Dynamic Programming

Points to Understand:

- Understand if it is a dp problem, if a problem asks for:
  → count number of ways
  → get max or min

- When a concept of all possible ways or best way comes in ---> there we mostly use recursion

- To write recursion-----
  1. Try to represent problem in terms of index
  2. Do all possible stuffs on that index acc to the problem statement
  3. if it says count all the ways -> sum of all stuff
  4. if min or max -> min of all stuff or max of all stuff

***Memoization:: tend to store value of the subproblem in some map/table*

- Steps to convert a recursion to a dp solution
  1. declare a dp array of size n+1 (considering zero)
  2. store the final ans of f(n) {eg..f(n-1) + f(n-2)} in dp[n]
  3. add a check --> if dp[n] == -1 then store the calculated value else return the value stored i.e., dp[n]

****Tabulation (Bottom Up ==== Base case to required ans)*

- Steps to convert into Tabulation:
  1. initialize the same dp array as we did earlier of size n+1
  2. write the base case
  3. since bottom up, we go from base case to final answer, we fill run loop from 1 to n and do the steps
  4. here we don't need to call f here like {f(n-1) + f(n-2)}, replace f with dp {dp[n] = dp[n-1] + dp[n-2]}

Problems:

Frog Jump

refer https://www.codingninjas.com/studio/problems/frog-jump_3621012

Maximum Sum of Non-Adjacent Elements

Explaination:

given an array of N integers. return the subsequence such tht no two elements are adjacent to each other

eg... [2 1 4 9] ==> o/p - 2+9 = 11

[1 2 4] ==> o/p - 4

Approach:

out of all subsequence , we need to pick one with maximum sum

Pick/unpick method---

pseudo code

```
f(index){
  if index = 0 ==> return a[0]
  If (index is negative) ==> return 0
  // either take that element
  pick = a[index] + f(index-2)
  // or dont pick the element
  inpick = 0 + f(index-1)
  // return max
  return Max(pick, unpick)
```

}

Recursion time complexity - 2^n

To optimize memoize -> tabulation -> space optimize

Memoization : refer steps above **

Tabulation : refer steps above ***

Space optimization

Since to calculate for any index, we need the previous 2 values, we dont need to save in array,

rather we can take two variables and replace after every iterationcurr = will be calculated

prev2 = prev

prev = curr

refer https://www.codingninjas.com/studio/problems/maximum-sum-of-non-adjacent-elements_843261

House Robber

Explanation:

A robber has to rob houses and take maximum amount of money,

condition is that he can't rob in two adjacent house and

all the houses are in circle which indicates that first and last house cannot be robbed at same time

Approach:

same as previous one with additional condition is circle

Since it is circle our answer can't have both first and last house

so if we take out last house and do the same logic for rest houses we will get one part

we will do the same by removing the first house

and then take the max one

edge case - if array has one house ==> return that house value

refer - https://www.codingninjas.com/studio/problems/house-robber_839733

Ninja's training (2D DP)

1. Ninja's training (2D DP)

Explanation:

Ninja has to train in N Day training schedule.

Each day he can perform any one task [Running, fighting practice, Learning new moves]

Each activity will have some point each day

Condition - can't do same activity in two consecutive days

find maximum merit points

[

1 2 5

3 1 1

3 3 3

] ---- 3

Approach:

Whenever maximum we'll think of Greedy but that will fail

like in below eg

10 50 1 --> Day 0

5 100 11 --> Day 1

If greedy --> Day 0 = 50 --> Day 1 = 11 == so total 61 which is wring

Correct ans will be 10 + 100 == 110

So next, we will try all possible ways and take max ===> recursion

For recursion, we have 3 steps

1. express in terms of index

2.do all steps

3. Max

So, a day can be termed as index

Generally, we'll do recursion where we do top-down approach from (n-1) -> 0th index

In order to select a task for that day, we need to know wich task we performed last time, so we need to pass last task performed

So, f(index, last)

last--

0 -> task 0

1 -> task 1

2 -> task 2

3 -> no task

so, f(n-1,3) we will run which will return the maximum merit options in the array given no task is done initially

<u>pseudo code</u>

```
f(day, last){
 //base case
 if(day == 0) {
  maxi = 0
  for(i = 0 -> 2)
   if(i!=last) maxi = Max(maxi, task[0][i]
  return maxi
 }
 //other cases
 maxi=0
 ****
 for(i = 0 -> 2){
  if(i!=last) point = task[day][i] + (maxi, f(day-1, i)
  maxi = Max(maxi, points)
  }
  return maxi
}
```

To optimize do memoization -> tabulation -> space optimization

dp array will be of size [N][4]

since there are 2 parameters so 3 for task values and 4th for the last

<u>tabulation</u>

//declare a dp array step 1

we will declare a dp 2d matrix of size [n][4] (4 because there can be 4 possible value of last 0,1,2,3) --> initialize with -1

we need to compute for every possible combination i.e.,

compute a the day when we performed

task 0 on last day

task 1 on last day

task 2 on last day

task 3 on last day i.e no task performed at last day

// base case step 2

day 0 is our base case so

dp[0][0] i.e on 0th day taken we did 0 for the last day so max(points[0][1], points[0][2])

dp[0][1] i.e on 1th day taken we did 0 for the last day so max(points[0][0], points[0][1]) ...and so on till dp[0][3]

```
//
for(day 1 -> n-1){
 //run for all sets of last value
for(last = 0 -> 3){
dp[day][last] = 0
//do the same as we did in recursion and store the value in dp[day][last] refer****
}
}
```

T.C memoization => o(N*4)*3(3 since we are running loop for size 3 every time)

S.C -> O(N) + O(N*4)

T.C tabulation => o(N*4)*3(3 since we are running loop for size 3 every time)

S.C -> O(N*4) == removed stack space

S.C (for space optimized one) -> O(1* 4) == O(4)

refer..https://www.codingninjas.com/studio/problems/ninja-s-training_3621003

Total Unique Paths (DP Grids)

Explanation:

we need to find total number of unique paths from start to destination

(start - 0,0 to dest - m-1,n-1)

condition- we can either move right or down

base case---

reached destination ? return 1 -> count

else return 0 -> dont count

exceeded boundary ir row < 0 or col < 0

<u>pseudo code</u>

```
    f(i,j){
    step 1...express everything in terms of index and write base case
```

```
if(i == 0 && j == 0) return 1
if(i < 0 || j < 0) return 0

//top down approach we can say we are going from m-1,n-1 to 0,0 ---> up and left direction
step 2...explore/do all stuff
  up = f(i-1, j)
  left = f(i, j-1)

  step 3...sum up to get count
  return up+left
}
```

T.C -> 2^(m*n) S.C -> stack space === path length (m-1)+(n-1)

Memoization

declare dp of size [m][n] and initialize with -1

store the ans that we're returning

check before call

T.C -> O(n*m)

S.C -> recursion stack space + dp space -> O((n-1)+(m-1)) + O(n*m)

Tabulation

```
dp[n][m]
for(i = 0 -> m-1){
  for(j = 0 -> n-1){
  //base case
   if(i == 0 && j == 0)  dp[0][0] = 1
   if i > 0 --> up = dp[i-1][j]
   if j > 0 --> left = dp[i][j-1]
  dp[i][j] = up+left
}
}
```

T.C -> O(n*m)

S.C -> O(n*m)

Space optimization

we are using the previous row and column, so we can just store the previous row in an array and whole matrix is not required

S.C -> O(m)

More optimized - n+m-2Cm-1 (Combination problem) ...refer https://www.youtube.com/watch?v=t_f0nwwdg5o

refer......https://www.codingninjas.com/studio/problems/total-unique-paths_1081470[i]

Minimum Path Sum

Explanation:

given a 2d grid with N rows and M columns. Each point in the grid has some cost associated with it.

Find path with minimum cost from 0, 0 to n-1,m-1.

Approach:

as last problem, here also greedy will fail

so we will apply same approach as previous problem with an additional parameter cost

Recursion/memoization:

```
f(row, col){
 if row == 0 && col == 0 return grid[0][0]
 if(row is neg || col is negative) return MAX_INT

 //step 2...explore all stuff
```

```
  up = grid[row][col] + f(i-1, j)

  left = grid[row][col] + f(i, j-1)


  //memoization - add a check before calling

  if(dp[row][col] != -1 return dp[row][col]


  //step 3...get min

  return min(up,left)

}
```

Time Complexity: O(N*M)

Reason: At max, there will be N*M calls of recursion.

Space Complexity: O((M-1)+(N-1)) + O(N*M)

Reason: We are using a recursion stack space:O((M-1)+(N-1)), here (M-1)+(N-1) is the path length and an external DP Array of size ‘N*M’.

Tabulation:

Time Complexity: O(N*M)

Reason: There are two nested loops

Space Complexity: O(N*M)

Reason: We are using an external array of size ‘N*M’’.

Space optimization:

Time Complexity: O(M*N)

Reason: There are two nested loops

Space Complexity: O(N)

Reason: We are using an external array of size ‘N’ to store only one row.

refer... https://www.codingninjas.com/studio/problems/minimum-path-sum_985349?source=youtube&campaign=striver_dp_videos&utm_source=youtube&utm_medium=affiliate&utm_campaign=striver_dp_videos&leftPanelTab=0

Triangle

[Minimum path sum in Triangular Grid]

Explanation:

We are given a Triangular matrix. We need to find the minimum path sum from the first row to the last row.

At every cell we can move in only two directions: either to the bottom cell or to the bottom-right cell

Approach:

Different: we don't have a fixed destination; we need to return the minimum sum path from the top cell to any cell of the bottom row

So, here we will start our recursion from 0,0 since we have fixed source and not a fixed destination.

Pseudo code:

```
f(row, col){
 //base case
If(i== N-1) return mat[I][j]
//step2
Down = mat[I][j] + f(I+1,j)
diag = mat[I][j] + f(I+1, j+1)
//step3
return min(down, diag)
}
```

Memoization:

1) declare dp array of size [N][N]
2) Before calling, add a check if dp[i][j]!= -1 return dp array with that row and column.
3) Set the ans of recursive call in dp

Time Complexity: O(N*N)

Reason: At max, there will be (half of, due to triangle) N*N calls of recursion.

Space Complexity: O(N) + O(N*N)

Reason: We are using a recursion stack space: O((N), where N is the path length and an external DP Array of size ‘N*N’.

Tabulation:

Steps:

1) Declare a dp[] array of size [N][N].
2) First initialize the base condition values, i.e the last row of the dp matrix to the last row of the triangle matrix.
3) Our answer should get stored in dp[0][0]. We want to move from the last row to the first row. So that whenever we compute values for a cell, we have all the values required to calculate it.
4) If we see the memoized code, the values required for dp[i][j] are dp[i+1][j] and dp[i+1][j+1]. So we only need the values from the ‘i+1’ row.
5) We have already filled the last row (i=N-1), if we start from row ‘N-2’ and move upwards we will find the values correctly.
6) We can use two nested loops to have this traversal.

Time Complexity: O(N*N)

Reason: There are two nested loops

Space Complexity: O(N*N)

Reason: We are using an external array of size 'N*N'. The stack space will be eliminated.

Space optimization:

Time Complexity: O(N*N)

Reason: There are two nested loops

Space Complexity: O(N)

Reason: We are using an external array of size 'N' to store only one row.

Refer...
https://www.codingninjas.com/studio/problems/triangle_1229398?source=youtube&campaign=striver_dp_videos&utm_source=youtube&utm_medium=affiliate&utm_campaign=striver_dp_videos&leftPanelTab=0

i