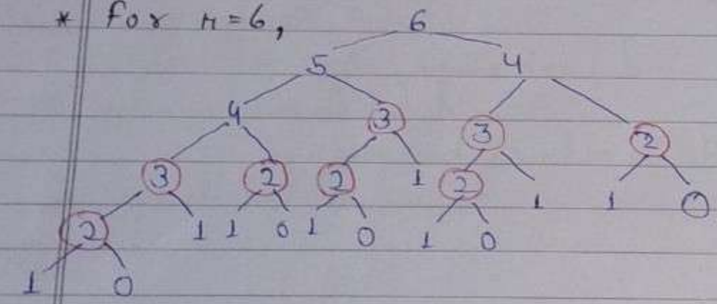
# DYNAMIC PROGRAMMING

★ FIBONACCI – 0 1 1 2 3 5 8 13 21 34.....

⇒
```
public static int fib(int n) {// Exponential Time Complexity
    if (n==0 || n==1) {
        return n;
    }
    return fib(n-1) + fib(n-2);
}
```
~~$O(n^2)$~~ $O(2^n)$

→ fib(n) = fib(n-1) + fib(n-2)

* for n=6,




∵ fib(2) → 5 times repeating

∵ fib(3) → 3 times repeating

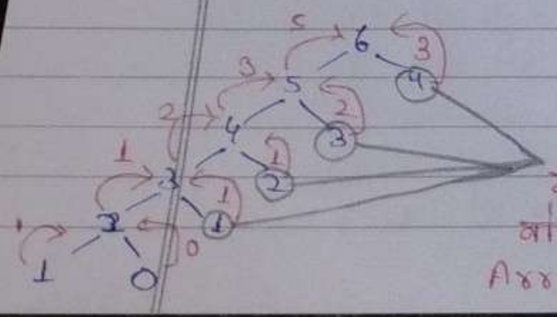
∴ This above code is very inefficient i.e., calculating fib(n) many times of a number, which have been calculated one time.

* We will DP approach.

→ fib(n) = fib(n-1) + fib(n-2)

* for n=6, → f[] =

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 5 | 8 |




इन सबको दुबारा ~~store~~ calculate होने की जरूरत नहीं है क्योंकि वो सब एक बार calculate होके Array में store है।



* What we have done in fibonacci, is known as Dynamic Programming. → We have used memoization approach.

⇒
```
public static int fib(int n, int f[]) {
    if (n==0 || n==1) {
        return n;                    //Linear Time Complexity ⇒ O(n)
    }
    if (f[n] != 0) { //fib(n) is already calculated
        return f[n];
    }
    f[n] = fib(n-1, f) + fib(n-2, f);
    return f[n];
}
public static void main(String args[]) {
    int n = 5;
    int f[] = new int[n+1]; //0, 0, 0, 0, 0, 0
    Sys.o(fib(n, f));
}
```
→ Because we need to store fibonacci values from 0 to n.

* The extra element ensures that the fibonacci value for n is stored at f[n]

★ WHAT IS DYNAMIC PROGRAMMING?

* DP is optimised recursion.
* Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using DP.
* The idea is to simply store the results of subproblems so that we do not have to re-compute them when needed later.

//Definition on Next Page

## ★ HOW TO IDENTIFY DP?

* Optimal Problem: finding best & accurate solution.
* Some Choice is given (multiple branches in recursion tree).
* Overlapping Problems

Definition: "Dynamic Programming is a technique in computer ~~organis~~programming that helps to efficiently solve a class of problems that have overlapping subproblems & optimal substructure property."

## ★ WAYS or TYPES OF DP

i) Memoization (Top-down approach) ↓
- In this approach, a recursive function is used to solve the problem, but before computing the result of each subproblem, it checks if the solution has already been computed & stored (in a memo table).
- If so, it retrieves the solution; otherwise, it calculates & stores it.

Ex⇒ On Page no. 552, we have seen example of Fibonacci

ii) Tabulation (Bottom-Up approach) ↑
- This approach avoids recursion & solves all subproblems in a sequential order, typically using an iterative process.
- Results are stored in a table, & each new entry builds on previously computed ones.



Ex: For n=5,

| dp[]: | 0 | 1 | 1 | 2 | 3 | 5 |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |

dp[i] = ith fib
or
dp[n] = nth fib

⇒
```
public static int fib (int n) {
    int dp[] = new int [n+1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i=2; i<=n; i++) {    // O(n)
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

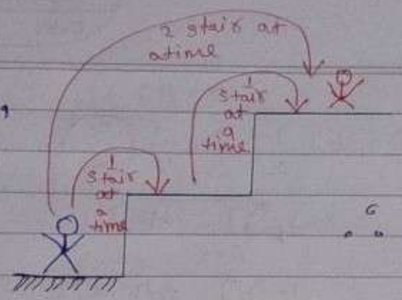
7 Important Question → 7 Concepts

1. Fibonacci
2. 0-1 Knapsack
3. Unbounded Knapsack
4. Longest Common Subsequence (LCS)
5. Kadane's Algorithm (Arrays)
6. Catalan Number
7. DP on Grid (2D-Arrays)

## ★ CLIMBING STAIRS

- Count ways to reach the nth stair. The person can climb either 1 stair or 2 stairs at a time.

* n=2,




∴ There are 2 ways.

1 1 → 1 way
2 → 1 way

* For n=3,

1 1 1 → 1 way
1 2 → 1 way
2 1 → 1 way
Total ways: 3 ways.

* For n=4,

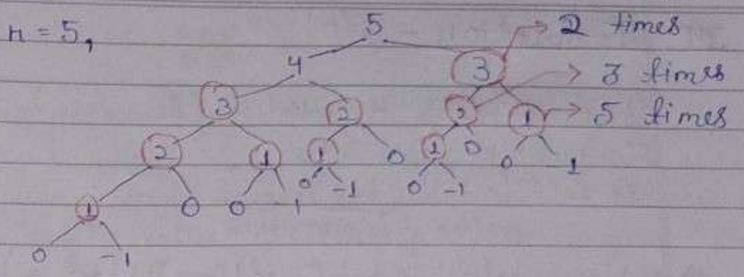
1 1 1 1 → 1 way
1 1 2 → 1 way
1 2 1 → 1 way
2 1 1 → 1 way
2 2 → 1 way
Total ways: 5 ways

## I. Recursion - $O(2^n)$

```
public static int countWays (int n) {
    if (n == 0) {
        return 1;
    }
    if (n < 0) {
        return 0;
    }
    return countWays (n-1) + countWays (n-2);
} // ways (n) = ways (n-1) + ways (n-2)
```



* For n=5,



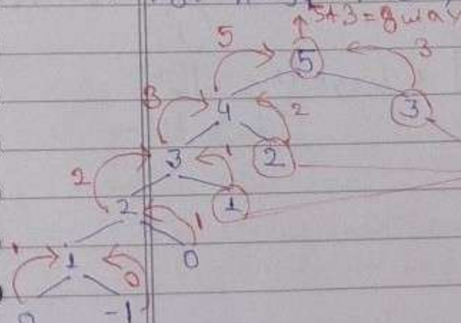

## II. Memoization - O(n)

```
public static int countWays (int n, int ways[]) {
    if (n == 0)
        return 1;
    if (n < 0)
        return 0;
    if (ways[n] != -1)
        return ways[n];
    ways[n] = CountWays (n-1, ways) + countWays (n-2, ways);
    return ways[n];
}
public static void main (String args[]) {
    int n=5;  int ways[] = new int [n+1];
    Arrays.fill (ways, -1);
    Sys.o (countWays (n, ways));
}
```

* For n=5, ⇒ ways[]:

| -1 | -1 | 2 | 3 | 5 | 8 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

5+3=8 ways



These are stored in ways[], so we haven't to calculate again & again.

## III. Tabulation – O(n)

S1: create table + intialise (base case)
S2: meaning for index.
S3: fill (small to large)
bottom-up approach

⇒
```
public static int CountWaysTab(int n) {
    int dp[] = new int[n+1];
    dp[0] = 1;
    // Tabulation loop
    for (int i=1; i<=n; i++) {
        if (i==1) {
            dp[i] = dp[i-1] + 0;
        } else {
            dp[i] = dp[i-1] + dp[i-2];
        }
    }
    return dp[n];
}
```

* For n=5,

→ dp[]:

| 1 | 1 | 2 | 3 | 5 | 8 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

index ith ⇒ at index, ith ways
meaning are stored for i.

→
```
for (int i=1 to n) {
    ways[n] = ways[n-1] + ways[n-2];
}
```
ways[1] = 1 + 0
ways[2] = 1 + 1
ways[3] = 2 + 1
ways[4] = 3 + 2
ways[5] = 5 + 3

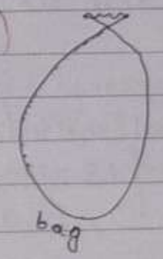


## ★ 0-1 KNAPSACK
↳ दोनों

* Problem Statement – We have to set of items, each with a weight & a value, & a knapsack with a maximum weight capacity. The goal is to maximize the total value of items you can carry in the knapsack without exceeding its weight capacity.
* 0/1 Condition – Each item can either be included (1) or not included (0) in the knapsack. We cannot take a fraction of an item or use it multiple times.
∵ This is why it's called the '0-1' Knapsack.

* Types of Knapsack: i) Fractional Knapsack (Greedy)
ii) 0-1 Knapsack
iii) Unbounded Knapsack

## I. Recursion – O(2^n)




item (val, wt)
include (1) ⇑ wt <= W — we have to check the condition for include in bag.
exclude (0)

Case 1: wt <= W
W = W - wt
X

⇒ For all items,
if (wt <= W)
→ include W-wt, i+1
→ exclude W, i+1
else
→ exclude W, i+1

* Base Case

W (capacity) = 0 then ans = 0.

Index i (items = 0) i = 0 then ans = 0.

⇒
```
public static int knapsack (int val[], int wt[], int W, int n){
  if (W == 0 || n == 0) {
    return 0;
  }
  if (wt[n-1] <= W) { //valid
    //include
    int ans1 = val[n-1] + knapsack(val, wt, W-wt[n-1], n-1);
    //exclude
    int ans2 = knapsack(val, wt, W, n-1);
    return Math.max(ans1, ans2);
  } else { //not valid
    return knapsack(val, wt, W, n-1);
  }
}
public static void main(String args[]) {
  int val[] = {15, 14, 10, 45, 30};
  int wt[] = {2, 5, 1, 3, 4};
  int W = 7;
  Syso (knapsack(val, wt, W, val.length)); //75
}
```

II. Memoization - In this, val[] wt[] → fixed & W, n → variables in recursion ↓↓ 2D Array.
O(n * W)



* At initial, all will be initialise as -1.

dp[][]:

| n (val.length) \ W | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 2 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 3 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 4 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 5 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

→ For any dp[i][j], → knapsack(w) = j (max Profit) → items = 0 to i

Ex: i=2, val = 15, 14; W=4, wt = 2, 5 ⇒ ans : dp[2][4]

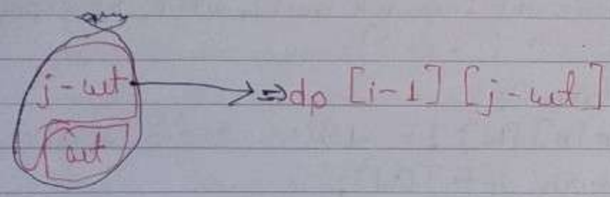
⇒
```
public static int knapsack (int val[], int wt[], int W, int n, int dp[][]){
  if (W == 0 || n == 0)
    return 0;
  if (dp[n][W] != -1)
    return dp[n][W];
  if (wt[n-1] <= W)
    int ans1 = val[n-1] + knapsack(val, wt, W-wt[n-1], n-1, dp);
    int ans2 = knapsack(val, wt, W, n-1, dp);
    dp[n][W] = Math.max(ans1, ans2);
    return dp[n][W];
  else
    dp[n][W] = knapsack(val, wt, W, n-1, dp);
    return dp[n][W];
}
public static void main (String args[]) {
  int val[] = {15,14,10,45,30}; int wt[] = {2,5,1,3,4};
  int W = 7; int dp[][] = new int[val.length+1][W+1];
  for (int i=0; i<dp.length; i++)
    for (int j=0; j<dp[0].length; j++)
      dp[i][j] = -1;
  Syso (knapsack(val, wt, W, val.length, dp));
```

## III. Tabulation - O(n*W)

S1: create table
S2: meaning of assign (i, j) + intialise Base Case
S3: fill (bottom Up) ↑ small to large

* Dp [ ] [ ] : (n+1) * (W+1) size 2D Array.
* In Dp[i] [j] ⟶ i items + j = W (knapsack size)
  ⇓
  max profit.
* Base Case: if W = 0 ⟶ profit = 0 } intialise in 2D Array.
  if i = 0 ⟶

⇒




→ item no.
For (int i = 1 to n+1) → capacity
  for (int j = 1 to W+1)
    (val, wt)
    if (wt <= j) //valid
      c1: include
        val [i-01] + dp[i-1] [j-wt]

      c2: //exclude
        dp [i-1] [j]

* We will not use recursion, we will use iteration.



⇒
```
public static int knapsack (int val[], int wt[], int W){
  int n = val.length;
  int dp[][] = new int [n+1][W+1];
  for(int i=0; i<dp.length; i++){ //0th Col
    dp [i][0] = 0;
  }
  for (int j=0; j<dp[0].length; j++){ //0th Row
    dp [0][j] = 0;
  }
  for (int i=1; i<n+1; i++){
    for (int j=1; j<W+1; j++){
      int v= val [i-1]; //ith item val
      int w = wt [i-1]; //ith item wt
      if (w<=j) { //valid
        int incProfit = v + dp [i-1][j-w];
        int excProfit = dp [i-1][j];
        dp [i][j] = Math.max(incProfit, excProfit);
      } else { //invalid
        int excProfit = dp [i-1][j];
        dp [i][j] = excProfit;
      }
    }
  }
  return dp[n][W];
}
public static void print (int dp[][]) { //print 2D-Array
  for (int i=0; i<dp.length; i++){
    for (int j=0; j<dp[0].length; j++){
      System.out.print (dp [i][j] + " ");
    }
    Sysо ();
  }
  Sysо ();
}
```

Page No: 563
Date: / /

## ★ TARGET SUM SUBSET (Tabulation)

* num[] = 4, 2, 7, 1, 3 {∵ Total Subset: $2^n$}
* Target Sum = 10

ans: ~~{7, 3}~~ true (means there is subset which can ~~{7, 2, 1}~~ achieve the target sum)
~~{~~ i.e., {7, 3}, {7, 2, 1}, {4, 2, 3, 1}

# This problem is variation of 0-1 Knapsack (tabulation).

S1: Create table
S2: meaning of (i, j) + intialization
S3: Bottom-Up manner (small to large) ↑

* 2D-Array → dp[n+1][sum+1]

| i → / j → | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | T | F | F | F | F | F | F | F | F | F | F |
| 4 1 | T | F | F | F | F | F | F | F | F | F | F |
| 2 2 | T | F | F | F | F | F | F | F | F | F | F |
| 7 3 | T | F | F | F | F | F | F | F | F | F | F |
| n+1 1 4 | T | F | F | F | F | F | F | F | F | F | F |
| 3 5 | T | F | F | F | F | F | F | F | F | F | F |

* final ans ⇒ dp[n][sum]

⇒ Answer → n items ⇒ subset sum = target ? T/F
⇒ dp(i, j) → i items ⇒ subset sum = j ? T/F

* Target sum = 0 is achievable because of { } (null subset).

* By default, 2D Array will be F but we will intialize 0th Column as T.

Page No: 564
Date: / /

```
// Time Complexity - O(n*sum)
⇒ public static boolean targetSumSubset(int arr[], int sum){
    int n = arr.length;
    boolean dp[][] = new boolean[n+1][sum+1];
    //i = items & j = target sum
    for(int i=0; i<n+1; i++)
        dp[i][0] = true;
    for(int i=1; i<n+1; i++)
        for(int j=1; j<sum+1; j++)
            int v = arr[i-1];
            if(v<=j && dp[i-1][j-v] == true)//include
                dp[i][j] = true;
            else if(dp[i-1][j] == true) //exclude
                dp[i][j] = true,
    return dp[n][sum];
}

public static void print(boolean dp[][]){ // print 2D-Array
    for(int i=0; i<dp.length; i++)
        for(int j=0; j<dp[0].length; j++)
            System.out.print(dp[i][j] + " ");
        Syso();
    Syso();
}
```

for above example:

```
true false false false false false false false false false false
true  "    "    "   true  "    "    "    "    "    "
true  "   true  "   true  "   true  "    "    "    "
true  "    "    "   true false  "   true  "   true  "
true true true true true true true true true true true
true  "    "    "    "    "    "    "    "    "    "
```

true

## ★ UNBOUNDED KNAPSACK (Tabulation)

Given: val[] = 15, 14, 10, 45, 30
wt[] = 2, 5, 1, 3, 4
W = 7kg (total allowed weight)

→ We can take an item multiple times. Find the maximum value you can achieve by putting items in the knapsack.

⇒ // Time Complexity - O(n * W)

```
public static int unbounded(int val[], int wt[], int W){
    int n = val.length;
    int dp[][] = new int[n+1][W+1];
    for(int i=0; i<n+1; i++)
        dp[i][0] = 0;
    for(int j=0; j<W+1; j++)
        dp[0][j] = 0;
    for(int i=1; i<n+1; i++){
        for(int j=1; j<W+1; j++)
            if(wt[i-1] <= j) //valid
                dp[i][j] = Math.max(val[i-1] + dp[i][j-wt[i-1]],
                                    dp[i-1][j]);
                                    //exclude
            else //invalid
                dp[i][j] = dp[i-1][j]; //exclude
    return dp[n][W];
}
```

Output : 100
(of above example)



## ★ COIN CHANGE

- variation of Unbounded Knapsack

Ex→ coins[] = 1,2,3 ans: {1,1,1,1}, {1,1,2}, {2,2}, {1,3}
sum = 4

Ex2→ coins[] = 2,5,3,6 ans: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} & {5,5}.
sum = 10

* We will use Tabulation.
* So, we will make an 2D-Array of dp[n+1][sum+1] size.

S1- Create 2D Array.
S2- Meaning of assigning (i,j) + initialization
S3- Fill in Bottom-Up manner ↑

Ex: coins[] = 2, 5, 3, 6
sum = 10

| n+1 \ sum+1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | | | | | | | | | | → (2) |
| 2 | 1 | | | | | | | | | | → (2,5) |
| 3 | 1 | | | | | | | | | | → (2,5,3) |
| 4 | 1 | | | | | | | | | | → (2,5,3,6) |

→ (i, j) → i: n, j: sum ⇒ final ans

S1: * dp[][] → (i, j) < i coins, j sum > no. of ways

S2: * Initialization: If sum = 0, then there is one way to do which is ϕ (null).
→ If we give 0 then it will indicate not possible.
→ If coins = 0, then this is not possible i.e., 0.

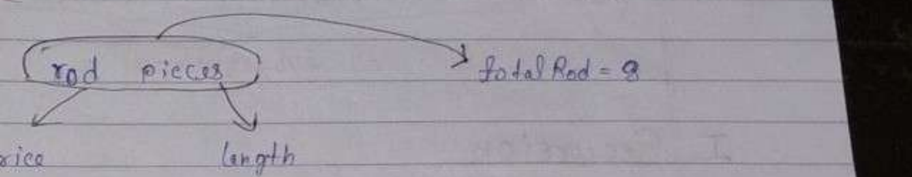
⇒
```
public static int coinChange(int coins[], int sum){
  int n = coins.length;
  int dp[][] = new int[n+1][sum+1];
  //initialise - sum is 0
  //i -> coins; j -> sum/change
  for(int i=0; i<n+1; i++){
    dp[i][0] = 1;
  }
  for(int j=1; j<sum+1; j++){
    dp[0][j] = 0;
  }
  //O(N*SUM)
  for(int i=1; i<n+1; i++){
    for(int j=1; j<sum+1; j++){
      if(coins[i-1] <= j){
        dp[i][j] = dp[i][j-coins[i-1]] + dp[i-1][j];
      }else{
        dp[i][j] = dp[i-1][j];
      }
    }
  }
  return dp[n][sum];
}
public static void main(String args[]){
  int coins[] = {2, 5, 3, 6};
  int sum = 10; //ans = 5.
  Sys o("No. of way to give Coins Change is: "
      + coinChange(coins, sum);
}
```



★ ROD CUTTING - Given a rod of length n inches & an array of prices that includes prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod & selling the ~~prices~~ pieces.

Input: length = 1 2 3 4 5 6 7 8
price[] = 1 5 8 9 10 17 17 20
rod length = 8 (2+6=8 length, 17+5=22, Prof:)

Output: 22




★ It's the variation of Knapsack Question. (unbounded)

⇒
```
// Time Complexity - O(n * totalRod)
public static int rodCutting(int length[], int price[], int totRod){
  int n = price.length;
  int dp[][] = new int[n+1][totRod+1];
  for(int i=0; i<n+1; i++)
    for(int j=0; j<totRod+1; j++)
      if(i==0 || j==0)
        dp[i][j] = 0;
  for(int i=1; i<n+1; i++)
    for(int j=1; j<totRod+1; j++)
      if(length[i-1] <= j)
        dp[i][j] = Math.max(price[i-1] + dp[i][j-length[i-1]],
                            dp[i-1][j]);
      else
        dp[i][j] = dp[i-1][j];
  return dp[n][totRod];
}
```

(LCS)

★ LOWEST COMMON SUBSEQUENCE – A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

Ex1: str1 = "abcde", str2 = "ace" ; ans = 3 // "ace"

Ex2: str1 = "abcdge", str2 = "abedg" ; ans = 4 // "abdg"

abc ⇒ "abc", "ab", "ac", "bc", "a", "b", "c", "" ...

all subsequences

## I. Recursion

str1 = "abcde" , str2 = "ace"

* From index last → first, → abcde (e) same
→ ac (e)

S1: char[n] , char[m]
str1(n-1) str2(m-1)

if same → तो हमें string की length घटा देनी है।

* Now, → abcd (d) not same → ac (e)

(अगर same नहीं है तो किसी एक string को छोटा कर देंगे।)

जिसका subsequence बड़ा होगा उसका length छोटा कर देंगे।

n-1, m (abc) (ac) same

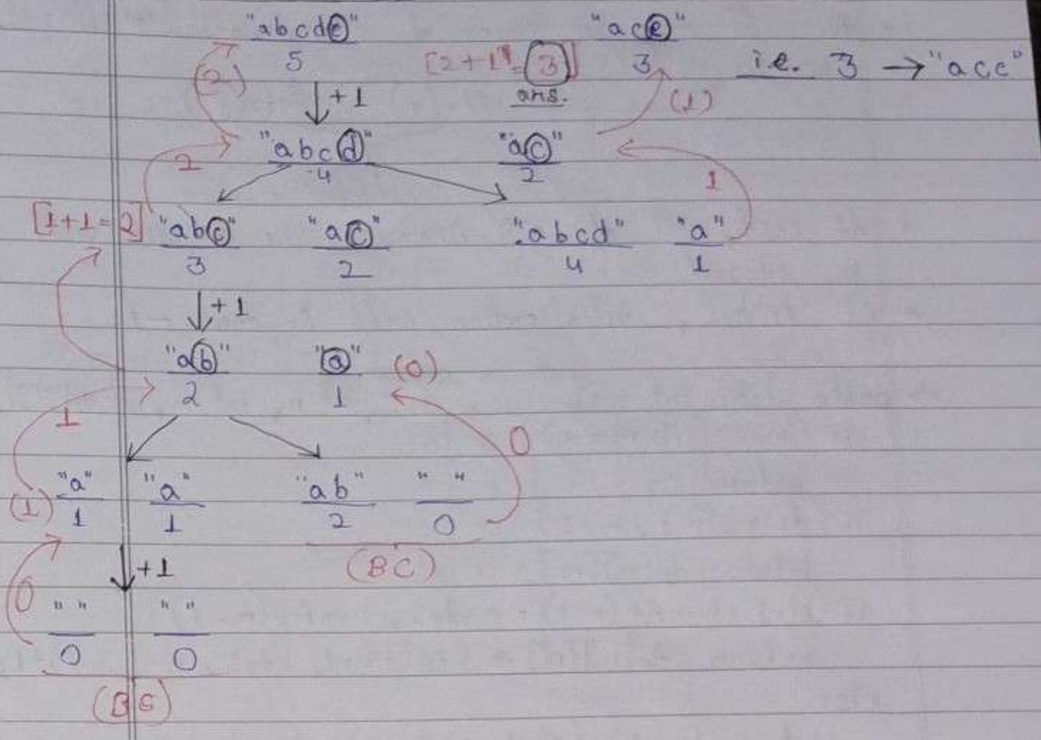
n, m-1 (abcd) (a)

n-2, m-1
ab , a
⋮
& so on.

* Base case: str1 = "" // str2 = ""
n==0 m==0
then LCS = 0.



* Recursion Tree:




⇒
```
public static int lcs(String str1, String str2, int n, int m){
    if (n==0 || m==0)
        return 0;
    if (str1.charAt(n-1) == str2.charAt(m-1))
        return lcs(str1, str2, n-1, m-1)+1;
    else
        int ans1 = lcs(str1, str2, n-1, m);
        int ans2 = lcs(str1, str2, n, m-1);
        return Math.max(ans1, ans2);
}
```

## II. Memoization

* We will make an array of String length $[n+1][n+1]$.
* String length $[i, j]$ → str(i) & str(j) → LCS.
* At last, cell of 2D-Array, the answer will be stored.
* At initial, initialisation will be done $-1$.

⇒
```
public static int lcs(String str1, String str2, int n, int m, int dp[][]){
    if (n==0 || m==0)
        return 0;
    if (dp[n][m] != -1)
        return dp[n][m];
    if (str1.charAt(n-1) == str2.charAt(m-1))
        return dp[n][m] = lcs(str1, str2, n-1, m-1)+1;
    else
        int ans1 = lcs(str1, str2, n-1, m, dp);
        int ans2 = lcs(str1, str2, n, m-1, dp);
        return dp[n][m] = Math.max(ans1, ans2);
}
public static void main(String args[]){
    String str1 = "abcdge";   String str2 = "abcdg";
    int n = str1.length();    int m = str2.length();
    int dp[][] = new int[n+1][m+1];
    for(int i=0; i<n+1; i++)
        for(int j=0; j<m+1; j++)
            dp[i][j] = -1;
    }
}
Sysо(lcs(str1, str2, n, m, dp)); // 4, ⇒ "abdg".
}
```



## III. Tabulation

S1: Create Table
S2: Meaning + initialisation (Base Case)
S3: Fill in Bottom-Up (small → large)

* str1 = "abcde" (n=5), str2 = "ace" (m=3)
  str1.length (n), str2.length (m)

S1. * dp[n+1][m+1]: → m+1 (4)

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | | | |
| 2 | 0 | | | |
| 3 | 0 | | | |
| 4 | 0 | | | |
| 5 | 0 | | | |

n+1 (6)

S2. * In a call (i, j), → str1(i), str2(j) → LCS will be stored.

* Initialisation: n↓ → 0 → n+1 → for(int i=1 to n+1)
  m↓ → 0 → m+1 → for(int i=1 to m+1)

```
if (str1(i-1) == str2(j-1))
    dp[i][j] = dp[i-1][j-1]+1
else
    ans1 = dp[i-1][j] // up cell
    ans2 = dp[i][j-1] // left cell
    dp[i][j] = max(ans1, ans2)
```

| | | | a | c | e |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| | 0 | 0 | 0 | 0 | 0 |
| a | 1 | 0 | 1 | 1 | 1 |
| b | 2 | 0 | 1 | 1 | 1 |
| c | 3 | 0 | 1 | 2 | 2 |
| d | 4 | 0 | 1 | 2 | 2 |
| e | 5 | 0 | 1 | 2 | 3 |

→ final answer.

⇒
```
public static int lcs(String str1, String str2) {
    int n = str1.length();
    int m = str2.length();
    int dp[][] = new int[n+1][m+1];
    for(int i=0; i<n+1; i++){
        for(int j=0; j<m+1; j++){
            if(i==0 || j==0){
                dp[i][j] = 0;
            }
        }
    }
    for(int i=1; i<n+1; i++){
        for(int j=1; j<m+1; j++){
            if(str1.charAt(i-1) == str2.charAt(j-1)){
                dp[i][j] = dp[i-1][j-1] + 1;
            } else {
                int ans1 = dp[i-1][j];
                int ans2 = dp[i][j-1];
                dp[i][j] = Math.max(ans1, ans2);
            }
        }
    }
    return dp[n][m];
}
public static void main(String args[]){
    String str1 = "abcdge";
    String str2 = "abedg"; //4, ⇒"abdg"
    Syso(lcs(str1, str2));
}
```



★ LONGEST COMMON SUBSTRING – a substring is a contiguous sequence of characters within a string.

Ex1: S1 = "ABCDE", S2 = "ABGCE" ⟹ 2 (AB)
Ex2: S1 = "ABCDGH", S2 = "ACDGHR" ⟹ 4 (ACDGH)

* It is a variation of LCS.
* If characters same → length + 1.
  ⇒ str1, n ; str2, m
  → nth == mth
  ⇒ (str1, n-1 ; str2, m-1) + 1

* 1. Create 2D-Array
  2. Meaning + Initialization
  3. Fill (bottom-up).
  } Notes में नहीं लिख रहे हैं / It's same like LCS.
* Time Complexity – O(n * m)
* Space Complexity – O(n * m)

⇒
```
public static int longCommonSubstring(String str1, String str2){
    int n = str1.length(); int m = str2.length();
    int dp[][] = new int[n+1][m+1]; int ans = 0;
    for(int i=0; i<n+1; i++)
        dp[i][0] = 0;
    for(int j=0; j<m+1; j++)
        dp[0][j] = 0;
    for(int i=1; i<n+1; i++)
        for(int j=1; j<m+1; j++)
            if(str1.charAt(i-1) == str2.charAt(j-1)){
                dp[i][j] = dp[i-1][j-1] + 1;
                ans = Math.max(ans, dp[i][j]);
            } else
                dp[i][j] = 0;
    return ans;
}
```

## ★ LONGEST INCREASING SUBSEQUENCE (LIS)

Ex: arr[] = {50, 3, 10, 7, 40, 80}
1 2 3 4
Length of LIS = 4
3, 7, 40, 80

* Sorted Ascending Order.
Unique

* We know HashSet, that are sorted automatically.

⇒
```
public static int lcs(int arr1[], int arr2[]){
    int n = arr1.length;
    int m = arr2.length;
    int dp[][] = new int [n+1][m+1];
    for (int i=0; i<n+1; i++)
        dp[i][0] = 0;
    for (int j=0; j<m+1; j++)
        dp[0][j] = 0;
    for (int i=1; i<n+1; i++) {
        for (int j=1; j<m+1; j++) {
            if (arr1[i-1] == arr2[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
            } else {
                int ans1 = dp[i-1][j];
                int ans2 = dp[i][j-1];
                dp[i][j] = Math.max(ans1, ans2);
            }
        }
    }
    return dp[n][m];
}
```



```
public static int lis(int arr1[]) {
    HashSet<Integer> set = new HashSet<>();
    for (int i=0; i<arr1.length; i++)
        set.add(arr1[i]);
    int arr2[] = new int [set.size()];
    int i=0;
    for (int num : set) {
        arr2[i] = num;
        i++;
    }
    Arrays.sort(arr2);
    return lcs(arr1, arr2);
}
```

## ★ EDIT DISTANCE

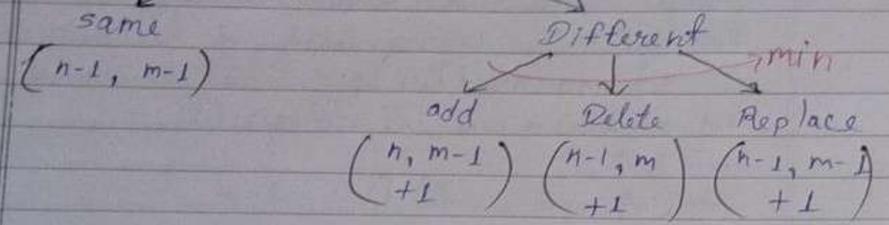
- Given two strings word1 & word2, return the minimum number of operations required to convert word1 to word2. You have the following 3 operations permitted on a word:
  - Insert a character
  - Delete a character
  - Replace a character

Input: word1 = "intention", word2 = "execution"
Output: 5.

⇒
- intention → inention (remove 't')
- inention → enention (replace 'i' with 'e')
- enention → exention (replace 'n' with 'x')
- exention → exection (replace 'n' with 'c')
- exection → execution (insert 'u')

} 5 steps

Approach:




S1. Create 2D-Array → dp[n+1][m+1]

S2. Meaning → (i, j) → str1(i), str2(j) → min operations (str1 → str2)

+

Initialisation →

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 1 | | | |
| 2 | 2 | | | |
| 3 | 3 | | | |

↓ n+1

→ m+1

S3. Fill in a Bottom-Up Manner.

* final answer will be installed in last cell.

Ex> str1 = "abc"
str2 = "abd"

⇒ if (str1(i-1) == str2(j-1))
dp[i][j] = dp[i-1][j-1]
else
dp[i][j] = min(dp[i][j-1], add
dp[i-1][j], delete
dp[i-1][j-1]) + 1
replace

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 1 | 1 | 2 | 3 |
| 2 | 2 | 2 | 1 | 2 |
| 3 | 3 | 3 | 2 | (2) |

→ final ~~ans~~ answer



```
⇒ public static int editDistance(String str1, String str2){
    int n = str1.length();
    int m = str2.length();
    int dp[][] = new int[n+1][m+1];
    //initialize
    for(int i=0; i<n+1; i++){
        for(int j=0; j<m+1; j++){
            if(i==0){
                dp[i][j] = j;
            }else
            if(j==0){
                dp[i][j]=i;
            }
        }
    }
    //bottom Up
    for(int i=1; i<n+1; i++){
        for(int j=1; j<m+1; j++){
            if(str1.charAt(i-1)==str2.charAt(j-1)){//same
                dp[i][j] = dp[i-1][j-1];
            }else{//Different
                int add = dp[i][j-1]+1;
                int del = dp[i-1][j]+1;
                int rep= dp[i-1][j-1]+1;
                dp[i][j] = Math.min(add, Math.min(del, rep));
            }
        }
    }
    return dp[n][m]
}
```

★ STRING CONVERSION – Convert String1 to String2 with only insertion & deletion. Print number of deletions & insertions.

⇒
```
public static int[] convertString (String str1, String str2){
    int m = str1.length();
    int n = str2.length();
    // find the LCS length
    int lcslen = lcs(str1, str2, m, n);
    //calculate deletions & insertions
    int delete = m - lcslen;
    int insert = n - lcslen;
    return new int[] {delete, insert};
}
private static lcs (String str1, String str2, int m, int n){
    int dp[][] = new int [m+1] [n+1];
    for (int i=1; i<=m; i++) {
        for (int j=1; j<=n; j++) {
            if (str1.charAt(i-1) == str2.charAt(j-1)){
                dp[i][j] = 1 + dp[i-1][j-1];
            } else{
                dp[i][j] = Math.max (dp[i-1][j], dp[i][j-1]);
            }
        }
    }
    return dp[m][n];
}
public static void main (String args[]) {
    String str1 = "heap";    String str2 = "pea";
    int result[] = convertString (str1, str2);
    Sysо ("Deletions: " + result[0]);
    Sysо ("Insertions: " + result[1]);
}
```



★ GATALAN'S NUMBER

→ $C_0 = 1$
→ $C_1 = 1$
→ $C_2 = C_0 * C_1 + C_1 * C_0 = 1*1 + 1*1 = 2$
→ $C_3 = C_0 * C_2 + C_1 * C_1 + C_2 * C_0 = 1*2 + 1*1 + 2*1 = 5$
→ $C_4 = \ldots = 14$, → $C_5 = 42$, ……

Ex: Find $C_4$. $0 \to n-1$ : $C_0 + C_1 + C_2 + C_3$
$n-1 \to 0$ : $C_3 + C_2 + C_1 + C_0$

$C_0C_3 + C_1C_2 + C_2C_1 + C_3C_0$
$1*5 + 1*2 + 2*1 + 5*1$
$5 + 2 + 2 + 5 = 14$

⇒ $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!}$

⇒ $C_n = C_0C_{n-1} + C_1C_{n-2} + C_2C_{n-3} + \cdots + C_{n-1}C_0$

I. Recursion

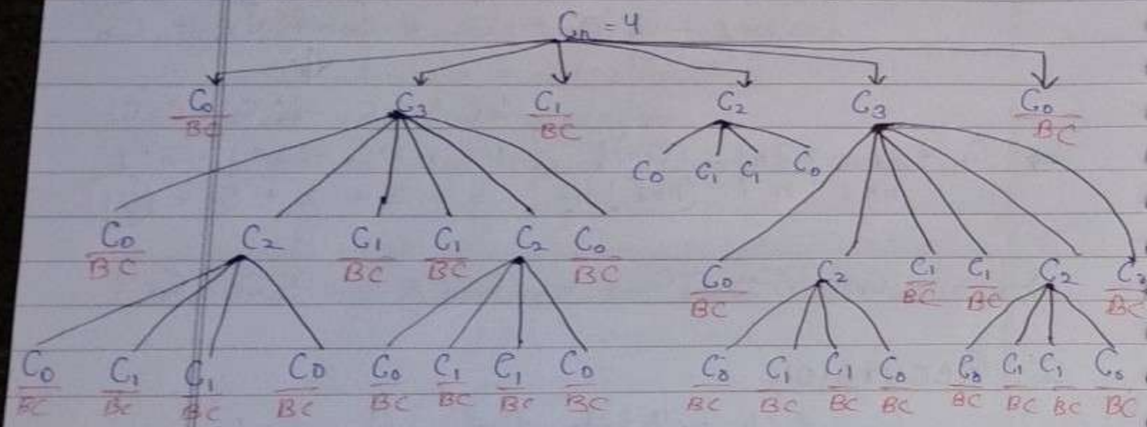
* We will note that, 0 n-1
1 n-2
2 n-3
$\therefore C_{(i)} * C_{(n-i-1)}$

* For recursion,
for (int i=0 to n-1)
$C_n$ += $C_i$ * $C_{n-i-1}$

* Base Case : if (n == 0 || n == 1)
return 1;
}

// NEXT PAGE

⇒
```
public static int catalanRec(int n) { // O(2^n)
    if (n==0 || n==1)
        return 1;
    int ans = 0; // Cn
    for (int i=0; i<n-1; i++) {
        ans += catalanRec(i) * catalanRec(n-i-1);
    }
    return ans;
}
```

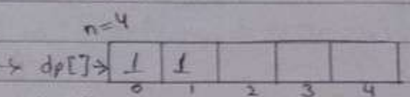



II. Memoization – We have to not calculate a $C_n$ many times, we will store value in an array.

⇒
```
public static int catalanMem(int n, int dp[]) { // O(n)
    if (n==0 || n==1)
        return 1;
    if (dp[n] != -1)
        return dp[n];
    int ans = 0;
    for (int i=0; i<n; i++)
        ans += catalanMem(i, dp) * catalanMem(n-i-1, dp);
    return dp[n] = ans;
}
```



III. Tabulation




S1: Create 1D array → dp[]

S2: Meaning of cell → dp[i] = ith Catalan

\+

Initialization → at i=0 & 1, Catalan will 1.

S3: Fill in bottom-up manner.

↳ for (i=2 to n)

for (j=0 to i-1)

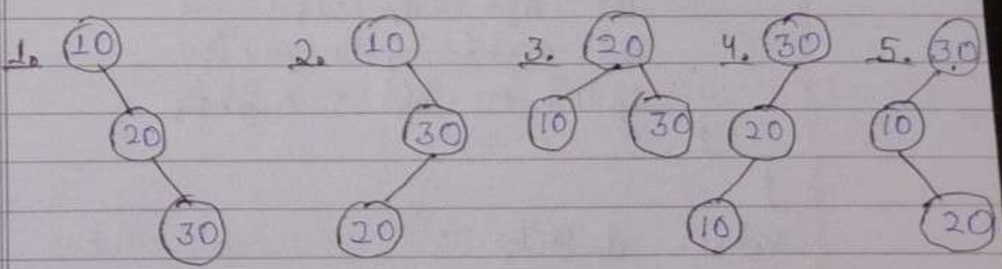
dp[i] += dp[j] * dp[i-j-1]

⇒
```
public static int catalanTab(int n) { // O(n*n)
    int dp[] = new int[n+1];
    dp[0] = 1;   dp[1] = 1;
    for (int i=2; i<=n; i++)
        for (int j=0; j<i; j++)
            dp[i] += dp[j] * dp[i-j-1]; // Ci = Cj * Ci-j-1
    return dp[n];
}
```

★ COUNTING BSTs – Find number of all possible BSTs with given n nodes.

Ex: n = 3 (10, 20, 30)

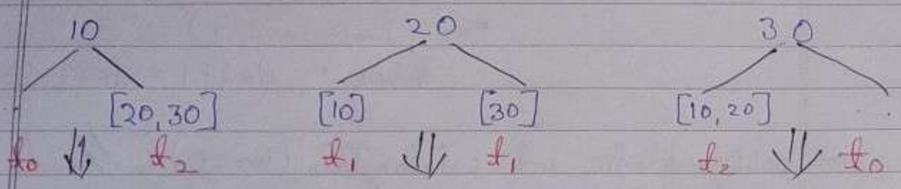
ans = 5



* For n=0, there will be 1 BST. (null node)
* For n=1, there will also be 1 BST.

⇒

| | BST |
|---|---|
| n = 0 | 1 |
| n = 1 [10] | 1 |
| n = 2 [10,20] | 2 |
| n = 3 [10,20,30] | 5 |




$t_0 * t_2 + t_1 * t_1 + t_2 * t_0$

$1 * 2 + 1 * 1 + 2 * 1$

$2 + 1 + 2$

(5)

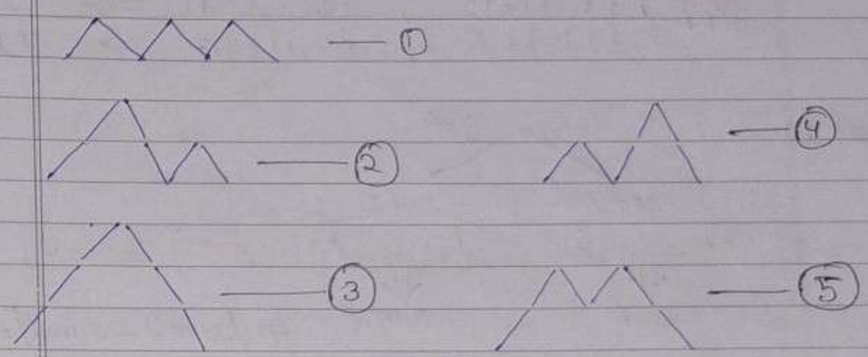
⇒
```
//Time Complexity - O(2^n)
public static int countBST (int n) {
    int dp[] = new int[n+1];
    dp[0] = 1;    dp[1] = 1;
    for (int i=2; i<n+1; i++) {
        //Ci -> BST (i nodes) -> dp[i]
        for (int j=0; j<i; j++) {
            int left = dp[j];
            int right = dp[i-j-1];
            dp[i] += left * right;
        }
    }
    return dp[n];
}
```



★ MOUNTAIN RANGES - at any moment the no. of down strokes can't be more than up strokes

Given: Up Stroke (/) Down Stroke (\)

Ex: pairs = 3 (/\, /\, /\)




Output: 5.

* It's concept is same as Count BSTs & Catalan Number.

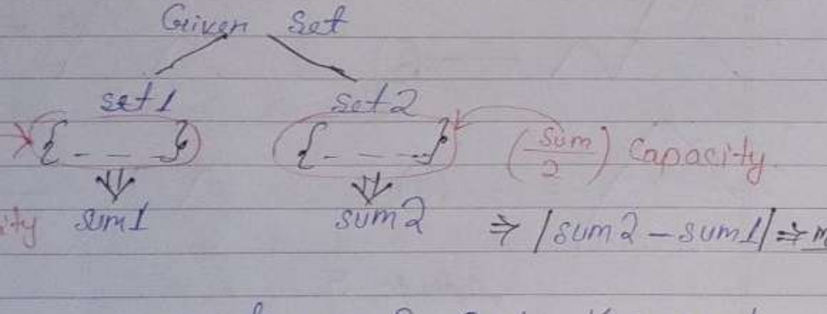
⇒
```
public static int mountainRanges (int n) { //O(n^2)
    int dp[] = new int[n+1];
    dp[0] = 1;    dp[1] = 1;
    for (int i=2; i<n+1; i++) {
        //i pairs -> mountain ranges => Ci
        for (int j=0; j<i; j++) {
            int inside = dp[j];
            int outside = dp[i-j-1];
            dp[i] += inside * outside; //Ci = Cj * Ci-j-1
        }
    }
    return dp[n]; //n pairs.
}
```

## ★ MINIMUM PARTITIONING

Input: numbers[] = {1, 6, 11, 5}
Output: min diff = 1.

| | | | | difference |
|---|---|---|---|---|
| | {1,6} = 7 | {11,5} = 16 | ⇒ | 9 |
| min diff | {1,11} = 12 | {6,5} = 11 | ⇒ | 1 |
| | {1,5} = 6 | {6,11} = 17 | ⇒ | 11 |

Approach:




* It is a variation of 0-1 Knapsack.

1. find sum of n elements.
2. dp [n+1] [W+1]
   num. length → sum/2 = (11)+1 = 12 (23/2)
   ⇓
   dp [5] [12]
3. 0-1 Knapsack
4. dp [n] [W] = sum1
5. sum2 = sum − sum1
6. min diff = |sum1 − sum2|
   ⇓
   Finalans = Math.abs (sum1 − sum2)



```
// Time Complexity - O(n * W)
⇒ public static int minPartition (int arr[]) {
    int n = arr.length;
    int sum = 0;
    for (int i=0; i< arr.length; i++) {
        sum += arr[i];
    }
    int W = sum/2;
    int dp[][] = new int [n+1] [W+1];
    //bottom up
    for (int i=1; i<n+1; i++) {
        for (int j=1; j<W+1; j++) {
            if (arr[i-1] <= j) {
                dp[i][j] = Math.max (arr[i-1] + dp[i-1][j-arr[i-1]],
                                     dp[i-1][j]);
            } else {
                dp[i][j] = d[i-1][j];
            }
        }
    }
    int sum1 = dp[n][W];
    int sum2 = sum - sum1;
    return Math.abs (sum1 - sum2);
}
```

★ MIN ARRAY JUMPS – Given an array of non-negative integers, where each element represents the maximum number of steps ~~can~~ that can be jumped forward from that element, find the minimum number of jumps required to reach the end of the array (starting from 1st element). If the end of the array is not reachable, return -1.

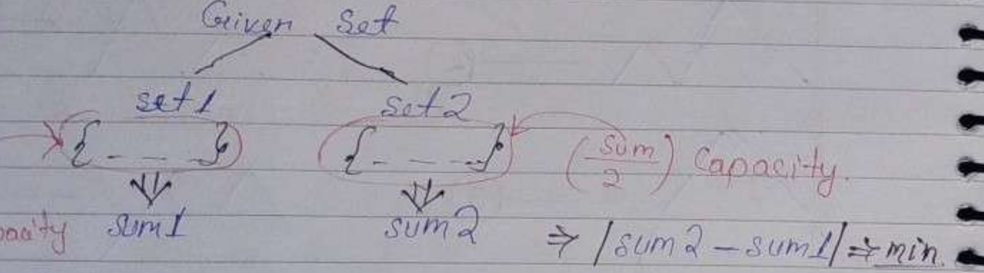
Page No: 585
Date: / /

## ★ MINIMUM PARTITIONING

Input: numbers[] = {1, 6, 11, 5}
Output: min diff = 1.

| | | | | diff |
|---|---|---|---|---|
| | {1,6} = 7 | {11,5} = 16 | ⇒ | 8 |
| min diff | {1,11} = 12 | {6,5} = 11 | ⇒ | 1 |
| | {1,5} = 6 | {6,11} = 17 | ⇒ | 11 |

Approach:




* It is a variation of 0-1 Knapsack.

1. find sum of n elements.
2. dp[n+1][W+1]
   n+1 → num. length
   W+1 → sum/2 = (11)+1 = 17 (23/2)
   ⇓
   dp[5][12]
3. 0-1 Knapsack
4. dp[n][W] = sum1
5. sum2 = sum − sum1
6. min diff = |sum1 − sum2|
   ⇓
   finalans = Math.abs(sum1 − sum2)

Page No: 586
Date: / /

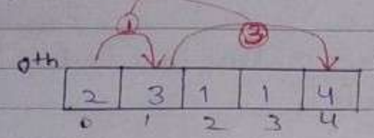
```
// Time Complexity - O(n * W)
⇒ public static int minPartition(int arr[]) {
    int n = arr.length;
    int sum = 0;
    for(int i=0; i<arr.length; i++) {
        sum += arr[i];
    }
    int W = sum/2;
    int dp[][] = new int[n+1][W+1];
    //bottom up
    for(int i=1; i<n+1; i++) {
        for(int j=1; j<W+1; j++) {
            if(arr[i-1] <= j) {
                dp[i][j] = Math.max(arr[i-1] + dp[i-1][j-arr[i-1]],
                                    dp[i-1][j]);
            } else {
                dp[i][j] = d[i-1][j];
            }
        }
    }
    int sum1 = dp[n][W];
    int sum2 = sum - sum1;
    return Math.abs(sum1 - sum2);
}
```

★ MIN ARRAY JUMPS - Given an array of non-negative integers, where each element represents the maximum number of steps ~~can~~ that can be jumped forward from that element, find the minimum number of jumps required to reach the end of the array (starting from 1st element). If the end of the array is not reachable, return -1.

Page No: 587
Date: / /

Input: int arr[] = {2, 3, 1, 1, 4} → यह सब numbers बताते हैं कि max कितना jump कर सकते हैं।
Output: min jumps = 2

max 2 jumps, max 3 jumps ...




* We will use Tabulation.

S1. Create an 1D Array of size n.

0 → n-1 Big problem
1 → n-1
2 → n-1
n-1 → n-1 Small problem

dp[n]: | -1 | -1 | -1 | -1 | 0 |
0 jumps
0 1 2 3 4

S2. Meaning → dp[i] = (i to n-1) is min jumps.
+
Initialisation → dp[n-1] (n-1) → (n-1) = 0 jumps.

S3. Fill in a Bottom Up manner (small → big problem)

| 2 | 3 | 1 | 1 | 4 |
0 1 2 3 4
dp[2] dp[3] dp[4]
i = 1

1 → 2 → (steps = 1) → jumps = dp[2] + 1
1 → 3 → (steps = 2) → jumps = dp[3] + 1 } min
1 → 4 → (steps = 3) → jumps = dp[4] + 1

Pseudo Code:
```
for (int i=n-2 to 0)
    int steps = arr[i]  // ans = ∞
    for(int j=i+1 ; j<=i+steps && j<n; j++)
        if (dp[j] != -1)
            ans = min(ans, dp[j]+1)
    dp[i] = ans ≠ ∞
```

Page No: 588
Date: / /

⇒
```
public static int minJumps(int nums[]) {
    int n = nums.length;
    int dp[] = new int[n];
    Arrays.fill(dp, -1);
    dp[n-1] = 0;
    for(int i=n-2; i>=0; i--) {
        int steps = nums[i];
        int ans = Integer.MAX_VALUE;
        for(int j=i+1; j<=i+steps && j<n; j++){
            if(dp[j] != -1) {
                ans = Math.min(ans, dp[j]+1);
            }
        }
        if(ans != Integer.MAX_VALUE) {
            dp[i] = ans;
        }
    }
    return dp[0];
}
```