**DP = Recursion + Storage**

Storage can be don’t through memorization or topdown approach.

**How to identify problems that it is based on DP.**

1. Choice

There will be choice to user to consider the element or not.

1. Overlapping

If there is overlapping in the call. Overlapping means, suppose there is recursion call on left and right and there is scenario that some part of the left recursion is same as right recursion. So its better to store that call op somewhere and while recursion in right side if same condition is met we will return the result without doing all the same recursion once again.

1. Optimal/Minimal/Largest/Smallest

In the problem it will be asked like find the optimal solution/ Minimal Solution.

So if these scenarios are matching in any problem they we can think of to use DP.

**Note: if there is recursion only one side either left or right then we can’t use the DP.**

**Knapsack**

1. Fractional Knapsack……We can store the fraction of item in the item. Suppose knapsack weight is 10 kg and it is filled till 9 kg and we have item which weight is 2 kg. so here we can add the half of this item to make the knapsack weight 10 kg. if we are taking half of any item then profit also will be half for that item. This is greedy approach and here we don’t need to use DP.
2. 0/1 Knapsack…….. Here we cant take item in fraction and we can take the one item once. Either we take the item or leave the item. Suppose knapsack weight is 10 kg and it is filled till 9 kg and we have item which weight is 2 kg. so here we cannot add the half of this item, we have to take full item. But if we take the full item then knapsack weight will increase to 11 kg. so first we need to remove 1 kg from knapsack so knapsack weight will be 8 kg then add this item of 2 kg to make the knapsack weight 10 kg.
3. Unbounded Knapsack….. Here we can take the same item multiple times.

**0/1 Knapsack**

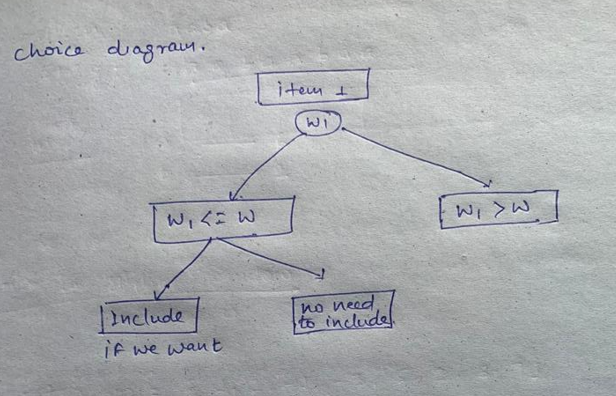
1. **Knapsack**

**int**[] wt = { 1, 4, 5, 6, 2, 3, 7, 8 };

**int**[] value = { 2, 3, 4, 5, 1, 6, 8, 0 };

**int** w = 10;

Here wt[], value[] and weight is given. We have to find out maximum profit by picking the element and weight should not exceed w kg.



Here suppose W is weight of the knapsack and w1 is weight of the item. If w1 > w then we don’t have choice to add it as knapsack cant store that much weight. If w1 <= w then we have choice to take it or not. If by taking this item we are getting the solution then we will take this item and if we find that even if we take this item we wont get solution then we wont take it. So here in this case we have choice to take it or not.

Knapsack function

**private** **int** solve(**int**[] wt, **int**[] value, **int** w, **int** n) {

// Base condition

// Choice diagram

}

**Base Condition: Think of the smallest valid input**

**int**[] wt = { 1, 4, 5, 6, 2, 3, 7, 8 };

**int**[] value = { 2, 3, 4, 5, 1, 6, 8, 0 };

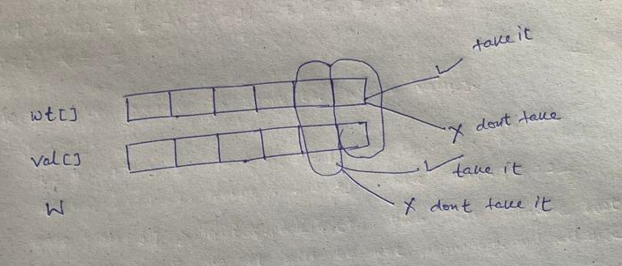
**int** w = 10;

Here smallest size of the array can be 0 and weight of the knapsack can be 0. So this is the smallest valid input so this will be our base condition. Just think about the output of this base condition. Suppose we don’t have any element in the array or weight size is 0 then our profit will also be 0. So this will be our output for this base condition.

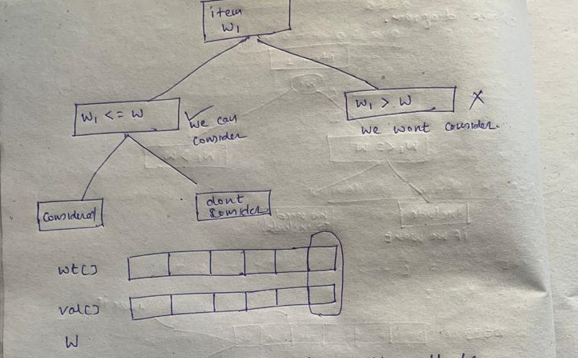
**if** (n == 0 || w == 0)

**return** 0;

Recursive function… function(n) ----- > function(smaller input)….we should always call the recursive function on the smaller input.



Here in the array we start from the end of the array. We will decide wether we need to take that item or not. If we take it or not once we make the decision we will move ahead to n-1th element. And for that item also we will make the choice to take it or not.



We start from the end of the array. If we decide to consider the element then value of the last index will be val[n-1]

Weight on the last index will be wt[n-1]. So if you calculate the profit then profit will be val[n-1] + profit from the rest of the item. To find the profit from the rest of the item we need to calculate it recursively. Once we select the item then wt for remaining item will also decrease to wt-wt[n-] as we already have selected wt[n-1]. So overall profit will be

value[n - 1] + solve(wt, value, w - wt[n - 1], n - 1)

If we decide to not consider the element then solve(wt, value, w, n - 1)

And we want the max profit from these two. So over all profit for the scenario w1<= W will be

Math.*max*(value[n - 1] + solve(wt, value, w - wt[n - 1], n - 1), solve(wt, value, w, n - 1));

And profit for the scenario w1 > W will be

solve(wt, value, w, n - 1)

if we combine these steps over all code will look like

<https://github.com/hareramcse/Datastructure/blob/master/DP/src/com/hs/dp/knapsack/zeroone/Knapsack.java>

1. Knapsack With Memoization.

**int**[] wt = **new** **int**[] { 5, 4, 6, 3 };

**int**[] val = **new** **int**[] { 10, 40, 30, 50 };

**int** w = 10;

We saw in the knapsack problem that only w and n is changing in every recursive call. So we will take a 2d matrix like this int *dp[][]* = **new** **int**[n + 1][w + 1];

Here n is the length of the array and w is the weight of the knapsack.

Here we will initialize the matrix with -1

Before doing recursive call we will check this matrix and see if its value is not -1 then it means we have some value for this call and we will return the same.

**if** (*dp*[n][w] != -1)

**return** *dp*[n][w];

If we don’t have any value means value of this grid is -1 then it means we don’t have any value then we will do the recursive call and before returning the value we will store this value to matrix so that if we do the same recursive call in future we will get the value from the matrix itself and no need to do the recursive call.

*dp*[n][w] = solve(wt, val, w, n - 1);

and we will return the matrix *dp*[n][w] as we will have the result on n,w index in the grid.

<https://github.com/hareramcse/Datastructure/blob/master/DP/src/com/hs/dp/knapsack/zeroone/KnapsackMemoization.java>

1. **Knapsack Bottom up approach.**

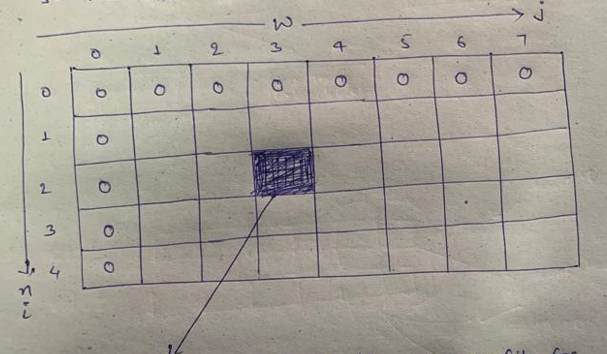
**int**[] wt = **new** **int**[] { 1, 3, 4, 5 };

**int**[] val = **new** **int**[] { 1, 4, 5, 7};

**int** W = 7;

There is two steps in this approach

1. Initialization
2. Function Call



Here we have array length n = 4 and capacity of the knapsack is 7

So we will make the matrix of dp[n+1][w+1] which is dp[5][8]

Value in the painted grid shows that it is the maximum value for first 2 element of the input array and knapsack capacity 3.

So for our problem dp[n][w] will give the maximum profit.

In the recursive function we have base condition…..in this approach this base condition changes into initialization of the matrix.

**if** (n == 0 || w == 0)

**return** 0;

Here we see that n == 0 or w == 0 then we are returning 0

In our tabular approach we will do the same thing and first row and first column will be initialized by 0

as shown in the above pic. If in the base condition we are returning -1 then here also we would initialize

-1. It totally depends on what we are returning in the recursive approach.

In this code n - > i and w - > j and solve function is changed with dp name that’s it.

<https://github.com/hareramcse/Datastructure/blob/master/DP/src/com/hs/dp/knapsack/zeroone/KnapsackBottomUp.java>

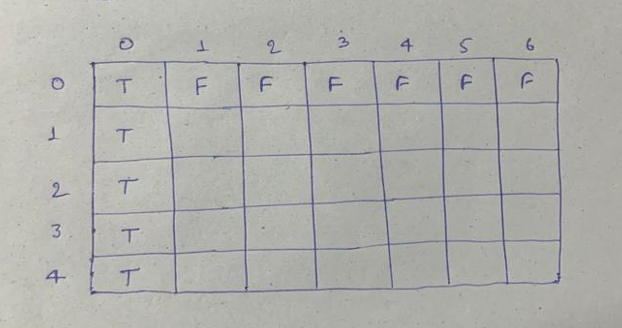
In next problem somewhere if there is only one array given then we will consider this is as wt[] only. val[] we will discard in this case.

1. **Subset sum problem**

**int**[] arr = **new** **int**[] { 1, 3, 5, 7, 8, 9 };

**int** sum = 4;

**Here we have to find out if there is any subset whose sum == 4**



**Initialization:**

Here (0, 0) mean we have array with 0 length and we need sum 0. It is possible we can take empty set {}

(1, 0) means we have array of length 1 and we need sum 0. It is possible we can take empty set {}

(2, 0) means we have array of length 2 and we need sum 0. It is possible we can take empty set {}

Similarly for (3, 0) , (4, 0)

For (0, 1) means we have array of length 0 and we need sum 1. It is not possible

For (0, 2) means we have array of length 0 and we need sum 2. It is not possible

Similarly, for (0, 3), (0, 4), (0, 5), (0, 6)

If sum is possible we will initialize with True and if it is not possible we will initialize with False

Problem where we were finding the maximum profits there we are using the max of 2 numbers.

But here we are storing Boolean value in the matrix. So we will use || operator instead of max.

<https://github.com/hareramcse/Datastructure/blob/master/DP/src/com/hs/dp/knapsack/zeroone/SubsetSum.java>

1. **Equal sum partition problem**

**int**[] arr = **new** **int**[] { 5, 4, 6, 3 };

We have to find out it is possible to divide the array in 2 half so that its one half sum is equal to another half sum.

Algo:

Just find out the sum of whole array, lets say sum = x. now we will see if x%2 == 0 it means we can divide the number in two half. If it not even number then we cant have 2 subset with equal sum.

Then call the subset problem with array and sum x/2.

<https://github.com/hareramcse/Datastructure/blob/master/DP/src/com/hs/dp/knapsack/zeroone/EqualSumSubset.java>

1. **Subset sum count problem**

**int**[] arr = **new** **int**[] { 5, 4, 6, 3 };

**int** sum = 10;

Here we have to find out the count of the subset whose sum is 10.

Algo:

Same as the subset sum problem. Instead of true/false we need to initialize with 0/1 and instead of ||

Operator we need to use +

<https://github.com/hareramcse/Datastructure/blob/master/DP/src/com/hs/dp/knapsack/zeroone/SubsetSumCount.java>

1. **Minimum Subset sum difference**

**int**[] arr = { 4, 5, 2, 7, 1, 3 };

Here we have to find out 2 subset whose difference is minimum.

Algo:

This problem is similar to partition problem. In partition problem we find out difference of 2 subset is equal to 0. Here we have to find out difference of two subset is minimum.

Here how we will find out two subset s1, s2 ?

Idea is we will use….

We will find the range of the 2 subset. What will be the range ?

Here first we will find out the sum of all the array elements. So we can take this as s1. And other subset we can take as 0.

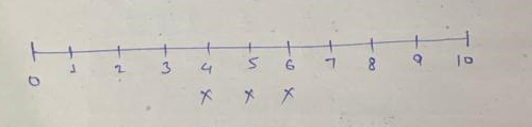
So here range is Σarr[i] and 0

Now in this range there will be all the elements from 1 to n. And there will be some of the elements whose sum will not be equal to Σarr[i]. so we will discard those elements.

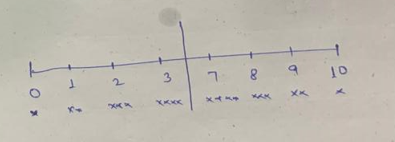
Eg:

**int**[] arr = { 1, 2, 7};

here sum of all the elements = 10



Here we have all the elements from 0 to 10. We can see that we cant have sum of 4, 5, 6 with our array elements {1, 2, 7}. So we will discard it from the subset.



Now we have this many elements which can be added to each other to get sum 10.

Here we can have 0 in one set and 10 in another set.

1 in one set and 9 in another set

2 in one set and 8 in another set

3 in one set and 7 in another set

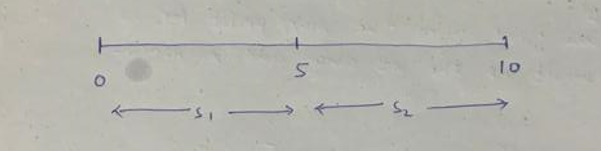
Here we have to find out min(S2 – S1)

Here if we find S1 then we can find S2 as Range – S1

So we don’t need to find S2 explicitly. If we are able to find out S1 then we can find out S2 as Range – S1

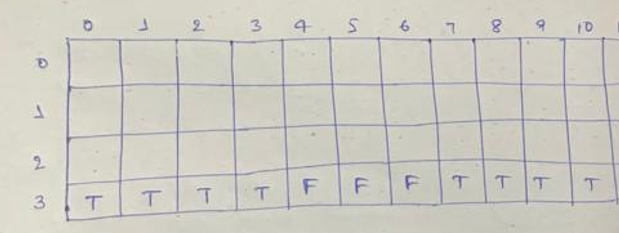
We have to find out min(S2 – S1) means min(Range – S1 – S2)

Means we have to find out min(Range – 2S1)



Here we have to take the abs of S2 – S1 so we can take the middle element of the Range as pivot.

Before middle element whatever element are there will consider for the S1 subset. So that it will never give S2 – S1 negative value. As S2 elements will always be higher.



Here in this we need to take only those values whose value is true and only in first half of the range.

<https://github.com/hareramcse/Datastructure/blob/master/DP/src/com/hs/dp/knapsack/zeroone/MinimumSubsetSumDifference.java>