DP

Maximum sum of Non adjacent elements

Q You are given an array of n integers. Return the maxm sum of subsequence with the constraint that no 2 elements are adjacent in given arraylist.

1 2 4 → 0 : 5
2 1 4 9 → 0 : 11

S1 Let's try all subsequence with given constraint

Recursion → indexes → do all stuff on index → return best

Pick one with max sum

f(ind) → max sum from picking (0 ... ind)

print all subsequence ⇒ pick / not pick

arr, dp

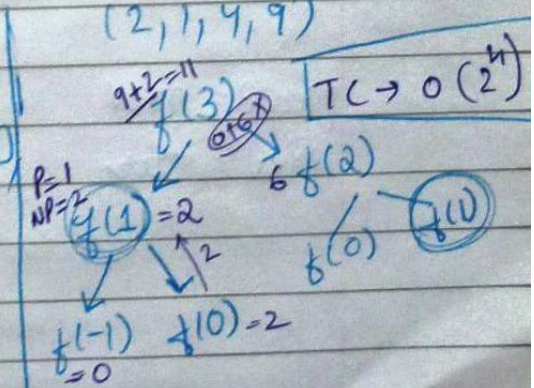
```
f(ind) {
  if(ind==0) return a[ind];
  if(ind<0) return 0;
  if(dp[ind] != -1) return dp[ind];
  pick = a[ind] + f(ind-2);
  not pick = 0 + f(ind-1);
  return dp[ind] = max(pick, not pick);
} // dp[n] because original array also has n elements.
```

T(n) → O(2^n)
S(n) → O(n) + O(n)
stack dp

S2 optimize Recursion ⇒ Overlapping subproblems ⇒ Memoization

0 1 2 3
(2, 1, 4, 9)

T(n) → O(2^n)



Tabulation

dp [n] → 0

prev

dp[0] = a[0];

int neg = 0;

prev

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

dp part

(if (i>1) else a[i]+0;)

take = a[i] + dp[i-2]; prev2

nontake = 0 + dp[i-1]; prev

curi

dp[i] = max (take, nontake);

prev2 = prev

prev = curi

}

return dp[n-1] prev

T.C O(n)

S.C O(n)

Space Optimization

prev2 prev (i)

(i-2) (i-1) (i)

. . . . . .

dp(i-2) dp(i-1)

O(N)

O(1)

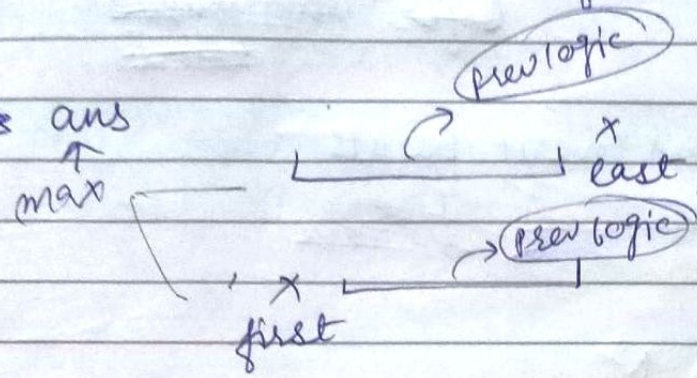
prev2 = prev

prev = cur

House Robber 2

→ Same Q as previous. But here houses are arranged in a circle.

So, we can either pick first or last



Ninja's Training

Ninja is taking 'N' days training schedule. There are 3 activities, each day he can perform any 1 or 3 activities. Each activity has some merit points. He can't perform the same activity on consecutive days. Find the max profit he can earn

	A0	A1	A2
D0	(10)	5	0	1
D1	5	(100)	11

Ans → 110

greedy fails ⇒ try all possible ways ⇒ Recursion
↳ express in term of index
↳ do all stuff on index
↳ find max

f(day, last)

day → 1 2 3 to know what task to perform here, we need to know what we did on previous day

last 0 → task 0 was done
1 → " 1 was done
3 → no task was done

f(n-1, 3){ // max merit points on array 0 → n-1 ~~befor this day 3 starts~~.

f(2,1) max merit points
⇓
this means we are checking on day 2 and the task day 3 has performed was A1

Recursion starts from n-1.
start with f(n-1, 3)
all tasks get tried

~~day~~ ind ~~ind~~ last
f(~~day~~, ~~ind~~) {
  if(ind == 0) {
    maxi = 0;
    for(i = 0 → 2) {
      if(i != last)
        maxi = max(maxi, task[0][i])
    }
    return maxi;
  }
  if(dp[ind][last] != -1) return dp[ind][last];
  for(i = 0 → 2) {
    if(i != last) {
      points = task[ind][i] + f(ind - 1, i)
      maxi = max(maxi, points);
    }
  }
  dp[ind][last] = maxi;
  return maxi;
}

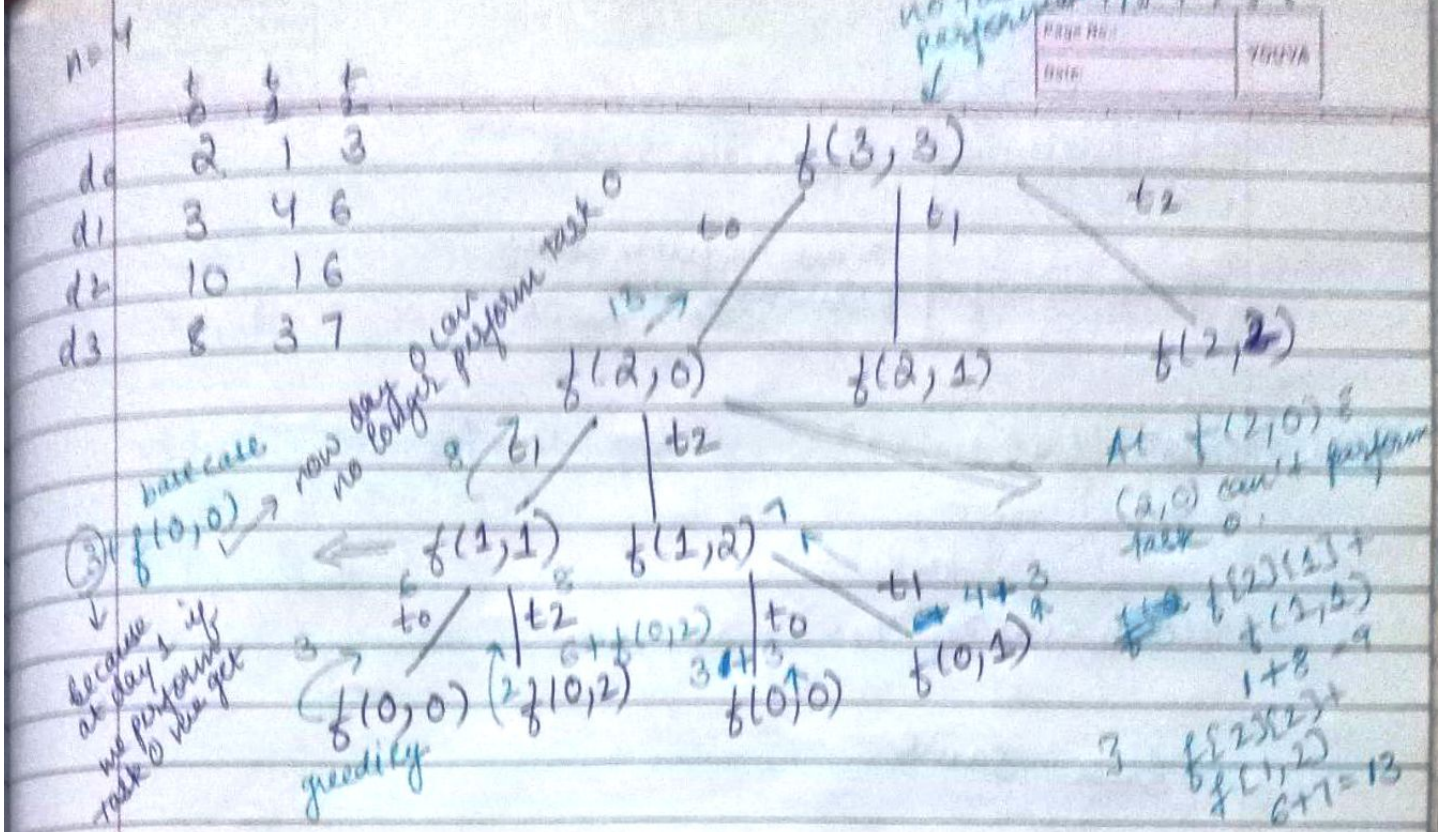
because on last day we can greedily pick best. As there is no next day to consider.

for previous day and tell him this i-th task was performed.

Total steps
1 + O(N*4)*3
3 + O(N)
+ O(N*4)

n = 4

	t0	t1	t2
d0	2	1	3
d1	3	4	6
d2	10	1	6
d3	8	3	7



Overlapping SubProblems & Memoization

Changing parameters → day, last (no task)

dp[N x 4] — N → 4 → (0, 1, 2, 3)

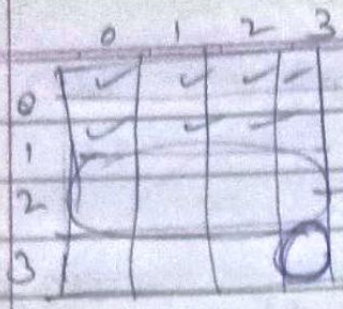
Tabulation :

```
int dp[N][4]
dp[0][0] = max(arr[0][1], arr[0][2])
dp[0][1] = max(arr[0][0], arr[0][2])
dp[0][2] = max(arr[0][0], arr[0][1])
dp[0][3] = max(arr[0][1], arr[0][2], arr[0][0])

for(int i = 1 → n-1)        // day
{
    for(last = 0 → 3)       // included
    {
        dp[i][last] = 0;
        int mani = 0;
        for(int task = 0; task < 3; task++)
        {
            if(task != last)
            {
                point = points[i][task] + dp[i-1][task];
                mani = max(mani, point)
            }
        }
        dp[day][last] = mani
    }
}
return dp[n-1][3];
```

base case { ind → 0 }

for (ind → 0) (0,0) (0,1) (0,2) (0,3) make 4 cases can't come

task acts as last for dp table



dp[day][last] = + day[day-1][task]

vector<int> prev(4,0);

//Base same

for(int day=1; day<n; day++){
vector<int> temp(4,0);
for(int last=0; last<4; last++){
temp[last]=0;

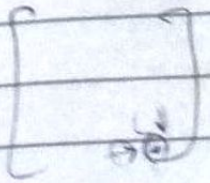
for(int task=0; task<3; task++){
if(task != last){
temp[last] = max(temp[last], points[day][task] + prev[task]);

}
}
}
prev = temp;
}
return prev[n-1][3];

DP on Grids / Matrix

Q Total Unique paths

You are at pt A top left cell of a matrix. You have to reach B(n-1)(m-1). find no. of unique paths. (R & D) movements possible



$(0,0) \rightarrow (n-1, m-1)$

recursion

$f(i,j)$ → no. of unique ways

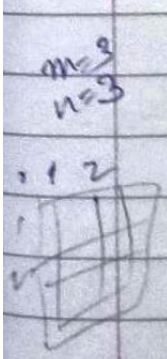
start from here $f(n-1, n-1)$

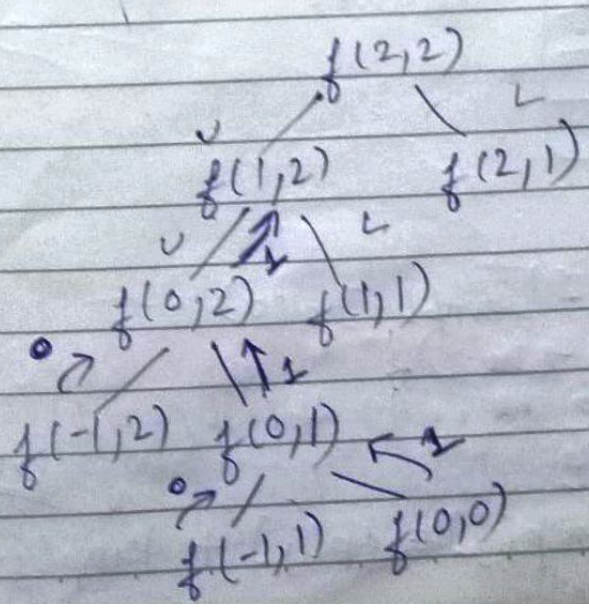
TC → $2^{m \times n}$

```
f(i,j){
if(i==0 && j==0){
    return 1;
}
if(i<0 || j<0) return 0;
up = f(i-1, j);
left = f(i, j-1);
return up + left;
}
```

dp[i][j] = ?

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





changing parameter

dp[m][n] = {-1}

① declare base case
② express various states in for loops
③ write recurrence

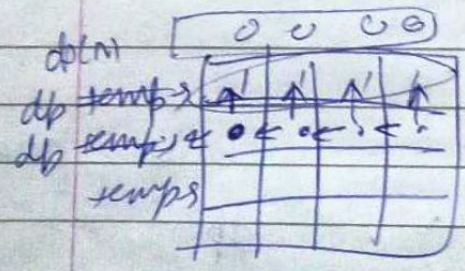
Tabulation

```
dp[n X m]

dp[0][0] = 1;
for(i=0 → n-1)
 {
  for(j=0 → m-1){
    if(i==0 && j==0) continue;
     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;
 }
print dp[n-1][m-1]
```

TC O(n×m)
SC O(n×m)

Space optimization
prev (now) & prev column

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




```
dp[n] = {0}
for(i=0 → n-1) temp[n] = {0}
  { for(j=0 → m-1){
     if(j==0 && i==0) dp[i][j] = 1;
       else {                    ↓ j>0
         temp[j] = dp[j] + temp[j-1]
  } } dp = temp;
```

Minimum Path Sum

Q Each point on the grid has some cost associated with it. Find the path from (0,0) to (n-1, m-1) with minimum cost.
Down & Right only allowed

1	3	1
1	5	1
4	2	1

try all paths.
→ keep indexes
→ do all stuff
→ return min
f(n-1, m-1)

Recursion

```
f(i,j) {
  if(i==0 && j==0) return a[0][0];
  if(i<0 && j<0) return INT_MAX;
  if(dp[i][j]!=-1) return dp[i][j];
  up = a[i][j] + f(i-1,j);
  left = a[i][j] + f(i,j-1);
  dp[i][j] =
  return min(up, left);
}
```

Memoization

dp[n][m] = -1, TC: O(path length) (m-1)(n-1)

Tabulation

(i,j) 0 → n-1 0 → m-1

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

Longest Repeating Character Replacement

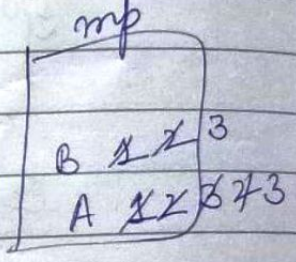
Given a string s and integer k. You can choose any char in string and change it. You can perform this operation at most k times.

Return the length of max substring you can get after performing these operations.

s = "A A B A B B A" k = 1

Output → 4

Let s = A A B A B B A. k = 2



To know if any substring is valid

(r - l + 1) − maxCount = char we need to replace ≤ k

↑ window length

↓ char that occurs max in that window

↑ operations allowed

For every r, add it to mp and inc. count

res = 3

at index 2,

3 − 2 = 1 ≤ 2 valid

at ind 5, 6 − 3 = 3 ≤ 2 X not valid

Inc l, and dec count of s[l]

5 − 3 = 2 ≤ 2 true

TC: O(n), X O(26) because finding maxCount each time in mp

```
maxCount ( mp <char, int> mp) {
    int ans = 0;
    for (auto i : mp) {
        ans = max ( ans, i.second);
        }
    return ans;
    }

int characterReplacement ( string s, int k) {
    int n, l=0, r=0, res = 0;
     mp.
    while (r < n) {
        mp[s[r]) ++;   int same = maxCount(mp
// window is not valid  while ( (r-l+1) - same) > k) {
                mp[s[l]]--;
                l++;
            }
        res = max( (r-l+1), res);
        r++;
    }
    return res;
  }
};
```

Fixed And Variable pa...

Date: YOUVA

Q You are given a triangular array 'TRIANGLE'. Your task is to ~~reach~~ return the min path sum to reach from top to bottom row.

The triangle has n rows and ith row will have i+1 elements.

You can only move to the adjacent no. of row below each step. At ind j on row i, then you can move to i or i+1 inden on row j+1 in each step.

→ Represent (i,j)
→ Explore all paths (↓ ↘) (i+1,j) (i+1,j+1)
→ Pick min

fixed

1
2 3
3 6 7
8 9 6 10

Variable ⇒

TC: $2^{1+2+\cdots+n}$
SC: O(N) stack space

f(0,0) ⇒ min path sum from (0,0) to last row

```
f(i,j) {
  if(i == n-1) return a[n-1];
  if(dp[i][j] != -1) return dp[i][j];
  d = a[i][j] + f(i+1, j);
  dg = a[i][j] + f(i+1, j+1);
  return dp[i][j] = min(d, dg);
}
dp[n][n];
```

// we will never go out of boundary so no need for other base case

TC: O(n×n) (no. of states)
SC → no of states O(n²) (rec stack)
O(n²)

Thumb rule: Tabulation is always opposite of recursion

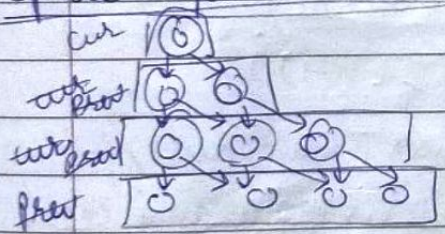
Page No.: Date: YOUVA

Tabulation: Recursion → 0 → (n-1); opposite here (n-1) → 0

```
dp[n][n];
for(j=0; j<n; j++){
  dp[n-1][j] = a[n-1][j];
}
for(i=n-2; i>=0; i--){
  for(j=i; j>=0; j--){
    d = a[i][j] + dp[i+1][j];
    dg = a[i][j] + dp[i+1][j+1];
    dp[i][j] = min(d, dg);
  }
}
cout(dp[0][0]);
```

TC → O(n×n)
SC → O(n²)

space optimization:



TC: O(n×n)
SC: O(n)

```
front[n];
for(j=0; j<n; j++){
  front[j] = ar[n-1][j];
}
for(i=n-2; i>=0; i--){
  cur[n];
  for(j=i; j>=0; j--){
    d = a[i][j] + front[j];
    dg = a[i][j] + front[j+1];
    cur[j] = min(d, dg);
  }
  front = cur;
}
return front[0];
```

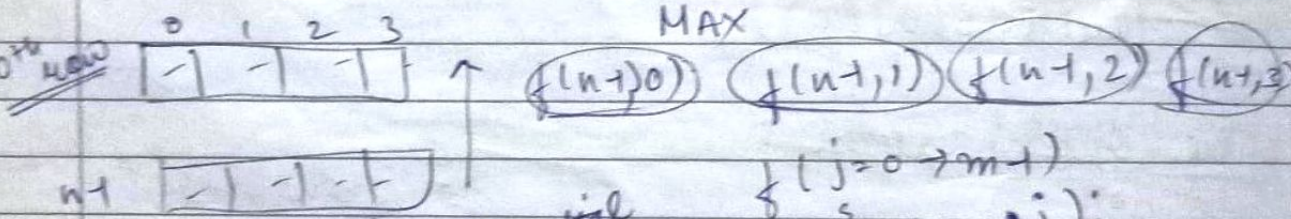
Max^m Path Sum In Matrix

You have a n * m matrix filled with integer numbers, find the max sum that can be obtained from any cell in row 1 to any cell in row N.

From any cell you can move:
directly below, directly below L or R.

(row+1, col) (row+1, col+1) (row+1, col-1)

1	2	10	4
100	3	2	1
1	1	20	2
1	2	2	1



TC: 3^m ≈ exponential
SC: O(N) RSS

```
for(j=0->m-1)
{ ans = f(n-1, j);
  maxi = max(maxi, ans);
}
```

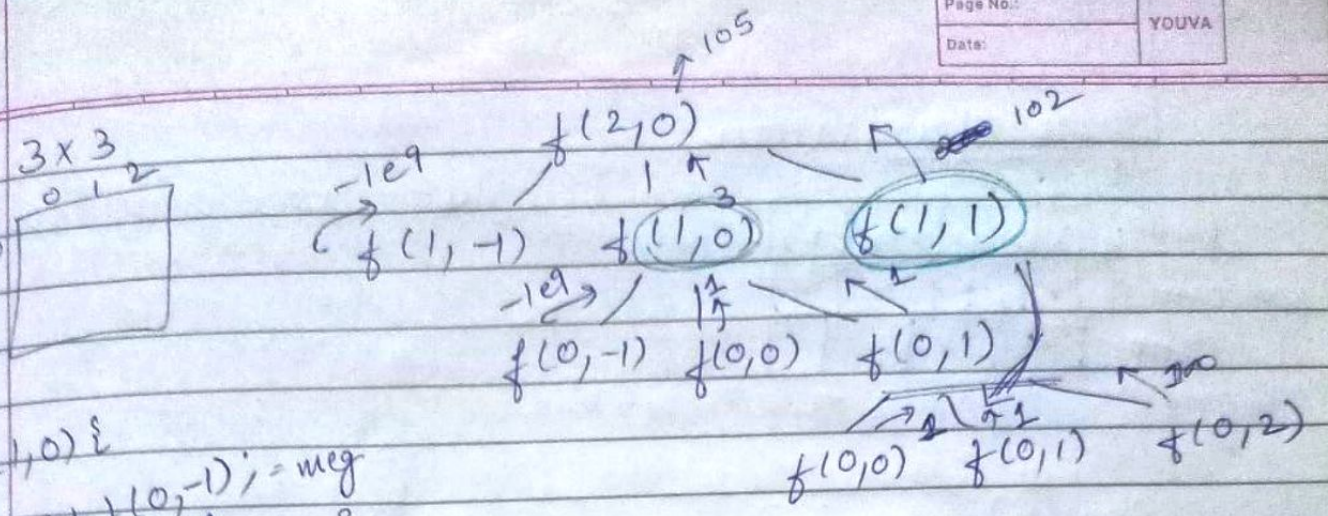
dp[n][m];

f(i, j) → max path sum to reach i,j from any cell from first row

```
f(i,j) {
  if(j<0 || j>=m) return -1e9;
  if(i==0) return a[0][j];
  if(dp[i][j]!=-1) return dp[i][j];
  int s = a[i][j] + f(i-1, j);
  int ld = a[i][j] + f(i-1, j-1);
  int rd = a[i][j] + f(i-1, j+1);
  return dp[i][j] = max(s, ld, rd);
}
```

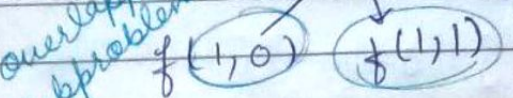
↑ col<0 ↑ col>m-1

TC: O(n x m)
SC: O(n x m) + O(n) RSS




```
f(i,j){
ld = 2+f(0,-1); = neg
s  = 2+f(0,0); =3
rd = 2+f(0,1); =3
}
```

overlapping subproblems

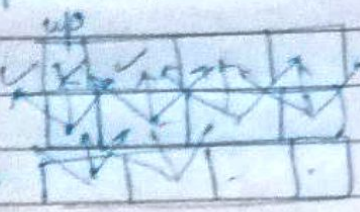


1. base cases
2. observe states i,j and convert to loops.

Tabulation:

```
dp[n][m];
for(j=0;j<m;j++){
    dp[0][j] = a[0][j];
}
for(int i=1; i<n; i++){                TC: O(n*m)
    for(j->0 to m-1){                  SC: O(N*M)
        u  = a[i][j] + dp[i-1][j];
if(j-1>=0)  ld = a[i][j] + dp[i-1][j-1];
if(j+1<m)   rd = a[i][j] + dp[i-1][j+1];
        dp[i][j] = max(u,ld,rd);
    }
}
maxi = dp[n-1][0];
for(j=1->m-1){
    maxi = max(maxi, dp[n-1][j]);
}
return maxi;
```

Space optimization (just store prev row)



prev(n), cur[n] // initial row

for (int i = 1 → n-1)

for (j = 0 → m-1) {

u = a[i][j] + prev[j];

ld = a[i][j] + prev[j-1];

rd = " + prev[j+1];

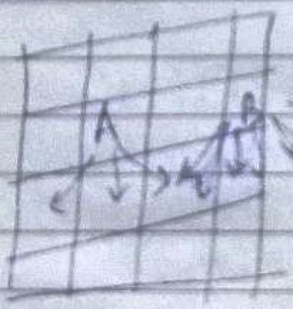
cur[j] = max( )

}

prev = cur;

}

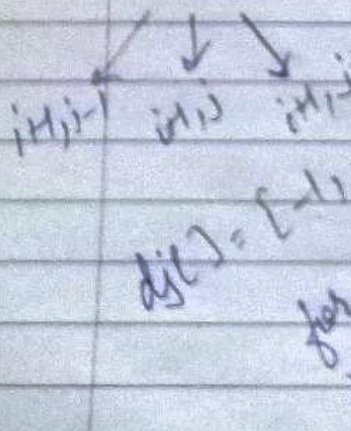
return max in prev;



for every move of Alice, Bob can have 3 corresponding moves.

⇐

3×3 = 9 combos



dj[] = {-1, 0, +1}

for(i → 1 → n)

for(j → -1 → +1)

TC: $(3^n \times 3^n)$

SC: O(n) ≈

3D - DP (On Grids)

Q Ninja has a grid 'R' x 'C'. Each cell of the grid contains some chocolates.
Initially, Alice is standing at (0,0) and Bob is at (0, C-1). Each can move from their current cell to the one just below. When anyone passes a cell, he will take all chocolates in it.

If Alice or Bob is at (i, j) then they can move (i+1, j) (i+1, j-1) (i+1, j+1)

Your task is to find max no of chocolates Ninja can collect with the help of his friends.

Because in any case both A & B will move 1 row down. So, i will always be same

A			B
2	3	1	2
3	4	2	2
5	6	3	5

12 + 9 = 21 Ans

// If a cell is common in both paths its counted once.

(0,0) (0, m-1)

fixed start pt ⇒ Variable end. pt

(1) Express everything in terms of $i_1 j_1$ $i_2 j_2$. (Alice + Bob)
(2) Explore all paths ↙ ↓ ↘
(3) Max sum possible

f(0, 0, 0, m-1) (A, B)

```
f(i, j1, j2) {
  if(j1 < 0 || j1 >= m || j2 < 0 || j2 >= m) return -1e8;
  if (i == n-1) {
     if(j1 == j2) return a[i][j1];
     else return a[i][j1] + a[i][j2];
  } maxi = 0;  if(dp[i][j1][j2] != -1) return dp[i][j1][j2]
  for(dj1 -> -1 to +1) {
     for(dj2 -> -1 to +1) {
  if(j1 == j2) maxi = max(maxi, a[i][j1] + f(i+1, j1+dj1, j2+dj2))
     maxi = max(maxi, a[i][j1] + a[i][j2] + f ↓ ));
  } return dp[i][j1][j2] = maxi;
```

Memoization: i j1 j2 → [N][M][M]

TC: O(n×m×m) × 9

(1) Base case
(2) Express states in for loops i, j1, j2

n-1 to 0 (0 to m-1)

Tabulation:

```
dp[n][m][m];
for(j1 → 0 to m-1) {
  for(j2 → 0 to m-1) {
if(j1 == j2) dp[n-1][j1][j2] = grid[n-1][j1];
   else dp[n-1][j1][j2] = grid[n-1][j1] +
                          grid[n-1][j2];
  }
}
for(i = n-2; i>=0; i--) {
  for(j1 = 0 → m-1) {
    for(j2 = 0 → m-2) {
      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][j1]
          else value = grid[i][j1] + grid[i][j2];
          if(j1+dj1 >= 0 && j1+dj1 < m && j2+dj2 >= 0
                                       && j2+dj2 < m)
            value += dp[i+1][j+dj1][j+dj2];
          else value += -1e8;
          maxi = max(maxi, value);
        }
      dp[i][j1][j2] = maxi;
    }
  }
}
return dp[0][0][m-1];
}
```

DP On Subsequences / Subsets & target

subsequence
↙ ↘
contigous non-contigous
{1,2,3} → {1,3} ✓
{3,2} ✗✗

Q. Subset sum equal to K

you are given array of n positive integers and an integer K. check if there exists a subset in arr with sum equal to K.

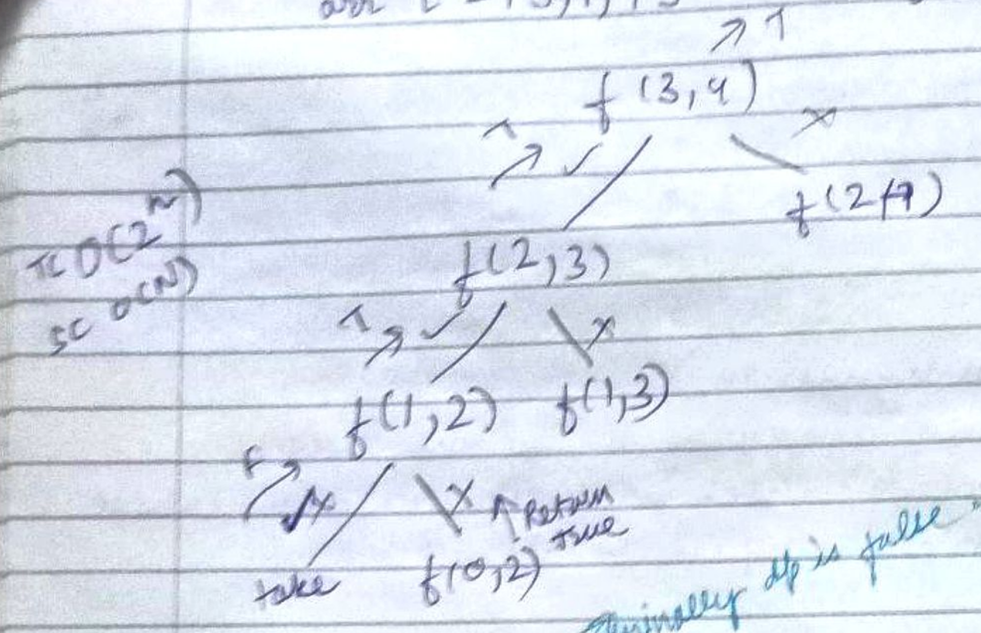
⇒ Recursion
① Express ind, target
② Explore possibilities of that ind
pick / not pick
③ return T/F

f(ind, target){ f(n-1, target) 1,2,3,4 k=7

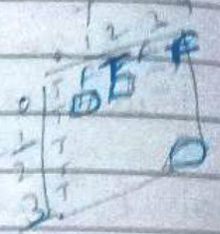
if(target==0) return true;
if(ind==0) return (a[0]==target);
if(dp[ind][target]!=-1) return dp[ind][target];
bool not take = f(ind-1, target);
bool take = false;
if(target >= a[ind])
take = f(ind-1, target-a[ind]);
dp[ind][target] =
return (take) or (not take);

memoization dp[ind][target+1]
TC: O(N×target)

arr [2, 3, 1, 1]   target = 4



Initially dp is false.



Tabulation:

for (i = 0 to n-1)   // for every ind if targ is 0 its true;
  dp[i][0] = true;

dp[0][a[0]] = true;

TC O(N×target)
SC O(N×target)

for (ind → 1 to n-1)
  for (1 to target) {
    bool notTake = dp[ind-1][target];
    if (arr[i] >= target) bool Take = dp[ind-1][target - arr[ind]]
    dp[ind][target] = take | NotTake
  }
}
return (n-1, target);
}

Space optimization

i=0 [T | F F T F F]
i=1 [T | - - - -] → prev
cur

(... 1 & are possible)

Page No.: Date: YOUVA

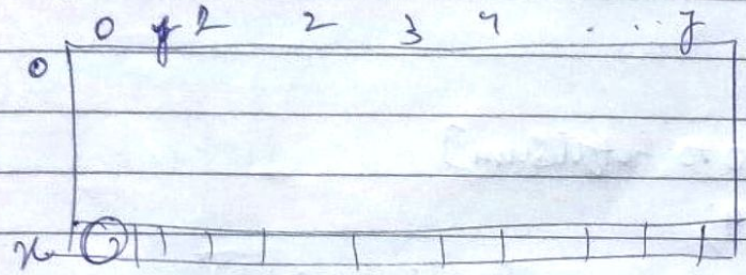
Q Partition a ~~subset~~ set into 2 subsets such that the difference of subset sums is minimum.

$|3 - 7| = |4|$

[1, 2, 3, 4] → {(1,2) (3,4)}
{(1,3) − {2,4)} = |2|
{(1,4) − (2,3)} = |0| Ans.

ex: {3, 2, 7}
total = 12

(3) (2) (1) (3,2) (3,7) (2,7) possible combinations



// In Previous question if (x,2) or (x,3) or (x,y) tells if y sum is possible at index x.

So, if total sum is 12

	✓		✓	✓	✓		✓	✓	✓	✓
	0	1	2	3	4	5	6	7	8	9	10	11	12
S_1	0		2	3	4	5	6	7	8	9	10	11	12
S_2	12		10	9		7		5		3	2		0

possible

(Repeats After this)

So check for every (3,0) (3,1) (3,2) ... (3,12) if its possible.

we can find sum of S_2 by total − S_1.
And take min difference pair.

Code:

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

vector<vector<bool>> dp(n, vector<bool>
```

```
for (int i=0; i<n; i++){
        dp[i][0] = true;
}
if arr[0] <= totalSum
dp[0][arr[0]] = true;

for (int ind=1; ind<n; ind++){
 for (int target = 1; target <= totalSum; 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;
   }
}
}

// dp[n-1][col -> 0 to totalSum]
int mini = 1e9;
for (int s1=0; s1<= totalSum/2; s1++){
  if (dp[n-1][s1] == true){
   mini = min(mini, abs((totSum - s1) - s1));
  }
}
return mini;
```

Date: YOUVA

Q Given an array of the integers, return how many ways of selecting elements from array such that sum of choosen elements is equal to target no. "tar".

I: [1 2 2 3] tar=3
O: {1,2} {1,2} {3}

```
f(ind, s) {
  if(ind==0){
    return (a[ind]==s);
  }
  if(s==0) return 1;
  notPick = f(ind-1, s);
  pick=0;
  if(arr[ind]<=s) pick = f(ind-1, s-arr[ind]);
  return notPick + pick;
}
```

f(n-1,s) → this will be not pick case. el=4 because even if el=4 & sum=0, I can not pick and meet cond^n

TC O(2^n)
SC O(n) ASS

before

```
if(dp[ind][s]!=-1) return dp[ind][sum];
return dp[ind][sum]=
```

O(N x sum) TC
O(N x sum) + O(n)

Memoization

dp[n][sum+1] = {-1}
ind

Tabulation:

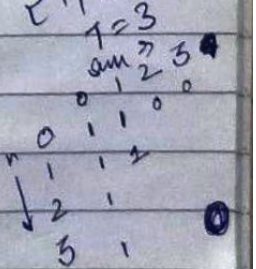
1 Base case 2. Look at changing parameters & write nested loops 3. copy recursion

int dp[n][sum+1]; = {0} → 2mp, because here we are adding elements so can't be -1.

```
for(0 to n-1) { dp[i][0] = 1; }
dp[0][a[0]] = 1;
for(ind = 1 to n-1){
  for(sum = 1 to s) {
    // copy the Recurence
  }
}
return dp[n-1][s];
```

if(a[0]<=s) otherwise everyrow, because dp[n][s+1]

en
{1,2,3}
T=3



In prev ques, (curr ques)

If case is (0,0,1)
0's are included in array after base case

basic logic
How many ways we can represent zeroes x curr_ans.

4x1 = 4

0 → 0,1
0 → 1,0
00 → 0,0,1
{} → 1.

Remove (If s==0) because we have to go deep & check all

```
If (ind==0){
    If (s==0 && a[ind]==0)
        return 2;
    If (s==0) {
        return 1;
    }
```

because that a[0]=0 has 2 ways either it gets picked or doesn't get picked either way its fine.

because If sum is 0 and any other elements is a[ind] we will not pick it.

If (s==a[0]) return 1;

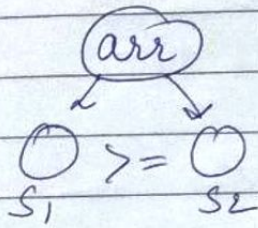
Tabulation

```
int n = num.size()
vector<vector<int>> dp(n, vector<int>(tar+1,0));
If (num[0] == 0) dp[0][0] = 2;
else dp[0][0] = 1;
If (num[0] != 0 && num[0] <= tar) dp[0][num[0]] = 1;
```

If [0] and sum == 0 we can both ✓ ✗
If [?] and sum = 0 we can ✗
If [5] and sum == 5 we can ✓

Partitions with Given Difference

Given an arr, partition it into 2 subsets (possibly empty) such that their union is the orginal array. Let the sum of elements of these subsets be S_1 and S_2.
Given a difference 'Δ' count the no. of partitions in which S1 is >= S2 and diff b/w S_1 & S_2 = Δ.



$$S_1 - S_2 = \Delta$$

arr = {5, 2, 6, 4} $\Delta = 3$

(5,2) S_2 (6,4) S_1 10 − 7 = 3
7 10

$S_1 - S_2 = \Delta$ $S_1 > S_2$

TotSum − S_2 − S_2 = Δ S_1 = totalSum − S_2

TotSum − Δ = 2 × S_2

$$S_2 = \frac{TotSum - \Delta}{2}$$

count of subset $\left(\frac{totSum - \Delta}{2}\right)$ ↑ Target (Modified)

Constraints
$0 \le arr[i]$ → so totSum − Δ cannot be negative (how can sum of subset that has the no. be neg).

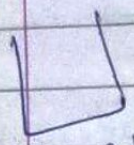
↑ they can't be fraction totSum − Δ has to be even.

(to this code with 0 cases in mind)

0/1 Knapsack

n=3

wt →	3	4	5
val →	30	50	60



Bag → W=8

Total → 60+30 = 90

Try all combinations ⇒ take best

↓ Recursion

→ Express everything as index (ind, bag weight W)

→ Explore all possibilities

Pick ✓ Not pick X

→ Pick Max (all possibilities)

TC 2^n / SC O(n)

```
f(ind, W)
if(ind==0){ if(wt[0]<=W) return val[0]; else return 0; }
if(dp[ind][W]!=-1) return dp[ind][W];
notTake = 0 + f(ind-1, W);
take = INT_MIN;
if(wt[ind] <= W)
    take = val[ind] + f(ind-1, W-wt[ind]);
return dp[ind][W] = max(take, notTake);
```

f(n-1, W)

Let 3 2 4

30 40 60

ind 0 1 2

f(2,6)

⇓

till index 2, what val will you get with bag W = 6

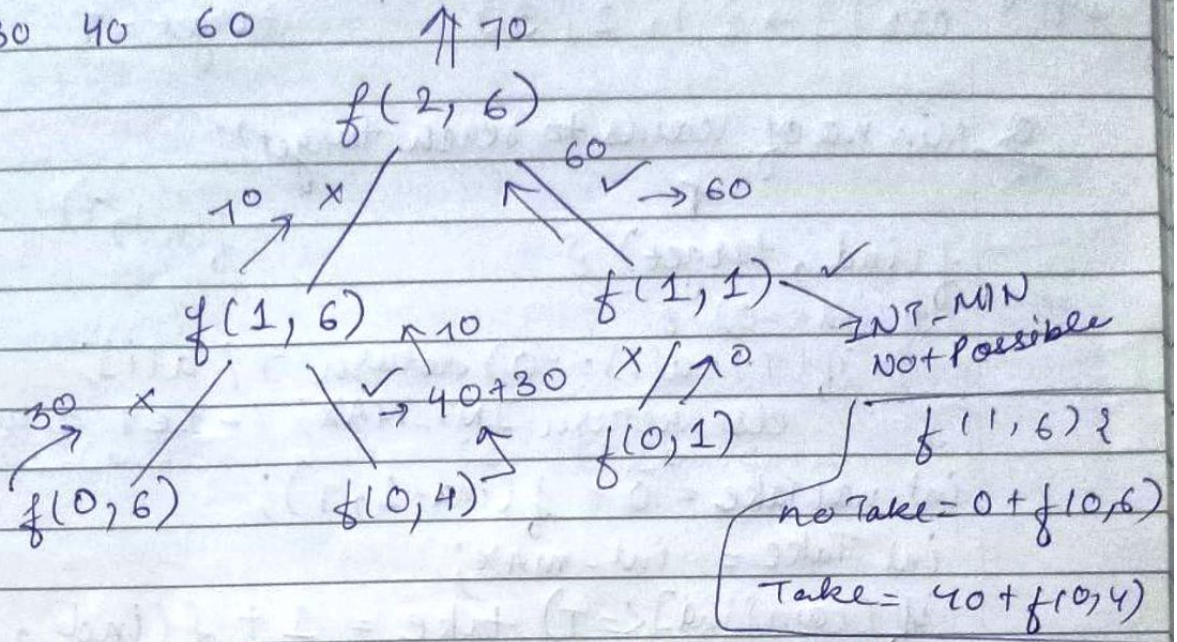
Memoization dp[N][W+1];

TC: O(n×W)

SC: O(n×W) + O(N) ASS

	0	1	2
wt	3	2	5
val	30	40	60

W = 6

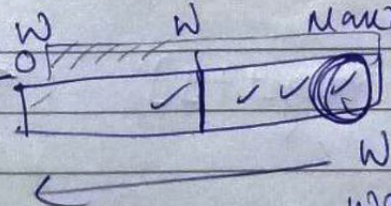


Tabulation: 1. base case 2. Changing Parameters 3. same Recurrence in loop (W, ind)
0→W 1 to n-1

```
int dp[N][W+1]; = {0}
for (i = wt[0] → W) dp[0][i] = val[0];   // because if wt coming is more than its own wt we can take it

for (int ind = 1; ind ≤ n-1; ind++) {
    for (int W = 0; W ≤ W; W++) {          // W = bagweight
        int notTake = 0 + dp[ind-1][W];
        int take    = val[ind] + dp[ind-1][W - wt[ind]];   // if (wt[ind] <= W)
        dp[ind][W] = max(take, notTake);
    }
}
return dp[n-1][W];
```

Space optimize ⇒ ind-1 just use cur & prev

More optimal ⇒ ind cur



dp[ind-1][W - wt[ind]]

W (if we fill like this) works because last one is dependent of ind-1 row

Minimum Coins

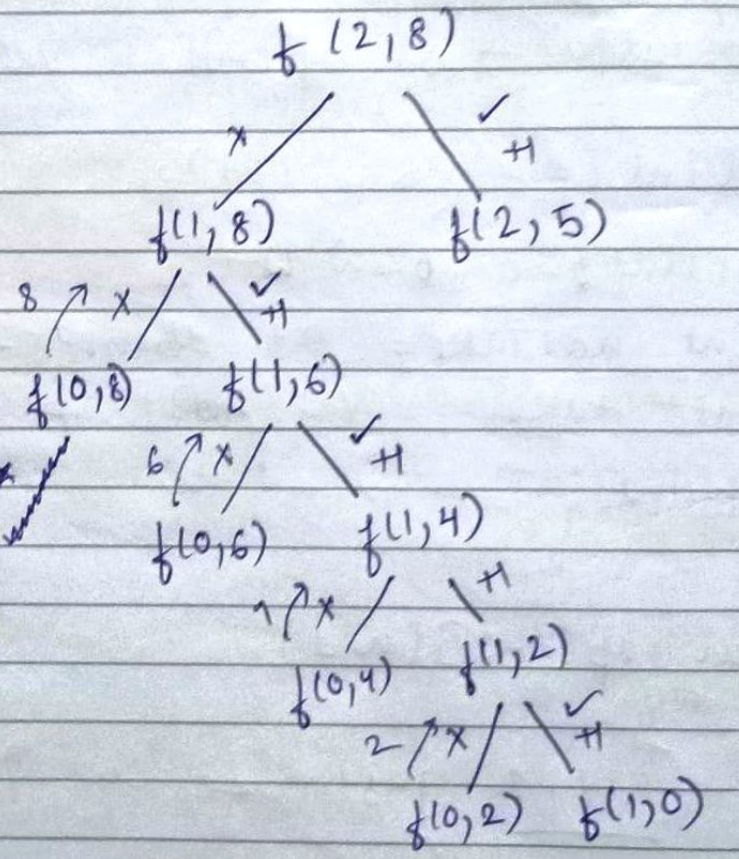
Infinite supply of coins

arr[] → {1, 2, 3} target = 7 (3, 3, 1) ③

Q Min no of coins to reach target??

```
f(ind, target) {
  if(ind==0) {
    if(T%a[i]==0) return T/a[i];
    else return INT_MAX;
  }
  int notTake = 0 + f(ind-1, T);
  int Take = int_MAX;
  if(coin[ind]<=T) take = 1 + f(ind, T-coins[ind]);
  return min(take, notTake);
}
```

arr = {1, 2, 3}
tar = 8



TC: → O(2^N) exponential
SC: O(N) → more than that O(target)

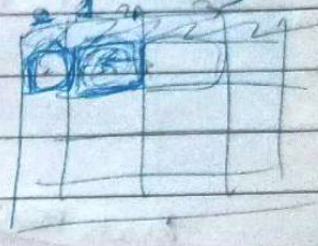
Memoization: dp[N][target+1]; TC O(N*T) SC O(N*T) + AS

Tabulation: base case changing parameters Recurrence

ind ⇒ 1 → n-1
tar → 0 → T

dp[n][tar+1];

for(T=0 → target){ if (T%a[0]==0)
dp[0][T] = T/a[0];
else dp[0][T] = INT_MAX;
}



for(int ind = 1 → n-1){
for(int T = 0 → target){
int take = 2NT_MAX;
if(nums[ind] <= T){
take = 1 + dp[ind][T-nums[ind]];
}
int notTake = 0 + dp[ind-1][T];
dp[ind][T] = min(Take, notTake);
}
}
return dp[ind-1][target];

space optimize ⇒ prev & curr.

Target Sum

arr[] = {1, 2, 3, 1}
target = 3

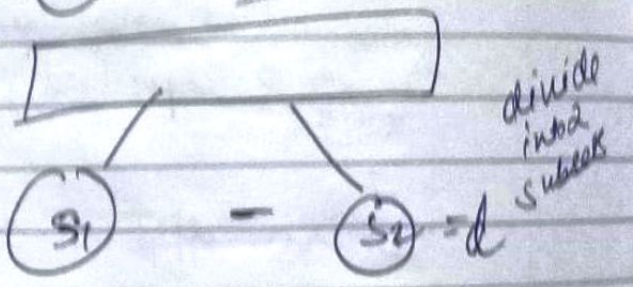
+1 +2 -3 -1 = -1
-1 +2 +3 -1 = 3 ✓ (2 ways)
+1 -2 +3 +1 = 3 ✓

assign signs either +ve or -ve to array elements.
how many ways such that you get given target.

- + + -
1 2 3 1

3 + 2 - 1 - 1 = 3

(3, 2) - (1, 1) = 3



Brute Force

f(arr, T)
Σ
plus
neg
return plus+neg
}

Code: Same as partitions with given difference because, we only need to find how many ways S_2 can be built. We do not actually need to assign signs.

Date: YOUVA

Coin Change 2

coins >= 1 (value)

1 2 3 target = 4

you are given an array you have to form target. Any no can be used any no. of times. count ways

f(2,4) → till index 2 how many ways can you form 4?

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

if(dp != -1) → dp[ind][t]

TC: exponential
SC: ~~O(T)~~ ASS >> O(N) O(target) Worst case

Memoization dp[N][T]

TC: O(NXT)
SC: O(NXT) + ASS

Tabulation 1) base case 2) changing parameters 3) Recurrence

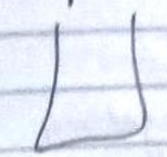
```
dp[N][T]
for(T → 0 → target){
    dp[0][T] = (T % arr[0] == 0)
}
for(i = 1 to n-1){
for(t = 0 to target){
notTake = dp[ind-1][t]
take = 0;
if(arr[ind] <= t) take = dp[ind][t - arr[i]);
dp[i][t] = take + notTake
}
}
return dp[n-1][target];
}
```

space optimization → use prev & cur.

Unbounded Knapsack

(Infinite supply)

wt → { 2, 4, 6 }
val → { 5, 11, 13 }



Maximize value

```
f(ind, W) {                    f(n-1, W) → max value you can generate
  if(ind==0){
     return (W/wt[0]) * val[0];
  }
  nottake = 0 + f(ind-1, wt);
  take = INT_MIN;
  if(wt[ind] <= W){
     take = val[ind] + f(ind, W - wt[ind])
  }
  return max(take, nottake);
```

TC: Exponential
SC → O(W) + O(N)

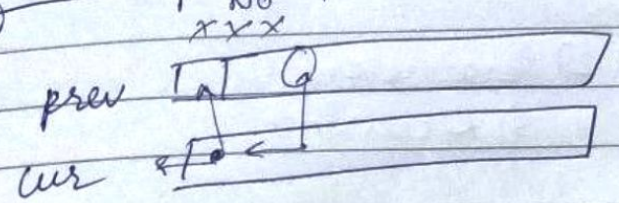
Memoization: Same as before. (prev ques)

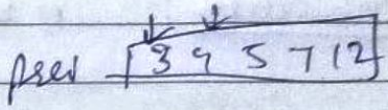
Tabulation dp[N][W+1]

```
for(W → 0 to → W){   // original weight of bag
    dp[0][W] = W / wt[0] * val[0]
                 // W that is changing in loop
}
for(int i = 1 to N-1)
  for(int t = 0 to W){
     nottake = 0 + dp[i-1][t];
     take = 0;
     if(wt[i] <= t){
        take = val[i] + dp[i][t - wt[i]];
     }
     dp[i][W] = max(take, nottake);
  }
return dp[n-1][W];
```

space optimize / prev ques

further space optimization





not take = 3
take = 0
not take = 4
take = —

not take = 0 + prev[w]

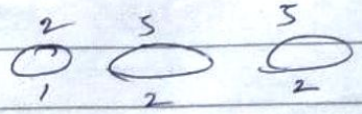
take = val[ind] + cur[w - wt[ind]]

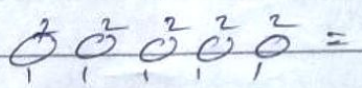
Rod Cutting Problem

rod length
N = 5

cut the rod and maximize the cost.

price[] → | 2 | 5 | 7 | 8 | 10 |
1 2 3 4 5





Infinite supply of rods.

↳ for n = 5, rod lengths can be 1 2 3 4 5
collect rod lengths to make n and maximize price

| 2 | 5 | 7 | 8 | 10 |
0 1 2 3 4

try to pick lengths & sum them up to make n.

$f(4, N)$ — till index 4 what is max price you can obtain

$f(n-1, N)$

if rod length becomes 0 since we are * N it will return 0.

```
f(ind, N) {
  if(ind == 0) {
    return N * price[0]
    // At end if 12 is required length then 12 pieces of length 1 will be required.
  }
  int notTake = 0 + f(ind-1, N)
  int take = INT_MIN;
  rod_length = ind+1;
  if(rod_length <= N) take = price[ind] + f(ind, N - rod_length);
  return max(take, notTake);
```

Memoization dp[N][N+1]

Tabulation: 1) base case 2) Parameters
3) Recurrence.

```
dp[N][N+1];
for(int i=0 to N){
    dp[0][i] = i*price[0];
}

// copy recurrence.
```

DP On Strings

M T W T F S S
Page No.:
Date:
YOUVA

Longest common Subsequence

abc → a, b, c, ab, bc, ac, abc, ∅ " "

Can be consecutive or not but maintains order
↳ Subsequence

S_1 = " a d c b c "
a d b

S_2 = " d c a d b "
a d b

brute force: generate all subsequences. Compare and get longest
$O(2^n)$ (exponential in nature)

⇒ generate all subsequence && compare On Way

a c d | c e d
0 1 2 0 1 2

// d is Matching we can say subsequence of length 1 is there.

① Express ind1, ind2
② Explore All Possibilities
③ Take best

f(2) → string till index 2

f(2,2) → lcs of str1(0..2) str2(0..2)

Do comparisons character wise

// Match on indexes
if (s1[ind1] == s2[ind2])
1 + f(ind1-1, ind2-1)

// Not Match
Max (f(ind1-1, ind2), f(ind1, ind2-1))

shrink the string and find there

f(1,1) a c | c e

+0 / +0

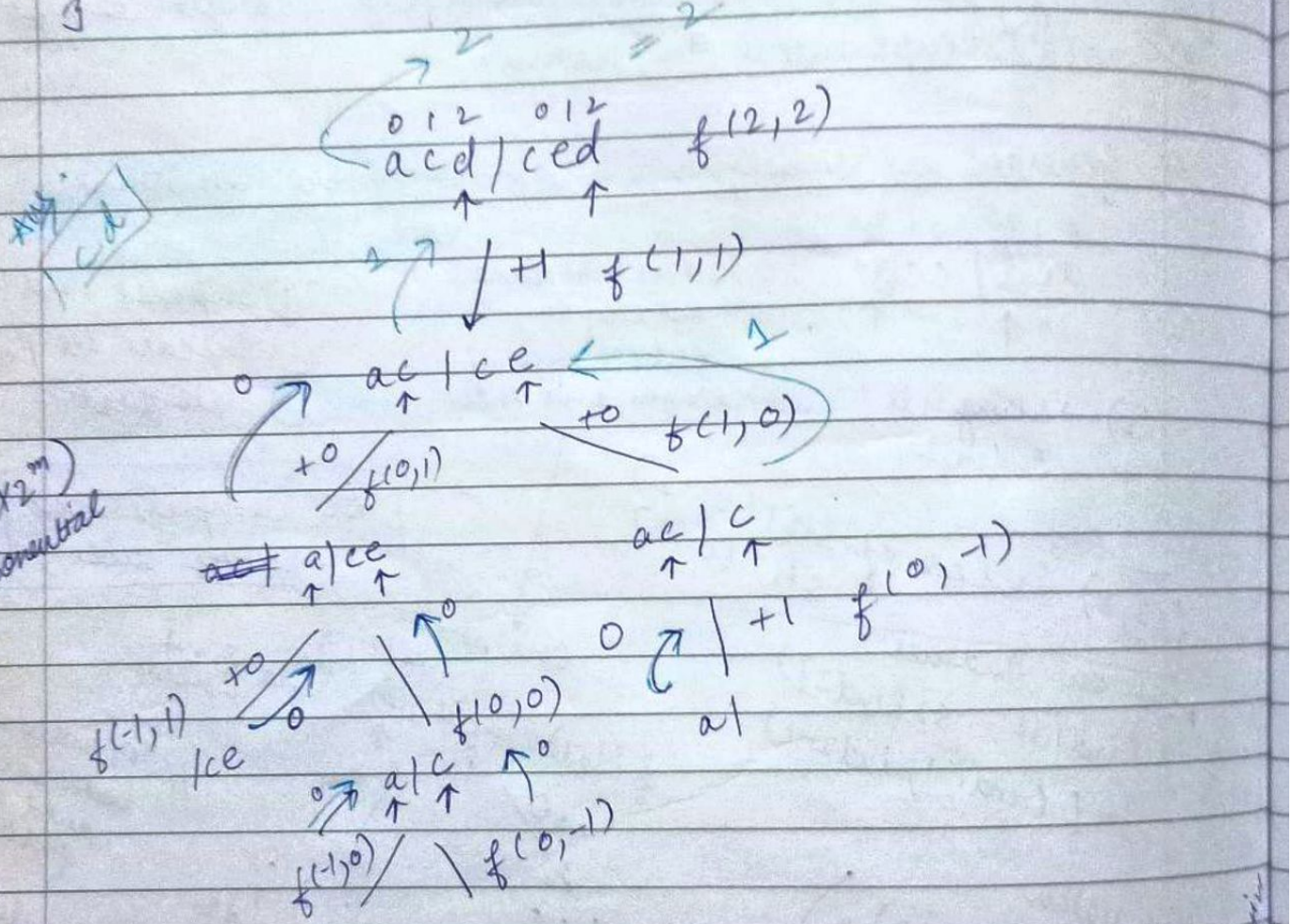
a | c e
ind1 ind2

a c | c
ind1 ind2

// Try moving both because could match further

f(ind-1, ind2-1) (negative) // neg means end of string

```
f(ind1, ind2) {
Base case   if (ind1<0 || ind2<0)
                return 0;
            if (dp[ind1][ind2] != -1) return dp[ind1][ind2];
Match       if (s1[ind1] == s2[ind2])
                return dp[ind1][ind2] = 1 + f(ind1-1, ind2-1);
No match    return dp[ind1][ind2] = 0 + max(f(ind1-1, ind2), f(ind1, ind2-1));
}
```



$O(2^n \times 2^m)$ near exponential

Overlapping Subproblems ⇒ Memoization

f(i,j) → len of str1[0..i] str2[0..j]

dp[n][m]

↑ len of str1 ↑ len of str2

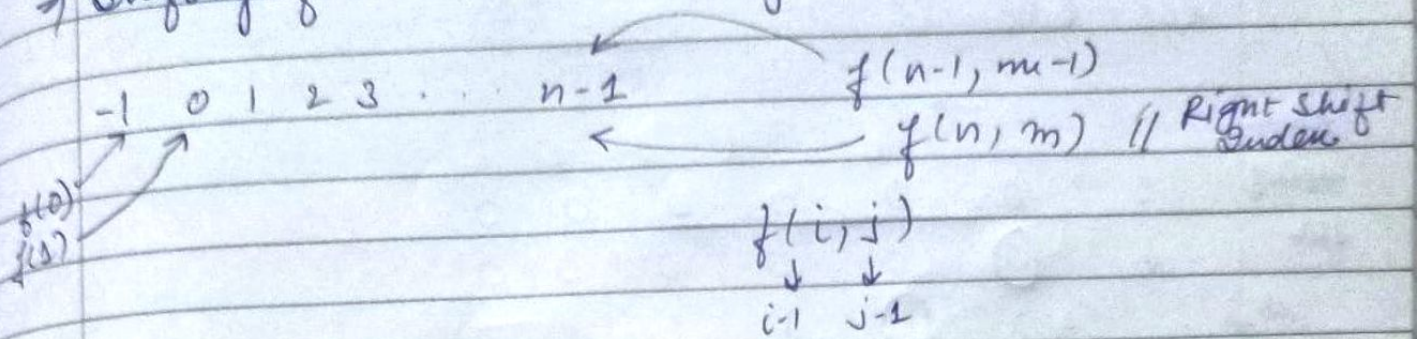
TC: O(n*m)

SC: O(n*m) + O(n+m)

Date: YOUVA

Tabulation: 1) Copy base case 2) Changing Parameters 3) Copy Recurrence

→ Shifting of index because neg is not possible



if (i == 0 || j == 0) return 0 New Base case

dp[0][j] 0 to m-1 dp[i][0] 0 to n-1

Code

```
vector<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 (s1[i-1] == s2[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];
```

f(n, m) if we check s1(n-1) s2(m-1) because we have shifted index need to check prev index of string

→ Space optimization ⇒ Prev & cur

Print lcs

s1 = "abcde" (0 1 2 3 4) s2 = "bdgek" (0 1 2 3 4)

lcs = bde

dp array for

		j→ 0	1 b	2 d	3 g	4 e	5 k
i	0	0	0	0	0	0	0
↓	1 a	0	0	0	0	0	0
	2 b	0	1	1	1	1	1
	3 c	0	1	1	1	1	1
	4 d	0	1	2	2	2	2
	5 e	0	1	2	2	3	3

(n, m)

whenever match add to ans.

Ans: bde

If string is exhausted stop.

dp[5,5]

abcde bdgek

⇒ lcs = 3

dp[4][2]

abcd bd

lcs = 2

len = dp[n][m];

ans = "";

assign dummy values

ans = $ $ $

ind

ind = len-1

Code

```
i = n, j = m;
while(i>0 && j>0){
if(s1[i-1] == s2[j-1])
{
    ans[ind] = s1[i-1];
    ind--;
    i--; j--;
}
else if( dp[i-1][j] > dp[i][j-1])
    i = i-1;
else
    j = j-2;
}
```

TC: O(N+M)

return ans;

Longest common Substring
↳ continuous

S_1 = "abcjklp" S_2 = "acjkp"

Longest common subsequence:

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

no match: $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$ XX can't omit in substring

acd | axd

ac | ax

a|ax ac|a

|ax a|a ac| a|a

abcd abzd

		a	b	z	d
	0	1	2	3	4
0	0	0	0	0	0
a 1	0	1	0	0	0
b 2	0	0	1+1	0	0
c 3	0	0	0	0	0
d 4	0	0	0	0	1

// In substring we only check if current elements are matching, and not if any prev match cannot have been made

// In Match we check if prev was also matching.

ans: Manual in Matrix

prev 0 / space optimize

In prev code,

Not match = 0 every time

Keep ans as global & update in loop

Longest Palindromic Subsequence

S = "bbbab" bbbb ans 4

s1 = "bbabcbcab"

babcbab

reverse string in ~~reverse order~~

s2 = bacbcbabb

lcs(s1, s2) will give ans.

subsequence → palindrome because s2 is lcs of s1

Minimum Insertions required to make string Palindrome

S = abcaa → min 2 insertions → asbcbaa

↓ reverse

abcaa aacba (Now this whole string is palindrome)

st rev(S) Min operation = len(s) - lcs(s, rev(s)) you can insert any char anywhere

Approach: 1) keep the palindrome portion intact

abcaa → aa, a, aba, aaa) etc

2) Longest portion intact

a b c a a c b a

a b a c a b a

Eg: coding ninjas

codingninjas
Sajn

cadisajningninjasidoc

no. of Insertions = len(str) - longest Palindromic ~~Substring~~ Subsequence

Min operations (Ins/Del) (Edits)
to make StringA to StringB

Str1 = "abcd" Str2 = "anc"

Intuition
→ what not to touch
→ try to make the smaller among Str1, Str2

abcd → anc
2del
1ins ac
anc
3op

abcd → anc
del → ac → Ins
lcs

deletions = n - len(lcs)
Insertions = m - len(lcs)

total op = n + m - 2 × len(lcs)

Shortest Common Supersequence

⇓ make a string such that s1 & s2 is in it ⇓ order has to be same

s1 = "brute" s2 = "groot"

ex: brutegroot (can be possible answer)

Ans bgruoote (len=8)

s1 = bleed s2 = blue

length=?? string=?? both

"bleued"

s1 = brute s2 = groot

(1) common guys taken once.
⇓
lcs

lcs = "rt" (bru) (goo)

bgruoote

Ans = n + m − len(lcs)

figuring the string

		g	r	o	o	t
	0	1	2	3	4	5
0	0	0	0	0	0	0
b 1	0	0	0	0	0	0
r 2	0	0	1 ←	1	1	1
u 3	0	0	1	1 ←	1	1
t 4	0	0	1	1	1	2
e 5	0	0	1	1	1	2

exhausted (0,1)

br|gro

groo bru | gro bru

groot brut

ans bgruoote

etouorbg (we will get this) ← reverse to get ans

```
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][j] >= dp[i][j-1]) {   // up
          ans += s1[i-1];
            i--;
   }
   else {
          ans += s2[j-1];
            j--;
   }
}
while ( i>0 ) ans += s1[i-1]; i--;
while (j>0) ans += s2[j-1]; j--;

reverse (ans);          // Supersequence.
```

Distinct Subsequences

Given a string and a string, find how many times s2 appears in s1.

s = "babgbag" s2 = "bag"

ans: 5
babgbag
babgbag
babgbag
babgbag
babgbag

Different methods of comparing → Trying All Ways → Recursion

1) Express in terms of (i, j) s2 s2
2) Explore all possibilities
3) Return sum of all possibilities

count ways

```
f() {
  Base case ? 0 ? 1
  return f() + f()
}
```

f(n-1, m-1) → means no. of distinct subsequences of s2 from [0...j] in s1 [0...i]

(n) babgbag (m) bag
i (n-1), j (m-1)

```
f(i, j) {
  if(j<0) return 1;
  if(i<0) return 0;
  if(dp[i][j] != -1) return dp[i][j];
  if(s1[i] == s2[j]) {
    return dp[i][j] = f(i-1, j-1) + f(i-1, j)
  }
  Else return dp[i][j] = f(i-1, j)
}
```

TC: exponential
SC: O(N+M)

TC: O(N×M)
SC: O(N×M) + O(N+M)

memoization dp[n][m];

1 based Indexing (same as lcs)

Tabulation:

```
int dp[n+1][m+1];

Base case  for (i = 0 → n) dp[i][0] = 1   // because j=0 str2 is exhausted
j from 1   for (j = 1 → m) dp[0][j] = 0
to be
filled

for(int i = 1 ; i<=n; i++){
  for(int j=1 ; j<=m; j++){
    if(s[i-1] == s2[j-1]){
      dp[i][j] = dp[i-1][j-1] + dp[i-1][j];
    }
    Else dp[i][j] = dp[i-1][j];
  }
}
return dp[n][m];
```

Space optimization:

prev & cur

base case prev[0] = cur[0] = 1

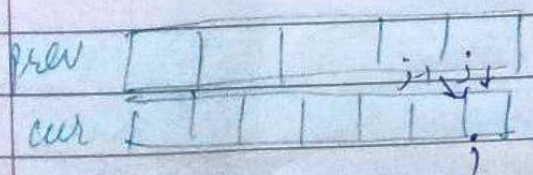
More optimized

i = 1 → n

j = 1 → m can do m → 1

~~dp[i][j]~~ cur[j] = prev[j-1] + prev[j]

else cur[j] = prev[j];



Instead of cur, do changes in prev[j]

Edit Distance

3 operations
1) Insert
2) Remove
3) Replace
Min steps to convert S1 to S2.

S1 = "horse" S2 = "ros"

replace h → r rorse
remove r rose (3 operations)
remove e ros

S1 = "intention" S2 = "execution"

remove t inention
replace e enention
replace n → x exention
replace n → c exection
insert u execution

String Matching path

horse
ros

f(n-1, m-1) → find min operations to convert S1[0..i] to S2[0..j]

If not matching
→ Insert same char
→ Delete & try finding someone else
→ Replace & Match

try all ways
↓
Recursion
take minimum

```
f(i, j) {
(Next Page) if(i<0) return j+1;
            if(j<0) return i+1;
Base case
if(s1[i] == s2[j]) {
    return 0 + f(i-1, j-1);
}
else {
    min { 1 + f(i, j-1)   // Insert
        { 1 + f(i-1, j)   // Delete / Remove
        { 1 + f(i-1, j-1) // Replace
```

TC: exponential
SC: O(N+M)

something inserted in front of i and replaces a hypothetical match, which would make

Base case

S1 gets exhausted

i
horse f(-1, 1)
ros (2 operations to make ro)
min to convert empty string to S2[0...1]
& insert

return j+1

S2 gets exhausted

S1 = horse S2 = ros
0 1 2

min operations to convert S1 to empty string.

Memoization

TC: O(N x M)
SC: O(N x M) + x2 dp[n][m]; // changing parameters.

Tabulation: 1 Based Indexing ⇒ to accumulate base case

```
for(j=0 to m) dp[0][j]=j;
for(i=1 to n) dp[i][0]=i;

for(int i=1; i<=n; i++){
  for(int j=1; j<=n; j++){
    if(s1[i-1]==s2[j-1]) dp[i][j]=dp[i-1][j-1]
    else{
    dp[i][j]=1+min(dp[i][j-1], dp[i-1][j], dp[i-1][j-1])
    }
  }
}
return dp[n][m];
}
```

(Leetcode) Wildcard Matching

S1 = "?ay"
S2 = "ray"
S1 Matches S2 true

? ↓ matches with single char

* ↓ matches with sequence of length 0 or more.

S1 = "abc*cd"
S2 = "abdefcd"
S1 Matches S2 true

i: ab*cd
j: abdefcd

① If * matches choose * = ""
* = f
* = ef
* = def

```
f(i, j) {
  if(i<0 && j<0) return true;
  if(i<0 && j>=0) return false;
  if(j<0 && i>=0) { for - - - }
  if(s1[i]==s2[j] || s1[i]=='?')
      return f(i-1, j-1);
  else if(s1[i]=="*")
      return f(i-1, j) or f(i, j-1)
  Else
      return false;
}
```

Base case

TC exponential $O(2^{N+M})$

ab* / abdef*
ab / abdef ab* / abdef
ab / abdef

S1 is exhausted
i<0 && j<0 true
if(i<0 && j>=0) return false

S2 exhausted
j<0 && i>=0
// if S1 is left it has to be all *
for(i=0 → i) → left
S[i] == *

Memoization: dp[n][m]; TC O(N×M) SC O(N×M) + O(N+M)

Tabulation: (1 based Indexing)

```
vector<vector<bool>> dp(n+1, vector<bool>(m+1, false));
dp[0][0] = true;
for(int j=1; j<=m; j++) dp[0][j] = false;
for(i=1; j<=n; i++){
  for(ii=1; ii<=i; i++){ flag=true;
    if(s1[ii-1] != '*')
      dp[i][0]=false;
      break; flag=false;
  }
} dp[i][0] = flag;

// copy the Recurrence

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

DP On Stocks

Best Time to Buy & Sell Stock

arr[] → { 7, 1, 5, 3, 6, 4 } n = 6

Decide a day to buy stock & day to sell.

If you are selling on ith day, you buy on min price from 0 to (i-1)

```
mini = a[0], profit = 0
for(int i = 1; i<n; i++) {
    cost = a[i] - mini;
    profit = max(profit, cost);
    mini = min(mini, a[i]);
}
return profit.
```

Best Time To Buy & Sell Stocks II

price[] = { 7, 1, 5, 3, 6, 4 }

B S B S

1 4 3 = 7

X X Not allowed

7 1 5 3 6 4

B S

5

0 1 2 3 4 5

| 7 | 1 | 5 | 3 | 6 | 4 |

↑

f(ind, buy)

if(ind==n) return 0;

starting on 0th day with buy we make max profit you can make

buy = 1 you can buy

buy = 0 you cannot buy

f(0, 1)

Buy as many times as you want sell as many times as you want to give max profit

Recursion

Try all ways

Get best answer

- At every index, we have to know if we can buy that stock.
- If can't buy 2 options sell / Not sell
- Whenever you are buying, at 1 you are doing -1 to the index and if selling at 0 it is +1

```
if(buy){
    profit = max{ -prices[ind] + f(ind+1, 0) // take
                  0 + f(ind+1, 1) // Not take }
}
else{
    profit = max{ prices[ind] + f(ind+1, 1);
                  0 + f(ind+1, 0); }
}
```

Date: YOUVA

Overlapping Subproblems → Memoization

TC O(N*2) SC O(N*2) dp[N][2] // because buy can be either 0 or 1.

ASS + O(N*2)

Tabulation

1. base case 2. Changing Parameters (n-1 → 0) (0 → 1) 3 Recurrence

```
dp[N+1][2];
dp[N][0] = dp[N][1] = 0;
for(int ind = n-1; ind >= 0; ind++) {
    for(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]; (because at begining we can only buy)
```

	0	1
0	15	8
1	9	8
2	9	4
3	7	4
4	6	4
5	4	0
6	0	0

Space optimization

only using next line to get cur.

Do like prev & next cur.

But prev will be next this time.

Best Time To Buy & Sell Stocks 3

prices = {3, 3, 5, 0, 0, 3, 1, 4}

At max 2 transaction

0 → 1 → trivial = 2

f(ind, buy, cap) {
  if(ind == n) return 0; // exhausted all days can't get anything
  if(cap == 0) return 0; // Exhausted transactions

  if(buy) {
    return profit = max{ -prices[ind] + f(ind+1, 0, cap)
                         0 + f(ind+1, 1, cap) }
  }
  // sell
  else {
    return max{ prices[ind] + f(ind+1, 1, cap-1);
                0 + f(ind+1, 0, cap); }
  }

TC: exponential SC: O(N)

Memoization: dp[N][2][3]; (either 0, 1 or 2) TC: N×2×3
SC: N×2×3 + ASS

Tabulation: 1) Base case 2) changing Parameters (n→1→0, 0→2, 0→2) 3) copy Recurrence

{ for(ind = 0→n-1)
    for(buy → 0→1)
      dp[ind][buy][0] = 0 }

dp[n+1][2][3];

{ for(buy = 0→1)
    for(cap = 0→2)
      dp[n][buy][cap] = 0 }

→ because as per base case for every cap 0, ans will be 0.

```
for (ind = n-1; ind >= 0; ind--){
  for (buy = 0; buy <= 1; buy++){
    for (cap = 1; cap <= 2; cap++){
      if (buy == 1){
        dp[ind][buy][cap] = max(-prices[ind] +
                                dp[ind+1][0][cap]
                              , 0 + dp[ind+1][1][cap])
      }
      Else {
      dp[ind][buy][cap] = max(prices[ind] + dp[ind+1]
                                [1][cap-1]
                              , 0 + dp[ind+1][0][cap]);
      }
    }
  }
}
return dp[0][1][2];
```

Space Optimization: ind+1 can be turned into next. And turn this into 2d dp.

Buy & Sell stocks IV

At most k transactions.

Same as stocks III problem just there k was 2.

Buy & Sell with cooldown

Can't buy right after sell.

prices[] → {4, 9, 0, 4, 10}

```
f(ind, buy){
  if(ind >= n) return 0;
    // >= n because from ind = n-1, +2 will be out of bounds.
  if(buy){
  return max(-prices[ind] + fn(ind+1, 0) , 0 + fn(ind+1,1));
  }
  Else {
  return max(prices[ind] + f(ind+2, 1) , 0 + f(ind+1,0));
  }
```

Tabulation:

```
vector<vector<int>> dp(n+2, vector<int>(2,0));
// don't need to write base case because we are initializing everything as 0 anyways.

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+2][1],
                     dp[ind+1][0]);
   }
  }
 return dp[0][1];
}
```

Buy & Sell with Transaction fee

arr = {1, 3, 2, 8, 4, 9} , fee = 2
7-2   5-2 = 5+3 = 8

Unlimited B & S. Everytime a B+S a fee will be charged. Maximize net profit.

```
f(ind, buy){

If (ind == n) return 0;

If (buy == 1){
 return max( same condⁿ )
```

→ can do - fee also when you are buying. same thing. Just do it either in buy or sell.

```
Else {
return max( prices[ind] - fee + fn(ind+1,1)
          , fn(ind+1, 0));
```

Tabulation: Same just add - fee in sell.

Find the length of longest common subsequence

arr[] → {10, 9, 2, 5, 3, 7, 101, 18}
(indices 0–7 written above; ✓ over 10, x over 9; "ind" under 10 and 9)

start here f(0, -1)

If you take 10 [10, 9 cannot be a part. So you have to keep store of prev.

```
f(ind, prev_ind){
   if(ind == n) return 0;

// Not take
nottake = 0 + f(ind+1, prev_ind)
   if(dp[ind][prev_ind+1] != -1) return ...
// Take
   if(prev_ind == -1 || arr[ind] > arr[prev_ind])
   take = 1 + f(ind+1, ind)

return len = max(take, not-take);
```

TC: 2^n
SC: ASS O(N)

not checking with -1 here, because prev_ind+1 me sirf store prev_ind+1 ho raha hai, call to prev_ind se hi ho rahi hai

dp[ind][prev_ind+1] ← for coordinate change

Memoize   dp[n][n+1];   // prev_ind goes from -1 to n-1.

TC: O(N×N)   SC = O(N×N) ASS

co-ordinate change

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

Tabulation: dp[n+1][n+1] = {0};   // don't need base case everything is 0 anyway.

```
for(ind = n-1 → 0)
   for(prev_ind = ind-1 → -1){
// Not take
   int len = 0 + dp[ind+1][prev_ind+1];
   if(prev_ind == -1 || arr[ind] > arr[prev_ind])
// take
   len = max(len, 1 + dp[ind+1, ind+1]
```

dp [ind] [prev_ind + 1] = len;

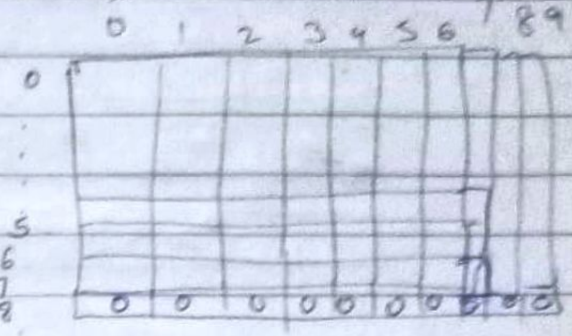
}

}

[-1+1];

return dp[0] ~~[...]~~ ;

}



space Optimization : next & cur
(ind+1)

TC : $O(N^2)$ SC : $O(N)$ + 2

Explanation another Method

ans = from (0 → n-1) max (dp[i]).

5	4	11	1	16	8
1	1	2	1	3	2

// longest possible

dp[i] → signifies the longest inc. subsq. that ends at index i.

0	1	2	3	4	5
5	4	11	1	16	8
1	1	~~1~~ 2	1	~~1~~ 3	~~1~~ 2

even if there is or isn't prev element, min len will atleast be 1. initially dp is 1.

there is only 5 as prev which i can take

because i know there is a subsequence that ends at 5 which i can take. So at 11 = 1+1.

check with 4 also. But since max remains 2 no updations

for (ind = 0 → n-1)
  for (prev → 0 → ind-1) {
    if (arr [ind] < arr [ind])
      dp [ind] = max (1+ dp [prev], dp [ind])
  }
}

// checking if we can take it and whether improves the answer

Max of dp will be ans.

Print the LIS

0	1	2	3	4	5
5	4	11	1	16	18
4	1	2	1	3	4

dp

0	1	0	3	2	4

hash (tells the prev guy)

Start at index of max value in dp

(5) → (4) → (2) → (0)

16 → 16 → 11 → 5

rev → 5 11 16 18

dp[n] = {1} hash[n] = {Initialize with index nos.}

```
int maxi = 1, last_ind = 0;

for(int i = 0; i<n; i++){
    hash[i] = i;
    for(int prev = 0; prev<i; prev++){
        if(arr[prev] < arr[i] &&
                1 + dp[prev] > dp[i]){
            dp[i] = 1 + dp[prev]
            hash[i] = prev;
        }
    }

    if(dp[i] > maxi){
        maxi = dp[i];
        last_ind = i;
    }
}
```

→ int lis[maxi];
lis[0] = arr[last ind];
int ind = 1;

```
while( hash[last_ind] != last_ind){
    last_ind = hash[last_ind];
    lis[ind] = arr[last_ind];
}
```

reverse the lis and print

Longest Increasing Subsequence Using Binary Search.

{1, 7, 8, 4, 5, 6, -1, 9}

{1, 7, 8, 9} len = 4
{1, 4, 5, 6, 9} len = 5 ✓ can't generate all because TC ↑↑
{-1, 9} len = 2

{1, 7, 8, 4, 5, 6, -1, 9}
{1, 7, 8} // where 4 can fit in missing subsequence

we can replace 7 with 4 here even if 4 is not the subsequence, we still do know now it has something less missing in between.

{1, 4, 8}
{1, 4, 5, 6}
{-1, 4, 5, 6, 9} // not the subsequence but gives the length

optimizes space

Insert At → first element after our element is just greater to 4

arr() → {1, 4, 5, 4, 2, 8}

{1, 4, 5, 8} len = 4
4 rewrite

In C++ lower_bound() → gives first ind of arr[i] just or first index of element arr[i]

```
vector<int> temp;
temp.push_back(arr[0]);
for(int i=1; i<n; i++){
    if(arr[i] > temp.back()){
        temp.push_back(arr[i]);
    }
    else{
        int ind = lower_bound(temp.begin(), temp.end(), arr[i]) - temp.begin();
        temp[ind] = arr[ind];
    }
}
return temp.size();
```

Binary search part

Largest Divisible subset

arr[] = {1, 16, 7, 8, 4}

{16, 8, 4} (16, 8) (8, 4) (16, 4)

Largest → {1, 16, 8, 4}

→ because subset can be any order

Print any answer ⇒ Thought process

Elements should be dividing in the subset. Like if you pick 2 elements then either arr[i] % arr[j] == 0 or arr[j] % arr[i] == 0

1) sort the array {1, 4, 8, 8, 16}

{1, 4, 8, 16}

We can pick 8 because if 8/4 and 4/1 then 8/1 is also true.

// we can call prev of longest divisible subsequence

a/b, b/c ⇒ a/c given a<b<c

// In previous ques code,

change if condition

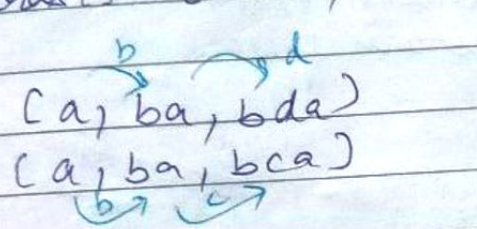
if (arr[i] > arr[j]) ✗

if (arr[i] % arr[j] == 0) ✓✓

you can also use hash array to backtrack & print.

Longest String Chain

words[] = {"a", "b", "ba", "bca", "bda", "bdca"}



1st can be anything. Insert another for 2nd. Another for 3rd. Insertion can be anywhere.

Longest → {a, ba, bda, bdca} Print length.

// Like LIS, if we check the difference b/w 2 subsequent elements to be 1 char, it is kind of like LIS

// Sort acc. to length in beginning because this is also subset.

for (i = 1 → n)

  for (j = 0; j < i; j++)

    compare (arr[i], arr[j]) → you can form arr[i] by adding only one char to arr[j]

    if (arr[i] > arr[j] && dp[j] + 1 > dp[i])

      dp[i] = 1 + dp[j]

  }

  Do max(max, dp[i])

}

return max;

LIS

arr[j] → bda

arr[i] → bdca

bdca

bda

only one difference. Keep pointers. Do they reach together simultaneously, return true.

greater guy

```
bool compare(string &s1, string &s2){
    if(s1.length() != s2.length())
        return false.

    int first=0, second=0;

    while(first < s1.size()){
        if(s1[first] == s2[second]){
            first++;
            second++;
        }
        else{
            first++;
        }
    }

    if(first == s1.size() && second == s2.size())
```

Longest Bitonic Subsequence

arr[] = { 1, 11, 2, 10, 4, 5, 2, 1 } → Increasing then Decreasing

ans { 1, 2, 10, 4, 2, 1 } len = 6
{ 1, 11, 10, 5, 2, 1 }

** Bitonic can also be just Inc or just Dec

arr[] = { 10, 10, 2, 3, 4, 5, 6 }

ans → { 1, 2, 3, 4, 5, 6 } len = 6

```
for(i=0; i<n; i++){
    dp[i] = 1;
    for(j=0; j<i; j++)
    {
        if( arr[i]>arr[j] && dp[i]<dp[j]+1)
            dp[i] = dp[j]+1;
    }
}
```

	1	11	2	10	4	5	2	1
dp[] →	1	2	2	3	3	4	2	1
revdp[]	1	5	2	4	3	3	2	1
bitonic[] ←	1	6	3	6	5	6	3	1

dp[i] + revdp[i] - 1 = len
At any ind i.

Code LIS from front
LIS from back
Make bitonic → take max

No. of Longest Increasing Subsequence

arr[] → {1, 3, 5, 4, 7}

{1, 3, 4, 7}
{1, 3, 5, 7}
} how many subsequences are there of max length
ans → 2.

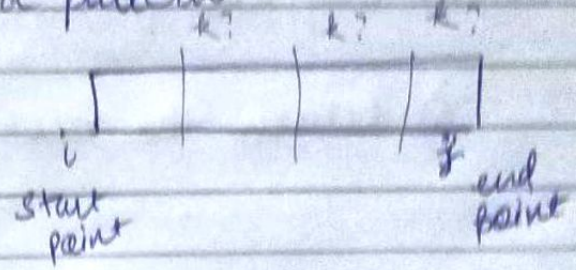
arr[] = {1, 5, 4, 3, 2, 6, 7, 10, 8, 9}
dp[] = 1 x2 x2 x2 2 ~~2~~3
count[] = 1 1 1 1 1 ~~2~~ ~~3~~ 4 ~~5~~

1 & 6 was there now
5, 6 is there
4, 6
3, 6
2, 6
5 LIS of len 2 at 6.

No need to check for len 3.
1, 5, 6
1, 4, 6
1, 3, 6
1, 2, 6

DP On Partition

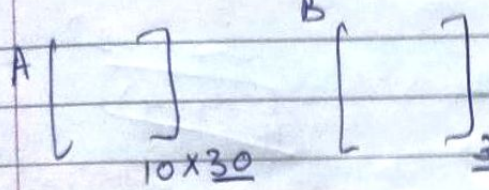
solve a problem in a pattern



Matrix Chain Multiplication

ABC

A = 10×30
B = 30×5
C = 5×60



operations = 10×30×5

10×30 30×5
(AB) C = 10×30×5 operations
[]10×5 []5×60 = 10×5×60
= 1500 + 3000 = 4500 operations.

30×5 5×60
A (BC) = 30×5×60
10×30 30×60 = 10×30×60
= 9000 + 18000 = 27000 op.

Min operations = (4500) → Print this

Q Given the N matrix dimensions, find the min cost to multiply them.

given arr[] → [10, 20, 30, 40, 50]
0 1 2 3 4
→ dimensions of n-1 matrices

n = 5

[n+th → A[i-1]A[i]]

ABCD

A[0][2]
A → 10×20
A[1] A[2]
B → 20×30
A[2] A[3]
C → 30×40
A[3] A[4]
D → 40×50

Rules : 1) Start with entire block / array (i, j)
2) Try all partitions → Run a loop to try all partitions
3) Return the best possible 2 partitions

(ABCD) with i, j

ABC(CD)   A(BCD)   (ABC)D

i → start point
j → end point

[10, 20, 30, 40, 50]
A B C D
i → 1, j → (n-1)

f(1, 4) → return min multiplications for Matrix 1 to Matrix 4.

```
f(i, j) {
  if (i == j) return 0;        // Base case
  if (dp[i][j] != -1) return dp[i][j];
  min = 1e9;
  for (k → i to j-1) {
    steps = ar[i-1] x ar[k] x ar[j]
            + f(i, k) + f(k+1, j)
    mini = min(mini, steps);
  }
  return dp[i][j] = min(all steps);
```

TC → exponential

[10, 20, 30, 40, 50] (indices 0 1 2 3 4)
A B C D

k = (i → j-1)

f(i, k), f(k+1, j)

k=1 f(1,1) f(2,4)
k=2 f(1,2) f(3,4)
k=3 f(1,3) f(4,4)

OR

k = (i+1 → j)

f(i, k-1), f(k, j)

k=2 f(1,1) f(2,4)
k=3 f(1,2) f(3,4)
k=4 f(1,3) f(4,4)

Memoization ⟹ Overlapping Subproblems

dp[N][N]

TC: O(N×N) × N (due to loop) = O(N³)

SC: O(N²) + O(N)

10 20 30 40 50

i k k j

A BCD

f(1,1) f(2,4)

(10x20) (20x50)

steps = 10x20x50

f(1,2) f(3,4)

10x30 30x50

A = 10x20

B = 20x30

C = 30x40

D = 40x50

B C D

(20x30) (30x40) (40x50)

20x40 30x50 40x50

20x50 20x50

A B C D

10x20 20x30 30x40 40x50

10x30 30x50

Decision Tree

~~A B C D~~

i k k j

f(10 20 30 40 50)

A B C D

A [20] i j [30 40 50] i j

B C D

AB 20 30 i j

CD 40 50 i j

ABC 20 30 40 i j

D 50 i j

10x20x50 + f(A) + f(BCD)

BCD

30 40 50

i k k j

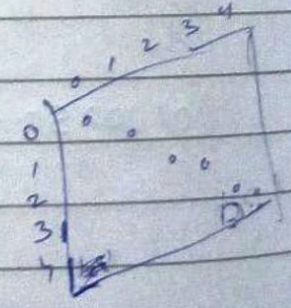
BCD

B CD

C D

Tabulation : 1) copy base case
2) changing parameters
i = n-1 → 1 j = i+1 → n-1
3) copy recurrence

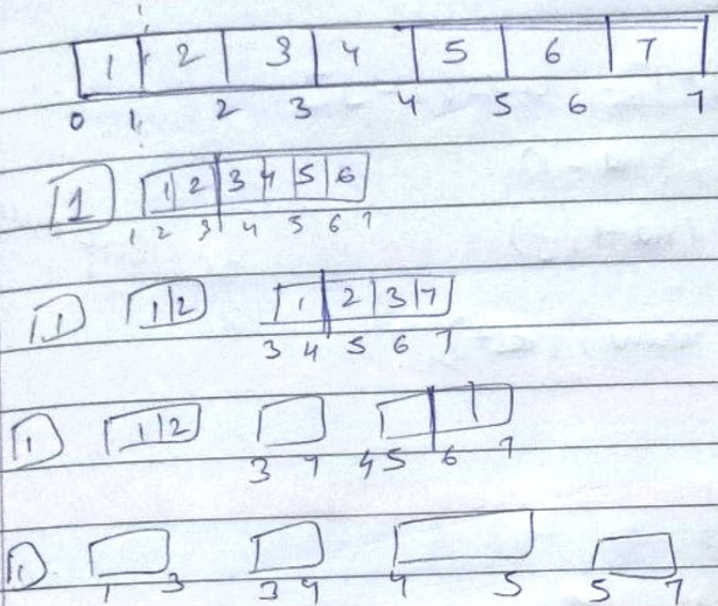
```
dp[N][N]                                  // f(1,1) represents min cost to multiply mat 1 to last
for(i = 1; i<N; i++){
    dp[i][i] = 0;
}

for(i = N-1 → 1)
  for(j = i+1 → N){
      int mini = 1e9;
    for(k = i; k<j; k++){
      int steps = arr[i-1] * arr[k] * arr[j]
                + dp[i][k]
                + dp[k+1][j];
      if(steps < mini) mini = steps;
    }
    dp[i][j] = mini;
  }
}
return dp[1][n-1];
}
```



Min cost to Cut the Stick

arr [] = [1, 3, 4, 5]   n = 7

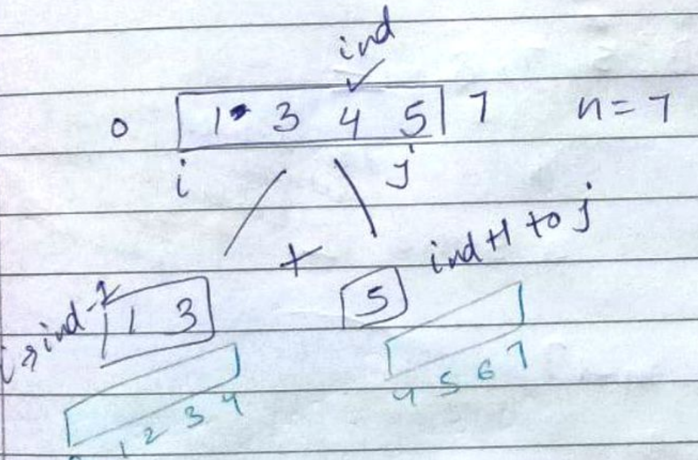
cost = len of stick you are cutting



7+6
+4+3
= 20

you can follow cuts in any order

ar = [3, 5, 1, 4]
cost = 3+4+3+2
= 16



Solve them independently
↓
because it is sorted

7+

→ If it was 1 3 4 5 2
1 3   5 2
↓
now 2 is dependent on right part

cost = length of that stick in which 4 is.

cuts[j+1] - cuts[i-1]
7-0=7

```
f(i, j) {
  if (i > j) return 0;
    mini = INT_MAX
  for (i = i → j) {
    cost = cuts[j+1] - cuts[i-1]
         + f(i, ind-1)
         + f(ind+1, j)

    mini = min(mini, (cost));
  }
  return mini;
}
```

TC: exponential

ex: 0, 1, 2, 3, 5, 7, 8, 12

cut

i = 1, j = 8

0 1 2 3 — i j

3 5 7 8 12 — i j

don't need to attach anything because arbar to pura hi pass hoga only (i, j) change

Memoization: dp[N+1][n+1]

↑ size of original cuts array

TC: O(N³) or O(n²) × N

SC: O(n²) + N

Tabulation:

// base case entire dp is initialized 0.

dp[c+2][c+2];

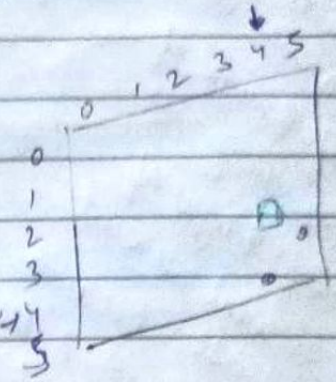
i → n → 1

j → 1 → n

c = size of cuts array.

```
for(int i = c; i >= 1; i--) {
  for(int j = 1; j <= c; j++) {
    if (i > j) continue;
    int mini = INT_MAX;
    for (int ind = i; ind <= j; ind++) {
```

```
cost = cuts[j+1] - cuts[i-1] + dp[i][ind] +
                                dp[ind+1][j];
mini = min(mini, cost);
}
dp[i][j] = mini;
}
}
return dp[1][c];
}
}
```



Burst Ballons

ar = [3, 1, 5, 8] n=4

every balloon has a no. written on it.

If you burst 5, coins = 1×5×8.
you can go in any order. Get Max coins.

[3, 1, 5, 8]	3×1×5	= 15
[3, 5, 8]	3×5×8	= 120
[3, 8]	1×3×8	= 24
[8]	1×8×1	= 8
		167 (Max)

[b_1 b_2 b_3 b_4 b_5 b_6]

$b_3 \times b_4 \times b_5$ + ($b_1 b_2 b_3$) + ($b_5 b_6$)

b_1 b_2 b_3 ~~b_5~~ b_6

b_1 b_2 b_3 ~~b_6~~

NO XX
because if burst b_3 or b_5 they will depend on either left or right element in other block

NOT INDIVIDUAL SUBPROBLEMS

given

1 [3 1 5 8] 1

3×1×5 5×1×3

[3 5 8] → 3×5×8

[3 8] → 1×3×8

(8) → 1×8×1

decide last guy

ind

b_1 b_2 b_3 b_4 b_5 b_6

i j

$a[i-1] \times a[ind] \times a[j+1]$ + f(i, ind-1) + f(ind+1, j)

b_1 b_2 b_3 b_4 b_5 b_6

Probable [b_1 b_4] or [b_2 b_4] or [b_3 b_4] or [b_4 b_5] or [b_4 b_6]

Independent subproblem

[b_4] 1×b_4×1

0 1 2 3 4 5 6

1 b_1 b_2 b_3 b_4 b_5 1

i j

f(1, 5)

f(i, j) {

if (i > j) return 0;

for (ind = i → j) anyone can be last guy

cost = a[i-1] × a[ind] × a[j+1] + f(i, ind-1) + f(ind+1, j)

ans = max(cost, ans)

Memoization : dp[n+1][n+1]

Tabulation :

```
int n = a.size();
a.push_back(1);
a.insert(a.begin(), 1);
dp[n+1][n+1] = {0};
for(int i = n; i>=1; i--){
    for(int j = 1; j<=n; j++){
        if(i>j) continue;
        int mani = 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];
            mani = max(cost, mani);
        }
        dp[i][j] = mani;
    }
return dp[1][n];
}
```

Boolean ~~Expression~~ Evaluation To True

expression = "T|T&F"

(T&F)
(F)

or → |
and → &
XOR → ^

Solve expression in such a way that end of it you get true.

(T)|(T&F)
T | F = T

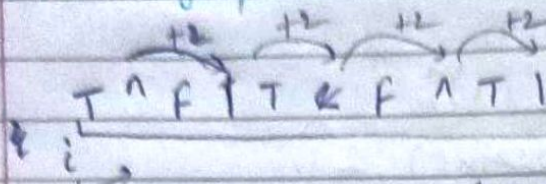
Return ways

Ex: F|T^F → F|T^F (F|T)^F

2 ways.

T ^ F | T & F ^ T | F

at every operand, we can do a partition



i+1 → +2 → +2 to find operands, till j-1

ind

T ^ F | T & F ^ T | F

f(i, ind-1) f(ind+1, j)

left & right

ways = x y

total ways = x * y

no ways on left is T

ways in which right is T.

left | right

T = x1 T = x2 x1 * x2

F = x4 F = x3 x1 * x3

x4 * x2

left ^ right

T = x1 T = x2 T ^ T = F

F = x3 F = x4 F ^ F = F

T ^ F = T

F ^ T = T

total = x1 * x4 + x3 * x2

isTrue = if looking for true = 1

false = 0

f(0, n-1, 1) → no of ways to make true the entire exp

initial call f(0, 10, 1)

```
f(i, j, isTrue)
{
    if (i > j) return 0;
    if (i == j) {
        if (isTrue == 1) return a[i] == 'T' // returns 1 if true, 0 if false
        else return a[i] == 'F' // ways to make false
    }
    int ways = 0
    for (ind = i+1; ind <= j-1; ind = ind+2)
    {
        lT = f(i, ind-1, 1)
        lF = f(i, ind-1, 0)
        rT = f(ind+1, j, 1)
        rF = f(ind+1, j, 0)
```

// its single digit — base case

T | F ^ T

1 way to make it true → (T) (F ^ T) — i j

1 way → T | F T — i j i j

```
if (a[ind] == '&')
if (istrue) ways += LT * RT;  else ways += (LT*RF) + (LF*RT) + (RF*LF)
Else if (a[ind] == '|') {
if (istrue) ways += LT * RT + LT * RF + RT * LF   // write false cond" also
else if (a[ind] == '^')
if (istrue) ways += LT * RF + RT * LF;   // write false case also
}
return ways;
}
```

Memoization: dp[N][N][2];

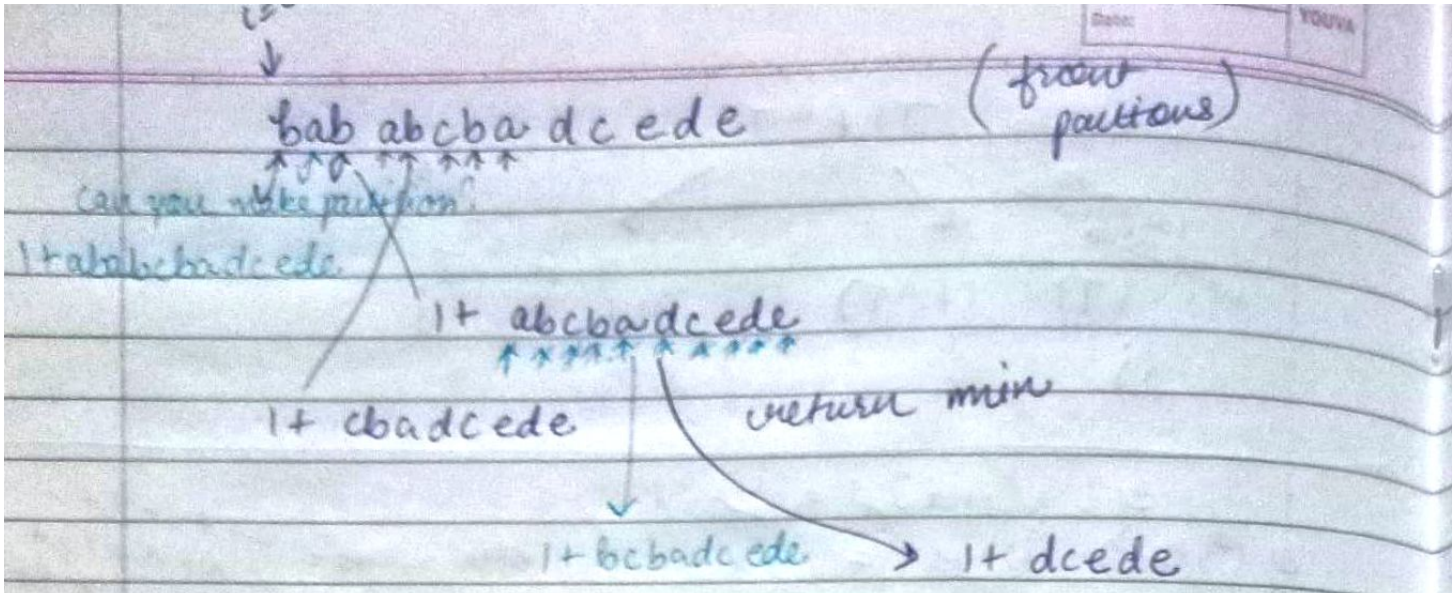
Tabulation: Code on Coding Ninjas..

Palindrome Partitioning II

Str = babbabcbadcce
ans = 4 partitions

Return Minimum partitions to make string as palindrome.

Str = aabb n=4
a|a|b|b (n-1) partitions
↳ every individual substr will be a palindrome
aa|bb 1 partition



Recurrence: Rules: 1) Express as index
2) All ways
3) Take minimal of all possibilities
4) Base case

```
f(i)
{
  if(i == n) return 0;
  temp = ""; mini = INT_MAX;
  for(j = i; j < n; j++) {
      temp += s[j];
      if( isPalindrome(temp)) {
          cost = 1 + f(j+1);
          mini = min( mini, cost);
      }
  }
  return mini;
}
```

At end return f(0) - 1 because in case of ABC our code makes partition at end of case.

Memoization: dp[n]
TC: O(N) x N
SC: O(N) + ASS O(N)

Tabulation :

```
int n = str.size();
vector<int> dp(n+1, 0);
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(cost, minCost);
        }
    }
    dp[i] = minCost;
}
return dp[0] - 1;
}
```

Partition Array for Max Sum

ind: 0 1 2 3 4 5 6

arr[] = [1, 15, 7, 9, 2, 5, 10] K = 3 → length at max

can only have 3 elements

[1 | 15 | 7 9 2 | 5 10]

[15, 15 | 9 9 9 | 10 10] = 77

[1 15 7 | 9 | 2 5 10]

[15 15 15 | 9 | 10 10 10] = (84) Ans.

Try all partitions

end max ele

Try All Ways → Recursion → Take Best

front Partition

f(0) give me max sum if array from 0 to n-1

f(ind) {

for(i = ind

```
f(ind){
  if (ind == n) return 0;

len=0, maxi = INT_MIN , maxAns = INT_MIN
for(j = ind; j < ~~ind+k~~ min(n, ind+k); j++){
    len++;
    maxi = max(maxi, arr[j]);
    sum = len*maxi + f(j+1)
    maxAns = max(sum, maxAns);
    }
  return maxAns;
  }
```

Memoization: dp[n] TC: O(N) X O(K)
SC: O(N) X O(N) ~~ASS~~

Tabulation:
```
dp[n+1];
dp[n]=0;
for(int ind= n-1; ind>=0; ind--){
    int maxi = INT_MIN;
    int len =0;
    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(sum, maxAns);
    }
  dp[ind] =maxAns;
  }
return dp[0];
```

Max Rectangle Area with All 1's

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

→ [1][0][1][0][0] largest ①

2×3=6 [2][0][2][1][1] 2

[3][2][3][2][2] & 2×3=⑥

[4][0][0][3][0] 4

ans.

create histogram at every row.

```
int maxArea = 0;   vector<int> height(m, 0);
for(int i = 0; i < n; i++) {
    for(int j = 0; j < m; j++) {
        if(mat[i][j] == 1) height[j]++;
        else height[j] = 0;
    }
    int area = Histogram(height);   → call histogram fn to find max rectangle area in a histogram
    maxArea = max(area, maxArea);
}
return maxArea;
}
```

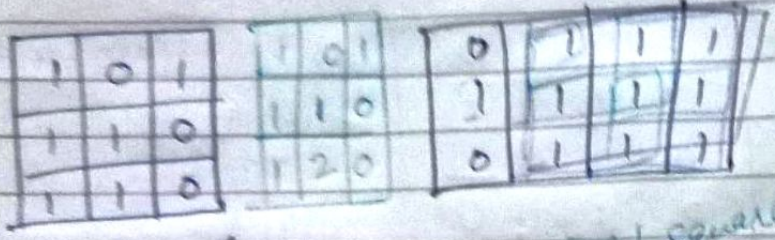
TC: O(N*(N+M))

SC: O(N)

Date 26/7/22

DP On Rectangles

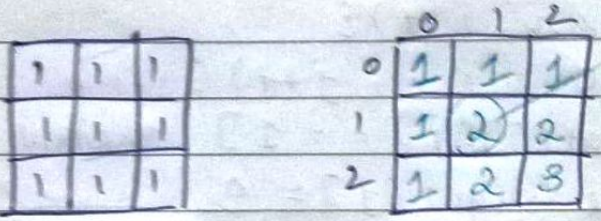
Count no. of Square Submatrices



6 size 1 squares
1 size 2 square
total = 7

10 size 1 squares
4 size 2 square
1 size 3 square
15 total

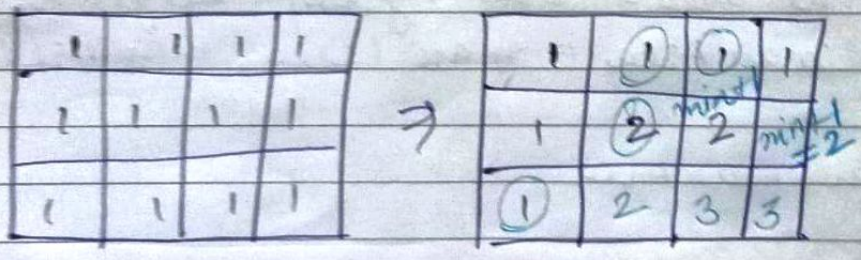
Brute: Stand at a particular index & try to expand it keep checks.



→ min (all surrounding) + 1

← sum ⇒ total squares

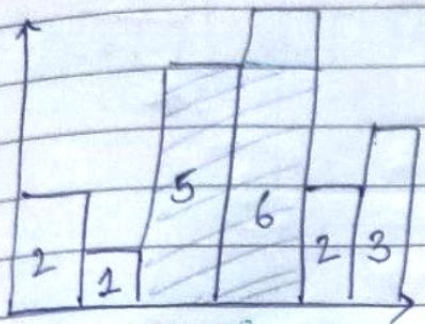
dp[i][j] how many squares end at i,j



first row & col is replicated

```
// copy first Row & col
for(int i=1; i<n; 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], dp[i-1][j-1], dp[i][j-1]);
    sum+=dp[i][j];
  }
}
return sum;
```

Largest Rectangle in Histogram



$5 \times 2 = 10$

Brute: at 2, 9 can take max $2 \times 1 = 2$ (will take consecutive)

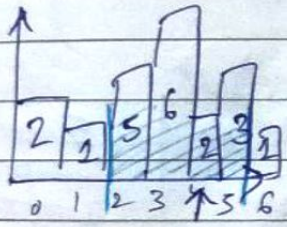
at 1, $1 \times 6 = 6$

5, $5 \times 2 = 10$

6, $6 \times 1 = 6$

2, $2 \times 4 = 8$

3, $3 \times 1 = 3$



if at 2, check right for 1st smaller
check left for 1st smaller
that will give the width

check for All ⇒ Brute force Soln.
$O(N^2)$

Dry run on leetcode

better

left small

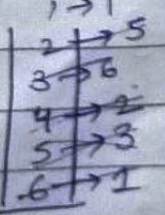
0	1	2	3	4	5	6
0	0	2	3	2	5	0

if ele is ? pop else put index in stack

& ele to add ele in stack

right small

0	1	2	3	4	5	6
0	6	3	3	5	5	6



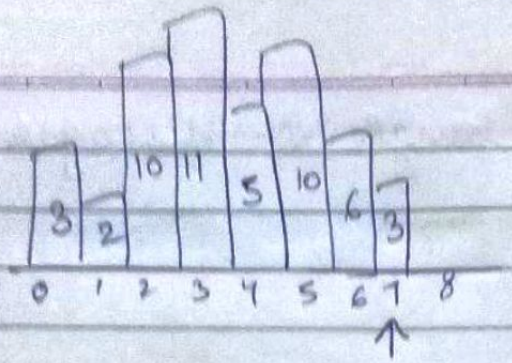
$TC \rightarrow O(N) + O(N)$

$O(N) + O(N)$

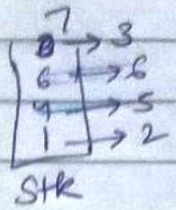
$\sim O(N)$

Date: YOUVA

Optimal



At a[7] = 3



At 6, 6 × (RS − LS + 1)

ar[7] ar[4]

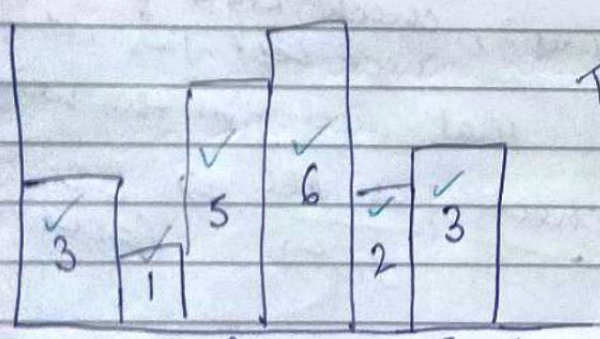
At 4, 5 × (ar[7] − ar[1] + 1)

At 1,

At a[8], RS = 8

LS = a[2]

maxA = 0 3 6 10



TC: O(N) for stack
+O(N)
SC: O(N) stack

for 0, LS ⓪ RS ① 3 × 1 = 3

a[1] = 1

a[3] ⇒ LS ② RS ④ 6 × (4 − 2 − 1) = 6
6 × 1

a[4] = RS ⑥ LS ②
(6 − 1 − 1) × 2
= 2 × 4 = 8

a[2] ⇒ LS ① RS ④ 5 × (4 − 1 − 1)
5 × 2 = 10

a[5] ⇒ LS ④ RS ⑥ 3 × (1) = 3

a[1] = RS ⑥ LS ⓪

```
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;
  }
}
```