Easy Dynamic Programming

# Print first n Fibonacci Numbers

* If n = 0 then 0, if n = 1 then 1.
* Recursive Equation: F(n) = F(n-1) + F(n-2)

class Solution {

public:

vector<long long> printFibb(int n) {

vector<long long> dp(n,1);

if(n <= 2) return dp;

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

return dp;

}

};

# Count numbers containing 4

class Solution:

def countNumberswith4(self, N):

ans = 0

for i in range(4, N+1):

if '4' in str(i):

ans += 1

return ans

# Climbing Stairs

* You are climbing a staircase. It takes n steps to reach the top.
* Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

If you are in the ith stair, then you have 2 options, you can go to n+1 or n+2 stair. If you reach the last stair then you journey will be end and the path you followed to reach is one way to reach the destination. We can also restructure and say,

If n = 0 then will return 1.

F(n) = F(n-1) + F(n-2)

Recursive Solution

Time : O(2^n) and Space : O(n)

class Solution {

public:

int climbStairs(int n) {

if(n <= 1) return 1;

return climbStairs(n-1) + climbStairs(n-2);

}

};

class Solution {

public:

int climbStairs(int n) {

if(n <= 1) return 1;

int a = 1, b = 1;

for(int i = 2; i <= n; i++) {

int temp\_a = a;

a = b;

b = temp\_a + b;

}

return b;

}

};

# Frog Jump

#include <bits/stdc++.h>

int frogJump(int n, vector<int> &h){

if(n <= 1) return 0;

else if(n == 2) return abs(h[1] - h[0]);

return min(

abs(h[n-1] - h[n-2]) + frogJump(n-1, h),

abs(h[n-1] - h[n-3]) + frogJump(n-2, h)

);

}

This is recursion solution.

#include <bits/stdc++.h>

int frogJump(int n, vector<int> &h){

if(n <= 1) return 0;

else if(n == 2) return abs(h[1] - h[0]);

int a = 0, b = abs(h[1] - h[0]);

for(int i = 2; i < n; i++){

int temp\_a = a;

a = b;

b = min(

abs(h[i] - h[i-1]) + b,

abs(h[i] - h[i-2]) + temp\_a

);

}

return b;

}

# From Jump K Distance

// Online C++ compiler to run C++ program online

#include <bits/stdc++.h>

using namespace std;

class Solution{

int recFunc(vector<int> &arr, int n, int k) {

if(n <= 1) return 0;

int ans = INT\_MAX;

for(int i = k, j = n-1; i > 0 && j >= 1; i--, j--) {

ans = min(

ans,

recFunc(arr, j, k) + abs(arr[n-1] - arr[j-1])

);

}

return ans;

}

public:

int minJump(vector<int> arr, int n, int k) {

return recFunc(arr, n, k);

}

};

int main() {

Solution s;

cout << s.minJump({10,10} , 2, 100) << endl;

return 0;

}

# Maximum sum of non-adjacent elements

int recFunc(vector<int> &nums, int n){

vector<int> dp(n , 0);

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

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

dp[0] = nums[0];

dp[1] = max(nums[0] , nums[1]);

for(int i = 2 ; i < n ; i++)

dp[i] = max(

nums[i] + dp[i-2] , dp[i-1]

);

return dp[n-1];

}

int maximumNonAdjacentSum(vector<int> &nums){

return recFunc(nums , nums.size());

}

# House Robber

1. This is take or not-take type of problem. We have 2options to perform. If we take the nth value then can't consider n-1th value, have to consider n-2 value.

class Solution {

int recFunc(vector<int> &nums, int n) {

if(n <= 0) return 0;

cout << n << endl;

return max(nums[n-1] + recFunc(nums, n-2) , recFunc(nums, n-1));

}

public:

int rob(vector<int>& nums) {

return recFunc(nums, nums.size());

}

};

class Solution {

public:

int rob(vector<int>& nums) {

int n = nums.size();

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

int a = nums[0], b = max(nums[0], nums[1]);

for(int i = 2; i < n; i++) {

int temp = a;

a = b;

b = max(temp + nums[i], b);

}

return b;

}

};

# House Robber II

class Solution {

int recFunc(vector<int> &nums, int n) {

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

int a = nums[0], b = max(nums[0], nums[1]);

for(int i = 2; i < n; i++) {

int temp = a;

a = b;

b = max(temp + nums[i], b);

}

return b;

}

public:

int rob(vector<int>& nums) {

int n = nums.size();

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

vector<int> nums1 = nums, nums2 = nums;

nums1.erase(nums1.begin());

nums2.pop\_back();

int ans1 = recFunc(nums1, n-1);

int ans2 = recFunc(nums2, n-1);

return max(ans1, ans2);

}

};

1. We can't consider 1st and last house at the same time, so we have to split the array from arr[0…n-1] and arr[1…n].
2. Then we need to calculate maximum value for those two array
3. Then need to return max of two value.