



Experiment- 6A

Student Name: Om Ankur Prajapati

UID: 22BCS15926

Branch: BE-CSE

Section/Group: 602-A

Semester: 6th

Date of Performance: 20/02/25

Subject Name: AP Lab-2

Subject Code: 22CSH-352

1. TITLE:

Climbing Stairs.

2. AIM:

You are climbing a staircase. It takes n steps to reach the top.

3. Algorithm

- **Define a DP array** dp where $dp[i]$ represents the number of distinct ways to reach the i -th stair.
- Initialize base cases:
 $dp[0] = 1 \rightarrow$ There is 1 way to stay at the ground without climbing.
 $dp[1] = 1 \rightarrow$ There is 1 way to reach the first stair (taking a single step).
- **Iterate from $i = 2$ to n** and use the recurrence relation:
 $dp[i] = dp[i-1] + dp[i-2]$
- **Return $dp[n]$** , which contains the total number of ways to reach the n -th stair.

Implementation/Code

```
class Solution {  
public:  
    int climbStairs(int n) {  
        // dp[i] := the number of ways to climb to the i-th stair  
        vector<int> dp(n + 1);  
        dp[0] = 1;  
        dp[1] = 1;  
  
        for (int i = 2; i <= n; ++i)  
            dp[i] = dp[i - 1] + dp[i - 2];  
    }  
};
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
return dp[n];  
}  
};
```

Output:

← All Submissions

Accepted 45 / 45 testcases passed

roshanKumar864 submitted at Feb 24, 2025 10:55

Editorial

Solution



Time Complexity : $O(n)$

Space Complexity : $O(n)$

Learning Outcomes:-

- The given solution is a **bottom-up DP** approach, where smaller subproblems are solved first.
- This makes it an ideal candidate for **DP** rather than a naive recursive approach (which has exponential complexity).



Experiment - 6B

Student Name: Roshan Kumar

UID: 22BCS16490

Branch: BE-CSE

Section/Group: NTPP- 602-A

Semester: 6th

Date of Performance: 20/02/25

Subject Name: AP Lab-2

Subject Code: 22CSH-352

1. TITLE:

Maximum Subarray.

2. AIM:

Given an integer array `nums`, find the `subarray` with the largest sum, and return *its sum*.

3. Algorithm

- Define DP array `dp[i]`, where:

`dp[i]` represents the **maximum sum subarray that ends at index `i`**.

This ensures that every subarray considered **includes `nums[i]`**

- Base Case:
`dp[0] = nums[0]` → The maximum sum subarray ending at index 0 is the element itself.
- State Transition (Recurrence Relation):
For each `i` from 1 to `n-1`, compute:
$$dp[i] = \max(nums[i], dp[i-1] + nums[i])$$
- Final Answer:
The overall maximum sum subarray is obtained by computing: $\max(dp[0], dp[1], \dots, dp[n-1])$

Implementation/Code:

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        // dp[i] := the maximum sum subarray ending in i
        vector<int> dp(nums.size());

        dp[0] = nums[0];
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
for (int i = 1; i < nums.size(); ++i)
    dp[i] = max(nums[i], dp[i - 1] + nums[i]);

return ranges::max(dp);
}
```

Output:

← All Submissions

Accepted 210 / 210 testcases passed

roshanKumar864 submitted at Feb 24, 2025 10:56

Editorial

Solution

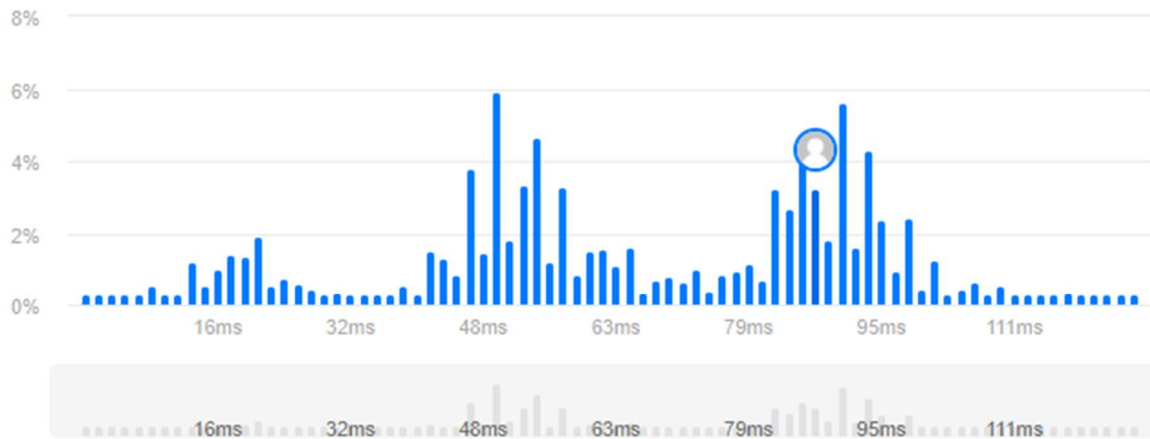
Runtime

87 ms | Beats 36.94%

Analyze Complexity

Memory

32.61 MB | Beats 45.57%



Code | Python3

Time Complexity : $O(N)$

Space Complexity : $O(1)$

Learning Outcomes:-

- Recognizing **overlapping subproblems** (each subarray solution builds on the previous).
- The optimized approach shows the **power of greedy techniques** in reducing space complexity.



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.