DYNAMIC PROGRAMMING

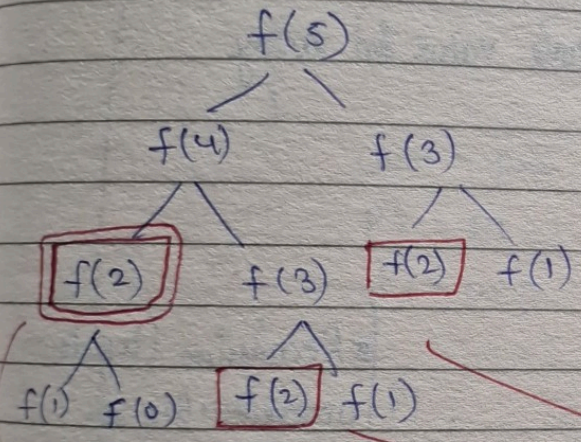
Raj
DATE / /
PAGE

① Memorization

Tend to store value of subproblem in Map/array/table

eg: Fibonacchi series 0 1 1 2 3 5 8 13 21

RECURSSION TREE



→ Already computed value → It is again calculated (BAD)
So we use memorization

BASIC CODE

```
f(n) // requires nth term.
{
  if(n<=1) return n;
  return f(n-1) + f(n-2)
}
```

memo[]

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

whenever you calculate a subproblem update the array.

Raj
DATE / /
PAGE

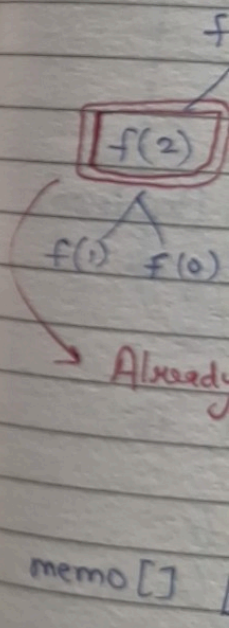
SHORTCUT

1) Try to represent in terms of Index
2) Do all possible "Stuff" on that fresh states
   ↳ count ways : Sum of stuff
   min way : find min of stuff

2D : DP Steps :

MEMO	TAB
1) Out of Bound · Check	1) Initialise dp
2) Base Case	2) Start traversal opposite of MEMO
3) Pre calculated dp value	3) Base case (inside loop)
4) Recursive call in all directions	4) else : Move in direction with Help of dp[][]
5) Store & return dp[i][j] = max/min of direction + arr[i][j]	5) Same

eg:



Raj

DATE / /
PAGE

DP code of fibonacchi

TC : O(n)
SC : O(n) + O(n)
↓ recursion stack ↓ array.

```
f(n)
{
  if(n<=1) .n ;

  if(memo[n+1] != -1) return memo[n]; // returns already stored value

  return memo[n] = f(n-1) + f(n-2); // Store value in array
}
```

② Tabulation (Bottom - Up)

↳ Base case to required answer.

* Recursion (top-down)
answe → Base case

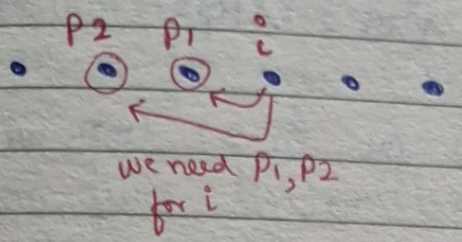
```
{ dp[0] = 0, dp[1] = 1

  for (i=2 to n)
  {
    dp[i] = dp[i-1] + dp[i-2]
  }
}
```

TC : O(n)
SC : O(n)

TAB code of fibonacchi

Raj
DATE / /
PAGE

let's eliminate space.



So no need to store everything in arr[] since we only need 2 prev. value for each ith

```
int P1 = 0, P2 = 1
for (i = 2 to n)
{
    curr = P1 + P2
    P2 = P1
    P1 = curr
}
return P1;
```

Most optimise tabulation code of Fibonacchi

TC : O(n)
SC : O(1)

HOW TO DETERMINE Q. is of Dynamic Pro.

- TRY all Possible ways = recursion ⇒ DP

1) Count total number of ways
2) Among multiple ways, determine max / min

1-D

Raj
DATE / /
PAGE

Q. Climb Stairs : Determine no. of ways to climb n stair if you can only take 1 or 2 steps

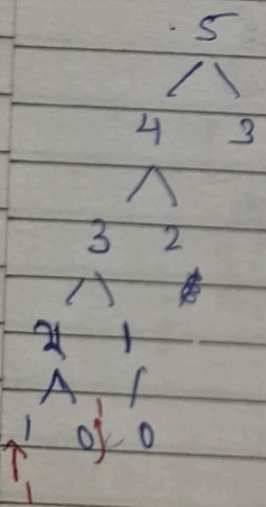
~~for~~ from n we can go to (n-1) or (n-2) until we are at 0/1

Basic code (recurssion)

```
f(n)
{
  if(n==0) return 1;
  if(n==1) return 1;
left = ~~int~~ f(n-1)
right = f(n-2)
return left + right
}
```

DP + Tabulalation Code

```
if (n==1) return 1
int p1=1, p2=1, curr=-1;
for ( i=2 to n)
{
   curr = p1 + p2
   p2=p1
   p1= curr
}
return p1+p2
```



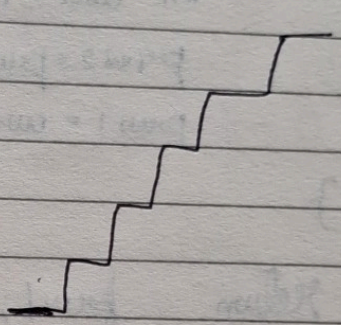
Method for Tabulation

1) ~~First~~ write code for memorization
2) then make Base case
3) Write Loops.

FROG Jump.

DATE Raj
PAGE

Q1. from 1st stair (total N stair), frog wanna go to top
HEIGHT[i] is the height of [i+1] stair. if frog jump
from i^{th} to j^{th} stair : energy lost = HEIGHT[i+1] - HEIGHT[j+1]
He can jump (i+1) or (i+2) th stair. find min energy used
by frog to reach N^{th} stair.



Recursion (Top down) code

```
f(ind)
{
  if(ind==0) return 0
  left = f(ind-1) + abs(arr[ind] - arr[ind-1]);
 right = f(ind-2) + abs(arr[ind] - arr[ind-2]);
 return min(left, right);
}
```

Now, Using DP

1) Memoization : look at parameter that are changing

1 → store value before you return
2 → Always check if value is computed or not

```
f(n)
{
  if(n==0) return 0
if(dp[n] != -1) return dp[n];
```

check

```
left = ~~ same
right = ~~ same
return dp[n] = min(left, right);
```

store

2) Tabulation (Bottom-up)

- int prev1 = 0 , prev2 = 0 // Base case

for (i = 1 to n)
{
    int l = prev1 + abs(arr[i] - arr[i-1]),
if(i>1) int r = prev2 + abs (arr[i] - arr[i-2]);

    int curr = min(l, r)
    prev2 = prev1
    prev1 = curr
}

return prev1;

Q2. Instead of 1,2 jump, you can take 'k' jumps

```
for (i=1 to n)
{
  minsteps = INT_MAX
  for(j=1 to k)
  {
    if (i-j>=0)
    { jumpE = dp[i-j] + abs(dp[i-j]
                - dp[i]);
    }
  }
  minsteps = dp[i] = minsteps
}

return dp[n-1]
```

↓

min energy to reach last step

HOUSE ROBBER.

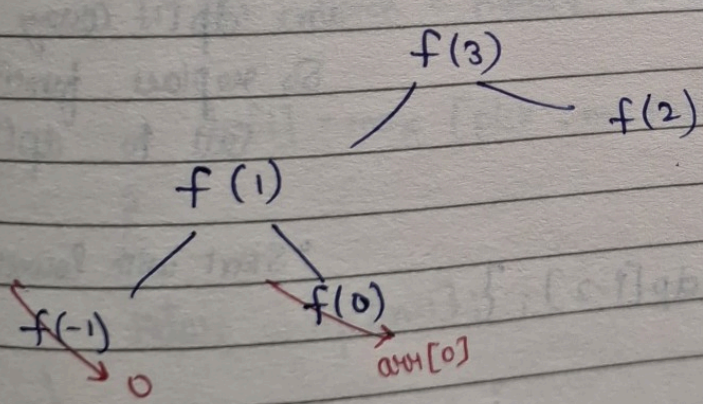
Raj
DATE / /
PAGE 1

Each house has certain money and you can't rob adjacent houses, so, find max. money you can rob

eg: INPUT: [1,2,3,1]
OUTPUT: 4

1) Recursion

```
f(n)
{
  if(n==0)  // you reach zero only when you not pick "1"
  {                    // Best to pick it
     return arr[n];
  }                 // if you reach -ve index then return 0;
  pick = arr[n] + f(n-2)
  not pick = 0 + f(n-1)

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



- Memo
  - ↳ for i^{th} element we only
  - Store f(i) value in dp[]

Raj
DATE / / /
PAGE

(2) Memorization

```
int f( int ind , vector<int> &nums)
{
    if (ind == 0) return nums[ind] // index = 0 then return arr[0]
    if (ind < 0) return 0;
    if (dp[ind] != -1) return dp[ind];
    int Pick = nums[ind] + f(ind-2, nums)
    int notPick = 0 + ~~nums[ind]~~ f(ind-1, nums)

    return dp[ind] = max(Pick , notpick);
}
```

(3) Tabulation. (Bottom - Up)

Base Case

```
dp[n] = 0;   dp[0] = 0

for( i=1 to n)
{
    int take = 0
    if (i-2 > 0)
    { take = arr[i] + dp[i-2]; }

    not take = 0 + dp[i-1];
    .
    dp[i] = max(Pick, not Pick)
}

return
```

- we store f(i) value in dp[i] array So replace funcn call to dp[i]
- Start with lowest index

instead of dp[] You can take prev1 and prev2 to store (i-1), (i-2) th value

* dp[i] = max money robbed from 0 to ith house

Raj
DATE / /
PAGE

int Rob(vector<int>& nums)

CODE

```
{
    // no of Houses
    int n = nums.size();

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

    vector<int> dp(n,0);

    dp[0] = nums[0];
    dp[1] = max(nums[0], nums[1]); // Rob first | second   OR

// we start from '2' we need value of '0','1'

    for(i=2 to n)
    {
        int Pick = nums[i] + dp[i-2];
        int notpick = 0+dp[i-1];

        dp[i] = max[pick, not_pick];
    }

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

2D

Raj

DATE / /
PAGE

Q. Ninja training has 'N' days with 4 activities each day he can perform any 3, each activity have some merit points. He can't do same activity in two consecutive days
find max points ninja can attain

INPUT: 1 2 5
3 1 1
3 3 3

	A	B	C
Day 0	10	50	1
Day 1	5	100	11

OUTPUT: 11 (5+3+3)

ANS: Day 0: A Day 1: B = 110 pts

Keep a track of task you did on previous day

① Recursion (top down)

```
f(day, last)
{
  1> Base case
  if(day=0)
  {
    for(i=0 to n) if(i!=last) maxi = max(maxi, task[day][i])
    return maxi;
  }
  Other wise find max task (pts) except last
  maxi=0
  for(i=0 to n)
  {
    if(i!=last)
    { points = task[day][i] + f(day-1, i);
      maxi = max(maxi, points);
    }
  }
  return maxi
```

Raj

DATE / /

PAGE

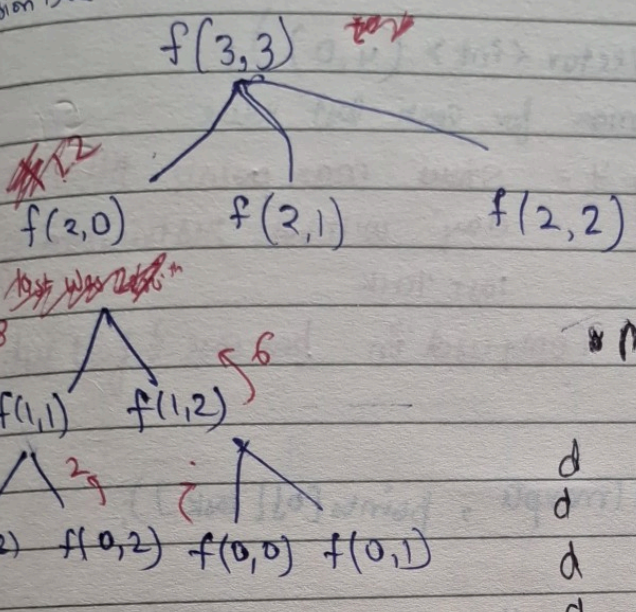
Eg:

Using DP, we will need a matrix for memorization

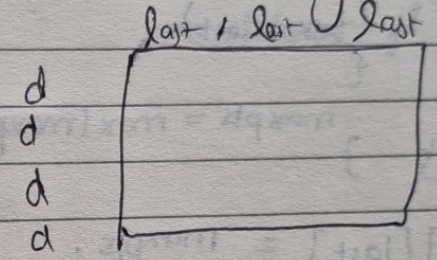
n = 4

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

Recursion Tree:



* Memo Array.



Simlay add

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

return d[day][last] = maxi

(3) Tabulation (Buttom - up)

Base case remain same

for(days)

{ for(last 0 to 4)

Raj
DATE / /
PAGE

CODE

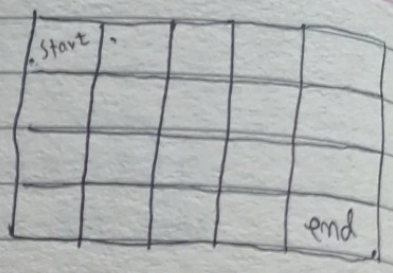
```
int ninja (int n, matrix : points );
{       dp array of  n x 4
  vector<vector<int>> dp (n, vector <int> (4,0));
  // Base case : fill day[0] by max for each last value
   for( last =0 to 4)  why 4 = store max points till that
  {  for   maxpts = 0;                'day' without restriction of
      for( task =0 to 3)                last task
      {                          (only used in base case & final return)
        if (task ! last)
          {
    }  }  }  maxpts = max (maxpts , points[0][task]);

    dp[0][last] = maxpts.   // fill dp array with maximum
}

// Now fill dp array from 1 to (n-1)

for (day=1  to  n)
{ for( last = 0 to 4)
  { maxPts =
    for( task =0 to 3)
    {   int Point = Points [day] [task] + dp [day-1] [task];
        if( task ! = last)  maxpts = max (maxpts , points)
    }
   dp[day][last] = maxpts ;
}
```

UNIQUE PATHS

Raj
DATE / /
PAGE

m x n grid, move only right and bottom

Return Possible ~~unique~~ unique ways b/t start and end.



① Recursion + Memoization

```
void ways(int i, j, m, n, &ans)
{
  if (i==m-1 && j==n-1)
  {
    ans++;   return;
  }

if (i+1<m) way
if (j+1<n) way

}
```

Intution

number of down + right calls is equal to
||
No of Unique Path

```
int ways (int i,j,m,n, matrix &dp)
{  Base case
   if (i==m-1 && j==n-1)
   { return 1 }
   int down=0, right=0
if (i+1<m)
    down = ways (i+1, j, m, n, dp)
if (j+1<n)
    right = ways (i, j+1, m, n, dp)

return dp[i][j] = down + right;

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

UNIQUE PATH + OBSTACLES

Raj
DATE / /
PAGE

If there is obstacle in the Path (==1) then you can't travel that cell

- Tabulation

Start 0	0	0
0	0	1
0	1	0
0	0	end

Memorization

Just add a Base case when obstacle is tackled (return 0)

```
funct^n ( Parameter - - - )
{ // Obstacle / out of Bound
  if (i>=m || j>=n) return 0
if( Grid[i][j]==1) return 0

if (i==m-1 && j==n-1)
{ return 1 }

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

down = ways (i+1, . . .)
right = ways (j+1, . . .)

return dp[i][j] = down+right
```

MIN PATH SUM

Raj
DATE / /
PAGE

Given non-negative integers in (m x n) Grid find Path with min sum (top left to bottom right)

1	3	1
1	5	1
4	2	1

- Do Recurs + memoization (from (m-1, n-1) to (0,0)) + traverse opposite [left & up]

```
f(i, j)
{   // Reached (Base case)
    if( i==0 && j==0 ) return grid[0][0]
    // Out of Bound
    if( i<0 || j<0) return INTMAX
    // If Already Calculated
    if ( dp[i][j] != -1) return dp[i][j]
```

Otherwise, Now calculate ups and lefts

```
    up = grid[i][j] + f(i-1, j)
    left = grid[i][j] + f(i, j-1);

    return dp[i][j] = min(left, up);
}
```

TC: O(NXM)
SC: O(NXM) + O(path len)
(m-1) + (n-1)

Raj
DATE / /
PAGE

• Tabulation (Traverse opposite as of memorization)

```
int minPath (matrix: grid)
{   // Store Min Path sum
  matrix: dp (m, vector<int>(n, 0));

  for (i = m-1 to =0 )
  { for(j = n-1 to =0)
    {
      if(i == m-1 && j == n-1)
      { Base case.
        dp[i][j] = grid[i][j];
      }
      else
      {
        int down = right = INT_MAX
        if(i+1<m)  down = dp[i+1][j]
        if(j+1<m)  right = dp[i][j+1].

        dp[i][j] = grid[i][j] + min(down, right);
      }
    }
  }
}
return dp[0][0]
```

SPACE OPTIMIZATION.

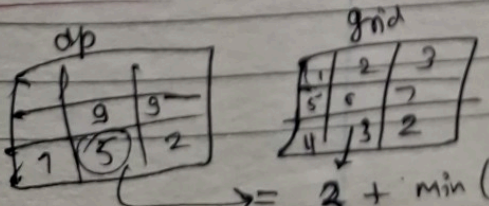
In every iteration we need.

dp[i+1][j]
dp[i][j+1]

Make them as variables

• dp[i][j] stores min Path sum from (m-1, n-1) to (i,j)

We are filling dp[][] in reverse while we still move down/right

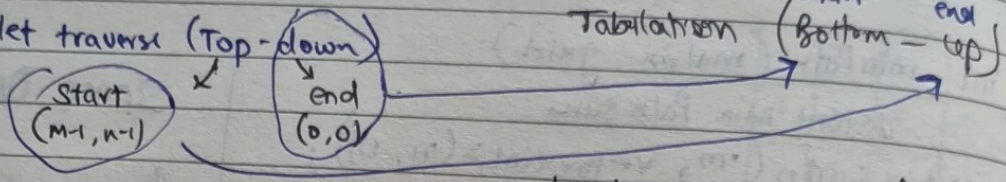


UNIQUE PATH - Tabulation

Raj
DATE / /
PAGE

Q Cal. unique Path frm $(m-1),(n-1)$ to $(0,0)$

Memo: let traverse (Top-down)




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

* dp[i][j] stores unique path from (i,j) to (0,0) by only taking left and up

* if we start frm (m-1, n-1) at last dp[m-1, n-1] will give unqiu paths

• Tabulation tips

1) Declare Base case
2) Compres of states in for loop
3) Copy the recursion call and replace by dp[][] arr

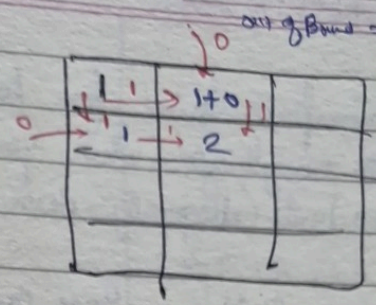
Convert (left) call to dp[i-1][j]
(up) call to dp[i][j-1]

Raj

DATE / /

PAGE

Tabulation



~~for~~ dp [m][n] = {all = 0}

```
for (i=0 to m)
{ for (j=0 to n)
{
    if (i==0 && j==0)
        return 1.

    else {
        int left=0, up=0;
    if(i-1>0) up = dp[i-1][j]
     if(j-1>0) left = dp[i][j-1];
```

→ Instead of left call we use left of dp.

```
    dp[i][j] = up+left;
    }
  }
}
return dp[m-1][n-1];
```

MIN PATH SUM: Triangular Matrix

Raj

DATE / /
PAGE

Q. find min path sum from top to bottom row in right angle Δ, when you can move (down) and (diagonally - down).

```
1
2 3
3 6 7
8 9 6 10
```

- Represent (i,j), Base Case
- Explore all paths (↓ ↘)
- Min of all paths

(Non Uniform Grid)

Recurr + Memo.

```
f(i,j)
{
  if(i==n-1) return arr[n-1][j]
  if(dp[i][j] != -1)
     return dp[i][j]
  d = arr[i][j] + f(i+1, j)
  dia = arr[i][j] + f(i+1, j+1);

  return dp[i][j] =
     min(d, dia);
}
```

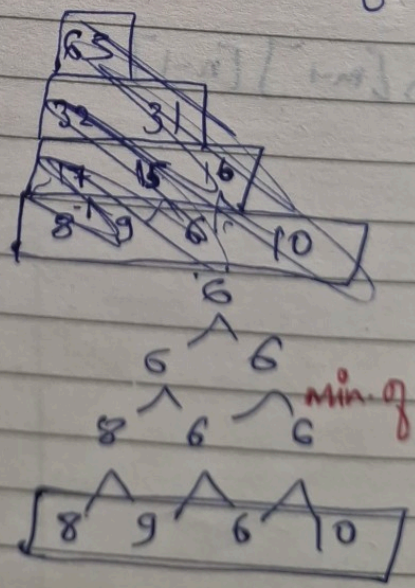
0 to (n-1)
row row

- Tabulation

→ If recurrence Then tabulate: 0 → n-1, n-1 → 0

1) first fill last row (Base) of DP matrix == last row.

Then from (n-2) row fill till up



SPACE OPT. ✓

~~matrix~~ dp = {0 all}

vector<int> front(n,0), curr(n,0)

// Base case: fill out last Row of DP

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

{

dp[n-1][j] = arr[n-1][j];

front[j] = arr[n-1][j]

}

for(i=n-2; i>=0; i--) Start from last second Row

{

for(j=i; j>=0; j--)

{

down = arr[i][j] + dp[i+1][j] — front[j]

dia = arr[i][j] + dp[i+1][j+1] — front[j+1]

(down: dp[i+1][j]; diagonally down: dp[i+1][j+1])

curr[j] dp[i][j] = min(down, dia);

}

front = curr;

}

return dp[0][0]; — front[0]

Raj

DATE / /

PAGE

SPACE OPTIMIZATION

We don't need dp[][].

we just need the prev row to make curr row

(eg: we need 3rd row to make 2nd row of dp array)

MAX/MIN FALLING PATH SUM

Variable point

DATE / /
PAGE

Q:- Given N×M matrix . find max sum in path bt 2 cells
- You can move down, diagonal (left and right).
- We can start and end at any point.

Start : first row any cell
end : last row any cell

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

- Express (i,j), Base Case
- Explore all paths
- Maxm among all path

lets move (n-1) to 0 (↙↓↘)

- Recurr + Memo

We will traverse through all the cells of last row and call funct to see which has max path

- Tabulation

STEPS:
1) Out of Bound CHECK
2) Base Case
3) dp calculated or not
4) move in direction
5) store min/max and return dp.

int n = matrix.size();

• Memoize. ((n-1) To (0))

Raj
DATE / /
PAGE

```
int f (int i, j, grid : matrix , dp)
{ ① → // out of Bound : return -ve value
    if ( j<0 || j>=n ) return INT_MIN;
  ② → Base case : reached first Row
    if(i==0 ) return matrix [i] [j];
  ③ → Already computed dp
    if(dp[i][j] != -1) return dp[i][j]

  ④ → // Recurssive call to 3 possible direction ( ↕ )
    int up = f( i-1 , j, matrix , dp)
    int dialeft = f( i-1, j-1 , matrix , dp)
    int diaRight = f(i-1 , j+1 , matrix , dp)
      Store dp = curr cell + max of 3 direct^n
return  dp[i][j] = matrix[i][j] + max( up, dialeft , diaRight);

}

int maxfallingPs ( matrix)
[
  int maxSum = INT_MIN
```

Call with each last row cell.

```
  for (j=0 to n)
  {
    maxSum = max(maxSum, f( n-1, j , matrix , dp)
  }

  return maxSum
}
```

traverse [0 to (n-1)]

Raj
DATE / /
PAGE

Tabulation

CODE

```
int maxFallingPS ( Matrix)
{ create dp[][] , first row equal to matrix's first row
① → // Initialise first row of dp
   for (j=0 to n-1)
   {
      dp[0][j] = matrix[0][j];
   }

② → Start filling dp (traverse) from 2nd row
   for(int i=1  to n)
   {
   for (j=0 to n)
   {  int diaLeft = diaRight = INT_MIN
③ → traverse all possible path via dp
   int up = dp[i-1][j];
if(j>0)  int diaLeft = dp[i-1][j-1]
if(j<n-1)  int diaRight = dp[i-1][j+1]

④ Store (max among all + curr matrix cell value)
   dp[i][j] = max(up, diaLeft , diaRight) + matrix[i][j]
   }
}
```

Max PS is max among last row of dp

```
   for (j=0 to n)
{
   maxSum = max(maxSum, dp[n-1][j]);
}
return maxSum
```

DP on SUBSEQ / SUBSETS

Raj
DATE / /
PAGE

Q Subset Sum = K : given array of 'N' +ve integers check if there is subset in array with sum = K

n-1 ↓ K ↓

f(ind, target) // Initial search from 0 to (n-1) slowly reduce the search index range

{

(1) // Base Cases

if (target == 0) return true ; // target value attained

if (ind == 0) {

{ if (arr[0] == target) return true

} else return false;

(2) // Make subseq by inclusion & exclusion

if (dp[ind][target]) return dp[ind][target];

bool notTake = f(ind-1, target) // reduce index

if (target > arr[ind]) // then only take

{

bool take = false; // Reduce index and target value

take = f(ind-1, target - arr(ind));

}

return (take || notTake); any 1 of them true = return TRUE

dp[ind][target] =

Add 2 steps of Memo

eg:

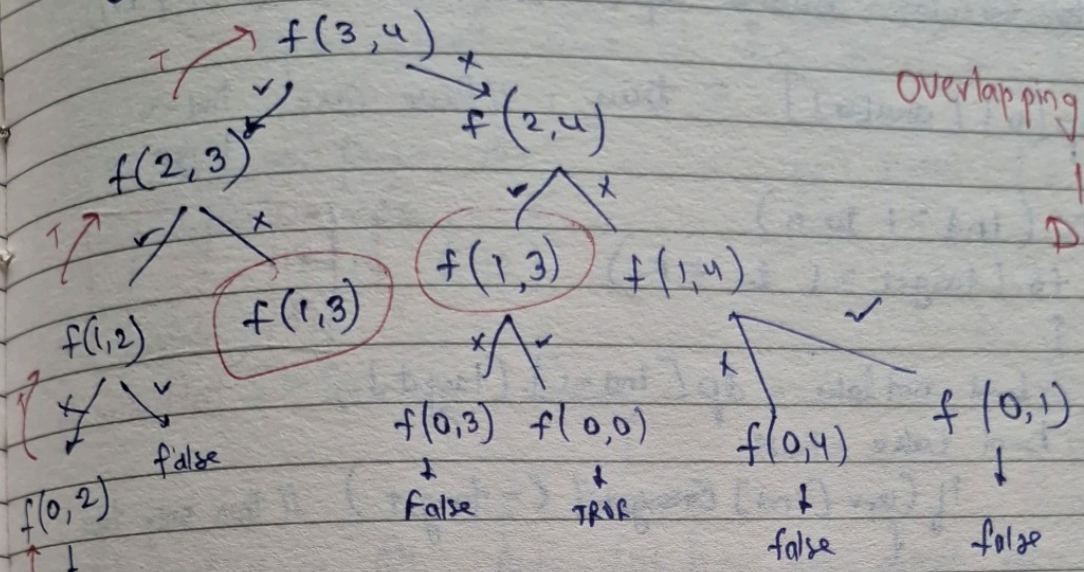
f(1,2)

f(0,2)

TRUE

Raj
DATE / /
PAGE

eg: arr = [2, 3, 1, 1] k = 4



Overlapping Subprob
||
DP

• Memoization

let's say constraints : 1 ≤ n ≤ 10^3 , target <= 10^3

```
bool dp[10^3+1][10^3+1]
```

range of index | target sum

traverse (n-1) to 0 in (recursive memo)

• Tabulation

traverse 0 to (n-1)

for each index, check for all possible target
targets ∈ (0, 1, 2, ... k)
BASE

- DP array initialize.
1) Base case
- traverse opp to Memo
- make loops

K/target ∈ (0, 1, 2 ---- K)

k+1

Raj

DATE / /

PAGE

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

dp[0][arr[0]] = true; // Base case

for(ind = 1 to n)
{ for(target = 1 to =k)
  {
    bool notTake = dp[ind-1][target];
    bool take = false
      if(arr[ind] <= target) // then only take
      {
        take = dp[ind-1][target - arr[ind]]
      }
    dp[ind][target] = take || notTake;
  }
}

Return dp[n-1][k];
```

ind = 0

target = 0

k

n

Initialise dp

Space Comp.

In every Iteration we need dp[ind-1][target]

& dp[ind-1][target - arr[ind]

Partition a Subset in 2

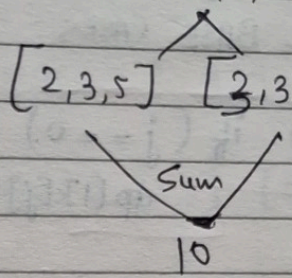
Raj
DATE / /
PAGE

- Q1. Partition the given array of the integer (N size) in 2 equal subset that both subset sum is equal

Same as previous Q.
just replace 'K' with (S/2)

where,
S = Sum of array

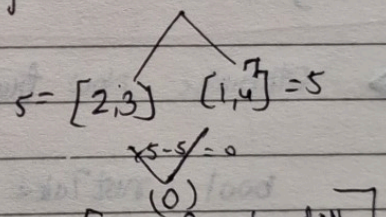
eg: INPUT: [2,3,3,3,4,5]



Q2 Same Q as Q1 just diff is: Both subset have min abs ~~sum~~ difference

Basics of Previous Q.

eg: [1,2,3,4]



① dp[i][j]

i = range of index we can take to create a sum of j

j = targeted sum (x)

BASE CASE 1 : for (i = 0 to n) dp[i][0] = true

My Code

(Code of)
AGAIN → [Tabulation of Subset with sum K]

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

for(i=0 to n)
{ for(j=0 to =k)

→ BASE CASES.

BASE CASES

① if (j==0) // i.e. j / req Sum = 0 then dp = true (attained ✓)
dp[i][j] = true; continue.

② if (i==0) range of taking index is till 0th index i.e only nums[0]
{ if nums[0] == target Sum (i.e j) → TRUE
if (nums[i] == j) dp[i][j] = true
else dp[i][j] = false
continue;
}

→ Otherwise Now traverse via dp[][] (here inclusion/exclusion)

bool notTake = dp[i-1][j] ; Shrink index range
bool take = false
if (nums[i] <= j) // if we are able to take curr value
{
take = dp[i-1][j - nums[i]];
}

dp = true [if any of 'notTake' / 'take' is true]
else dp[i][j] = false;
}

→ finally we have (n-1)th index range to create sum of k, if this is true the we got ans ✓

return dp[n-1][k]

Partition Subset to have Min abs diff.

Raj
DATE / /
PAGE

Q. Given array nums of $2*n$ integers. Partition 'nums' in equal parts (subarr of len n) so that sum of both the sub arrays have minimum abs difference.

Return min diff.

eg: nums = [3, 9, 1, 3]

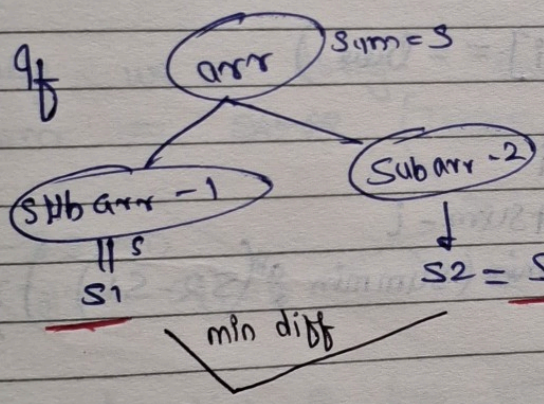
answer = [3,9] [7,3]
12 10

diff = |12-10| = 2

OUTPUT: 2 (min abs diff)

we know from prev Q,
we can check if a sum 'k' can be made by subset of an array or not

APP



- we only need to find 'S1' such that

$S1 - S2 = S1 - S + S1 = \boxed{2S1 - S}$ = mini possible

eg: [3, 2, 7]

S1:	0	2	3	5	7	9	10	12
S2:	12	10	9	7	5	3	2	0

We can take half

Raj

DATE / /

PAGE

dp[n][totSum+1]

{Call subset sum prob ki code.} Previous code

we will get dp filled and in last row we have,

All possible values of Sums (S1) that are ~~pos~~ possible with size of 'n' elements

* We check in last row of dp

```
Summin = INT_MAX
for(i=0 to to totSum)
{
    if(dp[n-1][i] == true)
    {
        S1 = i
        S2 = totSum - i
        Summin = min(Summin, abs(S2-S1));
    }
}
return Summin.;
```

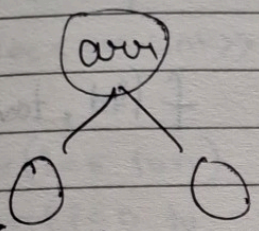
① Count Subset with Sum = K

Raj
DATE / /
PAGE

Same code as of Subset Sum = K just while returning : return notTake + take

② Count Partition With [|Sum1 - Sum2| = 0]

↳ Q. arr



follow these 2 conditions with S1, S2 and return how many possible Partition Possible

[S1 - S2 = D] && [S1 >= S2]
abs diff

Basically we are doing count subsets with sum = ~~S1 - S2~~ $\left(\frac{\text{TotSum} + \text{diff}}{2}\right)$

S1 - S2 = D
S1 - (TotSum - S1) = D
S1

we do first Q. but with modified sum

Count Partition with Sum = $\frac{(totSum - Diff)}{2} = tar$

S1 - S2 = D

Raj
DATE / /
PAGE

Modified Q:

Since Both are Zero so Pick | not Pick won't change anything

Base Case: ind = 0

- tar = 0 & arr[0] = 0 → Pick | not Pick] ②
- arr[0] == tar || tar = 0 ①
  - Pick last to attain target
  - attained

• Memo $(n-1)^{th}$ index $\longrightarrow 0^{th}$ $f(ind, tar) \rightleftharpoons dp[ind][tar]$

n-1 (tot-D)/2

```
int f( ind , tar , arr , dp)
{
   //Base case
    if (ind == 0)
    {
*     if (tar == 0 && arr[0] == 0 ) return 2;
      if (tar == arr[0] || tar == 0)   return 1; //only take
      else return 0;
    }

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

    int nottake = f( ind-1 , tar , arr , dp)
    int take = 0;
    if (arr[ind] <= tar)  take = f (ind-1, tar-arr[ind], arr, dp)

    return dp[ind][tar] = (nottake + take) % MOD
}
```

Base Case

Tabulation

Raj
DATE / /
PAGE

```
{  Check target validity of biring integer
   tar = (Tsum + D)/2;
   if ((Tsum + D) % 2 != 0)  return 0;

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

   for(i=0 ton)
   { for(j=0 to =tar)
   {   //Base Case
       if (i==0)
       {
           if (j==0 || arr[0] == j) {dp[i][j] = 1; continue }
           if (j==0 && arr[0] ==0) { dp ... =2; continue }
           dp[i][j] = 0 ; continue
       }

       // Inclusion - exclusion
       int Nta = dp[i-1][j]
       int t = 0;
       if (arr[i] <= j)   t = dp[i-1][j-arr[i]];

       dp[i][j] = t + nt

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

Base Case

∞ Supply Pattern

Raj

DATE / /
PAGE

COIN CHANGE

Given coins and a amount, return fewest no. of coins that you need to make up that amount. Return -1 if not possible. (You may have each coin ∞ no. of times.

WHY NOT GREEDY ?

INPUT: [1,2,5] , 11
OUT: 3 (5+5+1)

{9,6,5,1} amt = 11

9x1 + 2x1 = 11 (3 coins) → By Greedy
6x1 + 5x1 = 11 (2 coins) ← Fails

→ There is no uniformity in diff of coin values

So, Try all combinations to form "amount"

RULES

- Express in terms of f(ind, tar)
- Base cases
- Min of all

But we have ∞ supply of each element

NOT TAKE

- Dec Index

TAKE ☆

dec the amount, ind remain same
- Don't change index ↓

++

( )

Use leg instead of INT_MAX in DP — leg have less chance of overflow

Since leg = 10^9, 1

INT_MAX = 2.1 Billion

DATE / / PAGE

• Memo

```
int f(ind, tar, arr, dp)
{
// Base Cases
if (tar == 0) return 0;   // No more coin needed

if (ind == 0)
{
   if (tar % arr[ind] == 0)
        return tar / arr[ind];   // req coins
   return leg
}

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

// Inclusion - exclusion

int nottake = f(ind-1, tar, arr, dp);
int take = INT_MAX;
if (arr[ind] <= tar)
   take = 1 + f(ind, tar-arr[ind], arr, dp)
              ↓ curr coin

return dp[ind][tar] = min(take, nottake);
}
```

• TAB

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

for(i=0 to n) {
for(j=0 to =tar){
// Base Case
① if(i==0)  index = 0 only
   {
      if( j % coins[i] == 0)   → Still can attain target
            dp[i][j] = j / coins[i];
      else dp[i][j] = leg;
      continue;
   }
② if ( j==0 ) → target already attained
   { dp[i][j] = 0;
     continue;
   }
int nt = dp[i-1][j], t = leg;
if(j >= coins[i])
      t = dp[i][j-coins[i]] + 1;   // add 1

dp[i][j] = min(nt, t);
}
}

if (dp[n-1][tar] >= leg) return -1;
else return dp[n-1][tar];
```

TARGET VALUE.

Raj
DATE / /
PAGE

Q How many ways you can assign sign, so you can target

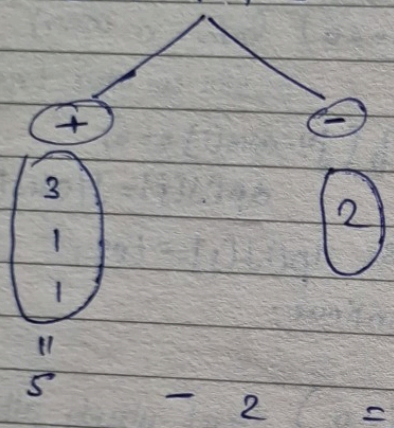
This Q. is similar to question in which we have to "count partitions" : S1 - S2 = 0

eg: arr[] = [1,2,3,1], tar

+ - + +
1 2 3 1 = 3

- + + -
1 2 3 1 = 3

ans = 2 (ways)



Subset: S1 = +ve sign
S2 = -ve sign

We need S1 - S2 = target

So, $S1 = \frac{(totdiff - tar)}{2}$

Answer is 'bt 32 bit integer

1) Use unsigned int instead of int
2) Convert at last to int : static_cast<int>(ans);

Use case

General = int
large val = long long
loop counters = unsigned
count combinat^n / val = unsigned int

Bitwise operat^n = unsigned
-ve value needed = signed

∵ unsigned ≥ 0 to 2x signed max.

COIN CHANGE II

Raj
DATE / /
PAGE

→ Same Q.

But Return no. of combinations that make up that amount
return 0 if not possible

Same as counting subset problem.
here we have ∞ supply of elements.

- **Base Case**

```
if (i==0)
{ if we can attain with 1 element
  if (j % coins[i] == 0)
      dp[i][j] = 1
  else
      dp[i][j] = 0;
```

* Remember in ∞ supply Qs

take = dp[i][j-coins[i]];
Don't reduce the index

continue

}

last

dp[i][j] = take + notTake;

Basic in Count Problems

(∞ Supply of items)

UNBOUNDED KNAPSACK

do 0/1 knapsack first

Raj
DATE / /
PAGE

Q. A Theif goes in house of n elements each have (wt, val)
now Thief can pick each item ∞ times
Return max value stolen

0/1 and Unbounded have 1 diff
That you can pick 1 item
∞ no of time (until
your Bag capacity is over)

INFINITE SUPPLY Q.
* When you take, don't reduce the index

• Basic recursive code

```
f(ind, W)
{
    Base Case
int nt = f(ind-1, W);
int t=INT_MIN
if(wt[ind]<=W)
    t = f(ind, W-wt[ind]);

return max(nt,t)
```

CODE
• Memo, Tab, space opt, ID is similar as of 0/1 knapsack

• Base Case (n-1 to 0)
At 0th index, steal as many of 0th element as you can
if(ind == 0)
return $\frac{W}{wt[0]} \times val[0]$

no. of times you can steal 0th item for Bag Cap of W

0/1 KNAPSACK

★ (take | skip)

Raj
DATE / /
PAGE

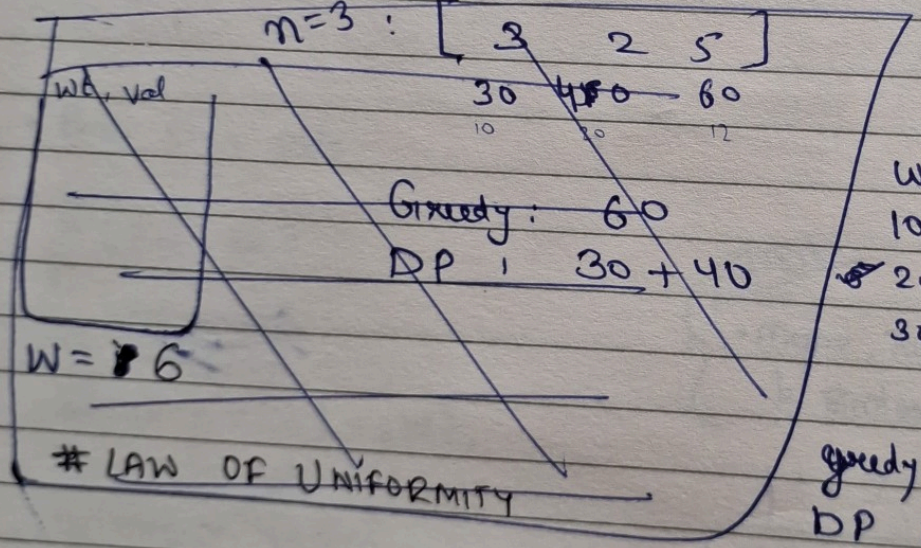
Thief goes in house to steal 'n' items, each item has (wt, val)
He has Bag of wt capacity : W

↳ Steal such that you get max value in bag of wt capacity : W

n = 3
eg: wt : 3 4 5
Val : 30 50 60
W = 8

ans: 90 (30 + 60) (3) (5) → optimal

Q. WHY NOT GREEDY ?



eg:
W = 50

wt	val	val/wt
10	60	6
20	100	5
30	120	4

greedy : 60 + 100 = 160 ✗ (10) (20)
DP : 100 + 120 = 220 ✓ (20) (30)

LAW OF UNIFORMITY
No Uniformity in values and wt given.

When greedy works
- If you can take partial items
- local Best gives global best solution

Recurr : (n-1) to '0' So Base case will be at '0'

determin dp[ ][ ] ← dp[N][wt+1]

STEPS
- Express in terms of Index (ind, wt_bag)
- Expore all possibilities = Pick and Not Pick
- take max of all possibilities

• Basic Recursion Code

```
f(int, W)
{
    Base Case
    nottake = f(ind-1, W)
    take = INT_MIN;
    if (wt[ind] <= W)
        t = f(ind-1, W-wt[ind]);

    return max(take, nottake)
}
```

• Base Cases

at ind = 0
if we still can take it = Return wt[0]
if can't take = return 0

```
if (ind == 0)
{ if (wt[0] <= W) return wt[0]
    else return 0;
```

Time and Space complexity.

① Memo
TC = O(N×W)
SC = O(N×W) + O(N)
recursion stack.
can be removed by Tabulation

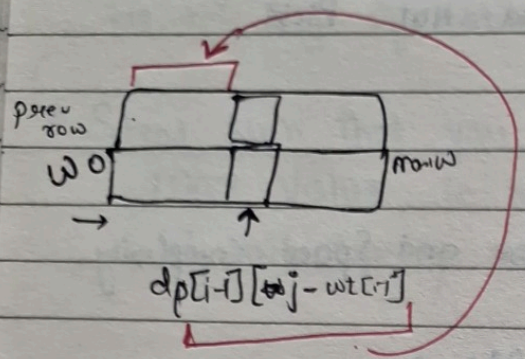
• Tab

You will use 2 for loops and yeah, you will do it

But lets do 1 row optimization

we were still using 2 vectors

- 1D optimization of this (refer DP:19 32:00)



we are filling left n R

- But you can Even fill Right to left ①
(since its depended on prev row)

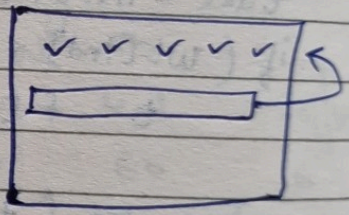
② Don't store in curr then at last update to prev = curr
↳ instead directly store new val in prev

```
for(i=0 ton)
{
  for( j = W to = 0)
  {
    Same Base Case ✓

    int nt = prev[j]
    int t = INT_MIN;
    if (wt[i] <= j) t = prev[j-wt[i]];

    prev[j] = max(nt, t)
  }
}
return prev[W];
```

no need for curr

Space opt of TAB

dp[ ][ ] → W+1



You need prev row for curr row in each iteration

So,
make vector<int> prev (W+1)

So, instead of dp use curr row to store and

prev = curr

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

* dp[i][j] = curr[j]

Raj

DATE / /

PAGE

Space optimed

```
{ vector<int> prev(W+1, 0); }
{ vector<int> curr(W+1, 0); }
```

```
prev[j] = wt[0]
```

Method	Tc	Sc
Memo	N×W	N×W + O(N)
Tab	N×W	N×W
Space opti	N×W	2×W
1D	N×W	W

```
curr[j] = max(nt, t)
}
prev = curr
}
```

Tabulation

```
vector<int>(W+1, 0);
vector<vector<int>> dp(n, [ ]);

for(i = 0 to n)
{ for(j = 0 to = W)
  {
    // Base Case
    if(i == 0)
    { if(wt[i] <= w) dp[i][j] = wt[0];
      else dp[i][j] = 0;
    }
    continue

    // Inclusion - exclusion
    int nt = dp[i-1][j]   (prev)
    int t = INT_MIN;

    if(wt[i] <= j)
      t = dp[i-1][j-wt[i]];   (prev)

    dp[i][j] = max(t, nt);
  }
}
return dp[n-1][W];   (prev)
```

replace prev to curr

DP on STRINGS..

Raj
Page:
Date: / /

commonly used in → comparision, replace/edits

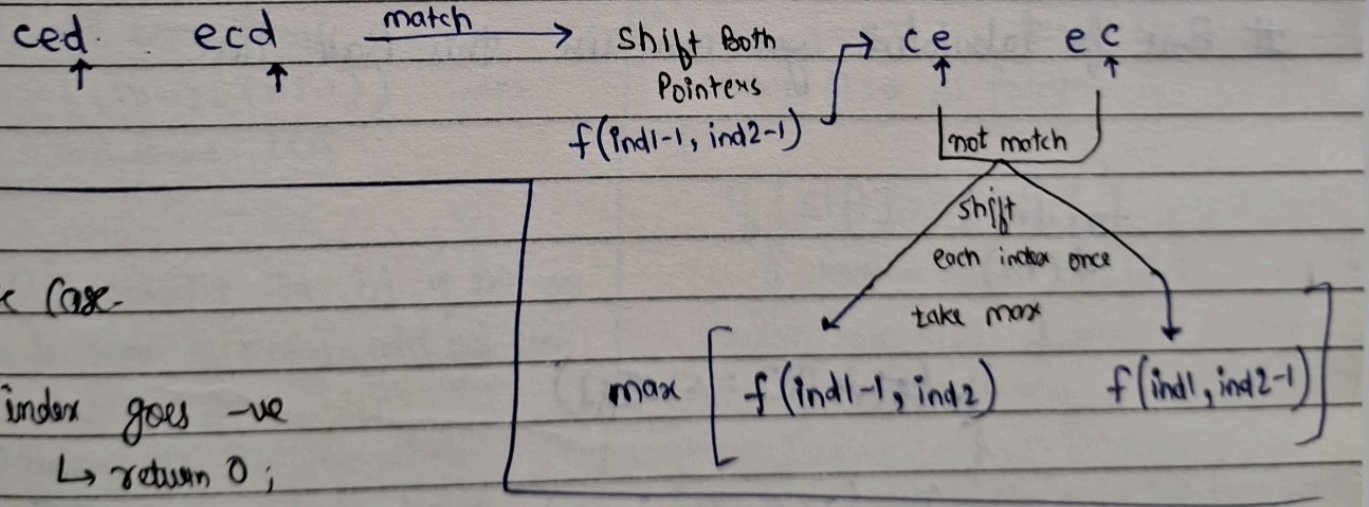
1 Longest common Subsequence in 2 strings

INPUT : "adebc", "dcadb"
OUTPUT : 3 (adb)

STEPS

- express in Index : f(ind1, ind2)
- explore all possibilities
- Take best common len among them

VIS



Base Case

when index goes -ve
↳ return 0;

```
if (ind1 < 0 || ind2 < 0) return 0;
```

Raj
Page:
Date:

```
f(ind1, ind2, str1, str2, dp)
{
    if(ind1<0 || ind2<0) return 0;
    if(dp != -1) return dp[ind1][ind2];
    if(str1[ind1] == str2[ind2])
        return dp[ind1][ind2] = 1 + f(ind1-1, ind2-1);
}
```

reduce both index

add 1 for current (inclusion)

```
    return dp[ind1][ind2] = max(f(ind1-1, ind2),
                                f(ind1, ind2-1));
```

else reduce each and take max common length

But for tabulation you can use this Base case

```
for(n1)
 f(n2)
  {
    if(str1[i] = str2[j])
    {
      if(i>0 && j>0) dp[i][j] = 1 + dp[i-1][j-1]
      else dp[i][j] = 1;
    }
else {{done}}
```

only if you can do (i-1) - means (i>0)
Other wise = 0

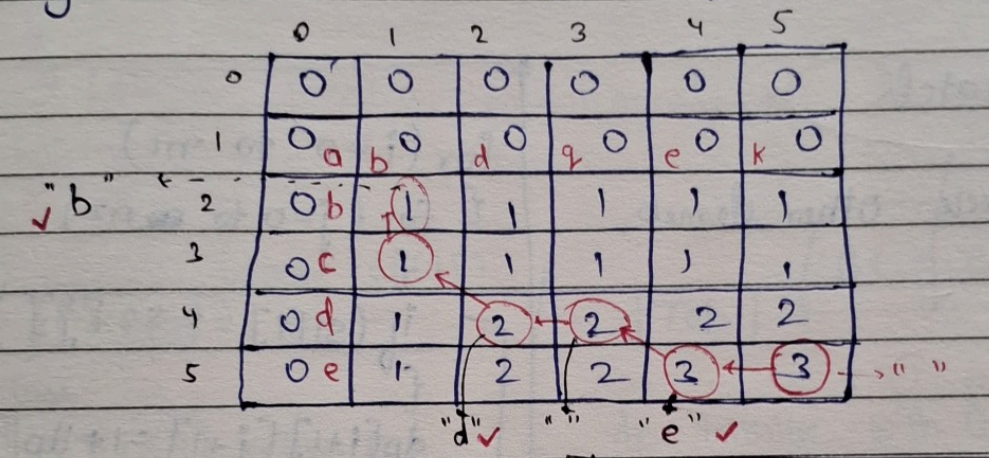
at end ⟶ return dp[n1][n2]

PRINT LCA.

Q Print the largest common string

- Currently dp[i][j] stores max common length b/t str1 (till ith index) and str2 (till jth index)

eg: "abcde" "bdgek"



We will back track from last: ((n1-1), (n2-1)) index and make LCA

DP

1) If currently the i,j of DP arr is same of string, add the str.
   ↳ reduce Both i,j
2) traverse the that side which gives max value on dp table (i.e max LCA)

At end reverse the string

Code

```
i=n1, j=n2 , String ans;
while(i>0 && j>0)
{
  if(s1[i]==s2[j])
  { ans = ans + s1[i];
  // reduce Both index if equal
    i--, j--
  }
// otherwise traverse to max value side (max LCA)
  else if (dp[i-1][j] >= dp[i][j-1])
  {
    i--; // we go to greate direction
  }
  else {j--}
}
return reverse of (ans)
```

(not subsequence)

Page: Raj
Date: / /

Q Longest common SUBSTRING

in this we just need consecutive substring

everything is same as of prev Q. (subsequence)

- Only Diff is how we deal when char dont matth

Mismatch

don't check other demes

return 0;

```
for(i=0 to n1)
{ for(j=0 to n2)
  {
  if(s1[i]==s2[j])
  {
  dp[i+1][j+1] = 1+dp[i][j]
  maxlen = max(maxlen, dp[i+1][j+1])
  }
  else
  {
  dp[i+1][j+1] = 0
  }
```

```
if(s1[i]==s2[j])
{
  if(i==0 || j==0) dp[i][j]=1;
  else dp[
  {
  dp[i][j] = 1+dp[i-1][j-1]
  }
  maxlen = max(maxlen, dp[i][j]);
}
else
  { dp[i][j] = 0 → directly put zero
```

Longest PALINDROMIC SUBSEQUENCES (LPS)

Page: Date: Raj

CATCH

eg: "bbabcbcab"

OUTPUT: "babcbab"

Q. S1 = "bbabcbcab"

rev(S1) → S2 = "bacbcbabb"

LPS = longest common subsequence b/t S1 and rev of S1

(longest Palindromic Subseq. is)

Memo

initialize dp[][] with -1

```
// Base Case
if(i<0 || j<0) return 0;
if(dp[i][j] != -1) return dp[i][j];

if(s1[i] == s2[j])
{ dp[i][j] = 1 + f(i-1, j-1);
}
else
{
return dp[i][j] = max( f(i-1, j),
                       f(i, j-1) );
}
```

BC: -1 42

Tab

BC = zero 42

dp initialise with 0 (n x n)

```
for(i=0 to n)
{for(j=0 to n)
{
  if(s1[i] == s2[j])
  { if(i>0 && j>0)
      dp[i][j] = 1 + d[i-1][j-1] + dp[i][j-1];
    else // Base Case  i==0 || j==0
      dp[i][j] = 1
  }
  // if valid then only do i-1 | j-1
else { int o1 = (i>0)? dp[i-1][j] : 0
       int o2 = (j>0)? dp[i][j-1] : 0
  dp[i][j] = max(o1, o2)
}
}
return dp[n-1][n-1];
```

MIN INSERTION TO MAKE PALINDROMIC STRING

Q min insertion req to make a string palindromic

eg: "a b c a a"
a b

a a b c b a a

1) Keep palindrome part intact
2) we take the remaining part and insert that in reverse to make a palindrom

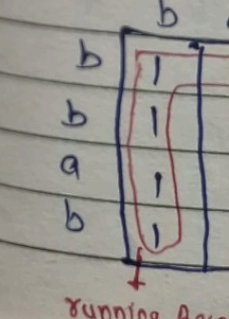
eg: a b c a a

a a a = palindrom
bc = non-pali
↳ rev = cb
↓
a bc a cb a = pali ✓

a c a = pali
ba = non pali
↓
rev = ab
↓
a ba a c ab a ≠ pali.

Min Insertions = (Total Length) − Length of longest Palindromic Subsequence

CODE ON NEXT TO NXT PAGE →



Shifting Index

↳ whenever you compare Str1 instead of 'i' use (i-1) since you start with i=1 (not 0)

Raj
Page:
Date: / /

Filling dp[] with 0 and 1 Base Indexing in LPS

'0' Based Index

```
for(i=0 to n)
{ for (j=0 to n)
  {
    if (s1[i] == s2[j]
    {
      if(i>0 && j>0)
      {
        dp[i][j] = 1+dp[i-1][j-1];
      }
      else i==0 || j==0
        dp[i][j] = 1;
    }
    else
    {
      int o1 = (i>0) ? dp[i-1][j] : 0
      int o2 = (j>0) ? dp[i][j-1] : 0
      dp[i][j] = max(o1, o2);
    }
  }
}
```

'1' Based Index ~ Faster

```
for (int i = 1 to =n)   ] go 1 to =n
{ for(int j = 1 to =n)
  {
    if(s[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]);
    }
  }
}
```

here when i==0 || j==0
dp[i][j] = 0

} You do this in 0 Based Indexing

But it's already done } here.

s1 = "bbab" (0 1 2 3 / 1 2 3 4) s2 = "bab" (0 1 2 / 1 2 3)

		b	a	b
	b	1	1	1
	b	1	1	2
	a	1	2	2
	b	1	2	3

running Base case

SAME

		0	b (1)	a (2)	b (3)
0		0	0	0	0
1	b	0	1	1	1
2	b	0	1	1	2
3	a	0	1	2	2
4	b	0	1	2	3

Initialized

Min ~~Insert/Del~~ to make ~~String~~ Palindrome

SHORTEST COMMON SUPERSEQUENCE

- Supersequence: Make a string such that ~~string~~ S1 and S2 Both are in it

Q→ So we need that String

eg: S1: "abac" S2: "cab" Ans: cabac

So,
LCS = "ab"

When we Backtrack in this dp array we will be getting our SCS

		0	c	a	b	
0		end 0	0	0	0	"c"
1	a	0	0	1	1	"a"
2	b	0	0	1	2	"b"
3	a	0	0	1	2	"a"
4	c	0	1	1	2	'c' it was ignored

HOW?

TRY TO DO = PRINT LCS

① Whichever char you ignore / move away / gives lesser value in dp[][] → add that in sequence

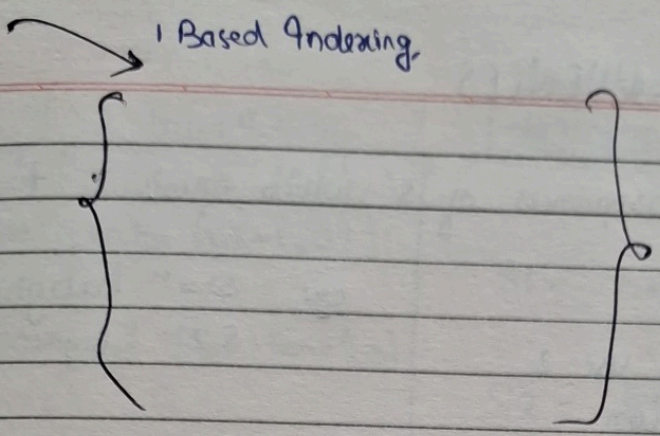
② Whenever you got equal char and you move diagonal add that common char once.

③ Add remaining char until i>0 && j>0

④ Reverse the String

LCS

LCS Code



Raj
Page:
Date: / /

//Building SCS

```
int i=n1, j=n2;
String ans = "";

While (i>0 && j>0)
{
  if (str1[i]==str2[j])
    {
      ans += str1[i-1];      i--; j--;
    }
  else if (dp[i-1][j] > dp[i][j-1])
  {
  } ans += str1[i-1]; i--;

  else   ans += str2[j-1]; j--;
}

Add Remaining
While (i>0) { ans += str1[i-1]; i--; }
While (j>0) { ans += str2[j-1]; j--; }
Reverse the String
reverse(ans);
return ans
```

DISTINCT SUBSEQUENCES

Number of distinct subsequences of 's' which equals 't'

eg: S1 = "babgbag"
S2 = "bag"

OUTPUT: 5

- In case of 2 string we use 2 parameter in recursion

n1-1 ↘ ↙ n2-1

f(i, j)

here it means No. of diff subseq in S1 (till i^{th} index) of S2 (till j^{th} index)

Steps: Recurr (n-1 to 0)

We will take element from S2 and if matches then we move in Both

Otherwise, we search via recurr only in S1

Recursion

```
f(i, j)
{ if(j<0) return 1;
  if(i<0) return 0;

  if(s1[i]==s2[j])
      f(i-1, j-1)
+ else
    f(i-1, j);
}
```

Base Case (0-Based Indexing)

- If you finished searching in S1
  ↳ i.e (i<0) → return 0
- If you S2 is over then
  ↳ return 1

Raj
Page:
Date: / /

TAB 1-Based Indexing

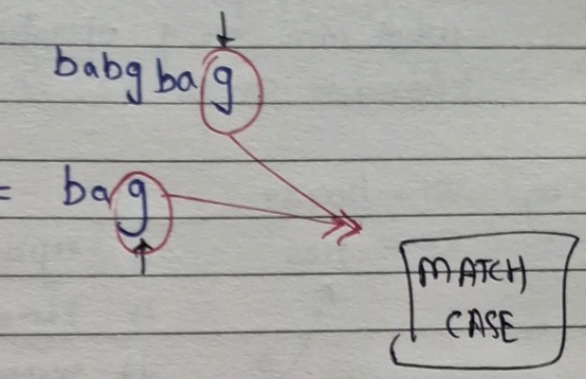
```
dp(n1+1, vector(n2+1, 0));
```

acc^n to Base case (for 1 Based Indx)

all (i==0) = 0
all (j==0) = 1

```
for(i=1 to n1)
{for(j=1 to n2)
{
  if(i==0) dp[i][j]=0; continue
  if(j==0) dp[i][j]=1; continue

  // Selection and traverse
  if(s1[i-1]==s2[j-1])
    dp[i][j]=dp[i-1][j-1] + dp[i-1][j];   // match
  else
    dp[i][j]=dp[i-1][j];
}
}
return dp[n1][n2]
```

DRY RUN



① Reduce Both Index and search

s1 = [babgba] g

s2 = [ba] g

② Only reduce s1 and keep s2 same since we need all occurances

s1 = [bababa] g

s2 = [bag]

Add ① and ② for all distinct matches in s1 & s2

EDIT DISTANCE

Page: Raj
Date: # S1 Char

Operation :-
1) INSERT
2) DELETE
3) REPLACE

You have 3 operations to do with (str1), what min no of operation in which you can convert S1 to S2

eg: S1 = horse
S2 = ros

1) replace : hr → r
2) remove → r (i = 2)
3) remove → e

ros

Output : 3

f(i,j) = min operations to convert (0 to i) of S1 into (0 to j) of S2

NOT MATCH CASE

① Insertion

horses
ros
insert

• Insert and reduce 'j' 'i' remains same

f(i, j-1)

② Delete

horse
ros

• Delete and shift 'i' 'j' remain same

f(i-1, j)

③ Replace

horse
ros

• Replace and reduce both since they matched

f(i-1, j-1)

min of all

Raj
Page:
Date: / /

0 Boxed Indexing

Base Cases

① S1 is exhausted — f(-1, j)

i

horse
-1 0 1 2 3 4

ros
j

f(-1, j)

So it says : no of operation req. to make a empty string into a string (s2) till 0 to j^{th} index

i.e we need to insert all (0 to j) chars : (j+1) char

return (j+1) operations

② S2 is exhausted — f(i, -1)

i

horse
-1 0 1 2

ros
j

f(i, -1) says : min operations req to convert S1 (0 to i^{th} index) into a null string

i.e remove a char of S1 (0 to i)

return (i+1) operations

• Recursion :

```
{ Base case : if(i<0) return j+1;   if(j<0) return i+1;

    if (s1[i] == s2[j])  f(i-1, j-1);   matching case
    else                 add 1
    {   int insertOP = 1 + f(i, j-1)
        int delOP    = 1 + f(i-1, j)        } Not matching case operat"s
        int replaceOP = 1 + f(i-1, j-1)
    }

min of all   return min(insertOP, delOP, replaceOP);
}
```

1 Based : So recurr" me -1 idhar '0' ban jata h

Page: Raj
Date: / /

↑

- TAB

Base case changes a bit

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

BASE CASE

```
if (j==0) dp[i][j] = i
if (i==0) dp[i][j] = j;
```

Since its 1 Based indexing

TRAVERSE

```
if (s1[i-1] == s2[j-1]
    dp[i][j] = dp(i-1)[j-1];
```

operation of not match case

```
else {
   insertOP = 1+ dp[i][j-1];
   del OP =  1+ dp[j-1][j]
   Replace OP = 1+ dp[i-1][j-1];

dp[i][j] = min(insert, del, replace);
}

)

return dp[n,m]
```

- MEMO

WILDCARD MATCHING

Raj
Page:
Date: / /

eg: S1 = ?ay
S2 = "ray" ✓

? = matches w single char
* = match w seq of len 0 or more

matches ✓ (since ? can be a single char

eg: S1 = "ab*cd" ✓
S2 = "abdefcd"

matches (* = def) *seq.

S1 = "**abcd" ✓
S2 = abcd

* = "" = NULL

S1 = "ab?d" ✗
S2 = "abcc"

?=c
but c=d ✗ not matd

Q. Check if S1 and S2 matches ?

- f(i, j) = is S1(0 to i) matches with S2(0 to j)

1. Match^n case

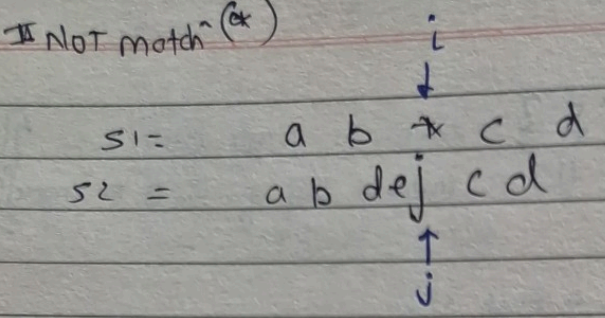
↳ (S1[i] == S2[j] or S1[i] == '?') ← ? matches with any single char

⇓

Shrink both indices

Page: Raj
Date: / /

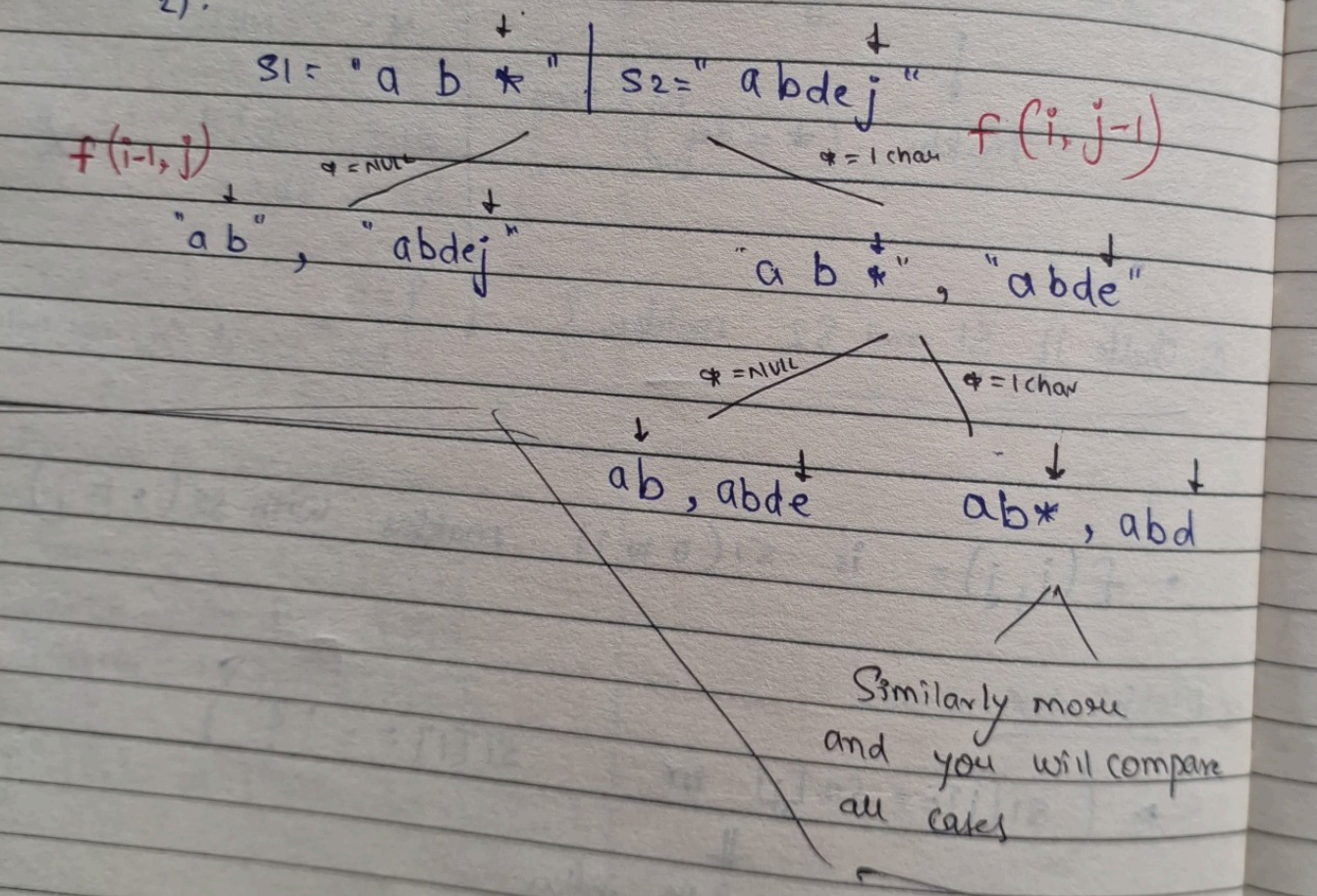
If NOT match (*)



- Lets match 0, 1, 2, 3 ..... with *

1. Lets say * mean NULL

2).



Base

only S

if only

then o

Since

ex

Both e

- if (i<

- if(i<0

Raj
Page:
Date: / /

Base Case

only S1 is exhausted

if only 1 string is exhausted (S1) then off they do not match

Since Both have to be exhausted to match

Both exhausted

- if(i<0 && j<0) return True

- if(i<0 && j>=0) return False;

S2 is exhausted

means there are all '*' in S1, that's why we traversed S2 in hope that * will match

But since it's all '*' in S1 then it will match

If there are all '*' in S1
↳ TRUE

else → FALSE

→ 0-Based Index

Recurr BASE CASES.

1) Both ended together
if (i<0 && j<0) TRUE

2) S1 ended but not S2
if (i<0 && j>=0) FALSE

3) S2 ended not S1
if (j<0 && i>=0)
↳ check if S1 is all star or NOT
for (k=0 to i) check 0 to i
{ if (S1[k] != '*') FALSE
}
TRUE
}

• MATCHING CASE
if (S1[i]==S2[j] || S1[i]=='?')
f(i-1, j-1); Shrink Both

• NOT MATCHING WITH '*'
if (S1[i]=='*')
return { f(i-1, j), f(i, j-1) }
NULL, 1 char → if any is True ↳ TRUE

• If nothing matche
return false

1-Based → TAB

Page: Raj
Date: / /

Base Cases
dp[0][0] = true; ← is like
// S2 is empty, S1 must be all '*'
for (j=1; j<=m; j++)
if (S1[j-1]=='*')
{ dp[0][j] = dp[0][j-1]
}

for (i=1 to n)
{ for (j=1 to m)
{ matching
if (S1[i-1]==S2[j-1]
|| S1[i-1]=='?')
{
dp[i][j] = dp[i-1][j-1];
}
'*'
else if (S1[i-1]=='*')
{
dp[i][j] = (dp[i-1][j] ||
dp[i][j-1])
else {
dp[i][j] = false;
}
}
return dp[n][m];

→ Space opt. is Important

DP on STOCKS

Raj
Page:
Date: / /

Q. Best time to Buy and sell stock

- All we need is min price on ~~R~~ to buy stock

(Index) day:	0	1	2	3	4	5
(Value) cost:	7	1	5	3	6	4

- Iterate one arr (from i=1) keep track of min on left to mark our ~~sell~~ buying pair

Sell - buy = Profit (maximize this)

Min price is first (Buying Price

```
int minLeft = arr[0] , profit = -1;
Iterate for Selling Price on from 2nd day
for( i=1 to n)
{
    if( arr[i] - minLeft > Profit)
        profit = arr[i] - minLeft ;

    Update min on left side
        minLeft = min (minLeft, arr[i]);
}
if (profit == -1) return 0;
else return profit;
```

Buy & Sell Stock II

In this Part, you can buy-sell as many times as you want till array allow

You can Buy again until you have sold previous one

days (index) : 0 1 2 3 4 5
prices (val) : [7 1 5 3 6 4]

Recur = TRY ALL POSSIBLE WAY

In plain eng: Take / Not Take type

VIS

- f(i, j) = Start on ith day with j(0|1) = what is the max profit

1. Express in (index)
   ↳ f(ind, Buy)
   (1) Buy or Not (0)
2. Explore possibilities
3. Take max of profit
4. Base case

APP

- Buy == 1 (you CAN Buy)

I
1) minus the price of that index
2) Now recurr call of Buy = 0 (Since now you can't Buy)

II Don't Buy : just move to next day

YOU CAN'T Buy (Sell)

- Not Buy (Buy == 0)

1) Add price to ans
2) go to next day with Buy == 1

Sell < Yes / NO (Both Options)

→ 0 to (n-1)

0-Based Index

Raj
Page:
Date: / /

Recursion

0th index 1

```
f(ind, bool buy)
{  Arr is Over : Profit = 0
   if(ind == n) return 0;
   If you can Buy
if(buy)
{  1) You will Buy ; 2) Won't Buy
profit = max( f(ind+1,0) - arr[ind];
              , f(ind+1,1) + 0) ;
}
else { You can sell }      Sold
   profit = max( f(ind+1, 1) + arr[ind])
                , f(ind+1, 0)) } ;
return profit.
```

Base Case

- If you Bought the Stock But ari is over (But it won't matter)

```
if(ind == n) return 0;
        (zero profit)
```

at nth day even if it's Buy or Sell
↳ your profit is zero

TAB. 1-Based it Be opp of Recurr (n-1 to 0)

```
dp[n+1, 2] = {0}
// Base Case
dp[n][0] = 0 = dp[n][1]

for( i = n-1 to = 0)
{
  for( Buy = 0 to = 1)
  {
    long profit = 0
```

```
if(Buy)
{ profit = max(dp[i+1][0] - arr[i])
               , dp[i+1][1] + 0);
}
else {
  profit = max(dp[i+1][i] + arr[i],
               dp[i+1][0]);
}
//dp[i][Buy] = max of profit
  dp[i]
}
}
return dp[0][1]
```

ahead = prev

Space opt.

```
int nextBuy = 0      dp[i+1][1]
int nextSell = 0     dp[i+1][0]
int currSell = 0;    // dp[i][0]
int currBuy = 0;     // dp[i][1]
```

currSell → dp[ind][0] currBuy → dp[ind][1]

for each dp[ind][Buy]

we only need (dp[ind+1][0] → nextSell
dp[ind+1][1] → nextBuy)

Use 4 variables

for (n-1 to =0)

j=Buy {

j=1 currBuy = max(nextSell - arr[i] , nextBuy); ← dp[i+1][1]

j=0 currSell = max(nextBuy + arr[i] , nextSell);

nextBuy + curr price if selling

nextBuy = currBuy ;
nextSell = currSell ;
}

return currBuy as we start with buy permissions on day 0

currBuy = max(

- nextSell - arr[i] → go to i+1 index with selling option

)

nextBuy : go to i+1, without buying with option of buy next day

we need this one for (n-2)

n-1

n-2 → n-1 (start)

curr	nextSell	0
curr	nextBuy	1

→ need dp arr [n+1] [2] [cap+1]

2 Transaction - Buy Sell Stock.

Page: 0 1 Naj
Date: / /

Q. In this Q. you can only do 2 transaction for max profit
(Rest Same as prev Q.)

we will just introduce another limiting variable
Say: CAP.
⟹ This ensure we are within given transaction limit.

prices = [3, 3, 5, 0, 0, 3, 1, 4]
0 1 2 3 4 5 6 7

Transaction = Buy + Sell

BASE CASE

1) Arr over ⇒ Profit = 0
2) Transact^n limit reach ⇒ Profit = 0

```
f(ind, Buy, cap)
  if(cap == 0) return 0;
  if(ind == n) return 0;

  if(dp[ind][buy][cap] != -1) return dp[ind][Buy][cap];

  if(Buy)
  {
    profit = max(-arr[ind] + dp[ind+1][0][cap], dp[ind+1][1][cap])
  }
  else          Only one Transact when you did Both (Buy+Sell) for Stock
  {
    profit = max(+arr[ind] + dp[ind+1][1][cap-1], dp[ind+1][0][cap])
  }

  return dp[ind][Buy][cap] = profit;
```

TAB

1) Base case (JUST SO YOU KNOW)

1) If cap == 0 and & Buy can be anything

2) If ind == n cap and buy can be anything

cap = 0

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

ind = n

② for (Buy = 0 to = 1)
{ for (cap = 0 to cap)
{
dp [n] [Buy] [c] = 0

{ Since all are '0' and we initialise dp arr with 0 so, no need to write These Base Cases in code (for this q) }

TAB CODE

Q [Cap = 2]

Page: Raj
Date: / /

vec<vec<vec<int>>> dp(n+1, vec<vec<int>>(2, vec<int>(3,0)));

n elements

Buy: 0 | 1

Cap: 0 | 1 | 2 → Transactions

```
for(ind = n-1 to =0)
{ for (buy=0 to =1)
  { for (cap=0 to =2)
    {
        BASE CASE
      if(cap==0) {dp[ind][buy][cap]=0 ; continue ;}

      SAME SHIT

    }
  }
}
return dp[0][1][cap];
```

cap → 2

DP on LIS { Longest Increasing Subsequence }

Raj
Page:
Date: / /

LIS : eg : [1, 3, 2, 5] → Subseq
[1, 2, 3, 5] → Inc^n Subseq

STEPS

- Expres in Index. : f(ind, prev) → Index of previously taken element

So,
f(3,0) = Length of LIS, start^n from 3rd index whose previously taken element was 0

- Recurn Code

```
f(ind, prev_ind)          // ind: 0 ↓, prev_ind: -1 ↓
{
    // Base case: arr over, length = 0
    if (ind == n) return 0;

    // NOT take                          → prev remain same
    len = 0 + f(ind+1, prev_ind);

    // take condition
    if( prev_ind == -1 || arr[ind] > arr[prev_ind] )
    {
        len = max( len, f(ind+1, ind) );
                                  // ↳ update prev
    }

return len;
```

Page: Raj
Date: / /

0 to n-1

- MEMO dp[n][n+1] But here we can't store -1 as Index.

COORDINATE CHANGE

old :	-1	0	1	2	. . .	n-1	
new :	0	1	2	3	. . . .	n	
	←					→	n+1

2nd para of dp (i.e: prev) will be in (prev+1) (+1 state)

```
f(ind, prev_ind, arr, n, dp)
{   // Base.
    if(ind == n) return 0;
    // Already cal.
    if(dp[ind][prev_ind+1] != -1) return dp[ind][prev_ind];
    // Not take
    int len = 0 + f(ind+1, prev_ind)
    // take
    if(prev_ind == -1 || arr[ind] > arr[prev_ind])
    {
        len = max(len, 1 + f(ind+1, ind))
    }
    return dp[ind][prev+1] = len
}
```

↳ +1 length when you take

return dp[0][0]
(-1+1)

Rules
2)
3)
TAB

PRINT LIS

{ TC : $O(N^2)$, SC : $O(N^2)$. }

Raj
Page:
Date: / /

TAB

Rules

1) Base Case (ind == n) → dp = 0 since we already init dp[][] with zero = NO NEED of BC

2) Write changing parameter in opposite manner

ind : (n-1) to 0
prev_ind : (n-1) to -1
(ind-1) (can't be more than ind

3) Copy the recurrance & follow the coordinate shift

TAB

```
dp[n+1][n+1] = {0}

for( i = n-1 to =0)
{ for( prev = i-1   to   = -1)
  {
    int nt = dp[i-1][prev+1]
    int t = 0
    if( prev == -1 || nums[i] > nums[prev])
      {
        t = dp[i+1][i+1]+1
      }
    dp[ind][prev+1] = max(t, nt)
  }
}

return dp[0][0];
```

Due to coordinate shift → second para is always in (+1) state

when you ~~add~~ Take always add 1 in length

TC: O(n log n)

Page: Raj
Date: / /

BEST & OPTIMIZED (Better) METHOD OF LIS

5	4	11	1	16	8

dp[n]

• dp[i] = longest inc^n subseq that at index 'i'

	0	1	2	3	4	5
	5	4	11	1	16	8
possible IS = (lengths)	1	1	2	1	3	2
	{5}	{4}	{5,11}	{1}	{5,11,16}	{1,8}
			{4,11}		{4,11,16} etc	{3,8}
			max len=2		{16} etc	{4,8} etc
					max len=3	{8}
						max len=2

Longest IS: 1/

∴ LIS = max (dp[i])

for, [i = 0 to (n-1)]

ALGO

• Even if there is no smaller element behind ith element then too, dp[i] = 1 (itself)

• 5 — Can it be with 16? → YES → 16 (1+1) → 2 new max

• 4 — YES → 16 (1+1=2)

{4,16} = 2 ✓

dp:

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

• 11 — YES → 16 (1)

2+1 = 3 new max → final max ✓

Can 1 be with 16

↓

- 1 YES 16

1+1 ≠ 2 not a new max

BETTER method for Space

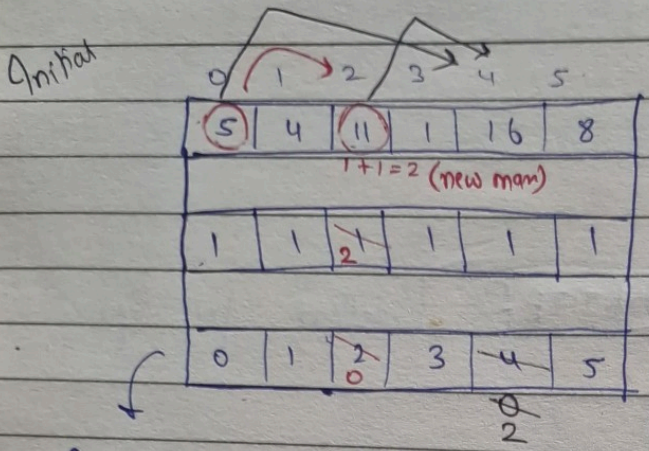
CODE

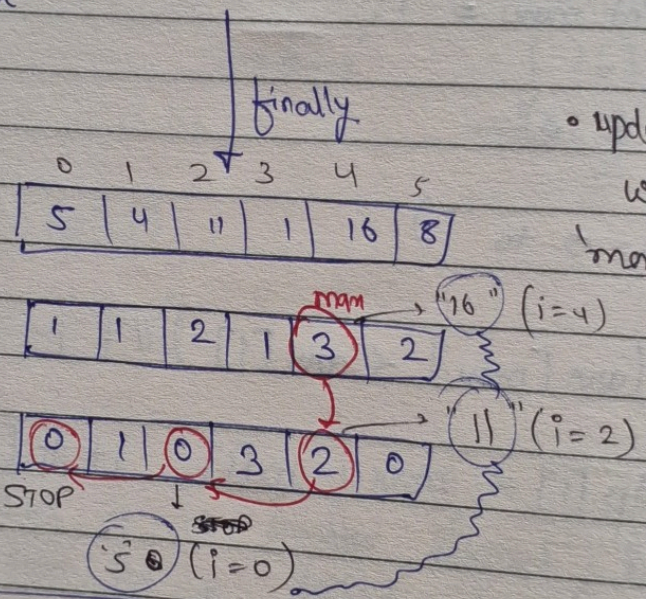
```
dp[n] = {1};
int maxi = 1;

for(i=0 to n)
{
  for(prev = 0 to i)
  {
    if (arr[prev] < arr[i]) // if prev can be with curr.
    {
      dp[i] = max(dp[i], 1+dp[prev]);
    }
  }
  maxi = max(maxi, dp[i]);
}

return maxi.
```

→ $O(N^2)$

Raj
Page:
Date: / /

⇒ PRINTING LIS (Use extra hash map for in prev method)





- update in hash only when we get new 'max' on dp

LIS ← 5 11 16

after reversal

Raj

Page:
Date: / /

```
vector<int> dp(n,1) , hash(n);
int maxi = 1;
int lastind = 0

for( i=0 to n)
{ ~~for~~ hash[i] = i;
  for (prev = 0 to i)
  {                    less than curr  &&   got new max on dp
    if (arr[prev] < arr[i] && 1+dp[prev] > dp[i])
    {
      dp[i] = 1+dp[prev]      update dp
      hash[i] = prev;         update hash array
    }
  }
  if( dp[i] > maxi )
  { maxi = dp[i];
    lastind = i               keep 'lastind' at max of dp
  }
}
ans.push(arr[lastind]);
while ( hash[lastind] != lastind)   start^n from max on dp
{
  lastind = hash[lastind]
  ans.push( arr[lastind]);
}
reverse(ans)
```

Put it in a loop and traverse until → until you come to ind 0.

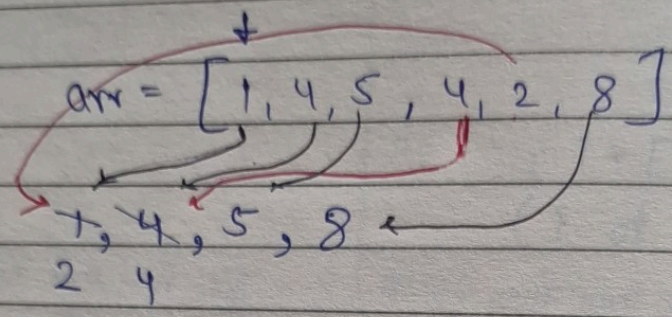
At end reverse

(Best) TC : O(n log n)

LIS by Binary Search

This approach is only to get LIS (length term).

eg: arr = [1, 4, 5, 4, 2, 8]



we will replace each coming element to a "suitable place" by using Binary Search

↓

this is how we get = length of LIS

Lower_bound function in C++

↳ lower-bound ( ) gives first element (index) greater than x

SYNTAX

lower_bound (v.begin(), v.end(), x)

↓

give first element ≥ x

{1, 2, 3, 4}
0 1 2

x = 2

↳ give = 3 (i=2)

Raj
Page:
Date: / /

CODE

```
int len = 0;
vector<int> temp;
temp.push_back(arr[0])  push first element
Now Itivate over array
for(i=1 to n)
{       directly add if it satisfy LIS
    if( arr[i] > temp.back())
    {
        temp.push(arr[i])
        len++;
    }
    else // Use lower bound and replace
    {
get lower_bound   int ind = lower_bound(temp.begin(), temp.end(), arr[i]);
replace in temp   temp[ind] = arr[i];
    }
}
}
return len;
```

NOTE: whatever is generated is for getting length of LIS not the actual LIS

TC: O(n x log n)
SC: O(N)

LARGEST DIVISIBLE SUBSET

Raj
Page:
Date: / /

SUB SEQUENCE	SUB - SET
[1] [16] 7 [8] 4	1 16 7 8 4
{1,16,8} = sub seq ✓	{1,16,8} = subset ✓
{1,8,16} = sub seq ✗	{1,8,16} = subset ✓
• Order must be same	• Order can vary

Divisible Subset ?

SAY
We have, {4, 16, 8}

(4,8)
(8,16)
(16,4)
} Pair of Subset

all these Pairs must be divisible by each other
(can be either way)

STEPS

1) Sort the given Array

[Same code as Print LIS except] → change condition 8: if(arr[i] % arr[prev] == 0)

divisibility

{
}

Raj
Page:
Date: / /

DP

given: [1, 16, 7, 8, 4] —sort→ [1, 4, 7, 8, 16]

now we have → [1, 4, 7, 8, 16]

1 4 8 16

- Since array is sorted and each coming element is divisible by all previous
  Just check divisibility with the element in last of the row

↓

SIMILAR TO picking (print LIS)

LIS Basic Code

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

LONGEST STRING CHAIN

Raj
Page:
Date: / /

In words[], each (word[i-1]) is predecessor of (word[i]). { Pred = Insert 1 letter in word(i-1) to make it word(i) without change order of letters of word(i-1).

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

eg of chain: a, ba, bda ✓
a, ba, adb ✗ (b/a order is not same)

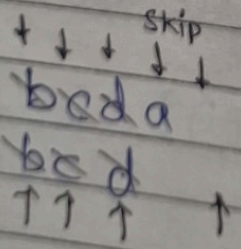
here longest string chain: ["a", "ba", "bca", "bdca"] answer

APP

→ Just check if each word[i-1] and word[i] are have difference of 1 character only or not

- Ordering must be same.

- Code is same as LIS, instead we use a compare function

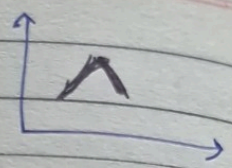


end simultaneous (after 1 skip) ✓

LIS / LDS / (LIS + LDS)

LONGEST BITONIC SUBSEQUENCE

Page: Raj
Date: / /



Bitonic: Series that first inc then dec or just be inc^n or just be dec^n

given: arr = [1, 11, 2, 10, 4, 5, 2, 1]

STEPS

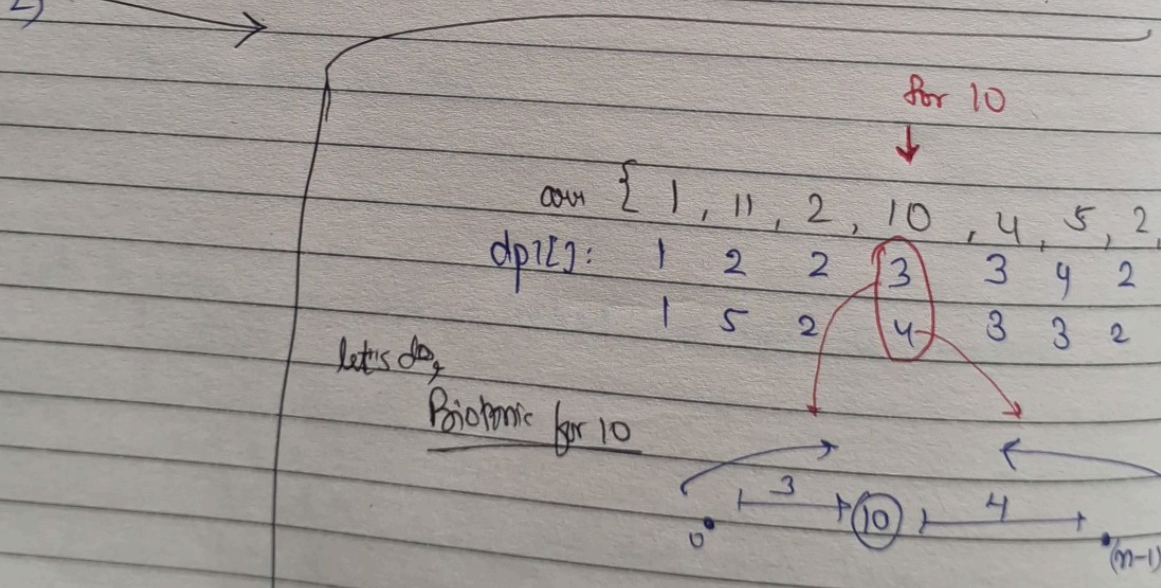
- for LIS we used to make a dp[i] which denotes len of LIS till index 'i'

1) let's make 2 dp arrays

① Start for 0th index and make it

② Start for (n-1)th index and make dp

2)



3+4 = 7 (10 is common)

7-1 = 6

= Bitonic

```
vector<int> dp1(n,1), dp2(n,1)

// Calculate dp1 (LIS from left)
for(i=1 to n)
{ for(j=0 to i)
  {
    if(arr[i] > arr[j] && dp[i] < dp1[j]+1)
    {
      dp1[i] = dp1[j]+1
    }
  }
}

// Calculate dp2 (LDS from Right)
for(i=n-2 to =0)
{ for(j=n-1 to i)
  {
    if(arr[i] > arr[j] && dp2[i] < dp2[j]+1)
    {
      dp[i] = dp[j]+1;
    }
  }
}

find max of (LIS + LDS - 1)   int max_len = 0
for(i=0 to n)
{
  max_len = max(max_len, dp1[i] + dp[j] - 1);
}
return max_length;
```

No. of LIS / Count LIS

Raj
Page:
Date: / /

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

{ {1,3,4,7} –
{1,3,5,7} – cnt = 2 (len LIS = 4)

APP

arr: [1 5 4 3 2 6 7 10 8 9]

{1,5} {1,4} {1,3} {1,2}

dp: 1 2 2 2 2 2 3

cnt: 1 1 1 1 1 5 9

{ cnt[i] represent no. of LIS of length dp[i] }

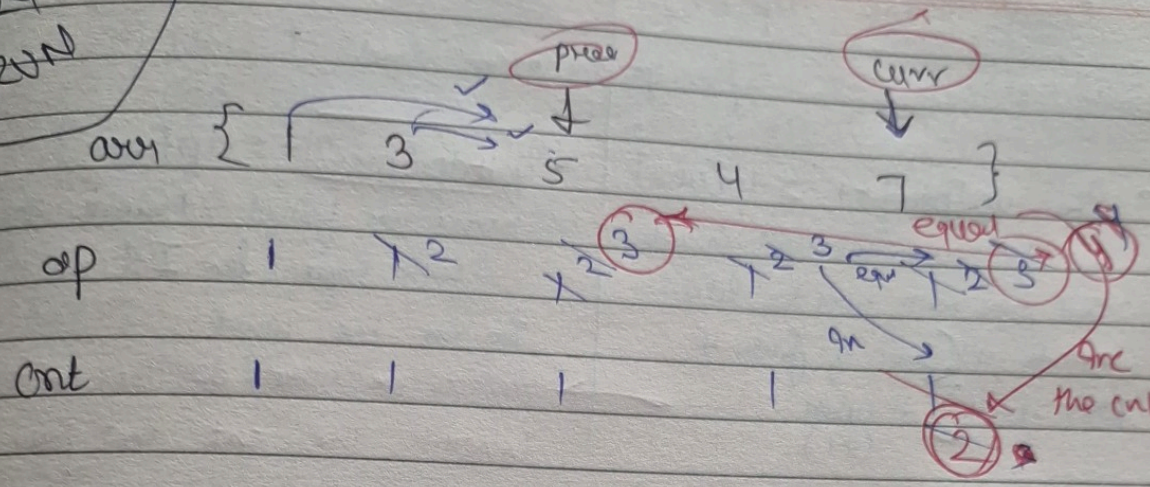
{ dp just stores length of lncs sub seq. }

at (6)
{1,6}
{5,6}
{4,6}
{3,6}
{2,6}
} 5 = cnt

at 7
{1,5,7}
{1,4,7}
{1,3,7}
{1,2,7}
} 4

But at (6) cnt = 5
{1,6,7} } 5
(2,3,4,5,1)

Correct DRY RUN



at (7)

* Increase the count only when dp of previous element which can go along with curr element is equal

inc dp as 1,3 go with 7 ✓ {1,7} {1,3}

Now at 5 (can go with 7) ∴ $dp_7 = 3$

But But But.

~~cnt~~ $dp_5 = 3$ and also $dp_7 = 3$

Both equal

∴ inc the cnt

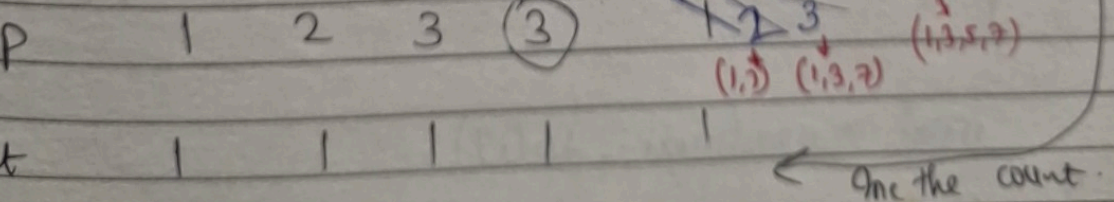
AGAIN for (7)

curr

{ 1 3 5 4 7 }

we got another 4

3+1 (4) → 4



common cond^{n} is that $arr[prev] < arr[curr]$

Raj
Page:
Date: / /

LIS

Cond^{n} to inc $dp[i]$	Cond to inc $cnt[i]$
except the common cond^{n}	
• if $dp[prev]+1 > dp[curr]$ we add 1 as we will include that prev with curr so lets include all chain of prev + prev itself + curr (+1)	• if $1+dp[prev] == dp[curr]$ hence we found 1 more seq with same len IS so we inc count → whenever we update dp then we also inherit the cnt that prev had.

CODE

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

for(int i=0 to n)
{ for(prev=0 to i)
   {   // check dp
      if( arr[i] > arr[prev] && dp[i] < dp[prev]+1 )   ①
      {
          dp[i]=dp[prev]+1;   //update dp
          cnt[i] = cnt[prev]; //inherit cnt from prev
      }
      //check for cnt
      else if( arr[i] > arr[prev] && dp[i] == dp[prev]+1 )   ②
      {
          cnt[i] = cnt[i] + cnt[prev];
      }
   }

   maxi = max(maxi , dp[i]);
}
```

Page:
Date: / /

// Now we have cnt and dp array filled and also have length of LIS

Now just return the cnt where you have longest length of LIS

```
int ans=0;
for (i=0 to n)
{
    if (dp[i]==maxi) ans+=cnt;
}
return ans;
```

Whenever you tackle (maxi) length just that it's count value to ans

≈ (Partition DP)

MATRIX CHAIN MULTIPLICATION

Page: Raj
Date: /

Solve a prob in a particular way | diff patters

* MCM

Q. Given N matrix dimensions (n x m) determine cost of multiplicat^n (cost = no. of operations done for multiplication)
Find min. cost.

eg: A B C D given

A = 10 x 30
B = 30 x 20
C = 20 x 5

no of ways of multiplication are:

A(B(CD)), (AB)(CD)
(A(BC))D, etc.

A x B = X
10x30 30x20

Operat^n = 10 x 30 x 20 = 6000

X x C
10x20 20x5

op = 10 x 20 x 5

tot opt = 6000 + 1000
= 7000. (min)

* Whenever a prob. can be solved in diff patterns
↳ Partion DP.

∴ Q. given 'N' size arr have dimension of (N-1) matrices

INPUT: [10, 20, 30, 40, 50]

A = 10 x 20
B = 20 x 30
C = 30 x 40
D = 40 x 50
⋮

general dimension: M_i = arr[i-1] x arr[i]

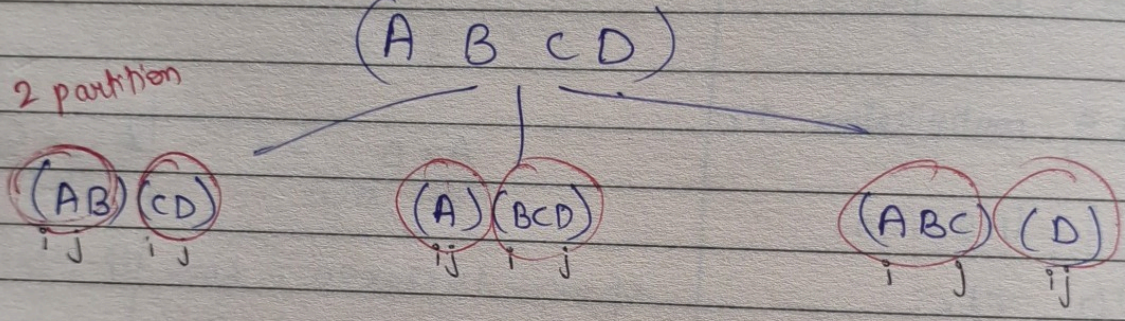
Raj
Page:
Date: / /

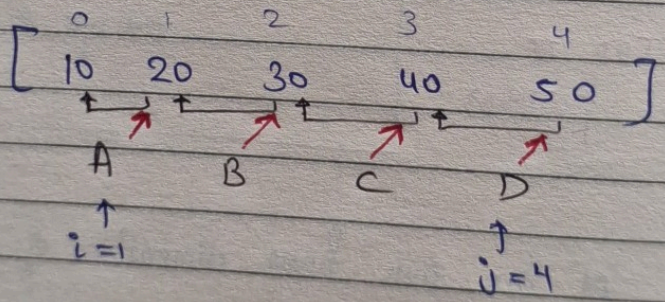
RULES of PARTITION DP

1) Start with entire ∮ block / array

2) Run a loop to try all partitions

3) Return Best possible '2 partitions'





① EXPRESS on Indices

- $f(1,4)$ = min. no. of multiplications to multiply matrix 1 to matrix 4

② Base Case

↳ When only single matrix is left

↳ `if (i == j) return 0;` → Zero operation to multiply Single matrix

③ Explore all Possibility | Try all partitions

A B C D

[10 20 30 40 50]

i j

Let run loop

```
loop ( K = i to = (j-1) )
{
[Steps + f (i, k) + f (K+1, j)]
}
```

LOOP for PARTITION.

K = 1 f(1,1), f(2,4) : A (BCD)

K = 2 f(1,2), f(3,4) : (AB) (CD)

K = 3 f(1,3), f(4,4) : (ABC) (D)

} all possibilities of ABCD are considered

Steps = arr[i-1] * arr[K] * arr[j]

↓

for ~~any~~ multiplication

(current recurn call)

A = 10 × 20
B = 20 × 30
C = 30 × 40
D = 40 × 50

Page:
Date:

DRY RUN

$f(1,2) + f(3,4) = f(A \cdot B) + f(C \cdot D)$

10×20 20×30 30×40 40×50

10×30 30×50

total steps = 10 × 30 × 50

REWRN TREE

[10 20 30 40 50]
A B C D
i j

[20] [30 40 50]
A B C D
i j

[10 20] [30 40]
A B C D
i j

[10 20 30] [50]
A B C D
i j

return 0

$f(A) + 10 \times 20 \times 50 + f(BCD)$

$A \times (BCD)$

f(BCD)
30 40 50
i j

[B] [CD]

[BC] [D]

[B] [C]

$f(B) + BCD + f(CD)$

20×

```
f(i,
{
  // S
  if(
    in
  for(
  { cu
    int
  If we get
    if(s
  }
  return
}
```

Raj

Page:
Date: / /

- Basic Recurr code

```
f(i, j, arr)
{
  // Single Matrix left → 0 operatn req.
  if(i==j) return 0;
    int mini = leg
  for(k=i to =j-1)
  { curr call steps
    int Steps = [arr[i-1] x arr[k] x arr[j]
                + f(i, k)
                + f(k+1, j)]
  If we get new minimum no of steps
    if(Steps < mini) mini = Steps;
                     update
  }

  return mini;
}
```

MEMO

here changing para are (i, j)

which ranges to i = 0 to N-1
j = 0 to N-1

- dp[N][N] ✓ = {-1}

```
f(i, j, arr, &dp)
{
  if(i==j) return 0;

if(dp[i][j] != -1) return dp[i][j]
  int mini = leg
for(k=i to =(j-1))
{
int steps = arr[i-1] * arr[k] * arr[j] +
            f(k+1, j)
            + f(i, k);

  if(mini > Steps) mini = Steps;
}

return dp[i][j] = Steps
```

mainni loop

TC: O(N×N) × N

Raj
Page:
Date: / /

TAB

Rules

① Copy Base Case

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

② Write changing Parameters / States (i, j)

* i = 1 to N-1 (memo)
* i = n-1 to 0 (tab)

Since 'i' will always be on Left of 'j'

j = 1 to n-1 ✗

* j = i+1 to n-1 (tab)

③ Copy the recurrence

```
dp[N][N]

for (i = N-1 to = 1)
{ for (j = i+1 to N-1)
 {
    int mini = leg
    for (k = i to = j-1)
    {
int steps = arr[i-1]*arr[k]*arr[j]
          + dp[i][k]
          + dp[k+1][j]

if (mini > steps) mini = steps;
    }
 }
}

return mini;
```

MIN COST TO CUT STICK

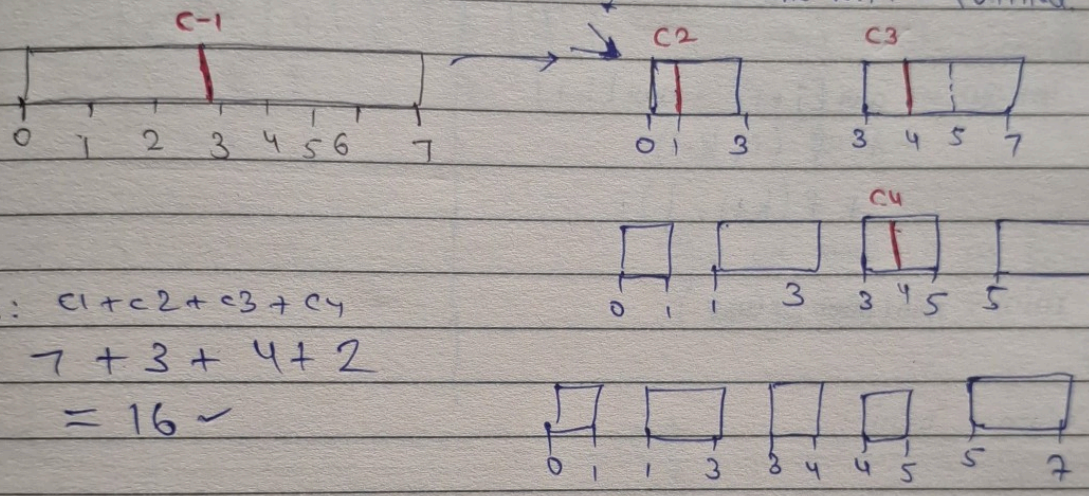
Raj
Page:
Date: / /

Given a stick of length 'n' units. You wise to cut stick at places (length) given in array 'arr' You can although change order of given cutting lengths to obtain min length

eg: IN: n=7, arr [1,3,4,5]

reorder → [3,5,1,4]

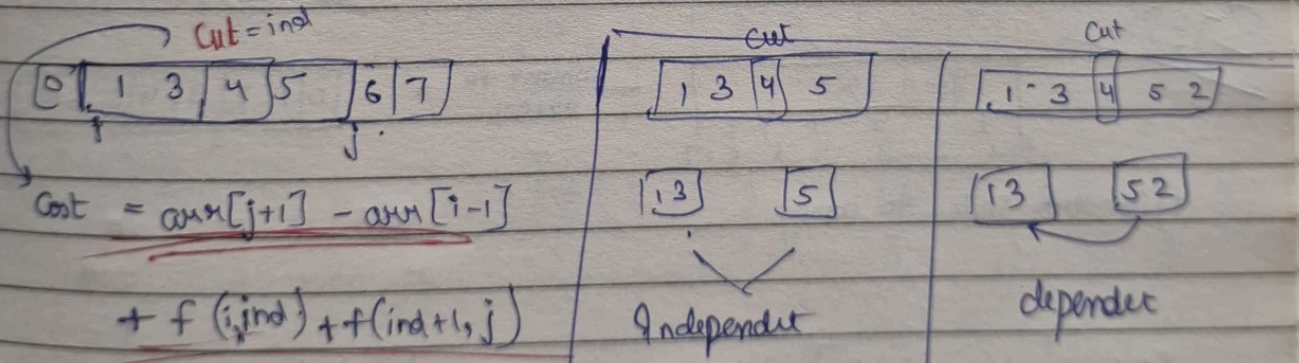
OPTIMAL WAY OF CUTTING



tot Cost: c1+c2+c3+c4

7 + 3 + 4 + 2

= 16 ✓



Cost = arr[j+1] - arr[i-1]

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

Independent

dependent

Cuts in sorted array are always independent to eachother

Page:
Date:

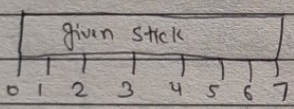
Recur

c

```
f(i, j)
{
  if(i>j) return 0;
  int mini = INT_MAX;
  for(k= i to =j)
  {
    int cost = arr[j+1] - arr[i-1]
             + f(i, k-1)
             + f(k+1, j);
    mini = min(mini, cost);
  }
  return mini;
```

```
int cost
sort(arr)
f(1, c, arr)
```

STEPS



convert to array →

0	1	2	3	4	5	6	7

1) Convert stick to array

add 0 and n at start & end

eg: [1, 4, 5, 6] , n = 9 → | 0 | 1 | 4 | 5 | 6 | 9 |

position of cuts

~~cut~~ length

1) Sort this
2) add '0' at start, 'n' at end

2) Sub prob.

- dp[i][j]

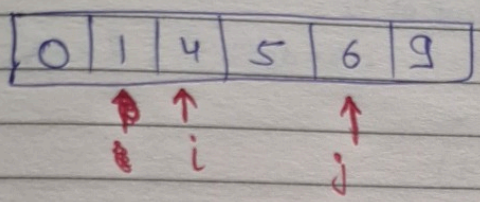
0

Cutting proc

Raj
Page:
Date: / /

2) Subprob. define

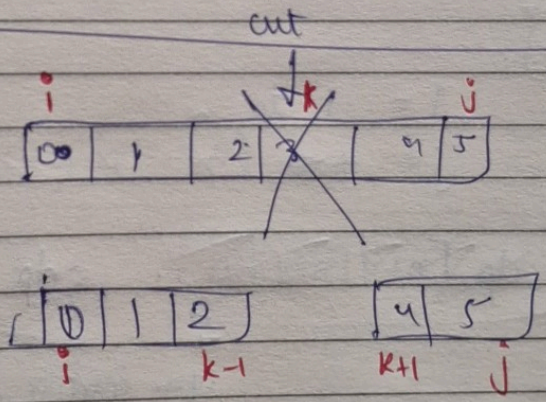
- dp[i][j] = no. of min cost to cut the sticks between arr[i-1] and arr[j+1]



- we can't include boundary no. (0,n) in cutting position

- But we need it as boundary to cut

Cutting process



Page: Raj
Date: / /

Recursion Tree.

Q. [1,3,4,5] n = 7

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

→ dp(1,4) (1,4) = index

k=1 (cut at 1)

cost = 7-0 , r = dp(2,4) , l = ~~dp(1,0)~~ = 0

dp(2,4)

k = 2 (cut at 3)

cost = 7-1 = 6 , l = dp(1, k-1) r = dp(k+1, j)

l = dp(2, 1) , r = (3, 4)

MEMO

i = 1 to n

j = ~~1~~ n to 1

~~dp[n+1][n+1]~~ dp stores cut. (size = c)

after inserting (0, n) we get (c+2) size

dp[c+2][c+2]

TAB

1)

2)

TAB

1) Base case : it's zero so no need to write, just define dp array to zero

2) i, j traversal :

i = n to 1
j = 1 to n
} Tab

int c = arr.size(); no of cut positions.

① Insert Boundries
arr.push_back(n); arr.insert(arr.begin(), 0);

② Sort arr.
sort(arr)

dp[c+2][c+2] = {0}

```
for (i=c to 1)
{ for(j=i to =c)
  {
```

Same as recurrence

```
  }
}
```

BRUST BALLOONS

Raj
Page:
Date: / /

Given n balloons with no. written on it, if you brust a balloon you will get ↓ arr[i-1] * arr[i] * arr[i+1]
. Try to get max Points

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

~~Brust~~ : Brust : 3 : 1 x 3 x 1 = 3 [1,5,8]
APP-1 Brust 1 : 1 x 1 x 5 = 5 [5,8]
" 5 1 x 5 x 8 = 40 [8]
" 8 1 x 8 x 1 = 8

56

APP-2 Brust 1 : 3 x 1 x 5 = 15 [3,5,8]
" 5 ~~1 x 3~~ 3 x 5 x 8 = 120
" 3 1 x 3 x 8 = 24
" 8 8 1 x 8 x 1 = 8

160
max

STEPS

LAST
2nd LAST
f(i,j)
loop(k=
{
}

eg: 1 [b1 b2 b3 b4 b5] 1

Page: Raj
Date: / /

- Unlike previous, the subprob. here can't be Independent to each other.

But if we move in opposite way, subproblems are independent

LAST ← let (b4) be the last to burst : 1 x b4 x 1

2nd LAST : [b1 b4] [b2 b4] [b4 b3] [b4 b5] ..... etc
lot of possible combinat But b4 must be there

that is how we cover the all patterns Independently

f(i,j)

Cost = arr[i-1] x arr[ind] x arr[j+1]

loop(k=i to j)
{
}

why not ind-1

why not ind+1

* we assume k^{th} as last balloon to burst in i to j

So k is only surrounded by (i-1) and (j+1);

i n

f(i, j)

Base Case

{

if (i > j) return 0;

maxi = INT_MIN;

for (ind = i to = j)

{

int cost = arr[ind-1] * arr[ind] * arr[j]
+ f(i, ind-1) + f(ind+1, j)

maxi = max(maxi, cost);

}

return maxi;

}

for MEMO

i = 1 to n

j = n to 1

dp[n+2][n+2]

[1, 'n' size array, 1]

n+2

(n+2) size to be safe after Padding

TAB

Raj
Page:
Date: / /

i = n to 1 ✓

j = i to n ✓ Since j can't be smaller than i

return dp[i][n]

TYPE-2

of PARTITION DP

Q. PALINDROME PARTITION. : given string s, partition s such that every sub. string is palindrome

Return mini cuts needed for this.

eg: "a a b b"

a|a|b|b

max cuts = n-1 , But we need minimum

APP

b a b a b c b a d c e d e

↑

Partition < Yes / No] for each point we have 2 option

↓

- If string on left is palindrome we put a partition - YES

(Take min of all)

1) Express in terms of Index f(i)

2) Explore all ways

3) Take min

4) Base Case

```
f(i)
{
  if (i == n)
  { return 0}

  minCost = INT_MAX.
  for (k = i to n)
  {
     if (isPalindrome [i, j, str)); // check if (i to j) string is palindrome
     {
        cost = 1 + f(j+1);
        minCost = min (minCost, cost);
     }
  }
  return minCost
```

Base case

- MEMO

TC = $O(N^2)$

SC = $O(N) + O(N)$

i : 0 to n-1

dp [n]

This code does a partition at end, so (-1) always

Raj

Page:
Date: / /

TAB

1) Base Case

if $dp[n] = 0$

So make $dp[n+1] = \{0\}$ So that $dp[n] = 0$

2) Traversed By loop

$i = n-1$ to $= 0$ tab (opp of memo)

3) Return the Base

return $dp[0] - 1$;

PARTITION for MAX SUM.

Raj
Page:
Date: / /

Q Given array, partition it in such a way that all the subarray are atmost of length 'k' and after partition each subarr has their value changed to max value of that subarray

Return largest sum of array after partition.

Similar to pevv prob eg: [1, 15, 7, 9, 2, 5, 10], k=3

[15, 15, 15, 9, 10, 10, 10]

ans = 84 — (optimal)
(for max sum)

1) We will keep track of max within a subarray dynamically

MEMO

```
f(i)
{   Base Case
    if(i==n) return 0;
    max ~~sum~~ in Subarray
    int maxWin = INT_MIN
    int maxSum = INT_MIN

    ~~if~~ for(j=i; ~~j<n~~ && j<(i+k); j++)
                    max length is k
    {   Max in Subarr
    maxWin = max(maxWin, arr[j])
        int sum = maxWin * (j-i+1)
                  + f(j+1)

    maxSum = max(maxSum, sum)
    }

    return maxSum
```

dp[n] = -1 } easy