Lecture 79 (DP-1)

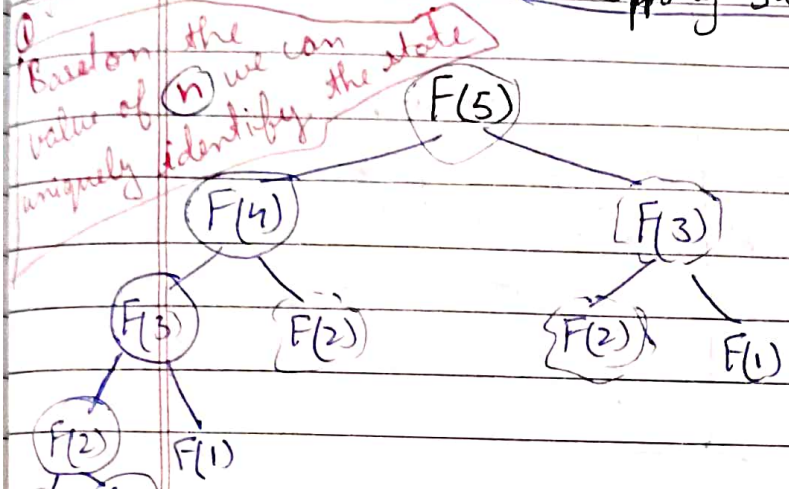
CLASSTIME Pg. No.
Date / /

⇒ What is Dynamic Programming?

→ reuse more repeat less

→ if you always already know about something dont recompute it.

⇒ What are overlapping Subproblems? (repeating)



① Based on the value of (n) we can uniquely identify the state

recursion used in fibonacci

② $f(n) = f(n-1) + f(n-2)$

{All Dotted Circles are Repeating}

⇒ What is optimal Solution?

Solve a problem by adding optimal Solution of its small/sub problems.

Greedy	Dynamic Programming
① A greedy algorithm builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.	① A dynamic programming algorithm builds up the solution to a problem by solving its subproblems recursively.
② It is useful for solving problems where making locally optimal choices at each step leads to a global optimum.	② It is used where the optimal solution can be obtained by combining optimal solutions to Subproblems.

Greedy	Dynamic Programming
(3) It does not necessarily consider the future consequences of the current choice (Local optimum)	(3) Dynamic programming stores the solution to subproblems and reuses them when necessary to avoid solving the same subproblems multiple times. (Global optimum)
(4) The greedy approach is generally faster and simpler, but may not always provide the optimal solution.	(4) Dynamic programming guarantees the optimal sol. but is slower and more complex.

⇒ Where to Use DP?

The optimal structures or overlapping sub problems

⇒ Various approaches :→ TD (Top Down) (recursive) Memoization | BU (Bottom Up) Tabulation (iterative)

How ⇒ Where to use DP (1) → State of the DP
(2) → How the subproblems are related / identify formula bw them.
(3) → storing the results once completed.

CLASSTIME | Pg. No.
Date / /

⇒ Solving Fibonacci by applying Dynamic Programming

Step ①, ② ⇒ written in page 1

Step ③ ⇒ Make an array to store the values which are founded once.
So that these values (not find again).

⊙ for f(n) there are (n+1) unique states

```
# include<vector>
using namespace std;

Vector < int > dp;      // store ans for states which which are computed for the first.

int f(int n){
    if (n == 0 or n == 1) return n;
    if (dp[n] ≠ -1) return dp[n];
    return dp[n] = f(n-1) + f(n-2);
}
int main(){
    int n;
    cin>>n;
    dp.resize(n+1, -1);
    cout << f(n) << "\n";
    return 0;
}
```

Memoization :- Ek bdi problem se choti problem par jaa rhe ho phir vapis aa rhe ho or aap bich me kbhi bhi apna kaam kar skte ho kuki aap apni values store karva rhe ho.
(nyi state ki calculated)

recursive approach

CLASSTIME | Pg. No.
Date / /

Tabulation ⇒ Downward (Bottom UP) approach

iterative approach

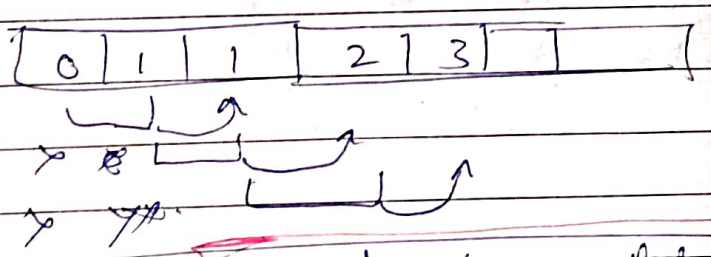
Chati se badi problem ki trf jaate hain.

↳ order/structure maintained
⇒ konsi -2 subproblems ko use krke badi sub-problems solve hongi.

function jitne variables pe depend karega utne dimensions ki array bnegi.

Jab tum subproblems ko solve karoge then unke solve hone k baad to bigger problem aayegi bss vhi needed hogi sub problem needed nahi hogi. {Iska space variant kr skte hain}

eg:



Continued to previous fibonacci code

Code Approach ①

```
int fbu(int n) {
    dp.resize(n+1, -1);
    dp[0] = 0;
    dp[1] = 1;
    for(int i=2; i<=n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

if(n==0||n==1) return n;

Used for space complexity O(1)

CLASSTIME Pg. No.
Date / /

Approach 2 (optimized bottom up)

```
int fbuo(int n) {
    int a=0;
    int b=1;
    int c;
    for(int i=2; i<=n; i++){
        c=a+b;
        a=b;
        b=c;
    }
    return c;
}
```

(int main() Same as prev,

Leetcode 198 Professional House Robber

[Given an array of money in houses] {No 2 adjacent houses can be robbed}

Find the Max profit by robbing the houses.

recursive approach

f(i, arr) = max { arr[i] + f(i+2, arr) ; 0 + f(i+1, arr) }

Ek hi parameter (i) se sari states define ho rhi hai to humara kam 1D array me hi ho jaega

f(i, arr) → max profit by robbing all the houses from [i to n-1]

f[0, arr]

Normal recursive array ka soln time limit exceed karega so use modify karenge (dp array) (bnayenge)

To reduce more time complexity we use bottom up approach.
Both codes written in leetcode.

Lecture - 80 (DP-2)

CLASSTIME | Pg. No.
Date / /

Q8 leetcode [91] Decode Ways

{recursive call}

eg 152 → decoding of 52 → x = 1 → 1,5,2

↳ decoding of 2 → y = 1 → 15,2

(x) Phle ek no. ko Rok kar bche hue ki decodings nikalni hai

(y) Phle 2 no. ko Rok kar bche hue ki decodings nikalni hai.

If y > 26 then not posible consider only x

Total decodings (x + y)

Code concept

$f(str, idx) =$ f(str, idx+1) + if (str[idx] × 10 + str[idx+1] ≤ 26) f(str, idx+2)

returns the counts of the decodings of the string str from idx to n-1;

//base case: if (idx == n-1) ↳ decody → str[n-1]

if (idx ≥ n) ↳ 0

find ans. f(str, 0)

Problems on Tabulation → N Stones

Q → There are N stones, numbered 1, 2, —, N. The height of ith stone is h_i. There is a frog who is initially on stone 1. He will repeat an action some no. of times to reach stone N. The action is that if the frog is currently on stone i, it jumps to one of the following: stone i+1, i+2, —, i+k

Here a cost of $|h_i - h_j|$ is incurred where j is the stone to land on.

→ Find the minimum possible total cost incurred before the frog reaches stone N.

input : n = 5, k = 3, Arr = 10, 30, 40, 50, 20
output: 30

Sol:-

$$f(height, i, K) = \min \begin{cases} |h_i - h_{i+1}| + f(heights, i+1, k) \\ |h_i - h_{i+2}| + f(heights, i+2, k) \\ |h_i - h_{i+3}| + f(heights, i+3, k) \\ \vdots \\ |h_i - h_{i+k}| + f(heights, i+k, k) \end{cases}$$

min cost to reach $(n-1)^{th}$ stone from i^{th} stone with max K jump allowed.

$$f(heights, i, k) = \min_{\forall\, j \in [1, k]} \left(f(heights, i+j, k) \right)$$

```
for (j=1, j<=k, j++)
    ans = min(ans, f(heights, i+j, k))
```

Code

```
#include <vector>
using namespace std;
vector<int> dp;
f(vector<int> &heights, int i, int k) {
    if (i == heights.size() - 1) return 0;
    if (dp[i] != -1) return dp[i];
    int ans = INT_MAX;
    for (int j = 1; j <= k; j++) {
        if (i+j >= heights.size()) break;
        ans = min(abs(heights[i] - heights[i+j]) +
                  f(heights, i+j, k));
    }
    return dp[i] = ans;
```

```
int main(){
    int n;
    cin>>n; dp.clear(); dp.resize(100005, -1);
    int k
    cin>>k;
    Vector<int> V(n, 0)
    for(int i=0; i<n; i++) cin V[i];
    cout << f(V, 0, k);
    return 0;
}
```

```
int fbu(vector<int> &heights, int k) {
    int n = heights.size();
    dp[n-1] = 0;
    for(int i = n-2; i>=0; i--){
        for(int j=1; j<=k; j++) {
            if(i+j >= n) break;
            dp[i] = min(dp[i], abs(heights[i] -
                                   heights[i+j])
                        + dp[i+j]);
        }
    }
    return dp[0];
}
```

Problem on Calculation: Coin Change (ESES)

Q⇒ Given ⇒ array ⇒ coins ⇒ diff currency. The task is to find the no. of ways to make sum by using different combinations from coins[]. Assume that ⇒ infinit supply of coins.

input1: sum = 4, coins[] = {1,2,3},

output 1: 4

Explanation there are 4 solutions

{1,1,1,1}, {1,1,2}, {2,2}, {1,3}.

```
int fbu(vector<int> &coins, int x, int n) {
    vector<int> dp(100005, 0);
    int MOD = 1000000007;
    dp[0] = 1;
    for(int j=0; j<n; j++) {
        //go to each coin
        for(int i=1; i<=x; i++){
            if(i-coins[j]<0) continue;
            dp[i] = (dp[i] % MOD +
                    dp[i-coins[j]] % MOD) % MOD;
        }
    }
    return dp[x];
}
int main() {
    int n;
    cin>>n;
    int x;
    vector<int>coins(n);
    for(int i=0; i<n; i++) cin>>coins[i];
    cout<<fbu(coins,x,n);
    return 0;
}
```