# Experiment 7

**Student Name:** Tarun Baliyan **UID:** 22BET10184

**Branch:** IT **Section/Group:** 22BET\_IOT-702/A

**Semester:** 6th **Date of Performance:** 07/03/25

**Subject Name:** Advance Programming-II **Subject Code:** 22ITP-367

**Problem: 1.6.1:** Climbing Stairs

**Problem Statement:** The problem aims to determine the number of distinct ways to reach the top of a staircase when a person can climb 1 or 2 steps at a time. It is a classic **dynamic programming** problem that helps in understanding recursion, memoization, and iterative solutions.

1. **Objective:** Understand the mathematical pattern behind the problem (**Fibonacci sequence**).
2. **Code:**

## class Solution {

## public int climbStairs(int n) {

## if (n <= 2) return n;

## int first = 1, second = 2;

## 

## for (int i = 3; i <= n; i++) {

## int temp = first + second;

## first = second;

## second = temp;

## }

## 

## return second;

## }

## }

## 3. Result:

## 

**Problem 1.6.2: Maximum Subarray**

**Problem Statement:** To find the contiguous subarray with the **maximum sum** within a given integer array, using efficient algorithms like **Kadane’s Algorithm**.

1. **Objective:** Understand **brute force**, **prefix sum**, and **Kadane’s algorithm**.
2. **Code:**

class Solution {

public int maxSubArray(int[] nums) {

int maxSum = nums[0];

int currentSum = nums[0];

for (int i = 1; i < nums.length; i++) {

currentSum = Math.max(nums[i], currentSum + nums[i]);

maxSum = Math.max(maxSum, currentSum);

}

return maxSum;

}

}

1. **Result:**

## 

**Problem 1.4.3: Kth Largest Element in an Array**

**Problem 1.6.3: House Robber**

**Problem Statement:** To find the **maximum amount of money** that can be robbed from a row of houses **without robbing two adjacent houses**.

1. **Objective:** nderstand the **dynamic programming (DP) approach** for solving the problem efficiently.
2. **Code:**

class Solution {

public int rob(int[] nums) {

if (nums.length == 0) return 0;

if (nums.length == 1) return nums[0];

int[] dp = new int[nums.length];

dp[0] = nums[0];

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

for (int i = 2; i < nums.length; i++) {

dp[i] = Math.max(dp[i - 1], nums[i] + dp[i - 2]);

}

return dp[nums.length - 1];

}

}};

1. **Result:**

## 

**Problem 1.6.4:** Jump Game

**Problem Statement:** To determine whether it is possible to reach the last index from the first index in an array where each element represents the **maximum jump length** from that position.

1. **Objective:** Learn to optimize from **exponential time** to **linear time** using greedy methods.
2. **Code:**

class Solution {

public boolean canJump(int[] nums) {

int lastPos = nums.length - 1;

for (int i = nums.length - 2; i >= 0; i--) {

if (i + nums[i] >= lastPos) {

lastPos = i;

}

}

return lastPos == 0;

}

}

1. **Result:**

## 

**Problem 1.6.5: Unique Paths**

**Problem Statement:** To find the number of unique paths from the **top-left** corner to the **bottom-right** corner of an **m × n** grid, moving **only right or down**.

1. **Objective:** Optimize from **exponential time** to **constant time** using **mathematical formulas**
2. **Code:**

class Solution {

private int[][] memo;

public int uniquePaths(int m, int n) {

memo = new int[m][n];

return dfs(0, 0, m, n);

}

private int dfs(int i, int j, int m, int n) {

if (i == m - 1 && j == n - 1) return 1;

if (i >= m || j >= n) return 0;

if (memo[i][j] != 0) return memo[i][j]; // Use cached result

memo[i][j] = dfs(i + 1, j, m, n) + dfs(i, j + 1, m, n);

return memo[i][j];

}

}

1. **Result:**

## 