# Dynamic Programming_1

@kbbhatt04

@September 27, 2023

## Climbing Stairs

Each time you can either climb `1` or `2` steps. In how many distinct ways can you climb to the top?

- Approach
  - Recursive Brute-force
    - Recursively count number of steps
    - F(n) = F(n-1) + F(n-2)
    - Time Complexity: $O(2^n)$
    - Space Complexity: $O(n)$ for function call stack
  - Memoization
    - Store each answer in array
    - When calculating next time, first check if it is already calculated
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$
  - Tabulation
    - Bottom-Up approach
    - We know the recurrence relation so we can fill table by looking at last 2 values i.e dp[n-1] and dp[n-2]
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$
  - Optimal

- We don't need whole table actually, we only need last two values
- Time Complexity: $O(n)$
- Space Complexity: $O(1)$

```
# Python3
# Recursive Brute-force Solution
class Solution:
    def fun(self, curr):
        if curr == 0:   return 1
        if curr == 1:   return 1

        return self.fun(curr-1) + self.fun(curr-2)

    def climbStairs(self, n: int) -> int:
        return self.fun(n)
```

```
# Python3
# Memoization Solution
class Solution:
    def fun(self, n, dp):
        if dp[n] != -1:
            return dp[n]
        dp[n] = self.fun(n-1, dp) + self.fun(n-2, dp)
        return dp[n]

    def climbStairs(self, n: int) -> int:
        dp = [-1 for _ in range(n+1)]
        dp[0] = 1
        dp[1] = 1
        self.fun(n, dp)
        return dp[n]
```

```python
# Python3
# Tabulation Solution
class Solution:
    def climbStairs(self, n: int) -> int:
        dp = [0 for _ in range(n+1)]
        dp[0] = 1
        dp[1] = 1

        for i in range(2, n+1):
            dp[i] = dp[i-1] + dp[i-2]
        return dp[n]
```

```python
# Python3
# Optimal Solution
class Solution:
    def climbStairs(self, n: int) -> int:
        if(n==1 or n==2 or n==3):
            return n
        a = 2
        b = 3
        for i in range(3, n):
            c = a + b
            a = b
            b = c
        return c
```

```cpp
// C++
// Optimal Solution
class Solution {
public:
    int climbStairs(int n) {
        if(n == 1 || n == 2 || n == 3)  return n;

        int a = 2, b = 3, c;
        for(int i = 3; i < n; i++)
        {
            c = a + b;
            a = b;
            b = c;
        }
        return c;
    }
};
```

## Frog Jump - 1

Given a number of stairs and a frog, the frog wants to climb from the 0th stair to the (N-1)th stair. At a time the frog can climb either one or two steps. A height[N] array is also given. Whenever the frog jumps from a stair i to stair j, the energy consumed in the jump is abs(height[i]- height[j]), where abs() means the absolute difference. We need to return the minimum energy that can be used by the frog to jump from stair 0 to stair N-1.

- Approach
    - Recursive Brute-force
        - Recursively try all possible combinations
        - Recurrence relation = `min(jump1, jump2)`
            - `jump1 = fun(heights, ind-1) + abs(heights[i] - heights[i-1])`
            - `jump2 = fun(heights, ind-2) + abs(heights[i] - heights[i-2])`
        - Time Complexity: $O(2^n)$
        - Space Complexity: $O(n)$
    - Memoization
        - Create a dp[n] array initialized to -1
        - Whenever we want to find the answer of a particular value (say n), we first check whether the answer is already calculated using the dp array (i.e dp[n] != -1 ). If yes, simply return the value from the dp array

        - If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set dp[n] to the solution we get
        - Time Complexity: $O(n)$
        - Space Complexity: $O(n)$
    - Tabulation
        - Declare a dp[] array of size n
        - First, initialize the base condition values, i.e dp[0] as 0
        - Set an iterative loop that traverses the array( from index 1 to n-1) and for every index calculate jumpOne and jumpTwo and set dp[i] = min(jumpOne, jumpTwo)
        - Time Complexity: $O(n)$
        - Space Complexity: $O(n)$
    - Optimal
        - If we closely look at the values required at every iteration, dp[i], dp[i-1], and  dp[i-2]
        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$

```
# Python3
# Recursive Brute-force Solution
def fun(heights, ind):
    if ind == 0:
        return 0

    jump1 = fun(heights, ind-1) + abs(heights[ind] - heights[in
    jump2 = float("inf")
    if ind > 1:
        jump2 = fun(heights, ind-2) + abs(heights[ind] - heights
    return min(jump1, jump2)

def frogJump(n: int, heights: List[int]) -> int:
    return fun(heights, n-1)
```

```
# Python3
# Memoization Solution
def fun(heights, ind, dp):
    if ind == 0:
        return 0

    if dp[ind] != -1:
        return dp[ind]

    jump1 = fun(heights, ind-1, dp) + abs(heights[ind] - heights
    jump2 = float("inf")
    if ind > 1:
        jump2 = fun(heights, ind-2, dp) + abs(heights[ind] - hei

    dp[ind] = min(jump1, jump2)
    return dp[ind]

def frogJump(n: int, heights: List[int]) -> int:
    dp = [-1 for _ in range(n)]
    return fun(heights, n-1, dp)
```

```
# Python3
# Tabulation Solution
def frogJump(n: int, heights: List[int]) -> int:
    dp = [-1 for _ in range(n)]
    dp[0] = 0

    for i in range(1, n):
        jump1 = dp[i-1] + abs(heights[i] - heights[i-1])
        jump2 = dp[i-2] + abs(heights[i] - heights[i-2]) if i >
        dp[i] = min(jump1, jump2)

    return dp[n-1]
```

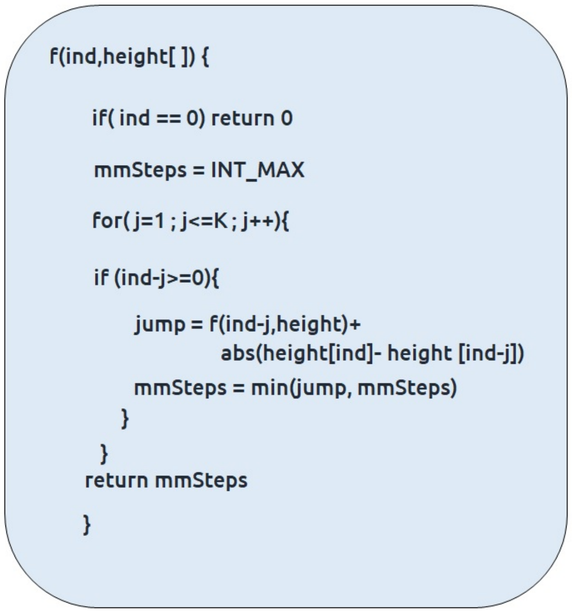
```
# Python3
# Optimal Solution
def frogJump(n: int, heights: List[int]) -> int:
    prev = 0
    prev2 = 0
    for i in range(1, n):
        jumpOne = prev + abs(heights[i] - heights[i - 1])
        jumpTwo = float("inf")
        if i > 1:
            jumpTwo = prev2 + abs(heights[i] - heights[i - 2])

        cur_i = min(jumpOne, jumpTwo)
        prev2 = prev
        prev = cur_i

    return prev
```

```
// C++
// Optimal Solution
int main() {

  vector<int> height{30,10,60,10,60,50};
  int n=height.size();
  int prev=0;
  int prev2=0;
  for(int i=1;i<n;i++){

      int jumpTwo = INT_MAX;
      int jumpOne= prev + abs(height[i]-height[i-1]);
      if(i>1)
        jumpTwo = prev2 + abs(height[i]-height[i-2]);

      int cur_i=min(jumpOne, jumpTwo);
      prev2=prev;
      prev=cur_i;

  }
  cout<<prev;
}
```

## Frog Jump - 2

The frog is allowed to jump up to ‘K’ steps at a time. If K=4, the frog can jump 1,2,3, or 4 steps at every index.

- Approach
    - Recursive Brute-force
        - Recursively check all the possible paths to reach the last stone ‘N’. For each path, calculate the total cost incurred by summing up the absolute differences in the heights of the stones occurring in the path. We then return the minimum cost path as the final answer.



```
f(ind,height[ ]) {

    if( ind == 0) return 0

    mmSteps = INT_MAX

    for( j=1 ; j<=K ; j++){

    if (ind-j>=0){

        jump = f(ind-j,height)+
                abs(height[ind]- height [ind-j])
        mmSteps = min(jump, mmSteps)
      }
    }
   return mmSteps

   }
```

        - Time Complexity: $O()$
        - Space Complexity: $O()$
    - Memoization
        - Create a dp[n] array initialized to -1
        - Whenever we want to find the answer of a particular value (say n), we first check whether the answer is already calculated using the dp array(i.e dp[n] != -1 ). If yes, simply return the value from the dp array
        - If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set dp[n] to the solution we get
        - Time Complexity: $O(n * k)$
        - Space Complexity: $O(n)$
    - Tabulation
        - Declare a dp[] array of size n with dp[0] = 0
        - Set an iterative loop which traverses the array( from index 1 to n-1) and for every index calculate and set dp[i] = min(min_steps, jump)
        - Time Complexity: $O(n * k)$
        - Space Complexity: $O(n)$

```
# Python3
# Recursive Brute-force Solution
def fun(k, heights, ind):
    if ind == 0:
        return 0

    min_steps = float("inf")
    for j in range(1, k+1):
        if ind-j >= 0:
            jump = fun(k, heights, ind-j) + abs(heights[ind] - h
            min_steps = min(min_steps, jump)
    return min_steps



def minimizeCost(n : int, k : int, heights : List[int]) -> int:
    return fun(k, heights, n-1)
```

```
# Python3
# Memoization Solution
def fun(k, heights, ind, dp):
    if ind == 0:
        return 0

    if dp[ind] != -1:
        return dp[ind]

    min_steps = float("inf")
    for j in range(1, k+1):
        if ind-j >= 0:
            jump = fun(k, heights, ind-j, dp) + abs(heights[ind]
            min_steps = min(min_steps, jump)
    dp[ind] = min_steps
    return dp[ind]



def minimizeCost(n : int, k : int, heights : List[int]) -> int:
    dp = [-1 for i in range(n)]
    dp[0] = 0
    return fun(k, heights, n-1, dp)
```

```
# Python3
# Tabulation Solution
def minimizeCost(n : int, k : int, heights : List[int]) -> int:
    dp = [-1 for _ in range(n)]
    dp[0] = 0
    for i in range(1, n):
        min_steps = float("inf")
        for j in range(1, k+1):
            if i - j >= 0:
                jump = dp[i-j] + abs(heights[i] - heights[i-j])
                min_steps = min(min_steps, jump)
        dp[i] = min_steps
    return dp[n-1]
```

```
// C++
// Tabulation Solution
int minimizeCost(int n, int k, vector<int> &height){
    vector<int> dp(n, -1);
    dp[0] = 0;
    for (int i = 1; i < n; i++) {
        int min_steps = INT_MAX;
        for (int j = 1; j <= k; j++) {
            if (i - j >= 0) {
                int jump = dp[i-j] + abs(height[i] - height[i-j
                min_steps = min(min_steps, jump);
            }
        }
        dp[i] = min_steps;
    }
    return dp[n-1];
}
```

## Maximum sum of non-adjacent elements

- Approach
    - Recursive Brute-force
        - To generate all the subsequences, we can use the pick/non-pick technique

        - If we pick an element then, pick = arr[ind] + f(ind-2). The reason we are doing f(ind-2) is because we have picked the current index element so we need to pick a non-adjacent element so we choose the index ‘ind-2’ instead of ‘ind-1’
        - Next we need to ignore the current element in our subsequence. So nonPick= 0 + f(ind-1). As we don’t pick the current element, we can consider the adjacent element in the subsequence
        - Time Complexity: $O(2^n)$
        - Space Complexity: $O(n)$
    - Memoization
        - Store these intermediate results in dp array
        - Time Complexity: $O(n)$
        - Space Complexity: $O(n)$
    - Tabulation
        - pick = arr[i] + dp[i-2]
        - not_pick = dp[i-1]
        - Set dp[i] = max(pick, not_pick)
        - Time Complexity: $O(n)$
        - Space Complexity: $O(n)$
    - Optimal
        - We do need only the last two values in the array
        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
def fun(nums, ind):
    if ind >= len(nums):
        return 0

    pick = fun(nums, ind+2) + nums[ind]
    not_pick = fun(nums, ind+1) + 0

    return max(pick, not_pick)

def maximumNonAdjacentSum(nums):
    return fun(nums, 0)
```

```
# Python3
# Memoization Solution
def solveUtil(ind, arr, dp):
    # Check if the solution for this index has already been calc
    if dp[ind] != -1:
        return dp[ind]

    # Base case: when the index is 0, return the value at that i
    if ind == 0:
        return arr[ind]

    # Base case: when the index is negative, return 0 (out of bo
    if ind < 0:
        return 0

    # Calculate the maximum value when picking the current eleme
    pick = arr[ind] + solveUtil(ind - 2, arr, dp)

    # Calculate the maximum value when not picking the current e
    nonPick = 0 + solveUtil(ind - 1, arr, dp)

    # Store the maximum of the two choices in the DP table
    dp[ind] = max(pick, nonPick)

    # Return the maximum value for the current index
    return dp[ind]



def maximumNonAdjacentSum(nums):
    dp = [-1 for i in range(n)]

    # Call the recursive utility function to find the maximum va
    return solveUtil(n - 1, arr, dp)
```

```
# Python3
# Tabulation Solution
def solveUtil(n, arr, dp):
    # Initialize the 0th element of the DP table with the 0th el
    dp[0] = arr[0]

    # Loop through the array starting from the second element
    for i in range(1, n):
        # Calculate the maximum value when picking the current e
        pick = arr[i]

        # Check if there are at least two elements before the cu
        if i > 1:
            pick += dp[i - 2]

        # Calculate the maximum value when not picking the curre
        non_pick = 0 + dp[i - 1]

        # Store the maximum of the two choices in the DP table
        dp[i] = max(pick, non_pick)

    # Return the maximum value for the last index
    return dp[n - 1]

# Function to solve the problem for the given array
def maximumNonAdjacentSum(nums):
    # Initialize a DP table with -1 values to store intermediate
    dp = [-1 for _ in range(len(nums))]
    return solveUtil(len(nums), nums, dp)
```

```
# Python3
# Optimal Solution
def maximumNonAdjacentSum(arr):
    n = len(arr)
    prev = arr[0]  # Initialize with the first element of the ar
    prev2 = 0      # Initialize with 0 because there is no eleme

    # Loop through the array starting from the second element
    for i in range(1, n):
        # Calculate the maximum value when picking the current e
        pick = arr[i]

        # Check if there are at least two elements before the cu
        if i > 1:
            pick += prev2

        # Calculate the maximum value when not picking the curre
        non_pick = 0 + prev

        # Calculate the maximum value for the current index
        cur_i = max(pick, non_pick)

        # Update the 'prev' and 'prev2' variables for the next i
        prev2 = prev
        prev = cur_i

    # Return the maximum value for the last index, which represe
    return prev
```

```
// C++
// Optimal Solution
int solve(int n, vector<int>& arr) {
    int prev = arr[0];   // Initialize the maximum sum ending at
    int prev2 = 0;       // Initialize the maximum sum ending tw

    for (int i = 1; i < n; i++) {
        int pick = arr[i];  // Maximum sum if we pick the curren
        if (i > 1)
            pick += prev2;  // Add the maximum sum two elements

        int nonPick = 0 + prev;  // Maximum sum if we don't pick

        int cur_i = max(pick, nonPick);  // Maximum sum ending a
        prev2 = prev;   // Update the maximum sum two elements a
        prev = cur_i;   // Update the maximum sum ending at the
    }

    return prev;  // Return the maximum sum
}
```

## House Robber - 2

All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one.

- Approach
    - Make two reduced arrays, arr1(arr-last element) and arr2(arr-first element)
    - Apply same logic as Maximum Sum of non-Adjacent Elements (above question)

```
# Python3
# Recursive Brute-force Solution
class Solution:
    def fun(self, nums, idx):
        if idx >= len(nums):
            return 0

        pick = nums[idx] + self.fun(nums, idx+2)
        not_pick = 0 + self.fun(nums, idx+1)

        return max(pick, not_pick)

    def rob(self, nums: List[int]) -> int:
        if len(nums) == 1:  return nums[0]
        return max(self.fun(nums[:-1], 0), self.fun(nums[1:], 0
```

```
# Python3
# Memoization Solution
class Solution:
    def fun(self, nums, idx, dp):
        if idx >= len(nums):
            return 0

        if dp[idx] != -1:
            return dp[idx]

        pick = nums[idx] + self.fun(nums, idx+2, dp)
        not_pick = 0 + self.fun(nums, idx+1, dp)

        dp[idx] = max(pick, not_pick)
        return dp[idx]

    def rob(self, nums: List[int]) -> int:
        if len(nums) == 1:  return nums[0]
        dp1 = [-1 for _ in range(len(nums)-1)]
        self.fun(nums[:-1], 0, dp1)
        dp2 = [-1 for _ in range(len(nums)-1)]
        self.fun(nums[1:], 0, dp2)
        return max(dp1[0], dp2[0])
```

```
# Python3
# Tabulation Solution
class Solution:
    def rob(self, nums: List[int]) -> int:
        if len(nums) == 1:  return nums[0]

        nums1 = nums[:-1]
        nums2 = nums[1:]

        dp1 = [-1 for i in range(len(nums1))]
        dp2 = [-1 for i in range(len(nums2))]

        dp1[0] = nums1[0]
        dp2[0] = nums2[0]

        for i in range(1, len(nums1)):
            pick = nums1[i]
            if i != 1:
                pick += dp1[i-2]
            not_pick = dp1[i-1]
            dp1[i] = max(pick, not_pick)

        for i in range(1, len(nums2)):
            pick = nums2[i]
            if i != 1:
                pick += dp2[i-2]
            not_pick = dp2[i-1]
            dp2[i] = max(pick, not_pick)

        return max(dp1[-1], dp2[-1])
```

```python
# Python3
# Optimal Solution
def solve(arr):
    n = len(arr)
    prev = arr[0]
    prev2 = 0

    for i in range(1, n):
        pick = arr[i]
        if i > 1:
            pick += prev2
        nonPick = 0 + prev

        cur_i = max(pick, nonPick)
        prev2 = prev
        prev = cur_i

    return prev

def robStreet(n, arr):
    arr1 = []
    arr2 = []

    if n == 1:
        return arr[0]

    for i in range(n):
        if i != 0:
            arr1.append(arr[i])
        if i != n - 1:
            arr2.append(arr[i])

    ans1 = solve(arr1)
    ans2 = solve(arr2)
    return max(ans1, ans2)
```

```
// C++
// Optimal Solution
long long int solve(vector<int>& arr){
    int n = arr.size();
    long long int prev = arr[0];
    long long int prev2 =0;

    for(int i=1; i<n; i++){
        long long int pick = arr[i];
        if(i>1)
            pick += prev2;
        int long long nonPick = 0 + prev;

        long long int cur_i = max(pick, nonPick);
        prev2 = prev;
        prev= cur_i;

    }
    return prev;
}

long long int robStreet(int n, vector<int> &arr){
    vector<int> arr1;
    vector<int> arr2;

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

    for(int i=0; i<n; i++){

        if(i!=0) arr1.push_back(arr[i]);
        if(i!=n-1) arr2.push_back(arr[i]);
    }

    long long int ans1 = solve(arr1);
    long long int ans2 = solve(arr2);

    return max(ans1,ans2);
}
```

---

## Template

- Approach
  - Brute-force
    - 
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Better
    - 
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Optimal
    - 
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
# Python3
# Better Solution
```

```
# Python3
# Optimal Solution
```

```
// C++
// Optimal Solution
```