DP

Maximum sum of Non adjacent elements

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

1 2 4 — O: 5
2 1 4 9 — O: 11

S1 Let's try all subsequence with given constraint

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

pick one with max sum

$f(ind) \rightarrow$ max sum (your picking) $(0 \ldots ind)$

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 max(pick, not pick);
}
```

dp[ind] =

// dp[n] because original array also has n elements.

TC → O(n)
SC → O(N) + O(N)
stack + array

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

9+2 = 11

f(3)

f(1) = 2

f(-1) = 0

f(0) = 2

f(2)

f(0)

f(1)

TC → O(2^n)

S2 optimize Recursion ⇒ Overlapping subproblems ⇒ Memoization

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

T.C O(n)

S.C O(1)

curi

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

prev2 = prev

prev = curi

return dp[n-1] prev

Space Optimization

prev2 prev (i) (i)

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

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

O(N)

O(1)

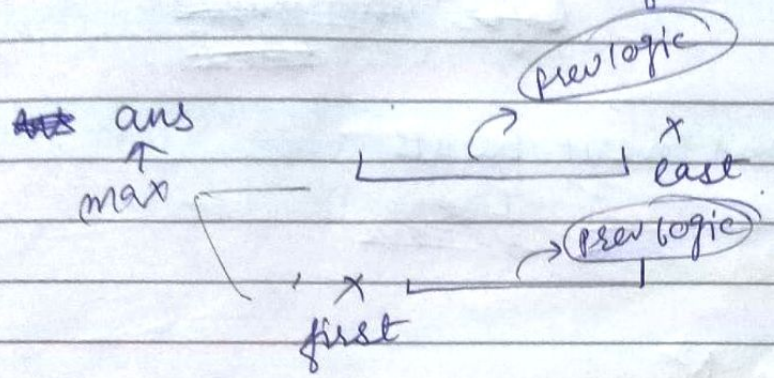
prev2 = prev

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	50	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 start~~.

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

~~0 2 3rd index~~

Recursion starts from n-1.
~~start with f(n-1, 3) 0 0 all tasks get used~~

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[~~day~~ ind][i] + f(~~day~~ ind - 1, i)
      maxi = max(maxi, points);
    }
  }
  dp[ind][last] = maxi;
  return maxi;

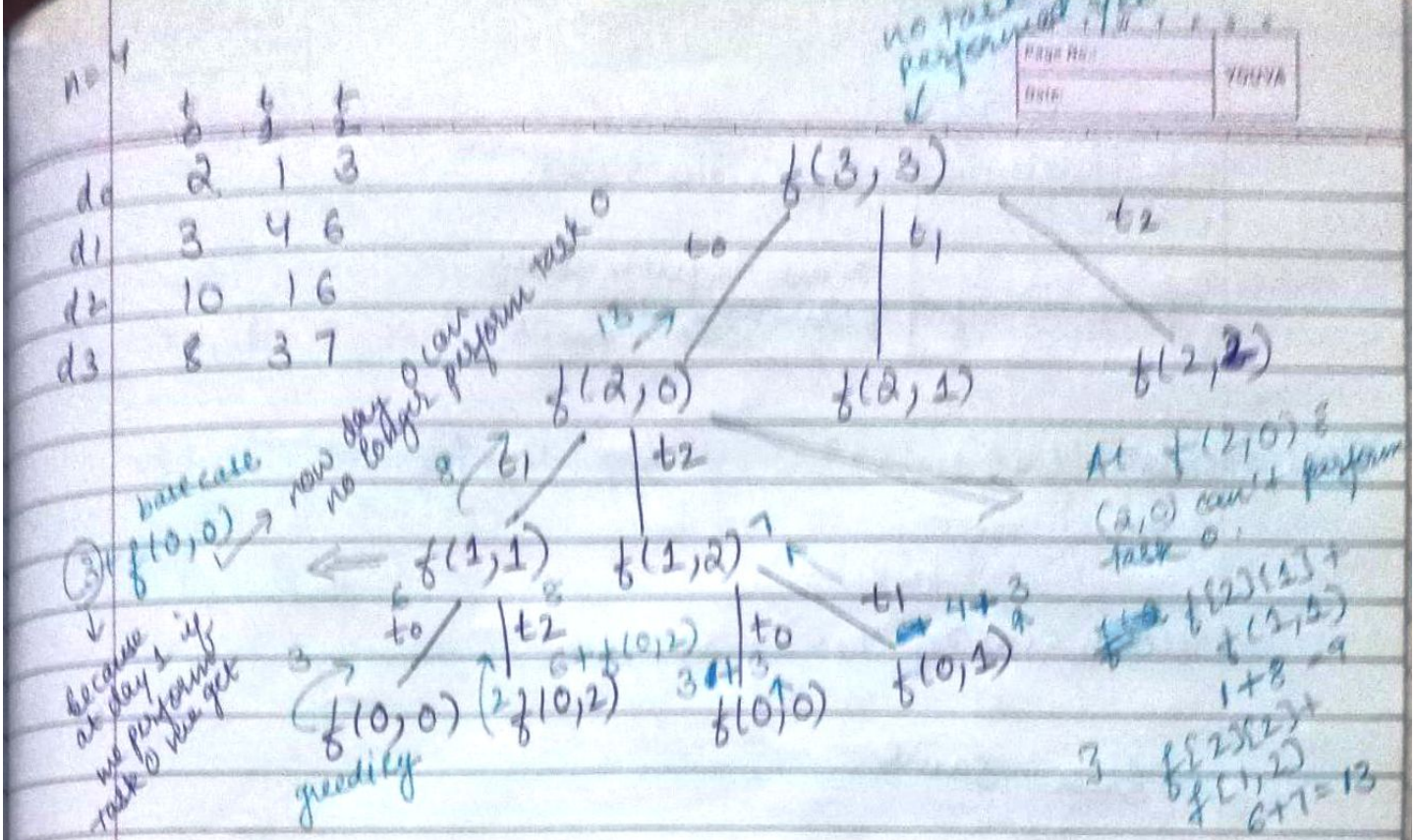
because on last day we can greedily pick best. As there is no next day to consider.

for recursion call for previous day and tell him this ith task was performed.

total steps
1 O(N×4)×3
2 O(N)
3 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

dp[N x 4] N 4 → (0, 1, 2, 3) ← no task

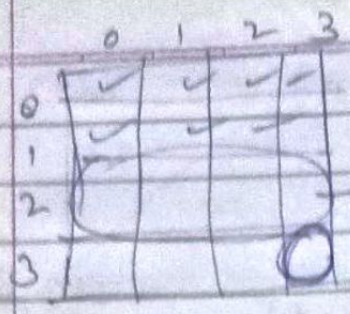
Tabulation :

```
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, included
{
    for (last = 0 → 3) {    // last, 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 come

task acts as last for dp table



// we require previous row

→ we require only previous row. Not row 0

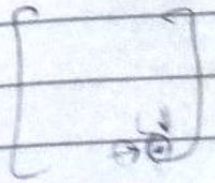
dp[day][last] = + day[day-1][task]

```
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

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

start from here $f(m-1, n-1)$ L U

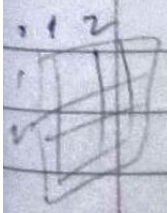
$TC \rightarrow 2^{m \times n}$

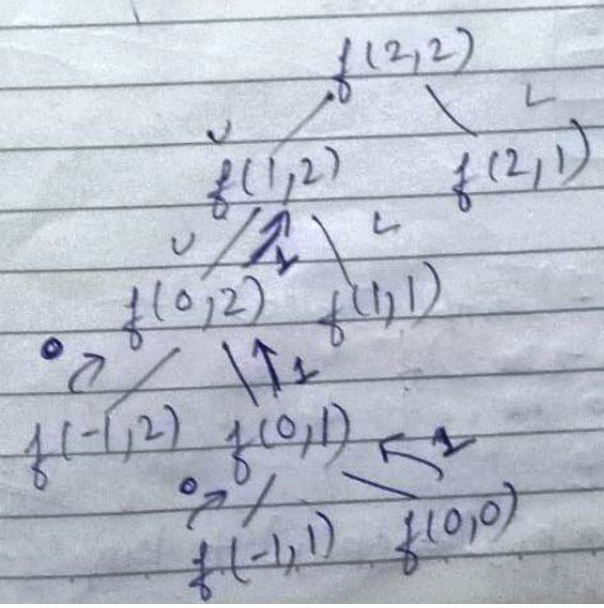
```
f( i, j) {
If (i==0 && j==0){
    return 1;
}

If( i<0 || j<0) return 0;
up = f(i-1, j);
left = f(i, j-1);
return up + left;
```

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

dp[i][j] =

m=3
n=3





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

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

Tabulation

```
dp[n][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]
```

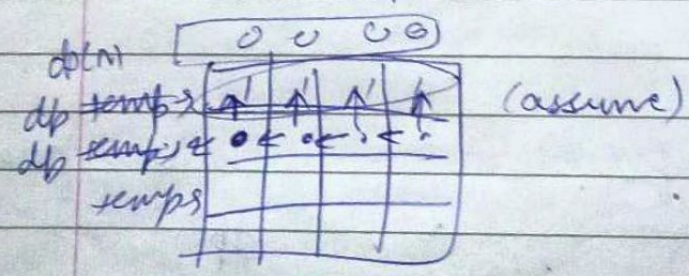
TC $O(n \times m)$

SC $O(n \times m)$

Space optimization

prev & prev column (now)

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




```
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)

(i,j) 0 → n-1 0 → m-1

Tabulation

```
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

0 1 2 3 4 5 6

l l r r r r r r

mp

B	~~1~~ ~~2~~	3
A	~~1~~ ~~2~~	~~3~~ 2 3

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

at index 2, res = 3

3 − 2 = 1 ≤ 2 valid

TC: O(n) × O(26)

at ind 5, 6 − 3 = 3 ≤ 2 ✗ not valid

because finding maxCount each time in mp

Inc l, and dec count of s[l]

5 − 3 = 2 ≤ 2 true

```
max(out( 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 = max(out(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 path

Date:

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 index on row j+1 in each step.

→ Represent (i,j)

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

→ Pick min

fixed

1

2 3

3 6 7

Variable ⇒ 8 9 6 10

TC: $2^{1+2+\dots n}$

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);
}
```

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

dp[n][n];

TC: O(n×n) no. of states

SC → no of states O(n²) RSC

Thumb Rule: Tabulation is always opposite of Recursion

Page No.: Date: YOUVA

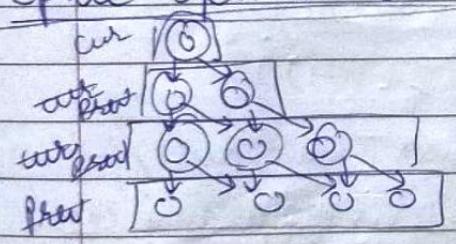
Tabulation:

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);
    }
}
print(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];
```

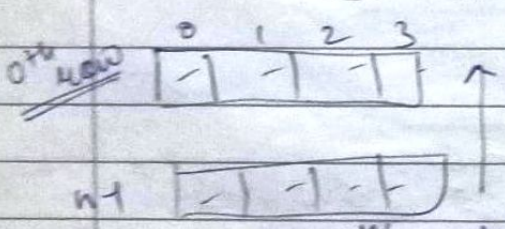
Max^m Path Sum In Matrix

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



MAX

f(n-1,0) f(n-1,1) f(n-1,2) f(n-1,3)

TC: 3^n ≈ exponential
SC: O(N) RSS

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

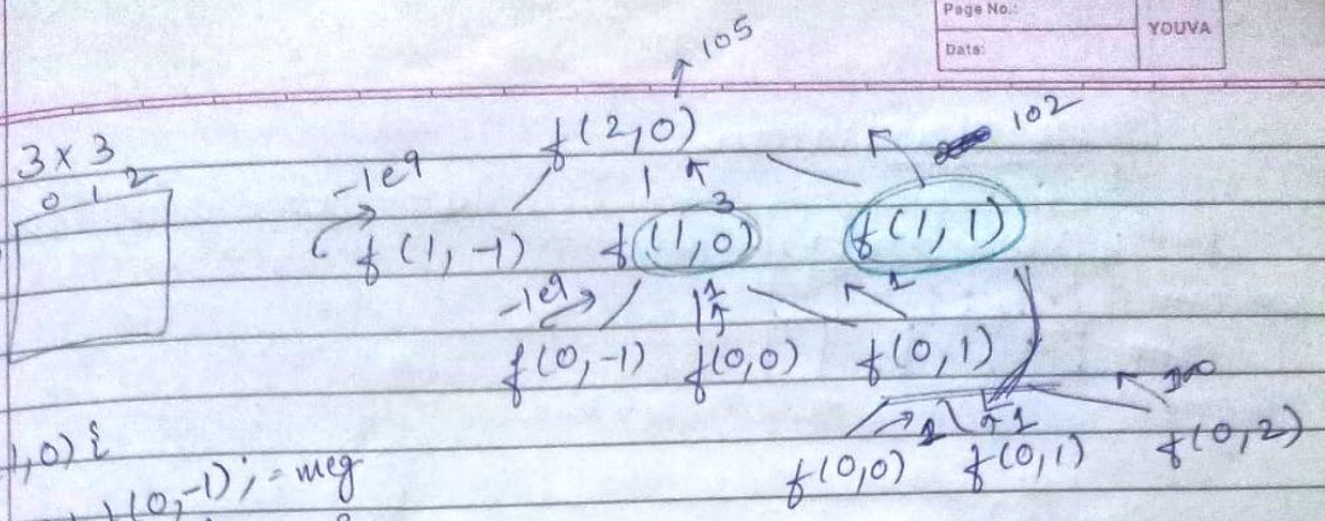
dp[n][m];

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

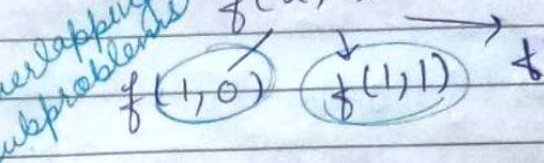
col<0 col>m-1

TC: O(n×m)
SC: O(n×m)
O(n) RSS




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

overlapping subproblems



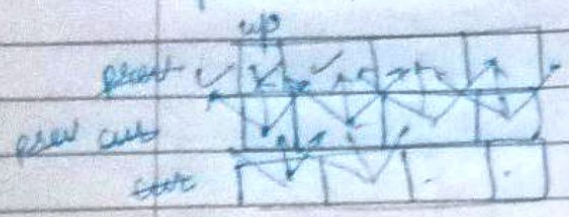
1. base cases
2. observe states i, j and convert to loops.

Tabulation:

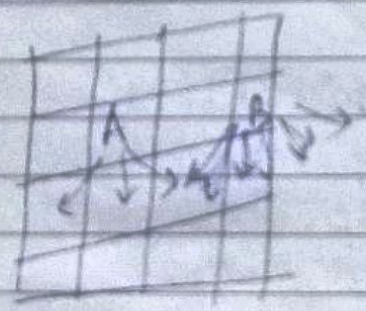
```
dp[n][m];
for(j=0;j<m;j++){
    dp[0][j] = a[0][j];
}
for(int i = 1; i<n; i++){
    for( j -> 0 to m-1){
        u = a[i][j] + dp[i-1][j];
if(j-1 ≥ 0)   ld = a[i][j] + dp[i-1][j-1];
  if(j+1 < m) rd = a[i][j] + dp[i-1][j+1];
        dp[i][j] = max(u, ld, rd);
    }
}
maxi = dp[n-1][0];
for( j = 1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

TC: O(n×m)
SC: O(N×M)

Space optimization (just store prev row)




```
prev(n), cur(n)   // initialize prev
  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.

i+1, j-1 | i+1, j | i+1, j+1

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

for (i) -> 1 -> n

for (j) -> -1 -> +1

3x3 = 9 combos

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 first case Bob will move down so, it will always be same

A 2	3	1	B 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(i, j1, i2, 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];
  } mani=0;
  for(dj1 → -1 to +1) {
    for(dj2 → -1 to +1) {
      if(j1==j2) mani=max(mani, a[i][j1] + f(i+1, j1+dj1, j2+dj2))
      mani=max(mani, a[i][j1]+a[i][j2] + f(↙))
  } return mani;
```

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

Memoization
i j1 j2
[N][M][M]

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

TC: O(n×m×m) ×9

(1) Base case
(2) Express states in for loops i, j_1, j_2
$n-1 \to 0$ ($0 \to m-1$)

Tabulation :

```
dp[n][m][m];
for(j1 -> 0 to m-1) {
  for(j2 -> 0 to m-1) {
base case
    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 mani = -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+d2>=0
                                          j2+d2 < m)
              value += dp[i+1][j+dj1][j+dj2];
else value += -1e8;
              mani = max(mani, value);
            }
        }
        dp[i][j1][j2] = mani;
      }
    }
  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.

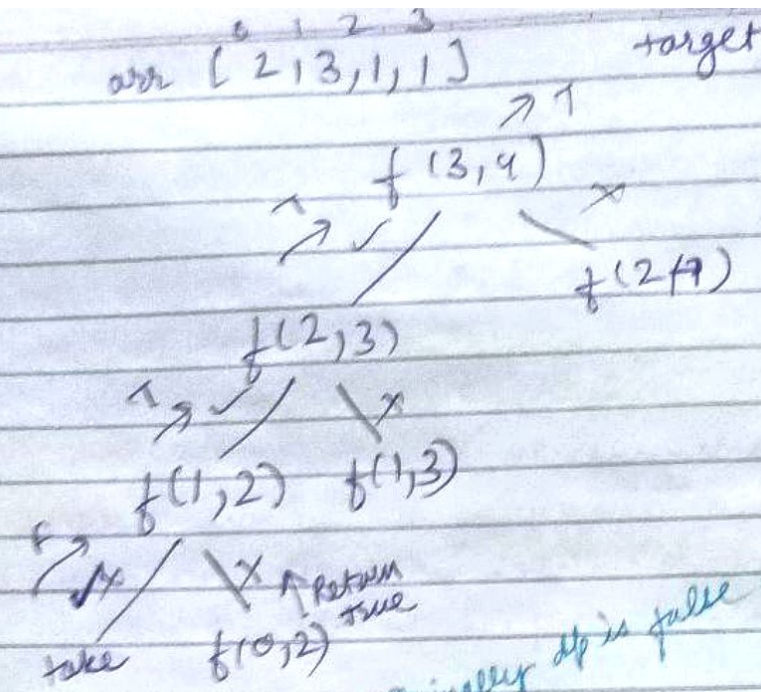
→ Recursion
① Express index, target
② Explore possibilities of that index
pick / not pick
③ return T/F

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

```
f(ind, target){
    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]);
    return (take) or (not take);
```

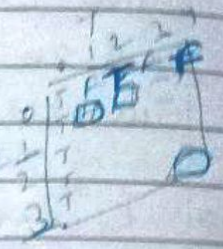
memoization dp[ind][target] = 1 TC: (O N×target)

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



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

Finally dp is false



Tabulation:

```
for (i = 0 to n-1)
    dp[i][0] = true;

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

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

// for every ind if targ is 0 its true,

TC $O(n \times target)$
SC $O(n \times target)$

Space optimization

i=0 [T | F F F F F]
i=1 [T | - - - -]
curr prev

(... 1 & are positive)

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

$3 - 7 = |4|$

[1, 2, 3, 4] $\{(1,2)\ (3,4)\}$

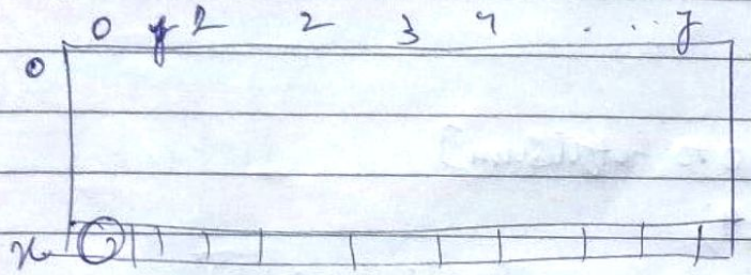
$\{(1,3)^4 - (2,4)^6\} = |2|$

$\{(1,4)^5 - (2,3)^5\} = |0|$ Ans.

ex: {3, 2, 7}

total = 12

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



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

So, if total sum is 12

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

(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] <= total sum  dp[0][arr[0]] = true;

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

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(n-1, s)

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

this will be not pick case. because even if el = 4 & sum = 0, I can not pick and meet cond^n

TC $O(2^n)$

SC $O(n)$ AS

before

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

return dp[ind][sum] =

$O(N \times sum)$ TC

$O(N \times sum) + O(n)$

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}
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];
```

→ 2mp, because here we are add^n later so can't be -1.

if (a[0] ≤ s)

otherwise overflow, because dp[n][s+1]

ex {1, 2, 3}

T = 3

n \ sum	0	1	2	3
0	1	1	0	0
1	1	1	1	
2	1			
3	1			

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 × curr_ans.

	0	0,1
4×1	0	1,0
=4.	0 0	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 element is a[ind] we will not pick it.

```
If (s==ar[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 ✓ ✗

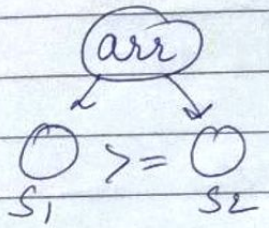
If 1 and sum = 0 we can ✗

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.



$$\boxed{S_1 - S_2 = \Delta}$$

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 $- S_2 - S_2 = \Delta$ S_1 = totalSum $- S_2$

TotSum $- \Delta = 2 \times S_2$

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

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

so totSum $- \Delta$ cannot be negative (how can sum of subset that has the no. be neg).

they can't be fraction totSum $- \Delta$ has to be even.

Constraints

$0 \le arr[i]$

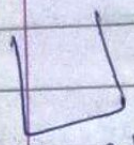
(So this code will 0 cases in mind)

0/1 Knapsack

n=3

wt →	3	4	5
val →	30	50	60

Total → 60+30 = 90



Bag → W=8

Try all combinations ⇒ take best

↓

Recursion

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

→ Explore all possiblities

Pick ✓ Not pick X

→ Pick Max (all possiblities)

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

f(n-1, W)

Let

	3	2	1
	30	40	60
ind	0	1	2

f(2,6)

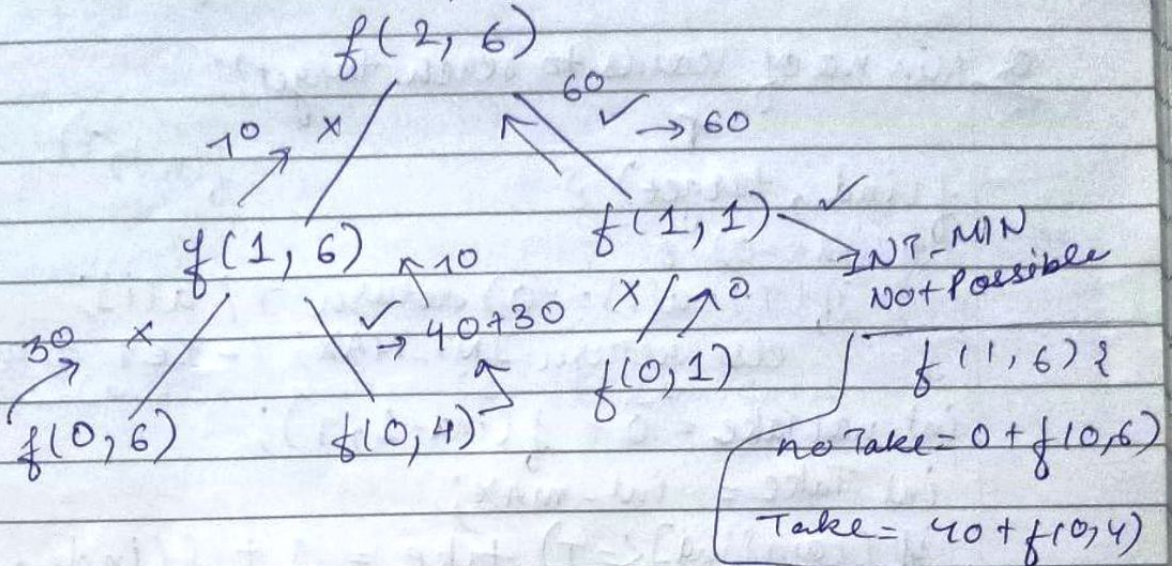
⇓

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 (W, ind) 0 → W, 1 to n-1
3. Same Recurrence in loop

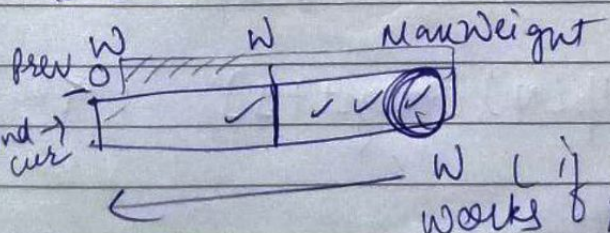
```
int dp[N][W+1]; = {0}
for ( i = wt[0] → W) dp[0][i] = val[0];

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

// because if wt coming is more than its own wt we can take it

bagweight

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

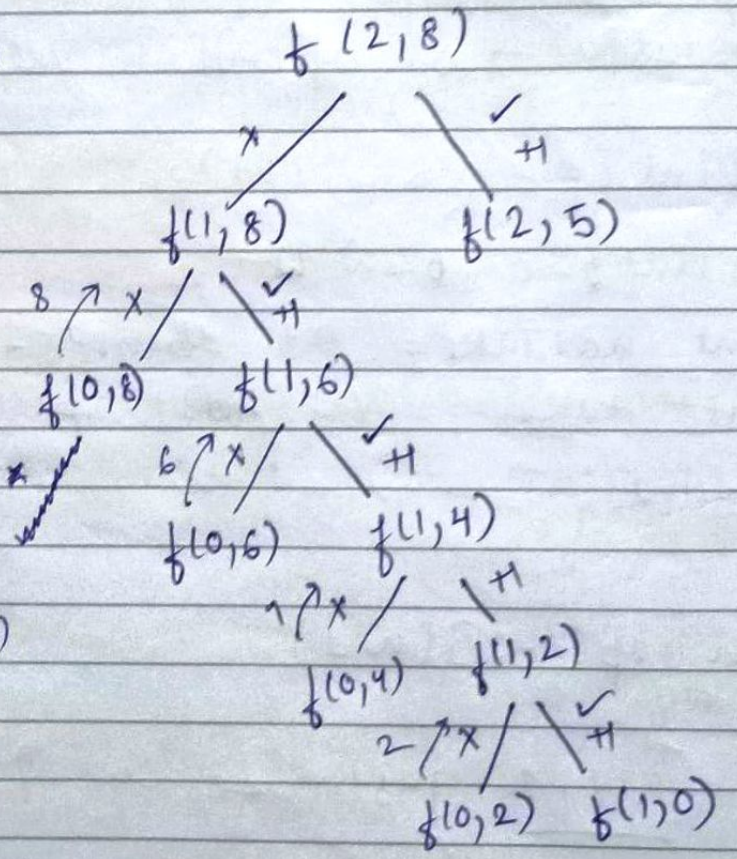
Infinite supply of coins

arr[] → { 1, 2, 3 } (0, 1, 2) target = 7 (3, 3, 1) ③

Q Min no of coins to reach target??

```
                T                    f(n-1, T)
f(ind, target) {
  if (ind==0) {
      if(T % a[i]==0) return T / a[i];
      else return INT_MAX; (* 1e9 to avoid overflow);
  }
  if(dp[ind][T]!=-1) return (dp[ind][T]);
  int notTake = 0 + f(ind-1, T);
  int Take = int_MAX;
  if (coin[ind]<=T) take = 1 + f(ind, T-coins[ind]);
  dp[ind][T] =
  return min(take, notTake);
}
```

arr = { 1, 2, 3 } (0, 1, 2)
tar = 8



TC: → $O(2^n)$ exponential

SC: $O(N)$ ↑↑ more than $O(target)$

Memoization: dp[N][target+1]; TC $O(N \times T)$ SC $O(N \times T)$ + ASS

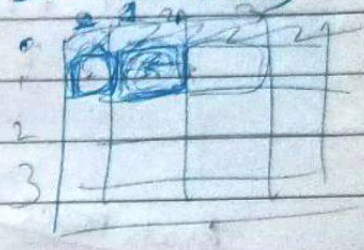
Tabulation: base case changing Parameters Recurrence

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

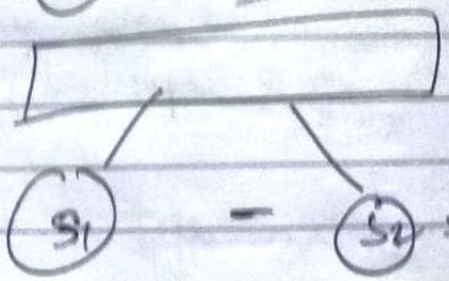
+1 +2 -3 -1 = -1
-1 +2 +3 -1 = 3 ✓ (2 ways)
+1 -2 +3 +1 = 3 ✓

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



divide into 2 subsets

Brute Force

f(arr, T)
{
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.

Coin Change 2

value
coins >= 1

1 2 3 target = 4

you are given an array you have to form target. Any no ~~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(1)~~ ASS
>> O(N)
O(target) Worst case

Memoization

dp[N][T]

TC: O(N×T)
SC: O(N×T) + 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

Maximize value

wt → { 2, 4, 6 }
val → { 5, 11, 13 }

W = 10

f(ind, ~~wt~~ 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)

Memoization: Same as before. (prev ques)

Tabulation dp[N][W+1]

for (W → 0 to → W) { original weight of bag

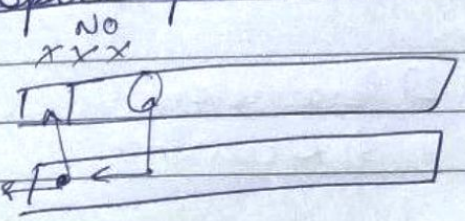
dp[0][W] = $\frac{W}{wt[0]}$ × val[0] → W that is changing in loop

}

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



prev [3 4 5 7 12]

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

2 5 3 = 12
1 2 2

2 2 2 2 2 = 10
1 1 1 1 1

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.

$f(\overset{n-1}{4}, N)$
till index 4 what is max price you can obtain

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

f(ind, N) {
if(ind==0) {
return N X 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

Longest Common Subsequence

abc → a, b, c, ab, bc, ac, abc, ∅ " "

Can be consecutive or not but maintains order
↳ Subsequence

S_1 = "adcbc" S_2 = "dcadb"
adb adb

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

→ Generate all subsequence && compare on way

0 1 2 0 1 2
acd | ced
↑ ↑

// 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
O^2 (f(ind1-1, ind2), f(ind1, ind2-1))

shrink the string and find there

f(1,1) a | ce
↑ ↑

+0 +0

// Try moving both because could match further

a | ce
ind1 ind2

ace | e
ind1 ind2

+1

f(ind1-1, ind2-1)
(negative)

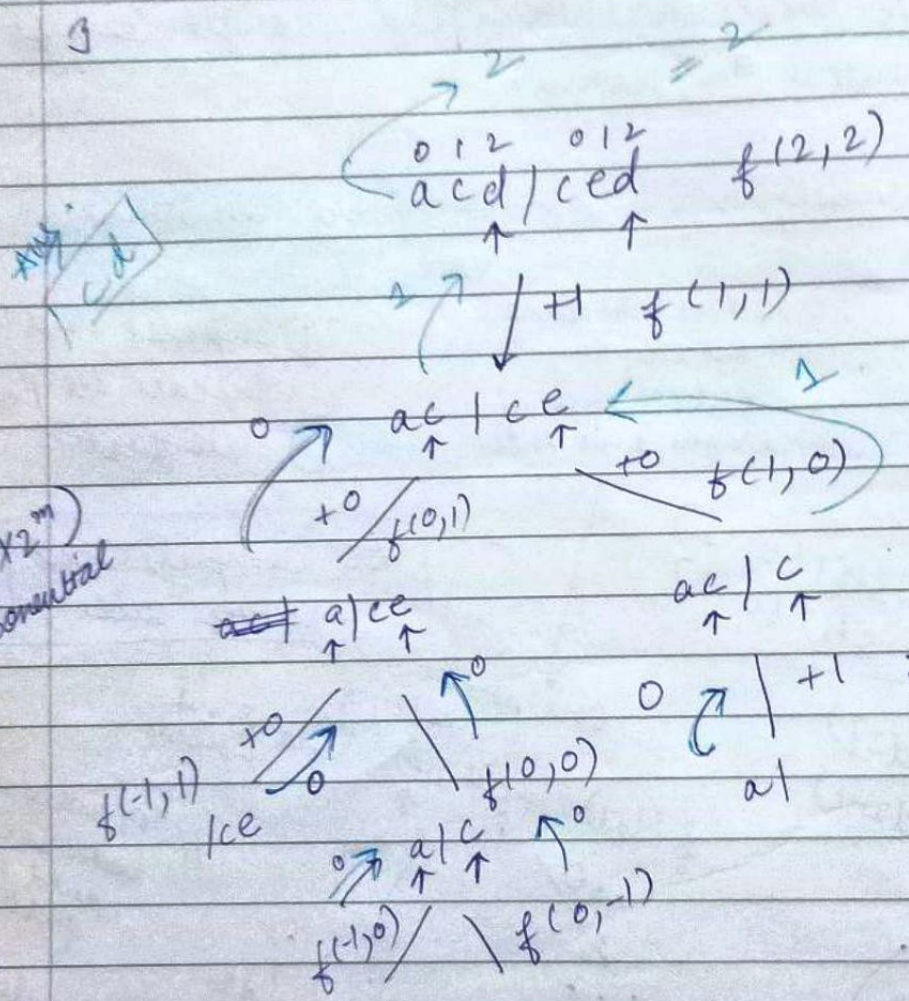
// neg means end of string

```
f(ind1, ind2) {
  if (ind1<0 || ind2<0)
        return 0;
  if (dp[ind1][ind2] != -1) return dp[ind1][ind2];

  if (s1[ind1] == s2[ind2])
  return 1 + f(ind1-1, ind2-1);

  return 0 + Max(f(ind1-1, ind2), f(ind1, ind2-1));
}
```

Base case

Match

No match



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

Overlapping Subproblems ⇒ Memoization

$f(i,j) \rightarrow$ lcs of str1[0 .. i] str2[0 ... j]

dp[n][m]

↑ len of str1 ↑ len of str2

TC: $O(n \times m)$

SC: $O(n \times m) + O(n+m)$

Date: YOUVA

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

→ Shifting of Index because neg is not possible

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

f(0)
f(1)

f(n-1, m-1)

f(n, m) // Right shift Index

f(i, j)
↓ ↓
i-1 j-1

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

dp[0][j] dp[i][0]
0 to m-1 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 look at s1(n-1) & s2(m-1) because we have shifted & we need to check prev ind 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

i \ j	0	1 b	2 d	3 g	4 e	5 k
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.

abcde bdgek

dp[5,5] ⇒ lcs = 3

abcde bdgek

lcs = 2

dp[4][2]

abcd bd

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-1;
}
```

TC: O(N+M)

Return ans;

Longest Common Substring
↳ continuous

S1 = "abcjklp" S2 = "acjkp"

Longest common subsequence:

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

nomatch: $dp[i][j] = max(dp[i-1][j], dp[i][j-1])$ ✗✗

can'omit in substring

acd | axd

ac | ax

a|ax ac|a

|ax a|a ac| a|a

abcd abzd

		a 1	b 2	z 3	d 4
	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 could have been made

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

ans: Max Val In Matrix

prev & curr (space optimize)

In prev code,

Not max = 0 every time

Keep ans as global & update in loop

Longest Palindromic Subsequence

S = " bbbab " bbbb ans 4

s1 = " bbabcbcab "

babcbab

write 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 abcbaa 2 insertions

↓ reverse

abcaa aacba (Now this whole string is palindrome) string

st rev(S) Max operation = len( )

you can insert any char anywhere

Approach: 1) keep the Palindrome portion intact

abcaa → aa, a, aba, aaa) aca etc.

2) Longest portion intact

a b c a a c b a

a b a c a b a

Ex: coding ninjas

codingni njasidoc
Sajn

cadisajn ingninjasidoc

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

Min operations (Ins/Del) convert to make String A to String B

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)

$\boxed{\text{total op} = n + m - 2 \times len(lcs)}$

Shortest Common Supersequence

⇓

make a string such that S1 & S2 is in it

⇓

order has to be same

length = ??

string = ??

both

S1 = "brute" S2 = "groot"

ex. brutegroot (can be possible answer)

Ans bgruoote (len = 8)

S1 = bleed S2 = blue

"bleued"

S1 = brute S2 = groot

bru goo

lcs = "rt"

bgruoote

① common guys taken once.

⇓

lcs

Ans = n + m − len(lcs)

			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

figuring the string

br / gro

groo bru / gro bru

groot brut

ans

bruote

reverse to get ans

etouorbg (we will get this)

```
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, tell how many times s2 appears in s1.

s = "babgbag" s2 = "bag"

ans: s babgbag
ba bgbag
babg bag
⋮

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 s1 from [0...i] in s2 [0...j]

(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
 take
 notice

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

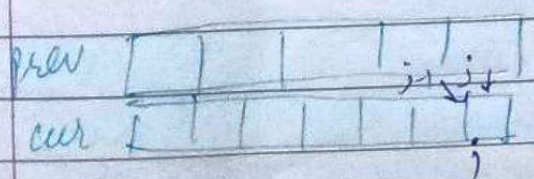
More optimized

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 execition
insert u execution

String Matching path

horse (i)
ros (j)

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 matches a hypothetical match, why would i move

Base case

S1 gets exhausted

-1 horse f(-1,1)

ros (2 operations to make ro)
min to convert empty string to S2[0...1]
2 Insert

return j+1

S2 gets exhausted

S1 = horse S2 = ros
0 1 2 -1

min operations to convert S2 to empty string.

Memoization

TC: O(N×M)
SC: O(N×M) + ~~N~~ 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) Wild Card Matching

S1 = "?ay"
S2 = "ray"

S1 Matches S2
true

? → matches with single char

* → matches with sequence of length 0 or more.

S1 = " ab*cd "
S2 = " abdefcd "

S1 Matches S2
true

ab*cd 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
SC O(N+M)

ab*
ab de f
ab / abdef ab* / abde

ab / abde

S1 is exhausted
$\boxed{i<0}$ && 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 $O(N \times M)$ SC $O(N \times M) + O(N+M)$

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(str1 [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 }

BS BS

4 3 = 7

~~BS~~ ~~BS~~ Not allowed

7 1 5 3 6 4

B S

5

0	1	2	3	4	5
7	1	5	3	6	4

↑

f(0, 1) → starting on 0th day with buy whole max profit you can make

buy = 1 you can buy

buy = 0 you cannot buy

Buy as many times as you want sell as many times as you want 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 option sell/Not sell
  - Whenever you are buying, at 1 you are losing -1 to the answer and if selling at 0 it's +1

f(ind, buy)

- if (ind == n) return 0;

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); }
```

Overlapping Subproblems → Memoization

TC $O(N \times 2)$ SC $O(N \times 2)$ dp[N][2] // because buy can be either 0 or 1

ASS + $O(N)$

Tabulation

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

```
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] + ~~
            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 transactions

$\rightarrow 0 \quad \rightarrow 1 \quad \rightarrow$ initial = 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)

either 0, 1 or 2

Memoization: dp[N][2][3]; TC: N×2×3
SC: N×2×3 + ASS

Tabulation: 1) Base case 2) changing Parameters 3) copy Recurrence

n→1→0
0→2 0→2

{ 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];
```

<u>space optimization</u>: 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));
  }
```

<u>Tabulation</u>:

```
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 cond" )
```

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}

ind ↑ ind

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, nottake);
```

TC: 2^n

SC: ASS O(N)

not checking union -1 here, because prev_ind+1 me sirf store ho rahe hai cuel 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, 0 → 1, 1 → 2, 2 → 3, ... n-1 → n

Tabulation:

```
dp[n][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;

3
3

return dp[0][-1+1];

1

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

space optimization : next & curr
(ind+1)

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

Explanation Another Method

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

// longest possible

ans = from (0 → n-1) max(dp[i]).

dp[i] → signifies the longes 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.

because 3 now is a subsequence that ends at 5 which I can take. So at 3 = 1+1. Check with 4 also. But since max increases 2 no updations

we'll go and check prev which we can take

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

1	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

18 → 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}

// where 4 can fit in existing subsequence

{1, 7, 8}

we can replace 7 with 4 here even if 4 is not the 1,4,8 subsequence, we still do know now it's less something existing and 4 is no bigger in no bigger.

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

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

optimizes space

arr[] → {1, 4, 5, 4, 2, 8}

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

4 remove

In (++) 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 {
        ind 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

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

Print any answer ⇒ Thought process

1) sort the array {1, 4, 7, 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 this as longest divisible subsequence

$\boxed{a|b,\ b|c \Rightarrow a|c \quad \text{given } a<b<c}$

// 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"}

[a, ba, bda] [b, ba, bca]

[a, ba, bca]

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 begining because this is also subset.

```
for (i = 1 → n)
    for (j = 0; j < i; j++)
        compare (arr[i], arr[j])
        if (arr[i] > arr[j] && dp[j] + 1 > dp[i])
            dp[i] = 1 + dp[j]
        }
        Do maxi (maxi, dp[i])
    }
return maxi;
```

you can form arr[i] by adding only one char to arr[j]

LIS

arr[j] → bda

arr[i] → bdca

b d c a

b d a

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

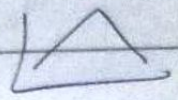
```
                    greater guy
bool compare(string &s1, string &s2){
    if (s1.length() != s2.length())
                        return false.

    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())
```

s2[second.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

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

LIS

```
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;
    }
}
```

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.

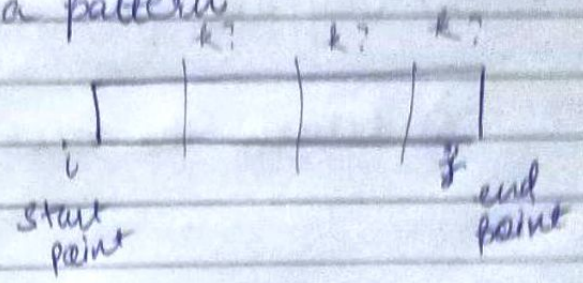
arr[] = {1, 5, 4, 3, 2, 6, 7, 10, 8, 9}
dp[] = 1 ~~1~~2 ~~1~~2 ~~1~~2 2 ~~2~~3
count[] = 1 1 1 1 1 ~~2~~ ~~3~~ 4 ~~4~~ ~~5~~

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

No we check for len 3:
1, 5, 6
1, 4, 6
1, 3, 6
1, 2, 6

DP On Partition

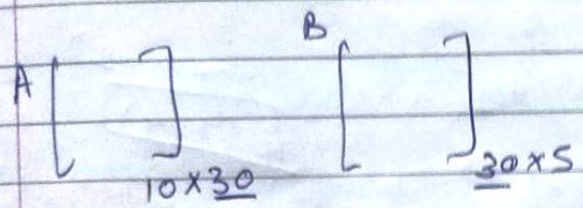
solve a problem in a pattern



Matrix Chain Multiplication

ABC

A = 10×30
B = 30×5
C = 5×60



operations = 10×30×5

$\overset{10\times30}{(AB)}\ \overset{30\times5}{C}$ = 10×30×5 operations

$[\,]_{10\times5}\ [\,]_{5\times60}$ = 10×5×60

= 1500 + 3000 = 4500 operations.

$\overset{30\times5}{A}\ \overset{5\times60}{(BC)}$ = 30×5×60

$\underset{10\times30}{A}\ \underset{30\times60}{(BC)}$ = 10×30×60

= 9000 + 18000 = 27000 op.

Min operations = 4500 → Find this

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

given arr[] → [10, 20, 30, 40, 50] n = 5

dimensions of n-1 matrices

$\boxed{n^{th} \to A[i-1]A[i]}$

ABCD

A[0][1] A[1] A[2] A[2] A[3] A[3] A[4]

A → 10×20 B → 20×30 C → 30×40 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

i j
(ABCD)

ij i j ABC(CD) | ij i j A(BCD) | i j ij (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 multiplication for matrix 1 to matrix 4.

```
f(i, j) {
Base case  if (i == j) return 0;
           min = 1e9;   if (dp[i][j] != -1) return dp[i][j];
           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);
           }
           dp[i][j] =
           return min(all steps);
```

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

k = (i → j-1)
1 3

f(i, k), f(k+1, j)
1 1 2 4

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

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)

TC → exponential

Memoization ⟹ Overlapping Subproblems

dp[N][N]
TC: O(N×N) × N ⇒ O(N^3)
↓ due to loop
SC: O(N^2) + O(N)

10 20 30 40 50

i k k j

A BCD

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

(10×20) (20×50)

steps = 10×20×50

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

10×30 30×50

A = 10×20

B = 20×30

C = 30×40

D = 40×50

B C D

(20×30) (30×40) (40×50)

20×40 30×50 40×50

20×50 20×50

A B C D

10×20 20×30 30×40 40×50

10×30 30×50

Recusion Tree

~~ABCD~~

i k k j

f(10 20 30 40 50)

A B C D

0 A i j

[20] [30 40 50]

i j B C D

AB CD ABC D

20 30 40 50 20 30 40 50

i j i j i i j i j

10×20×50 + f(A) + f(BCD)

0

BCD

30 40 50

i k k j

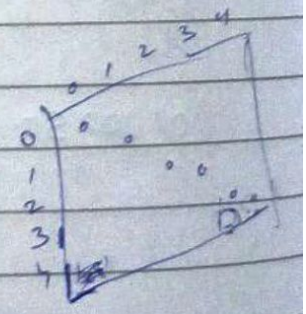
0 BCD

B CD

C D

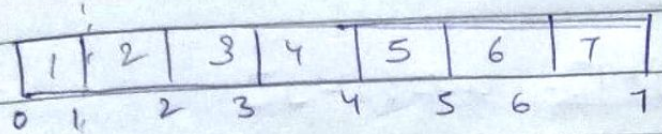
Tabulation : 1) Copy base case
2) Changing parameters
$i = n-1 \to 1$ $j = i+1 \to n-1$
3) Copy Recurrence

```
dp[N][N]                                   // f(1,1) represents
for(i = 1; i<N; i++){                         min cost to
    dp[i][i] = 0;                             multiply
}                                             mat1 to mat4

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

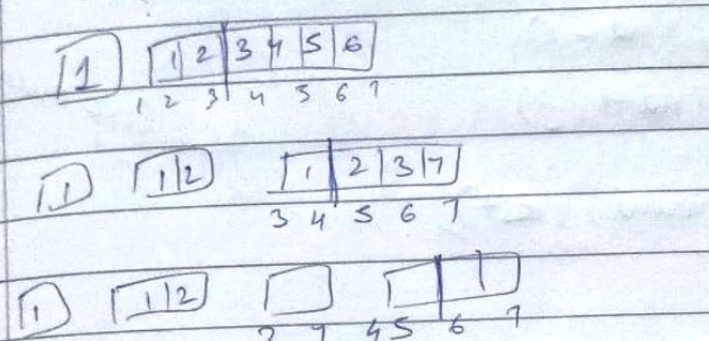
arr [] = [1, 3, 4, 5] n = 7



len of stick at time of cutting

7 + 6
+ 4 + 3
= 20

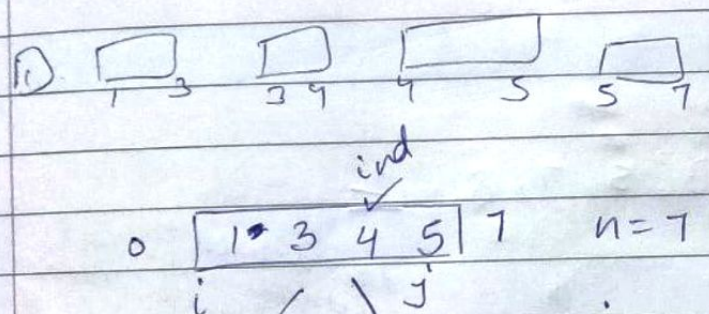
cost = len of stick you are cutting

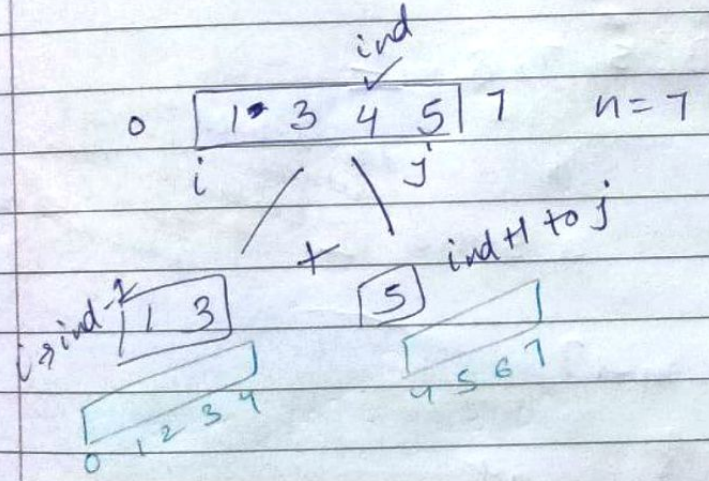


you can follow cuts in any order

arr = [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
13

now 2 is dependent on right part

cost = length of that stick in which it 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 j

0 1, 2 3 — i j

3 5, 7, 8 12 — i j

don't need to attach anything because array to pass logic only i, j change

Memoization: dp[M+1][m+1]

size of original cuts array

TC: $O(M^3)$

SC: $O(M^2) + ASS$

Tabulation:

// base case entire dp is initialized 0.

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

$i \to n \to 1$

$j \to 1 \to n$

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++) {
```

Date:

cost = cuts[j+1] − cuts[i−1] + dp[i][ind−1] + dp[ind+1][j];

mini = min(mini, cost);

}

dp[i][j] = mini;

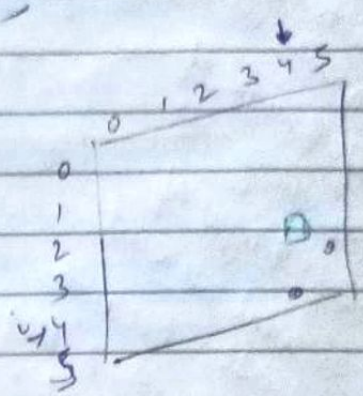
}

}

return dp[1][c];

}

}



Burst Ballons

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

every ballon 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

~~5×1×3~~ 3×1×5

[3 ~~5~~ 8] → 3×5×8

[~~3~~ 8] → 1×3×8

(~~8~~) → 1×8×1 decide last guy

we can go opposite and whatever is left & right of our range, is left & right of element we want to burst. because all else is burst about it.

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 [b1 b4] or [b2 b4] or [b3 b4] or [b4 b5] or [b4 b6] Independent subproblem

[b4] 1×b4×1

0 1 2 3 4 5 6

1 b1 b2 b3 b4 b5 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)

max = max(cost, max)

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++){
                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];
}
```

if (i>j) continue;

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 2 ways.
(F | T) ^ F
F | T

(T ^ F | T & F ^ T | F)

at every operand, we can do a partition

+2 +2 +2 +2

T ^ F | T & F ^ T | F

i j

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 in left is T

ways in which right is T.

left | right

F=x4 F=x3

T=x1 T=x2 x1 * x2

x1 * x3

x4 * x2

left ^ right

T=x1 T=x2

F=x3 F=x4

T^T=F

F^F=F

T^F=T

F^T=T

isTrue = if looking for true = 1

false = 0

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

total = x1 * x4 + x3 * x2

f(i, j, isTrue)

{

initial call

f(0, 10, 1)

if (i > j) return 0;

// its single guy

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

base case

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)

T | F ^ T

1 way to make it true

(T) (F ^ T)
i j

i j

T | F T
i j i j

1 way

```
if ( a[ind] == '&' )
if (is true) ways += LT X RT ; else ways += (LT*rF) +
                                            (lF*rT) +
                                            (rF*lF)
Else if (a[ind] == '|') {
if (is true) ways += lT * RT + lT * RF + RT * lF
                                  // write false cond" also
Else if ( a[ind] == '^' )
if (is true) 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 = bab|abc ba|d|c|e bde

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

bab abcba dcede (from partitions)

can you make partition

1+ abab cbadcede

1+ abcbadcede

1+ cbadcede

return min

1+ bcbadcede

1+ dcede

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 = " ";~~ 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)

Date:

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);
      }
  }
  ~~return~~ 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 → ~~max~~ length at Max

can only have 3 elements

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

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

Try all partitions

0	1	2	3	4	5	6
1	15	7 15	9	2	5	10
15	15					

j

Ans. ~~Ans~~ ele

✓ (1 15 7 | 9 | 2 5 10)

[15 15 15 | 9 | 10 10 10] = 84

Try All Ways → Recursion → Take Best

~~f(ind) {~~

~~for(j = ind~~

front Partition

f(0)

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

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

 len=0, mani=INT_MIN , maxAns=INT_MIN
for(j=ind; j< ~~ind+k~~ min(n, ind+k); j++){
      len++;
      mani = max(mani, arr[j]);
       Sum = len*mani + 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 mani = INT_MIN;
      int len=0;
      int maxAns = INT_MIN;
  for(int j=ind; j<min(ind+k, n); j++){
      len++;
      mani = max(mani, num[j]);
      int sum = len* mani + 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

largest

→ 1 0 1 0 0 — 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);
    maxArea = max(area, maxArea);
}
return maxArea;
}
```

→ call histogram fn to find max rectangle area in a histogram

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

SC: O(N)

Date 26/7/22

DP On Rectangles

Count no. of Square SubMatrices

1	0	1
1	1	0
1	1	0

1	0	1
1	1	0
1	2	0

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

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.

1	1	1
1	1	1
1	1	1

	0	1	2
0	1	1	1
1	1	2	2
2	1	2	3

min (all surrounding) + 1

sum ⇒ total squares

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

1	1	1	1
1	1	1	1
1	1	1	1

⇒

1	1	1	1
1	2	2	2
1	2	3	3

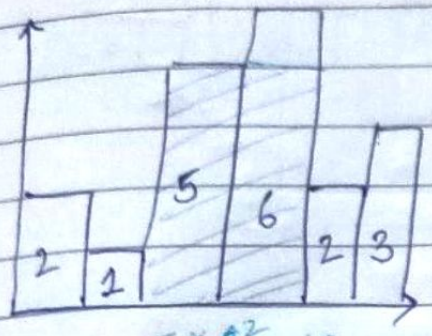
min 1

min 1 = 2

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 $2 \times 1 = 2$ we take consecutive

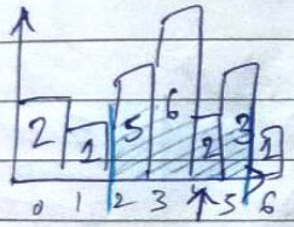
at 1, $1 \times 6 = 6$

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 Sol^n.
$O(N^2)$

Done on leetcode

better

left small

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

5	→ 3
4	→ 2
3	→ 6
2	→ 5
~~1~~	~~→ 1~~

val in array

stack → ind

if ele is > pop
else put index in ans
& ele to add ele to stk

right small

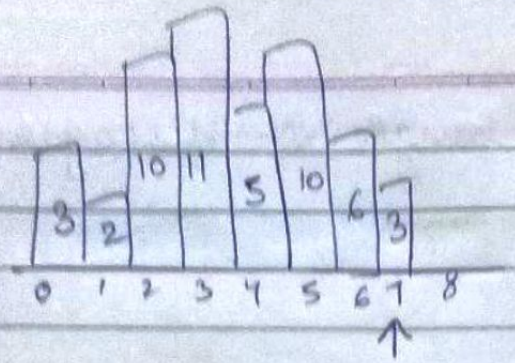
0	6	3	3	5	5	6
0	1	2	3	4	5	6

1	→ 1
2	→ 5
3	→ 6
4	→ ~~2~~
5	→ 3
6	→ 1

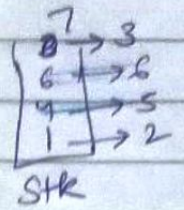
stack → ind

TC → $O(N) + O(N)$
$O(N) + O(N)$
$\approx O(N)$

Optimal



At a[7] = 3



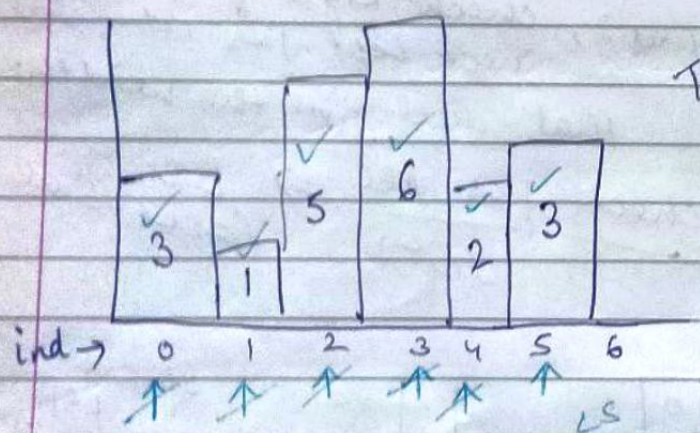
At 6, 6 × (RS − LS + 1)
a[7] a[4]

At 4, 5 × (a[7] − a[1] + 1)

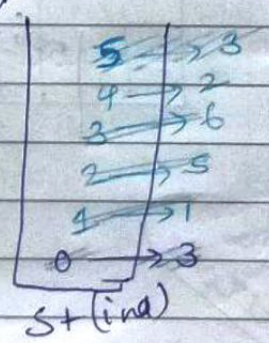
At 1,

At a[8], RS = 8
LS = a[1]

maxA = 0 ~~3~~ ~~6~~ 10



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



a[1] = 1

for 0, LS (0) RS (1) 3×1 = 3

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)
if LS = 0, RS = 6

```
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;
  }
}
```