NAME: MIHIR KUMAR	**UID: 22BCS13396**
BRANCH : BE_CSE	**SECTION: IOT_615_A**
SEMESTER: 05	**DATE: 27/12/2024**

Very Easy

1. 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

Example 2: Input: n = 25

Output: 1389537

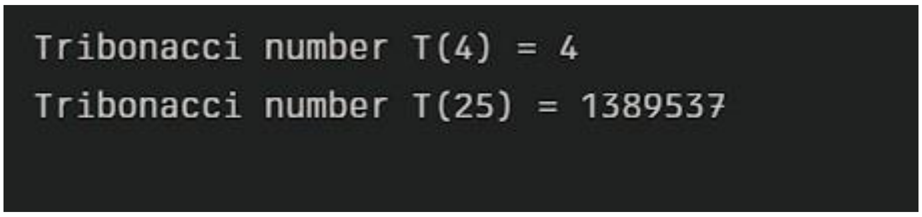
Constraints: 0 <= n <= 37

The answer is guaranteed to fit within a 32-bit integer, ie. answer <= 2^31 - 1.

Code:

```
#include <iostream> #include

<vector> int tribonacci(int n)

{     if (n == 0) return 0;     if
(n == 1 || n == 2) return 1;
std::vector<int> T(n + 1);
   T[0] = 0;
   T[1] = 1;     T[2] = 1;     for
(int i = 3; i <= n; ++i) {
      T[i] = T[i - 1] + T[i - 2] + T[i - 3];
   }    return
T[n];
} int main() {    //
Example inputs
int n1 = 4;     int n2
= 25;


   std::cout << "Tribonacci number T(" << n1 << ") = " << tribonacci(n1) <<
std::endl;
   std::cout << "Tribonacci number T(" << n2 << ") = " << tribonacci(n2) <<
std::endl;


   return 0;
}
```

Output:



```
Tribonacci number T(4) = 4
Tribonacci number T(25) = 1389537
```

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.

Example 2:

Input: n = 3

Output: false

Explanation: Alice chooses 1, Bob chooses 1, and Alice has no more moves.

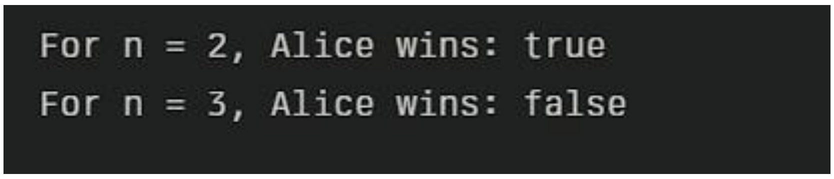
Constraints: 1 <= n <= 1000

Code:

```
#include <iostream>
```

```
bool divisorGame(int n) {

// Alice wins if n is even

return n % 2 == 0;

}


int main() {     //

Example inputs

int n1 = 2;     int n2

= 3;     std::cout <<

"For n = " << n1 <<

", Alice wins: " <<

(divisorGame(n1) ?

"true" : "false") << std::endl;

   std::cout << "For n = " << n2 << ", Alice wins: " << (divisorGame(n2) ?
"true" : "false") << std::endl;


   return 0; }
```

OUTPUT:



```
For n = 2, Alice wins: true
For n = 3, Alice wins: false
```

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

Explanation: "abab" is a substring in "ababc".

Example 2:

Input: sequence = "ababc", word = "ba"

Output: 1

Explanation: "ba" is a substring in "ababc". "baba" is not a substring in "ababc".

Example 3:

Input: sequence = "ababc", word = "ac"

Output: 0

Explanation: "ac" is not a substring in "ababc".

Constraints:

1 <= sequence.length <= 100 1 <= word.length <= 100

sequence and word contains only lowercase English letters.

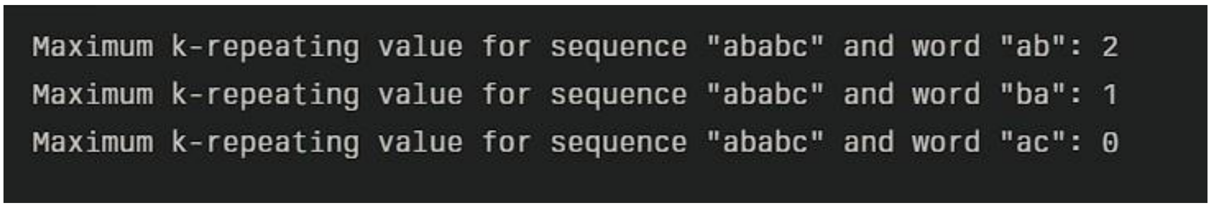
CODE:

```
#include <iostream>
```

```
#include <string>

int maxRepeating(const std::string& sequence, const std::string& word) {
   int k = 0;     std::string
repeatedWord = word;

   // Keep concatenating the word until it is no longer a substring of the
sequence     while (sequence.find(repeatedWord) != std::string::npos) {
      k++;         repeatedWord += word; // Concatenate
word to itself
   }

   return k;
}

int main() {     // Example inputs
std::string sequence1 = "ababc";
std::string word1 = "ab";

   std::string sequence2 = "ababc";
std::string word2 = "ba";

   std::string sequence3 = "ababc";
std::string word3 = "ac";
```

```
    std::cout << "Maximum k-repeating value for sequence \"" << sequence1 <<
"\" and word \"" << word1 << "\": " << maxRepeating(sequence1, word1) <<
std::endl;
    std::cout << "Maximum k-repeating value for sequence \"" << sequence2 <<
"\" and word \"" << word2 << "\": " << maxRepeating(sequence2, word2) <<
std::endl;
    std::cout << "Maximum k-repeating value for sequence \"" << sequence3 <<
"\" and word \"" << word3 << "\": " << maxRepeating(sequence3, word3) <<
std::endl;


    return 0; }
```

OUTPUT:



```
Maximum k-repeating value for sequence "ababc" and word "ab": 2
Maximum k-repeating value for sequence "ababc" and word "ba": 1
Maximum k-repeating value for sequence "ababc" and word "ac": 0
```

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

Example 2:Input: n = 3

Output: 3

Explanation: There are three ways to climb to the top.

1. 1 step + 1 step + 1 step

2. 1 step + 2 steps

3. 2 steps + 1 step

Constraints:1 <= n <= 45

CODE:

```
#include <iostream>

#include <vector>


int climbStairs(int n) {

if (n == 1) return 1;


   std::vector<int> dp(n + 1);     dp[1] = 1;

// One way to climb one step     dp[2] = 2; //

Two ways to climb two steps     for (int i =

3; i <= n; ++i) {

      dp[i] = dp[i - 1] + dp[i - 2]; // Sum of the ways to reach the previous two
steps

   }


   return dp[n];

}


int main() {     //

Example inputs
```

int n1 = 2;   int n2

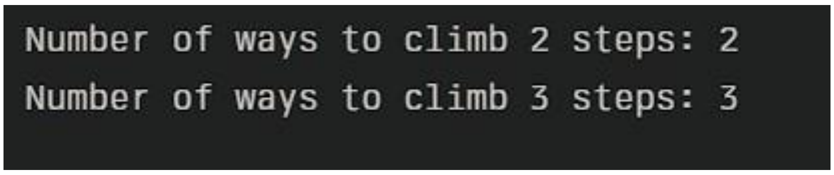
= 3;

    std::cout << "Number of ways to climb " << n1 << " steps: " << climbStairs(n1) << std::endl;

    std::cout << "Number of ways to climb " << n2 << " steps: " << climbStairs(n2) << std::endl;

    return 0; }

OUTPUT:



```
Number of ways to climb 2 steps: 2
Number of ways to climb 3 steps: 3
```

3. 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

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:Input: prices = [7,6,4,3,1]

Output: 0

Explanation: In this case, no transactions are done and the max profit = 0.

Constraints:1 <= prices.length <= 105

0 <= prices[i] <= 104

CODE:

```
#include <iostream>

#include <vector>


int maxProfit(const std::vector<int>& prices) {     int maxProfit = 0; //
Initialize maximum profit     int minPrice = INT_MAX; // Initialize
minimum price to a large value


   for (int price : prices) {
       // Update minimum price if the current price is lower
if (price < minPrice) {            minPrice = price;
       }
       // Calculate profit if selling at current price
int profit = price - minPrice;

       // Update maximum profit if the calculated profit is higher
if (profit > maxProfit) {            maxProfit = profit;
       }
   }


   return maxProfit;
}
```

int main() {   // Example inputs

std::vector<int> prices1 = {7, 1, 5, 3, 6, 4};

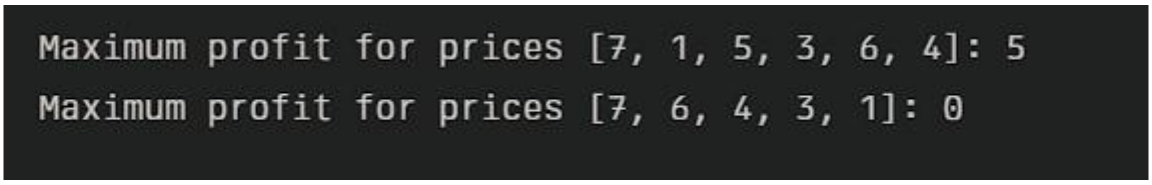
std::vector<int> prices2 = {7, 6, 4, 3, 1};

   std::cout << "Maximum profit for prices [7, 1, 5, 3, 6, 4]: " << maxProfit(prices1) << std::endl;

   std::cout << "Maximum profit for prices [7, 6, 4, 3, 1]: " << maxProfit(prices2) << std::endl;

   return 0; }

OUTPUT:



```
Maximum profit for prices [7, 1, 5, 3, 6, 4]: 5
Maximum profit for prices [7, 6, 4, 3, 1]: 0
```

4. Counting Bits

3. 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

Example 2:Input: n = 5

Output: [0,1,1,2,1,2]

Explanation:0 --> 0   1 --> 1   2 --> 10   3 --> 11   4 --> 100   5 --> 101

Constraints:0 <= n <= 105

Follow up:It is very easy to come up with a solution with a runtime of O(n log n). Can you do it in linear time O(n) and possibly in a single pass?

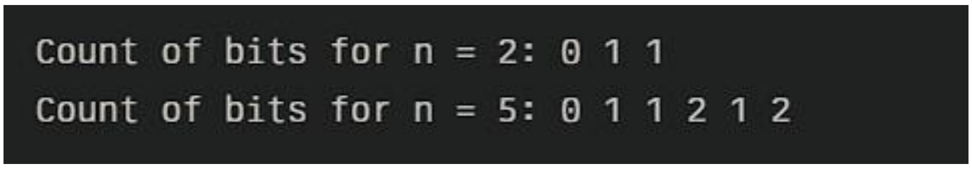
Can you do it without using any built-in function (i.e., like __builtin_popcount in C++)?

CODE:

```
#include <iostream>

#include <vector>


std::vector<int> countBits(int n) {

std::vector<int> ans(n + 1);


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

if (i % 2 == 0) {            ans[i] =

ans[i / 2]; // Even

      } else {            ans[i] = ans[i /

2] + 1; // Odd

      }

   }

   return ans;

}


int main() {    //

Example inputs

int n1 = 2;    int n2

= 5;
```

```
    std::vector<int> result1 = countBits(n1);
std::vector<int> result2 = countBits(n2);


    std::cout << "Count of bits for n = " << n1 << ": ";
    for (int bit : result1) {
std::cout << bit << " ";
    }     std::cout <<
std::endl;


    std::cout << "Count of bits for n = " << n2 << ": ";
    for (int bit : result2) {
std::cout << bit << " ";
    }     std::cout <<
std::endl;


    return 0;
}
```

OUTPUT:



```
Count of bits for n = 2: 0 1 1
Count of bits for n = 5: 0 1 1 2 1 2
```

Medium:

1. Longest Palindromic Substring

Given a string s, return the longest palindromic substring in s.

Example 1: Input: s = "babad"

Output: "bab"

Explanation: "aba" is also a valid answer.

Example 2: Input: s = "cbbd"

Output: "bb"

Constraints: 1 <= s.length <= 1000 s

consist of only digits and English letters.

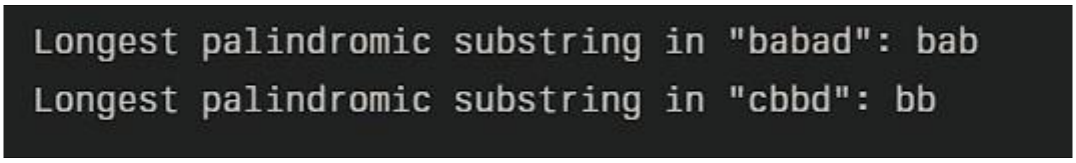
CODE:

```
#include <iostream>
#include <string>

std::string longestPalindrome(const std::string& s) {
if (s.empty()) return "";

    int start = 0, end = 0;

    for (int i = 0; i < s.length(); ++i) {          //
Check for odd-length palindromes          int
len1 = expandAroundCenter(s, i, i);          //
Check for even-length palindromes          int
len2 = expandAroundCenter(s, i, i + 1);
// Get the maximum length from both cases
int len = std::max(len1, len2);
```

```
        if (len > end - start) {
start = i - (len - 1) / 2;
end = i + len / 2;
        }
    }

    return s.substr(start, end - start + 1);
}

int expandAroundCenter(const std::string& s, int left, int right) {
while (left >= 0 && right < s.length() && s[left] == s[right]) {
        left--;
        right++;
    }
    return right - left - 1; // Length of the palindrome
}

int main() {     // Example
inputs     std::string s1 =
"babad";     std::string s2 =
"cbbd";

    std::cout << "Longest palindromic substring in \"" << s1 << "\": " <<
longestPalindrome(s1) << std::endl;
    std::cout << "Longest palindromic substring in \"" << s2 << "\": " <<
longestPalindrome(s2) << std::endl;
```

```
    return 0; }
```

OUTPUT:



```
Longest palindromic substring in "babad": bab
Longest palindromic substring in "cbbd": bb
```

2. Generate Parentheses

1. Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

Example 1: Input: n = 3

Output: ["((()))","(()())","(())()","()(())","()()()"]

Example 2: Input: n = 1

Output: ["()"]

Constraints: 1 <= n <= 8

CODE:

```
#include <iostream>

#include <vector>

#include <string>


void backtrack(std::vector<std::string>& result, std::string current, int open, int
close, int n) {

    // If the current string is of the maximum length, add it to the result

if (current.length() == 2 * n) {        result.push_back(current);

        return;

    }
    // If we can add an open parenthesis, do it     if (open
< n) {        backtrack(result, current + "(", open + 1,
close, n);
    }
    // If we can add a close parenthesis, do it     if (close
< open) {        backtrack(result, current + ")", open,
close + 1, n);
    }
}

std::vector<std::string> generateParenthesis(int n) {
std::vector<std::string> result;     backtrack(result,
"", 0, 0, n);     return result;
}

int main() {     //
Example inputs
int n1 = 3;     int n2
= 1;

    std::vector<std::string>          output1          =          generateParenthesis(n1);
std::vector<std::string>   output2   =   generateParenthesis(n2);   std::cout   <<
"Combinations of well-formed parentheses for n = " << n1 << ":
```

";   for (const auto& str :

output1) {   std::cout << str

<< " ";

   }   std::cout <<

std::endl;

   std::cout << "Combinations of well-formed parentheses for n = " << n2 << ":
";   for (const auto& str :

output2) {   std::cout << str

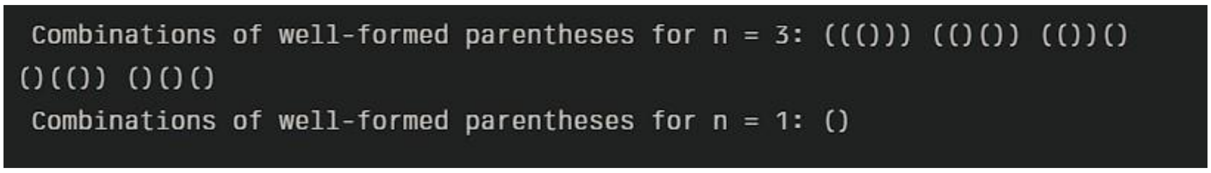
<< " ";

   }   std::cout <<

std::endl;

   return 0; }

OUTPUT:



```
 Combinations of well-formed parentheses for n = 3: ((())) (()()) (())()
()(()) ()()()
 Combinations of well-formed parentheses for n = 1: ()
```

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

Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.

Example 2: Input: nums = [3,2,1,0,4]

Output: false

Explanation: You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.

Constraints: 1 <= nums.length <= 104

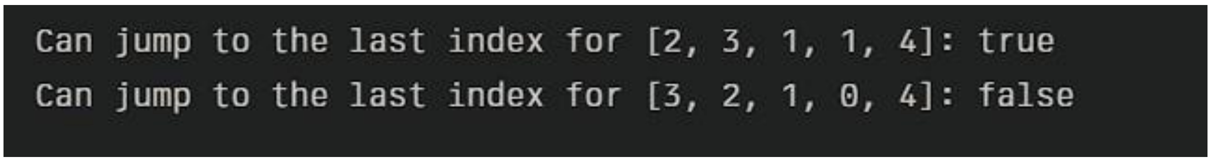
0 <= nums[i] <= 105

CODE:

```
#include <iostream>

#include <vector>


bool canJump(const std::vector<int>& nums) {     int

maxReach = 0; // The farthest index we can reach     int

n = nums.size();


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

       // If we are at an index that is beyond the maximum reachable index, return

false         if (i > maxReach) {

           return false;

       }

       // Update the maximum reachable index

maxReach = std::max(maxReach, i + nums[i]);         // If

we can reach or exceed the last index, return true         if

(maxReach >= n - 1) {

           return true;

       }
```

```
    }

    return false; // If we've gone through all indices and didn't reach the last one }


int main() {     // Example inputs
std::vector<int> nums1 = {2, 3, 1, 1, 4};
std::vector<int> nums2 = {3, 2, 1, 0, 4};


    std::cout << "Can jump to the last index for [2, 3, 1, 1, 4]: " <<
(canJump(nums1) ? "true" : "false") << std::endl;
    std::cout << "Can jump to the last index for [3, 2, 1, 0, 4]: " <<
(canJump(nums2) ? "true" : "false") << std::endl;


    return 0; }
```

OUTPUT:



```
Can jump to the last index for [2, 3, 1, 1, 4]: true
Can jump to the last index for [3, 2, 1, 0, 4]: false
```

5 Given an integer n, return the least number of perfect square numbers that sum to n.

A perfect square is an integer that is the square of an integer; in other words, it is the product of some integer with itself. For example, 1, 4, 9, and 16 are perfect squares while 3 and 11 are not.

Example 1:

Input: n = 12

Output: 3

Explanation: 12 = 4 + 4 + 4.

Example 2:

Input: n = 13

Output: 2

Explanation: 13 = 4 + 9.

Constraints: 1

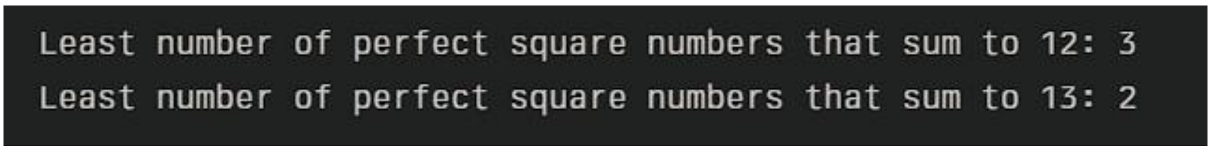
<= n <= 104

CODE:

```
#include <iostream>

#include <vector>

#include <cmath>


int numSquares(int n) {
   std::vector<int> dp(n + 1, INT_MAX); // Initialize dp array with maximum
values     dp[0] = 0; // Base case


   for (int i = 1; i <= n; ++i) {         for (int j =
1; j * j <= i; ++j) {            dp[i] =
std::min(dp[i], dp[i - j * j] + 1);
       }
   }


   return dp[n];
}
```

```
int main() {     //
Example inputs
int n1 = 12;     int
n2 = 13;     std::cout
<< "Least number
of perfect square
numbers that sum
to " << n1 <<
": " << numSquares(n1) << std::endl;
    std::cout << "Least number of perfect square numbers that sum to " << n2 <<
": " << numSquares(n2) << std::endl;


    return 0; }
```

OUTPUT:



```
Least number of perfect square numbers that sum to 12: 3
Least number of perfect square numbers that sum to 13: 2
```

Hard:

1. Maximal Rectangle

1. 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

Explanation: The maximal rectangle is shown in the above picture.

Example 2: Input: matrix = [["0"]]

Output: 0

Example 3: Input: matrix = [["1"]]

Output: 1

Constraints: rows ==

matrix.length cols ==

matrix[i].length 1 <=

row, cols <= 200

matrix[i][j] is '0' or

'1'.

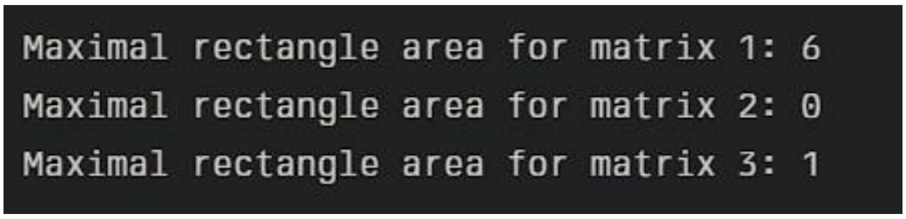
CODE:

```
#include <iostream>

#include <vector>

#include <stack>


using namespace std;


// Function to calculate the maximum area of rectangle in a histogram int

largestRectangleArea(const vector<int>& heights) {

   stack<int> st;     int

maxArea = 0;     int n =

heights.size();
```

```
   for (int i = 0; i <= n; i++) {         while (!st.empty() && (i == n ||
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;
}

// Function to find the maximal rectangle containing only 1's int
maximalRectangle(const vector<vector<char>>& matrix) {
   if (matrix.empty()) return 0;

   int rows = matrix.size();
int 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++) {
// Update heights             if
(matrix[i][j] == '1') {
```

```
heights[j]++;            } else {
heights[j] = 0;
            }
        }
        // Calculate max area for current row's histogram
maxArea = max(maxArea, largestRectangleArea(heights));
    }

    return maxArea;
}

int main() {
    // Example inputs
vector<vector<char>> matrix1 = {
        {'1', '0', '1', '0', '0'},
        {'1', '0', '1', '1', '1'},
        {'1', '1', '1', '1', '1'},
        {'1', '0', '0', '1', '0'}
    };

    vector<vector<char>> matrix2 = {
        {'0'}
    };

    vector<vector<char>> matrix3 = {
        {'1'}
    };


    cout << "Maximal rectangle area for matrix 1: " <<
maximalRectangle(matrix1) << endl;
    cout << "Maximal rectangle area for matrix 2: " <<
maximalRectangle(matrix2) << endl;
    cout << "Maximal rectangle area for matrix 3: " <<
maximalRectangle(matrix3) << endl;


    return 0; }
```

OUTPUT:



```
Maximal rectangle area for matrix 1: 6
Maximal rectangle area for matrix 2: 0
Maximal rectangle area for matrix 3: 1
```

2. Dungeon Game

2. 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 topleft 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

Explanation: The initial health of the knight must be at least 7 if he follows the optimal path: RIGHT-> RIGHT -> DOWN -> DOWN.

Example 2: Input: dungeon = [[0]]

Output: 1

Constraints:

m == dungeon.length n

== dungeon[i].length 1

<= m, n <= 200

-1000 <= dungeon[i][j] <= 1000

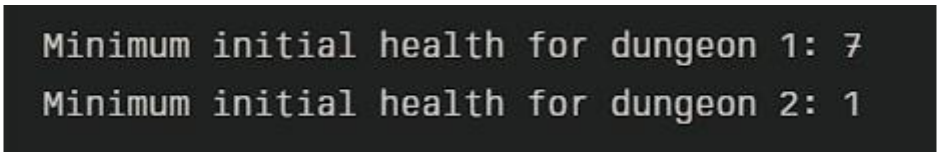
CODE:

```
#include <iostream>

#include <vector>

#include <algorithm>


using namespace std;


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

   int m = dungeon.size();

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

```
   // Create a DP table     vector<vector<int>>
dp(m, vector<int>(n, 0));

   // Start from the princess's room     dp[m - 1][n - 1]
= max(1, 1 - dungeon[m - 1][n - 1]);

   // Fill the last row     for (int j = n - 2; j >= 0; j--) {         dp[m
- 1][j] = max(1, dp[m - 1][j + 1] - dungeon[m - 1][j]);
   }
   // Fill the last column     for (int i = m - 2; i >= 0; i--) {
dp[i][n - 1] = max(1, dp[i + 1][n - 1] - dungeon[i][n - 1]);     }
   // Fill the rest of the DP table     for (int i = m - 2; i >= 0;
i--) {         for (int j = n - 2; j >= 0; j--) {             int
minHealthOnExit = min(dp[i + 1][j], dp[i][j + 1]);
dp[i][j] = max(1, minHealthOnExit - dungeon[i][j]);
       }
   }

   return dp[0][0]; // Minimum health needed at start
}

int main() {     // Example inputs     vector<vector<int>> dungeon1 = {{-2,
-3, 3}, {-5, -10, 1}, {10, 30, -5}};     vector<vector<int>> dungeon2 =
{{0}};
```

```
    cout << "Minimum initial health for dungeon 1: " <<
calculateMinimumHP(dungeon1) << endl;
    cout << "Minimum initial health for dungeon 2: " <<
calculateMinimumHP(dungeon2) << endl;


    return 0; }
```

OUTPUT:



```
Minimum initial health for dungeon 1: 7
Minimum initial health for dungeon 2: 1
```

3. Number of Digit One

Given an integer n, count the total number of digit 1 appearing in all nonnegative integers less than or equal to n.

Example 1: Input: n = 13

Output: 6

Example 2: Input: n = 0

Output: 0

Constraints: 0 <= n <= 109

CODE:

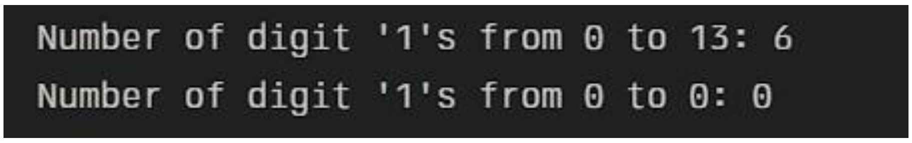
```
#include <iostream>


int countDigitOne(int n) {     long long count = 0; // To avoid overflow
long long factor = 1; // Represents the current digit position (1, 10, 100, ...)


    while (factor <= n) {
        long long lowerNumbers = n - (n / factor) * factor; // Numbers lower than
the current position         long long currentDigit = (n / factor) % 10; // Current
digit
        long long higherNumbers = n / (factor * 10); // Numbers higher than the
current position


        // Count the contribution of the current digit
if (currentDigit == 0) {              count +=
higherNumbers * factor;
        } else if (currentDigit == 1) {             count +=
higherNumbers * factor + lowerNumbers + 1;
        } else {
            count += (higherNumbers + 1) * factor;
        }


        factor *= 10; // Move to the next digit position
    }


    return count;
}


int main() {     //
Example inputs
int n1 = 13;     int
n2 = 0;
```

std::cout << "Number of digit '1's from 0 to " << n1 << ": " << countDigitOne(n1) << std::endl;

std::cout << "Number of digit '1's from 0 to " << n2 << ": " << countDigitOne(n2) << std::endl;

return 0; }

OUTPUT:



```
Number of digit '1's from 0 to 13: 6
Number of digit '1's from 0 to 0: 0
```

Very Hard:

1. Cherry Pickup

1. You are given an n x n grid representing a field of cherries, each cell is one of three possible integers.

0 means the cell is empty, so you can pass through,

1 means the cell contains a cherry that you can pick up and pass through, or

-1 means the cell contains a thorn that blocks your way.

Return the maximum number of cherries you can collect by following the rules below:

Starting at the position (0, 0) and reaching (n - 1, n - 1) by moving right or down through valid path cells (cells with value 0 or 1).

After reaching (n - 1, n - 1), returning to (0, 0) by moving left or up through valid path cells.

When passing through a path cell containing a cherry, you pick it up, and the cell becomes an empty cell 0.

If there is no valid path between (0, 0) and (n - 1, n - 1), then no cherries can be collected.

Input: grid = [[0,1,-1],[1,0,-1],[1,1,1]]

Output: 5

Explanation: The player started at (0, 0) and went down, down, right right to reach (2, 2).

4 cherries were picked up during this single trip, and the matrix becomes [[0,1,-1],[0,0,-1],[0,0,0]].

Then, the player went left, up, up, left to return home, picking up one more cherry.

The total number of cherries picked up is 5, and this is the maximum possible.

Example 2: Input: grid = [[1,1,-1],[1,-1,1],[-1,1,1]]

Output: 0

Constraints: n ==

grid.length n ==

grid[i].length 1 <=

n <= 50

grid[i][j] is -1, 0, or 1.

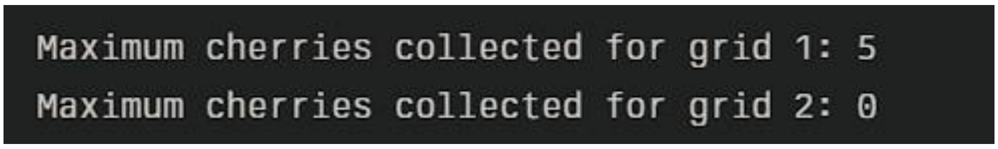
grid[0][0] != -1 grid[n

- 1][n - 1] != -1

CODE:

```
#include <iostream>

#include <vector>

#include <algorithm>


using namespace std;


int cherryPickup(vector<vector<int>>& grid) {
    int n = grid.size();
    // Create a DP table initialized to -1
    vector<vector<vector<int>>> dp(n, vector<vector<int>>(n, vector<int>(n,
1)));

    // Initialize the starting position     dp[0][0][0] = grid[0][0];
// Starting at (0, 0) for both players

    for (int step = 0; step < 2 * n - 1; ++step) {         for (int x1
= 0; x1 <= min(step, n - 1); ++x1) {             for (int x2 = 0; x2
<= min(step, n - 1); ++x2) {                 int y1 = step - x1; //
Calculate y position for player 1                 int y2 = step - x2; //
Calculate y position for player 2                 if (y1 >= n || y2 >=
n || grid[x1][y1] == -1 || grid[x2][y2] == -1) {
continue; // Invalid positions
                }

                // Collect cherries from both positions                 int cherries =
grid[x1][y1];                 if (x1 != x2) { // Avoid double counting if both are in
the same cell                     cherries += grid[x2][y2];
                }

                // Update DP table
                for (int newX1 = x1; newX1 <= x1 + 1 && newX1 < n; ++newX1) {
                    for (int newX2 = x2; newX2 <= x2 + 1 && newX2 < n; ++newX2)
{                       if (newX1 >= n || newX2 >= n)
continue;
                      dp[newX1][newX2][step + 1] = max(dp[newX1][newX2][step +
1],                                          dp[x1][x2][step] +
cherries);                   }
               }
           }
       }
   }

   return max(0, dp[n - 1][n - 1][2 * n - 2]); // Return maximum cherries
collected
}

int main() {     vector<vector<int>> grid1 = {{0, 1, -1}, {1, 0, -
1}, {1, 1, 1}};     vector<vector<int>> grid2 = {{1, 1, -1}, {1, -1,
1}, {-1, 1, 1}};

   cout << "Maximum cherries collected for grid 1: " << cherryPickup(grid1)
<< endl;
   cout << "Maximum cherries collected for grid 2: " << cherryPickup(grid2) <<
endl;

   return 0; }
```

OUTPUT:



```
Maximum cherries collected for grid 1: 5
Maximum cherries collected for grid 2: 0
```

2. Sliding Puzzle

On an 2 x 3 board, there are five tiles labeled from 1 to 5, and an empty square represented by 0. A move consists of choosing 0 and a 4-directionally adjacent number and swapping it. The state of the board is solved if and only if the board is [[1,2,3],[4,5,0]]. Given the puzzle board board, return the least number of moves required so that the state of the board is solved. If it is impossible for the state of the board to be solved, return 1.

Example 1:

Input: board = [[1,2,3],[4,0,5]]

Output: 1

Explanation: Swap the 0 and the 5 in one move.

Example 2:

Input: board = [[1,2,3],[5,4,0]]

Output: -1

Explanation: No number of moves will make the board solved.

Example 3:

Input: board = [[4,1,2],[5,0,3]]

Output: 5

Explanation: 5 is the smallest number of moves that solves the board.

An example path:

After move 0: [[4,1,2],[5,0,3]]

After move 1: [[4,1,2],[0,5,3]]

After move 2: [[0,1,2],[4,5,3]]

After move 3: [[1,0,2],[4,5,3]]

After move 4: [[1,2,0],[4,5,3]]

After move 5: [[1,2,3],[4,5,0]] Constraints:

board.length == 2 board[i].length

== 3

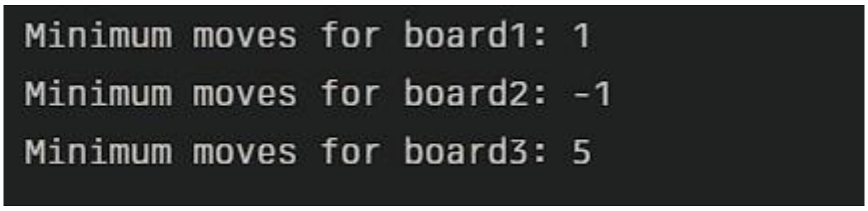
0 <= board[i][j] <= 5

Each value board[i][j] is unique.

CODE:

```
#include <iostream>

#include <vector>

#include <queue>

#include <string>

#include <unordered_set>


using namespace std;


string boardToString(const vector<vector<int>>& board) {
   string result;
   for (const auto& row : board) {
for (int num : row) {
result += to_string(num);
      }    }
return result;
}


vector<vector<int>> stringToBoard(const string& str) {
    return {{str[0] - '0', str[1] - '0', str[2] - '0'},
            {str[3] - '0', str[4] - '0', str[5] - '0'}};
}

int slidingPuzzle(vector<vector<int>>& board) {     string
target = "123450"; // Target configuration as a string     string
start = boardToString(board);

    if (start == target) return 0; // Already solved

    // Directions for moving the empty space     vector<pair<int, int>>
directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}}; queue<string> q;
unordered_set<string> visited;

    q.push(start);
visited.insert(start);

    int moves = 0;

    while (!q.empty()) {
        int size = q.size();         for
(int i = 0; i < size; ++i) {
string current = q.front();
            q.pop();
            // Find the position of the empty space (0)
int zeroPos = current.find('0');              int x =
zeroPos / 3;              int y = zeroPos % 3;

            // Try all possible directions
for (const auto& dir : directions) {
int newX = x + dir.first;                  int
newY = y + dir.second;

                // Check if new position is valid
                if (newX >= 0 && newX < 2 && newY >= 0 && newY < 3) {
                    string nextState = current;
                    swap(nextState[zeroPos], nextState[newX * 3 + newY]);

                    if (nextState == target) {
return moves + 1; // Found solution
                    }

                    if (visited.find(nextState) == visited.end()) {
visited.insert(nextState);
                        q.push(nextState);
                    }
                }
            }
        }
        moves++;
    }

    return -1; // No solution found
}

int main() {     vector<vector<int>> board1 = {{1, 2,
3}, {4, 0, 5}};     vector<vector<int>> board2 = {{1, 2,
3}, {5, 4, 0}};     vector<vector<int>> board3 = {{4, 1,
2}, {5, 0, 3}};

    cout << "Minimum moves for board1: " << slidingPuzzle(board1) << endl; //
Output: 1
    cout << "Minimum moves for board2: " << slidingPuzzle(board2) << endl; //
Output: -1
    cout << "Minimum moves for board3: " << slidingPuzzle(board3) << endl; //
Output: 5

    return 0; }
```

OUTPUT:



```
Minimum moves for board1: 1
Minimum moves for board2: -1
Minimum moves for board3: 5
```

3. Race Car

Your car starts at position 0 and speed +1 on an infinite number line. Your car can go into negative positions. Your car drives automatically according to a sequence of instructions 'A' (accelerate) and 'R' (reverse):

When you get an instruction 'A', your car does the following:

position += speed speed *= 2

When you get an instruction 'R', your car does the following:

If your speed is positive then speed = -1 otherwise speed = 1

Your position stays the same.

For example, after commands "AAR", your car goes to positions 0 --> 1 --> 3 --> 3, and your speed goes to 1 --> 2 --> 4 --> -1.

Given a target position target, return the length of the shortest sequence of instructions to get there.

Example 1:

Input: target = 3

Output: 2

Explanation:

The shortest instruction sequence is "AA".

Your position goes from 0 --> 1 --> 3.

Example 2:

Input: target = 6

Output: 5

Explanation:

The shortest instruction sequence is "AAARA".

Your position goes from 0 --> 1 --> 3 --> 7 --> 7 --> 6.

Constraints: 1 <= target <= 104

CODE:

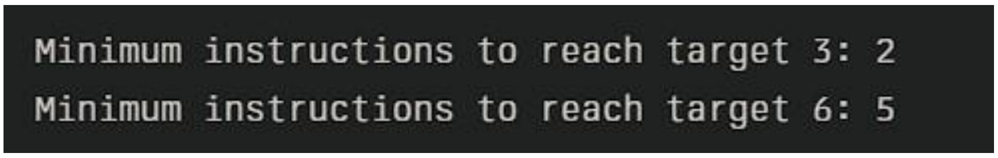
```
#include <iostream>

#include <queue>

#include <set>

#include <tuple>


using namespace std;


int raceCar(int target) {
   // Queue for BFS: (position, speed)
queue<tuple<int, int, int>> q; // (position, speed, steps)
   q.push(make_tuple(0, 1, 0)); // Start at position 0 with speed 1 and 0 steps

   // Set to track visited states
   set<pair<int, int>> visited;
visited.insert({0, 1});

   while (!q.empty()) {        auto [position,
speed, steps] = q.front();
      q.pop();

      // Check if we've reached the target
if (position == target) {           return
steps;
      }
```

```
        // Option 1: Accelerate          int
newPosition = position + speed;          int
newSpeed = speed * 2;

        if (abs(newPosition) <= 2 * target && visited.find({newPosition,
newSpeed}) == visited.end()) {             visited.insert({newPosition,
newSpeed});
            q.push(make_tuple(newPosition, newSpeed, steps + 1));
        }
        // Option 2: Reverse          newSpeed = (speed > 0) ? -1 :
1; // Reverse speed          if (visited.find({position,
newSpeed}) == visited.end()) {
visited.insert({position, newSpeed});
            q.push(make_tuple(position, newSpeed, steps + 1));
        }
    }

    return -1; // Should not reach here
}

int main() {     int
target1 = 3;     int
target2 = 6;

    cout << "Minimum instructions to reach target " << target1 << ": " <<
raceCar(target1) << endl; // Output: 2
```

cout << "Minimum instructions to reach target " << target2 << ": " << raceCar(target2) << endl; // Output: 5

return 0; }

OUTPUT:



```
Minimum instructions to reach target 3: 2
Minimum instructions to reach target 6: 5
```