Course Name:	**Competitive Programming Laboratory (216U01L401)**	**Semester:**	**IV**
Date of Performance:	**__09_ / _04__ / __25____**	**DIV/ Batch No:**	**E2**
Student Name:	**Shreyans Tatiya**	**Roll No:**	**16010123325**

Experiment No: 5

Title: To implement a competitive programming problem on a platform (eg. Leetcode) optimised using the dynamic programming approach.

Aim and Objective of the Experiment:

1. **Understand** the concepts of dynamic programming approach.
2. **Apply** the concepts to solve the problem.
3. **Implement** the solution to given problem statement.
4. **Create** test cases for testing the solution.
5. **Analyze** the result for efficiency of the solution.

COs to be achieved:

CO3: Solve intricate problems involving graphs, tree structures, and algorithms.

Books/ Journals/ Websites referred:

1. https://leetcode.com/problems/burst-balloons/
2. Competitive Programmer's Handbook (CPH) - Antti Laaksonen

Theory:

Dynamic Programming (DP) is an efficient problem-solving technique used for problems with overlapping subproblems and optimal substructure. It works by breaking down a complex problem into simpler subproblems, solving each subproblem once, and storing their results to avoid redundant computations. To apply DP, we first understand the problem and identify the subproblems and how they relate through a recurrence relation. We then implement the solution using either a top-down (with memoization) or bottom-up (with tabulation) approach, ensuring base cases are handled correctly. After implementation, we create a variety of test cases—including normal, edge, and large inputs—to verify the accuracy and robustness of the solution. Finally, we analyze the time and space complexity to evaluate the efficiency of our approach compared to brute-force methods, often observing significant performance improvements.

Dynamic Programming is widely used in optimization problems such as shortest paths, knapsack, matrix chain multiplication, and many more. It helps in reducing exponential time complexities to polynomial time by avoiding repeated calculations. The key to solving any DP problem lies in clearly defining the state variables, formulating the recurrence relation, and choosing the right

storage structure to hold intermediate results. While implementing, attention must be given to edge conditions and ensuring that all subproblems are correctly evaluated. Through systematic testing and performance analysis, we can ensure that the DP solution is not only correct but also highly efficient and scalable for large inputs.

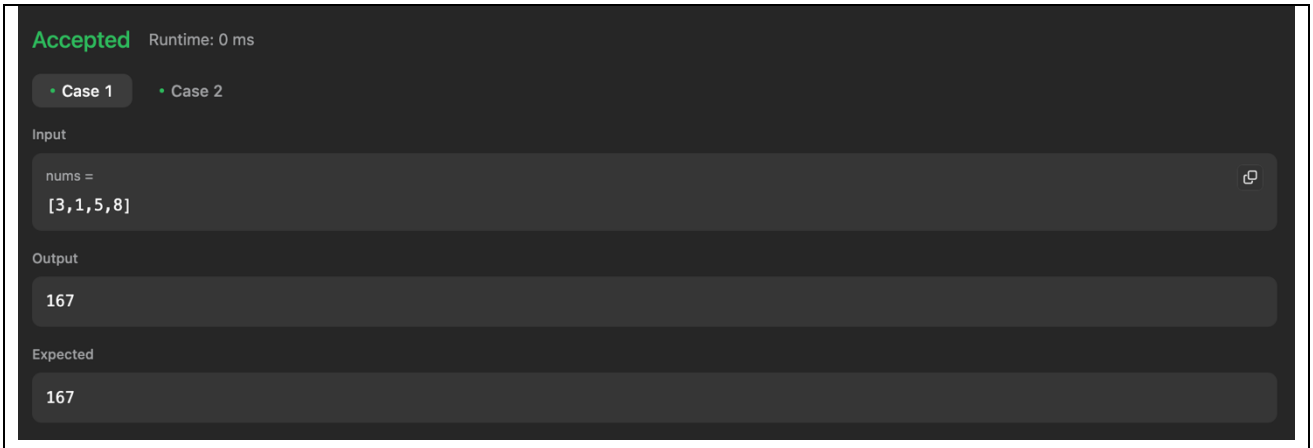
Problem statement

You are given n balloons, indexed from 0 to n - 1. Each balloon is painted with a number on it represented by an array nums. You are asked to burst all the balloons.
If you burst the ith balloon, you will get nums[i - 1] * nums[i] * nums[i + 1] coins.
If i - 1 or i + 1 goes out of bounds of the array, then treat it as if there is a balloon with a 1 painted on it.
Return the maximum coins you can collect by bursting the balloons wisely.

Code :

```
class Solution {
public:
    int maxCoins(vector<int>& nums) {
        //including the nums[-1] and nums[n]
        int n = nums.size() + 2;
        vector<vector<int>> dp(n, vector<int>(n));
        vector<int> new_nums(n, 1);
        int i = 1;
        for(auto num : nums) {
            new_nums[i++] = num;
        }
        for(int len = 2; len <= n; len++) {
            //iterate from interval length from 2 to n
            for(int i = 0; i <= n - len; i++) {
                int j = i + len - 1;
                //select between left and right boundary (i, j)
                for(int k = i + 1; k < j; k++) {
                    dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + new_nums[i] * new_nums[k] * new_nums[j]);
                }
            }
        }
        return dp[0][n - 1];
    }
};
```

Output:



Post Lab Subjective/Objective type Questions:

1. What is the role of Recurrence Relation in dynamic Programming? What do you mean by State Space Tree?

Role of Recurrence Relation in Dynamic Programming:

It defines how a problem can be broken into smaller subproblems.
Helps build the solution using previously solved subproblems.

Example:
Fibonacci: F(n) = F(n-1) + F(n-2)

State Space Tree:

It is a tree structure that shows all possible states or decisions of a problem.
Used to explore subproblems in recursion or backtracking.

Example:
In 0/1 Knapsack, each node shows a choice: include or exclude an item.

Conclusion:

The above experiment highlights usage of dynamic programming and application of chain matrix multiplication in the above problem statement.