VERY EASY

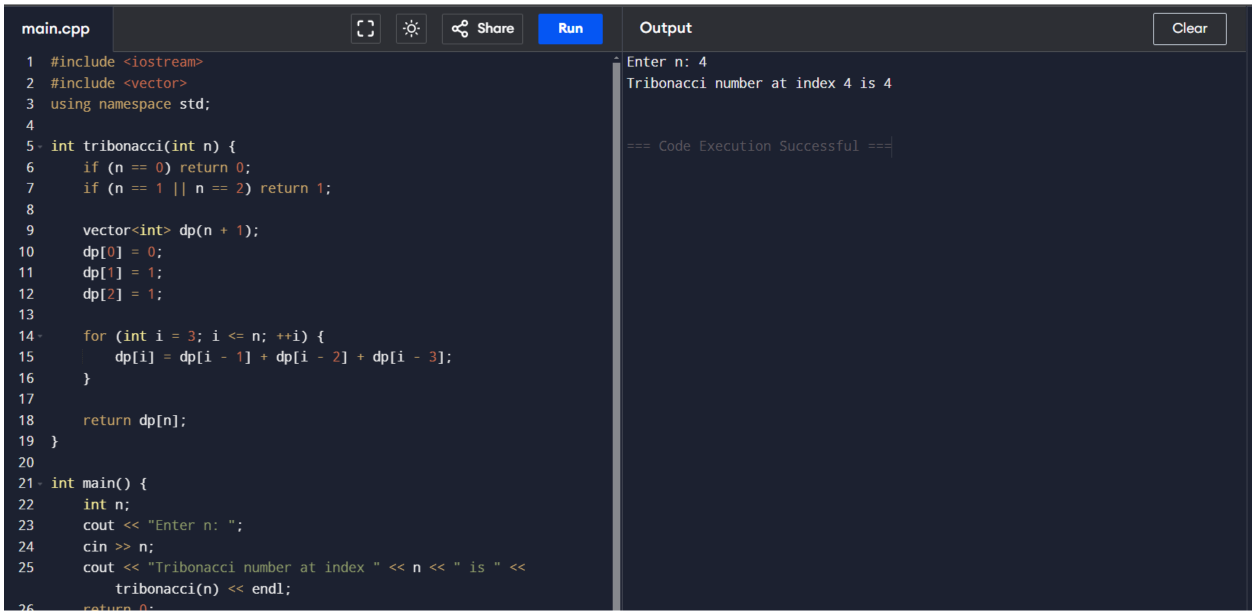
N-th Tribonacci Number

The Tribonacci sequence Tn is defined as follows:
T0 = 0, T1 = 1, T2 = 1, and Tn+3 = Tn + Tn+1 + Tn+2 for n >= 0.
Given n, return the value of Tn.
Example 1:
Input: n = 4
Output: 4
Explanation:
T_3 = 0 + 1 + 1 = 2
T_4 = 1 + 1 + 2 = 4

Code:

```
#include <iostream>
#include <vector>
using namespace std;

int tribonacci(int n) {
    if (n == 0) return 0;
    if (n == 1 || n == 2) return 1;
    vector<int> dp(n + 1);
    dp[0] = 0;
    dp[1] = 1;
    dp[2] = 1;
    for (int i = 3; i <= n; ++i) {
        dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3];
    }

    return dp[n];
}
int main() {
    int n;
    cout << "Enter n: ";
    cin >> n;
    cout << "Tribonacci number at index " << n << " is " << tribonacci(n) << endl;
    return 0;
}
```

Output:



```
#include <iostream>
#include <vector>
using namespace std;

int tribonacci(int n) {
    if (n == 0) return 0;
    if (n == 1 || n == 2) return 1;

    vector<int> dp(n + 1);
    dp[0] = 0;
    dp[1] = 1;
    dp[2] = 1;

    for (int i = 3; i <= n; ++i) {
        dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3];
    }

    return dp[n];
}

int main() {
    int n;
    cout << "Enter n: ";
    cin >> n;
    cout << "Tribonacci number at index " << n << " is " <<
        tribonacci(n) << endl;
```

```
Enter n: 4
Tribonacci number at index 4 is 4


=== Code Execution Successful ===
```

2. Divisor Game

Alice and Bob take turns playing a game, with Alice starting first. Initially, there is a number n on the chalkboard. On each player's turn, that player makes a move consisting of:

Choosing any x with 0 < x < n and n % x == 0.
Replacing the number n on the chalkboard with n - x.
Also, if a player cannot make a move, they lose the game.

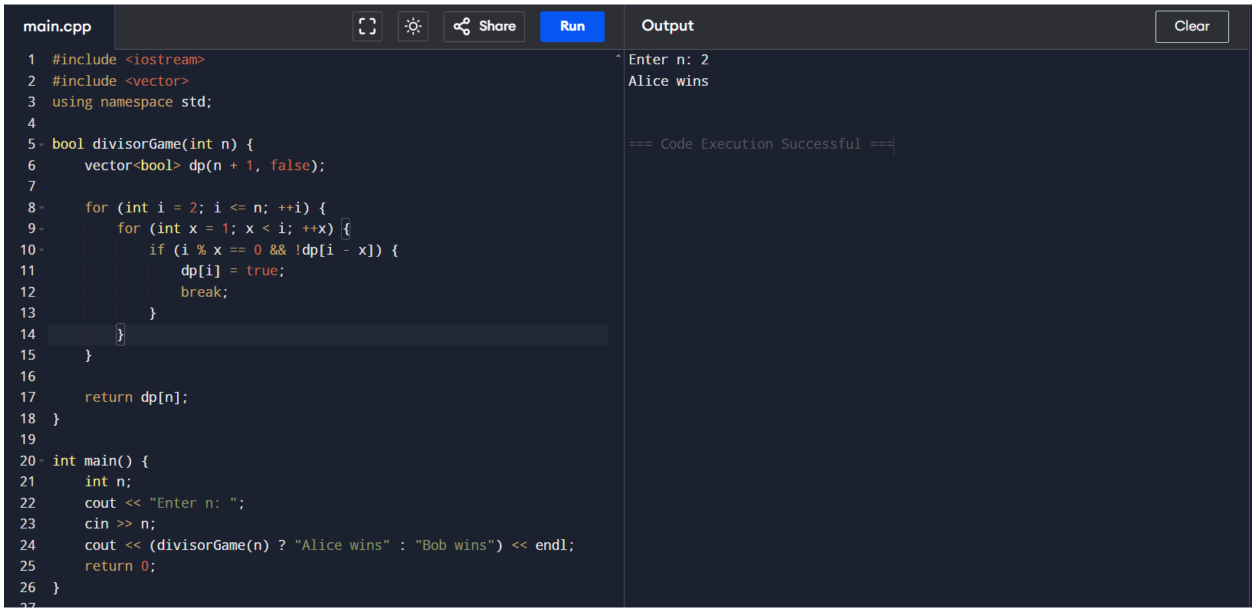
Return true if and only if Alice wins the game, assuming both players play optimally.
Example 1:

Input: n = 2
Output: true
Explanation: Alice chooses 1, and Bob has no more moves.

Code:

```
#include <iostream>

#include <vector>

using namespace std;


bool divisorGame(int n) {

    vector<bool> dp(n + 1, false);


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

        for (int x = 1; x < i; ++x) {

            if (i % x == 0 && !dp[i - x]) {

                dp[i] = true;

                break;

            }

        }

    }


    return dp[n];

}

int main() {

    int n;

    cout << "Enter n: ";

    cin >> n;

    cout << (divisorGame(n) ? "Alice wins" : "Bob wins") << endl;

    return 0;

}
```

OUTPUT:



```
#include <iostream>
#include <vector>
using namespace std;

bool divisorGame(int n) {
    vector<bool> dp(n + 1, false);

    for (int i = 2; i <= n; ++i) {
        for (int x = 1; x < i; ++x) {
            if (i % x == 0 && !dp[i - x]) {
                dp[i] = true;
                break;
            }
        }
    }

    return dp[n];
}

int main() {
    int n;
    cout << "Enter n: ";
    cin >> n;
    cout << (divisorGame(n) ? "Alice wins" : "Bob wins") << endl;
    return 0;
}
```

```
Enter n: 2
Alice wins


=== Code Execution Successful ===
```

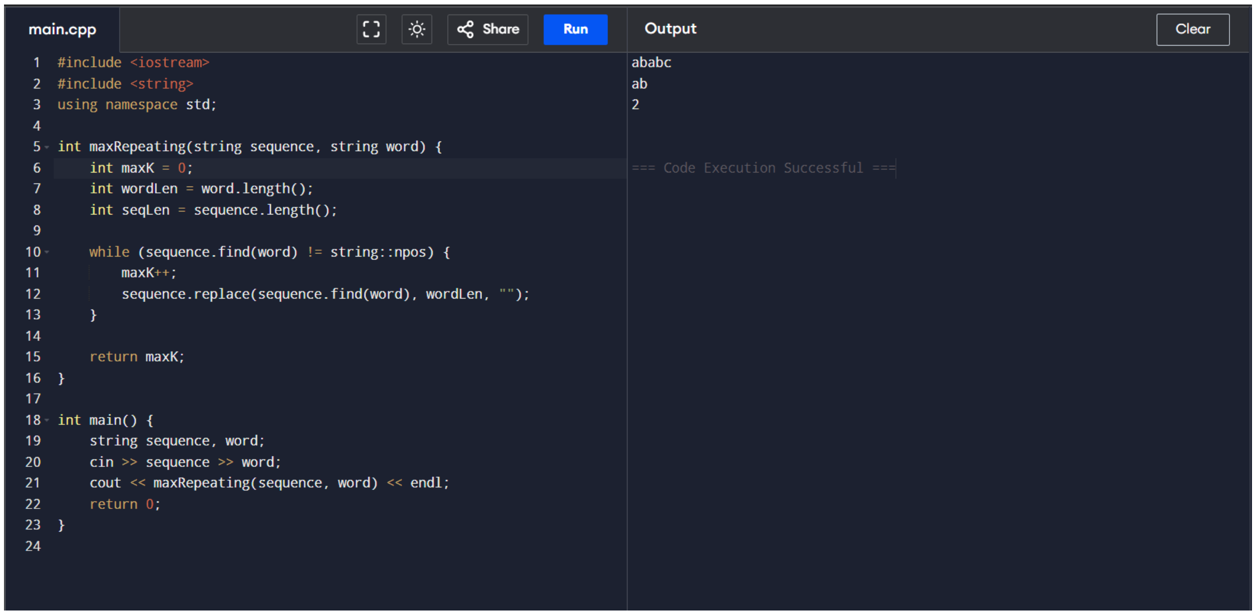
3. Maximum Repeating Substring

For a string sequence, a string word is k-repeating if word concatenated k times is a substring of sequence. The word's maximum k-repeating value is the highest value k where word is k-repeating in sequence. If word is not a substring of sequence, word's maximum k-repeating value is 0.
Given strings sequence and word, return the maximum k-repeating value of word in sequence.
Example 1:
Input: sequence = "ababc", word = "ab"
Output: 2

Code:

```
#include <iostream>

#include <string>

using namespace std;


int maxRepeating(string sequence, string word) {

    int maxK = 0;
    int wordLen = word.length();
    int seqLen = sequence.length();
    while (sequence.find(word) != string::npos) {
        maxK++;
        sequence.replace(sequence.find(word), wordLen, "");
    }
    return maxK ; }
int main() {
    string sequence, word;
    cin >> sequence >> word;
    cout << maxRepeating(sequence, word) << endl;
    return 0;
}
```

OUTPUT:



main.cpp

```
#include <iostream>
#include <string>
using namespace std;

int maxRepeating(string sequence, string word) {
    int maxK = 0;
    int wordLen = word.length();
    int seqLen = sequence.length();

    while (sequence.find(word) != string::npos) {
        maxK++;
        sequence.replace(sequence.find(word), wordLen, "");
    }

    return maxK;
}

int main() {
    string sequence, word;
    cin >> sequence >> word;
    cout << maxRepeating(sequence, word) << endl;
    return 0;
}
```

Output

```
ababc
ab
2


=== Code Execution Successful ===
```

Easy:

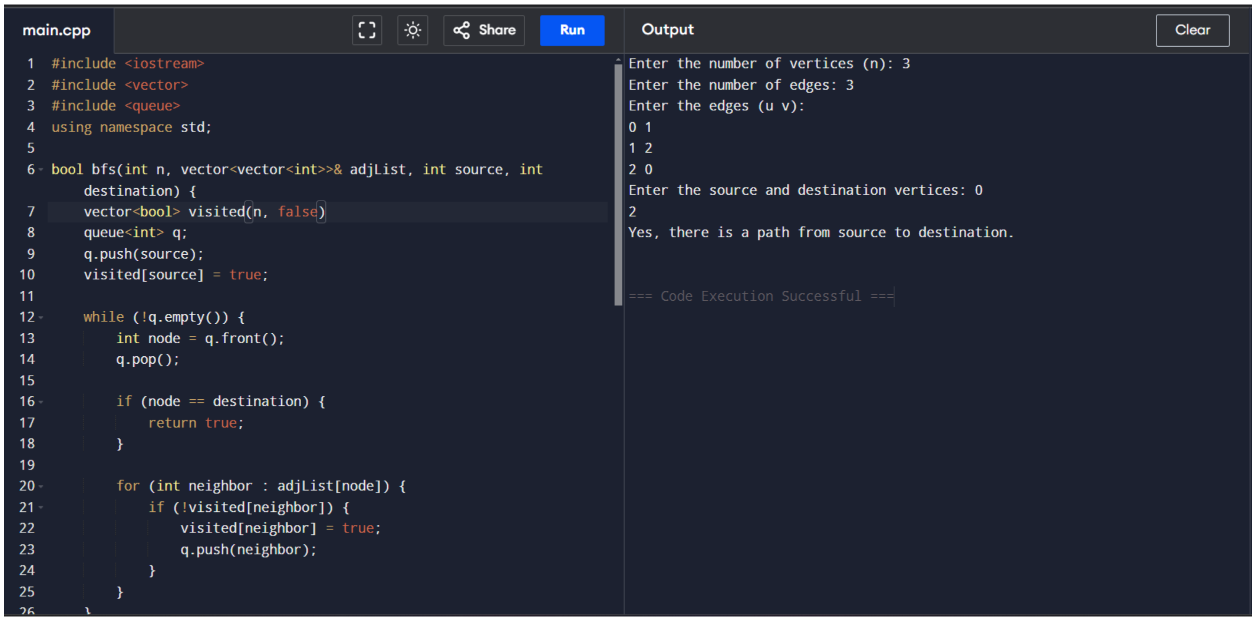
1. Climbing Stairs

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?
Example 1: Input: n = 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps

CODE:

```
#include <iostream>

#include <vector>

using namespace std;


int climbStairs(int n) {

    if (n <= 2) return n;

    vector<int> dp(n + 1);

    dp[1] = 1;

    dp[2] = 2;


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

        dp[i] = dp[i - 1] + dp[i - 2];

    }


    return dp[n];

}


int main() {

    int n;
    cin >> n;

    cout << "ways of climbing stairs are : " << climbStairs(n) << endl;

    return 0;

}
```

OUTPUT:



```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

bool bfs(int n, vector<vector<int>>& adjList, int source, int
    destination) {
    vector<bool> visited(n, false);
    queue<int> q;
    q.push(source);
    visited[source] = true;

    while (!q.empty()) {
        int node = q.front();
        q.pop();

        if (node == destination) {
            return true;
        }

        for (int neighbor : adjList[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
```

```
Enter the number of vertices (n): 3
Enter the number of edges: 3
Enter the edges (u v):
0 1
1 2
2 0
Enter the source and destination vertices: 0
2
Yes, there is a path from source to destination.


=== Code Execution Successful ===
```

2. Best Time to Buy and Sell Stock

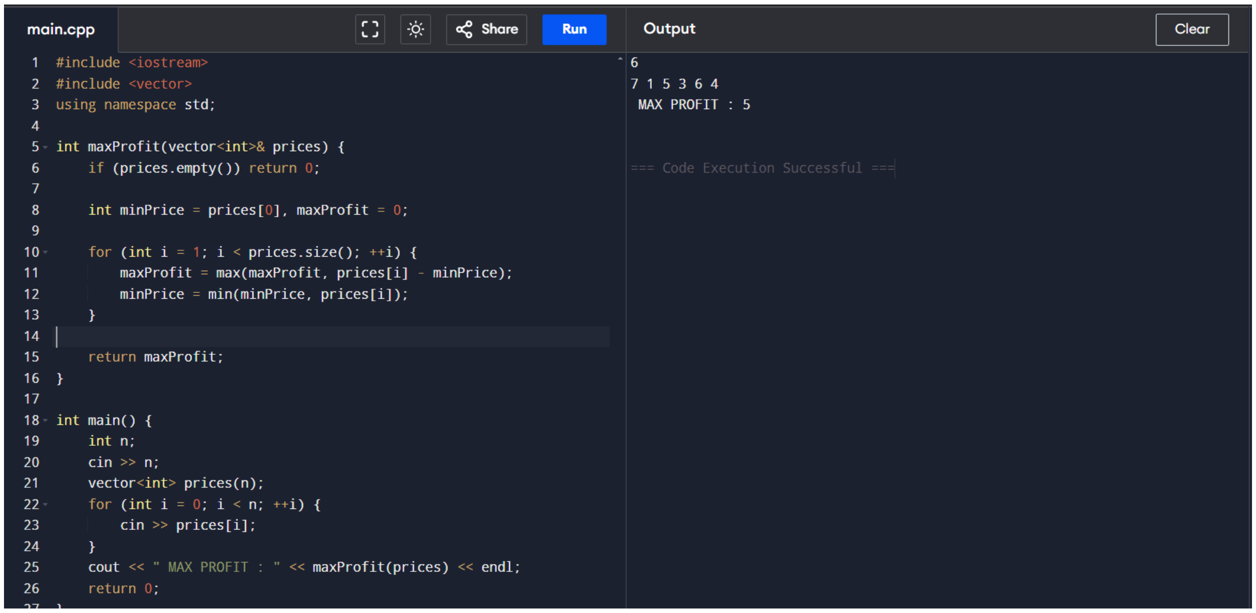
You are given an array prices where prices[i] is the price of a given stock on the ith day.You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.
Example 1:Input: prices = [7,1,5,3,6,4]
Output: 5

CODE:

```
#include <iostream>

#include <vector>

#include <climits>

using namespace std;
```

```
int maxProfit(vector<int>& prices) {
    int minPrice = INT_MAX, maxProfit = 0;
    for (int price : prices) {
        minPrice = min(minPrice, price);
        maxProfit = max(maxProfit, price - minPrice);
    }
    return maxProfit;
}
int main() {
    int n;
    cin >> n;
    vector<int> prices(n);
    for (int i = 0; i < n; ++i) cin >> prices[i];
    cout << maxProfit(prices) << endl;
    return 0;
}
```

OUTPUT:



```
#include <iostream>
#include <vector>
using namespace std;

int maxProfit(vector<int>& prices) {
    if (prices.empty()) return 0;

    int minPrice = prices[0], maxProfit = 0;

    for (int i = 1; i < prices.size(); ++i) {
        maxProfit = max(maxProfit, prices[i] - minPrice);
        minPrice = min(minPrice, prices[i]);
    }

    return maxProfit;
}

int main() {
    int n;
    cin >> n;
    vector<int> prices(n);
    for (int i = 0; i < n; ++i) {
        cin >> prices[i];
    }
    cout << " MAX PROFIT : " << maxProfit(prices) << endl;
    return 0;
```

```
6
7 1 5 3 6 4
 MAX PROFIT : 5


=== Code Execution Successful ===
```

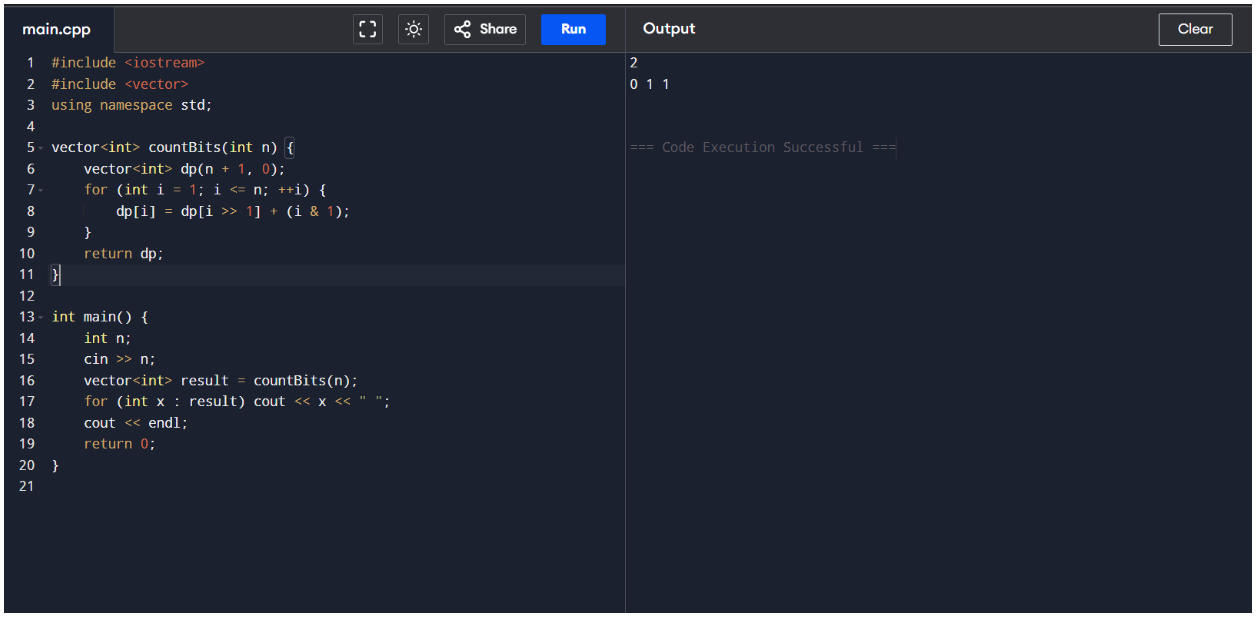
3. Counting Bits

Given an integer n, return an array ans of length n + 1 such that for each i (0 <= i <= n), ans[i] is the number of 1's in the binary representation of i.
Example 1:Input: n = 2 | Output: [0,1,1]
Explanation:0 --> 0
1 --> 1
2 --> 10

CODE:

```
#include <iostream>
#include <vector>
using namespace std;


vector<int> countBits(int n) {
    vector<int> dp(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        dp[i] = dp[i >> 1] + (i & 1);
    }
    return dp;
}


int main() {
    int n;
    cin >> n;
    vector<int> result = countBits(n);
    for (int x : result) cout << x << " ";
    cout << endl;
    return 0;
}
```

OUTPUT:



```
#include <iostream>
#include <vector>
using namespace std;

vector<int> countBits(int n) {
    vector<int> dp(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        dp[i] = dp[i >> 1] + (i & 1);
    }
    return dp;
}

int main() {
    int n;
    cin >> n;
    vector<int> result = countBits(n);
    for (int x : result) cout << x << " ";
    cout << endl;
    return 0;
}
```

```
2
0 1 1


=== Code Execution Successful ===
```

Medium

1. Longest Palindromic Substring

Given a string s, return the longest palindromic substring in s.
Example 1: Input: s = "babad"
Output: "bab"

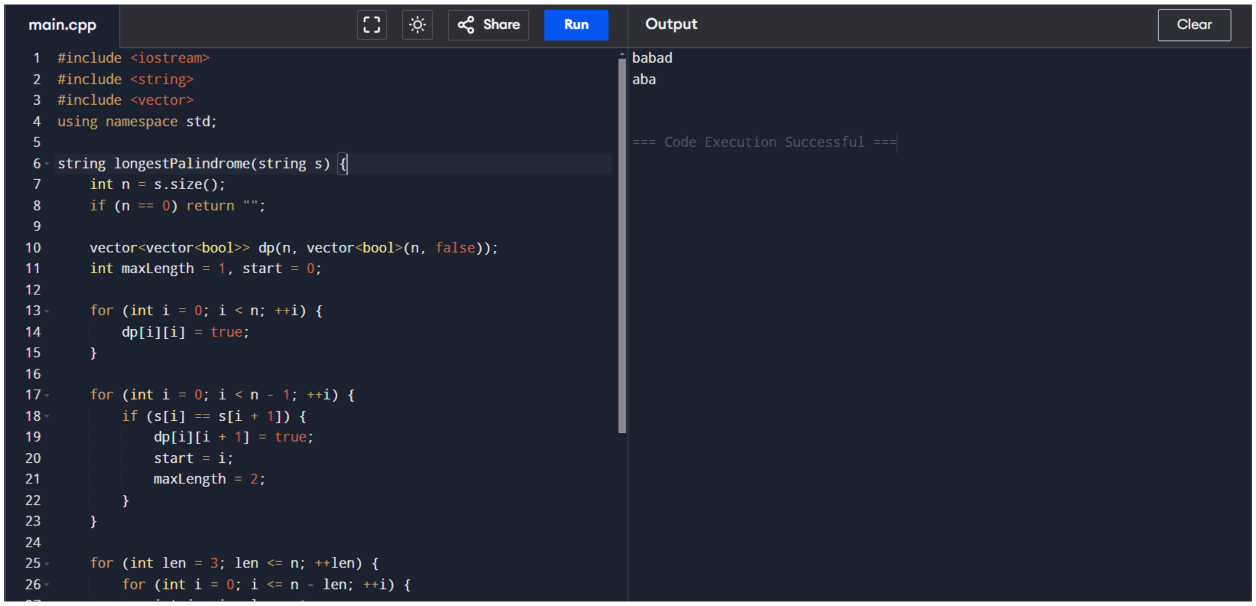
CODE:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;


string longestPalindrome(string s) {
    int n = s.size();
```

```
if (n == 0) return "";

vector<vector<bool>> dp(n, vector<bool>(n, false));
int maxLength = 1, start = 0;

for (int i = 0; i < n; ++i) {
    dp[i][i] = true;
}

for (int i = 0; i < n - 1; ++i) {
    if (s[i] == s[i + 1]) {
        dp[i][i + 1] = true;
        start = i;
        maxLength = 2;
    }
}

for (int len = 3; len <= n; ++len) {
    for (int i = 0; i <= n - len; ++i) {
        int j = i + len - 1;
        if (s[i] == s[j] && dp[i + 1][j - 1]) {
            dp[i][j] = true;
            start = i;
            maxLength = len;
        }
    }
}
```

```
return s.substr(start, maxLength);
}

int main() {
    string s;
    cin >> s;
    cout << longestPalindrome(s) << endl;
    return 0;
}
```

OUTPUT:



```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

string longestPalindrome(string s) {
    int n = s.size();
    if (n == 0) return "";

    vector<vector<bool>> dp(n, vector<bool>(n, false));
    int maxLength = 1, start = 0;

    for (int i = 0; i < n; ++i) {
        dp[i][i] = true;
    }

    for (int i = 0; i < n - 1; ++i) {
        if (s[i] == s[i + 1]) {
            dp[i][i + 1] = true;
            start = i;
            maxLength = 2;
        }
    }

    for (int len = 3; len <= n; ++len) {
        for (int i = 0; i <= n - len; ++i) {
```

```
babad
aba

=== Code Execution Successful ===
```

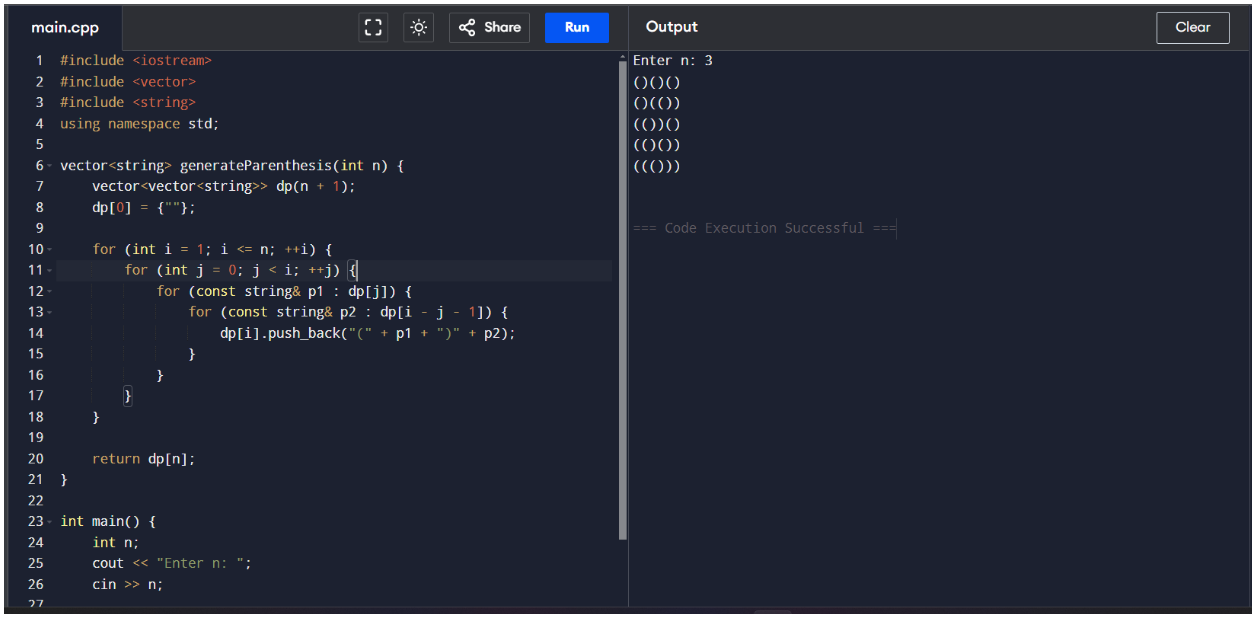
2. Generate Parentheses

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.
Example 1: Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]

CODE:

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

vector<string> generateParenthesis(int n) {
    vector<vector<string>> dp(n + 1);
    dp[0] = {""};

    for (int i = 1; i <= n; ++i) {
        for (int j = 0; j < i; ++j) {
            for (const string& p1 : dp[j]) {
                for (const string& p2 : dp[i - j - 1]) {
                    dp[i].push_back("(" + p1 + ")" + p2);
                }
            }
        }
    }

    return dp[n];
}

int main() {
    int n;
    cout << "Enter n: ";
    cin >> n;
    vector<string> result = generateParenthesis(n);
    for (const string& s : result) {
        cout << s << endl;
    }

    return 0;
}
```

OUTPUT:



main.cpp

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

vector<string> generateParenthesis(int n) {
    vector<vector<string>> dp(n + 1);
    dp[0] = {""};

    for (int i = 1; i <= n; ++i) {
        for (int j = 0; j < i; ++j) {
            for (const string& p1 : dp[j]) {
                for (const string& p2 : dp[i - j - 1]) {
                    dp[i].push_back("(" + p1 + ")" + p2);
                }
            }
        }
    }

    return dp[n];
}

int main() {
    int n;
    cout << "Enter n: ";
    cin >> n;
```

Output

```
Enter n: 3
()()()
()(())
(())()
(()())
((()))

=== Code Execution Successful ===
```

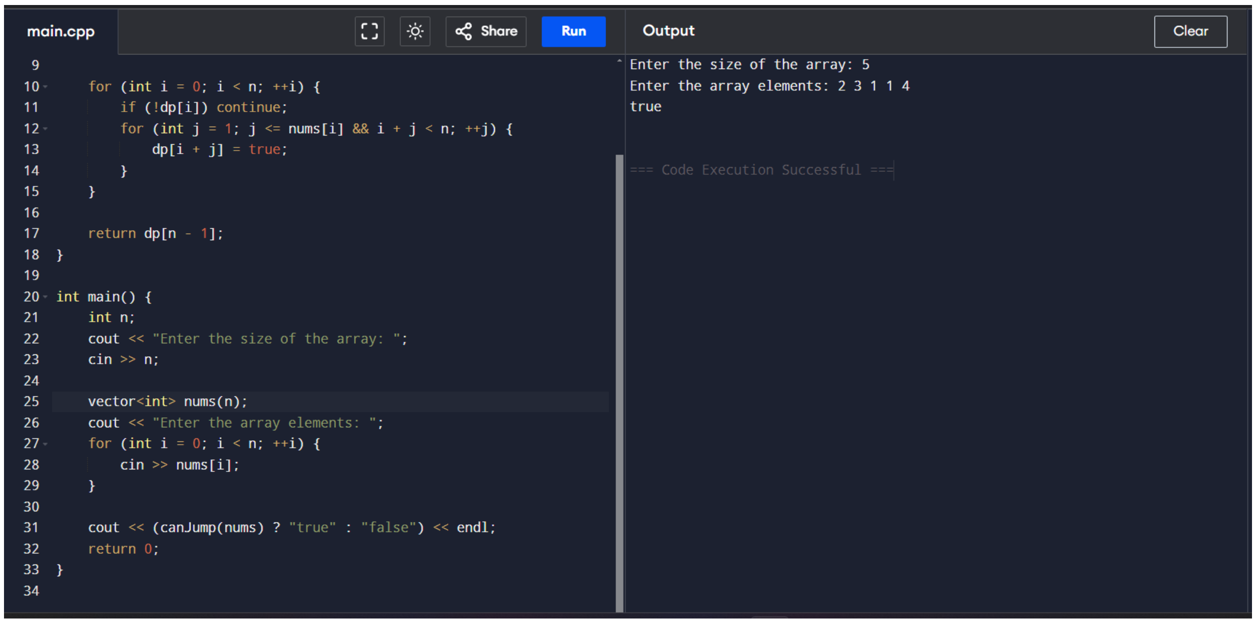
3. Jump Game

You are given an integer array nums. You are initially positioned at the array's first index, and each element in the array represents your maximum jump length at that position.
Return true if you can reach the last index, or false otherwise.
Example 1: Input: nums = [2,3,1,1,4]
Output: true

CODE:

```
#include <iostream>
#include <vector>
using namespace std;

bool canJump(vector<int>& nums) {
    int n = nums.size();
    vector<bool> dp(n, false);
    dp[0] = true;

    for (int i = 0; i < n; ++i) {
        if (!dp[i]) continue;
        for (int j = 1; j <= nums[i] && i + j < n; ++j) {
            dp[i + j] = true;
        }
    }

    return dp[n - 1];
}

int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;

    vector<int> nums(n);
    cout << "Enter the array elements: ";
    for (int i = 0; i < n; ++i) {
        cin >> nums[i];
    }

    cout << (canJump(nums) ? "true" : "false") << endl;
    return 0;
}
```

OUTPUT:



```
    for (int i = 0; i < n; ++i) {
        if (!dp[i]) continue;
        for (int j = 1; j <= nums[i] && i + j < n; ++j) {
            dp[i + j] = true;
        }
    }

    return dp[n - 1];
}

int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;

    vector<int> nums(n);
    cout << "Enter the array elements: ";
    for (int i = 0; i < n; ++i) {
        cin >> nums[i];
    }

    cout << (canJump(nums) ? "true" : "false") << endl;
    return 0;
}
```

```
Enter the size of the array: 5
Enter the array elements: 2 3 1 1 4
true

=== Code Execution Successful ===
```

HARD

Longest Increasing Path in a Matrix

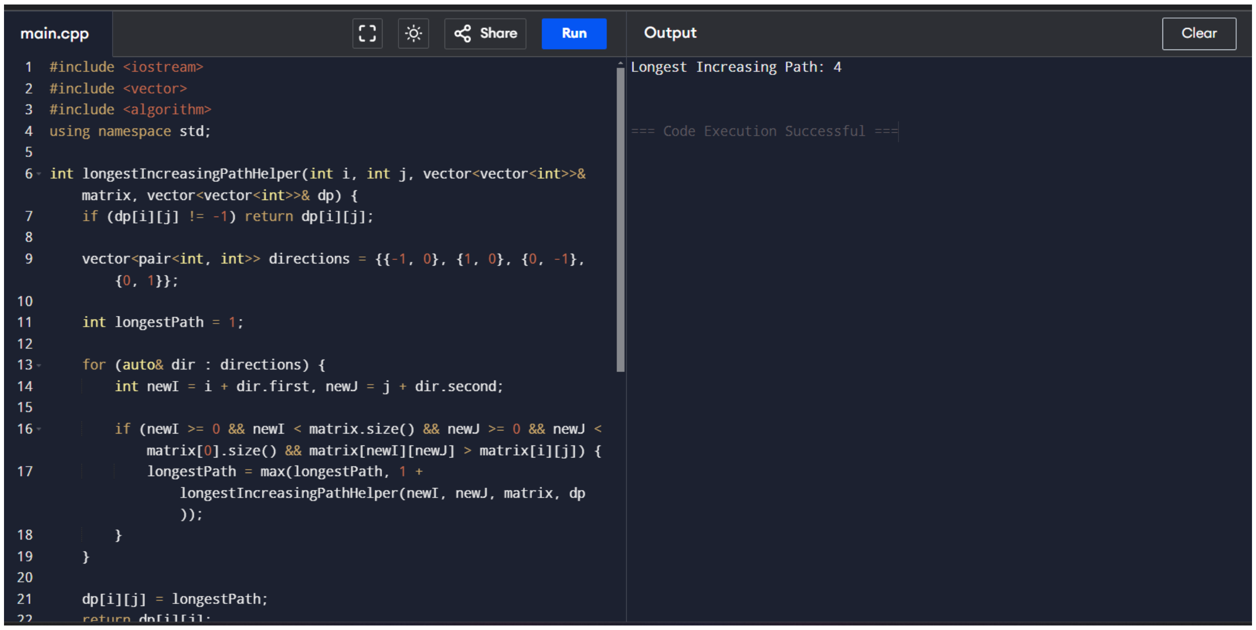
Given an m x n integers matrix, return the length of the longest increasing path in matrix. From each cell, you can either move in four directions: left, right, up, or down. You may not move diagonally or move outside the boundary (i.e., wrap-around is not allowed).
Example 1:
Input: matrix = [[9,9,4],[6,6,8],[2,1,1]]
Output: 4

CODE:

```
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;


int longestIncreasingPathHelper(int i, int j, vector<vector<int>>& matrix, vector<vector<int>>& dp) {

    if (dp[i][j] != -1) return dp[i][j];


    vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};


    int longestPath = 1;


    for (auto& dir : directions) {

        int newI = i + dir.first, newJ = j + dir.second;


        if (newI >= 0 && newI < matrix.size() && newJ >= 0 && newJ < matrix[0].size() && matrix[newI][newJ] > matrix[i][j]) {

            longestPath = max(longestPath, 1 + longestIncreasingPathHelper(newI, newJ, matrix, dp));

        }

    }
    dp[i][j] = longestPath;
    return dp[i][j];
}

int longestIncreasingPath(vector<vector<int>>& matrix) {
    if (matrix.empty() || matrix[0].empty()) return 0;

    int m = matrix.size(), n = matrix[0].size();
    vector<vector<int>> dp(m, vector<int>(n, -1));

    int maxPath = 0;

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            maxPath = max(maxPath, longestIncreasingPathHelper(i, j, matrix, dp));
        }
    }

    return maxPath;
}

int main() {
    vector<vector<int>> matrix = {{9,9,4},{6,6,8},{2,1,1}};

    cout << "Longest Increasing Path: " << longestIncreasingPath(matrix) << endl;

    return 0;
```

}

OUTPUT:



```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int longestIncreasingPathHelper(int i, int j, vector<vector<int>>& matrix, vector<vector<int>>& dp) {
    if (dp[i][j] != -1) return dp[i][j];

    vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    int longestPath = 1;

    for (auto& dir : directions) {
        int newI = i + dir.first, newJ = j + dir.second;

        if (newI >= 0 && newI < matrix.size() && newJ >= 0 && newJ < matrix[0].size() && matrix[newI][newJ] > matrix[i][j]) {
            longestPath = max(longestPath, 1 + longestIncreasingPathHelper(newI, newJ, matrix, dp));
        }
    }

    dp[i][j] = longestPath;
    return dp[i][j];
```

```
Longest Increasing Path: 4


=== Code Execution Successful ===
```

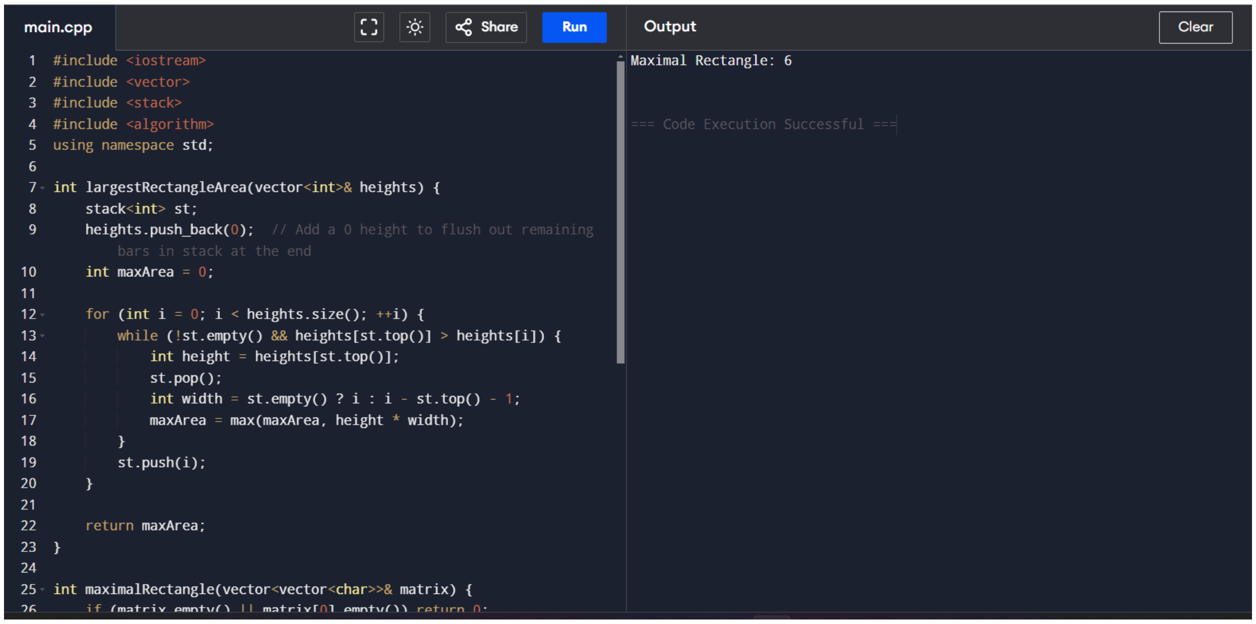
Maximal Rectangle

Given a rows x cols binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.
Example-
Input: matrix =
[["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
Output: 6

CODE:

```cpp
#include <iostream>

#include <vector>

#include <stack>

#include <algorithm>

using namespace std;


int largestRectangleArea(vector<int>& heights) {
    stack<int> st;
    heights.push_back(0);  // Add a 0 height to flush out remaining bars in stack at the end
    int maxArea = 0;

    for (int i = 0; i < heights.size(); ++i) {
        while (!st.empty() && heights[st.top()] > heights[i]) {
            int height = heights[st.top()];
            st.pop();
            int width = st.empty() ? i : i - st.top() - 1;
            maxArea = max(maxArea, height * width);
        }
        st.push(i);
    }

    return maxArea;
}

int maximalRectangle(vector<vector<char>>& matrix) {
    if (matrix.empty() || matrix[0].empty()) return 0;

    int rows = matrix.size(), cols = matrix[0].size();
    vector<int> heights(cols, 0);
    int maxArea = 0;

    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            heights[j] = (matrix[i][j] == '1') ? heights[j] + 1 : 0;
        }
        maxArea = max(maxArea, largestRectangleArea(heights));
    }

    return maxArea;
}
int main() {
    vector<vector<char>> matrix = {
        {'1', '0', '1', '0', '0'},
        {'1', '0', '1', '1', '1'},
        {'1', '1', '1', '1', '1'},
        {'1', '0', '0', '1', '0'}
    };
    cout << "Maximal Rectangle: " << maximalRectangle(matrix) << endl;
    return 0;
}
```

OUTPUT:



```
#include <iostream>
#include <vector>
#include <stack>
#include <algorithm>
using namespace std;

int largestRectangleArea(vector<int>& heights) {
    stack<int> st;
    heights.push_back(0);  // Add a 0 height to flush out remaining
        bars in stack at the end
    int maxArea = 0;

    for (int i = 0; i < heights.size(); ++i) {
        while (!st.empty() && heights[st.top()] > heights[i]) {
            int height = heights[st.top()];
            st.pop();
            int width = st.empty() ? i : i - st.top() - 1;
            maxArea = max(maxArea, height * width);
        }
        st.push(i);
    }

    return maxArea;
}

int maximalRectangle(vector<vector<char>>& matrix) {
    if (matrix.empty() || matrix[0].empty()) return 0;
```

```
Maximal Rectangle: 6


=== Code Execution Successful ===
```

Dungeon Game

The demons had captured the princess and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of m x n rooms laid out in a 2D grid. Our valiant knight was initially positioned in the top-left room and must fight his way through dungeon to rescue the princess. The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately Some of the rooms are guarded by demons (represented by negative integers), so the knight loses health upon entering these rooms; other rooms are either empty (represented as 0) or contain magic orbs that increase the knight's health (represented by positive integers). To reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step. Return the knight's minimum initial health so that he can rescue the princess.
Note that any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

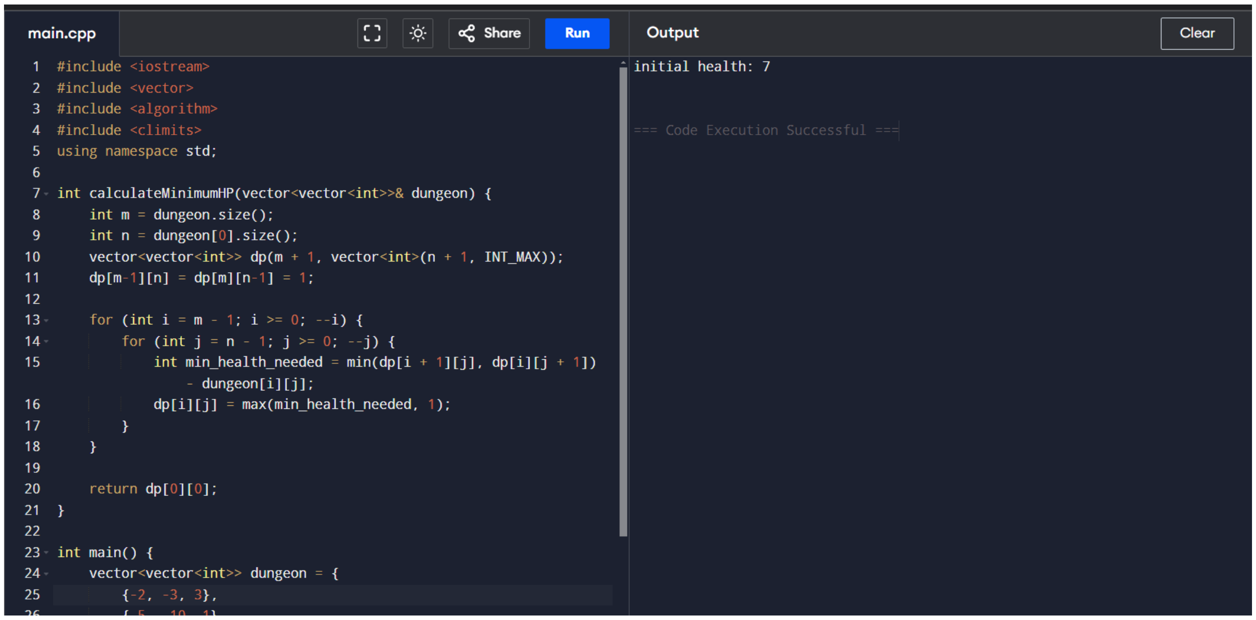
Example-
Input: dungeon = [[-2,-3,3],[-5,-10,1],[10,30,-5]]
Output: 7

CODE:

```
#include <iostream>

#include <vector>

#include <algorithm>

#include <climits>

using namespace std;


int calculateMinimumHP(vector<vector<int>>& dungeon) {

  int m = dungeon.size();

  int n = dungeon[0].size();

  vector<vector<int>> dp(m + 1, vector<int>(n + 1, INT_MAX));

  dp[m-1][n] = dp[m][n-1] = 1;


  for (int i = m - 1; i >= 0; --i) {

    for (int j = n - 1; j >= 0; --j) {

      int min_health_needed = min(dp[i + 1][j], dp[i][j + 1]) - dungeon[i][j];
```

```
            dp[i][j] = max(min_health_needed, 1);
        }
    }
    return dp[0][0];
}
int main() {
    vector<vector<int>> dungeon = {
        {-2, -3, 3},
        {-5, -10, 1},
        {10, 30, -5}
    };
    cout << "initial health: " << calculateMinimumHP(dungeon) << endl;

    return 0;
}
```

OUTPUT:



```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

int calculateMinimumHP(vector<vector<int>>& dungeon) {
    int m = dungeon.size();
    int n = dungeon[0].size();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, INT_MAX));
    dp[m-1][n] = dp[m][n-1] = 1;

    for (int i = m - 1; i >= 0; --i) {
        for (int j = n - 1; j >= 0; --j) {
            int min_health_needed = min(dp[i + 1][j], dp[i][j + 1])
                - dungeon[i][j];
            dp[i][j] = max(min_health_needed, 1);
        }
    }

    return dp[0][0];
}

int main() {
    vector<vector<int>> dungeon = {
        {-2, -3, 3},
```

```
initial health: 7


=== Code Execution Successful ===
```