text{diff}$$

$$\boxed{s_1 = \frac{S + \text{diff}}{2}}$$

If (S + diff) is not divisible by 2 we will simply return 0

Otherwise we will find the count of subset sum of subset S_1 and this will be our answer