0-1 Knapsack

- Sushen Patidar

- Concept Explanation.

1) Subset Sum
2) Equal Sum Partition
3) Count of Subset Sum
4) Minimum Subset diff
5) Target Sum
6) No. of Subset with a given diff.

Dynamic Programming (Aditya Verma)

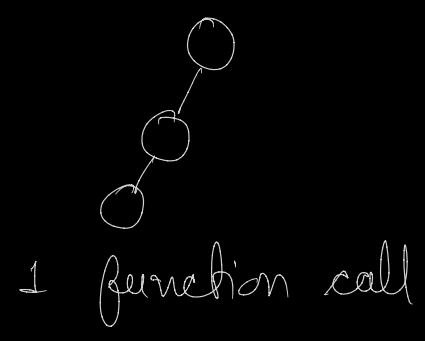
DP :- enhanced recursion

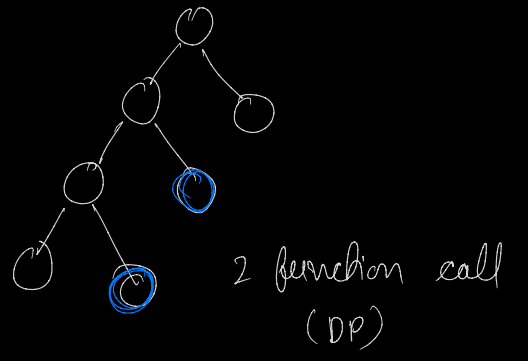
- Sushen Patidar

Q => How to identify DP Problem ?

1) Where there is recursion, DP is used
(for overlapping Problem)

a) Choice





b) Optimal :-

min, max, largest.

- Q => How to write DP code ?

Steps :-

1) Recursive Solution
2) Memoization
3) Top-down approach Solution.

=> Questions on DP.

1) 0-1 knapsack (6)
2) Unbounded knapsack (5)
3) Fibonacci (7)
4) LCS (15) (longest common Subsequence)
5) LIS (10) (longest increasing Subsequence)
6) Kadanis Algo. (6)
7) Matrix chain Multiplication (7)
8) DP on Tree (4)
9) DP on grid (4)
10) DThess (5)

Knapsack Problems

1) Subset Sum
2) Equal Sum Partition
3) Count of Subset Sum
4) Minimum Subset diff
5) Target Sum
6) No. of Subset with a given diff.

Knapsack Problem

- Fractional Knapsack
- 0-1 Knapsack
- Unbounded knapsack

⇒ 0-1 Knapsack ⇒

We are given some weight & some value. Then a max weight W. we can pick to maximize our profit

→ Can not pick one item more than 1 time

the weight has a given bound W., either we can pick that 2kg  brick if not then we have to exclude it

→ 10₹

Brick

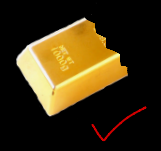
W = 7 kg (Capacity)

	I_1	I_2	I_3	I_4
Wt[] :	1	3	4	5
Val[] :	1	4	5	7

Max. Profit?

⇒ Fractional knapsack

⇒ if we can not pick 2kg  ✗ Brick

but can only pick 1kg more than we can take fraction of 2kg brick. 

greedy approach.

⇒ Unbounded knapsack

Can Pick one item Multiple times.

→ 0-1 knapsack

Q⇒ how to identify ?

1) Choice
   1) either put in knapsack
   2) either don't Put in knapsack

2) Optimal (max. Profit) (minimum, highest, lowest)

3

→ DP :- Recursive → Memoization → Top down (dp) (dp)

DP ⇒ Recursion + Storage

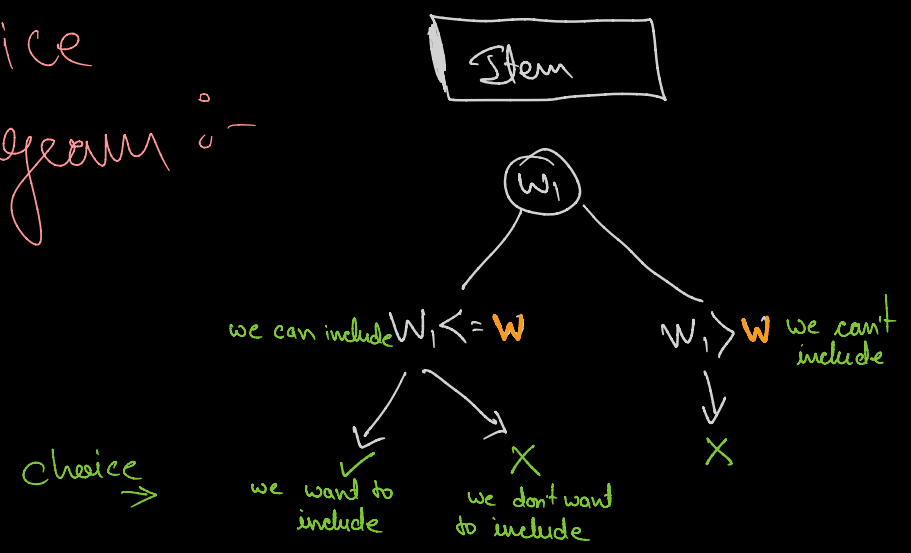
I/p Wt[] = | 1 | 3 | 4 | 5 |

Val[] = | 1 | 4 | 5 | 7 |

capacity  W = 7 kg

O/p ⇒ Max. Profit

Choice diagram :-



W = capacity of bag.

W_1 = weight of Brick

Becoz - we cannot fill 5l water in a bottle which has 4l capacity

NOTE :- we have to return the Max Profit
(So return type would be int.
can be bool... etc. Acc. to question

Base Condition :

Think of the smallest Possible Valid I/P.

I/P = wt: [ ] n → 0
val: [ ]
w: 10kg → 0kg.

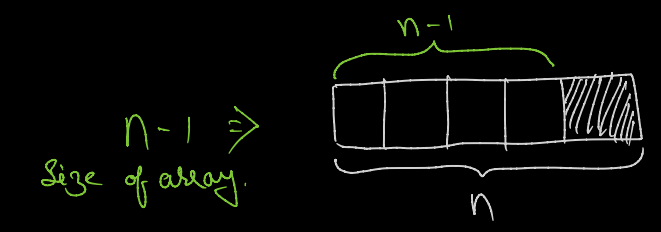
```
int knapsack (int wt[], int val[], int W, int n)
{
  // Base Condition
  if (n == 0 || W == 0)        (Base Cond^n can be different)
    return 0;
```

W = current capacity of knapsack

wt[n-1] = weight of current Brick, we want to Pick

val[n-1] = Price of Brick wt[n-1].

n-1 ⇒ Size of array.

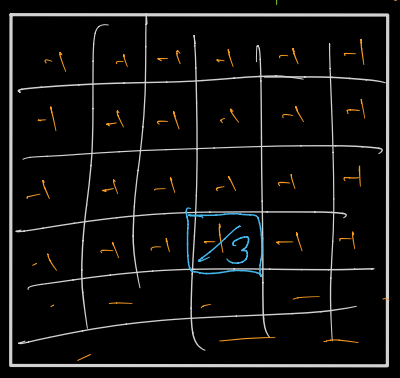



```
  // Choice diagram
  if (wt[n-1] <= W){
    return max( val[n-1] + knapsack(wt, val, W - wt[n-1], n-1),
                knapsack(wt, val, W, n-1));
                (dont want to include weight wt[n-1])
  }else if (wt[n-1] > W)
    { return knapsack(wt, val, W, n-1);
    }
}
```

0-1 Knapsack Memoization

Memoization = Recursion + (2/4 lines)

$t_{n \times m}$ → $n * W$



fun (I/P doesn't change / I/P Changes) (n, w)

Knapsack (wt[], val[], n, W)

n-1 , W - wt(n-1)

NOTE:-

⇒ Initialise this matrix with -1.

⇒ -1 3 if value is not -1 than value for that position is already calculated and stored so return it.

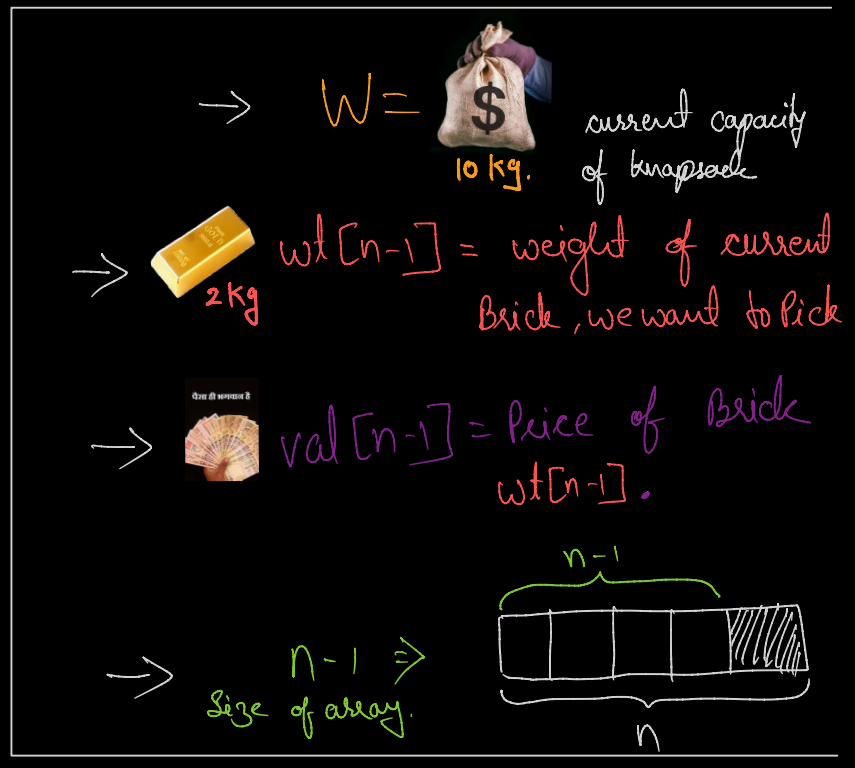
Changes

-> declare the matrix globally with value '-1'.

int static t[102][1002] (n, w) constraints: n <= 100, w <= 1000

```
int knapsack (int wt[], int val[], int w, int n)
{
  // Base Condition
  if (n==0 || w==0)
    return 0;

  if (t[n][w] != -1)
    return t[n][w];
```



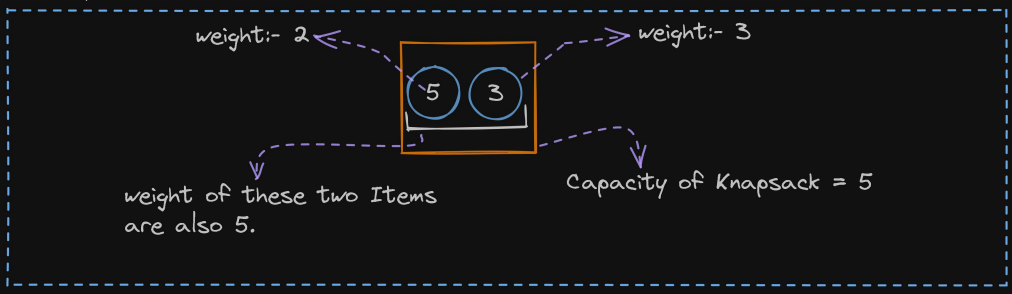

```
  if (wt[n-1] <= w){
    return t[n][w] = max( val[n-1] + knasack (wt, val, W - wt[n-1], n-1);
                         knapsack (wt, val, W, n-1);
                         (dont want to include weight wt[n-1])
  }else if (wt[n-1] > W)
    { return t[n][w] = knapsack (wt, val, W, n-1);
    }
}
```

0-1 Knapsack

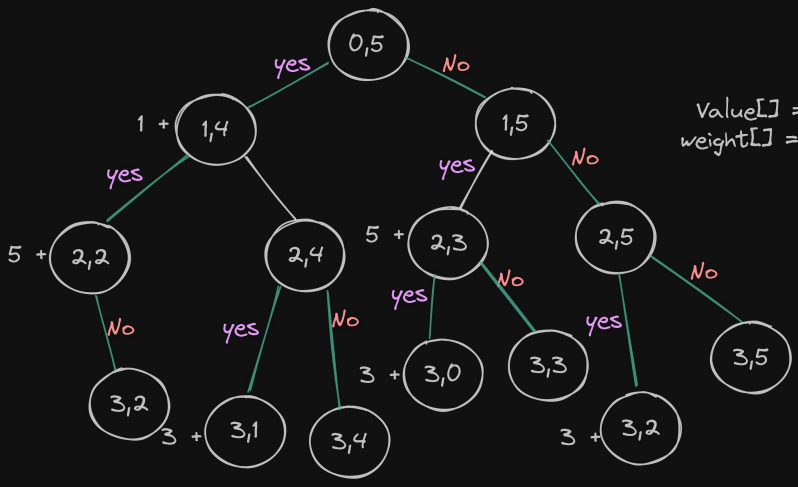
creater of this page - Unknown
Source - linkedin

We Have to create max. profit from this Knapsack with capacity 5.

So, from the above diagram we can say that the max. profit is 8,
How?



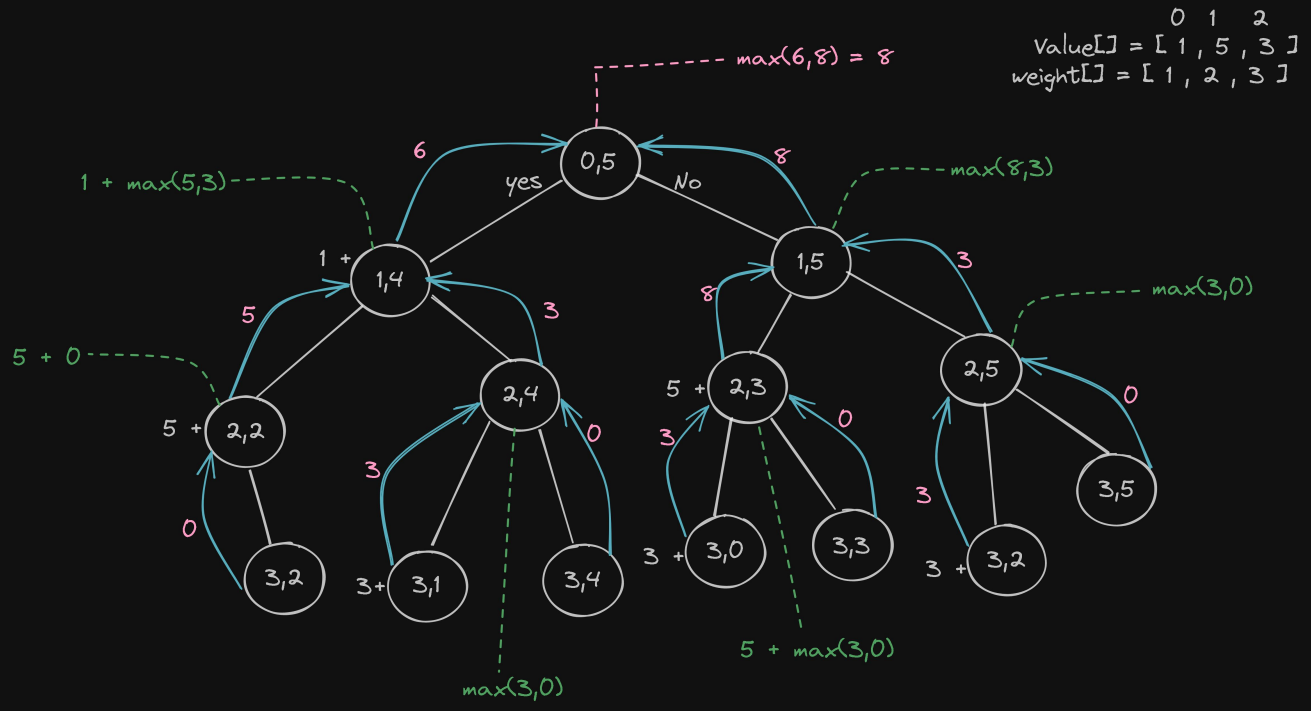
we have two option weather to take element in knapsack or not



0 1 2
Value[] = [1 , 5 , 3]
weight[] = [1 , 2 , 3]

* n = 0
* capacity = 5

Note:- when ever n > value.length return 0,
when ever capacity become -negative don't create that branch.



The Complexity of Top-down & Memoization Remains same but the Problem with memoization is the Stack gets full due to Repeated fun Calls.

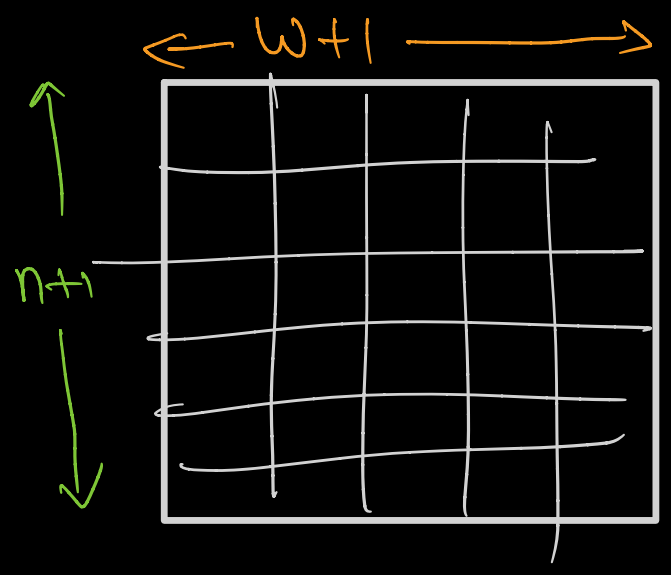
(5) (0-1 Top-down knapsack)

Recursive ⇒ Memoize ⇒ Top-down ⇒ 6 Problem

↓ BC + Recursive Calls

↓ R Calls + table

↓ only Table



⇒ only make table for the I/p which is changing with R. Calls.

⇒ 2 Steps to make table for Top-down

Step 1:- Initialization

Step 2:- Recursive code changes Iterative Code

Step 1 - Initialize

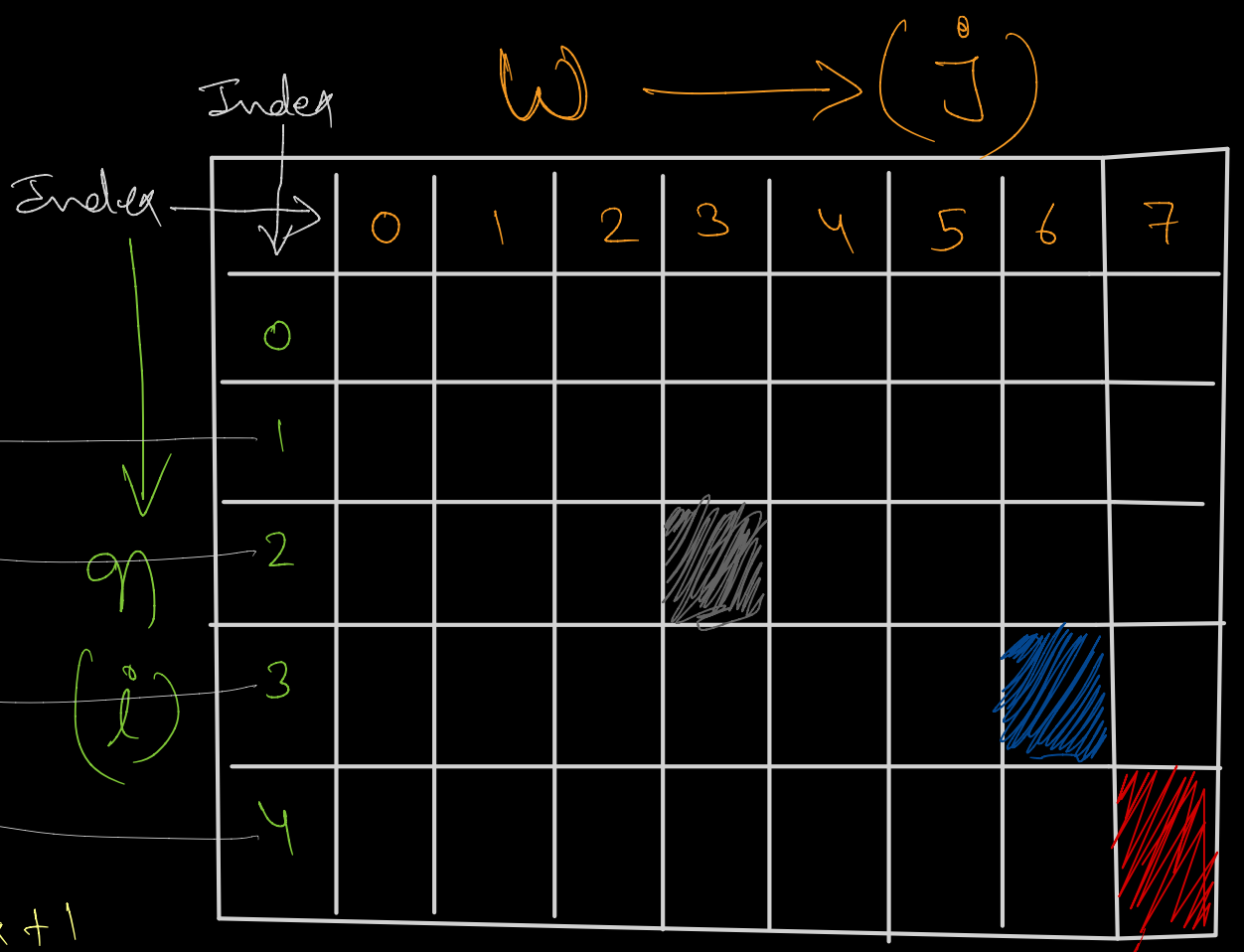
W = 7 

n = 4

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

Val[] = [1 4 5 7]

Val[]	wt[]	Index
1	1	0
4	3	1
5	4	2
7	5	3



n = index + 1
or
index = n - 1

wt[] : [1 3 4 5]

val[] : [1 4 5 7]

t[n][w] will give final Answer

wt[] = [1 3]

val[] = [1 4]

W = 3

n = 1 + 1 = 2

wt[] = [1 3 4]

Val[] = [1 4 5]

W = 6

n = 2 + 1 = 3

t(n, w) =

→ RC + table = table

Base Cond" = Initialization.

R.C. code

```
if (n==0 || w==0)
    return 0;
```

NOTE :- Base condition of Recursive Call is Converted into Initialization in Top-Down

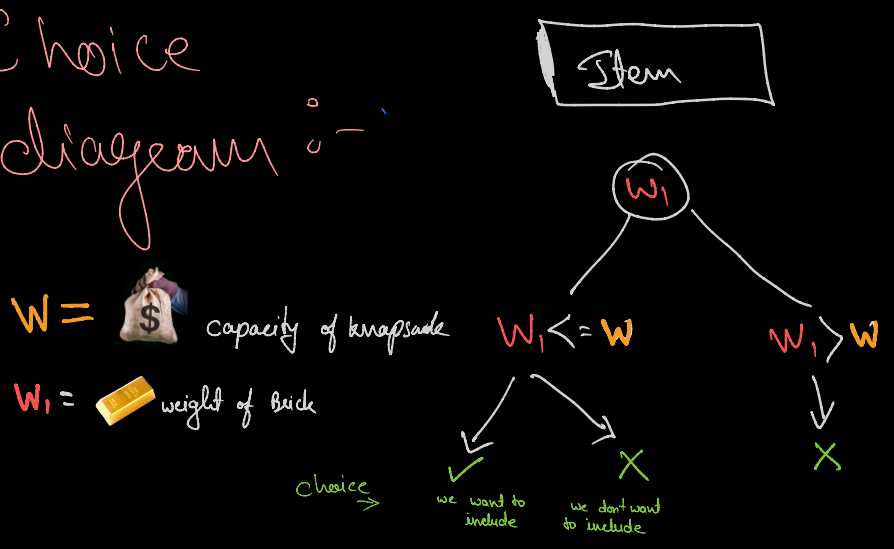
for Table

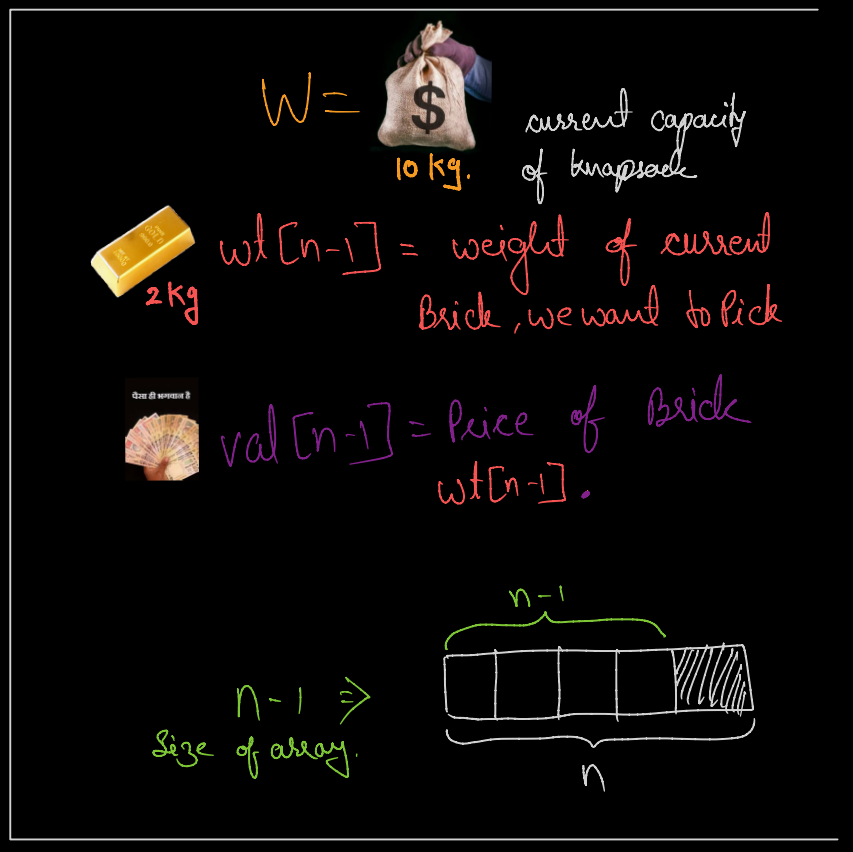
NOTE :- n ⇒ i , w ⇒ j

```
for(int i=0; i<n+1; i++)
{
    for(int j=0; j<w+1; j++)
    {
        if(i==0 || j==0)
            t[i][j] = 0;
    }
}
```

n \ w	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				

Choice diagram :-





Recursive

```
if (wt[n-1] <= w)
    return max = (val[n-1] +
        knapsack(wt, val, w-wt[n-1], n-1),
        knapsack(wt, val, w, n-1));

else if (wt[n-1] > w)
    return knapsack(wt, val, w, n-1)
```

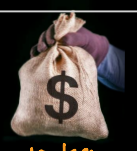
Top-down

```
if (wt[n-1] <= w)
    t[n][w] = max(val[n-1] + t[n-1][w-wt[n-1]],
                  t[n-1][w]);

else
    t[n][w] = t[n-1][w]
```

Top - down Approach

```
int t[n+1][w+1];
for (int i=1; i<n+1; i++)
{  for(int j=1; j<w+1; j++)
   {
      if( wt[i-1] <= j)
      {
         t[i][j] = max(val[i-1] + t[i-1][j - wt[i-1]],
                       t[i-1][j]);
      }
      else { t[i][j] = t[i-1][j]; }
   }
}
return t[n][w]; ⇒ Final Answer.
```

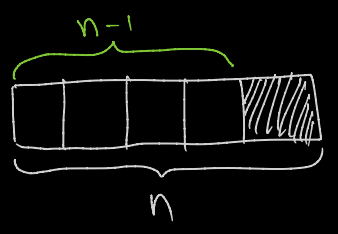
W = 

current capacity of knapsack

wt[n-1] = weight of current Brick, we want to Pick


val[n-1] = Peice of Brick wt[n-1].

n-1 ⇒ Size of array.



Knapsack Problems.

1) Subset Sum
2) Equal Sum Partition
3) Count of Subset Sum
4) Minimum Subset diff
5) Target Sum
6) No. of Subset with a given diff.

→ Ex. Maximum Profit.

while learning knapsack.

employee in Service Base Company. ex. TCS, wipro, infosys .. etc.

After learning knapsack


Getting Offer letter from Product Based Company

→ So we should learn DP (knapsack .. etc.) to get maximum Profit.

(1)

Q 1] :- Find if there is a subset present in an array with given sum?

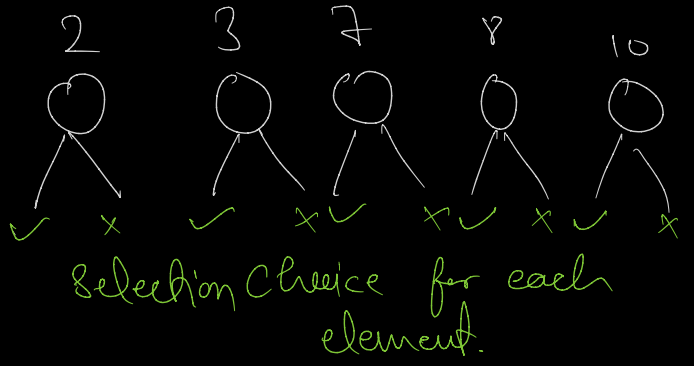
arr [ ] : 2 3 7 8 10

Sum : 11

- Similarity :-

item array[ ] : → 2 3 7 8 10

weight : Capacity : W :- 11



→ Code Variation

t [n+1] [w+1]

t [5+1] [11+1]

t [6] [12]

Initialization :-

w →

	0	1	2	3	4	5	6	7	8	9	10	11
0	T	F	F	F	F	F	F	F	F	F	F	F
1	T											
2	T											
3	T											
4	T											
5	T											

n ↓

O/P : True / False

(SUM) w :- can not be > 0 when size of array is 0.

t [n+1] [SUM+1]

```
for ( i=1  to n+1)
    for (j=1  to w+1)
        if (i==0)
            t[i][j] = False
        if (j==0)
            t[i][j] = True
```

Knapsack

```
if(wt[i-1] <= j){
  t[i][j] = max(val[i-1]+ t[i-1][j-wt[i-1]] ,
                t[i-1][j])
}else{
  t[i][j] = t[i-1][j];
}
```

SubSet Sum

```
if(arr[i-1] <= j){
  t[i][j] = t[i-1][j- arr[i-1]] ||        ← OR operator
                t[i-1][j])
}else{
  t[i][j] = t[i-1][j];
}
return t[n][sum];   Return type bool. T/F
```

2

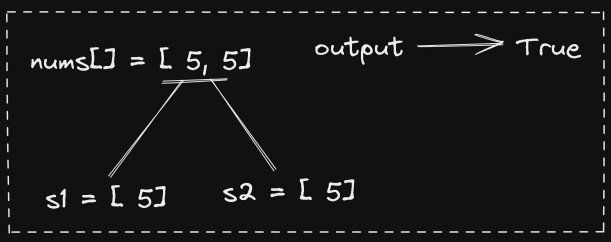
Creator of This Page - Unknown
Source :- LinkedIn

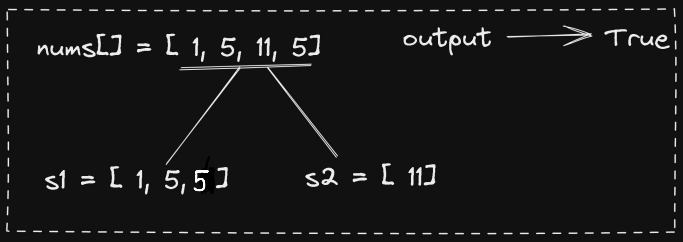
Partition Equal Subset Sum !-

Q Is it possible to divide the array such that both the subset gives an equal sum.

1st check :- check if its sum is odd , then return false.
2nd check: - if sum is even then make recursive call.

So, if given array sum is 10.
s1 + s2 = sum(nums);





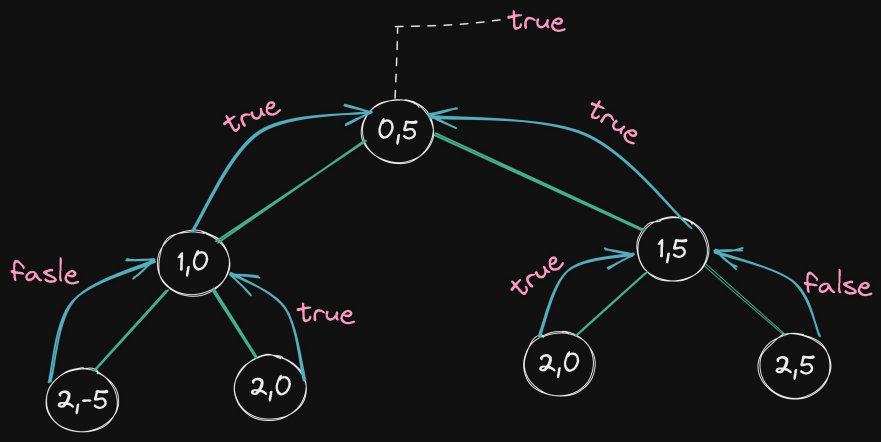

```
if (sum(nums) % 2 != 0)
    return false;
else
    return SubsetSum(arr, sum/2);
    (Previous question)
```

sum(s1) + sum(s2) = sum(nums)

sum(s1) + sum(s1) = sum = (nums)

2sum(s1) = sum(nums)
sum(s1) = sum(nums)/2;

nums[] = [5, 5]
0 1



NOTE:- If our left half give's us "true" , then just "return true" from there no need to check for the right half then.

Like, if we get 5 from left half , thats means that we diffenately going to get 5 from other half as well.

"All DP problem can be solve using backtracking. But, we cannot solve backtracking problem using DP. "

And Note that Knapsack is Only Valid for positive input.

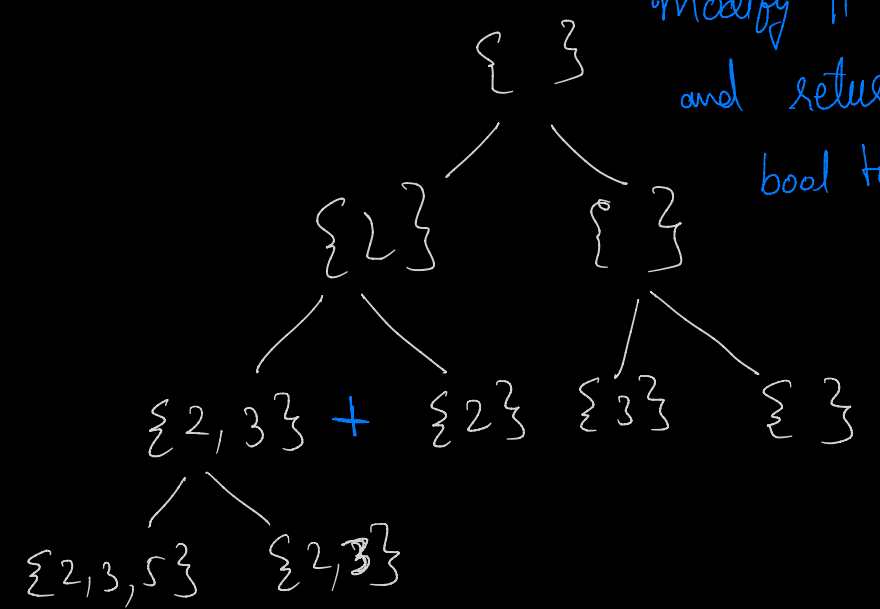
③

arr[] : 2 3 5 6 8 10

Sum : 10

{2,8}
{2,3,5}
{2,8}
— O/P 3

⇒ Same like Subset Sum
modify '||' with +
and return type
bool to int



Code :- if (arr [i-1] [j] <= j)

dp [i] [j] = dp [i-1] [j] + dp [i-1] [j - arr [i-1]]

else

dp [i] [j] = dp [i-1] [j]

⑨

Q⇒ Minimum Subset Sum difference

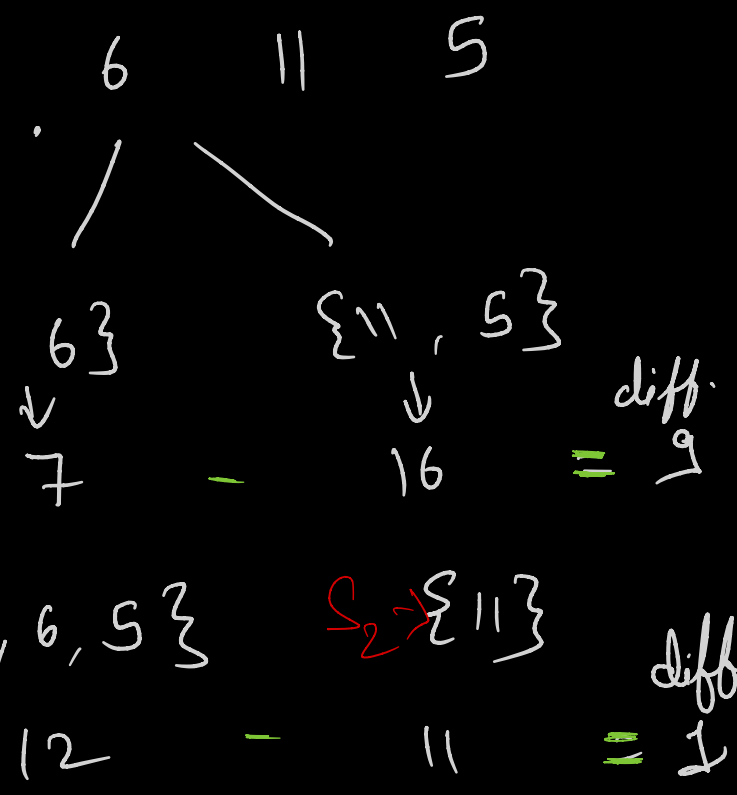
(Based on equal Sum partition)

arr[] : {1, 6, 11, 5}

o/p : 1

if equal Sum → $S_1 - S_2 = 0$

Minimum diff → abs($S_1 - S_2$) = should be minimum



→ It is similar to equal Sum Partition

=> we need to find S_1 and S_2, arr[] = {1, 2, 7}

Diff =>

=> Range = Sum(arr)

$S_2 - S_1 \Rightarrow ((Range - S_1) - S_1)$ — eq. (A)

$\Rightarrow (Range - 2S_1) \geq 0$

=> So we need to find the SUM of all Possible Subset. thn we use it to Calculate min diff.

=> Using Subset Sum Problem.

	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4	T	T	T	T	F	F	F	T	T	T	T

=> here, we are getting a Table which has Info. of all the possible Combination of Sum of Subsets in Last Row

=> So we get {0, 1, 2, 3} possible value of S_1, So we can calculate minimum diff using equation - (A)

S_2 will be in other Right half of Last Row in table

$S_1 = 3$, $S_2 = 10 - 3 = 7$

min. diff = $S_2 - S_1 = 7 - 3 = 4$ Ans

CODE

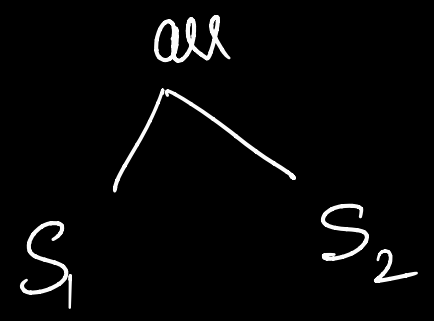
```
int min_Diff(int arr[], int n)
{
  int sum=0;
  for(int i=0;i<n;i++){
    sum+=arr[i];
  }

  bool dp[n+1][sum+1]; // dp table
  // Base Condition
  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;
    }
  }
  //Filling the values in table
  for( int i=1;i<n+1; i++){
    for(int j=1; j<sum+1;j++){
      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];
      }
    }
  }
  int diff=INT_MAX;
  for(int j=sum/2 ; j>=0;j--){
    if(dp[n][j] == true){
      diff= min( sum - 2*j , diff);
    }
  }
  return diff;// Final answer
}
```

(5) Q => Count the no. of Subset with a given diff.

arr[]: | 1 | 1 | 2 | 3 |

Diff : 1



Sum(S1) - Sum(S2) = diff — (1)

Sum(S1) + Sum(S2) = Sum(arr) — (2)

(1) + (2)

2 Sum(S1) = diff + Sum(arr)

$$Sum(S_1) = \frac{diff + Sum(arr)}{2}$$

$$Sum(S_1) = \frac{1 + 7}{2} = 4$$

=> So now we just need to count the no. of possible Subset with Sum, that will Satisfy the Condition of diff = 1.

=> int Sum = $\frac{diff + Sum\ of\ array}{2}$

return countofSubsetSum(arr, Sum);

Previous Problem.

CREATED By- Unknown

6

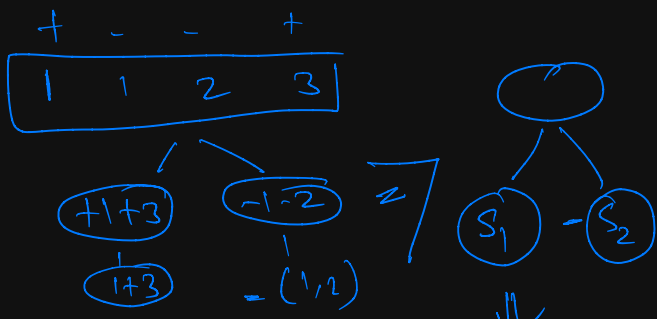
You are given an integer array `nums` and an integer `target`.

You want to build an expression out of nums by adding one of the symbols `'+'` and `'-'` before each integer in nums and then concatenate all the integers.

- For example, if `nums = [2, 1]`, you can add a `'+'` before `2` and a `'-'` before `1` and concatenate them to build the expression `"+2-1"`.

Return the number of different expressions that you can build, which evaluates to `target`.

Leetcode 494. Target Sum



nums[] = [1, 1, 1]

target = 1

-1 + 1 + 1 = 1
1 - 1 + 1 = 1
1 + 1 - 1 = 1

→ Output: 3

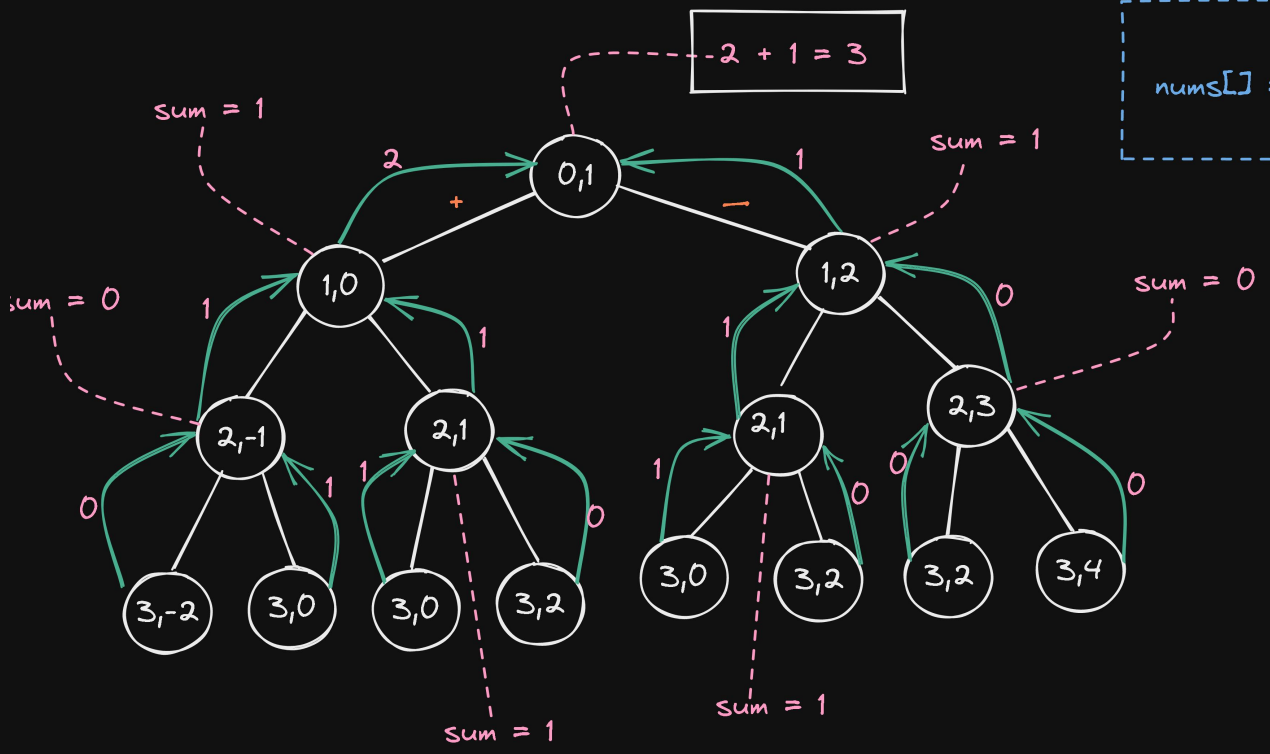
0 1 2
nums[] = [1, 1, 1]

TargetSum

↓

Count of Subset with a given difference.

→here SUM given is equal to diff. given in previous problem (5)



0 1 2
nums[] = [1, 1, 1]

Left branch = n+1 , target - nums[n]

Right branch = n+1, target - (-nums[n])

n+1 , target + nums[n]