

DYNAMIC PROGRAMMING

Handwritten Notes of
Striver(TUF) Playlist

by: Aashish Kumar Nayak

NIT Srinagar



AASHISH KUMAR NAYAK



aashishkumar.nayak

DP (Dynamic Programming)

1. DP1. Introduction to Dynamic programming | Memoization | Tabulation | Space optimization Techniques
2. DP2. Climbing Stairs | Learn How to write 1D Recurrence Relations
3. DP3. Frog Jump | Dynamic programming | Learn to write 1D DP
4. DP4. Frog Jump with K distance | Recursion | Follow up question
5. DP5. Maximum sum of Non-adjacent elements | House Robber | 1-D | DP on subsequences
6. DP6. House Robber 2 | 1D DP | DP on subsequences
7. DP7. Ninja's Training | Must for 2D concepts | 2D DP
8. DP8. Grid unique path | Learn everything about DP on grids / All Techniques
9. DP9. Unique paths 2 | DP on grid with Maze obstacles
10. DP10. Minimum path sum in grid | DP on grids | Asked in Microsoft
11. DP11. Triangle | fixed starting point and variable ending point | DP on grids
12. DP12. Minimum/maximum falling path sum | Variable starting and ending point | DP on grids
13. DP13. Cherry pickup II | 3D DP made easy | DP on grids
14. DP14. Subset sum Equals to Target | Identify DP on subsequences and ways to solve them
15. Partition Equal subset sum | DP on subsequences
16. Partition A set into two subsets with minimum Absolute sum difference | DP on subsequences
17. Count Subsets with sum K | DP on subsequences
18. Count partitions with given difference | DP on subsequences
19. 0/1 Knapsack | Recursion to string array space optimisation approach | DP on subsequences
20. Minimum coins | DP on subsequences | infinite supplies pattern
21. Target sum | DP on subsequences
22. DP22. Coin Change 2 | Infinite supplies problem | DP on subsequences
23. DP23. Unbounded Knapsack | 1-D Array space optimised Approach
24. DP24. Rod cutting problem | 1-D Array space optimised approach
25. DP25. Longest Common subsequences | Top-down | Bottom-up | Space optimised | DP on string
26. DP26. Print longest Common subsequences | DP on string

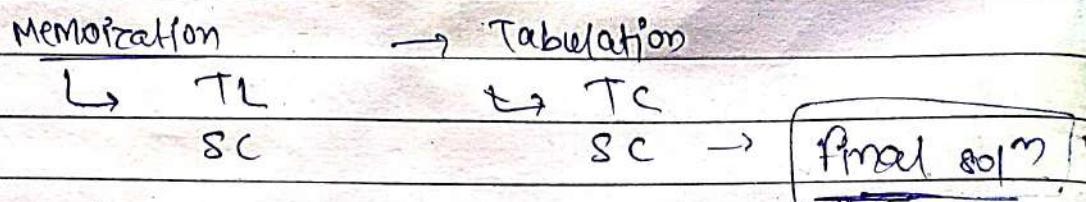
27. DP27. Longest Common Substring | DP on string
 28. DP28. Longest Palindromic subsequences |
 29. DP29. Minimum insertions to make string palindrome
 30. DP30. Minimum insertions/deletions to cover string A to string B
 31. DP31. Shortest common subsequences | DP on strings
 32. DP32. Distinct subsequences | 1D Array Optimization Technique
 33. DP33. Edit distance | Recursive to 1D Array optimised solution
 34. DP34. Wildcard matching | Recursive to 1D Array optimisation
 35. DP35. Best time to Buy and sell stocks | DP on stocks
 36. DP36. Buy and sell stocks - II | Recursion to space optimisation
 37. DP37. Buy and sell stocks - III | Recursion to space optimisation
 38. DP38. Buy and sell stocks - IV | Recursion to space optimisation
 39. DP39. Buy and sell stocks with cooldown | Recursion to space optimization
 40. DP40. Buy and Sell stocks with Transaction fee | Recursion to space optimis-
ation
 41. DP41. Longest increasing subsequence | Memoization
 42. DP42. Pointing Longest increasing subsequences | Tabulation | Algorithm
 43. DP43. Longest increasing subsequence | Binary search | Intution
 44. DP44. Largest divisible subset | Longest increasing subsequences
 45. DP45. Longest String chain | Longest increasing subsequence | LIS
 46. DP46. Longest Bitonic subsequence | LIS
 47. DP47. Number of Longest increasing subsequences
 48. DP48. Matrix chain Multiplication | MCM | partition DP starts
 49. DP49. Matrix chain Multiplication | MCM | Bottom-up | Tabulation
 50. DP50. Minimum cost to cut the stick |
 51. DP51. Burst Balloons | partition DP
 52. DP52. Evaluate Boolean Expression to True | partition DP
 53. DP53. Palindrome partitioning - II | front partition
 54. DP54. Partition array for Maximum sum | front partition
 55. DP55. Maximum Rectangle area with all 1's | DP on rectangles
 56. DP56. Count square submatrices with All ones | DP on Rectangles

DP (Dynamic programming)

PAGE NO.: _____
DATE: / /

Those who cannot remember the past
are condemned to repeat it.

- i) Tabulation ← Bottom up
- ii) Memoization ← Top-down



Q) Fibonacci No.

Pre requisite

Recursion practice

1st 10 fibo, 6th & 7th imp.

$$\begin{array}{ccccccccccccc}
 0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 \\
 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7
 \end{array}$$

$$f(n) = f(n-1) + f(n-2)$$

$$f(4) =$$

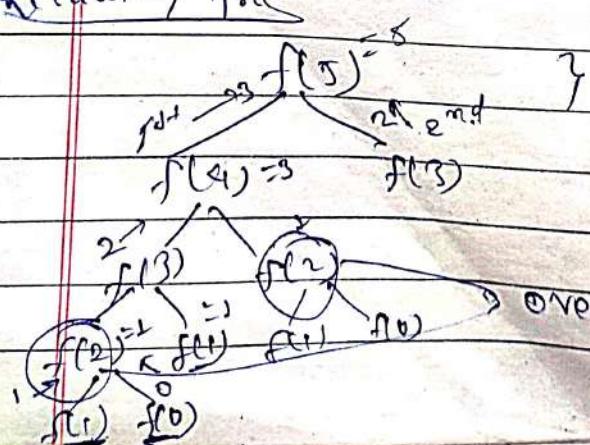
Recursion

if

$n \leq 1$

return n;

Recursion tree



return $f(n-1) + f(n-2)$;

$f(4) \rightarrow$

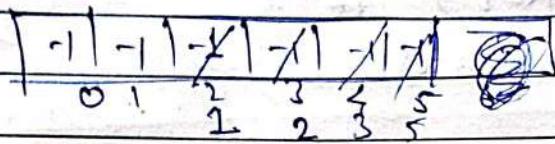
$f(3) \rightarrow$

overlapping sub problem

Memoization

© Aashish Kumar Nayak

Memorization) \rightarrow tend to store the value of sub problems in some map/table.



Initially = -1

dp[n+1]

f(n)

```

    {
        if (n <= 1)           if (dp[n] != -1)
            return n;         return dp[n];
        return f(n-1) + f(n-2); ①
    }
  
```

① convert Recursion into memorization.
②

dp[n]

Recursion \rightarrow DP

3 steps

declare a dp array

size of sub problem i.e. n

Step 1: storing the answer which is being computed for every subproblem.

Step 2: checking if the subproblem has been previously solved. If it is previously solved

Step 3: then the value will not be -1.

~~Step 3:~~

rect

for

```
int f( int n , vector<int> &dp )
```

{

```
if (n <= 1) {
```

```
    return n;
```

```
if (dp[n] != -1)
```

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

```
return dp[n] = f(n-1, dp) + f(n-2, dp);
```

{

```
int main()
```

{

```
int n;
```

```
cin >> n;
```

```
vector<int> dp(n, -1);
```

```
cout << f(n, dp);
```

```
return 0;
```

{

f(3)
f(2) o(1)

f(1)

f(3)
f(2) f(1)

f(2)
f(1)

f(1)
f(0)

linear

pattern

$T.C = O(N)$

$S.R = O(N) \rightarrow O(N)$

Recursion → Tabulation (Bottom - Up)
 ↪ (Top down)

Answer requires to
base case

Base case to the
Required

$$dp[n+1]$$

$$dp[0] = 0$$

$$dp[1] = 1$$

for($i=2; i \leq n; i++$)

$$dp[i] = dp[i-1] + dp[i-2];$$

}

Final Best solution (space optimization)

int main()

int n;

cin >> n;

int prev2 = 0;

int prev1 = 1;

Iterative solution

for($i=2; i \leq n; i++$),

$$\{ \text{curr} = \text{prev} + \text{prev2}$$

$$\text{prev2} = \text{prev};$$

$$\text{prev} = \text{curr};$$

}

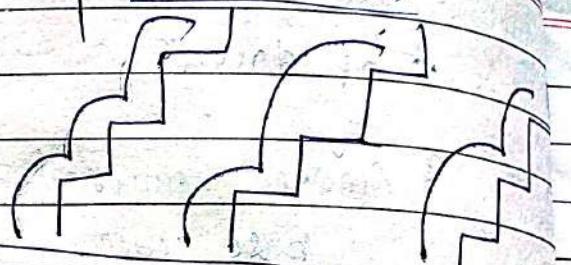
cout << prev;

Climbing Stairs problem

1D problem

$$n=3$$

~~Understand a DP problem~~



1. Concept of Try all possible way

Count

Best Way

2. Count the total no. of ways.

3. Optimal solution (min, max)

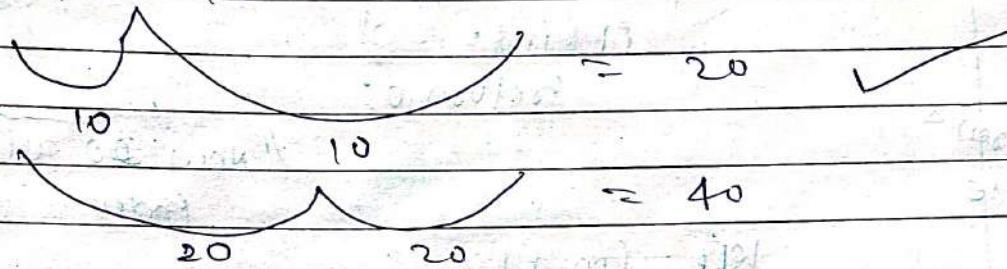
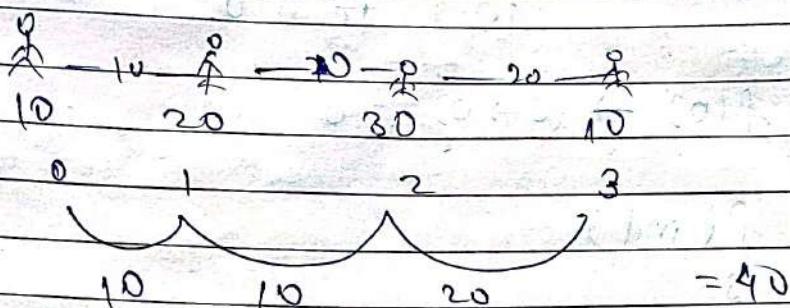
Then try to apply recursion

all possible ways

Shortcut

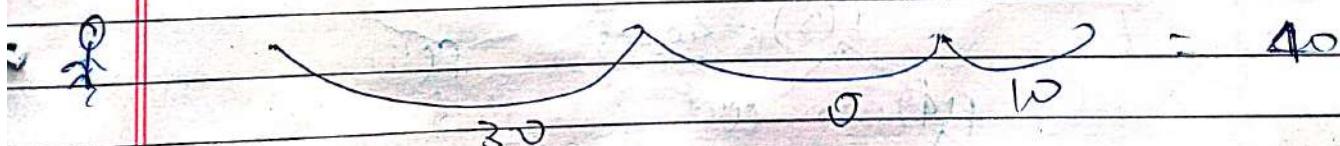
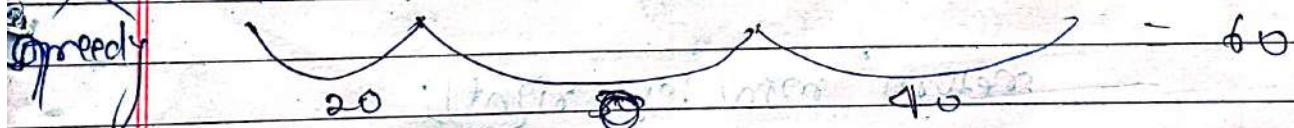
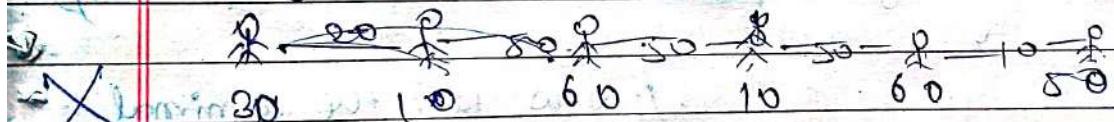
- (1) Try to represent the problem in terms of index.
- (2) Do all possible steps on that index according to the problem statement.
- (3) If question say count all ways then sum of all steps.
- (4) min (of all steps) → find min

Frog Jump Problem



(i) All possible ways, best ways, means Recursion
DP

Why a greedy soln does not work?



Q1 - P1
Greedy fails here, so we have to try Recursion.

Q2 - P2
Recursion → Try all possible ways.
(min energy)

Step 1: index

Step 2: Do all steps on that index acc. to question.

Step 3: Take @ the min (all steps).

$f(n-1) \rightarrow \min$ energy required to reach $(n-1)$ from 0.

$f(0) \rightarrow (0 \rightarrow 0) = 0$

$f(\text{ind})$

if $\text{ind} == 0$

return 0;

Recurrence
rule

// Now do all stuff on that
index +1 or -1, jump

left = $f(n-1)$

left = $f(\text{ind}-1) + \text{abs}(a[\text{ind}] - a[\text{ind}-1])$

if $\text{ind} > 1$

right = $f(\text{ind}-2) + \text{abs}(a[\text{ind}] - a[\text{ind}-2])$

// Now take the minimal
return $\min(\text{left}, \text{right})$;

$f(5) = 40 \text{ Ans}$

$f(4) = 30$

$f(3) = 20$

$f(5)$

$30 = f(4) + 10 = 40$

$f(3) = 20$

$f(2) = 20$

$f(1) = 20$

$f(0) = 20$

$\text{right} = f(3) + 40 = 60$

$\min(40, 60)$

-40
and so on 1998
 $= 998$

Q Aashish Kumar Nagarkar

$\begin{matrix} 80 & 20 & 10 & 60 & 10 & 60 & 50 \end{matrix}$
 f(1)

{ }

$$\text{left} = f(0) + \text{abs}(20)$$

$\text{right} = \text{INT_MAX}$ — $\because \text{ind} \geq 1 \Rightarrow \text{false}$
 } return (20, INT_MAX,
 } $\therefore f(1) = 20$

f(2) :

{ }

$$\text{left} = f(1) + (50) : = 70$$

$$\text{right} = f(0) + (30) : = 30$$

return (70, 30)

} $\therefore f(2) = 30$

f(3) :

{ } $\text{left} = f(2) + (50) : = 80$

right = f(1) + (0) : = 20

return (80, 20) $\therefore f(3) = 20$

}

f(4) :

{ } $\text{left} = f(3) + (50) : = 70$

right = f(2) + (0) : = 30

return (70, 30)

}

$f(4) = 30$

f(5) :

{ } $\text{left} = f(4) + (10) : = 40$

right = f(3) + (10) : = 60

}

return (40, 60) ; $f(5) = 40$ Ans

for (i)

Q8 (ii) Now convert Recurrence into DP

{

Recurrence \rightarrow DP

if

Memorization

→ look at the parameters
are changing.

ind is changing

∴ max size of ind is 5

∴ $5+1 = 6$

DP[6]

Step 1: declare an array of size $n+1$ [$DP[n+1] :=$] — (1)

Step 2: Before returning add it up.

Store it and return.

return $DP[ind] = \min(left, right);$ — (2)

Step 3:

check if it is previously computed or not.

if ($DP[ind] \neq -1$)return $DP[ind];$ — (3)TC $\rightarrow O(N)$ SC $\rightarrow O(N) + O(N)$

Recursion

array

Let's code

int f(int ind, vector<int> &heights, vector<int> &dp) :

if(ind == 0)

return 0; if(dp[ind] != -1) return dp[ind];

int left = f(ind-1, heights, dp) + abs(heights[ind] - heights[ind-1]);

int right = INT_MAX;

if(ind > 1)

right = f(ind-2, heights, dp) +

abs(heights[ind] - heights[ind-2]);

* return min(left, right);

✓ return dp[ind] = min(left, right);

only some

test case will

pass

int frogjump(int n, vector<int> &heights)

vector<int> dp(n+1, -1); — ① step

return f(n-1, heights, dp)

T.O.F.

Now optimized — MEMORIZATION.

TC =

TC $\rightarrow \Theta(N)$

SC $\rightarrow \Theta(N) + \Theta(N)$

recursion array
dp

Step 1: declare dp

dp[n+1]

Step 2: store it and return (overlapping)
return $\Rightarrow dp[ind] = \min(\text{left}, \text{right})$ (sub problem)

Step 3: check if it is previously
computed or not

if if(dp[ind] != -1)

return dp[ind];

(@Aashish Kumar Nagarkar)

Memoization

(TOP-down)

Tabulation

(Bottom-up)

Tabulation method

int dp[n] →
for zero based index
pnf f0ogsump (int n, vector<int> &heights)

{ vector<int> dp(n, 0);
dp[0] = 0;

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

{ int fs = dp[i-1] + abs(height[i] - height[i-1]);

int ss = INT_MAX;

if (i>1)

{ ss = dp[i-2] + abs(height[i] - height[i-2]);

dp[i] = min(fs, ss);

} return dp[n-1];

TC → O(N)

SC → O(N)

array

dp

+ O(N)
Recursion
reduced

Space Optimization

PAGE NO.:

DATE: / /

int frogJump (int n, vector<int> &height)

S

int prev = 0;

int prev2 = 0;

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

S

int fs = prev + abs(height[i] - height[i-1]);

int ss = INT_MAX;

if (i > 1)

ss = prev2 + abs(height[i] - height[i-2]);

int cur = min(fs, ss);

prev2 = prev;

prev = cur;

?

return prev;

3

T.C. = O(N)

SC = O(N) + O(N)

array

DP

Recursion

: SC = O(1)

both are reduced

Recurrence & Code

f(ind)

{ if (ind == 0)

return 0;

int minSteps = INT_MAX;

for (int j=1; j<=k; j++)

{

if (ind - j >= 0)

{

Jump = f(ind - j) + abs(a[ind] - a[ind - j])

minSteps = min(minSteps, jump);

}

Now Memoize the code

f(ind)

~~Tabulation~~

{

int dp[n];

dp[0] = 0;

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

{

for (;

minSteps = INT_MAX;

for (j=1; j<=k; j++)

if (i - j >= 0)

{

Jump = dp[i - j] + abs(a[i] - a[i - j])

minSteps = min(minSteps, jump);

{}

}

$dP[i] = \min_{\text{steps}}$

}

point $(dP[n-1, j])$

$T \rightarrow O(N \times K)$

$SC \rightarrow \underline{O(N)}$

can't optimize
further

as $SC = O(K) + \underline{O(N)}$

MAXIMUM SUM OF Non- ADJACENT ELEMENTS

lets try out all subsequences with the
given constraints

pick the one
with maximum sum.

Try out all ways
Recursion

Print all subsequences

→ pick / non-pick

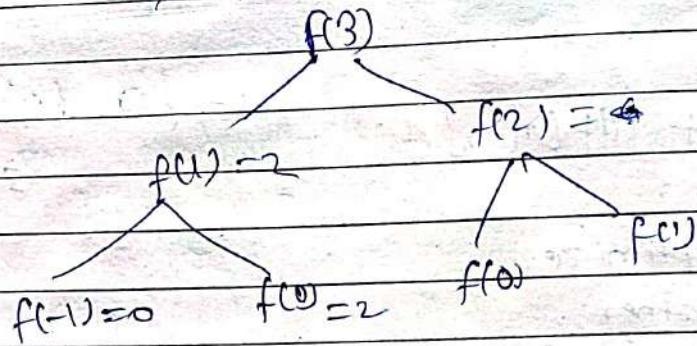
1. Index

2. Do stuffs

3. return the best you can get

PICK subsequence with no adjacent element.

0 1 2 3
2, 1, 4, 9



f(1)

g

$$f(1) \Rightarrow \text{sm}[0 \dots 1] \quad \text{pick} = a[1] + f(-1) = 1$$

$$\text{not pick} = 0 + f(0) = 2$$

$$\max(\text{pick}, \text{not pick}) \Rightarrow$$

$$\therefore \underline{\underline{f(1) = 2}}$$

f(3)

g

$$\text{pick} = a[3] + f(1) = 11$$

$$\text{not pick} = 0 + f(2) = 6$$

$$\max(\text{pick}, \text{not pick}) \therefore \underline{\underline{f(3) = 11}}$$

$$\text{pick} = a[2] + f(0) = 6$$

$$\text{not pick} = 0 + f(1) = 2$$

$$\max(\text{pick}, \text{not pick}) \therefore \underline{\underline{f(2) = 6}}$$

Tc - O(N)

Sc - O(N) + O(N)

(Ashish Kumar Nayak)

Recurrence code

```

int f(int ind, vector<int> &nums, &dp)
{
    if(ind == 0)
        return nums[ind];
    if(ind < 0)
        return 0;
    int pick = nums[ind] + f(ind - 2, nums, dp);
    int not_pick = 0 + f(ind - 1, nums, dp);
    return dp[ind] = max(pick, not_pick);
}

int maxSumNonAdjacent(vector<int> &nums)
{
    int n = nums.size();
    vector<int> dp(n - 1);
    return (n - 1, nums, dp);
}

```

Now Memoized To reduce Time

Tabulation

$$\text{int } \text{dp}[0] = a[0];$$

$$\text{int } \text{neg} = 0;$$

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

}

$$\text{take} = \text{dp}[i-1];$$

~~$\text{non-take} = 0;$~~

if ($i > 1$)

$$\text{take} = \text{take} + \text{dp}[i-2];$$

$$\text{Non-take} = 0 + \text{dp}[i-1];$$

$$\text{dp}[i] = \max(\text{take}, \text{non-take});$$

}

$$TC = O(N)$$

$$SC = O(1)$$

Space Optimization

$$\text{int } \text{prev} = a[0]$$

$$\text{int } \text{prev2} = 0;$$

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

{

$$\text{take} = \text{prev};$$

if ($i > 1$)

$$\text{take} = \text{take} + \text{prev2};$$

$$\text{non-take} = 0 + \text{prev};$$

$\text{curr} = \max(\text{take}, \text{non-take});$

$\text{prev2} = \text{prev};$

$\text{prev} = \text{curr};$

?
print(prev);

* int Max_sum_Non_Adjacent(vector<int> &numbers)

{

int n = numbers.size();

int prev = numbers[0];

int prev2 = 0;

for(int i = 0; i < n; i++)

{

int take = numbers[i];

if(i > 1)

take = take + prev2;

int notTake = 0 + prev;

int curr = max(take, notTake);

prev2 = prev;

prev = curr;

? return prev;

@Aashish Kumar Nayak

Aashish Kumar Nayak

HOUSE ROBBER

int max (vector<int> &nums)

{

int n = nums.size();

int prev = nums[0];

int prev2 = 0;

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

{ int take = &nums[i];
if (i > 1)

take = take + prev2;

int notTake = 0 + prev;

int curi = max(take, notTake);

prev2 = prev;

prev = curi;

return prev;

int houseRobber (vector<int> &valueInHouse)

{ vector<int> temp1, temp2;

int n = valueInHouse.size();

if (n == 1) return valueInHouse[0];

for (int i = 0; i < n; i++)

{ if (i == 0) temp1.push_back (valueInHouse[i]);

if (i == 1) temp2.push_back (valueInHouse[i]);

return max (maxNonAdj (temp1), maxNonAdj (temp2));

NINJA TRAINING problem

Whenever

Greedy fails

Try all possible ways

↓
Recursion

1. index

2. Do stuff on that index
3. Find max or min

$f_0 \quad t_1 \quad t_2$

$\underline{2} \quad 1 \quad 3 \quad d_0$
 $\underline{3} \quad 4 \quad 6 \quad d_1$

$10 \quad 1 \quad 6 \quad d_2$

$8 \quad 3 \quad 7 \quad d_3$

$f(3,3)$

$f(3,0)$

$f(2,1)$

$f(2,0)$

$f(2,1)$ $f(2,2)$

$f(1,0)$

$f(1,1)$

$f(1,2)$

$f(0,0) \quad f(0,1) \quad f(0,2)$

$\underline{3}$

$f(1,1)$

{

$\overset{1,3}{\cancel{f(1,2)}}$

$$8 + f(0,0) \underset{f(0,1)}{\cancel{f(0,2)}} = 6$$

$$6 + f(0,2) \underset{f(0,1)}{\cancel{f(0,2)}} = 8$$

?

$f(1,1) \rightarrow 8$

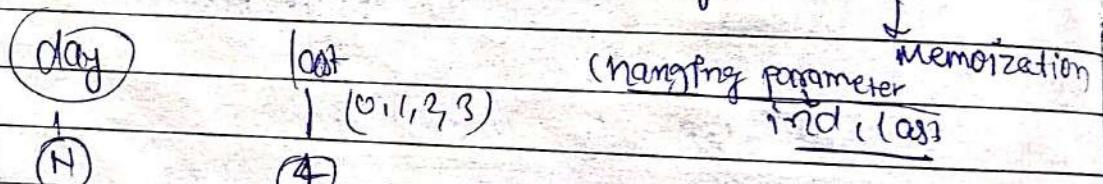
@Aashish Kumar Negi

$$\begin{array}{c}
 f(1,2) \\
 \left\{ \begin{array}{l} 3+3 = 6 \\ 4+3 = 7 \end{array} \right. \\
 f(0,0) \quad f(0,1) \quad \overbrace{\quad \quad \quad}^6
 \end{array}$$

$$\begin{array}{c}
 f(2,0) \quad \overbrace{\quad \quad \quad}^8 \\
 \left\{ \begin{array}{l} 1+f(1,1) = 9 \\ 6+f(1,2) = 12 \end{array} \right. \quad \overbrace{\quad \quad \quad}^{12} = 12
 \end{array}$$

$$f(2,1)$$

Overlapping subproblem



$N \times 4$ dp size array
dp[day][last]

Recursion code + memoized

```

int f(int day, int last, vector<vector<int>> &points, dp)
{
    if(day == 0)
    {
        int maxi = 0;
        for(int task = 0; task < 3; task++)
        {
            if(task != last)
            {
                maxi = max(maxi, points[0][task]);
            }
        }
        return maxi;
    }
}
  
```

return maxi;

```
if(dp[day][last] == -1)
    return dp[day][last];
```

```
int maxi = 0;
```

```
for(int task = 0; task < 3; task++)
```

```
}
```

```
if(task != last)
```

```
}
```

```
int point = points[day][task] + f(day-1, task,
    pointers, dp);
```

```
maxi = max(maxi, point);
```

```
}
```

```
return maxi;
```

```
}
```

```
return dp[day][last] = maxi;
```

```
int ninjaTrain(int n, vector<vector<int>> &points)
```

```
vector<vector<int>> dp(n, vector<int>(3, -1));
```

```
return f(n-1, 0, points, dp);
```

```
}
```

$T_L = O(N \times 4) \times 3$

$S_C = O(N) + O(N \times 4)$

Tabulation

$$dp[0][0] = \max(\text{ans}[0][1], \text{ans}[0][2])$$

$$dp[0][1] = \max(\text{ans}[0][0], \text{ans}[0][2])$$

$$dp[0][2] = \max(\text{ans}[0][0], \text{ans}[0][1])$$

$$dp[0][3] = \max(\text{ans}[0][0], \text{ans}[0][1], \text{ans}[0][2])$$

Feb

Calculation

PNT inserting (pnt n, vector<vector<int>> &points)

{ vector<vector<int>> dp(n, vector<int>(i, 0));

$$dp[0][0] = \max(points[0][1], points[0][2]);$$

$$dp[0][1] = \max(points[0][0], points[0][2]);$$

$$dp[0][2] = \max(points[0][0], points[0][1]);$$

$$dp[0][3] = \max(points[0][0], points[0][1], points[0][2]);$$

for (pnt day = 1; day < n; day + 1)

{ for (int last = 0; last < 4; last + 1)

$$\{ dp[day][last] = 0;$$

for (int task = 0; task < 3; task + 1)

{

$$\text{if } task == last$$

-

$$\text{int } task = last$$

{

$$\text{int point} = points[day][task]$$

$$= dp[day - 1][task];$$

$$dp[day][task] = \underline{\underline{m}}$$

$$= \max(dp[day][last], point);$$

?

?

TC = O(NV^3)

SC = O(NV^2)

return dp[n - 1][0]; } }

Space Optimization

int knapsack(int n, vector<vector<int>> &points)

 vector<int> prev(i, 0);

$$prev[0] = \max[points[0][1], points[0][2]]; \\$$

$$prev[1] = \max[points[0][0], points[0][2]]; \\$$

$$prev[2] = \max[points[0][0], points[0][1]]; \\$$

$$prev[3] = \max[points[0][0], points[0][1], points[0][2]]; \\$$

for (int day = 1; day < n; day++)

 vector<int> temp(i, 0);

 for (int last = 0; last < 4; last++)

$$temp[last] = 0; \\$$

 for (int task = 0; task < 3; task++)

 if (task == last)

$$temp[last] = \max(temp[last], \\$$

$$points[day][last] + prev[task]); \\$$

$$prev = temp; \\$$

return prev[3];

$$TC = O(N \times 4 \times 3)$$

$$SC = O(4)$$

constant

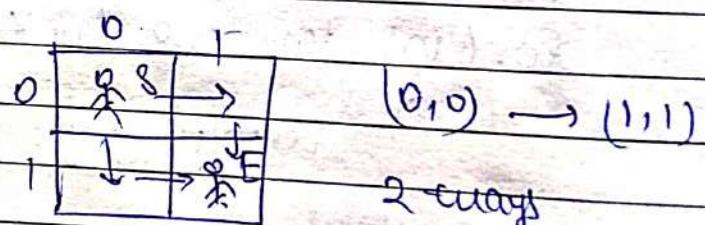
DP on Grids / 2D Matrix

- count paths
- count paths with obstacles
- min path sum
- max path sum
- Triangle
- 2 start points

6 problem

Total unique paths

unique path



(0,0) → (m-1, n-1)

Try all possible ways

Means apply Recursion

How to write recursion

1. Express in terms of index $f(i, j)$ now w/

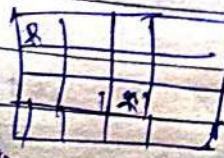
2. Explore no all steps

3. Sum up all ways (min) max.

$f(i, j) \rightarrow$ no of unique ways
(0,0) → (i, j)

$f(i, j)$

Base case



return 1 if reach destination

return 0 if can't reach destination

if $f(i, j)$

if

$i == 0 \& j == 0$

return 1;

Base case

EXPRESS

if ($i < 0 \& j < 0$)

return 0;

if ($dp[i][j] == -1$) return $dp[i][j]$;

EXPRESS

$up = f(i-1, j);$

$left = f(i, j-1);$

CUM UP

return $(up + left);$

if

$dp[i][j] =$

↑ upper
← ↗ 2 path
right

$TC \Rightarrow (2^{mn})$

$SC \Rightarrow O(\text{path length})$

$(0,0) (0,1) (0,2) (0,3)$

$(1,3)$

$(2,3)$

$(3,3)$

Recursion → DP

✓ Memoization

overlapping sub problem.

0	0	0
1		
2		.

let $m = 3, n = 3$

$f(2, 2)$

$\swarrow up \quad \searrow left$

$f(1, 2)$

$f(2, 1)$

$\swarrow f(0, 2) \quad \searrow f(1, 1)$

$f(-1, 2)$

$f(0, 1)$

$f(0, 0)$

$f(-1, 1)$

$f(0, 0)$

$f(0, 1)$

$up = f(1, 1)$

$left = (0, 0)$

return $0 + 1$

↓

$TC = O(N \times M)$

$f(1, 1)$

$f(0, 1)$

$f(0, 0)$

$SC = O(m+n) + O(m-1)$

$+ O(N \times M)$

DP

↑ deletion

Tabulation:

$dp[m][m];$
if $dp[0][0] = 1$

for ($i=0 \rightarrow m-1$)
 {
 for ($j=0 \rightarrow n-1$)

Memoization \rightarrow Tabulation

1. Declare Base Case
2. Express various states in for loop
3. copy the recurrence & write.

if ($i=0 \& j=0$)

$dp[0][0] = 1$

else

if ($i>0$) $up = dp[i-1][j];$

if ($j>0$) $left = dp[i][j-1];$

$dp[i][j] = up + left;$

3

3

print ($dp[i][j]$); $T.C = O(N \times M)$

$S.C = O(N \times M)$

→ If there is a previous row & previous column, we can space optimized it.

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

best way to optimised

$$dp[m] \rightarrow 0$$

for ($i=0 \rightarrow m-1$)

{ for ($j=0 \rightarrow n-1$)

 if ($i=-0 \& j=0$)

$$dp[i][j] = 1$$

else {

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

}

Code int f(int i, int j, vector<vector<int>> &dp)

{

 if ($i=-0 \& j=0$)

 return 1;

 if ($i<0 \text{ or } j<0$) return 0;

 if ($dp[i][j] \neq -1$)

 return dp[i][j];

 int up = f(i-1, j, dp);

 int left = f(i, j-1, dp);

 return dp[i][j] = up + left;

}

int uniquepath (int m, int n)

{ int dp[m][n];

for (int i=0; i<m; i++)

{ for (int j=0; j<n; j++)

} if (j==0 || j==0)

dp[i][j] = 1;

else

{

int up=0;

int left=0;

if (i>0) up

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

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

dp[i][j] = up + left;

{

? ?

return dp[n-1][j-1];

{

~~TLE~~

int UniquePath(int m, int n)

{

vector<int> prev(n, 0);

for (int i = 0; i < n; i++)

{

vector<int> cur(n, 0);

for (int j = 0; j < m; j++)

{

if (i == 0 && j == 0)

cur[j] = 1;

else {

int up = 0;

for (int left = 0;

if (i > 0) up =

prev[j];

if (j > 0)

left = cur[j - 1];

dp[i][j] = up + left;

}

}

prev = cur;

}

return prev[m - 1];

}

Unique paths 2 DP on Grid with Maze obstacle

0 → 0 → 0

↓ ↘ ↓
0 → 0 → 0

↓ ↘ ↓
0 → 0 → 0

0 ways

f(i,j)

{

if ($i >= 0$ & $j >= 0$ & $M[i][j] == -1$)

return 0;

if ($i == 0$ & $j == 0$)

return 1;

if ($i < 0$ || $j < 0$)

return 0;

if ($dp[i][j] == -1$) return $dp[i][j]$;

up = $f(i-1, j)$;

left = $f(i, j-1)$;

return up + left;

} return $dp[i][j] = up + left$;

Tabulation :-

```

for (i=0 → m-1)
  {
    for (j=0 → n-1)
      {
        if (i==0 & j==0)
          dp[i][j] = 1;
        else
          if (i>0) up = dp[i-1][j];
          if (j>0) left = dp[i][j-1];
        return up + left;
      }
  }

```

Code recursion → Memorization

```

int mod = (long)(1e9+7);
int f(int i, int j, vector<vector<int>> &mat, vector<vector<int>> &dp)
{
  if (i>=0 & j>=0 & mat[i][j]==-1) return 0;
  if (i==0 & j==0) return 1;
  if (i<0 || j<0) return 0;
  if (dp[i][j]==-1) return dp[i][j];
  int up = f(i-1, j, mat, dp);
  int left = f(i, j-1, mat, dp);
  return (up + left)%mod;
}
int mazePathCount(int n, int m, vector<vector<int>> &mat)
{
  vector<vector<int>> dp(n, vector<int>(m, -1));
  return f(n-1, mat, dp);
}

```

Tabulation :

```

int mod = (int)(1e9 + 7);
int dp[n][m];
for(int i=0; i<n; i++)
{
    for (int j=0; j<m; j++)
    {
        if (mat[i][j] == -1) dp[i][j] = 0;
        else if (i==0 && j==0) dp[i][j] = 1;
        else
            up = 0, left = 0;
            if (i>0) up = dp[i-1][j];
            if (j>0) left = dp[i][j-1];
            dp[i][j] = (up + left) % mod;
    }
}
return dp[n-1][m-1];
    
```

Space Optimization

int mod = (int)(1e9 + 7);

int maxObstacle (int n, int m, vector<vector<int>> &mat)

vector<int> prev(m, 0);

for (int i=0; i<n; i++)
 {

vector<int> curr(m, 0);

for (int j=0; j<m; j++)
 {

if (mat[i][j] == -1)
 curr[j] = 0;

else if (j==0 && i==0) curr[j] = 1;

else
 {

int up = 0, left = 0;

if (i>0) up = prev[j];

if (j>0) left = curr[j-1];

$$\text{cur}[j] = (\text{up} + \text{left}) \% \text{mod};$$

}

}

prev = cur;

}

return prev[m-1];

MINIMUM PATH SUM IN GRID

	0	1	2	
0	5	9	6	2x3
1	11	5	2	(0,0) → (0,2)

$$5 + 11 + 5 + 2 = 23 \downarrow \downarrow$$

$$5 + 9 + 5 + 2 = 21 \rightarrow \text{MM}$$

Why not follow greedy

	10	8	2
10	5	100	
1	1	2	

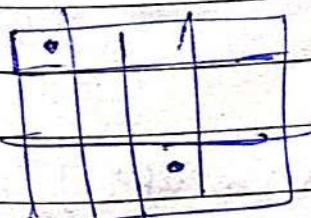
$$\text{greedy} = 10 + 8 + 2 + 1 + 2$$

$$\text{Min} = 10 + 10 + 1 + 1 + 2$$

Try out all paths

Recursion

Fell me the path which contains min cost



1.) index(i,j)

2.) EXPLORE / all Paths

3.) Take the min path

dp[i,j]

(0,0) → (i,j)

f(m-1, m-1) → (0,0) → (n-1, m-1)

$f(i, j)$

{

$f(i=0 \& j=0)$

return $a[0][0]$

$f(i < 0 \& j < 0)$

return INT_MIN

(2) $f(dp[i][j] = -1) \rightarrow dp[i][j]$

$up = a[i][j] + f(i-1, j)$

$left = a[i][j] + f(i, j-1)$

(1) $dp[i][j] =$

return $\min(up, left)$

}

③ Recursion → memorization

overlapping

sub problem

(1) $dp[n][m] \rightarrow -1$

Code

int f(int i, int j, vector<vector<int>> &grid,
vector<vector<int>> &dp)

{ if ($i == 0 \& j == 0$)

return grid[i][j];

if ($i < 0 \& j < 0$) return -1;

if ($dp[i][j] == -1$) return dp[i][j];

int up = grid[i][j] + f(i-1, j, grid, dp);

int left = grid[i][j] + f(i, j-1, grid, dp);

④ Aashish Kumar Nayak

return $dp[i][j] = \min(\text{left}, \text{up})$.

}

int minsumpath(vector<vector<int>> &grid)

{

int n = grid.size();

int m = grid[0].size();

vector<vector<int>> dp(n, vector<int>(m, -1));

return f(m-1, n-1, grid, dp);

}

$v(n, -1) \rightarrow v(m, \text{vector<int>}(-1))$

Tabulation =

$v(n, -1) \rightarrow v(m, \text{vector<int>}(-1))$

int minsumpath(vector<vector<int>> &grid)

{

int n = grid.size();

int m = grid[0].size();

vector<vector<int>> dp(m, vector<int>(n, 0));

for (int i=0; i<n; i++)

{ for (int j=0; j<m; j++)

if ($i=0 \& j=0$)

$dp[i][j] = grid[i][j];$

else {

int up = grid[i][j];

if ($i > 0$) up += dp[i-1][j];

else up += neg;

int left = grid[i][j];
if ($j > 0$) left += dp[i][j-1];
else left += 409;

$$dp[i][j] = \min(\text{left}, \text{up})$$

}
}
}

Space Optimization

PAGE NO.: / /
DATE: / /

vector<int> cuo(m, 0);

for (int j = 0; j < n; j++)

{

if (i == 0 && j == 0)

cuo[j] = grid[i][j];

else

int up = grid[i][j];

if (i > 0) up += prev[j];

else up += 1e9;

int left = grid[i][j];

if (j > 0) left += cuo[j - 1];

else left += 1e9;

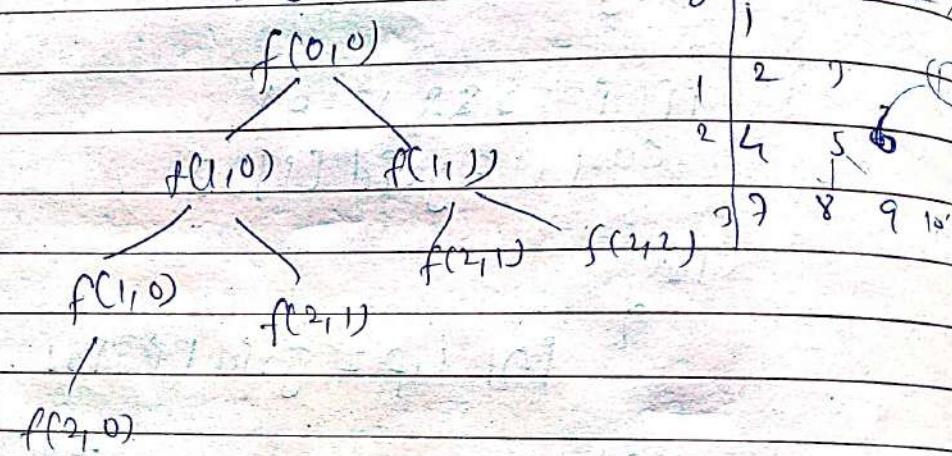
cuo[j] = min(left, up);

prev = cuo;

return prev[n - 1];

Q-10

triangle fixed starting point
variable ending point



Printf (int i, int j, vector<vector<int>> &triangle, int n, vector<vector<int>> dp)

TLE

{ if (i == n - 1)

return triangle[n - 1][j];

}

if (dp[i][j] != -1)

{

return dp[i][j];

int d = triangle[i][j] + f(i + 1, j, triangle, n, dp);
int dg = triangle[i][j] + f(i + 1, j + 1, triangle, n, dp);

return dp[i][j] = min(d, dg);

{

int minimumTriangle (vector<vector<int>> &triangle, int n),
vector<vector<int>> dp(n, vector<int>(n, -1));

return f(0, 0, triangle, n);

}

To convert TLE \Rightarrow Tabulation \Rightarrow

int minsumpath(vector<vector<int>> &triangle, int n)

{ vector<vector<int>> dp(n, vector<int>(n, 0)); }

for (int j = 0; j < n; j++)

{ dp[n-1][j] = triangle[n-1][j]; }

for (int i = n-2; i >= 0; i--)

{ for (int j = i; j >= 0; j++)

{

int d = triangle[i][j] + dp[i+1][j];

int dg = triangle[i][j] + dp[i+1][j+1];

dp[i][j] = min(d, dg);

}

return dp[0][0];

} Space Optimization \Rightarrow

int minsumpath(vector<vector<int>> &triangle, int n) } front = curr
{ vector<int> front(n, 0), curr(n, 0); } both loop ends

for (int j = 0; j < n; j++)

{ front[j] = triangle[n-1][j]; }

return front[0];

for (int i = n-2; i >= 0; i--)

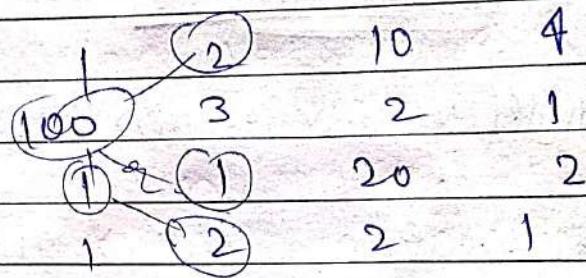
{ for (int j = i; j >= 0; j--) }

{ int d = triangle[i][j] + front[j]; }

int dg = triangle[i][j] + front[i+1]; }

curr[j] = min(d, dg); }

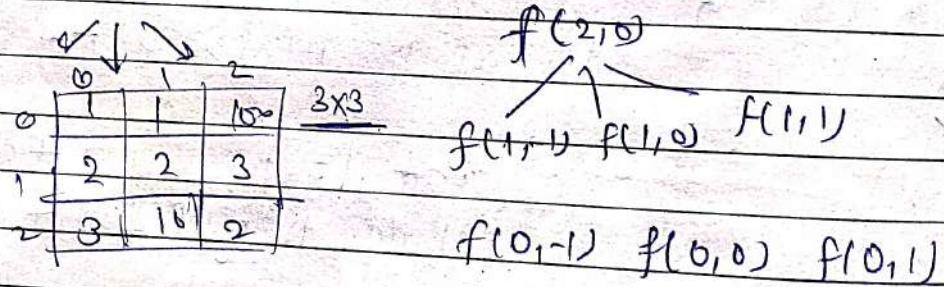
D 12

MINIMUM/MAXIMUM FALLING PATH SUM

Greedy \rightarrow fails because of uniformity

Greedy will pass only when elements are uniform.

We can not apply greedy where uniformity is not there.



$f(1,0)$

$$\{ \quad d = 2 + f(0,1) \quad - \text{eq}$$

$$u = 2 + f(0,0)$$

$$rd = 2 + f(0,1)$$

Rp

Recursion code

```

int f( int i, int j, vector<vector<int>> &matrix, vector<vector<
    <int>> &dp )
{
    if( j < 0 || j >= matrix[0].size() )
        return -1e9;

    if( dp[i][j] != -1 )
        return dp[i][j];

    if( matrix[i][j] == -1 ) return dp[i][j] = -1;

    int u = matrix[i][j] + f(i-1, j, matrix, dp);
    int ld = matrix[i][j] + f(i-1, j-1, matrix, dp);
    int rd = matrix[i][j] + f(i-1, j+1, matrix, dp);

    return dp[i][j] = max( u, max( ld, rd ) );
}

```

3
 $\text{int getmax}(\text{vector<} \text{vector<} \text{int} \text{>>} \text{&} \text{matrix})$

```

{
    int n = matrix[0].size();
    int m = matrix[0].size();
}
```

$\text{vector<} \text{vector<} \text{int} \text{>>} \text{dp}(n, \text{vector<} \text{int} \text{>}(m, -1))$

int maxi = -1e9;

for(int i = 0; i < n; i++)

maxi = max(maxi, f(n-1, i, matrix, dp));

return maxi;

Tabulation =

`int getmax (vector<vector<int>> &matrix)`

{
 int n = matrix.size();
 int m = matrix[0].size();

`vector<vector<int>> dp (n, vector<int>(m, 0));`

for (int j = 0; j < m; j++)

{
 dp[0][j] = matrix[0][j];

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

{
 for (int s = 0; s < m; s++)

 int u = matrix[i][s] + dp[i - 1][s];

 int ld = matrix[i][s] + dp[i - 1][s - 1];

 int rd = matrix[i][s] + dp[i - 1][s + 1];

 int ld = matrix[i][s];

 if (s - 1 > 0)

 ld += dp[i - 1][s - 1];

 else

 ld += -1e8; → prev[i - 1];

 int rd = matrix[i][s];

 if (s + 1 < m)

 rd += dp[i - 1][s + 1];

 else

 rd += -1e8; → prev[i + 1];

 dp[i][s] = max(u, max(ld, rd));

 prev = curr[s];

```
int maxi = -1e8;  
for (int j = 0; j < m; j++)  
{  
    maxi = max(maxi, dP[n-1][i][j]);  
}  
return maxi;
```

Space Optimization code :-

```
int curr[m], prev[m];
```

Just replace $dP[i][j] \rightarrow prev[j]$

& make $prev = curr$;

and \rightarrow return ~~prev~~.prev

in this case

~~return maxi = maxf(0, maxi);~~

return ~~maxf(maxi, prev);~~

Cherry Pickup

Rules

- ① Express everything in terms (i_1, i_1) & (i_2, i_2)
- ② Explore all-one paths $\leftarrow \downarrow$
- ③ Max sum

fixed starting point

variable ending point

$(0, 0)$

$(0, m-1)$

end (at any index in the last row)

(All paths by Alice) + (All paths by Bob)

Recursion

Recursion

face together

$f(0, 0, 0, m-1)$

use

not

base case

i) out-of-boundary

ii) last row case (Destination)

Boundary can always write first

$f(i_1, i_1, i_2, i_2)$

if ($i_1 < 0$ || $i_1 > m$ || $i_2 < 0$ || $i_2 > m$)

return -1e8;

↙↓ means they reach at same time.

$\text{if } i == n-1 \}$

{

$\text{if } (j_1 == j_2)$

return $a[i][j_1]$;

else

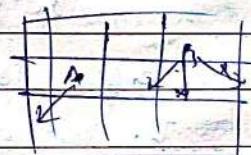
return $a[i][j_1] + a[i][j_2];$

{

② Explore all the paths

$(i+1, j_1) \rightarrow (i+1, j_2)$

In one step move Alice & Bob both



maxi = 0;

for(dj1 → -1 → +1)

{

for(dj2 → -1 → +1)

{

maxi = max(maxi, f(i+1, j_1,

if(j1 == j2) } , a[i][j1] + f(i+1, j1+dj1, j2+dj2)); }

} else

maxi = max(maxi, f(i)[j1] + f(i)[j2] + f(i+1, j1+dj1, j2+dj2)); }

}

$T \rightarrow 3^n \times 3^n = \text{exponential}$
 $S_C = O(N) \text{ A.S.C}$

Auxiliary stack space

Recursion code

int f(int i, int j1, int j2, int r, int c,
 vector<vector<int>> &grid, vector<vector<int>> &dp)

{

if (j1 < 0 || j2 < 0 || j1 >= c || j2 >= c)

{

return -1e8;

{

if (i == r-1)

{

if (j1 == j2)

return grid[i][j1];

else return grid[i][j1] + grid[i][j2];

}

if (dp[i][j1][j2] != -1)

{ return dp[i][j1][j2];

}

maxi = -1e8;

for (int dj1 = -1; dj1 <= +1; dj1++)

{

for (int dj2 = -1; dj2 <= +1; dj2++)

{

int value = 0;

if (j1 == j2) value = grid[i][j1];

else

value = grid[i][j1] + grid[i][j2];

value += f(i+1, j1+dj1, j2+dj2, r, c, grid, dp);

maxi = max(maxi, value);

{

{

return dp[i][j1][j2] = maxi;

{

```
int maximumchocolate( int r, int c, vector<vector<int>> &grid)
```

{
 vector<vector<vector<int>>> dp[n], vector<vector<int>>(c, vector<int>(c, -1));

```
return f(b, 0, c-1, r, c, grid, dp);
```

{

Tabulation

```
int MaxChocolate( int n, int m, vector<vector<int>> &grid)
```

{
 vector<vector<vector<int>>> dp[n], vector<vector<int>>(m, vector<int>(m, 0));

```
for( int j1 = 0; j1 < m; j1++ )
```

{

```
    for( int j2 = 0; j2 < m; j2++ )
```

{

```
        if( j1 == j2 )
```

```
            dp[n-1][j1][j1] = grid[n-1][j1];
```

else

```
            dp[n-1][j1][j2] = grid[n-1][j1] + grid[n-1][j2];
```

{

```
for( int i = n-2; i >= 0; i-- )
```

{

```
    for( int j1 = 0; j1 < m; j1++ )
```

{

```
        for( int j2 = 0; j2 < m; j2++ )
```

{

```
            int maxi = -1e8;
```

```
            for( int dj1 = -1; dj1 <= +1; dj1++ )
```

{

```
                for( int dj2 = -1; dj2 <= +1; dj2++ )
```

```
{ int value = 0;
  if (j1 == j2) value = grid[i][j2];
```

else

$$value = grid[i][j1] + grid[i][j2];$$

$$\text{if } (j1 + dj1) \geq 0 \text{ & } j1 + dj1 < m \\ \text{& } (j2 + dj2) \geq 0 \text{ & } j2 + dj2 < m\}$$

value += -1e8;

maxi = max(maxi, value);

}

}

$$dP[i][j1][j2] = maxi;$$

}

}

return dP[0][0][m-1];

}

Space Optimization

int dP[r][c][c];
 1D → two variable
 2D → 1D DP
 3D → 2D DP

int maximumChocolates(int n, int m, vector<vector<int>> &grid)

vector<vector<int>> front(m, vector<int>(m, 0));

vector<vector<int>> cur(m, vector<int>(m, 0));

for (int j1 = 0; j1 < m; j1++)

{ for (int j2 = 0; j2 < m; j2++)

 } if (j1 == j2) front[j1][j2] = grid[n-1][j1];

else front[s1][s2] = grid[n-1][j1] + grid[m-1][s2];

}

}

for (int i = n-2; i >= 0; i--)

{

for (int j1 = 0; j1 < m; j1++)

{

for (int s2 = 0; s2 < m; s2++)

{

int maxi = -1e8;

for (int dss = -1; dss <= +1; dss++)

{

for (int dj1 = -1; dj1 <= +1; dj1++)

{

int value = 0;

if (j1 + dj1) >= 0 & & j1 + dss < m;

if (s2 + dj1) >= 0 & & s2 + dss < m)

value += front[j1 + dj1][s2 + dss];

else

value += -1e8;

maxi = max(maxi, value);

}

curr[i1][i2] = maxi;

}

front = curr;

return front[0][m-1];

}

DP-14

PAGE NO.: / /
DATE: / /

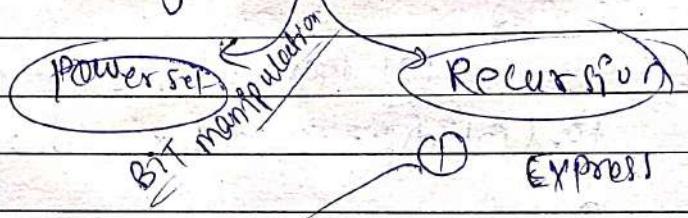
SUBSET SUM EQUAL TO K

DP m subsequences | subsets
↓

(contiguous) Non contiguous

$$[1, 3, 2] \rightarrow [1, 2], [3, 2]$$

→ Generate all subsequences & select if any of them gives a sum of K



- ① Express (ind, target)
 $f(n-1, \text{target})$

- ② Explore possibilities of the index.

Tabulation

- ① Base case

- ② form the Nested loops

$$\text{ind} \rightarrow [1 - n-1]$$

$$\text{target} \rightarrow (1 - \text{target})$$

$$\text{arr} [2, 3, 1, 4]$$

$$\text{target} = 5$$

$$f(3, 4)$$

index false.

$$f(2, 3)$$

$$(0, 4)$$

$$f(1, 2)$$

$$(0, 3)$$

$$f(0)$$

$$f(0, 2)$$

$$f(-1, 0)$$

$$f(-1, 2)$$

©Aashish Kumar Nayak

Recursion

PAGE NO.: _____
DATE: / /

bool f (int ind, int target, vector<int> &arr,
vector<vector<int>> &dp)

{ if (target == 0) return true; if (ind == 0) return (arr[0] == target);
if (dp[ind][target] != -1) return dp[ind][target];
return dp[ind][target] =

bool notTake = f(ind - 1, target, arr, dp);

bool take = false;
if (arr[ind] <= target)
take = f(ind - 1, target - arr[ind], arr, dp);

return dp[ind][target] = take | notTake;

} bool subsetsumk (int n, int k, vector<int> &arr)

{ vector<vector<int>> dp (n, vector<int> (k + 1, -1));

return f (n - 1, k, arr, dp);

$$TC = O(2^n)$$
$$SC = O(N)$$

Tabulation:

bool subsetSumTOK (int n, int k, vector<int> arr)

vector<vector<int>> dp (n, vector<int> (k + 1, 0));

for (int i = 0; i < n; i++)
 dp[i][0] = true;

dp[0][arr[0]] = true;

for (int ind = 1; ind < n; ind++)

{
 for (int target = 1; target <= k; target++)

bool notTake = dp[ind - 1][target];

bool take = false;

if (arr[ind] <= target)

take = dp[ind - 1][target - arr[ind]]

dp[ind][target] = take | notTake;

}

}

return dp[n - 1][k];

}

Whenever you see dp[int - 1]

means we can optimise space.

TC - O(N * target)
 SC - O(target)

Arashish Kumar Rayak

bool subsetSumOK(int n, int k, vector<int>&arr)

{
vector<bool> prev(k+1, 0), cur(k+1, 0);

prev[0] = cur[0] = true;

prev[arr[0]] = true;

for (int ind=1; ind < n; ind++)

{

 for (int target=1; target <= k; target++)

{

 bool notTake = prev[target];

 bool take = false;

 if (arr[ind] <= target)

 take = prev[target - arr[ind]];

 cur[target] = take | notTake;

}

 prev = cur;

}

- return prev[k];

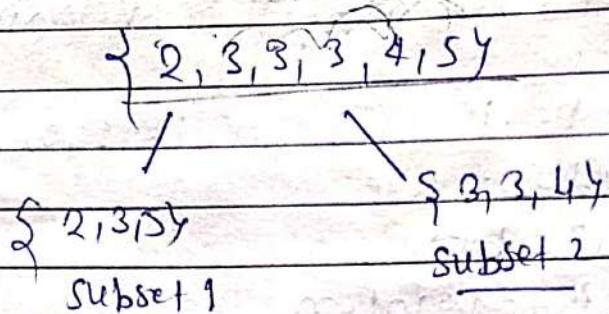
O(n)

3

DP-15

Partition Equal Subset Sum

PAGE NO.:
DATE:



Exactly 2 subsets

$$S_1 = S_2$$

Entire array sum = S

$$\begin{array}{c} S \\ \swarrow \quad \searrow \\ S_1 \quad S_2 \end{array} \quad S_1 = S_2 = \frac{S}{2}$$

$S = \text{total sum}$

If S = odd \rightarrow \times
not possible

If S = even

Then look for target sum $\frac{S}{2}$

If one subset you get $\frac{S}{2}$ then you already
got 2nd subset.

Now this question converts into

$\text{arr} \rightarrow []$

Subset sum = $\frac{S}{2}$

where

$S = \text{sum of given array}$.

Space optimization code :-

```
bool subsetsumTOK( int m, int K, vector<int> arr )
```

{

```
vector<bool> prev[ K+1, 0 ], curr[ K+1, 0 ];
prev[ 0 ] = curr[ 0 ] = true;
```

```
prev[ arr[ 0 ] ] = true;
```

```
for( int ind = 1; ind < m; ind++ )
```

{

```
    for( int target = 1; target <= K; target++ )
```

{

```
        bool notTake = prev[ target ];
```

```
        bool take = false;
```

```
        if( arr[ ind ] <= target )
```

```
            take = prev[ target - arr[ ind ] ].
```

```
        curr[ target ] = take | notTake;
```

```
    prev = curr;
```

or - ?

```
return prev[ K ];
```

}

```
bool Canpartition( vector<int> arr, int n )
```

{

```
    int totsum = 0;
```

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

{

```
        totsum += arr[ i ];
```

if(totsum % 2 != 0) return false;

```
    int target = totsum / 2;
```

```
    return subsetsumTOK( m, target, arr );
```

DP-16

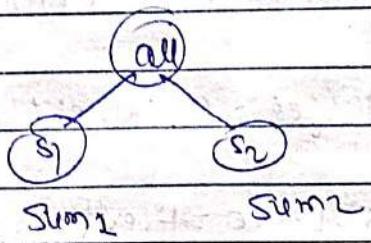
Partition A set into Two Subsets with minimum absolute sum

[1, 2, 3, 4]

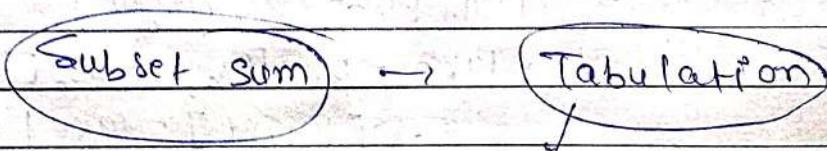
$$\{1, 2\} \quad \{3, 4\} = |7 - 3| = 4$$

$$\{1, 3\} \quad \{2, 4\} = |6 - 4| = 2$$

$$\{1, 4\} \quad \{2, 3\} = |5 - 5| = 0$$



$\text{abs}|\text{sum}_1 - \text{sum}_2|$ is minimum



If we check for a target $= k$

We can derive if every possible target between (1 & k) is \checkmark/x

[3, 2, 7]

S_1
min 0
max 12

all possible S_1

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

$\checkmark \times \sim \checkmark \checkmark \times \checkmark \times \checkmark \times \checkmark \times \checkmark \times \checkmark$

dp[8][12+1]

Any

$\rightarrow 12+1$

S_1	12	10	8	6	4	2	0	3	5	7	9	11	10	12	diff.
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

dp[3][12+1]

int $dp[n][totsum + 1]$

Subset sum code

int minSubsetSumDifference (vector<int> &arr, int n)

{

int totsum = 0;
for (int i=0; i<n; i++)
 totsum += arr[i];

int k = totsum;

vector<vector<bool>> dp (n, vector<bool> (k+1, 0));

for (int i=0; i<n; i++) dp[i][0] = true;

if (arr[0] <= k) dp[0][arr[0]] = true;

for (int ind = 1; ind < n; ind++)

{ for (int target = 1; target <= k; target++)

{ bool nottake = dp[ind-1][target];

bool take = false;

if (arr[ind] <= target)

take = dp[ind-1][target - arr[ind]];

dp[ind][target] = take | nottake;

y

y

int mini = 1e9

for (int i=1; i <= totsum/2; i++)

{ if (dp[n-1][n-i] == true)

{ mini = min (mini, abs((totsum

- i) - n));

return mini;

@Aashish Kumar Nayak

Number of subsets.

$\{1, 2, 2, 3\}$

$\{1, 2\}$ target = 3

$\{1, 2\}$

$\{3\}$

No. of subset = 3

If care of count

in base case

return 1 if true

return 0 if false

Always

Recurrence

- 1.) Express in terms of (index, target)
- 2.) Explore all the possibilities
- 3.) Sum up all the possibilities & return.

0	1	2	3
1	2	2	3

(ind)

$f(m-1, \text{target})$

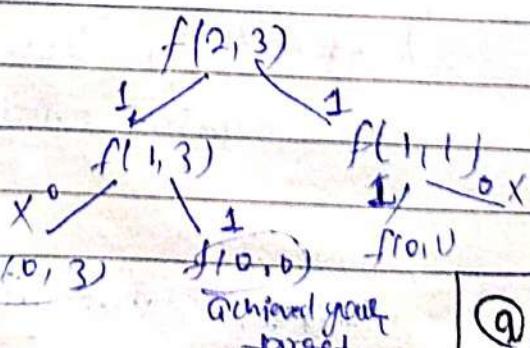
not pick = $f(\text{ind}-1, \text{target})$.

pick = 0;

if ($a[\text{ind}] \leq s$)

pick = $f(\text{ind}-1, \text{target} - a[\text{ind}])$

action $\text{pick} + \text{notpick};$



No. of subset = 2 $\{1, 2\}, \{3\}$

Tabulation

- ① Base Case
- ② looks at the changing parameter & write nested loops
- ③ copy the recursion

```
int DP[N][S+1]
for (i=0 → n-1) DP[i][0] = 1
if (a[i] ≤ S) DP[0][a[0]] = 1
```

for (i=1 → n-1)
for (s=0 → sum)

} copy the recursion

$$TC = O(N \times \text{sum})$$

$$SP = O(N \times \text{sum})$$

Recursion code :-

if

int f(int ind, int sum, ~~vector<int>~~ &num, vector<int> &dp)

{

 if (num == 0) return 1;

 if (dp[ind][num] != -1) return dp[ind][num];

 int notTake = f(ind-1, sum, num, dp);

 int take = 0;

 if (num[ind] <= sum)

 take = f(ind-1, sum - num[ind], num, dp);

 return dp[ind][num] = notTake + take;

}

int findways(vector<int> &num, int tar)

{

 int n = num.size();

 vector<vector<int>> dp(n, vector<int>(tar+1, -1));

 return f(n-1, tar, num, dp);

}

Tabulation :-

PAGE NO.:

DATE: / /

```
int findways (vector<int> &num, int tar)
```

{

```
    int n = num.size();
```

```
    vector<vector<int>> dp(n, vector<int> (tar+1, 0));
```

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

```
        { dp[i][0] = 1;
```

```
    if (num[0] <= tar)
```

```
        dp[0][num[0]] = 1;
```

```
    for (int ind = 1; ind < n; ind++)
```

{

```
        for (int sum = 0; sum <= tar; sum++)
```

{

```
            int nottake = dp[ind-1][sum];
```

```
            if (take == 0)
```

```
                if (num[ind] <= sum)
```

```
                    take = dp[ind-1][sum - num[ind]];
```

~~DP DP return~~

```
dp[ind][sum] = nottake + take;
```

? return dp[n-1][tar];

Space Optimization Code

`int findways(vector<int> &sum, int tar)`

```

    {
        int n = sum.size();
        vector<int> prev(tar+1, 0), curr(tar+1);
        prev[0] = curr[0] = 1;
        if (num[0] <= tar)
            prevnum[0] = 1;
    }

```

`for (int ind = 1; index < n; ind++)`

```

    {
        for (int sum = 0; sum <= tar; sum++)
    }

```

```

    {
        int notTake = prev[sum];
    }

```

`int take = 0;`

`if (num[ind] <= sum)`

`take = prev[sum - num[ind]];`

`curr[sum] = notTake + take;`

`prev = curr;`

```

    }
    return prev[tar];
}

```

Count positions with given sum differences.

Given arr $\rightarrow \{0, 0, 1\}$ sum = 1

(Ans = 1) \uparrow^4

$\{0, 1\}$

$\{0, 1\}$

$\{0, 0, 1\}$

$\{1\}$

No. of zeros $\rightarrow 2$

Power $(2, n) \times \text{ans}^{\uparrow}$

Power ref

$2^2 \times 1$

$4 \times 1 = 4$ subset

$S_1 = S_2$

2

$S_1 - S_2 = D$

$S_1 = \text{totsum} - S_2$

$D = \text{totsum} - S_2 - S_1$

$D = \text{totsum} - 2S_2$

5

$S_2 = \frac{\text{totsum} - D}{2}$

Count of subset $\in \left(\frac{\text{Totsum} - D}{2} \right)$

DP-17

modified
target

$\text{Totsum} - D \geq 0$

$\text{totsum} - D$ has to be even

Recursion + Memoization code

```
int mod = (int)(1e9 + 7);
```

```
fn1 f(int ind, int sum, vector<int> &num, vector<vector<int>> &dp)
```

if (ind == 0)

```
{ pf(sum == 0 && num[0])
```

```
return 2;
```

```
if (sum == 0 || sum == num[0]) r
```

```
eturn 1;
```

```
eturn 0;
```

2

```
sp(dp[ind][sum] := -1)
```

```
return dp[ind][sum];
```

```
int notTake = f(ind - 1, sum, num, dp);
```

```
int take = 0;
```

```
if (num[ind] == num)
```

```
take = f(ind - 1, sum - num[ind], num, dp),
```

return $dp[ind][sum] = (notTake + take) \mod$

return $dp[n-1][tar]$;

fn1 countPartitions(int n, int d, vector<int> arr)

int totsum = 0;

for (auto &i: arr) totsum += i;

if (totsum - d < 0 || (totsum - d) % 2) return false;

return f(n - 1, d, arr, (totsum - d) / 2);

```
int findways(vector<int> &sum, int tar)
```

{

```
int n = num.size()
```

```
vector<vector<int>> dp(n, vector<int>(tar + 1, -1));
```

```
return f(n - 1, tar, num, dp);
```

}

```
int countpartitions(int n, int d, vector<int> &arr)
```

{

```
int totsum = 0;
```

```
for (auto &it : arr)
```

```
    totsum += it;
```

```
If (totsum - d) <= 0 || (totsum - d) % 2
```

```
    return false;
```

```
return findways(arr, (totsum - d) / 2);
```

}

Tabulation

Point `FindWay(vector<int> &num, int target);`

`int n = num.size();`

`vector<vector<int>> dp(n, vector<int>(target+1, 0));`

`if (num[0] == 0)`

`dp[0][0] = 2;`

`else dp`

`dp[0][0] = 1;`

`if`

space optimization

Code

PAGE NO.: _____
DATE: / /

```
int findways (vector<int> &num, int , int )
```

```
{ int n = num.size();
```

```
vector<int> prev (tar + 1, 0), curr (tar + 1, 0);
```

```
if (num[0] == 0)
```

```
prev[0] = 2;
```

```
else prev[0] = 1;
```

```
if (num[0] != 0 && num[0] <= tar)
```

```
prev[num[0]] = 1;
```

```
for (int ind = 1; ind < n; ind++)
```

```
{
```

```
for (int sum = 0; sum <= tar; sum++)
```

```
{
```

```
int nottake = prev[sum];
```

```
int take = 0;
```

```
if (num[ind] <= sum)
```

```
take = prev[sum - num[ind]];
```

```
curr[sum] = (nottake + take) % mod;
```

```
prev = curr;
```

```
return prev[tar];
```

```
int
```

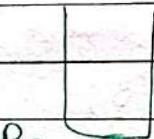
DP-19

PAGE NO.: _____
DATE 30/03/22

0-1 Knapsack

$$n = 3$$

wt.	→	(3)	4	5
val.	→	(30)	50	60



$$5 + 3 = 8 \text{ weight}$$

$$\text{val} = 60 + 30 \\ = 90$$

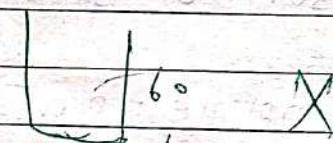
$$4 + 3 = 7$$

$$\text{val} = 50 + 30 = 80$$

① choose 1st costly products

Greedy Not wrong here

$$\begin{aligned} w_1 &\Rightarrow 3 & 2 & 5 \\ \text{val} &\Rightarrow 30 & 40 & 60 \end{aligned}$$



$$w_1 \cdot 2 + 3 = 7$$

$$\text{val} = 40 + 30 = 70$$

law of uniformity was not there
that's why ~~case~~ Greedy fails.

Try out all combination

Take the best value combination

Recursion



→ (i) Express in terms of (index, w)

(ii) Explore all possibilities

(iii) looking for maxm

3	4	5
30	50	60

$f(ind, w)$

if ($ind == 0$)

if ($wf[0] <= w$) return $val[0]$;

else {return 0; }

NOT take = $0 + f(ind-1, w)$

take = INT MIN.

if ($wf[ind] <= w$)

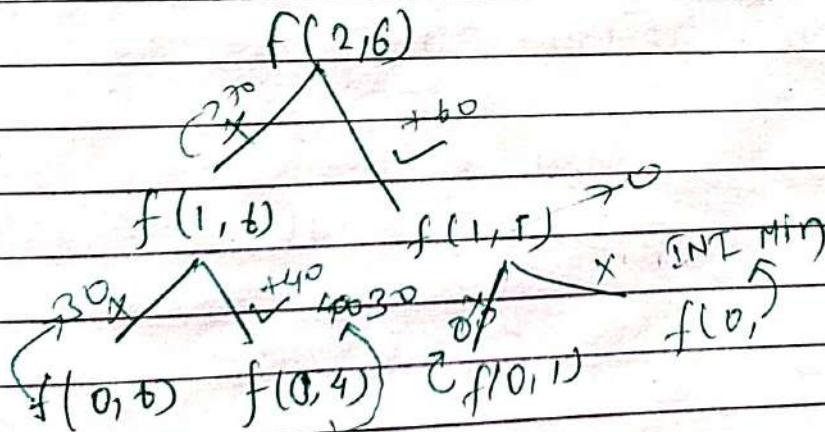
take = $val[ind] + f(ind-1, w-wf[ind])$

return max (not take, take);

}

wt →	6
wt →	3 2 5
val	30 40 60
wt = 6	0

$f(0, w)$



Memoization.

$DP[ind+1][w+1]$

$TC = O(n \times w)$

$SP = O(N \times w) + O(N)$

Total time

@Aashish Kumar Nayak

J@Aashish kuma

Tabulation :-

① Base Case

② Write the changing Parameters
pq, w

③ Loops

No. of loops = No. of changing parameters,
int dp[_p][pq][w];

Base case :

int dp[0][0][0] = 0;

for (i = val[0] → w) dp[0][i] = val[i].

Recursion + Memorization

PAGE NO.: _____
DATE: _____

~~int f(int ind, int w, vector<int>& wt, vector<int>& val, vector<vector<int>>& dp)~~

{ if(ind == 0)

{ if(wt[0] <= w)

{ return val[0];

}

return 0;

if(dp[ind][w] != -1)

{ return dp[ind][w];

}

int notTake = 0 + f(ind-1, w, wt, val, dp);

int take = INT_MIN;

if(wt[ind] <= w)

{

take = val[ind] + f(ind-1, w-wt[ind], wt,

val, dp);

}

return dp[ind][w] = max(take, notTake);

int knapsack(~~vector<int> weight, vector<int> value,~~

~~int n, int w)~~

{

~~vector<vector<int>> dp(n, vector<int>(w+1, -1))~~

return f(n-1, w, weight, value, dp);

3 : @Aashish Kumar Nayak

Tabulation:

```
int knapsack(vector<int> wt, vector<int> val,
int n, int weight)
```

{

```
vector<vector<int>> dp(n, vector<int>(weight+1, 0));
```

```
for (int w=wt[0]; w <= weight; w++)
```

{

```
dp[0][w] = val[0];
```

}

```
for (int ind=1; ind < n; ind++)
```

{

```
for (int w=0; w <= weight; w++)
```

{

```
int notTake = 0 + dp[ind-1][w];
```

```
int take = INT_MIN;
```

```
if (wt[ind] <= w)
```

{

```
take = val[ind] +
```

```
dp[ind-1][w-wt[ind]];
```

}

```
dp[ind][w] = max(take, notTake);
```

}

```
} return dp[n-1][weight];
```

}

Space optimization

PAGE NO.:

DATE:

```
int knapsack( vector<int> val, vector<int> wt, int n, int w)
```

{

```
    vector<int> prev(w+1, 0), cur(w+1, 0);
```

```
    vector<vector<int>> dp(n, vector<int> (maxweight+1, 0));
```

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

```
        for (int j = 0; j <= maxWeight; j++)
```

{

```
            if (wt[i] > j) dp[i][j] = prev[j];
```

{

```
            else dp[i][j] = max(val[i] + prev[j - wt[i]], prev[j]);
```

```
        int take = INT_MIN;
```

```
        if (wt[i] <= w)
```

{

```
            take = val[i] + prev[w - wt[i]];
```

{

```
        curr[w] = max(take, notTake);
```

{

```
        prev = curr;
```

{

```
    return prev[twight];
```

{

Single array space optimization

PAGE NO.:

DATE:

```
int knapsack(vector<int> wt, vector<int> val,  
int n, int maxWeight)
```

{

```
    vector<int> prev(maxWeight + 1, 0);
```

```
    for (int w = wt[0]; w <= weight; w++)
```

{

```
        prev[w] = val[0];
```

}

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

{

```
        for (int w = maxWeight; w >= 0; w--)
```

{

```
            int notTake = 0 + prev[w];
```

```
            int take = INT_MIN;
```

```
            if (wt[i] <= w)
```

{

```
                take = val[i] + prev[w - wt[i]];
```

}

```
            prev[w] = max(take, notTake);
```

}

}

```
return prev[maxWeight];
```

}

Minimum coins | DP on subsequence

arr [] $\rightarrow \{1, 2, 3\}$ target = 7

coins may be $\{3, 3, 1\}$ ans = 3

x $(2, 2, 2, 1)$ ans = 4

minimum coins

Greedy

$\{1, 2, 3\}$ target = 7
 remain = 1
 $7/3 = 2$
 $1/2 = 0$
 $1/1 = 1$

3 coins

Greedy work for this case - but might failed in other case

lets take an example where greedy fails

$\{9, 6, 5, 1\}$ target = 11
 remain = 2
 $11/9 = 1$
 $2/6 = 0$
 $2/5 = 0$
 $2/1 = 2$

ans should be $2+1=3$ but

if we choose 6 & 5 ans = 2
 so greedy fails here.

Approach :-

Trying out all combos to form target

Rules :- take the combo that has min coins

① Express the recurrence in terms of index.

② Express or do all stuff.

③ Min. of all.

Knapsack

Whenever there is a statement like
 if $T < \text{arr}[i]$
 or $i > n$
 it will not go back.

SC: $O(N)$ - recursive

```
f(ind, T)
{
    if(ind == 0)
        if(T == arr[0]) return T[arr[0]];
        else return res;
}
```

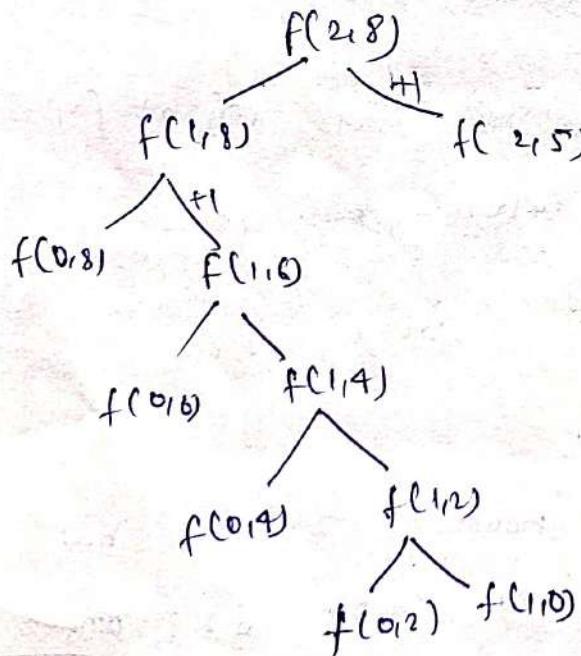
int not_fare = 0 + f(ind-1, T);

int fare = INT_MAX;

```
if(coins[ind] <= T)
    fare = 1 + f(ind, T-coins[ind]);
```

return min(fare, not_fare);

}



T.C. $\rightarrow \gg O(2^n)$

S.C. $\rightarrow \gg O(N)$

Exponential.

Now we will memoize it,

changing parameters \rightarrow ind, target

dp[ind][target] = None

T.C. = $O(N \times T)$

S.C. = $O(N \times T) + O(T)$

Auxiliary space

Tabulation

→ Any memoization can be converted into tabulation.

1. Base case

2. ind

target and write it in opposite fashion.

3. copy the formulae.

for any target

foo($T=0 \rightarrow \text{target}$)

if($T/a[0] == 0$)

$dp[0][T] = T/a[0]$

else

$dp[0][T] = \text{INT_MAX}$

Code :- int f(ind, int T, vector<int> &nums, vector<vector<int>> &dp)

{ if(ind == 0)

{ if($T/a[0] == 0$)

return $T/a[0]$;

return INT_MAX ;

}

int notTake = $0 + f(\text{ind}-1, T, \text{nums}, \text{dp})$;

int take = INT_MAX ;

if($\text{nums}[\text{ind}] \leq T$)

{ take = $1 + f(\text{ind}, T - \text{nums}[\text{ind}], \text{nums}, \text{dp})$;

}

return $\min(\text{take}, \text{notTake})$;

$\rightarrow dp[\text{ind}][T] = \min(\text{take}, \text{notTake})$.

int minimumElements(vector<int> &num, int target)

{ int n = num.size(); vector<vector<int>> dp(n, vector<int>(target+1, -1));

int ans = f(n-1, target, num, dp);

if($\text{ans} \geq \text{INT_MAX}$), return -1;

return ans;

}

Tabulation code =

```
int minimumElements (vector<int> &nums , int target)
{
    int n = nums.size();
    vector<vector<int>> dp (n, vector<int> (target+1, 0));
    for (int T=0; T<=target; T++)
    {
        if (T==0, T<=target, T++)
        {
            if (nums[0] == 0)
                dp[0][T] = T/nums[0];
            else
                dp[0][T] = yes;
        }
    }
    for (int ind=1; ind<n; ind++)
    {
        for (int T=0; T<=target; T++)
        {
            int notTake = 0 + dp[ind-1][T];
            int take = INT_MAX;
            if (nums[ind] <= T)
            {
                take = 1 + dp[ind][T-nums[ind]];
            }
            dp[ind][T] = min (take, notTake);
        }
    }
    int ans = dp[n-1][target];
    if (ans >= 1e9)
        return -1;
    return ans;
}
```

Space Optimization :- (using prev1 & prev2)

```
int minimumElements(vector<int> &nums, int target)
{
    int n = nums.size();
    vector<int> prev(target + 1, 0), cur(target + 1, 0);
    for (int i = 0; i <= target; i++)
    {
        if (nums[0] == 0)
            prev[i] = i / nums[0];
        else
            prev[i] = INT_MAX;
    }
    for (int ind = 1; ind < n; ind++)
    {
        for (int i = 0; i <= target; i++)
        {
            int notTake = 0 + prev[i];
            int take = INT_MAX;
            if (nums[ind] <= i)
            {
                take = 1 + cur[i - nums[ind]];
            }
            cur[i] = min(take, notTake);
        }
        prev = cur;
    }
    int ans = prev[target];
    if (ans >= INT_MAX)
        return -1;
    return ans;
}
```

Target sum problem | DP on subsequence

arr E.g. $\{1, 2, 3, 1\}$

target = 3

Assign +, -

$$+ \quad + \quad - \quad - \\ \underbrace{1 \quad 2 \quad 3 \quad 1}_{= 0} \quad = 3 \quad \text{count all the ways to get target.}$$

$$- \quad + \quad + \quad - \\ \underbrace{1 \quad 2 \quad 3 \quad 1}_{= 3} \quad = 3 \quad \text{2 ways}$$

$$+ \quad - \quad + \quad + \\ \underbrace{1 \quad 2 \quad 3 \quad 4}_{= 3} \quad = 3 \quad \text{3 ways}$$

$$S_1 - S_2 = D$$

sum partition

$$- \quad + \quad + \quad - \\ [1 \quad 2 \quad 3 \quad 1]$$

$$\underbrace{3 + 2}_{S_1} \quad \underbrace{-1 - 1}_{S_2} = 3$$

$$S_1 \quad S_2 \quad = 3$$

same problem like in DP-8

Code: int targetsum(int n, int target, vector<int> &arr)

{ return countpartition(n, target, arr); }

int countpartition(int n, int d, vector<int> &arr)

{ int totsum = 0;

for (auto &it : arr)

 totsum += it;

if (totsum - d < 0) || (totsum - d) % 2

 return false;

 return findwaysarr((totsum - d) / 2);

}

```
int findways(vector<int> num, int tar)
```

```
{ int n = num.size();
vector<int> prev(tar+1, 0), cur(tar+1, 0);
if (num[0] == 0)
    prev[0] = 2;
else
    prev[0] = 1;
if (num[0] != 0 && num[0] <= tar)
    prev[num[0]] = 1;
for (int sum = 0; sum <= tar; sum++)
{
    int notTake = prev[sum];
    int take = 0;
    if (num[0] <= sum)
        take = prev[sum - num[0]];
    cur[sum] = (notTake + take); not;
}
prev = cur;
}
return prev[tar];
}
```

DP-22

Ways to make coin change

N=3

1	2	3
---	---	---

target=4

any element can be used any no. of times.

$$\left. \begin{array}{l} \{1, 1, 1, 1\} = 4 \\ \{1, 1, 2\} = 4 \\ \{2, 2\} = 4 \\ \{1, 3\} = 4 \end{array} \right\} \quad \text{4 differ}$$

figure out total no. of ways \rightarrow Try out all the ways

f()

if destination is met. Recursion

Bare case

else

f(4)

f(1)

return (- + -);

)

Recurrance :

1. Express \rightarrow (ind, T)
2. Explore all possibilities $\xrightarrow{\text{not take}} \xrightarrow{\text{take}}$ By all the DP subsequ. problem.
3. Sum all possibilities and return.

1	2	3
0	1	2

f(2, 4)
ind
T

Tell index 2, in how many ways
can you form 4

```

f(ind, T)
{
    if(ind == 0)
        if(T % arr[0] == 0)
            return 1;
        else
            return 0;
    notTake = f(ind - 1, T);
    take = 0;
    if(arr[ind] <= T)
        take = f(ind, T - arr[ind]);
    return notTake + take;
}

```

Time Complexity: $O(2^n)$

Space Complexity: $O(n)$

Memoization will solve this issue

T.C. = $O(N \times T)$

S.C. = $O(N \times T) + O(\text{target})$

We Tabulation
to optimise A.S.S

Tabulation — Bottom-up approach

1. Basic case
2. ind $\in [1, n]$
 $T \rightarrow (0, T)$
3. Copy the sequence.

for($T \rightarrow 0 \rightarrow \text{target}$)

$dp[0][T] = (T \% arr[0] == 0)$

Code:

```

long countwaysToMakeChange(int denomination, int n, int value)
{
    vector<vector<int>> dp(n, vector<long>(value + 1, -1));
    return f(0, 1, value, denomination, dp);
}

long f(int ind, int target, int value, int denomination, vector<vector<int>> &dp)
{
    if(ind == 0)
        if(target % arr[0] == 0)
            return 1;
        else
            return 0;
    notTake = dp[ind - 1][target];
    take = 0;
    if(arr[ind] <= target)
        take = f(ind, target - arr[ind], value, denomination, dp);
    return notTake + take;
}

```

long f(int ind, int T, int n, vector<long> &dp) :-

if(ind == 0)

return (T % a[0] == 0); // 1 or 0

long notTake = f(ind - 1, T, a, dp);

long take = 0;

if(a[ind] <= T)

take = f(ind, T - a[ind], a, dp);

return (take + notTake);

} \rightarrow dp[ind][T] = take + notTake.

Tabulation :-

long countWaysToRecharge (int a[derivation], int n, int value)

{ vector<vector<long>> dp(n, vector<long> (value + 1, 0)); }

for (int T = 0; T <= value; T++)

{ dp[0][T] = (T % a[0] == 0); }

for (int ind = 1; ind < n; ind++)

{ for (int T = 0; T <= value; T++)

{ long notTake = dp[ind - 1][T]; }

long take = 0;

if(a[ind] <= T)

{ take = dp[ind][T - a[ind]]; }

dp[ind][T] = take + notTake;

return dp[n - 1][value];

Space Optimization

Whenever you see like $dp[i-1]$, $dp[i]$,
then you can always optimise by using couple of
stuffs.

Code :-

```
long countwaysTakes int a, int n, int value
{
    vector<vector<long>> dp(n, vector<long>(value+1, 0));
    vector<vector<
        vector<long> prev(value+1, 0), cur(value+1, 0);
        for (int i = 0; i <= value; i++)
        {
            prev[i] = i % a[0] == 0;
        }
        for (int ind = 1; ind < n; ind++)
        {
            for (int t = 0; t <= value; t++)
            {
                long notTake = prev[t];
                long take = 0;
                if (a[ind] <= t)
                {
                    take = cur[t - a[ind]];
                }
                cur[t] = take + notTake;
            }
            prev = cur;
        }
        return prev[value];
}
```

DP-23

unbounded knapsack

$$wt \rightarrow \{2, 4, 6\}$$

$$val \rightarrow \{5, 11, 13\}$$



But here infinite supply of every item. we can pick any one item many times.

$$\text{Ex: } \begin{array}{l} \xrightarrow{13} \\ \xrightarrow{10} \end{array} \begin{array}{l} wt \rightarrow 6 \\ wt \rightarrow 4 \end{array}$$

$$\begin{array}{l} \xrightarrow{10} \\ \xrightarrow{10} \end{array} \begin{array}{l} wt \rightarrow 2 \\ + 5 \times 2 = 23 \end{array}$$

$$wt \rightarrow 4 \times 2 + wt_2$$

$$11 \times 2 + 5 = \boxed{27} \quad \text{maximum possible value.}$$

$$(wt \rightarrow 2) \times 5 = \textcircled{10} \text{ wt}$$

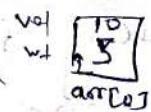
$$5 \times 5 = \textcircled{25} \text{ val}$$

f(ind, w)

```

    if(ind == 0)
        return [wt[0] X val[0]];
    else
        return [wt[0] X val[0]] + f(ind - 1, w);
    
```

base case: we write for 0



$$w=8$$

$$16 \text{ w=2}$$

$$\text{int} \left(\frac{w}{\text{wt}[0]} \right) \times \text{val}[0]$$

$$\text{not take} = 0 + f(\text{ind}-1, w);$$

$$\text{take} = \textcircled{0} \text{ INT_MIN}$$

$$\text{if } \text{wt}[\text{ind}] \leq w$$

$$\text{take} = \text{val}[\text{ind}] + f(\text{ind}, w - \text{wt}[\text{ind}]).$$

return $\max(\text{take}, \text{not take}).$

Memoize

T.C. — exponential

Tabulation

1. Base case

$\gg O(w)$

2. Changing parameter ind.

3. Copy the recurrence.

(Aashish Kumar Nayak)

pre requisite
DP-19 — 0/1 knapsack

In this there was
only one occurrence.

w can be anything 0, 1, 2.

W.B

for (w → 0 to wt[bag])

$$dp[0][w] = \left(\frac{w}{wt[0]} \right) \times val[0]$$

}

Code :- * memoized

```
int f(int ind, int w, vector<int> &val, vector<int> &wt, vector<vector<int>> &dp)
{
    if(ind == 0)
    {
        return ((int)(w / wt[0])) * val[0];
    }
    if(dp[ind][w] != -1) return dp[ind][w];
    int notTake = 0 + f(ind - 1, w, val, wt, dp);
    int take = 0;
    if(wt[ind] <= w)
    {
        take = val[ind] + f(ind, w - wt[ind], val, wt, dp);
    }
    return dp[ind][w] = max(take, notTake);
}
```

```
int unboundedKnapSack(int n, int w, vector<int> &val,
                      vector<int> &wt)
{
    vector<vector<int>> dp(n, vector<int> (w + 1, -1));
    return f(n - 1, w, val, wt, dp);
}
```

Tabulation :-

vec-

```
int unboundedSack(int n, int w, vector<int>& val,
                   vector<int>& wt)
```

```
{ vector<vector<int>> dp(n), vector<int>(wt+1, 0);
```

```
for(int wr=0; wr<=w; wr++)
```

```
{ dp[0][wr] = ((int)(w/wt[0])) * val[0]; }
```

```
for(int ind=1; ind<n; ind++)
```

```
{ for(int wr=0; wr<=w; wr++)
```

```
{ int notTake = 0 + dp[ind-1][wr]; }
```

```
int take = 0;
```

```
if(wt[ind]<=w)
```

```
{ take = val[ind] + dp[ind][w-wt[ind]]; }
```

```
dp[ind][wr] = max(take, notTake); }
```

```
}
```

```
return dp[n-1][w]; }
```

Space Optimization

int i, j so space optimized
is possible.

int unboundedKnapSack(int n, int w, vector<int> &val,
vector<int> &wt)

{

vector<int> prev(w+1, 0), cur(w+1, 0);

for (int w=0; w <= w; w++) remove,

{ prev[w] = ((int)(w/wt[0])) * val[0];
}

for (int ind = 1; ind < n; ind++)

{ for (int w=0; w <= w; w++)

{ int notTake = 0 + prev[w];
int take = 0;

if (wt[ind] <= w)

{ take = Val[ind] + cur[w-wt[ind]];
}

//prev[w-wt[ind]]

cur[w] = max(take, notTake);

//prev[w-wt[ind]]

}

R

prev = cur; //remove this line

{

return prev[w];

}

prev [8 9 5 6 7 1 2]

notTake = 3 * cur

take =

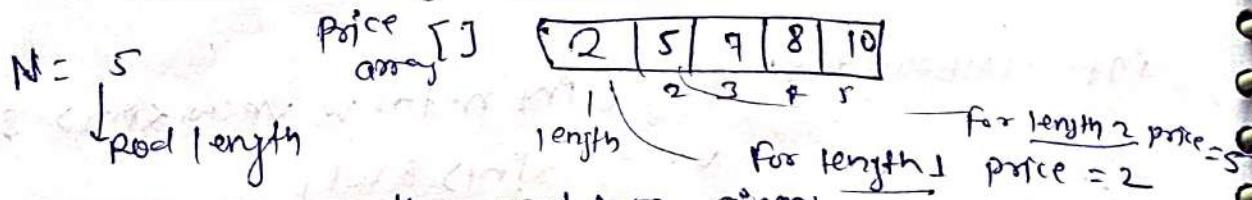
then we re write.

1D array

optimization

DP-24

Rod cutting problem | DP on subsequences



price	2	2	2	2	2	
length	1	1	1	1	1	= 10
	2	3	5			
	0	0	0			= 12 <u>↑ maximum</u>
	2	2	2			
	0	0	0			= 12 <u>maximum</u>

we need to print maximum cost.



Collect rod length to make N

↓

Maximize the price

Similar to knapsack

2	5	7	8	10
0	1	2	3	4

means 5 length
means 4 length

Try to pick lengths and sum them up to make the given N.
in all possible ways.

1. Expressed in terms of index

↳ Recurrence

2. Explore all possibilities

↙ not Take

3. Maximize the price or possibilities.

Take

2	5	7	8	10
0	1	2	3	4

$f(ind, N)$

{ if ($ind == 0$)
 return $N \times \text{price}[0]$;

\leftarrow
 $int take = 0 + f(ind - 1, N);$
 $int \cancel{skip} = INT\ MIN;$
 if ~~rod-length = ind+1:~~

 if ($rod_length \leq N$)

$take = \text{price}[ind] + f(ind, N - \cancel{\text{price}[rod_length]}),$

 return $\max(take, \cancel{skip});$

$f(4, N)$

Till index $\uparrow 4$, what is the minimum price you obtain

$m-1 \rightarrow 0$

✓ $\boxed{8}$ \downarrow
Price [0]
rod-length = 1
 $\{ 1, 1, 1, \dots \}$ \downarrow \times times
 $\cancel{N=12}$
looking for max it.

posting price = $N \times \text{price}[0];$

\nearrow Recursion \rightarrow Memoization

always /
greater than 2^n
i.e. expon

T.C. - $O(\text{Exponentially})$

S.C. $\rightarrow O(\text{Target})$

ind, N

T.C. \downarrow
 $[N] \times [N+1]$

T.C. $\downarrow O(N \times N)$

S.C. $\rightarrow O(N \times N) + O(\text{Target})$
Ans

Memo Tabulation :-

1. Base case

2. Changing parameter

3. Copy recurrence

$ind \rightarrow N$

Base case.

for ($N \rightarrow 0$ to n)

$dp[0][N] = N \times \text{price}[0];$

for $\cancel{ind} \rightarrow 1 \rightarrow n-1$

$N \rightarrow 0 \rightarrow N$

(Copy the recurrence)

```

int f(int ind, int N, vector<int> &price, vector<vector<int>> &dp)
{
    if(ind == 0)
    {
        return N * price[0];
    }
    if(dp[ind][N] != -1) return dp[ind][N];
    int notTake = 0 + f(ind - 1, N, price, dp);
    int take = INT_MIN;
    int rodLength = ind + 1;
    if(rodLength <= N)
    {
        take = price[ind] + f(ind, N - rodLength, price, dp);
    }
    return dp[ind][N] = max(take, notTake);
}

```

```

int cutRod(vector<int> &price, int n)
{
    vector<vector<int>> dp(n, vector<int>(n + 1, -1));
    return f(n - 1, n, price, dp);
}

```

Tabulation :

```

int cutRod (vector<int> &price, int n)
{
    vector<vector<int>> dp(0, vector<int>(n + 1, 0));
    for (int N = 0; N <= n; N++)
    {
        dp[0][N] = N * price[0];
    }
    for (int ind = 1; ind < n; ind++)
    {
        for (int N = 0; N <= n; N++)
        {
            int take = INT_MIN;
            int rodLength = ind + 1;
            if(rodLength <= N)
            {
                take = price[ind] + dp[ind][N - rodLength];
            }
            dp[ind][N] = max(take, dp[ind - 1][N]);
        }
    }
}

```

```

int notTake = 0 + dp[ind-1][N];
int take = INT - MIN;
if (rodLength <= N)
    take = price[ind] + dp[N - rodLength];
dp[ind][N] = max(take, notTake);
}
}
return dp[n-1][n];
}

```

Space Optimization :

```

int cutRod(vector<int> &price, int n)
{
    vector<vector<int>> dp(n, vector<int> (n+1, 0));
    vector<int> prev(n+1, 0), cur(n+1, 0);
    for (int N=0; N<=n; N++)
    {
        prev[N] = N * price[0];
    }
    for (int ind = 1; ind < n; ind++)
    {
        for (int N = 0; N <= n; N++)
        {
            int notTake = 0 + prev[N];
            int take = INT - MIN;
            int rodLength = ind + 1;
            if (rodLength <= N)
                take = price[ind] + cur[N - rodLength];
            cur[N] = max(take, notTake);
        }
    }
    return dp[n-1][n];
}

```

we can do
 1 D array
 optimization
 like previous
 one.

@Aashish Kumar Nayak

DP on Strings

[DP on strings]

Comparison of string

longest common [sub]sequences

replace / edit

$s_1 = "yadebc"$ $s_2 = "dcadb"$

DP-25

Longest Common Subsequence.

$abc \rightarrow [\alpha, b, c, ab, bc, ac, abc, "",]$

Power set / Recursion
or
recursion

We can use these two methods to print all the subset.

$s_1 = "ad\overset{1}{e}\overset{2}{b}c"$ — 2^5 subsequences

a, d, e, b, c, ad, ae, ac, de, db, dc, ade, — adb

$s_2 = "dc\overset{1}{a}\overset{2}{d}b"$ — 2^5 subsequences

d, c, a, d, b, dc, da, dd, db, — adb

We have to find all the subsequences and trace out the longest one.

Brute force

Generate all subsequences

take one & compare

T.C. = O(2^n) (Exponential)

New approach

⇒ Generate all subsequences & compare on way.

↓
Recursion method

We will use parameters to generate them.

© Ashish Kumar Maurya

We will write some recurrence which give the answer through the way.

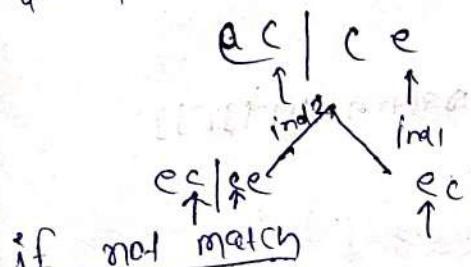
@Aashish Kumar Nayak

Rules to write the recurrence

- ① Express Everything in terms of index.
- ② Explore possibilities on that index.
- ③ Take the best among them.

Let's take an example

$$d = 4$$



$$\begin{array}{c|c} \text{a} & \text{c} & \text{d} \\ \uparrow & \uparrow & \uparrow \\ \text{c} & \text{e} \end{array} \quad \begin{array}{c|c} \text{c} & \text{e} & \text{d} \\ \uparrow & \uparrow & \uparrow \\ \text{e} & \text{d} \end{array}$$

if match if not match

$$1 + f(\text{ind}_1 - 1, \text{ind}_2 - 1)$$

$$+ 1 \rightarrow f(\text{ind}_1 - 1, \text{ind}_2 - 1) \quad \text{if goes -ve}$$

$$0 + \max[f(\text{ind}_1 - 1, \text{ind}_2), f(\text{ind}_1, \text{ind}_2 - 1)]$$

$f(\text{ind}_1, \text{ind}_2)$

-ve means end of the strings.

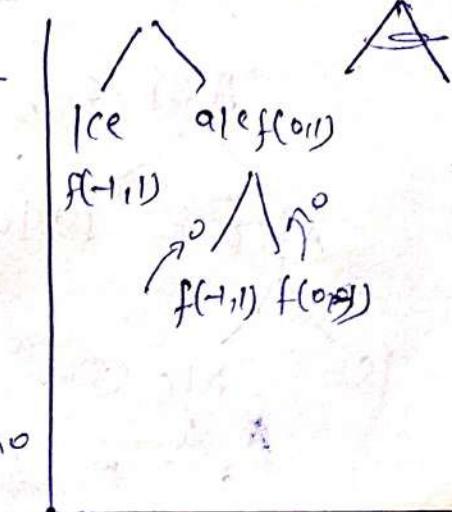
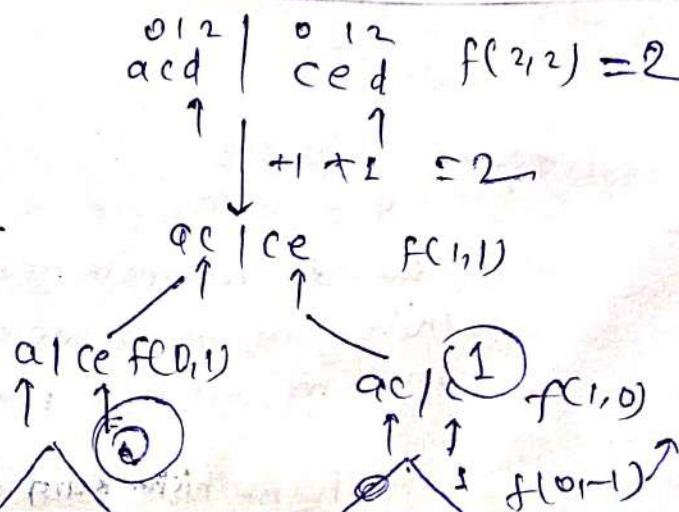
} base case
if ($\text{ind}_1 < 0 \text{ || } \text{ind}_2 < 0$)
return 0;

Match

if ($s_1[\text{ind}_1] == s_2[\text{ind}_2]$)

return $1 + f(\text{ind}_1 - 1, \text{ind}_2 - 1)$.

return $0 + \max[f(\text{ind}_1 - 1, \text{ind}_2), f(\text{ind}_1, \text{ind}_2 - 1)]$;



Recursion $2^n \times 2^m \approx$ Exponential

↓ optimised
if there are overlapping subproblems then apply
memoization.

Code :-

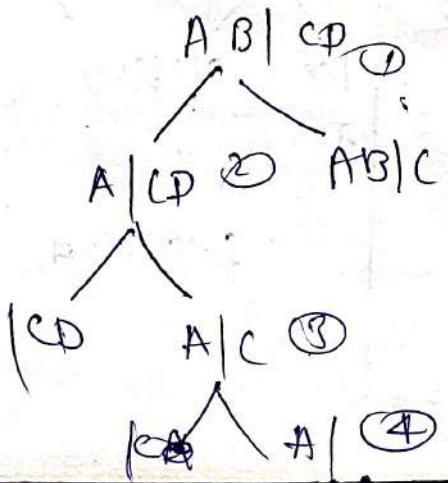
```
int f(int i, int j, string &s, string &t, vector<vector<int>> &dp)
{
    if (i < 0 || j < 0)
        return 0;
    if (dp[i][j] == -1)
        return dp[i][j];
    if (s[i] == t[j])
        return 1 + f(i-1, j-1, s, t);
    else
        return return dp[i][j] = max(f(i-1, j, s, t, dp), f(i, j-1, s, t, dp));
}
```

int lcs(string s, string t)

```
{
    int n = s.size();
    int m = t.size();
    vector<vector<int>> dp(n, vector<int>(m, -1));
    return f(n-1, m-1, s, t, dp);
}
```

T.C. - $O(N \times M)$

S.C. - $O(N \times M) + \frac{O(N+M)}{A.S.S.}$



$2^2 \times 2 = 4$ alternate deletion
If we do alternate deletion then
there can be n deletion from s_1
and m deletion from s_2 .

Tabulation :-

1. copy the base case
2. changing the parameter
3. copy the recurrence.

shifting of indexes.

-1 0 1 2 3 ... n-1

if ($i == 0 \text{ or } j == 0$) return 0
return $\cdot dp[0][j]$

for ($j = 0 \rightarrow m - 1$) $dp[0][j] = 0$

for ($i = 0 \rightarrow n - 1$) $dp[i][0] = 0$

Code :-

```

int lcs(string s, string t)
{
    int n = s.size();
    int m = t.size();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, -1));
    for (int j = 0; j < m; j++) dp[0][j] = 0;
    for (int i = 0; i < n; i++) dp[i][0] = 0;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (s[i - 1] == t[j - 1])
                dp[i][j] = 1 + dp[i - 1][j - 1];
            else
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
    return dp[n][m];
}

```

Space optimization

```
int lcs(string s, string t)
```

```
{ int n = s.size();
```

```
int m = t.size();
```

```
vector<int> prev(m+1, 0), cur(m+1, 0);
```

```
for (int j = 0; j <= m; j++)
```

```
{ prev[j] = 0;
```

```
}
```

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

```
{
```

```
for (int j = 0; j <= m; j++)
```

```
{
```

```
if (s[i-1] == t[j-1])
```

```
cur[j] = i + prev[j-1];
```

```
else
```

```
cur[j] = max(prev[j], cur[j-1]);
```

```
prev = cur;
```

~~```
cur = prev;
```~~

```
return prev[m];
```

```
}
```

## D6-26 print longest common subsequences

$s_1 = "abedc"$      $s_2 = "bdgeek"$

$\text{len}(lcs) = 3$  // in previous problem

$sout = \underline{bde}$

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

move(3,2)

$$dp[s][s] = 3$$

$\downarrow$        $\rightarrow$   
abedc      Bdgeek

come here.  
match

if not match then go to next of  
diagonal  $[i-1][j]$  &  $[i][j-1]$

$$dp[i][j] \rightarrow 2$$

$\downarrow$        $\downarrow$   
abcd      bd

$$dp[i][j] = s + dp[i-1][j-1]$$

only when  $s[i-1] == s_2[j-1]$

$$dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$$

Code  $i=m, j=m$

while( $i>0$  &  $j>0$ )

{ if( $s[i-1] == s_2[j-1]$ )  
 $\downarrow$

else  $dp[i][j] = \max(dp[i-1][j],$

~~else~~  $i = n, j = m$ ;  $\text{len} = \text{dp}[n][m]$ , string  $s = " "$ ;  $\text{index} = \text{len} - 1$   
 while ( $i > 0$  &  $j > 0$ ) for ( $i = 0 \rightarrow \text{len}$ )  $s += "g"$ ; 18/15/5

{ if ( $s_1[i-1] == s_2[j-1]$ )

{  $s[\text{index}] = s_1[i-1]$

$\text{index}--$   
 $\quad i--$ ;  $j--$

}

else if ( $\text{dp}[i-1][j] > \text{dp}[i][j-1]$ )

{

}

else

{

Code :-

void lcs(string s, string t)

{ int n = s.size();

int m = t.size();

vector<vector<int>> dp(n+1, vector<int>(m+1, 0));

for (int i=0; i<=n; i++)

    dp[0][i] = 0;

for (int i=0; i<=m; i++)

    dp[i][0] = 0;

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

{ for (int j=1; j<=m; j++)

{ if ( $s[i-1] == t[j-1]$ )

$\text{dp}[i][j] = 1 + \text{dp}[i-1][j-1]$ ;

else

$\text{dp}[i][j] = \max(\text{dp}[i-1][j], \text{dp}[i][j-1])$ ;

}

}

```
int len = dp[n][m];
```

```
String ans = "";
```

```
for (int i=0; i<len; i++)
 ans += '$';
```

```
int index = len - i;
```

```
int i=n, j=m;
```

```
while (i>0 && j>0)
```

```
{ if (s[i-1] == t[j-1])
```

```
{ ans[index] = s[i-1];
```

```
index--;
```

```
i--; j--;
```

```
}
```

```
else if (dp[i-1][j] > dp[i][j-1])
```

```
{ i--;
```

```
}
```

```
else j--;
```

```
}
```

```
cout << ans;
```

T.C.  $O(m+n)$

```
int main()
```

```
{ string s1 = "abcde";
```

```
string s2 = "badgeek";
```

```
lcs(s1, s2);
```

```
}
```

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

DP-27

## longest common Substring :-

$$S_1 = "ab\boxed{jk}lp"$$

$$S_2 = "a\boxed{kj}lp"$$

should be consecutive

Let's } matching

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

not matching

$$dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$$

This will  
not work  
on substring

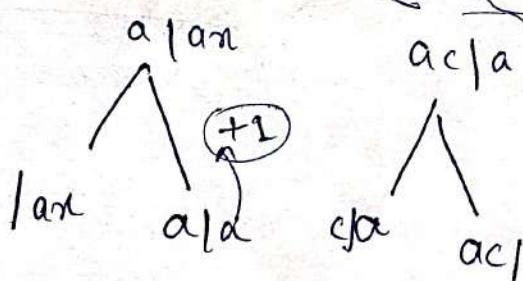
lets take an example

breaking the  
consecutive rule

acd | axd

+1

aclax



This is consecutive so we  
aren't dependent on the  
previous guys.

no need to do a merge  
directly put 0

$$dp[i][j] = 0$$

abcd

abzd

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | a | b | z | d |
| 0 | 1 | 2 | 3 | 4 |

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

The maximum value  
in the entire matrix

Code

```

int lcs(string s, string t)
{
 int n = s.size();
 int m = t.size();
 vector<vector<int>> dp(m+1, vector<int>(n+1, 0));
 for (int i = 0; i <= m; i++)
 dp[0][i] = 0;
 for (int j = 0; j <= n; j++)
 dp[j][0] = 0;
 int ans = 0;
 for (int i = 1; i <= m; i++)
 {
 for (int j = 1; j <= n; j++)
 {
 if (s[i-1] == t[j-1])
 {
 dp[i][j] = 1 + dp[i-1][j-1];
 ans = max(ans, dp[i][j]);
 }
 else
 dp[i][j] = 0;
 }
 }
 return ans;
}

```

## Space optimization :-

Code :-

```
int lcs(string &s, string &t)
{
 int n = s.size();
 int m = t.size();
 vector<int> prev(m+1, 0), cur(m+1, 0);
 int ans = 0;
 for(int i = 0; i < n; i++)
 {
 for(int j = 0; j < m; j++)
 {
 if(s[i] == t[j])
 cur[j] = 1 + prev[j];
 else
 cur[j] = 0;
 ans = max(ans, cur[j]);
 }
 prev = cur;
 }
 return ans;
}
```

DP-28

## longest palindromic subsequences

$S = "bbbab"$

→ ab  
→ bb  
→ bbb  
→ bb bb  
→ bab

longest one

length = 4

① Brute force

Generate all the subsequences

Check for palindrome & pick up the longest.

$S_1 = "babcbcbab"$

$S_2 = "bacbcbaabb"$  — reverse of  $S_1$

Just find  $\text{lcs}(S_1, \text{rev}(S_1))$

Code =

```

int lcs(string s, string t)
{
 int n = s.size();
 int m = t.size();
 vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
 for(int i=0; i<=n; i++) dp[i][0] = 0;
 for(int i=0; i<n; i++) dp[0][i] = 0;
 for(int i=1; i<=n; i++)
 {
 for(int j=1; j<=m; j++)
 {
 if(s[i-1] == t[j-1])
 dp[i][j] = 1 + dp[i-1][j-1];
 else
 dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
 }
 }
 return dp[n][m];
}

```

int longestPalindromeSubsequence(string s)

{ string t = s;

reverse(t.begin(), t.end());

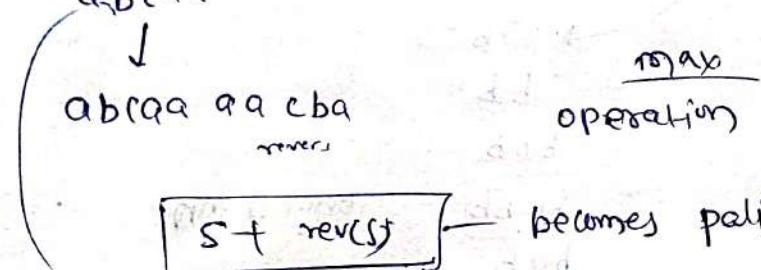
return lcs(s, t);

DP-29

## Minimum insertions to make a string palindrome

prerequisite → LCS  
→ LPS

$S = "abca"$



max  
operation =  $\text{len}(S)$

a b c b a → palindrome and min operation ≈

Approach:

keep the palindrome portion intact.

- $S = \underline{abc}aa$  → a lot of palindrome so I will try to keep longest palindrome intact.

a b c a a      we will match. If not match then we will place copy of that in right part.

coding interview

$\pi$  → longest palindromic subsequence

size of string.

Code :-

```
int minInsertions(string str)
{
 return str.size() - longestPalindromeSubsequence(str);
}

int longestPalindromeSubsequence(string s)
{
 string t = s;
 reverse(t.begin(), t.end());
 return LCS(s, t);
}

int LCS(string s, string t)
{
 int n = s.size();
 int m = t.size();
 vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
 for(int i=0; i<n; i++) dp[0][i] = 0;
 for(int j=0; j<m; j++) dp[i][0] = 0;
 for(int i=1; i<n; i++)
 {
 for(int j=1; j<m; j++)
 {
 if(s[i-1] == t[j-1])
 dp[i][j] = 1 + dp[i-1][j-1];
 else
 dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
 }
 }
 return dp[n][m];
}
```

DP-80

Minimum insertion/deletion to convert string A to string B

$\text{str1} \rightarrow \text{"abcd"} \quad \text{str2} = \text{"anc"}$

Delete everything from str1 and insert all characters from str2 to str1.

it takes  $\underline{\text{str1.size() + str2.size}}$  operations.

intuition

what can I not touch

a b c d  
x x → anc  
lets delete b c d

a c — 2 deletion  
now insert n

am-c — 1 insertion

longest common subsequences

deletions =  $n - \text{len(lcs)}$

insertions =  $m - \text{len(lcs)}$

$n+m - 2 \times \text{len(lcs)}$  → Ans.

Code

int canyoumake (string &str, string &ptr)

{  
return str.size() + ptr.size() - 2 \* lcs(str, ptr);  
}

int lcs(string s, string t)

{ int n = s.size();

int m = t.size();

```
vec<vector<int>> dp(n+1, vector<int>(m+1, 0));
```

```
for(int j=0; j<=m; j++)
 dp[0][j] = 0;
```

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

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

```
{ for(int j=1; j<=m; j++)
```

```
{ if(s[i-1] == t[j-1])
```

```
 dp[i][j] = 1 + dp[i-1][j-1];
```

```
else
```

```
 dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
```

```
}
```

```
return dp[n][m];
```

```
}
```

DP-31

## Shortest Common Supersequence :

$S_1 = \text{"brute"} \quad S_2 = \text{"goat"}$

both string should be present in the final string.  
and order had to be same.

"~~brute~~ "  $\underline{\text{bgoat}}$  " — len = 8

$S_1 = \text{"bleed"} \quad S_2 = \text{"blue"}$

"bleed"

fig. out lengths

print ≠

Common guys should be taken. Once  
common guy in two string means

lcs ( , )

it tells me longest  
common guys

"bgoate"

(S<sub>1</sub>) (S<sub>2</sub>)

ble (bt) goo

(n fm) → len(lcs)

print string

$S_1[i-1] == t[j-1]$

$dp[i][j] = l + dp$

we will create dp table.

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

if not match  $\max(1, 1) = 1$

common guys get added by 1

et0u09rbg

reverse it

gbrouofe

len(8)

i=n, j=m

while( i>0 && j>0 )

{ if( s1[i-1] == s2[j-1] )

{ ans += s1[i-1];  
i--, j--;

{ else if( dp[i-1] > dp[i][j-1] )

{ ans += s1[i-1];  
i--;

else

{ ans += s2[j-1];  
j--;

$i > 0$

String  $s_1$   
has some length

$j > 0$

$s_2$  has  
some length

@Aashish Kumar Nayak

while ( $i > 0$ )  $ans += s_1[i-1]; i--$

while ( $j > 0$ )  $ans += s_2[j-1], j--$

Code :-

string shortestSupersequence(string s, string t)

{ int n = s.size();

int m = t.size();

vector<vector<int>> dp(m+1, vector<int>(n+1, 0));

for (int j = 0; j <= m; j++)

dp[0][j] = 0;

for (int i = 0; i <= n; i++)

dp[i][0] = 0;

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

{ for (int j = 1; j <= m; j++)

{ if (s[i-1] == t[j-1])

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

else

dp[i][j] = max(dp[i-1][j], dp[i][j-1]);

}

String ans = "";

int i = n, j = m;

while (i > 0 && j > 0)

{ if (s[i-1] == t[j-1])

{ i--, j--;

}

else if (dp[i-1][j] > dp[i][j-1])

```
{
 ans += s[i-1];
 i--;
```

```
}
else
{
```

```
 ans += t[j-1];
 j--;
```

```
}
```

```
while (i > 0)
```

```
{
 ans += s[i-1];
 i--;
```

```
}
```

```
while (j > 0)
```

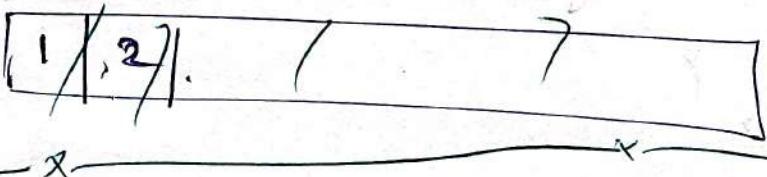
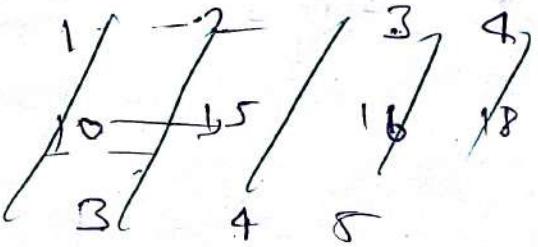
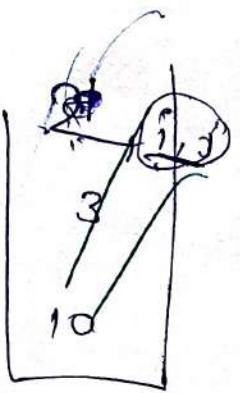
```
{
 ans += t[j-1];
 j--;
```

```
}
```

```
reverse(ans.begin(), ans.end());
```

```
return ans;
```

```
}
```



DP on  
Distinct subsequences

$s = \text{"babgbag"} \quad s_2 = \text{"bag"}$  occurrence of  $s_2$  in  $s_1$  in  
babgbag  $\rightarrow$  5 ways distinct ways

Different methods of comparing

Trying all the ways  
→ Recursion

Count the no. of ways

f1()

↓  
Base case f1() go

// Base case will return  
for 0 : count ways.

f1()

f2()

return f1() + f2();

How to write recursion

- (i) Express everything in terms of i & j
- (ii) Explore all possibilities
- (iii) Count always return the summation of all possibilities
- (iv) Base case

$f(i, j)$   
 $s_1 = \underline{babg} \underline{bag} \underline{bag}$  (i)  $n-1$   
 $s_2 = \underline{bag}$   
 $m$   
 $m-1$   
 If  $i < 0$ :  
 $f(m-1, m-1)$   
 $f(i-1, i)$  return 1;  
 $f(i-1, i)$  return 0;  
 start call  
 no. of distinct sub sequences of  $s_2[0 \dots j]$   
 $i \in s_1[0 \dots j]$   
 $\text{if } s_1[i] == s_2[j] \text{ if they are matching}$   
 $\text{return } f(i-1, j-1) + f(i-1, j);$  to ba from bag  
 $\text{else}$   
 $\text{return } f(i-1, j) \text{ Reduced and look for someone else.}$

### Space Complexity:

$T.C \rightarrow \text{exponential}$   
 $S.C \rightarrow O(N+m)$   
 $s_1 \uparrow \quad s_2 \uparrow$   
 $2^n \quad 2^m$   
 Optimise → overlapping subproblems  
 Memoization  
 $T.C. \rightarrow O(N \times m)$   
 $S.C. \rightarrow O(N \times m) + O(N+m)$

@Aashish Kumar Nayak

### Code:

```

Pnf(int i, int j, string s, string t, vector<vector<int>> &dp)
{
 if(j < 0)
 return 1;
 if(i < 0)
 return 0;
 if(dp[i][j] != -1)
 return dp[i][j];
 if(s[i] == t[j])
 return f(i-1, j-1) + f(i-1, j);
 return f(i-1, j-1, s, t, dp) + f(i-1, j, s, t, dp);
}
return dp[i][j] = f(i-1, j, s, t, dp);

```

3 Pnf numDistinct(string s, string t)

```

{ int n = s.size(); int m = t.size();
 vector<vector<int>> dp(m, vector<int>(n+1));
 return f(n-1, m-1, s, t, dp);
}

```

## Tabulation

- (i) declare the dp array of same size.
- (ii) changing parameter by opposite fashion.

Base case  $j = 0$   
 no string  
 $s_2$   
 $j = 0$   
 for ( $i = 0 \rightarrow n$ )  
 $dp[i][0] = 1$

1-based indexing instead of  
 0-based indexing in order  
 to avoid -1, negative

## Code :-

```

int numofstrict(string s, string t)
{
 int n = s.size();
 int m = t.size();
 vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
 for (int i=0; i<n; i++)
 dp[i][0] = 1;
 for (int j=0; j<m; j++)
 dp[0][j] = 0;
 for (int i=s; i<n; i++)
 for (int j=t; j<m; j++)
 if (s[i] == t[j])
 dp[i][j] = dp[i-1][j-1] + dp[i-1][j];
 else
 dp[i][j] = f(i-1, s, s, t, dp);
 return dp[n][m];
}

```

## Space Optimization

```

int numDistinct(string s, string t)
{
 int n = s.size();
 int m = t.size();
 vector<double> prev(m+1, 0), cur(m+1, 0);
 prev[0] = cur[0] = 1;
 for(int i=1; i<n; i++)
 {
 for(int j=1; j<=m; j++)
 {
 if(s[i-1] == t[j-1])
 cur[j] = prev[j-1] + prev[j];
 else
 cur[j] = prev[j];
 }
 prev = cur;
 }
 return (int) prev[m];
}

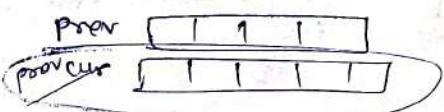
```

## single array optimization

$$i = 1 \rightarrow n$$

$$j = 1 \rightarrow m$$

if()  $\rightarrow$  cur[j] = prev[j-1] + prev[j]  
 Else  $\rightarrow$  cur[j] = prev[j]



```

int numDistinct(string s, string t)
{
 int n = s.size();
 int m = t.size();
 vector<double> dp(m+1, 0);
 dp[0] = 1;
 for(int i=1; i<n; i++)
 {
 for(int j=m; j>=1; j--)
 {
 if(s[i-1] == t[j-1])
 dp[j] = dp[j-1] + dp[j];
 }
 }
 return (int) dp[m];
}

```

DP-33

## Edit Distance

$s_1 = \text{"horse"} \quad s_2 = \text{"ros"}$

1. insert
2. Remove
3. Replace

The minimum no. of steps by which we can convert  $s_1$  into  $s_2$ .

(1) ~~Br~~ method delete all characters from  $s_1$  and then insert all characters from  $s_2$  to  $s_1$ .

it will take  $\underline{\text{len}(s_1) + \text{len}(s_2)}$  steps

(2) ~~method~~

replace

$s_1 = \text{"horse"} \quad s_2 = \text{"ros"}$

- replace h with r
- remove r
- remove e



It took 3 steps.

Example

$s_1 = \text{"intention"} \quad s_2 = \text{"execution"}$

remove i

intention

replace e  $\rightarrow$  e

execution

replace n  $\rightarrow$  n

execution

insert u

execution

5 Operations

$s_1$   
String matching  
using recursion

horse  
ros

match - ok  
not match then delete  
two case  
not match then replace  
or  
insert the same char

→ if not matching

→ insert of the same character

→ delete & try finding somewhere else

→ replace & match.

Try all ways  $\rightarrow$  recursion

@Aashish Kumar Nayak

How to write recurrence?

1. Express in terms of  $(i, j)$ .
2. Explore all paths of matching.
3. Return  $\min(\text{all paths})$ .
4. Base case.

$f(i, j)$

horses  
ros

$f(n-1, m-1) \rightarrow \min$  operation or  
to convert.

$f(i, j)$

{ if ( $s_1[i] == s_2[j]$ )

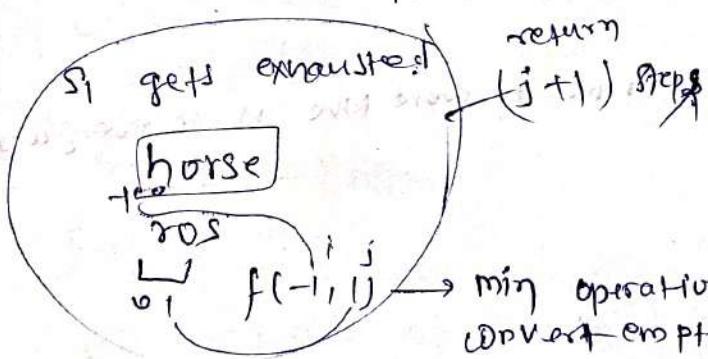
$s_1[0 \dots i] + s_2[0 \dots j]$

} return 0 +  $f(i-1, j-1)$ . // If they are matching shrink both  
string and move don't require  
any steps.

+  $f(i, j-1)$ ; // insert      // If not matching i can insert

+  $f(i-1, j)$ ; // delete

+  $f(i-1, j-1)$ ; // replace



min operations to  
convert empty string  
to a  $s_2[0 \dots -1]$

if ( $i < 0$ ) return  $j+1$

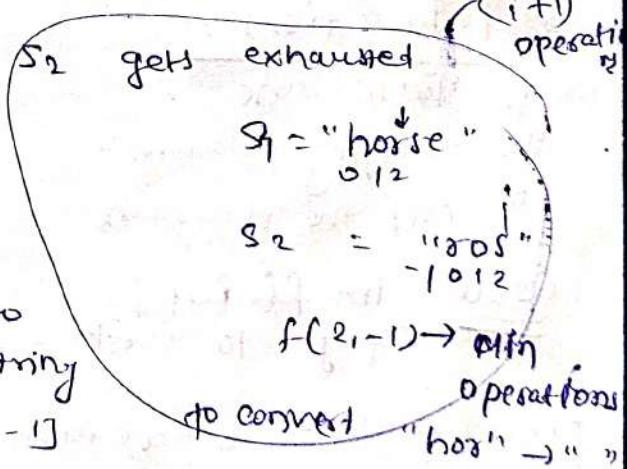
if ( $j < 0$ ) return  $i+1$  & insert

Recursion

TC  $\rightarrow$  Exponential

SC  $\rightarrow$   $O(N+m)$

overlapping subproblem  $\rightarrow$  memorize



After memorization

T.C.  $\rightarrow O(N \times M)$

S.C.  $\rightarrow O(N \times M) + O(N + M)$

int editdistance(string &s1, string &s2, vector<vector<int>> dp)

```
{ if(i<0) return j+1; // if(i==0) return j+1;
 if(j<0) return i+1; // if(j==0) return i;
 if(dp[i][j] != -1) return dp[i][j];
 if(s1[i] == s2[j]) return f(i-1, j-1, s1, s2);
 // s1[i-1] == s2[j-1]
 return dp[i][j] = min(f(i-1, j, s1, s2, dp), min(f(i, j-1, s1, s2, dp), f(i, j-1, s1, s2, dp)));
}
```

int editdistance(string str1, string str2)

```
{ int n = str1.size();
 int m = str2.size();
 vector<vector<int>> dp(n, vector<int>(m, -1));
 return f(n-1, m-1, str1, str2, dp);
}
```

γ

Tabulation :-

1. Base case → In order to avoid time +1 to everywhere
2. changing parameters.
3. copy the recurrence

Code :- int f( int i, int j )  
try to write Base case.

f(0, j) i == 0 return j — Here j can be anything.

Means for( j = 0 → m ) dp[0][j] = j; //Base case I

j == 0 return i

for( i = 0 → n ) dp[i][0] = i;

2.  $i = 1 \rightarrow n$

$j = 1 \rightarrow m$

3. copy the recurrence.

```
int editDPS(string s1, string s2)
{
 int n = s1.size();
 int m = s2.size();
 vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
 for (int i = 0; i <= n; i++) dp[i][0] = i;
 for (int j = 0; j <= m; j++) dp[0][j] = j;

 for (int i = 1; i <= n; i++)
 for (int j = 1; j <= m; j++)
 {
 if (s1[i-1] == s2[j-1])
 return dp[i][j] = dp[i-1][j-1];
 else
 dp[i][j] = 1 + min(dp[i-1][j], min(dp[i][j-1], dp[i-1][j-1]));
 }
 return dp[n][m];
}
```

Note space optimization

We can definitely optimise the space because there  $i-1$ , something like that is involved.

Bottom

Base Cases

$$dp[0][j] = j$$

$$dp[i][0] = i$$

can be written in single array

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | - | j |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |

can be written in single array.

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

int editDistance(string str1, string str2)

{ int n = str1.size(); }

int m = str2.size();

for (vector<

vector<int> prev(m+1, 0), cur(m+1, 0);

for (int j=0; j <= m; j++)

prev[j] = j;

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

{ cur[0] = i;

for (int j=1; j <= m; j++)

if (str1[i-1] == str2[j-1])

cur[j] = prev[j-1];

else

cur[j] = 1 + min( prev[j],  
min( cur[j-1],  
prev[j-1]));

}

prev = cur;

return prev[m];

}

## DP - 34 10. Wildcard Matching | DP on string (last problem)

$s_1 = "gday"$

$s_2 = "gflag"$    
 All three chars  
 are matching  
 return true

? → matches with single char.

\* → matches with sequence of  
length 0 or more.

$s_1 = "ab*cd"$

$s_2 = "abdefd"$

return true

$s_1 = "**abefd"$

$s_2 = "abefd"$

return true

$s_1 = "ab*g*d"$

$s_2 = "ab*d*$

return false

1<sup>st</sup> thing which comes in remain  
string matching

problem faced

we can do using recurrence

ab\*ef  
abdef\*

for this star we don't know  
how much of length of other  
string it will cover

ten ~

/ /  
1 2

means recursion

so we will try string matching by recursion.  
Rules to write Recurrence:

1. Express  $(i, j)$

2. explore Comparisons

3. out of all comparisons, if anyone can

be return true

$f(m-1, n-1) - \boxed{\text{Amrit Kumar Nagarkar}}$

$s_1[0 \dots 4]$   
 $s_2[0 \dots 6]$

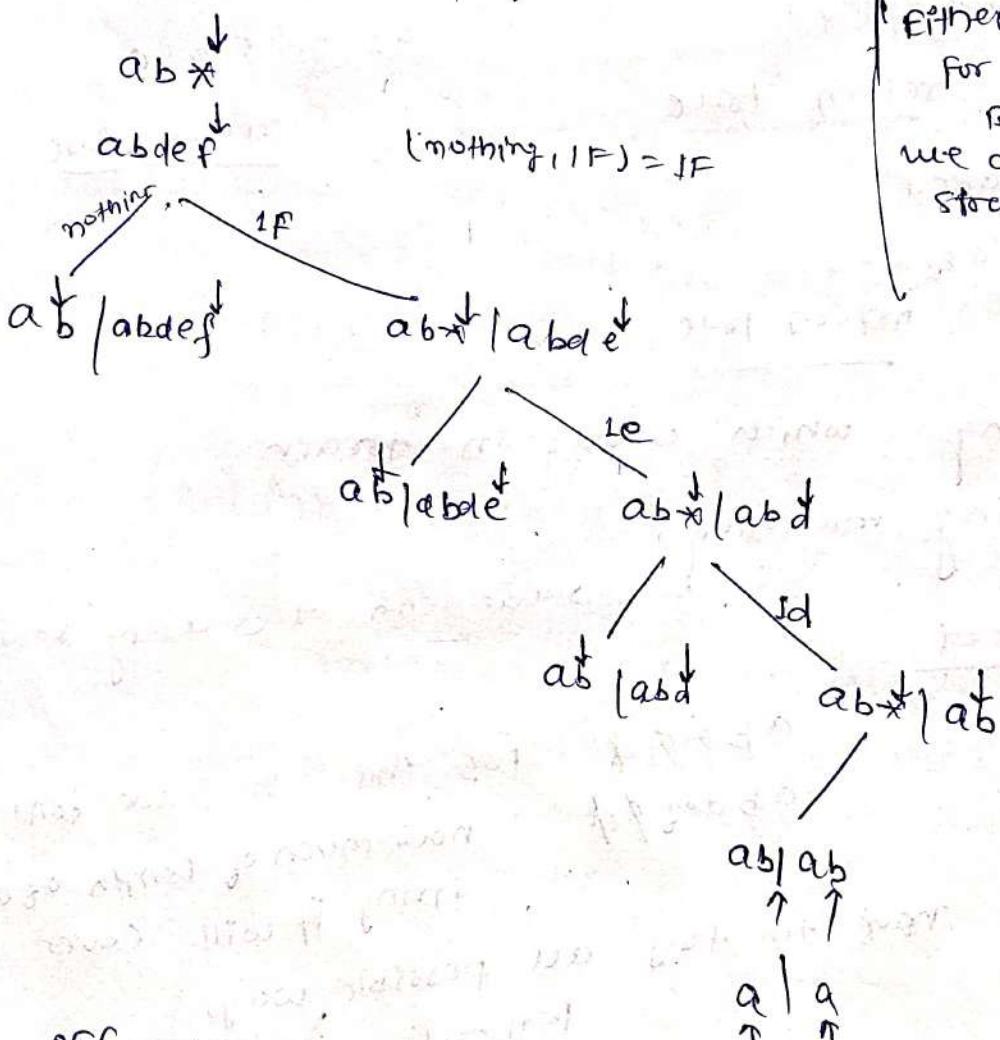
$f(m-1, m-1) \rightarrow$   
 $f(4, 6) \rightarrow T/F$

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| a | b | * | c | d |
| a | b | c | e | f |
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 |   |   |   |

if ( $s_1[i] == s_2[j]$ ) ||  $s_1[i] == ?$ )

return  $f(i-1, j-1)$

either we can write  
 for loop,  
 but  
 we can do this in  
 straight forward  
 recursion



if ( $s_1[i] == ?$ )

return  $f(i-1, j)$  or  $f(i, j-1)$

return false;

Now Analyze the Base case

Base case  $\rightarrow$  return true/false

Either  $s_1$  gets exhausted

if ( $i < 0 \text{ or } s_2 < 0$ ) return true

$s_1 \rightarrow$  has no option  
 $s_2 \rightarrow$  has no exhausted  
return true

if ( $i < 0 \& j = 0$ )  
only  $s_1$  is get exhausted  
return false

if ( $s_2 < 0 \& i >= 0$ ) — only  $s_2$  is yet exhausted.

if  $s_1$  is left it has to be all "\*"

for ( $i = 0 \rightarrow i$ )

if ( $s_1[i] = '*'$ )

return false

}  $f(i, j)$  return true.

if ( $i < 0 \text{ or } j < 0$ ) return true

if ( $i < 0 \text{ and } j > 0$ ) return false

if ( $j < 0 \text{ and } i > 0$ ) for ( $0 \leq p \leq m$ )

### Recursion

$\Rightarrow$  T.C. =  $O(\text{Exponential})$

S.C. =  $O(N + M)$

Auxiliary Stack Space



After memoization

T.C. =  $O(N \times m)$

S.C. =  $O(N \times m) + O(N + m)$

1  
Ans

if (int i, int j; string &pattern, string &text, vector<vector<int>> &dp)

{  
i == 0 & j == 0 if (i < 0 || j < 0) return true;

i == 0 & j > 0 if (i < 0 || j >= 0) return false;

j == 0 & i > 0 if (j < 0 || i >= 0)

{  
for (int ii = 0; ii <= i; ii++)

{  
if (pattern[ii] != '\*' )  
return false;

return true;

if (dp[i][j] != -1) return dp[i][j];

{  
if (pattern[i] == text[j] || pattern[i] == '?')  
return dp[i][j] = f(i-1, j-1, pattern, text, dp);

if (pattern[i] == '\*')

{  
return dp[i][j] = f(i-1, j, pattern, text, dp);

f(i, j-1, pattern, text, dp);

}  
return dp[i][j] = false;

bool wildMatching (const string pattern, const string text)  
{ int m = pattern.size();

int n = text.size();

vector<vector<int>> dp(m, vector<int> (n, -1));

return f(n-1, m-1, pattern, text, dp);

## Tabulation :-

- ① Base case
- ② changing the parameter
- ③ copy the recurrence

[@Aashish Kumar Nayak]

1st convert it into 1-based indexing.  
by just increasing +1 everywhere

```
bool wildcardMatching(string pattern, string text)
{
 int n = pattern.size();
 int m = text.size();
 vector<vector<bool>> dp(n+1, vector<bool>(m+1, false));
 dp[0][0] = true;

 for (int i=1; i<=n; i++)
 {
 dp[i][0] = false;
 }

 for (int i=1; i<=m; i++)
 {
 bool flag = true;
 for (int ii=1; ii<=i; ii++)
 {
 if (pattern[ii-1] != '*')
 flag = false;
 else
 break;
 }
 dp[0][i] = flag;
 }

 for (int i=1; i<=n; i++)
 {
 for (int j=1; j<=m; j++)
 {
 if (pattern[i-1] == text[j-1] || pattern[i-1] == '?')
 dp[i][j] = dp[i-1][j-1];
 else if (pattern[i-1] == '*')
 dp[i][j] = dp[i-1][j] | dp[i][j-1];
 }
 }
}
```

```

 else
 dp[i][j] = false;
 }
}

return dp[n][m];

```

### Space optimization :-

```

bool wildcardMatching(string pattern, string text)
{
 int n = pattern.size();
 int m = text.size();

 vector<bool> prev(m+1, false), cur(m+1, false);
 prev[0] = true;

 for (int j = 1; j <= n; j++)
 {
 prev[j] = false;
 }

 for (int i = 1; i <= n; i++)
 {
 bool flag = true;
 for (int ii = 0; ii < i; ii++)
 {
 if (pattern[ii] != '*')
 {
 flag = false;
 break;
 }
 }

 if (dp[i-1][0] == flag)
 cur[0] = flag;
 for (int j = 1; j <= m; j++)
 {
 if (pattern[i-1] == text[j-1] || pattern[i-1] == '?')
 dp[i][j] = cur[j-1];
 else
 dp[i][j] = false;
 }
 }
}

```

```
cur[0] = flag;
for (int i=1; i<=m; i++)
{ if (pattern[i-1] == text[i-1] || pattern[i-1] == '?')
 { cur[i] = prev[i-1];
 }
 else if (pattern[i-1] == '*')
 { cur[i] = prev[i] | cur[i-1];
 }
 else
 { cur[i] = false;
 }
 prev = cur;
}
return prev[m];
}
```

DP-35

## Best time to buy and sell stocks DP on Stocks

6 problems

$$arr[ ] = \{ \begin{matrix} 1^{\text{st}} \text{ day} \\ \text{price} \end{matrix}, \begin{matrix} 2^{\text{nd}} \text{ day} \\ \text{price} \end{matrix}, \begin{matrix} 3^{\text{rd}} \text{ day} \\ \text{price} \end{matrix} \} = \{ 7, 1, 5, 3, 6, 4 \}, n=6$$

only  
done  
once

Buy → 1      buying has to be done before selling.  
sell → 6

if we are selling on  $i^{\text{th}}$  day then we buy on the minimum price from  $1^{\text{st}} \rightarrow (i-1)$

$$\overbrace{\{ 7, 1, 5, 3, 6, 4 \}}^{mi}$$

keep track of minimal on left

$$mini = arr[0], profit = 0$$

for ( $i=1$ ;  $i < n$ ;  $i++$ )

cost =  $arr[i] - mini$ ;

$$profit = \max(profit, cost);$$

$$mini = \min(mini, arr[i]);$$

for 4 min on left is 1 so profit = 3

for 6 min on left is 1 so profit = 5

for 3 ————— 1 ————— 2

for 5 ————— 1 ————— 4

for 1 ————— 7 ————— 6

max = 5, Ans

Code:

int maximumProfit(vector<int> &prices)

{ int mini = prices[0];

int maxProfit = 0;

int n = prices.size();

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

{ int cost = prices[i] - mini;

maxProfit = max(maxProfit, cost);

mini = min(mini, prices[i]);

} return maxProfit;

T.C. = O(N)

S.C. = O(1)

# DP-3b Buy and sell stocks - II | Recursion to space optimization

## Q Best time to Buy & sell stocks - II

In this question we can buy and sell many number of times.

$$\text{prices}[] = \{ 7, 1, 5, 3, 6, 4 \}$$

$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow$

$$B \ S \quad B \ S$$

$4 + 3 = 7$

Buy and sell has to be done before next buy.  
i.e. Before buying a stock previous stock should be sold and maximise the profit

7, 1, 5, 3, 6, 4  
 $\uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow$

Buy  
not buy      sell  
not sell

B S    B S    B S    B S    B S    B S

A lot of ways

### Rule for recurrence

① Express everything in terms of index  
(ind, buy)

↓  
Try all ways

② Explore possibilities on that day

↓  
Recursion

③ Take the max all profits

↓  
Best answer

f(ind, buy)

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 7 | 1 | 5 | 3 | 6 | 4 |

f(0, 1)

→ Start on 0th day with buy, what

max profit?

find (ind, buy)

~~if~~ if (buy) max  
 profit =  $\{ \text{price}[i] + f(i+1, 0) \}$  take  
 else  $\{ 0 + f(i+1, 1) \}$  not take  
 sell → profit =  $\{ \text{price}[i] + f(i+1, 1); \}$   
 max  $\{ 0 + f(i+1, 0); \}$   
 return profit;  
 }

### Base case

when we exhausted or complete the array

```
if (ind == n)
 return 0;
```

T.C. → ~~O(exponential 2^n)~~

T.C. -  $O(2^n)$

S.C. -  $O(N)$

There is overlapping subproblem

so apply memoization.

dp[i][buy]

T.C. -  $O(1)$

```

int f(int ind, int buy, long values, int n, vector<vector<long>> &dp)
{
 if(ind == n)
 return 0;
 int profit = 0;
 if(buy)
 {
 profit = max(-values[ind] + f(ind+1, 0),
 0 + f(ind+1, 1, values, n, dp));
 }
 else
 {
 profit = max(values[ind] + f(ind+1, 1, values, n, dp),
 0 + f(ind+1, 0, values, n, dp));
 }
 return profit + dp[ind][buy] = profit;
}

```

```

long getMaximumProfit (long &values, int n)
{
 vector<vector<long>> dp(n, vector<long>(2, -1));
 return f(0, 1, values, n, dp);
}

```

### Tabulation :

1. Base Case

2.  $i \rightarrow (n-1 \rightarrow 0)$   
Buy 0  $\rightarrow$  1

3. copy the recurrence.

$$\begin{aligned} T.C. &= O(N \times 2) \\ S.C. &= O(N \times 2) + O(N) \end{aligned}$$

~~dp[n+1]~~

```
long getmaximumprofit(long *values, int n)
{
 vector<vector<long>> dp(n+1, vector<long>(2, 0));
 dp[0][0] = dp[0][1] = 0;
 for(int ind = n-1; ind >= 0; ind--)
 {
 for(int buy = 0; buy <= 1; buy++)
 {
 long profit = 0;
 if(buy)
 {
 profit = max(values[ind] + dp[ind+1][0],
 0 + dp[ind+1][1]);
 }
 else
 {
 profit = max(values[ind] + dp[ind+1][1],
 0 + dp[ind+1][0]);
 }
 dp[ind][buy] = profit;
 }
 }
 return dp[0][1];
}
```

## Space optimization :-

```
long getmaximumprofit(long *values, int n)
```

```
{ vector<long> ahead(2,0), cur(2,0);
```

```
ahead[0] = ahead[1] = 0;
```

```
for (int end = n-1; ind >= 0; ind--)
```

```
{ for (int buy = 0; buy <= 1; buy++)
```

```
{ long profit = 0;
```

```
if (buy)
```

```
{ profit = max(-values[ind] + ahead[0],
0 + ahead[1]);
```

```
}
```

```
else
```

```
{
```

```
profit = max(values[ind] + ahead[1],
0 + ahead[0]);
```

```
cur[buy] = profit;
```

```
}
```

```
ahead = cur; ahead = cur;
```

```
}
```

```
return ahead[1];
```

```
}
```

DP 37

## Buy and sell stocks - III

option stocks

Q Best time to buy & sell stock - II - at max

2 transaction

Prices = [3, 3, 5, 0, 0, 3, 1, 4]

$$\begin{array}{c} \text{3} \\ \text{3} \\ \text{5} \\ \text{0} \\ \text{0} \\ \text{3} \\ \text{1} \\ \text{4} \end{array} \xrightarrow{\text{buy}} \begin{array}{c} \text{3} \\ \text{3} \\ \text{5} \\ \text{0} \\ \text{0} \\ \text{3} \\ \text{1} \\ \text{4} \end{array} \xrightarrow{\text{sell}} \begin{array}{c} \text{3} \\ \text{3} \\ \text{5} \\ \text{0} \\ \text{0} \\ \text{3} \\ \text{1} \\ \text{4} \end{array}$$

at max 2 transaction.

$f(\text{ind}, \text{buy}, \text{cap})$  at max transaction

{ if ( $\text{ind} == n$ )

    return 0;

    if ( $\text{cap} == 0$ )

        return 0;

    if ( $\text{buy}$ )

        return  $\max [-\text{price}[\text{ind}] + f(\text{ind}+1, 0, \text{cap}),$

            0 +  $f(\text{ind}+1, 1, \text{cap})]$ .

}

else

{

    sell\_in →

        return  $\max [\text{price}[\text{ind}] + f(\text{ind}+1, 1, \text{cap}-1),$

            0 +  $f(\text{ind}+1, 0, \text{cap})]$ .

Recursion

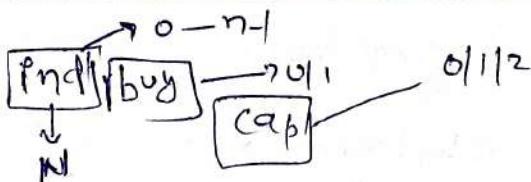
T.C. → exponential

S.C. →  $O(N)$

→ A.S (Auxiliary space)

overlapping subproblems

↓  
Memorization



three changing parameter | 3D DP

$$dp[N][2][3] \quad \underline{N \times 2 \times 3}$$

$$T.C. = N \times 2 \times 3$$

$$S.C. = O(N \times 2 \times 3) + O(N)$$

Code

int f (int ind, int buy, int cap, vector<int> &prices  
int n, vector<vector<vector<int>> &dp)

{  
if (ind == n || cap == 0)  
return 0;

if (buy)

{  
return -max[dp[ind][buy][cap]]  
+ f(ind+1, 0, cap, prices, n, dp);  
+ f(ind+1, 1, cap-1, prices, n, dp);  
}

return dp[ind][buy][cap] =  
max[prices[ind] + f(ind+1, 1, cap-1, prices, n, dp),  
0 + f(ind+1, 0, cap, prices, n, dp)];  
}

int Maxprofit (vector<vector<int>> &prices, int n)  
{  
vector<vector<vector<int>> dp(n,  
vector<vector<int>> (2, vector<int> (3, -1))));  
return f(0, 1, 2, prices, n, dp);  
}

## Tabulation

① Base Case

② changing parameter  $\text{ind}$   $\xrightarrow{\text{Buy}}$   $\xrightarrow{\text{Cap}}$

③ copy the recurrence

④ if ( $\text{cap} == 0 \& \text{ind} == n$ )  
    return 0;

if ( $\text{cap} == 0$ )                                  if ( $\text{ind} == n$ )  
 $\text{ind}$  & Buy can be anything                      Cap & Buy can be anything  
 $0 \rightarrow 1$                                            $0 \rightarrow 2$                                    $0 \rightarrow 1$

for  $\text{ind} = 0 \rightarrow n-1$

    for (buy  $\rightarrow 0 \rightarrow 1$ )

$$\text{dp}[\text{ind}][\text{buy}][0] = 0$$

    for (buy  $\rightarrow 0 \rightarrow 1$ )

        for (cap  $\rightarrow 0 \rightarrow 2$ )

$$\text{dp}[0][\text{buy}][\text{cap}] = 0$$

⑤

;  $\xrightarrow{\text{run}}$  in reverse order

buy  $\rightarrow 0 \rightarrow 1$

cap  $\rightarrow 0 \rightarrow 2$

⑥ answer will be initial cell i.e  $\text{dp}[0][1][0]$

```

int MaxProfit(vector<int> &prices, int n)
{
 vector<vector<vector<int>> dp(n, vector<vector<int>(2, vector<int>(3, 0)));
 so we don't write out base cases. already dp is full with 0.
 different like 1, 0 then we may write.

```

```

for (int i = 0; i < n - 1; i++)
{
 for (int buy = 0; buy <= 1; buy++)
 {
 for (int cap = 0; cap <= 2; cap++)
 {
 if (buy == 1)

```

$dp[i][buy][cap] = \max(-\text{price}[i] + dp[i+1][0][cap],$

 $0 + dp[i+1][1][cap]);$ 

else

$dp[i][buy][cap] = \max(\text{price}[i] + dp[i+1][1][cap-1],$

 $0 + dp[i+1][0][cap]);$ 

}

return dp[0][1][2].

}

## Space optimization :-

@Aashish Kumar Nayak

```
int maxProfit(vector<int> &prices, int n)
```

```
{ vector<vector<int>> after(2, vector<int> (3, 0));
```

```
vector<vector<int>> cur(2, vector<int> (3, 0));
```

```
for (int ind = n - 1; ind >= 0; ind--)
```

```
{ for (int buy = 0; buy <= 1; buy++)
```

```
{ for (int cap = 1; cap <= 2; cap++)
```

```
{ if (buy == 1)
```

```
cur[buy][cap] = max[prices[ind] + after[0][cap], 0 + after[1][cap]];
```

```
else
```

```
}
```

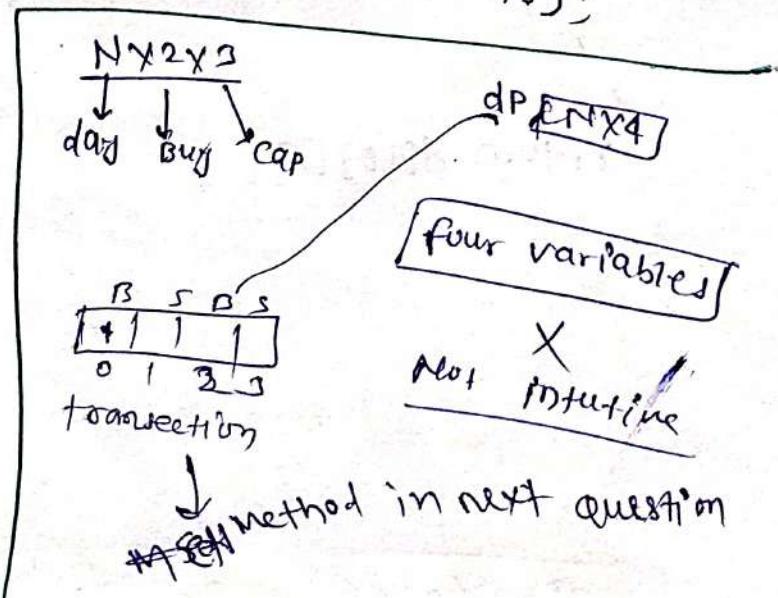
```
cur[buy][cap] = max[prices[ind] + after[1][cap - 1],
0 + after[0][cap]];
```

```
after = cur;
```

```
}
```

```
return after[1][2];
```

```
}
```



DP-38

## Buy and Sell Stocks - IV | DP on Stocks

At max K transactions

K can be anything

Code is similar only change is 1< 2  
Replace 2 with K

Code (Space Optimization)

```
int maxprofit(vector<int> &prices, int n, int k)
{
 vector<vector<int>> after(2, vector<int>(K+1, 0));
 vector<vector<int>> cur(2, vector<int>(K+1, 0));
 for (int ind = n-1; ind >= 0; ind--)
 {
 for (int buy = 0; buy <= 1; buy++)
 {
 for (int cap = 1; cap <= k; cap++)
 {
 if (buy == 1)
 {
 cur[buy][cap] = max(-prices[ind] + after[0][cap], 0 + after[1][cap]);
 }
 else
 {
 cur[buy][cap] = max(prices[ind] + after[1][cap], 0 + after[0][cap]);
 }
 }
 }
 after = cur;
 }
 return after[1][k];
}
```

## Code

```
int f(int ind, int transNo, vector<int>& prices, int n, int k,
 vector<vector<int>>& dp)
{
 if(ind == n || transNo == 2 * k)
 return 0;
 if(dp[ind][transNo] != -1)
 return dp[ind][transNo];
 if(transNo % 2 == 0)
 {
 return dp[ind][transNo] = max(-prices[ind] + f(ind+1, transNo+1, prices, n, k, dp),
 0 + f(ind+1, transNo, prices, n, k, dp));
 }
 return dp[ind][transNo] = max(prices[ind] + f(ind+1, transNo+1, prices, n, k, dp),
 0 + f(ind+1, transNo, prices, n, k, dp));
}

int find_maximum_profit(vector<int>& prices, int n, int k)
{
 vector<vector<int>> dp(n, vector<int>(2 * k, -1));
 return f(0, 0, prices, n, k, dp);
}
```

## Tabulation

```
int maxProfit(vector<int> &prices, int n, int k)
{
 vector<vector<int>> dp(n+1, vector<int>(2*k+1, 0));
 for (int ind = n-1; ind >= 0; ind--)
 {
 for (int transNo = 2*k-1; transNo >= 0; transNo--)
 {
 if (transNo % 2 == 0)
 dp[ind][transNo] = max(-prices[ind] + dp[ind+1][transNo],
 dp[ind+1][transNo]);
 else
 dp[ind][transNo] = max(prices[ind] + dp[ind+1][transNo-1],
 dp[ind+1][transNo]);
 }
 }
 return dp[0][0];
}
```

}, ~~Space~~

## Space Optimization :-

```
int maxProfit(vector<int> &prices, int n, int k)
```

{

```
vector<int> after(2*k + 1, 0);
```

```
vector<int> cur(2*k + 1, 0);
```

```
for (int ind = n - 1; ind >= 0; ind--)
```

```
{ for (int transNo = 2*k - 1; transNo >= 0; transNo--)
```

```
{ if (transNo % 2 == 0)
```

```
cur[transNo] = Max(prices[ind] + after[transNo + 1],
 0 + after[transNo]);
```

```
} else
```

{

```
cur[transNo] = Max(prices[ind] + after[transNo + 1],
 0 + after[transNo]);
```

{

```
after = cur;
```

}

```
return after[0];
```

{

T.C.  $\approx O(2k)$   
 $\approx O(k)$

5/25

DP-39

## Buy and sell stocks with cooldown

cooldown → can't buy right after sell.

price [] →

$$\{ 4, 9, 0, 4, 10 \}$$

5 + 10 = 15

cooldown

$$\{ 4, 9, 0, 4, 10 \}$$

B S X B

→ B V

$f(\text{ind}, \text{Buy})$

{ if ( $\text{ind} = n$ ) return 0 }

if (Buy)

return  $\max \left\{ -\text{price}[\text{ind}] + f(\text{ind}+1, 0), 0 + f(\text{ind}+1, 1) \right\}$

else

return  $\max \left( \text{price}[\text{ind}] + f(\text{ind}+1, 1), 0 + f(\text{ind}+1, 0) \right)$

only this minor change on  
D.P. on stocks II problem

we solved the D.P. on stock

II problem

D.P. on stock

| 5th problem  
| DP on stocks

```

int f(int ind, int buy, vector<int> &prices, vector<vector<int>> &dp)
{
 if(ind >= prices.size())
 return 0;
 if(buy == 1)
 if(dp[ind][buy] != -1) return dp[ind][buy];
 else
 dp[ind][buy] = max(-prices[ind] + f(ind+1, 0, prices, dp),
 0 + f(ind+1, 1, prices, dp));
 else
 return dp[ind][buy] = max(prices[ind] + f(ind+1, 1, prices, dp),
 0 + f(ind+1, 0, prices));
}
int maxProfit(vector<int> &prices)
{
 int n = prices.size();
 vector<vector<int>> dp(n, vector<int>(2, -1));
 return f(0, 1, prices, dp);
}

```

### Tabulation :-

```

int maxProfit(vector<int> &prices)
{
 int n = prices.size();
 vector<vector<int>> dp(n, vector<int>(2, 0));
 for(int ind = n-1; ind >= 0; ind--)
 {
 for(int buy = 0; buy <= 1; buy++)
 {
 if(buy == 1)
 dp[ind][buy] = max(-prices[ind] + dp[ind+1][0],
 0 + dp[ind+1][1]);
 else
 dp[ind][buy] = max(prices[ind] + dp[ind+1][1],
 0 + dp[ind+1][0]);
 }
 }
 return dp[0][1];
}

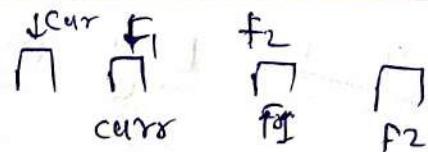
```

## more optimization

```

int maxprofit(vector<int> &prices)
{
 int n = prices.size();
 vector<int> front2(2, 0);
 vector<int> front1(2, 0);
 vector<int> cur(2, 0);
 for(int ind = n-1; ind >= 0; ind--)
 {
 cur[1] = max(-prices[ind] + front1[0],
 0 + front1[1]);
 cur[0] = max(prices[ind] + front2[1],
 0 + front2[0]);
 front2 = front1;
 front1 = cur;
 }
 return cur[1];
}

```



T.C. -  $O(N)$   
 S.C. -  $O(6) \approx O(1)$

DP-40

## Buy and sell stocks with Transaction fee

prices[] = {  
 1, 3, 2, 8, 4, 9 }  
 fee = 2

| DP on stocks

6th problem

f(ind, buy)

{ if (ind == 0) return 0;  
 if (buy == 1)

max(-prices[ind] - fee,  
 else                            or  
 max((price[ind]) - fee) + f(ind+1, 1), 0 + f(ind+1, 0)).

After the sell  
 is done deduct the fee or we can also deduct the fee  
 during buying.

Code E

```

int maximumprofit(int n, int fee, vector<int> &values)
{
 int aheadNotBuy, aheadBuy, curBuy, curNotBuy;
 aheadBuy = aheadNotBuy = 0;
 for (int ind = n-1; ind >= 0; ind--)
 {
 // sell
 curNotBuy = max(values[ind] + aheadBuy, 0 + aheadNotBuy);
 // buy
 curBuy = max(values[ind] - fee + aheadBuy, 0 + aheadNotBuy);
 aheadBuy = curBuy;
 aheadNotBuy = curNotBuy;
 }
 return aheadBuy;
}

```

DP-41

## Longest increasing Subsequence

arr[] = [10, 9, 2, 5, 3, 7, 10, 18]

{2, 3, 7, 10} → len=4

{2, 3, 7, 18} → len=4

arr[] = {8, 8, 8}

18 → len = 1

Brute Force  
 1st try out various subsequences  
 ↓  
 (2^n) Check for increase  
 exponential Store the longest.

→ power set |  
 Recursion  
 in order to generate  
 all subsequences

Trying all ways

Recurrence

→ Recursion

10, ✗

prev=10 to check

- (i) Express everything in terms of index
- (ii) explore all possibilities in subsequence — ✓
- (iii) Take the max length — ✗

f(ind, prev-ind)

f(0,-1) → length of LIS starting from 0.

f(0,0) → length of LIS starting from index where previous index is p. → prev

f(ind

f(0,-1)

+1

x

f(1,-1)

f(1,0)

+1

f(2,-1)

+1

f(2,1)

Base case:

```
if(ind == n)
 return 0;
```

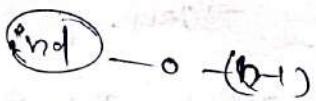
Recursion

2 options i.e  $O(2^n)$ .  
1/take not-take

T.C.  $O(2^n)$  for every index we have  
S.C. -  $O(N)$

↓ overlapping sub-problem  
memoization

$f(ind, prev\_ind)$



Co-ordinate change:

(1) 0 (2) 1 (3)

code:

T.C. -  $O(N \times N)$

S.C. -  $O(N \times N) + O(N)$

```
int f(int ind, int prev_ind, int arr[], int n, vector<vector<int>>&dp)
{
 if(ind == n) return 0;
 if(dp[ind][prev_ind+1] != -1) return dp[ind][prev_ind+1];
 int len = 0 + f(ind+1, prev_ind, arr, n, dp);
 if(prev_ind == -1 || arr[ind] > arr[prev_ind])
 {
 len = max(len, 1 + f(ind+1, ind, arr, n, dp));
 }
 return dp[ind][prev_ind+1] = len;
}
```

int longest\_increasing\_subsequence(int arr[], int n)

```
{ vector<vector<int>> dp(n, vector<int>(n+1, -1));
 return f(0, -1, arr, n, dp); }
```

}

## Tabulation :-

Rules.

1) Base case

2) Changing parameter

$$dp[n][n+1] = 0$$

$$\left\{ \begin{array}{l} \text{Ind} = n-1 \rightarrow 0 \\ \text{Prev-Ind} = \text{Ind} - 1 \rightarrow -1 \end{array} \right.$$

? we already assign 0 to our  
so no need to write base case.

3) copy the recurrence

Follows coordinate shift

Code :-

```

int longestIncreasingSubsequence(int arr[], int n)
{
 vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));
 for (int ind = n - 1; ind >= 0; ind--)
 {
 for (int prevInd = ind - 1; prevInd >= -1; prevInd--)
 {
 int len = 0 + dp[ind + 1][prevInd + 1];
 if (prevInd == -1 || arr[ind] > arr[prevInd])
 len = max(len, 1 + dp[ind + 1][ind + 1]);
 }
 dp[ind][prevInd + 1] = len;
 }
 return dp[0][n - 1];
}

```

## Space optimization

@Aashish Kumar Nayak

int longestIncreasingSubsequence (int arr[], int n)

{ vector<int> next(n+1, 0), cur(n+1, 0);

for (int i = n-1; i >= 0; i--)

{ for (int prevInd = i+1; prevInd >= -1; prevInd--)

{ int len = 0 + next[prevInd+1];

if (prevInd == -1 || arr[i] > arr[prevInd])

{ len = max(len, 1 + next[prevInd+1]);

next[prevInd+1] = len;

next = cur;

return next[-1+1];

T.C. -  $O(N^2)$

S.C. -  $O(N) \times 2$

Example

|   |   |    |   |    |   |
|---|---|----|---|----|---|
| 5 | 4 | 11 | 1 | 16 | 8 |
|---|---|----|---|----|---|

, dp[m]

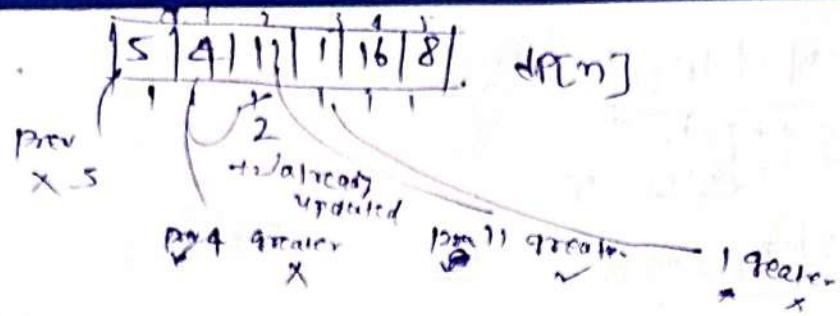
$dp[i]$  → signifies the longest increasing subsequence that ends at index  $i$ .

{5, 16}

{3, 11, 16}

$L[i] \rightarrow \max$  of all  $dp$  index.

$$L = \max_{i=0}^{n-1} \max(dp[i])$$



for (int i = 0 → n-1)

    for (int j = 0 → i-1,

        if (arr[i] > arr[j])

            dp[i] = max(1 + dp[j], dp[i]).

Code :-

{ int tangentIncreasingSubsequences(int arr[], int n)

    vector<int> dp(n, 1);

    int maxi = 1;

    for (int i = 0; i < n; i++)

        for (int j = 0; j < i; j++)

            if (arr[i] > arr[j])

                dp[i] = max(dp[i], 1 + dp[j]);

    maxi = max(maxi, dp[i]);

return maxi;

|                 |
|-----------------|
| T.C. - $O(N^2)$ |
| S.C. - $O(N)$   |

This solution will be  
require if we want  
to trace the Lij.

DP 42

5 | 9 | 11 | 11 | 16 | 18

↑ ↑ ~~11~~ ↑ ↑ ~~16~~ ↑ ↑ dp

0 | 1 | ~~2~~ | 3 | ~~4~~ | 80 hash

④ → ② → ①  
16      11      5

reverse it

5, 11, 16

Code :-

```
int longestIncreasingSubsequence(int arr[], int n)
```

```
{ vector<int> dp(n, 1), hash(n);
```

```
int maxi = 1;
```

```
int lastIndex = 0;
```

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

```
{ hash[i] = i;
```

```
for (int prev = 0; prev < i; prev++)
```

```
{ if (arr[prev] < arr[i]) {
```

```
 if (dp[prev] > dp[i])
```

```
 dp[i] = 1 + dp[prev];
```

```
 hash[i] = prev;
```

```
if (dp[i] > maxi)
```

```
{ maxi = dp[i];
```

```
lastIndex = i;
```

vector<int> temp;

temp.push\_back(arr[lastIndex]);

while(hash[lastIndex] != lastIndex)

{

lastIndex = hash[lastIndex];

temp.push\_back(arr[lastIndex]);

}

reverse(temp.begin(), temp.end());

for(auto it : temp)

{cout << it << " " ;}

cout << endl;

return maxi;

}

P.C. -  $O(M^2)$

~~$O(N)$~~

DP-43

## Longest Increasing Subsequence

In this lecture we will use  
Binary search  $1 \leq N \leq 10^5$

LIS using Binary Search

$[1, 7, 8, 4, 5, 6, -1, 9]$

Increasing subsequences are

$1, 4, 5, 6, 9$  — longest

Approach :  $1, 4, 6, 9$

Try to go every element and form subsequence

For,  $[1, 7, 8, 4, 5, 6, -1, 9]$

$[1, 7, 8, 4, 5, 6, -1, 9]$

$[1, 7, 8, *, 9] \leftarrow \text{len}=4$

$[1, 4, 5, 6, 9] \leftarrow \text{len}=5$

$[-1, 9] \leftarrow \text{len}=2$

But a lot of space will be consumed.

Intuition

- To avoid space we will overwrite instead of creating new array.

$[1, 7, 8, 4, 5, 6, -1, 9]$

$\cancel{1}, \cancel{4}, \cancel{5}, \cancel{6}, 9$

$[1, 4]$  — instead of making  $[1, 4]$  new overwrite  
in 1st array Replace  $\cancel{4}$  with 7

~~class~~  $[]$  { $1, 4, 5, 4, 2, 8$ }

$\cancel{1}, \cancel{4}, \cancel{5}, 8$

$\leftarrow \text{len}=4$  — space has been optimized.

in previous lecture  
Recursion  
Memoization  
Space optimization  
Tabulation  
Algorithmic approach

T.C. —  $O(N^2)$   
S.C. —  $O(N)$   
Fill more

Binary Element       $e \rightarrow arr[i]$

if( find arr[i] ).

else find 1st greater of ~~arr~~, arr[i]

or

lower\_bound()

→ it gives you 1st pointer of arr[i]  
or  
first index  $\leq arr[i]$

Code :-

```
int longestIncreasingSubsequence(int arr[], int n)
{
 vector<int> temp;
 temp.push_back(arr[0]); int len = 1;
 for (int i = 1; i < n; i++)
 {
 if (arr[i] > temp.back())
 { temp.push_back(arr[i]);
 len++;}
 else
 {
 len + i = lower_bound(temp.begin(), temp.end(),
 arr[i]);
 temp[len - 1] = arr[i];
 }
 }
 return temp.size();
}
return len; // len will take less space than temp
```

T.C. =  $O(N \log N)$

S.C. =  $O(N)$

lower bound.

$O(\log N)$

DP-44

# Largest Divisible Subset | Longest Increasing Subsequence

$$arr[] = \{1, 16, 7, 8, 9\}$$

@Aashish Kumar Nayak

{1, 16, 8} *subsequence*

{1, 8, 16}

{1, 16, 8}

{1, 7, 16}

Subset

not necessary  
to follow the  
order.

$$arr[i] \times arr[j] == 0$$

$$\text{or } arr[j] \times arr[i] == 0$$

Let's take an example

{16, 8, 4}      (16, 8)      (8, 4)      Every pair is divisible  
                        (16, 4)

{1, 16, 8, 4}      (1, 16), (1, 8), (1, 4), (16, 8), (16, 4), (8, 4)

len=4

All the pairs are divisible thus can be  
possible ans.

Point any answer you can jumble it and return.

{1, 16, 8, 4} or {4, 16, 1, 8}



Intuition

Sort the array

{1, 4, 7, 8, 16}

{1, 4, 8, 16}

: 4 is divisible by 1 so return it

7 is not divisible by 4.

: 8 is divisible by 4 is

and also if it is divisible by 8

we will get len=4 then it will automatically  
longest divisible increasing subsequence.

divisible by 1 or previous  
elements.

lis code :-

for (i=1; i < n; i++)

{ for (j=0; j < i; j++)

{ if (arr[i] % arr[j] == 0)

if (arr[j] < arr[i] && dp[i] < dp[j] + 1)

dp[i] = dp[j] + 1

, maxh[i] = j

because it is already

sorted just check arr[i] % arr[j],  
or arr[j] % arr[i]

## Code

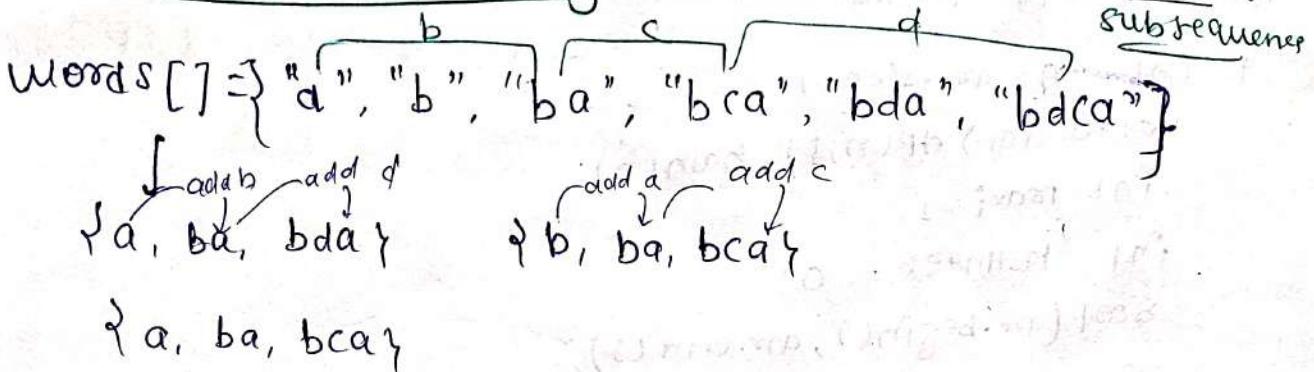
Method ( $\text{int} \rightarrow$  divisiblest (vector<int> arr))

```
?
int n = arr.size();
vector<int> dp(n, 1), hash(n);
int maxi = 1;
int lastIndex = 0;
sort(arr.begin(), arr.end());
for (int i = 0; i < n; i++)
{
 hash[i] = i;
 for (int prev = 0; prev < i; prev++)
 if (arr[i] % arr[prev] == 0 && 1 + dp[prev] > dp[i])
 dp[i] = 1 + dp[prev];
 hash[i] = prev;
}
if (dp[i] > maxi)
{
 maxi = dp[i];
 lastIndex = i;
}
vector<int> temp;
temp.push_back(arr[lastIndex]);
while (hash[lastIndex] != lastIndex)
{
 lastIndex = hash[lastIndex];
 temp.push_back(arr[lastIndex]);
}
reverse(temp.begin(), temp.end());
return;
```

T.C.  $O(N^2)$   
S.C.  $O(N)$

DP-4<sup>5</sup>

## Longest String Chain | longest increasing subsequence



for ( $i=1 \rightarrow n$ )

```

 for (j=0; j < i; j++) {
 if (arr[i] > arr[j]) dp[i] = dp[j] + 1;
 max[i] = max(max[i], dp[i]);
 }
 return max[i];
}

```

↑ ↓ ↓ ↓  
bdca  
↑ ↑ ↑

arr[1]  
↓ ↓ ↓  
bcd  
← ta arr[j]  
bcd

bcd  
↑ ↑ ↑ ↑  
if both pointers reaches its end means match.

In this question there is sequence Not sub-sequence so we can pick from anywhere.

② [ "abc", "pexbc r", "x<sup>①</sup>b", "c xbc", "pc xbc" ]

We will sort this array according to the length.

bool comp(string &s<sub>1</sub>, string &s<sub>2</sub>)  
write comparison  
return s<sub>1</sub>.size() < s<sub>2</sub>.size();

Code :- int longestStochain (vector<string> &arr)

{

    sort (arr.begin(), arr.end(), comp);

    int n = arr.size();

    vector<int> dp(n, 1);

    int maxi = 1;

    for (int i = 0; i < n; i++)

        for (int prev = 0; prev < i; prev++)

            if (checkpossible (arr[i], arr[prev])  
                && s[i] + dp[prev] > dp[i])

                dp[i] = s[i] + dp[prev];

        }

        if (dp[i] > maxi)

            maxi = dp[i];

    }

    return maxi;

}

bool comp(string &s1, string &s2)

{

    return s1.size() < s2.size();

bool checkpossible (string &s1, string &s2)

{

    if (s1.size() != s2.size() + 1) return false;

    int first = 0;

    int second = 0;

    while (first < s1.size())

        if (s1[first] == s2[second]) {

            first++; second++;

        } else {

            first++;

        } if (first == s1.size() & second == s2.size()) return true; else return false;

length of  
the string

T.C. -  $O(N^2 \times l)$

S.C. - ~~+ O(n log n)~~  
/ sorting.

DP-46

## Longest Bitonic Subsequence | LIS

$$\text{arr}[] = \{1, 11, 2, 10, 4, 5, 2, 1\}$$

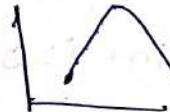
Bitonic  $\rightarrow$  increasing  $\rightarrow$  decreasing. Or it can be just increase or just decrease

$$\{1, 2, 10, 4, 2, 1\}$$

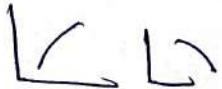
$$\{1, 2, 10, 5, 2, 1\}$$

$$\{1, 11, 10, 4, 2, 1\}$$

$$\{1, 11, 10, 5, 2, 1\}$$



be just increase  
or just decrease



$$\text{arr}[] = \{1, 10, 2, 3, 4, 5, 6\}$$

because  
of  
subsequence

$$\{1, 2, 3, 4, 5, 6\} \text{ len=6}$$

$$\text{arr}[] = \{5, 4, 3, 2, 1\}$$

$$\{5, 4, 3, 2, 1\} \text{ len=5}$$

for( $i=0$ ;  $i < n$ ;  $i+1$ )

{  
    dp[i] = 1

    for( $j=0$ ;  $j < i$ ;  $j+1$ )

        {  
            if( $\text{arr}[i] > \text{arr}[j]$  &  $\text{dp}[i] + 1 > \text{dp}[j]$ )  
                dp[i] = 1 + dp[j].

            longest increasing subsequence till index i.

$$\text{arr}[] = \{1, 11, 2, 10, 4, 5, 2, 1\}$$

$$\text{dp1}[] = \{1, 2, 2, 3, 3, 4, 2, 1\}$$

$$\text{dp2}[] = \{1, 2, 5, 3, 4, 3, 2, 1\}$$

$$2 \rightarrow \frac{1}{1} \quad \frac{6}{6} \quad \frac{5}{5} \quad \frac{6}{6} \quad \frac{1}{1} \quad 2 + 4 = 7$$

(0 is common)

= 6 long

Code :-   
 int longestBitonicSequence(vector<int> &arr, int n)   
 {   
 vector<int> dp1(n, 1);   
 for (int i = 0; i < n; i++)   
 {   
 for (int prev = 0; prev < i; prev++)   
 {   
 if (arr[prev] < arr[i] && 1 + dp1[prev] > dp1[i])   
 {   
 dp1[i] = 1 + dp1[prev];   
 }   
 }   
 int maxi = 0;   
 }   
 vector<int> dp2(n, 1);   
 for (int i = n - 1; i >= 0; i--)   
 {   
 for (int prev = n - 1; prev > i; prev--)   
 {   
 if (arr[prev] < arr[i] && 1 + dp2[prev] > dp2[i])   
 {   
 dp2[i] = 1 + dp2[prev];   
 }   
 }   
 maxi = max(maxi, dp1[i] + dp2[i] - 1);   
 }   
 return maxi;   
 }

Some  
 $O(N)$   
 T.C.  
 O(N)

T.C. -  $O(N^2)$

DP-47

## Number of longest increasing subsequence

$$\text{arr}[] = \{1, 3, 5, 4, 7\}$$

possible subsequences  
 $\{1, 3, 4, 7\}$        $\{1, 3, 5, 7\}$        $\{1, 3, 5, 4, 7\}$        $\{1, 3, 5, 4, 7, 10\}$        $\{1, 3, 5, 4, 7, 8, 9\}$

$\text{arr}[] \rightarrow \{1, 3, 5, 4, 7, 2, 8, 6, 7, 10, 8, 9\}$       attached with 1  
 length

$$\text{dp}[i] \rightarrow 1 \ x_2 \ x_2 \ x_2 \ x_2 \ x^2 \ 3$$

$$\text{cnt}[i] \rightarrow 1 \ 1 \cdot 1 \ 1 \ 1 \ 2 \ 9$$

4  
1, 5

4 itself of length 1 and cnt is 1, 1 can be attached with 4  
 increase cnt by 1 and dp remain,

1, 5, 7  
1, 4, 7  
1, 3, 7  
1, 2, 7  
1, 6, 7

$$\begin{aligned} \text{dp}[i] &= \{1, 3, 5, 4, 7\} \\ \text{cnt}[i] &= 1, 1, 1, 1, 1, 2 \end{aligned}$$

Code :- int findNumberofLIS(vector<int> &arr)

{ vector<int> dp(n, 1), cnt(n, 1);

int maxi = 1; int n = arr.size();

for (int i = 0; i < n; i++)

{ for (int prev = 0; prev < i; prev++)

  { if (arr[prev] < arr[i] && 1 + dp[prev] > dp[i])

    { dp[i] = 1 + dp[prev];

    } cnt[i] = cnt[j];

```

 else if (arr[prev] < arr[i] && i + dP[prev] == dP[i])
 {
 cnt[i] != cnt[prev];
 maxi = max(maxi, dP[i]);
 }
 int cnt = 0, nos = 0;
 for (int i = 0; i < n; i++)
 {
 if (dP[i] == maxi)
 nos += cnt[i];
 }
 return nos;
}

```

# DP - 48 Matrix chain multiplication (MCM) Partition DP starts

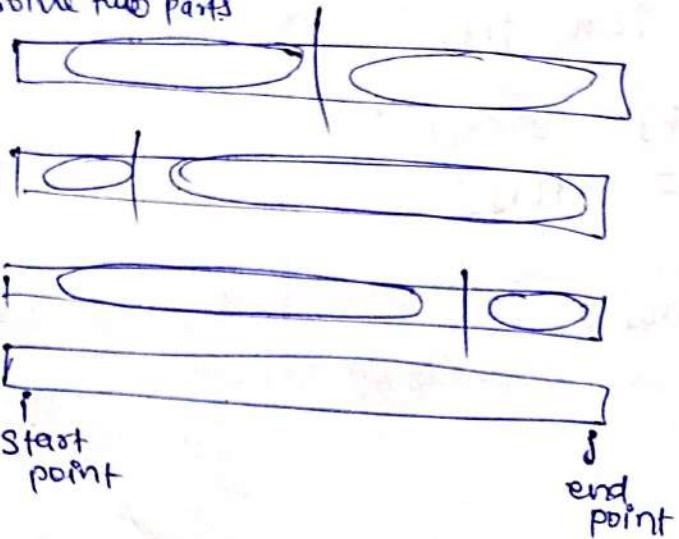
New pattern - [pattern DP] Tough pattern

Solve a problem in a pattern

$$(1+2+3) \times (5)$$

$$(1+2)+(3 \times 5)$$

We have to solve two parts



MCM → matrix chain multiplication

$$B = 30 \times 5$$

$$\begin{bmatrix} A \\ \end{bmatrix}_{10 \times 30} \quad \begin{bmatrix} B \\ \end{bmatrix}_{30 \times 5} \quad L = 5 \times 60$$

$= 10 \times 30 \times 5 = 1500$  operations required to multiply the matrix.

$$\begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix}_{2 \times 2} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix}_{2 \times 1} = \begin{bmatrix} 1 \times 2 & 2 \times 1 \\ 3 \times 2 & 1 \times 1 \end{bmatrix}_{3 \times 2}$$

$2 \times 2 \times 1 = 4$  operations

$$(A B C)$$

$$A = 10 \times 30$$

$$\underline{(A B) C}$$

$$10 \times 30 = 3000$$

$$B = 30 \times 5$$

$$C = 5 \times 60$$

$$\begin{bmatrix} \ ]_{10 \times 5} E \ ]_{5 \times 60} = (10 \times 5 \times 60)$$

$$= 1500 + 3000 = 4500$$

@Ashish Kumar Nayak

$$\begin{array}{c}
 A(BC) \\
 | \quad | \quad | \\
 B(30 \times 30) \quad C(30 \times 60) \\
 \text{Given } A(10 \times 30) \quad 9000 \\
 = 10 \times 30 \times 60 + 9000 \\
 = 18000 + 9000 \\
 = 27000
 \end{array}$$

The minimum operation required  
given the N matrix dimension

ABCD

$A(B(CD))$        $AB(CD)$

Given

$\text{arr}[] \rightarrow [10, 20, 30, 40, 50]$   
 $\downarrow$  dimensions of  $N-1$  matrices  $N=5$

$A \rightarrow 10 \times 20, B \rightarrow 20 \times 30, C \rightarrow 30 \times 40, D \rightarrow 40 \times 50$

1st  $\rightarrow A[0] \times A[1]$     2nd  $A[1] \times A[2]$     3rd  $A[2] \times A[3]$      $i^{th} \rightarrow A[i-1] \times A[i]$

Tell me the no. of operations required to multiply the  
matrices.

Approach

Partition Partition DP

Take the 4 guys and give one-the ans.

ABCD  
 $(AB)(CD)$      $(A)(BCD)$      $(ABC)(CD)$     { take the minimal of them

1. Start with entire block/array  $f(i, j)$      $\nearrow$  start point     $\searrow$  end point

2. Try all partition

$i$   $\nearrow$  start point    Run a loop to try all partitions

3. Return the best possible 2 partition

$[10, 20, 30, 40, 50]$

A    B    C    D  
 |       |       |       |  
 i       j  
 ↓       ↓  
 1       n-1

$f(1, 4) \rightarrow$  return the minimum multiplication to multiply  
matrix 1  $\rightarrow$  4

Base case :-

```
f(i, j)
{ if(i == j)
 return 0;
```

Try all partitions

$$\{ 10, 20, 30, 40, 50 \}$$

(A) (B) C (D)

( ) ( )  
( ) ( )

$$K = (i^1 \rightarrow j^3)$$

$$k=1 \quad f(i^1, k^1), f(k+1^2, j^4)$$

$$k=2 \quad f(1, 2), f(3, 4)$$

$$k=3 \quad f(1, 3), f(4, 4)$$

$$K = (i+1 \rightarrow j)$$

$$f(i^1, k^1), f(k, j)$$

for ( $k \rightarrow i \rightarrow j-1$ )

$$\{ \text{steps} = A[i-1] \times arr[k] \times arr[j] \} + f(i, k) + f(k+1, j)$$

$$f(1, 1) \quad f(2, 4) \quad A = 10 \times 20$$

$$(10 \times 20) \quad (20 \times 30) \quad B = 20 \times 30$$

$\swarrow$  (BCD)  $\searrow$

$$(10 \times 50) \quad (20 \times 50) \quad C = 30 \times 40$$

$\swarrow$  (10x20x50)  $\searrow$  operation or steps.

$$B \quad C \quad D$$

$$20 \times 30 \quad 30 \times 50 \quad (20 \times 30) \quad (30 \times 40) \quad (40 \times 50)$$

$\swarrow$  (20x40)  $\searrow$  (40x50)

$$(20 \times 50)$$

$\swarrow$  (20x50)  $\searrow$

$$P \quad i \quad j$$

$$10 \quad 20 \quad 30 \quad 40 \quad 50$$

$$P \quad B \quad C \quad D$$

$f(AB) + f(CD)$

$$10 \times 20 \quad 30 \times 30 \quad 40 \times 50$$

$$20 \times 30 \quad 30 \times 50$$

$$10 \times 30$$

$$10 \times 50$$

No. of steps

$$= 10 \times 50 \times 50$$

$$A(i-1) \times A(k) \times A(l)$$

Recursion Tree

$$(10, 20, 30, 40, 50)$$

\* single  
matrix  
multiplication

$$[20] [30, 40, 50]$$

$$ij \quad B \quad C \quad D$$

$$AB \quad CD$$

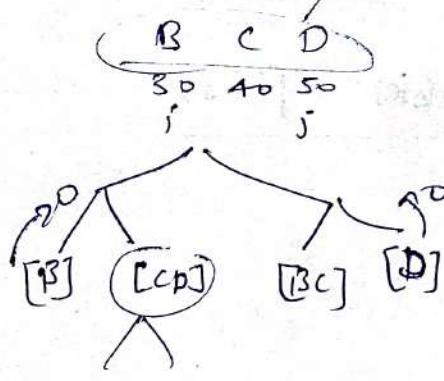
$$20 \times 30 \quad 40 \times 50$$

$$ij \quad ij$$

$$ij \quad ij$$

$$ij$$

$$(10 * 20 \times 50) + f + f$$



then take min  
of all.

Cost

Code =

```
int f(int i, int j, vector<int> &arr)
{
 if(i == j) return 0;
 int mini = 1e9;
 for(int k = i; k < j; k++)
 {
 int steps = arr[i-1] * arr[k] * arr[j] + f(i, k, arr);
 if(steps < mini)
 mini = steps;
 }
 return mini;
 return dpc[i][j] = mini;
}

int matrixMultiplication(vector<int> &arr, int N)
{
 vector<vector<int>> dpc(N, vector<int>(N, -1));
 return f(1, N-1, arr, dpc);
}
```

T.C. - Exponential

Apply memoization

changing variables f(i,j)

DP 49

## Matrix chain multiplication | Bottom-up | Tabulation

Rules :-

1. Copy the base case
2. write row changing states.

~~for(i)~~  
 $f(i, j)$   
 $dp[N][N]$

~~for(i = N-1; i >= 1; i--)~~  
 $i = \underline{\quad}$

Code :-

```
int matrixMultiplication(vector<int> &arr, int N)
{
 int dp[N][N];
 for (int i = N - 1; i >= 1; i--)
 {
 for (int j = i + 1; j < N; j++)
 {
 int mini = INT_MAX;
 for (int k = i; k < j; k++)
 {
 int steps = arr[i-1] * arr[k] * arr[j]
 + dp[i][k];
 if (steps < mini)
 mini = steps;
 }
 dp[i][j] = mini;
 }
 }
 return dp[1][N - 1];
}
```

return  $dp[1][N-1];$

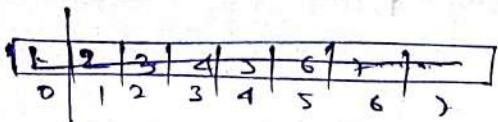
}

DP-50

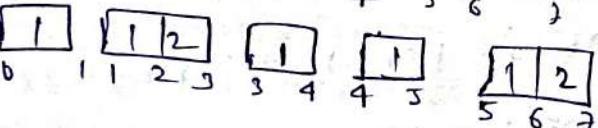
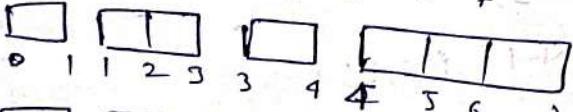
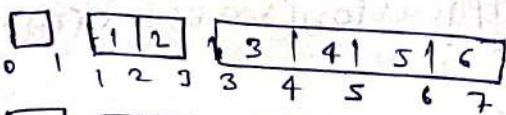
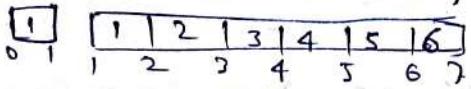
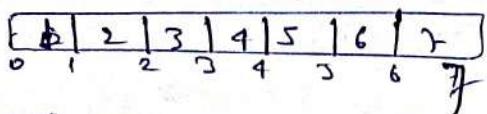
## Minimum cost to cut the stick

$$\text{arr}[] = [1, 3, 4, 5] \quad n=7$$

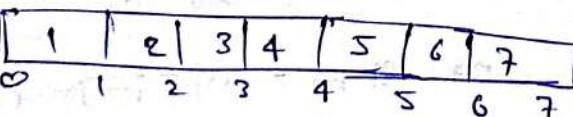
cost = length of the stick  
you are cutting



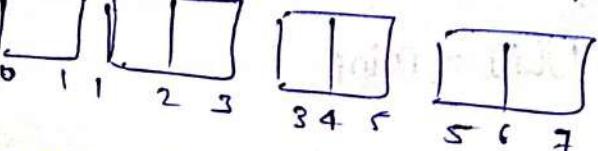
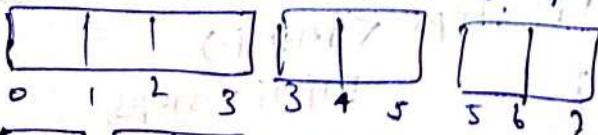
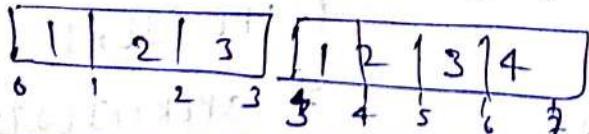
$$7 + 6 + 4 + 3 = 20$$



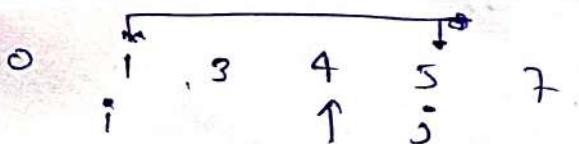
{ 3, 5, 1, 4 }



$$7 + 4 + 3 + 2 = 16$$



array is sorted

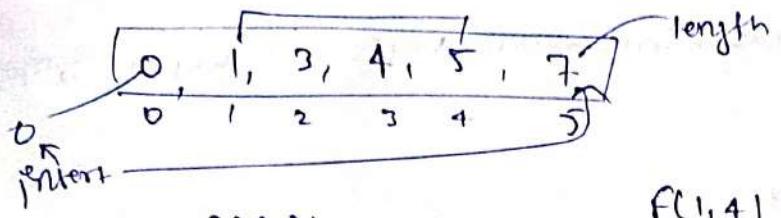


cost = length of that stick in which

$$\text{cuts}[j+1] - \text{cuts}[i-1]$$

4 is ...

$$7 - 0 = 7$$



$f(i, j)$

if ( $i > j$ )

return 0;

min<sub>i</sub> = INT\_MAX;

for (ind = i → j)

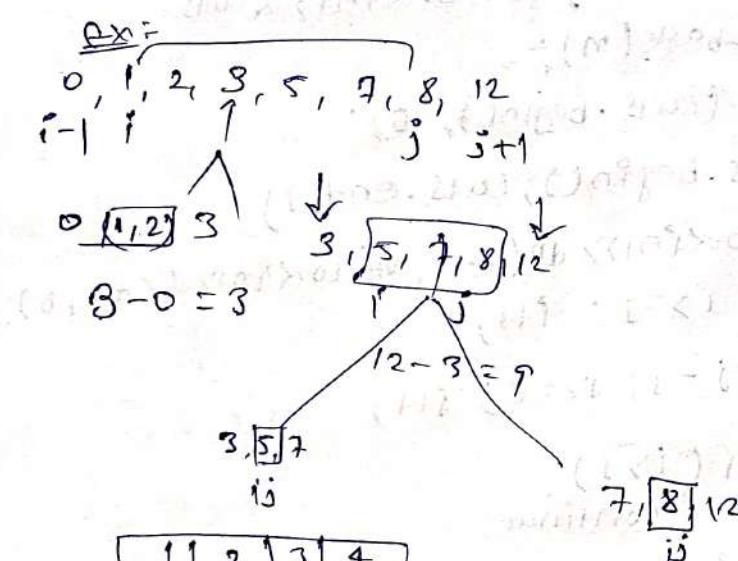
{ cost = cuts[j+1] - cuts[i-1]

+ f(i, ind-1) + f(ind+1, j)

mini = min(mini, cost);

}

return mini;



Code

int cut (int n, int c, vector<int> &cuts,

vector<int> &dp (c+1, vector<int> (c+1, -1))

{ cuts.push\_back (n); cuts.insert (cuts.begin(), 0);

sort (cuts.begin(), cuts.end());

return f (1, c, cuts, dp);

}

```

int f(int i, int j, vector<int>& cuts, vector<vector<int>>& dp)
{
 if(i > j) return 0;
 if(dp[i][j] != -1)
 return dp[i][j];
 int mini = INT_MAX;
 for(int ind = i; ind <= j; ind++)
 {
 int cost = cuts[i+1] - cuts[i-1] + f(i, ind-1, cuts, dp);
 + f(ind+1, j, cuts, dp);
 mini = min(mini, cost);
 }
 return dp[i][j] = mini;
}

```

T.C. ( $M^2 \times M$ )  $\approx O(M^3)$   
 S.C.  $\approx O(M^2)$  + Space Ass

### Tabulation :

```

int costC(int n, int c, vector<int>& cuts)
{
 cuts.push_back(n);
 cuts.insert(cuts.begin(), 0);
 sort(cuts.begin(), cuts.end());
 vector<vector<int>> dp(c+2, vector<int>(c+2, 0));
 for(int i = c; i >= 1; i--)
 {
 for(int j = i; j <= c; j++)
 {
 if(i > j)
 continue;
 int mini = INT_MAX;
 for(int ind = i; ind <= j; ind++)
 {
 int cost = cuts[i+1] - cuts[i-1] + dp[i][ind-1];
 + dp[ind+1][j];
 mini = min(mini, cost);
 }
 dp[i][j] = mini;
 }
 }
 return dp[1][c];
}

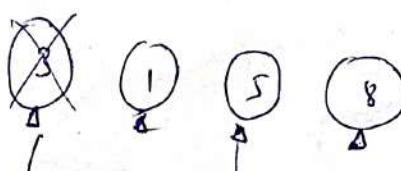
```

DP-51

## Burst Balloons | Partition DP

$$\text{arr}[] = [3, 1, 5, 8] \quad n=4$$

summation of those coins ↑



If we burst 5  
Left      right  
 $1 \times 5 \times 8 = 40$

$$1 \times 3 \times 1 = 3$$

for 1, now 3 is already burst

$$1 \times 1 \times 5 = 5$$

for 5

$$1 \times 5 \times 8 = 40$$

[8]

$$1 \times 8 \times 1 = 8$$

$$1 [3, 1, 5, 8] \quad 3 \times 1 \times 5 = 15$$

$$5 [3, 1, 8] \quad 3 \times 5 \times 8 = 120$$

$$3 [3, 8] \quad 1 \times 3 \times 8 = 24$$

$$8 [8] \quad 1 \times 8 \times 1 = 8$$

Method: Start from ~~min value~~ to ~~max value~~ → maximum  
Benefit.

[3, 1, 5, 8]

$b_1, b_2, b_3, b_4, b_5, b_6$  can be anyone. So we can think as many

$b_3 \times b_4 \times b_5$

↑

$b_1, b_2, b_3$

Subproblems

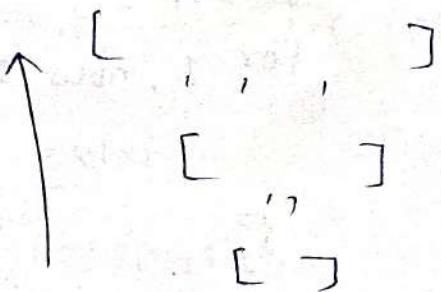
This will not work  
Solving independently

$[b_1, b_2, b_3, b_4], b_5, b_6]$

if we take subpart as an individual  
and try to solve for  $b_3$  subproblem  
then We can't become it  
depends on other portion by  
i.e.  $b_4$ .

Start thinking in opposite direction

$$\text{Ans} = 1 [ \boxed{3} \boxed{\cancel{1}, \cancel{2}, \cancel{4}} ] 1$$



$[3, \cancel{1}, 8]$

$[\cancel{2}, 8]$

$[\cancel{8}]$

$$3 \times 1 \times 5 = 15$$

$$3 \times 5 \times 8 = 120$$

$$\rightarrow 1 \times 3 \times 8 = 24$$

$$\leftarrow 1 \times 8 \times 1 = 8$$

$\therefore \{b_1, b_2, b_3, b_4, b_5, b_6\}$

$$= (a[i-1] \times a[find] \times a[j+1]) + f(j, find-1) + f(find+1, j)$$

$\{b_1, b_4\}$  or  $\{b_2, b_4\}$  or  $\{b_3, b_4\}$  or  
2nd last burst.

$[\cancel{1}, \cancel{2}, \cancel{3}]$

$[b_4]$

$$1 \times b_4 \times 1$$

(last  
burst)

```

 1 [a1, b2, b3, b4, b5] i
 0 1 2 3 4 5
 | | | | |
 i j lups
 f(i,j)
 {
 if (i >= j)
 return 0;
 for (int lnd = i + 1; lnd <= j; lnd++)
 maxi = INT_MIN;
 cost = a[i-1] * a[lnd] * a[j+1] + f(i, lnd-1) + f(lnd+1, j);
 mini = min(mini, cost);
 return mini;
 }

```

i      j  
 [N+1]     
  $\xrightarrow{\text{Recursion}}$  T.C. → T.C. → Expansion  
 ↓ Memorization     
 T.C. → N × N × M ≈ N<sup>3</sup>  
 S.C. → O(N<sup>3</sup>) + O(N)     
 ASC

Code :-

```

int f(int i, int j, vector<int> &a, vector<vector<int>> &dp)
{
 if (i > j) return 0;
 if (dp[i][j] != -1) return dp[i][j];
 int maxi = INT_MAX;
 for (int lnd = i + 1; lnd <= j; lnd++)
 {
 int cost = a[i-1] * a[lnd] * a[j+1] +
 f(i, lnd-1, a, dp) + f(lnd+1, j, a, dp);
 maxi = max(maxi, cost);
 }
 return dp[i][j] = maxi;
}

```

3

```

int m = a.size();
a.push_back(1);
a.insert(a.begin(), 1);
vector<vector<int>> dp(m+1, vector<int>(m+1, -1));
return f(1, m, a, dp);
}

```

## Tabulation

④

```

int maxcoins(vector<int>& a)
{
 int m = a.size();
 a.push_back(1);
 a.insert(a.begin(), 1);
 vector<vector<int>> dp(m+2, vector<int>(m+2, 0));
 for(int i = n; i >= 1; i--)
 {
 for(int j = 1; j <= n; j++)
 {
 if(i > j)
 continue;
 int maxi = INT_MIN;
 for(int ind = i; ind <= j; ind++)
 {
 int cost = a[i-1] * a[ind] * a[j+1]
 + dp[i][ind-1] + dp[ind+1][j];
 maxi = max(maxi, cost);
 }
 dp[i][j] = maxi;
 }
 }
 return dp[1][n];
}

```

TC =  $O(N^3)$   
SC =  $O(N^2)$

DP-52

## Boolean Expression to True / partition DP

Compression = "T | T & F"

"T & F"

"F"

You have to perform in such a way you will get true

"T | T & F"

"T | F"

① → no. of ways

"T"

" $T^F | F \& F \wedge T | F$ "  
(T) (F) (T & F) (T | F)

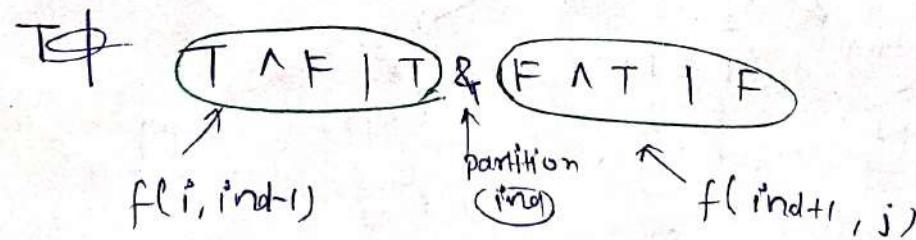
Similar to MCM

( ) | ( )

t | & ( ) ,

( ) , ? & ( ) ,

x ( ) | ( ) ,



left & right

both has to be true.

I'm looking for  
no. of ways left  
comes true OR no.  
of ways in which  
right is true.

left | right

$$T = x_1 \quad T = x_2 = x_1 x_2$$

$$F = x_3 = x_1 x_3$$

$$T \wedge T = F$$

$$F \wedge F = F$$

$$T \wedge F = T$$

$$F \wedge T = T$$

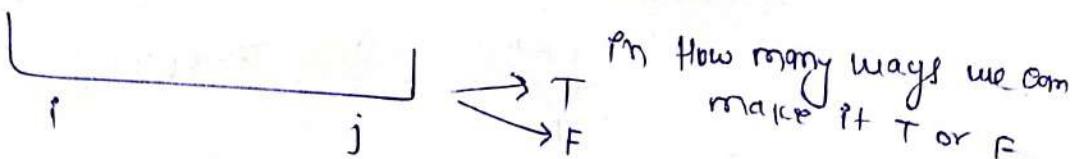
$x + y$   
left + right

$\text{left}$        $\text{right}$

$$T = x_1 \quad T = x_2$$

$$F = x_3 \quad R = x_4$$

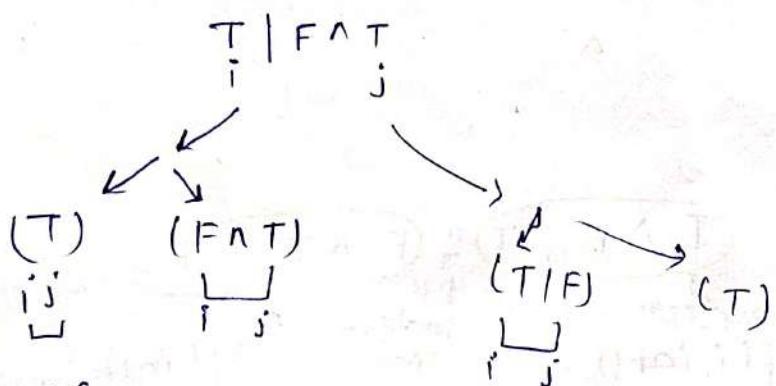
$x_1 \wedge x_4$   
 $x_3 \times x_2$



$f(i, j)$  is true

$f(i, j, \text{isTrue})$

{ if ( $i > j$ )  
return 0;



$\text{if } (i == j)$

{ if ( $\text{isTrue} == 1$ ) return  $a[i] == T$ ,

else return  $a[i] == F$  →  $i$  is false  
 $0$  is true.

looking  
for fault.

now and try to make partition.

int ways = 0  
for (int i = 0, j = 1; i < n; i = j + 1, j++)

{ LT = f(i, j, 1);

LF = f(i, j, 0);

RT = f(j + 1, n, 1);

RF = f(j + 1, n, 0);

ways = 0

if ( $Q[i][j] == '='$  &  $i < j$ )

$f[isTrue]$  ways +=  $L \times RT$ ; else ways +=  $(LT \times LF) + (LF \times RT) + (RF \times LF)$ ;

else if

'>'

else if

'|'

else

'^'

return ways;

So we are applying  
recursion

T.C. —  $O(\text{exponent}^N)$

$f(i, j, isTrue)$   
/ / /  
dp[N][N][2]

Code

int mod = 1000000007;

long long f(int i, int j, int isTrue, string &exp, vector<vector<vector<int>>&dp)

{ if (i > j) return 0;

long long &dp;

if (i == j)

{ if (isTrue)

{ return exp[i] == 'T'; }

}

else return exp[i] == 'F'; }

}

long long ways = 0;

for (int ind = i+1; i <= j-1; ind += 2)  
if (dp[i][j][isTrue] != -1) return dp[i][j][isTrue];

{ long long LT = f(i, ind-1, 1, exp, dp);

long long LF = f(i, ind-1, 0, exp, dp);

long long RT = f(ind+1, j, 1, exp, dp);

long long RF = f(ind+1, j, 0, exp, dp);

If (exp[ind] == "&")

{  
    if (isTrue) ways = (ways + (rT \* lT) % mod);  
    else ways = (ways + (lT \* rF) % mod + (rF \* lT) % mod + (rF \* lF) % mod);

    else if (exp[ind] == '|') (rF \* lF) % mod);

    if (isTrue)

        ways = (ways + (lT \* rT) % mod + (lT \* rF) % mod +  
                  (rF \* lT) % mod) % mod;

    else {

        ways = (ways + (rF \* lF) % mod) % mod;

    }

    if (isTrue)

        ways = (ways + (lT \* rT) % mod + (lF \* rT) % mod) % mod;

    else {

        ways = (ways + (lT \* rT) % mod + (lF \* rF) % mod) % mod;

    }

    return ways; dp[i][j][isTrue] = ways;

int evaluateExp (string & exp)

{ int n = exp.size();

vector<vector<long>> dp(n, vector<vector<long>> (n, vector<ll> (2, -1)));

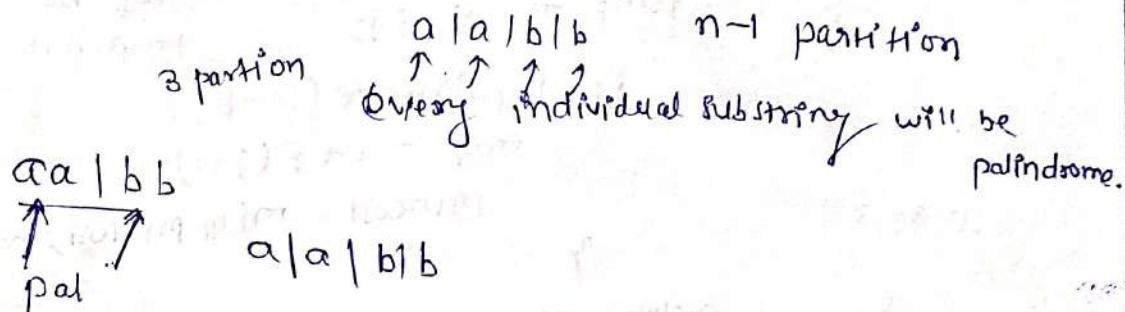
return f(0, n-1, 1, exp, dp);

DP-53

## Palindrome partitioning - II | Front partition

Given string = baba bc bad cede

small string str = aabb m=4



Such problems are generally solved using front partition

start from  
front

b [ a ] b / a / b / c b a d c e d e  
 ↑↑↑  
 palindrome.

1 + (abc b - - )  
 1 + (ab a b c b - - )

1 + (cb a d c e d e )

3 subproblems take the minimum no. of cuts.  
 take this one

1 + (abc b ad c e d e )



1 + (dc ed e)

Writing @ recurrence

1. Express everything in terms of index.
2. Express all the ways possibilities.
3. Take the min of all possibilities
4. write the base case.

Give me the min. cut to make all substring palindrome.

$f(i)$

```

 {
 if(i == n) return 0;
 temp = "" ; if(dp[i] != -1) return dp[i];
 for(j=i ; j < n ; j++)
 {
 temp += s[j];
 if(isPalindrome(temp))
 cost = 1 + f(j+1);
 minCost = min(minCost, cost);
 }
 }
 return minCost;
}

```

Recursion →  $\Theta T.C. = O(\text{Exponential})$

↓ Memoization

Changing parameters

$O \rightarrow N$

$dp[N]$

Code :-

```

int palindromepartitioning(string str)
{
 int n = str.size();
 vector<int> dp(n, -1);
 return f(0, n, str, dp);
}

int f(int i, int n, string &str, vector<int> &dp)
{
 if(i == n) return 0;
 if(dp[i] != -1) return dp[i];
 int minCost = INT_MAX;
 for(int j=i ; j < n ; j++)
 {
 if(isPalindrome(i, j, str))
 int cost = 1 + f(j+1, n, str, dp);
 minCost = min(minCost, cost);
 }
 dp[i] = minCost;
}

```

bool ispalindrome(int i, int j, string s)

{ while(i < j)

{ if(s[i] != s[j])  
return false;

i++;

j--;

}

return true;

}

Tabulation :-

int ① Base case ( $i == n$ ) / 0

②  $i = n-1 \rightarrow 0 \quad dp[n] = 0$ .

③ Copy the recurrence

int palindromepartitioning(string str)

{ int n = str.size();

vector<int> dp(n);

dp[n] = 0;

for (int i = n-1; i >= 0; i--)

{ int mincost = INT\_MAX;

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

{ if(ispalindrome(i, j, str))

{ int cost = 1 + dp[j+1];

mincost = min(mincost, cost);

}

dp[i] = mincost;

}

return dp[0];

}

T.C. -  $O(N^2)$   $O(N^2) = O(N^2)$

S.C.  $O(N) + O(N)$   
A.S.S.

T.C. -  $O(N^2)$

S.C. -  $O(N)$

DP-34

## Partition Array for Maximum Sum / front partition

$$arr[] = [1, 15, 7, 9, 2, 5, 10] \quad k=3$$

0 1 2 3 4 5 6

$$[15, 15, 7, 9, 9, 10, 10]$$

$$\underline{\text{Sum} = 77}$$

all elements of partition  
converted into max elements

$$[15, 15, 7, 9, 2, 5, 20]$$

$$[15, 15, 15, 9, 10, 10, 10]$$

$$\underline{\text{Sum} = 84 \uparrow\uparrow}$$

front partition logic

We have various ways → Try recursion

Rules :-

- Express everything in terms of index.
- Try partition possible from that index.
- Take the best partition.

$$[1, 15, 7, 9, 2, 5, 10]$$

↑  
ind

f(0)

give me the max sum if we  
have the array from 0.

f(3)

Base case :- Try out all partitions from that index

Base case :-

Whenever we cross crossing the k we will stop

f(ind)

Base case :-

maxAns = INT\_MIN;

$$15, 15, 7, 5 \\ j=1 \quad j=2 \quad j=3$$

len = 0; maxi = INT\_MIN;

for(j = ind; j < min(n, ind+k); j++)

len++;

maxi = max(maxi, arr[j]);

sum = (len \* maxi) + f(j+1);

maxAns = max(maxAns, sum);

return maxAns;

T.C. - O(exponential)

Memorization

I changing parameter

T.C. - O(N!) X O(k!)

S.C. - O(N!) + O(N!)

Another number n/another

```

int f(int ind, vector<int> &num, int k, vector<int> &dp)
{
 if(ind == num.size())
 return 0;
 int len = 0; if(dp[ind] != -1) return dp[ind];
 int maxi = INT_MIN;
 int MaxAns = INT_MIN;
 for(int j = ind; j < min(ind+k, n); j++)
 {
 len++;
 maxi = max(maxi, num[j]);
 int sum = len * maxi + f(j+1, num, k, dp);
 MaxAns = max(MaxAns, sum);
 }
 return dp[ind] = MaxAns;
}

```

```

int maximumSubarray(vector<int> &num, int k)
{
 int n = num.size();
 vector<int> dp(n, -1);
 return f(0, num, k, dp);
}

```

### Tabulation

#### ① Base Case

if (ind == n) return 0

i.e.  $dp[n] = 0$

#### ② Changing Variable

Original Ending index: ind = n-1 to 0  
 Opposite: ind = 0 to n

#### ③ Copy the recursion.

```

int maximumsubarray(vector<int> &num, int k)
{
 int n = num.size();
 vector<int> dp(n+1, 0);
 for (int ind = n-1; ind >= 0; ind--)
 {
 int len = 0;
 int maxi = INT_MIN;
 int maxAns = INT_MIN;
 for (int j = ind; j < min(ind+k, n); j++)
 {
 len++;
 maxi = max(maxi, num[j]);
 int sum = len * maxi + dp[j+1];
 maxAns = max(maxAns, sum);
 }
 dp[ind] = maxAns;
 }
 return dp[0];
}

```

DP-55

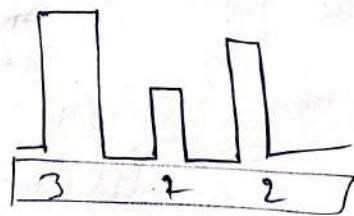
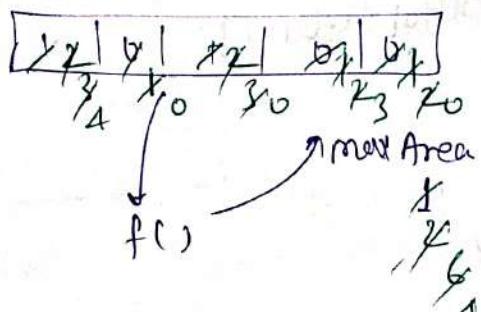
## Maximum Rectangle area with all 1's / upon

Rectangle

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |

$$2 \times 3 = 6 \text{ (maximum)}$$

We will solve this question using Histogram



if we find 0 then

it becomes 0

else previous height will be added.

$$\text{max} = 6$$

Code :- int maximalAreaOfSubmatrixOfAll( vector<vector<int>&mat, int i, int m)

```

 {
 int maxArea = 0;
 vector<int> height(m, 0);
 for(int i=0; i<m; i++)
 {
 for(int j=0; j<m; j++)
 {
 if(mat[i][j] == 1)
 height[j]++;
 else
 height[j] = 0;
 }
 int area = largestRectangleArea(height);
 maxArea = max(area, maxArea);
 }
 }

```

return maxArea;

@Aashish Kumar Rayak

int largestRectangleArea (vector<int> &histo)

```
{ stack<int> st;
 int maxA = 0;
 int n = histo.size();
 for (int i = 0; i <= n; i++)
 {
 while (!st.empty() && (i == n || histo[st.top()] >=
 histo[i]))
 {
 int height = histo[st.top()];
 st.pop();
 int width;
 if (st.empty())
 width = i;
 else
 width = i - st.top() - 1;
 maxA = max(maxA, width * height);
 }
 st.push(i);
 }
 return maxA;
}
```

We are trying to remember the heights  
that's why this is kept in dp.

T.C.  $O(N \times M)$   
T.C.  $O(N \times (m+n))$   
S.C.  $O(N)$

DP-56

## Count No. of square Submatrices

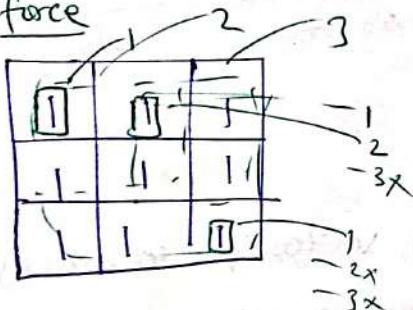
|   |   |   |
|---|---|---|
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 0 |

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 |

Size 1 - 6  
Size 2 - 1

size 1 - 10  
size 2 - 4  
size 3 - 1

Brute force



We can do better using dp.

Try to create dp of similar size

$dp[i][j]$

How many squares end at Right Bottom at  $(i,j)$

sum all of them we will get our ans.

but How we will fill this array.

|   |   |   |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

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

1 sq. he himself.  
he is right most bottom  
 $= 2$   
yuo

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

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

first row and first column they themselves are size 1.

$$\min = 2 + 1 = 3$$

If there are 1 and 1 this is 1.  
then it will form square of size 2.

Now add of of them and return the ans.

1 . 1 0  
1 1 1 →  
1 1 0

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

now add all of them.

$$\min(1, 1, 1) = 1 + 1 = 2$$

$$dp[i][j] = \min \left[ \begin{array}{l} dp[i-1][j], dp[i-1][j-1], dp[i][j-1] \\ \text{upper} \quad \quad \quad \text{diagonal} \quad \quad \quad \text{left} \end{array} \right] + 1$$

Code

```
int countSquares (int n, int m, vector<vector<int>> &arr)
{ vector<vector<int>> dp(n, vector<int>(m, 0));
 for (int i = 0; i < m; i++)
 dp[0][i] = arr[0][i];
 for (int i = 0; i < n; i++)
 dp[i][0] = arr[i][0];
}
```

```
for (int i = 1; i < m; i++)
 for (int j = 1; j < m; j++)

```

```
{ if (arr[i][j] == 0)
 dp[i][j] = 0;
else
 {
```

```
 dp[i][j] = 1 + min (dp[i-1][j], min (dp[i-1][j-1],
 dp[i][j-1]),
 dp[i-1][j-1]);
 }
```

int sum = 0;

```
for (int i = 0; i < n; i++)
 for (int j = 0; j < m; j++)
 sum += dp[i][j];
return sum;
```

add all  
of them  
in diff  
dp matrix  
and  
return.