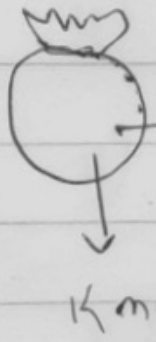
Dynamic Programming

DP -> Enhanced Recursion

↳ Stores overlapping problem.

Knapsack Problem



Fractional Knapsack | 0/1 | Unbounded

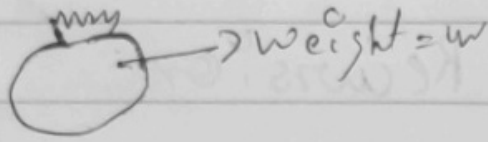
Can take values of items b/w 0 & 1

Take full item or leave it

Take as many as you want of one item

W.I ->	2	4	3	8
Profit ->	8	6	10	20

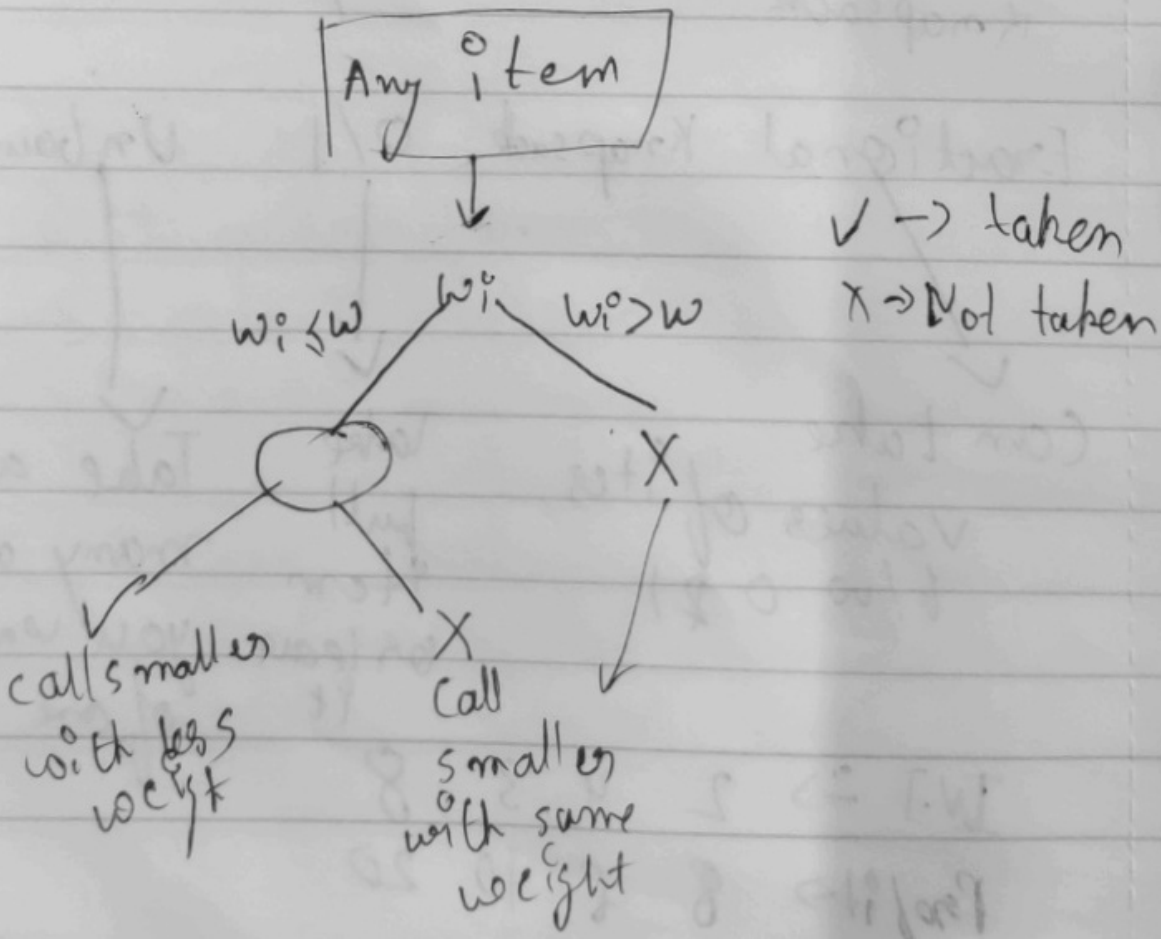
→ 0/1 Knap sack



A →	5	6	2	4	8
P →	10	11	13	20	14

Base case

```
if (W == 0 || n == 0)
{
    return 0;
}
```



Problems that are similar to 0/1 Knapsack problem

1) Subset Sum problem

2) Equal sum partition

3) Count of subset sum

4) Maximum Subset Sum diff

5) Target Sum

6) # of subset = given diff

→ Subset Sum problem

Is it possible to find a subarray of a given array whose sum is equal to the given value

2 3 7 8 10

Sum = 11

Answer in Yes/No

Simpley Knapsack

↳ But in Yes/No

Go on accepting or rejecting an element and decreasing the size of the array.

→ Equal sum partition problem

Given an array, one has to find if a partition is possible in the array, such that the sum of two partions is equal

arr = { 1, 5, 11, 5 }

↳ Yes → { 1, 5, 5 }, { 11 }

→ Output in Yes/No

→ Either put an element in the left set or don't put it.

Observation for improvement

→ if sum = odd → Never possible

→ The sum of the final partition will always be sum/2

→ Count of Subset Sum

Given an array, and a sum, find the number of subsets of that array.

Note, one subset is indepent of the other.

arr = 2 3 5 6 8 10

sum = 10

→ Ans = 3 → {10}, {2, 8}, {2,3,5}

Sol^n Same as subset sum, take a number and dont take a number.
But instead of true or false, return the number of subsets by taking or discarding that number.

→ 1D array optimization

Instead of 2D array with dimensions $W \times n$
(W → weight, n → number of elements)

We can use an array of size W.

For Knapsack 0/1

→ What we do

~~if dp[n][sum] = dp[n-1][sum], dp[n-1][sum]~~

dp[n][sum] = max(dp[n-1][sum], dp[n-1][sum - a[n-1]])

→ What can we do.

```
profit[W+1]
profit[0] = 0
for(i = 0 to n-1)
    for(j = W to a[i], j--)
        ~~profit[j] = profit max(profit[j], profit[j-a[i]])~~
        profit[j] = max(profit[j], profit[j - a[i]])
```

Optimization for Subset Sum

What we do

dp[n][sum] = dp[n-1][sum] || dp[n-1][sum - arr[n-1]]

What can we do

for(int i = n-1; i >= 0; i--)
  for(int j = sum; j >= arr[i]; j--)

    possible[j] = possible[j] || possible[j-arr[i]]

Similar optimizations can be Equal sum partition problem & Count of subset sum problem.

→ There is also a bitset optimization for return type 0 and 1

```
bitset <Max> possible -> global

Subset sum
{
  possible = 0
  possible[0] = 1;

  for (i = 0 : to n-1)
  { for (j = sum to a[i]; j--)
    {
      possible[j] = possible[j] || p[j-a[i]]
    }
  }
  return possible[sum]
}
```

→ Minimum Subset sum diff.

Given an array of numbers, you have to divide the array into subsets such that difference b/w them is minimum.

arr → 5, 7, 6, 5

Answer →

Take subsets: {6, 6}, {7, 5}
11 12 → 1

Soln:-

Method 1:- Use ts, if n = 0
ans = min(ans, (sum − 2*ts))

Method 2:- Use Subset sum to calculate the possible values for all when all integers are taken into consideration.
Iterate over these values and find ans.

→ Count the number of subset with given diff.

arr → | 1 | 1 | 2 | 3 |

diff = 1

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

Method

count = sum[s1] = $\frac{diff + sum(arr)}{2}$

↳ Subset sum

→ Target Sum

Given arr, assign either + or - sign to an element, then add all the elements. Find the result of all such assignments and return the ~~sum~~ number of all elements whose sum is the given target.

→ Same as previous problem. count the number of subsets with given diff.

-> Unbounded Knapsack

The difference b/w 0/1 and unbounded Knapsack is that you can take multiple occurences of a sigle item in unbounded Knapsack, where as only one occurence of an item is allowed in 0/1 Knapsack.

-> As you can take multiple occurences of a single element, you only stop considering a element, when you don't want it any more.

ans = func (val[n-1] + func(n), func(n-1))

Problems for unbounded Knapsack.

-> Rod cutting

-> Coin change - I

-> Coin change - II

-> Maximum Ribbon cut

→ Rod cutting

Given a peice of rod and prices of a particular peice, find the optimal way to cut a rod obtaining maximum profit.

→ Just a unbounded knapsack

→ Coin change - I

Given some coins, find the # of ways, it is possible to make a particular sum,

→ Unbounded knapsack on number of subset sone

→ Coin change - II

Coin [] : | 1 | 2 | 3 |

Sum = 5 Ans : 2 → 2+3=5

Find the minimum number of coins required to make up the given sum. Return -1 if not possible

→ solution: Do a minimum subset sum on the array with unbounded condition.

→ Longest common subsequence

1) Longest common Substring
2) Print LCS
3) Shortest common Superseq
4) Print SCS
5) Minimum number of insertion/deletion to a→b
6) Longest Repeating Subseq
7) Count of longest subseq of a which is a substring of b
8) Subsequence Pattern matching
9) Count how many time a appear as subseq in b.
10) Longest palindromic subseq
11) Longest Palindromic substring
12) Count of palindromic substring
~~13) Longest palindromic~~
13) Min number of deletion to make a palindrome
14) Minimum number of insertion to make a palindrome.

→ Longest common Subsequence.

Given two strings x, and y we need to find a subsequence of greatest length that is common in both

x = (a) (b) c (d) g (h)

y = (a) (b) e (d) f (h) r

Length of LCS = 4 → Ans

If an element is same consider it both strings if not, consider both string length reduced by one

→ Longest common Substring (LCSS)

Given two strings, find the length of the longest common substring.

→ In this question you need to find the Longest common substring, so you dont know if a previous problem is in with continuation with the current subproblem.

You can pass count, but its memoization will require a 3D DP.

What is possible is, you can store the length of the LCSS at ending at this character.
1D array optimization is also possible in this question.

→ Print Longest common subseq

Given two strings find the subsequence of largest length which is common in both, Print that subsequence.

GO in reverse manner, go from below, i/ s1(i-1) = s2(j-1) print and go across. otherwise go left or write dependion upon which gives larger subseq.

→ Shortest common Supersequence

Given two strings, find a string such that given two strings are subseq of the 5 string.

s = geek
t = eke

ans = geeke → 5

Find the LCS and use it once, Ans = n + m - LCS

→ Minimum number of insertion and deletion

Given two strings, find the minimum number of moves, required to convert string a to string b.

In one move you can insert or delete a characters.

Sol^n

If you want to convert a to b optimally you would remove an element not in b (considering position). Same thing with insertion.

Take LCS keep that as it.

Ans = $n + m - (2 \times LCS)$

→ Longest Palindromic Subsequence.

Given a string, find the longest subsequence of this string which is a palindrome.

a g b c b a → 5

↳ abcba

Solution,

Here, a palindrome is a string, that is same when read forwards and backwards.

So, LPS is the LCS of a and reverse of a

→ Minimum # of deletion in a string to make it a palindrome.

Given a string, find the minimum number of charectors to delete to make it a palindrome.

→ Delete all elements expet from LPS

answer = n - LPS

→ Print Shortest common Superseq

Given two strings a and b, print a string which is the shortest common superseq of both strings.

a: acbcf

b: abcdaf

The solution is same as printing LCS, with just one change.

While printing LCS, if $s_1[i-1] \neq s_2[j-1]$, we move to maximum of dp[i-1][j] and dp[i][j-1], ignoring the other one, here instead of ignoring the other one, we include it, also we take initial non included letters.

→ Longest Repeating Subsequence

Given ~~two~~ a strings a ~~and b~~, find a subsequence that repeats itself, you cannot repeat the same character on both the subsequences.

∴ str = a a b e b c d d

LRS = 3 → a b d

Clarification on the PS:

You cannot take the i^{th} character in both the strings if they give the same index in the repeating subsequence

Solution: LCS of same string with itself, just follow the condition $i \neq j$.

Proof of correctness:

In the question, it is given to us that the i^{th} character can be present in both the strings but not at same subsequence indices.

Now, when we are doing LCS we are always looking for the subseq index.

If $i == j$, and $s[i-1]$ would be equal to $s[j-1]$, this not allowed so we'll only consider two characters when $i \neq j$

Thus

```
if (s[i-1] == s[j-1] && i ≠ j)
{ dp[i][j] = 1 + dp[i-1][j-1]; }
else
{ dp[i][j] = max(dp[i-1][j], dp[i][j-1]); }
```

→ Sequence pattern matching

Given two strings a and b, find if a is a subsequence of b.

a = axy

b = adxcpy Ans = True

Just check if LCS == a.length()

→ Minimum # of insertions in a string to make it a palindrome

- Given a string, find the minimum # of insertions to make it a palindrome.

Solution: same as minimum # of deletions.

→ Matrix Chain Multiplication

Given some matrices, find an optimal order to multiply them, such that gives min cost.

Arr : 40 20 30 10 30

A_1 → 40 × 20
A_2 → 20 × 30
A_3 → 30 × 10
A_4 → 10 × 30

A_i → arr[i-1] * arr[i]

→ In these type of questions you divide the give arr or string and find ans. (Generally)

method of solving : BC
Calculate ans ← divide loop ↵

→ Method of this MCM

Lets 4 Matrix are given

A_1 A_2 A_3 A_4

Operation on these are

$(A_1\ A_2\ A_3\ A_4)$
$(A_1)(A_2\ A_3\ A_4)$
$(A_1\ A_2)(A_3\ A_4)$
$(A_1\ (A_2\ A_3)\ A_4)$ -- and so on.

We will consider each one of them.
We will create a function passing the starting and ending matrix which will further break the problem
(Solved on GFG for refrence)

May be different from Aditya a verma

Recursive code

```
int func(int low, int high; int * a)
{
    if (low >= high) { return 0; }
    if (dp[low][high] != -1) { return dp[low][high]; }
    if (high - low)
    ans = inf;
    for (int i = low; i < high; i++)
    {
        ans = min(ans, (a[low-1] * a[i] * a[high]
                + func(low, i, a) + func(i+1, high, a)
    }
        dp[low][high] =
    return ans;
}
```

Memoization with pencil

→ Palindrome Partitioning

Find minimum number of partition required to make string a palindrome.

Generally, questions of MCM can be solved using the following steps

1) Find boundries (i and j)
2) Find base condition
3) Find partition loop
4) Find ans using these

Here, same is done, a function is called which checks if the given string is a palindrome, and returns 0 if it is. If not, the functi partitions the string further soloing subproblems

May differ from Aditya Verma

Recursive code

```
int func(int low, int high, string s
{
    if (low >= high) { return 0; }
    if (dp[low][high] != -1) { return dp[
    if (checkPalindrome(string, low, high))
    { return 0; }

    int ans = INF;
    for (int i = low; i < high; i++)
    {
        ans = min(ans, 1 + func(low, i, s) +
                           func(i+1, high, s));
    }

    return dp[low][high] = ans;
}
```

Memoized with pencil.

⇒ Evaluate Exp to True

Given a string, consisting of T, F, &, |, ^ find # of ways in which this expression can be evaluated to true.

str: T ^ F & T

Ways to be true: → 2

1) (T ^ F) & T
2) (T) ^ (F & T)

→ In this question, there is a hidden technique. Depending on the operand, the left operator and the right operator either need to be 0 or 1

So, with low and high, we also need to pass what we want (either 0 or 1) to calculate

Code, may differ from A.V.

```
int func(int low, int high, int sign)
{
  if(low > high){return 0;}
  if(low == high){return abs(1-(arr+sign));}
  if(dp != -1){return dp;}
  if(sign == 1)
  {
    for(i = low to high-1)
    {
      ans += operation depending on operator
    }
    return dp=ans;
  }
  else
  {
    for(i = low to high-1)
    {
      ans += operation depending on operator;
    }
    return dp = ans;
  }
}
```

Operation depending on operator.

$

0 → $l_2 * r_2 + l_0 * r_2 + l_2 * r_0$

1 → $l_0 * r_0$

|

0 → $l_2 * r_2$

1 → $l_0 * r_2 + l_2 * r_0 + l_0 * r_0$

^

0 → $l_2 * r_2 + l_0 * r_0$

1 → $l_2 * r_0 + l_0 * r_2$