Dynamic Programming

~ by Striver Bhaiya.

Lecture - 1 (17/01/22)

① Fibonacci No

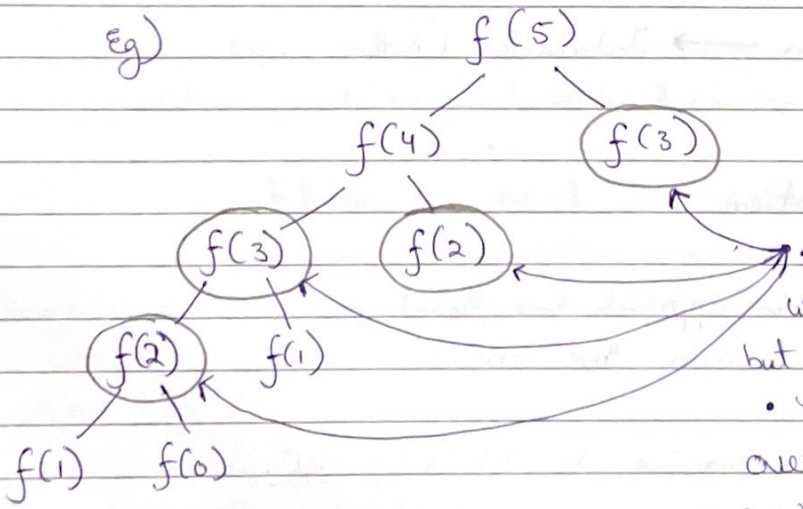
0 1 1 2 3 5 8 13 21 ....

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

Part - 1 :- Memoization

Overlapping Subproblems.

Eg)



- These problems were computed once but these getting repeat
- These are called overlapping Subproblems.
- We can save these recurrence and that process is called memoization.

Code

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

Base case
Memorization
recursive call

This method of Dynamic Programming is called Memoization or Top-Down Approach. (It is an optimization of Recursion).

TC ⟶ O(n)
SC ⟶ O(n) + O(n)
↳ array.

Now, Recursion ⟶ Tabulation. (Bottom-Up)

Part-2: Tabulation

Main logic: In this approach we start from the base case and reach the answer **bottom to top approach**

Steps to convert recursive to tabulation sol^n.

1. Declare dp[] array of size n+1.
2. Initialize the base case values, i.e i=0 & i=1 of the dp[] as 0 & 1 resp.
3. Set a loop which iterates from 2 to n and for every index set values as dp[i-1] + dp[i-2];

~~ed~~ code

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

dp[0] = 0 , dp[1] = 1;

for( int i = 2 ; i<=n ; i++)
    dp[i] = dp[i-1] + dp[i-2] ;

cout << dp[n] ;
```

TC → O(n)
SC → O(n)

Part 3: Space Optimization.

If we carefully ~~look~~ look at the relation

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

there is actually no need to maintain an array because notice. for every element all we need is it's last two elements. prev & prev2

Algorithm

- Calculate curr_i = prev + prev2;
- where prev = 1, prev2 = 0;
- After calculation: prev2 = prev;
  prev = curr_i;

for every iteration from 2 to n and print 'prev' as answer.

Code

```
int prev2 = 0;
int prev = 1.

for (int i=2; i<=n; i++)
{
    int curr_i = prev + prev2;
    prev2 = prev;
    prev = curr_i;
}
cout << prev;
```

TC ⟶ O(n)
SC ⟶ O(1).

Lecture - 2 : Climbing Stairs (18th Jan 2022)

① How to identify A problem is a DP prob?

Generally (but not limited to) if the problem statement asks :-

a) Count the total number of ways
b) Given multiple ways of doing a task, which will give it the min or max output.

In such problem we can apply recursion, then memoization for DP.
General trend.

Recursion ⟶ Memoization ⟶ Tabular form ⟶ Space Optimization

2) General Steps to Solve Problem After Identification

a) Try to represent the problem in terms of indexes.
b) Try all possible choices/ways at every index according to the problem statement.
c) If the ques^ states :

-) Count all ways → return sum of all ways.
-) find max/min → return the choice with max/min way.

for the Ques^ (climbing Stairs)

The question is very similar to fibonacci series so the recurrence is

```
if (ind==0) return 1;
if (ind==1) return 1;

return f(ind-1) + f(ind-2);
```

The memoization, Tabulation & Space Opt. is the same as fibonacci series.

Please Turn Over.

Memoisation

- Create dp[n+1] = -1.
- Our main goal is to memorize the overlapping subproblems. In this, ~~if~~ we wants to store the best soln for a particular value of n.
- Simply, we check if(dp[ind] != -1)
  yes → return dp[ind]
  otherwise, continue and store the result i.e
  dp[ind] = min(l, r);

TC → O(n)
SC → O(n).

Space - Optimization

If we notice carefully, the values required for every iterations are
dp[i], dp[i-1] & dp[i-2] like last problem we can handle this.

We Declare 2 variables prev, prev2 = 0; Calc JumpOne and JumpTwo and put their min in curr and update prev = prev2
prev2 = curr and continue.

PTO

Code

```
int f(int n, vector<int>& heights)
{
    int prev, prev2 = 0;

    for(int i=1; i<n; i++)
    {
        int l = prev + abs(heights[i]-heights[i-1]);
        int r = INT_MAX;
        if(i>1)
            r = prev2 + abs(heights[i] - heights[i-2]);

        int curr = min(l, r);
        prev2 = prev;
        prev = curr;
    }
    return prev;
}
```

TC → O(n).
SC → O(1).

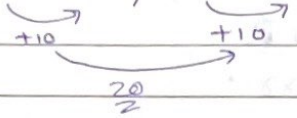
Lecture: 3 – Frog Jump (19th Jan 2022)

Problem Statement

Given no. of stairs and frog, frog has to jump from 0^{th} to $(n-1)^{th}$ stair. At the time he can only jump one (or) two steps. Every jump consumes an energy i.e abs(heights[i] - heights[j]) where heights is height of stairs.

Input N = 4
(10, 20, 30, 10)

Output 20 { 10 → 20, 20 → 10 }



Solution) The first approach which comes to mind is greedy. That is to compute min. jump but for the example below. Greedy fails.

[30, 10, 60, 10, 60, 50]

According Greedy ⇒ 30 → 10 → 10 → 50 = 60

But Non-Greedy ⇒ 30 → 60 → 60 → 50 = 40
optimal Sol^n

Steps for Recursive

Step-1) Express the problem in terms of indexes

(a) This can be easily done as their arr ind [0,1,2...n-1]
(b) We can say that f(n-1) signifies min. energy required to move from 0 to (n-1) stair.
(c) f(0) simply should give us the answer as '0' (base case)

Step-2) Trying all choices to reach the goal.

The frog can jump either by one step or by two step. We will calculate the cost of the jump from the height array. Note: The rest of the cost will be ~~jumped~~ returned by the recursive calls that we make.

Step-3) Take the minimum of all choices.

As the ques^n asks for, we return the min. of Step 2.

Note At ind == 1, we ~~fo~~ can not call f(n-2) we can only compute f(n-1).

Pseudo - Code

```
int f(ind, height[])
{
    if(ind == 0) return 0;

    int JumpOne = f(ind-1) + abs(height[ind] - height[ind-1]);

    if(ind > 1)
        int JumpTwo = f(n-2) + abs(height[ind] - height[ind-2]);

    return min(JumpOne, JumpTwo);
}
```

Tabulation

- Declare dp[] array.
- Set dp[0] = 0,
- Iterate from 1 to (n-1) and calc JumpOne and JumpTwo and set dp[i] = min(JumpOne, JumpTwo).

Pseudo-Code

```
int f(int ind, vector<int> &heights)
{
    vector<int> dp(n+1, -1);
    dp[0] = 0;

    for(int i=1; i<n; i++)
    {
        int l = dp[i-1] + abs(height[i] - height[i-1]);
        int r = INT_MAX;
        if(i>1)
            r = dp[i-2] + abs(height[i] - height[i-2]);

        dp[i] = min(l, r);
    }
    return dp[n-1];
}.
```

For Memoisation and Space optimazation refer pg 6

Lecture 4- Frog Jump with K Distance. (22nd Jan 2022)

Exactly like the last question, this time the frog can skip upto 'k' stones. We have to find minimum energy lost.

Eg) 10 4
40 10 20 70 80 10 20 70 80 60

Ans) 40
1 → 4 → 8 → 10

The code is exactly the same we just need to generalize (ind-1), (ind-2) to (ind-K) by using a loop from i=1 to K. We compute all the possible scenario.

Recursive Code with Memoization.

```
int f(int ind, vector<int>& arr, vector<int>& dp) {
    if (ind == 0) return 0;
    if (dp[ind] != -1) return dp[ind];
    int min_ans = INT_MAX;
    for(int i=1; i<=k; i++) {
        if(ind-i >= 0)
            int Jump = f(ind-k, arr, dp) + abs(arr[ind] -
                                               arr[ind-i]);
        min_ans = min(min_ans, Jump);
    }
    return min_ans;
}
```

Lecture 5: Maximum Sum of Non-Adjacent Elements
AKA House Robber (on leetcode)

Problem Statement

Given an array of 'N' positive int, return maximum subsequence such that no 2 adjacent elements are allowed.

Eg)

N = 3 , Arr = 1, 2, 4
Output = 1+4 = 5

N = 4 , Arr = 2, 1, 4, 9
Output = 11 (2+9)

Solution

Logic: As we need to find sum of subsequences, an approach that comes to mind is to generate all subs and pick the one with max. sum.

To Generate subsequence, we can use the pick/not pick technique. i.e.

1. At every index of the array we have two choices.
2. First, to pick the array element at that index and consider it in subs.
3. Second, to leave the array element at that index and not consider in subs.

Recursive And Memoization Sol"

Step 1) form the function in terms of indexes

- We can define our function f(ind) as: Max Sum of Subs from 0 to (ind).
- We need to return f(n-1) as our final answer.

Step 2) Try all the choices to reach the goal.

- Using the pick/not pick technique...
  if we pick an element then, pick = arr[ind] + f(ind-2)
  We are choosing (ind-2) because we can not choose adjacent elements.
- Next we need to ignore the current element in our subs. Non Pick = 0 + f(ind-1). As we don't pick the current element, and consider it's adjacent element.

Step 3) Take the max of all choices.

- Calculate the max between Step 2 options.

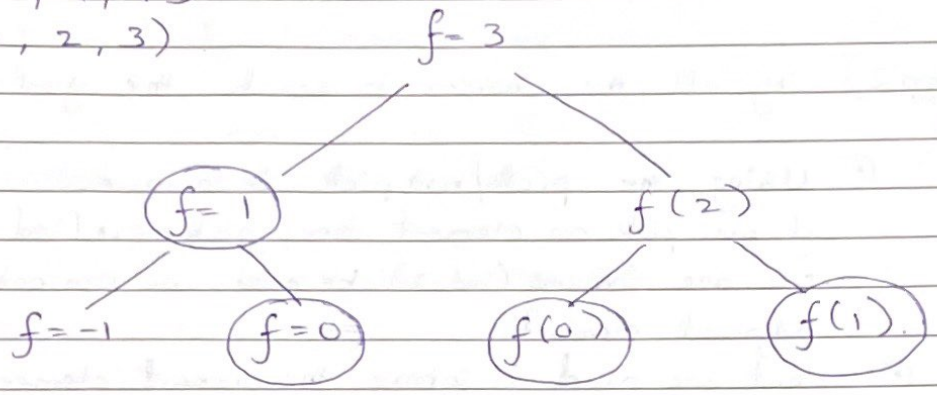
Base Case : ① If ind = 0, if we are at ind=0 this mean we had ignored ind=1 and we want to pick it Therefore we return arr[0].

② if ind < 0, this case can hit when we call f(ind-2) and we were at ind = 1; Thus we are trying to call f(-1) which is not possible and hence we return 0

Memoization.

Observe the tree below, there are overlapping sub-prob. Hence we can use memoization.

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



Simple, ① Creat DP[] set all values to -1
② before returning max, store it in dp[]
dp[ind] = max(pick, not pick).

Code

```
int f(int ind, vector<int> &arr, vector<int> &dp){
    if(ind==0) return arr[ind];
    if(ind<0) return 0;
    if(dp[ind]!=-1) return dp[ind];

    pick = arr[ind] + f(ind-2, arr, dp);

    notpick = f(ind-1, arr, dp);

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

TC → O(N)

SC → O(N) + O(N)

Stack Space Array Space

Tabulation

1. Declare a dp[] array size (n)
2. base condition, dp[0] = arr[0]
3. Set an iterative loop which traverse the array (from 1 to n-1) and for every index calculated pick and notpick.
4. Set dp[i] = max(pick, not pick)

```
int f(int n, vector<int> &arr, vector<int> &dp)
{
    dp[0] = arr[0];

    for(int i=1; i<n; i++)
    {
        int pick = arr[i];
        if(i>1)
            pick += dp[i-2];
        int notpick = 0 + dp[i-1];

        dp[i] = max(pick, not pick);
    }
    return dp[n-1];
};
```

TC → O(N)

SC → O(N)

↳ external array dp[n+1]

Space Optimization.

Notice, for space opt. try noticing the elements required to compute in tabulation. In this case.

dp[i], dp[i-1], dp[i-2]
like we have solve for prev. problems, it's the same logic.

```
int f(int n, vector<int> &arr)
{
    int prev = arr[0], prev_2 = 0;

    for (int i=1; i<n; i++)
    {
        int pick = arr[i];
        if(i>1)
            pick += prev_2;
        int notpick = prev;
        int curr_i = max(pick, notpick);
        prev_2 = prev;
        prev = curr;
    }
    return prev;
}.
```

TC → O(N)
SC → O(1)