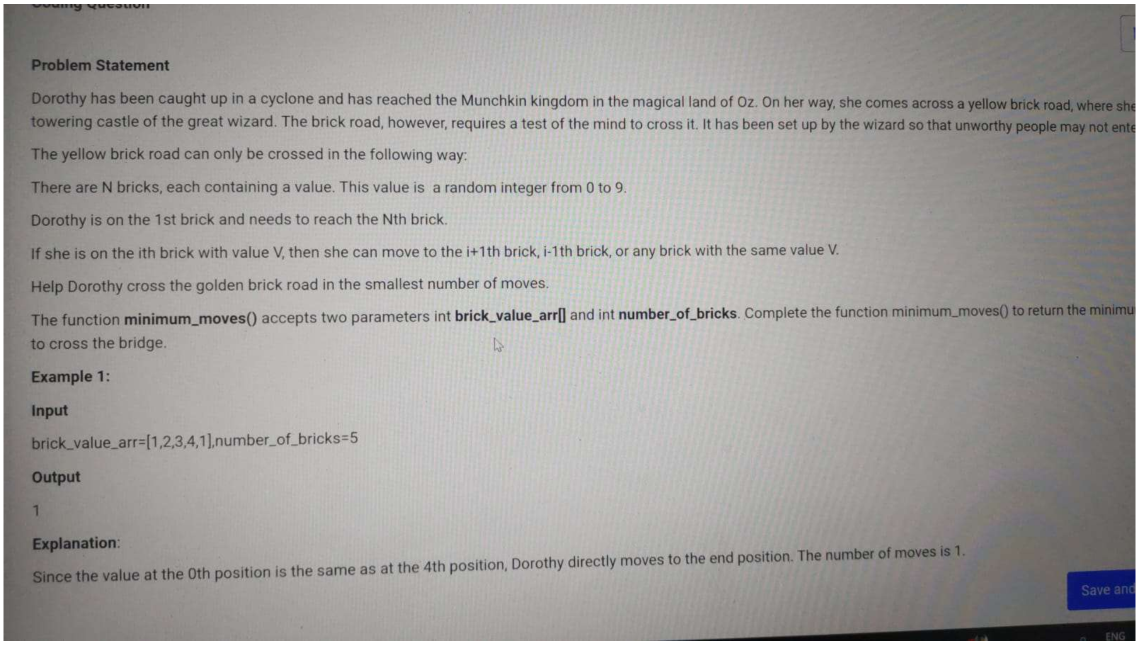
PET Coding Questions

Doorthy Question:


Coding Question

Problem Statement

Dorothy has been caught up in a cyclone and has reached the Munchkin kingdom in the magical land of Oz. On her way, she comes across a yellow brick road, where she
towering castle of the great wizard. The brick road, however, requires a test of the mind to cross it. It has been set up by the wizard so that unworthy people may not ente

The yellow brick road can only be crossed in the following way:

There are N bricks, each containing a value. This value is a random integer from 0 to 9.

Dorothy is on the 1st brick and needs to reach the Nth brick.

If she is on the ith brick with value V, then she can move to the i+1th brick, i-1th brick, or any brick with the same value V.

Help Dorothy cross the golden brick road in the smallest number of moves.

The function **minimum_moves()** accepts two parameters int **brick_value_arr[]** and int **number_of_bricks**. Complete the function minimum_moves() to return the minimu
to cross the bridge.

Example 1:

Input

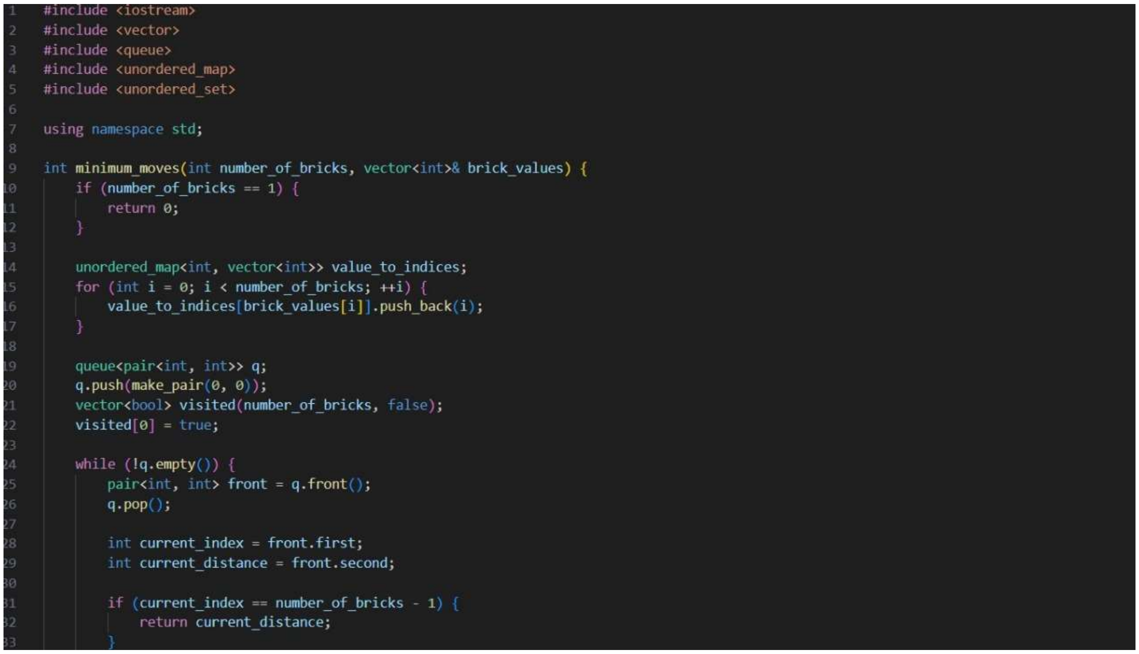
brick_value_arr=[1,2,3,4,1],number_of_bricks=5

Output

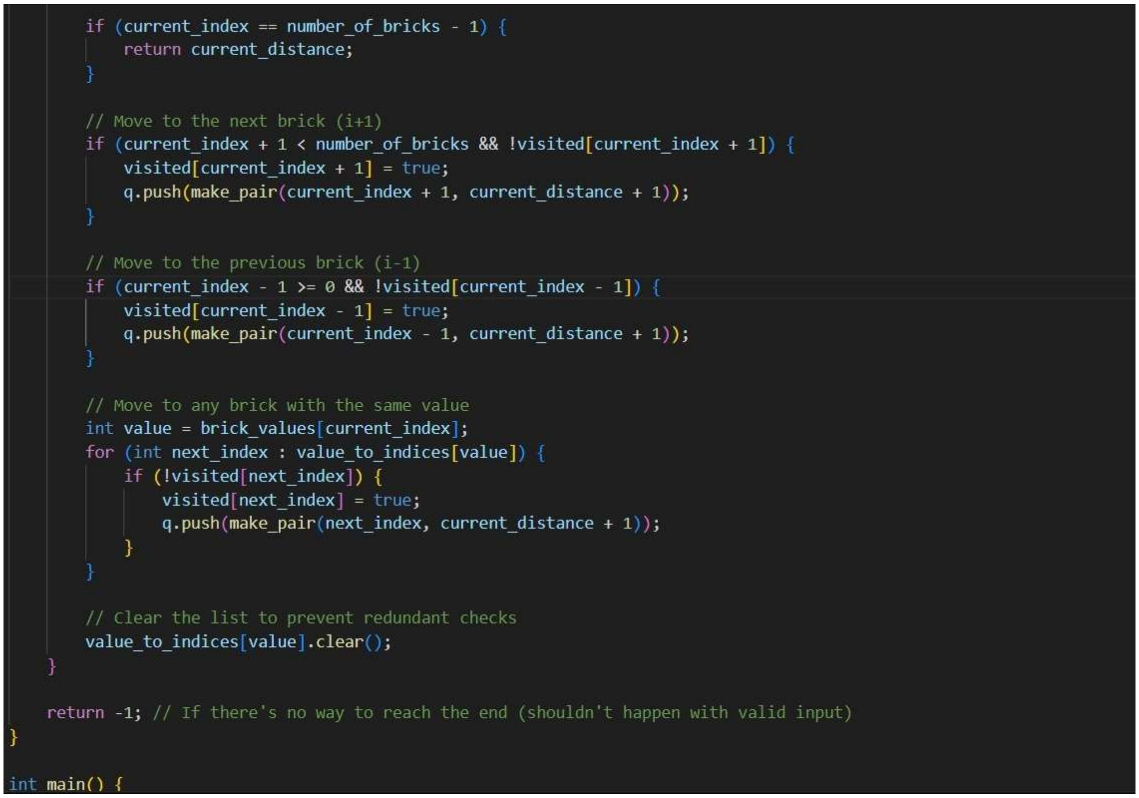
1

Explanation:

Since the value at the 0th position is the same as at the 4th position, Dorothy directly moves to the end position. The number of moves is 1.

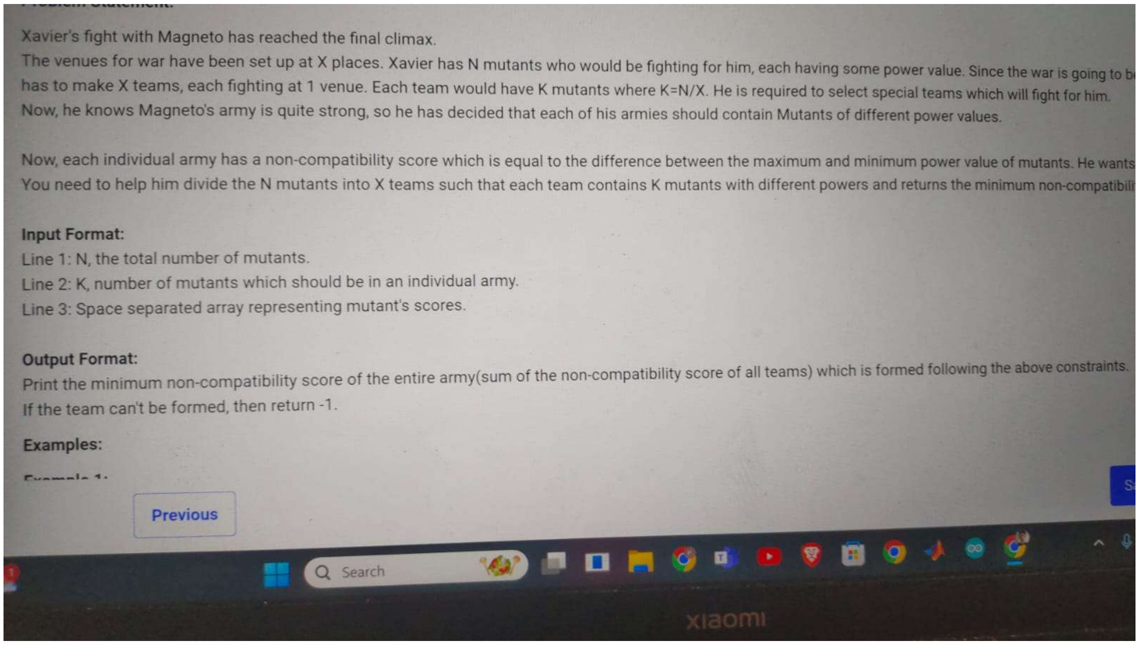


```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <unordered_set>

using namespace std;

int minimum_moves(int number_of_bricks, vector<int>& brick_values) {
    if (number_of_bricks == 1) {
        return 0;
    }

    unordered_map<int, vector<int>> value_to_indices;
    for (int i = 0; i < number_of_bricks; ++i) {
        value_to_indices[brick_values[i]].push_back(i);
    }

    queue<pair<int, int>> q;
    q.push(make_pair(0, 0));
    vector<bool> visited(number_of_bricks, false);
    visited[0] = true;

    while (!q.empty()) {
        pair<int, int> front = q.front();
        q.pop();

        int current_index = front.first;
        int current_distance = front.second;

        if (current_index == number_of_bricks - 1) {
            return current_distance;
        }
```



```
        if (current_index == number_of_bricks - 1) {
            return current_distance;
        }

        // Move to the next brick (i+1)
        if (current_index + 1 < number_of_bricks && !visited[current_index + 1]) {
            visited[current_index + 1] = true;
            q.push(make_pair(current_index + 1, current_distance + 1));
        }

        // Move to the previous brick (i-1)
        if (current_index - 1 >= 0 && !visited[current_index - 1]) {
            visited[current_index - 1] = true;
            q.push(make_pair(current_index - 1, current_distance + 1));
        }

        // Move to any brick with the same value
        int value = brick_values[current_index];
        for (int next_index : value_to_indices[value]) {
            if (!visited[next_index]) {
                visited[next_index] = true;
                q.push(make_pair(next_index, current_distance + 1));
            }
        }

        // Clear the list to prevent redundant checks
        value_to_indices[value].clear();
    }

    return -1; // If there's no way to reach the end (shouldn't happen with valid input)
}

int main() {
```

2. Xavier’s Question:



Xavier's fight with Magneto has reached the final climax.
The venues for war have been set up at X places. Xavier has N mutants who would be fighting for him, each having some power value. Since the war is going to b
has to make X teams, each fighting at 1 venue. Each team would have K mutants where K=N/X. He is required to select special teams which will fight for him.
Now, he knows Magneto's army is quite strong, so he has decided that each of his armies should contain Mutants of different power values.

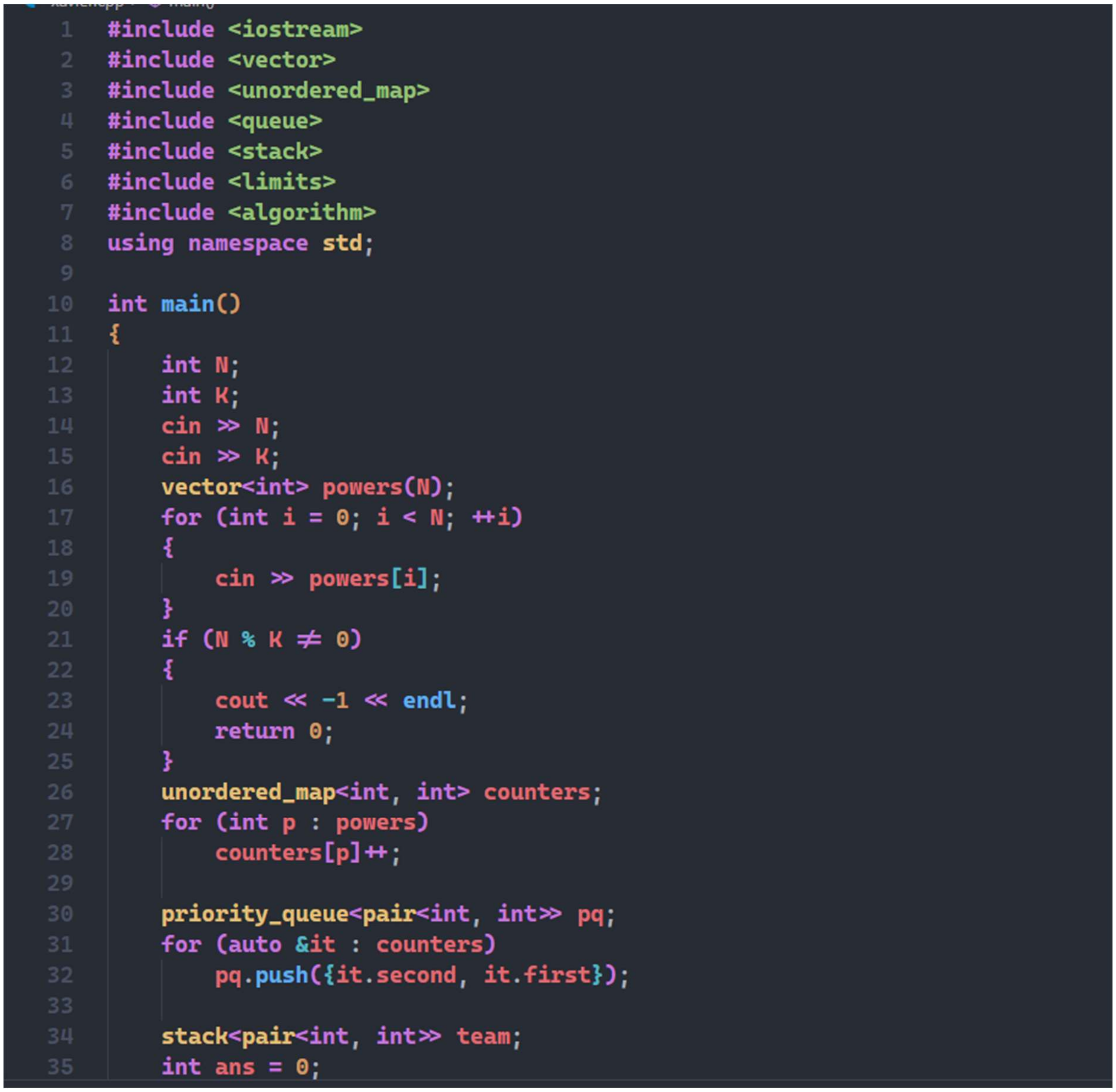
Now, each individual army has a non-compatibility score which is equal to the difference between the maximum and minimum power value of mutants. He wants
You need to help him divide the N mutants into X teams such that each team contains K mutants with different powers and returns the minimum non-compatibili

Input Format:
Line 1: N, the total number of mutants.
Line 2: K, number of mutants which should be in an individual army.
Line 3: Space separated array representing mutant's scores.

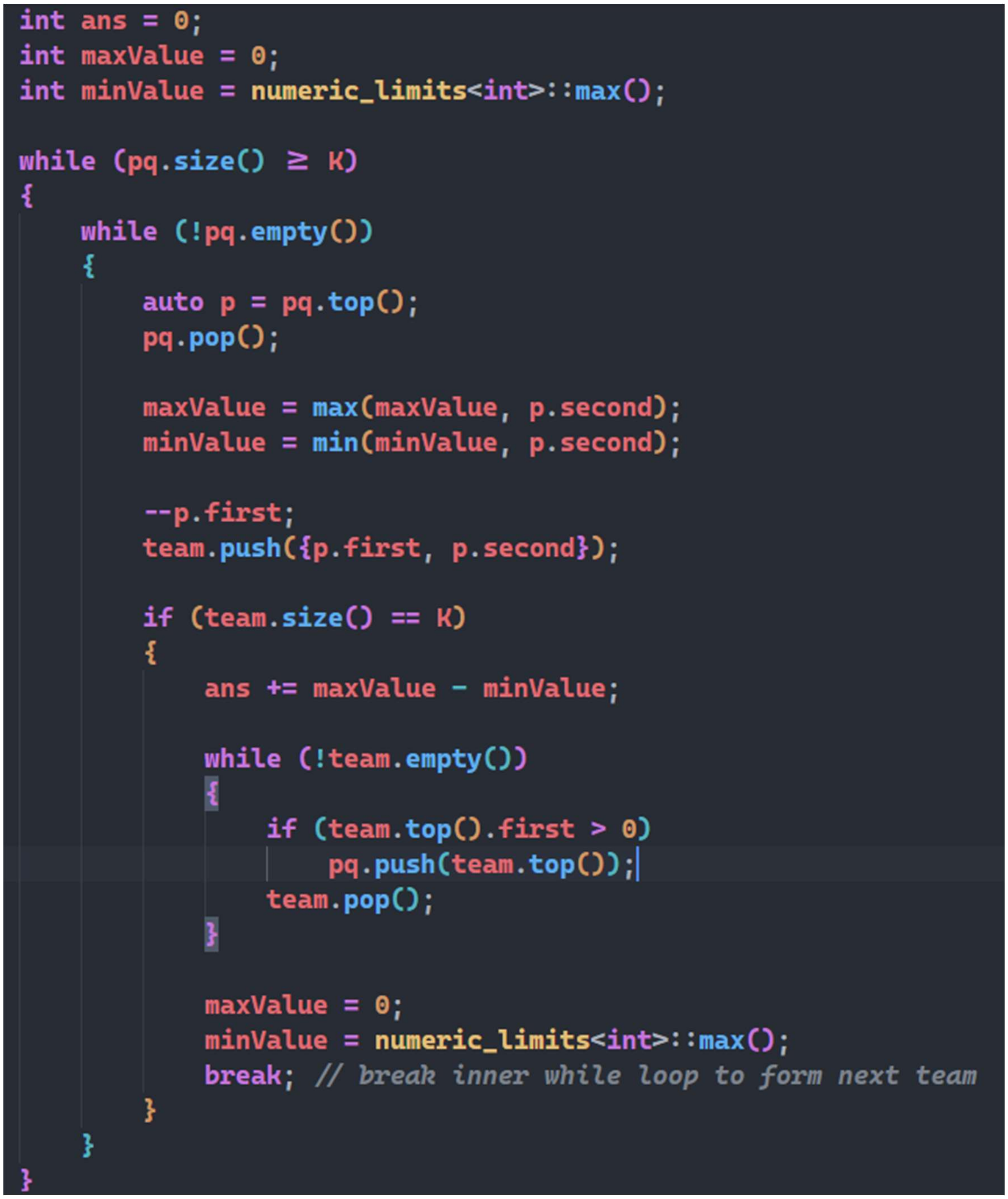
Output Format:
Print the minimum non-compatibility score of the entire army(sum of the non-compatibility score of all teams) which is formed following the above constraints.
If the team can't be formed, then return -1.

Examples:

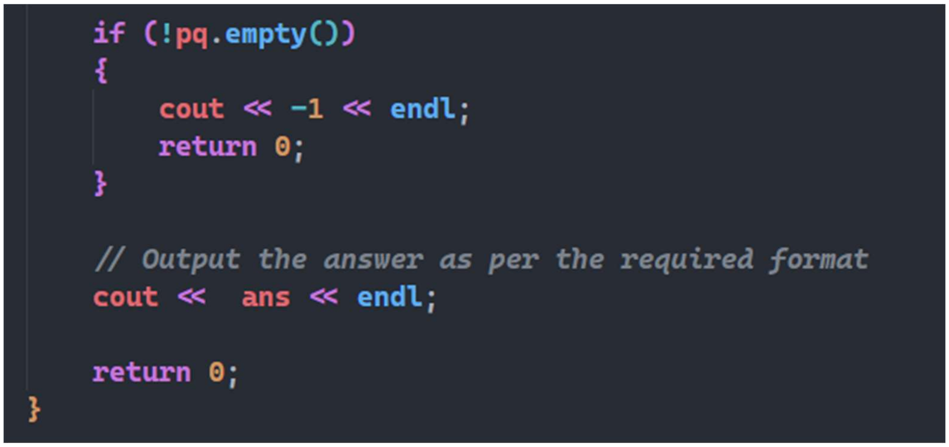
Example 1:



```cpp
#include <iostream>
#include <vector>
#include <unordered_map>
#include <queue>
#include <stack>
#include <limits>
#include <algorithm>
using namespace std;

int main()
{
    int N;
    int K;
    cin >> N;
    cin >> K;
    vector<int> powers(N);
    for (int i = 0; i < N; ++i)
    {
        cin >> powers[i];
    }
    if (N % K != 0)
    {
        cout << -1 << endl;
        return 0;
    }
    unordered_map<int, int> counters;
    for (int p : powers)
        counters[p]++;

    priority_queue<pair<int, int>> pq;
    for (auto &it : counters)
        pq.push({it.second, it.first});

    stack<pair<int, int>> team;
    int ans = 0;
```

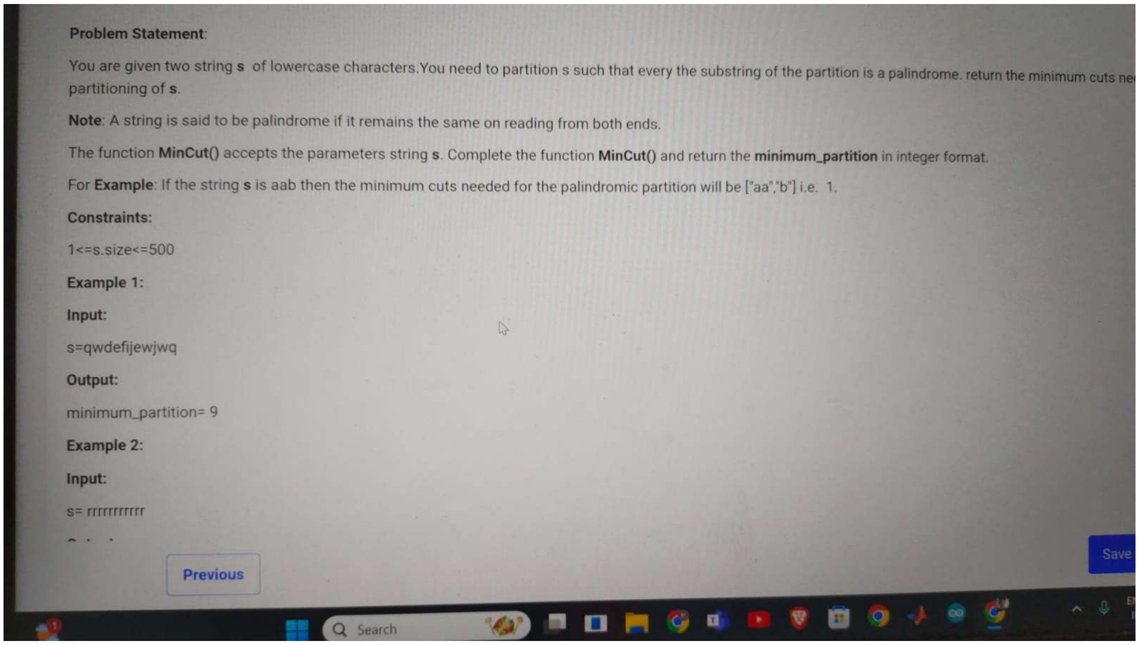


```
int ans = 0;
int maxValue = 0;
int minValue = numeric_limits<int>::max();

while (pq.size() >= K)
{
    while (!pq.empty())
    {
        auto p = pq.top();
        pq.pop();

        maxValue = max(maxValue, p.second);
        minValue = min(minValue, p.second);

        --p.first;
        team.push({p.first, p.second});

        if (team.size() == K)
        {
            ans += maxValue - minValue;

            while (!team.empty())
            {
                if (team.top().first > 0)
                    pq.push(team.top());
                team.pop();
            }

            maxValue = 0;
            minValue = numeric_limits<int>::max();
            break; // break inner while loop to form next team
        }
    }
}
```



```
    if (!pq.empty())
    {
        cout << -1 << endl;
        return 0;
    }

    // Output the answer as per the required format
    cout <<  ans << endl;

    return 0;
}
```

3. Minimum Cut:



Problem Statement:

You are given two string **s** of lowercase characters.You need to partition s such that every the substring of the partition is a palindrome. return the minimum cuts ne partitioning of **s**.

Note: A string is said to be palindrome if it remains the same on reading from both ends.

The function **MinCut()** accepts the parameters string **s**. Complete the function **MinCut()** and return the **minimum_partition** in integer format.

For **Example**: If the string **s** is aab then the minimum cuts needed for the palindromic partition will be ["aa","b"] i.e. 1.

Constraints:

1<=s.size<=500

Example 1:

Input:

s=qwdefijewjwq

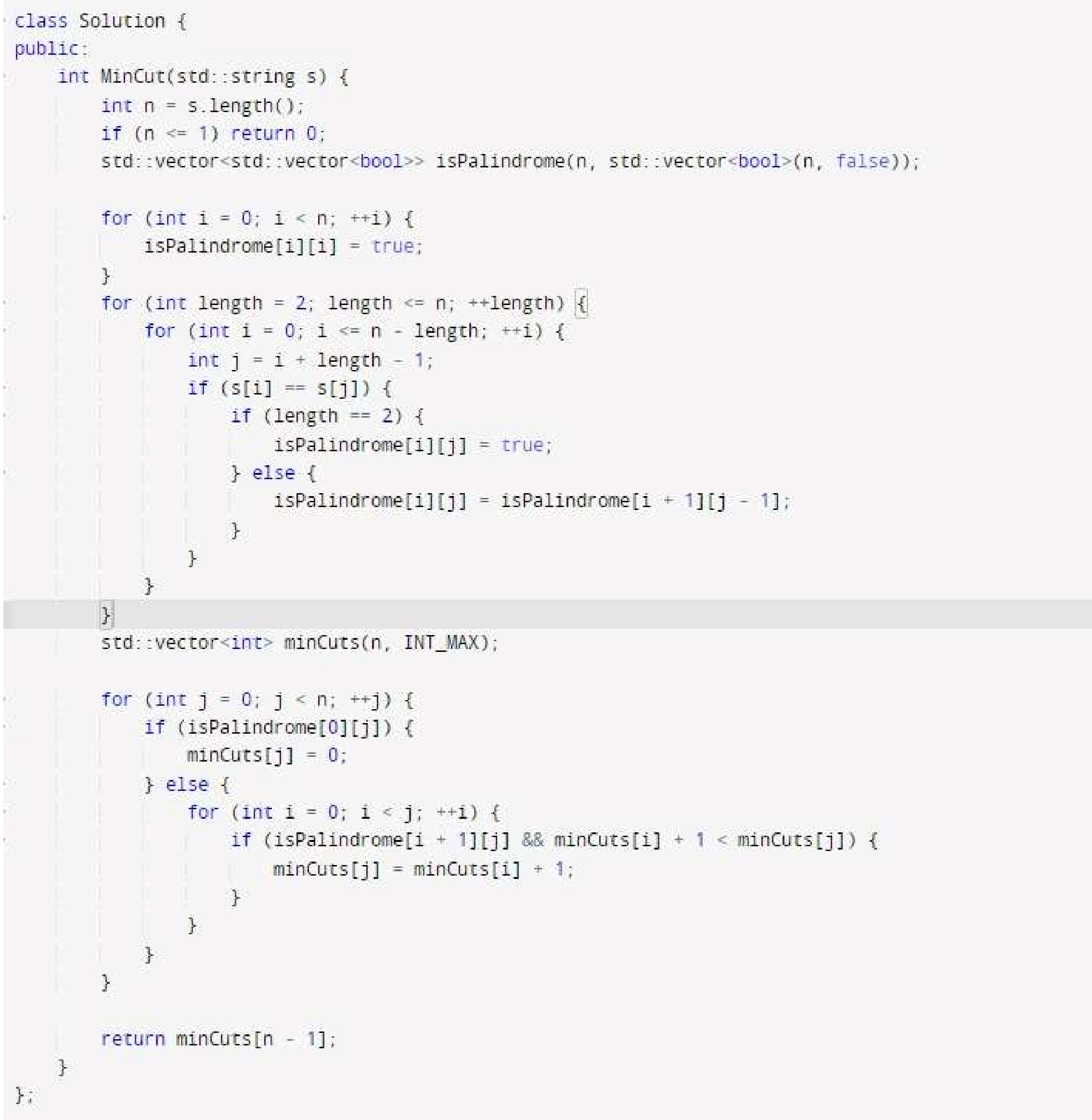
Output:

minimum_partition= 9

Example 2:

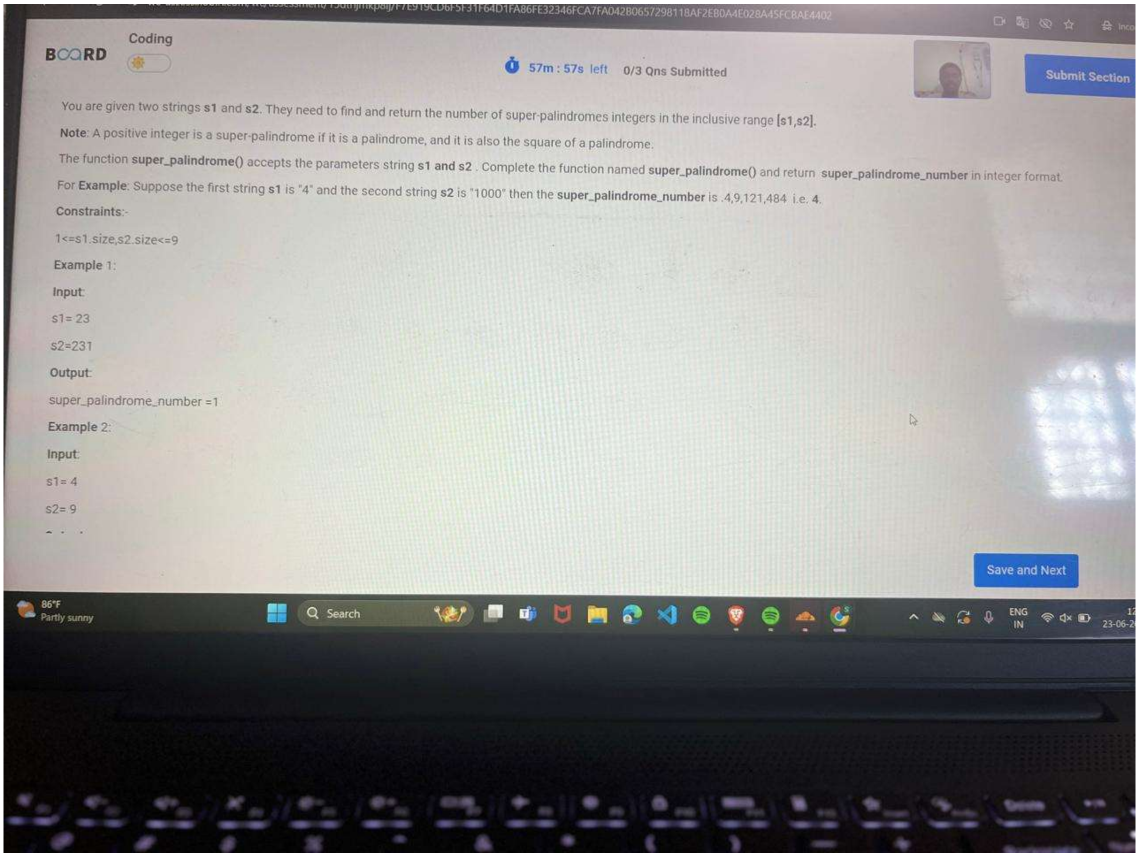
Input:

s= rrrrrrrrrrr



```cpp
class Solution {
public:
    int MinCut(std::string s) {
        int n = s.length();
        if (n <= 1) return 0;
        std::vector<std::vector<bool>> isPalindrome(n, std::vector<bool>(n, false));

        for (int i = 0; i < n; ++i) {
            isPalindrome[i][i] = true;
        }
        for (int length = 2; length <= n; ++length) {
            for (int i = 0; i <= n - length; ++i) {
                int j = i + length - 1;
                if (s[i] == s[j]) {
                    if (length == 2) {
                        isPalindrome[i][j] = true;
                    } else {
                        isPalindrome[i][j] = isPalindrome[i + 1][j - 1];
                    }
                }
            }
        }
        std::vector<int> minCuts(n, INT_MAX);

        for (int j = 0; j < n; ++j) {
            if (isPalindrome[0][j]) {
                minCuts[j] = 0;
            } else {
                for (int i = 0; i < j; ++i) {
                    if (isPalindrome[i + 1][j] && minCuts[i] + 1 < minCuts[j]) {
                        minCuts[j] = minCuts[i] + 1;
                    }
                }
            }
        }

        return minCuts[n - 1];
    }
};
```

4. Super Palindrome:



You are given two strings **s1** and **s2**. They need to find and return the number of super-palindromes integers in the inclusive range **[s1,s2].**

Note: A positive integer is a super-palindrome if it is a palindrome, and it is also the square of a palindrome.

The function **super_palindrome()** accepts the parameters string **s1 and s2** . Complete the function named **super_palindrome()** and return **super_palindrome_number** in integer format.

For **Example**: Suppose the first string **s1** is "4" and the second string **s2** is "1000" then the **super_palindrome_number** is .4,9,121,484 i.e. **4**.

Constraints:-

1<=s1.size,s2.size<=9

Example 1:

Input:

s1= 23

s2=231

Output:

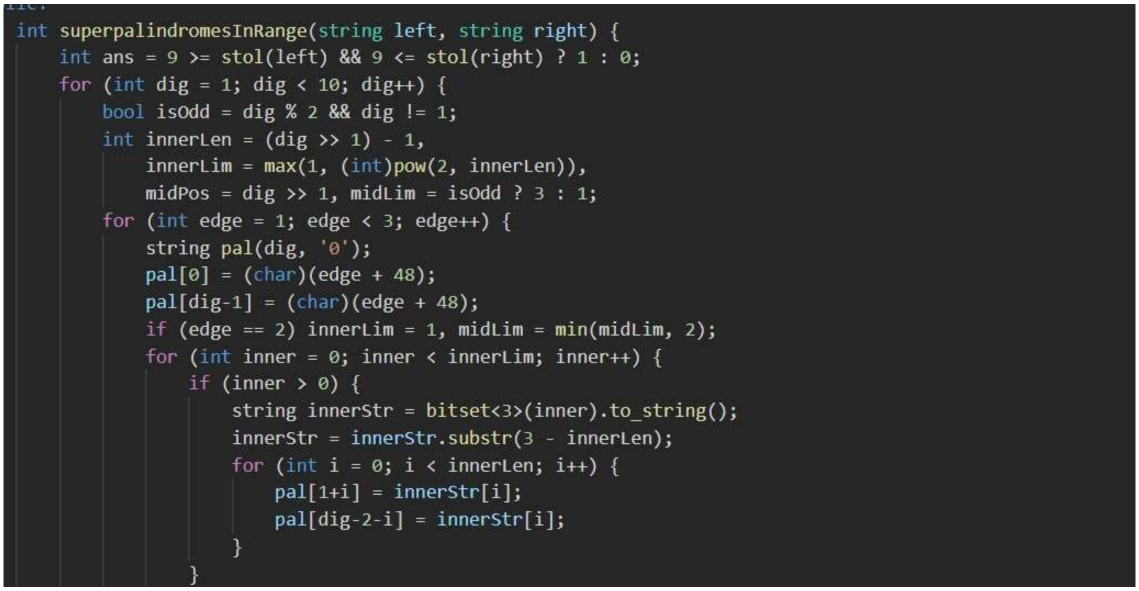
super_palindrome_number =1

Example 2:

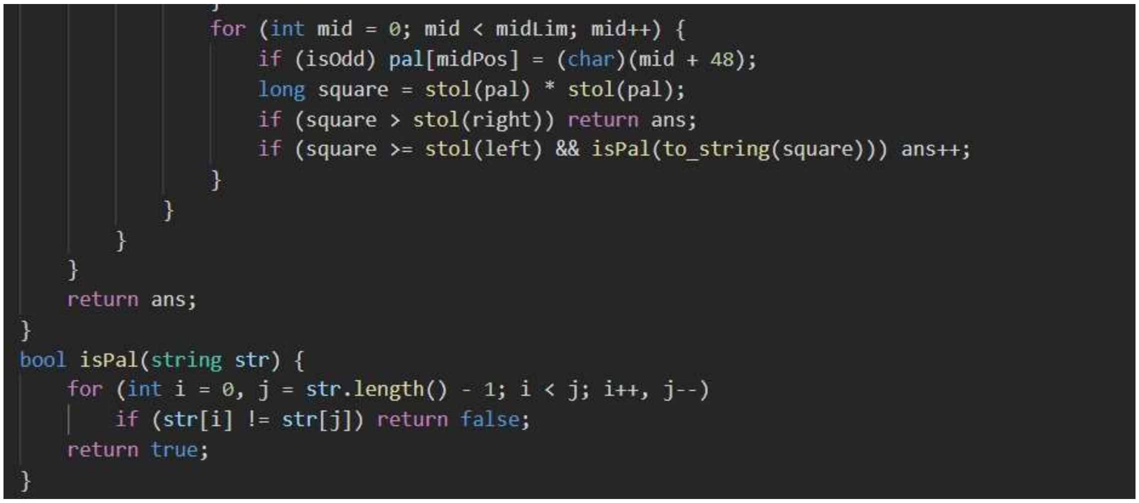
Input:

s1= 4

s2= 9

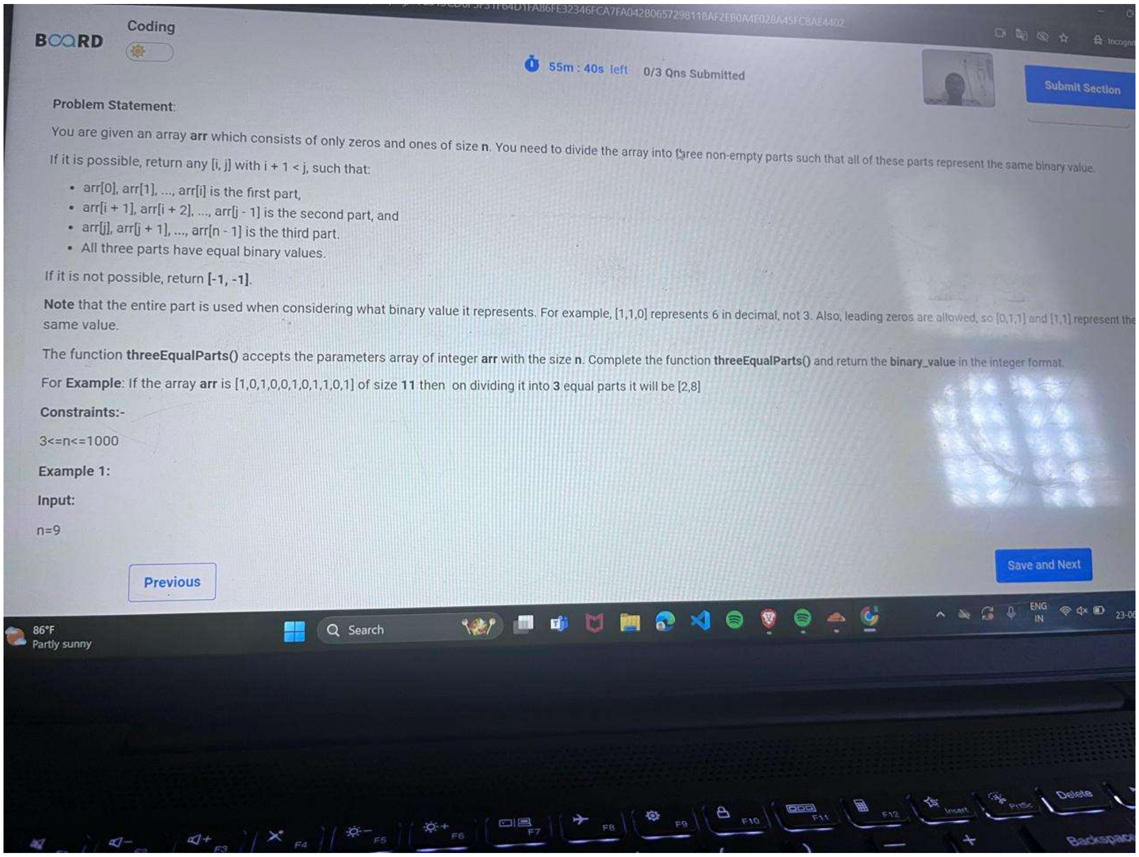


```
int superpalindromesInRange(string left, string right) {
    int ans = 9 >= stol(left) && 9 <= stol(right) ? 1 : 0;
    for (int dig = 1; dig < 10; dig++) {
        bool isOdd = dig % 2 && dig != 1;
        int innerLen = (dig >> 1) - 1,
            innerLim = max(1, (int)pow(2, innerLen)),
            midPos = dig >> 1, midLim = isOdd ? 3 : 1;
        for (int edge = 1; edge < 3; edge++) {
            string pal(dig, '0');
            pal[0] = (char)(edge + 48);
            pal[dig-1] = (char)(edge + 48);
            if (edge == 2) innerLim = 1, midLim = min(midLim, 2);
            for (int inner = 0; inner < innerLim; inner++) {
                if (inner > 0) {
                    string innerStr = bitset<3>(inner).to_string();
                    innerStr = innerStr.substr(3 - innerLen);
                    for (int i = 0; i < innerLen; i++) {
                        pal[1+i] = innerStr[i];
                        pal[dig-2-i] = innerStr[i];
                    }
                }
```



```
                for (int mid = 0; mid < midLim; mid++) {
                    if (isOdd) pal[midPos] = (char)(mid + 48);
                    long square = stol(pal) * stol(pal);
                    if (square > stol(right)) return ans;
                    if (square >= stol(left) && isPal(to_string(square))) ans++;
                }
            }
        }
    }
    return ans;
}
bool isPal(string str) {
    for (int i = 0, j = str.length() - 1; i < j; i++, j--)
        if (str[i] != str[j]) return false;
    return true;
}
```

5. Three Equal Parts:



Problem Statement:

You are given an array **arr** which consists of only zeros and ones of size **n**. You need to divide the array into three non-empty parts such that all of these parts represent the same binary value.

If it is possible, return any [i, j] with i + 1 < j, such that:

- arr[0], arr[1], ..., arr[i] is the first part,
- arr[i + 1], arr[i + 2], ..., arr[j - 1] is the second part, and
- arr[j], arr[j + 1], ..., arr[n - 1] is the third part.
- All three parts have equal binary values.

If it is not possible, return **[-1, -1]**.

Note that the entire part is used when considering what binary value it represents. For example, [1,1,0] represents 6 in decimal, not 3. Also, leading zeros are allowed, so [0,1,1] and [1,1] represent the same value.

The function **threeEqualParts()** accepts the parameters array of integer **arr** with the size **n**. Complete the function **threeEqualParts()** and return the **binary_value** in the integer format.

For **Example**: If the array **arr** is [1,0,1,0,0,1,0,1,1,0,1] of size **11** then on dividing it into **3** equal parts it will be [2,8]

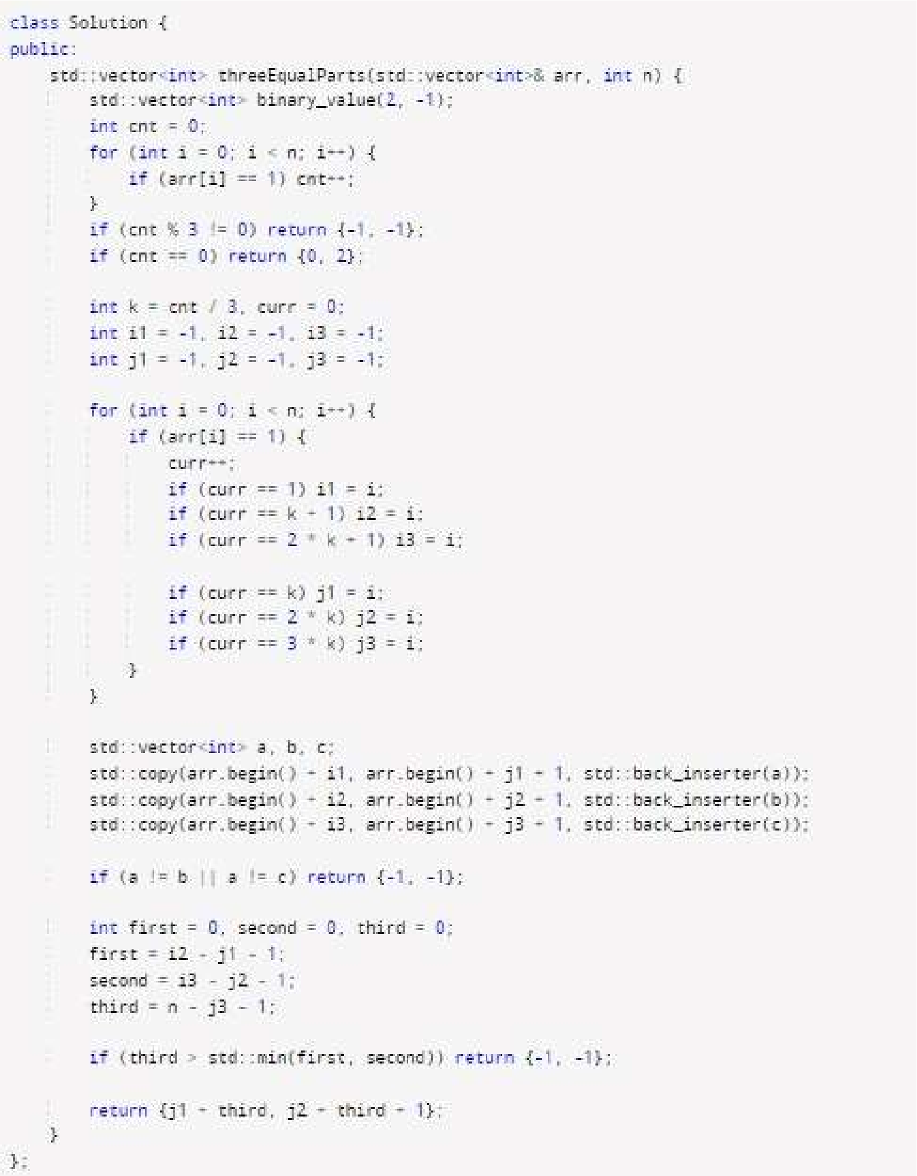
Constraints:-

3<=n<=1000

Example 1:

Input:

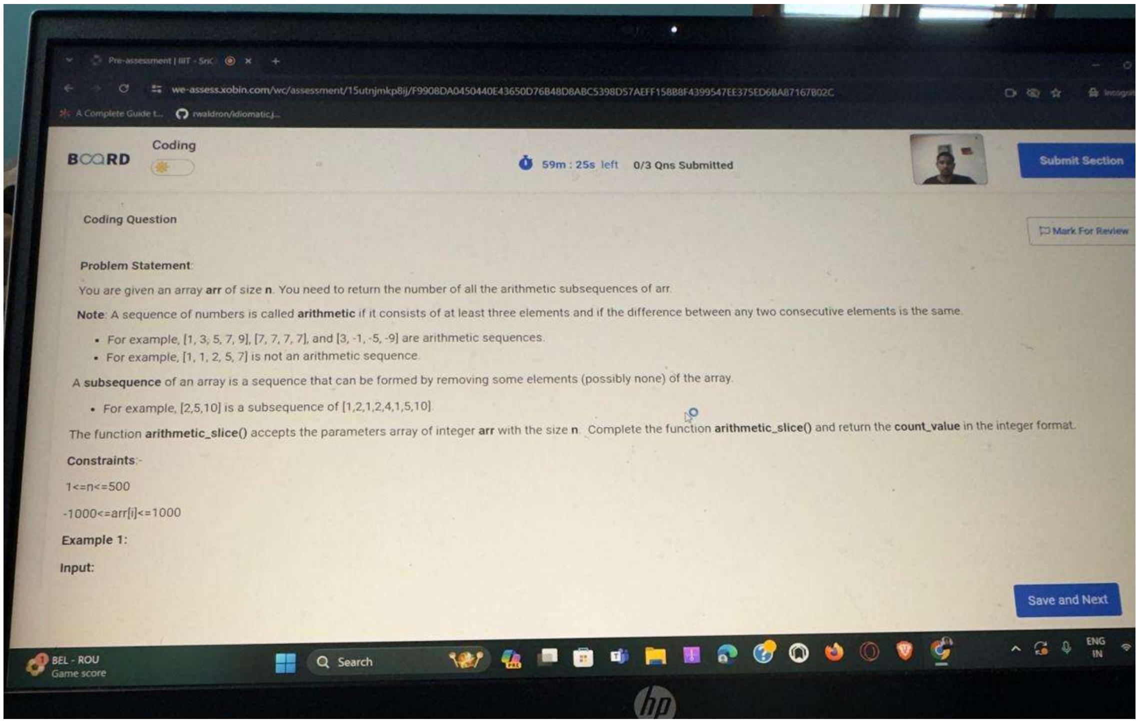
n=9



```
class Solution {
public:
    std::vector<int> threeEqualParts(std::vector<int>& arr, int n) {
        std::vector<int> binary_value(2, -1);
        int cnt = 0;
        for (int i = 0; i < n; i++) {
            if (arr[i] == 1) cnt++;
        }
        if (cnt % 3 != 0) return {-1, -1};
        if (cnt == 0) return {0, 2};

        int k = cnt / 3, curr = 0;
        int i1 = -1, i2 = -1, i3 = -1;
        int j1 = -1, j2 = -1, j3 = -1;

        for (int i = 0; i < n; i++) {
            if (arr[i] == 1) {
                curr++;
                if (curr == 1) i1 = i;
                if (curr == k + 1) i2 = i;
                if (curr == 2 * k + 1) i3 = i;

                if (curr == k) j1 = i;
                if (curr == 2 * k) j2 = i;
                if (curr == 3 * k) j3 = i;
            }
        }

        std::vector<int> a, b, c;
        std::copy(arr.begin() + i1, arr.begin() + j1 + 1, std::back_inserter(a));
        std::copy(arr.begin() + i2, arr.begin() + j2 + 1, std::back_inserter(b));
        std::copy(arr.begin() + i3, arr.begin() + j3 + 1, std::back_inserter(c));

        if (a != b || a != c) return {-1, -1};

        int first = 0, second = 0, third = 0;
        first = i2 - j1 - 1;
        second = i3 - j2 - 1;
        third = n - j3 - 1;

        if (third > std::min(first, second)) return {-1, -1};

        return {j1 + third, j2 + third + 1};
    }
};
```

p

6. Arithmetic Slice



Coding Question

Problem Statement:

You are given an array **arr** of size **n**. You need to return the number of all the arithmetic subsequences of arr.

Note: A sequence of numbers is called **arithmetic** if it consists of at least three elements and if the difference between any two consecutive elements is the same.

- For example, [1, 3, 5, 7, 9], [7, 7, 7, 7], and [3, -1, -5, -9] are arithmetic sequences.
- For example, [1, 1, 2, 5, 7] is not an arithmetic sequence.

A **subsequence** of an array is a sequence that can be formed by removing some elements (possibly none) of the array.

- For example, [2,5,10] is a subsequence of [1,2,1,2,4,1,5,10].

The function **arithmetic_slice()** accepts the parameters array of integer **arr** with the size **n**. Complete the function **arithmetic_slice()** and return the **count_value** in the integer format.

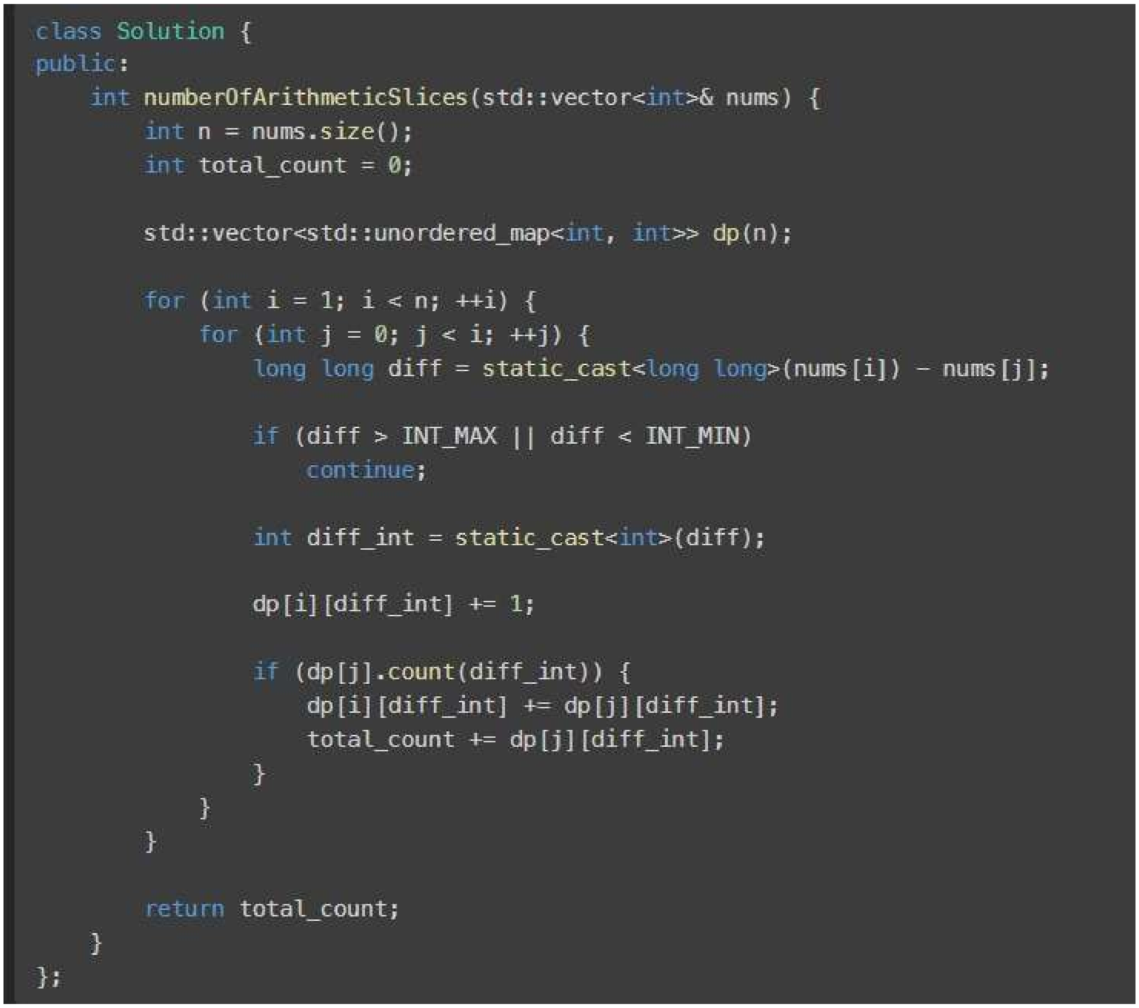
Constraints:-

1<=n<=500

-1000<=arr[i]<=1000

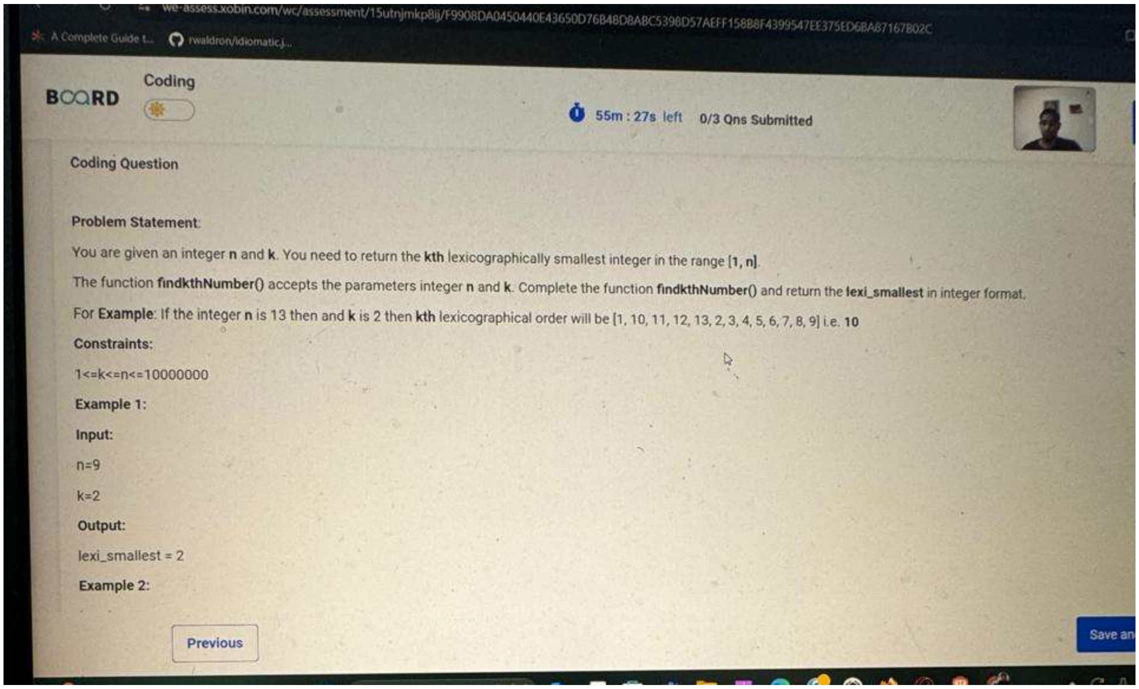
Example 1:

Input:



```
class Solution {
public:
    int numberOfArithmeticSlices(std::vector<int>& nums) {
        int n = nums.size();
        int total_count = 0;

        std::vector<std::unordered_map<int, int>> dp(n);

        for (int i = 1; i < n; ++i) {
            for (int j = 0; j < i; ++j) {
                long long diff = static_cast<long long>(nums[i]) - nums[j];

                if (diff > INT_MAX || diff < INT_MIN)
                    continue;

                int diff_int = static_cast<int>(diff);

                dp[i][diff_int] += 1;

                if (dp[j].count(diff_int)) {
                    dp[i][diff_int] += dp[j][diff_int];
                    total_count += dp[j][diff_int];
                }
            }
        }

        return total_count;
    }
};
```

7. Find Kth Number:



we-assess.xobin.com/wc/assessment/15utnjmkp8ij/F9908DA0450440E43650D76B48D8ABC5398D57AEFF158B8F4399547EE375ED6BA87167B02C
A Complete Guide t...
rwaldron/idiomatic.j...
BOQRD
Coding
55m : 27s left
0/3 Qns Submitted
Previous
Save an

Coding Question

Problem Statement:

You are given an integer **n** and **k**. You need to return the **kth** lexicographically smallest integer in the range **[1, n]**.

The function **findkthNumber()** accepts the parameters integer **n** and **k**. Complete the function **findkthNumber()** and return the **lexi_smallest** in integer format.

For **Example**: If the integer **n** is 13 then and **k** is 2 then **kth** lexicographical order will be [1, 10, 11, 12, 13, 2, 3, 4, 5, 6, 7, 8, 9] i.e. **10**

Constraints:

1<=k<=n<=10000000

Example 1:

Input:

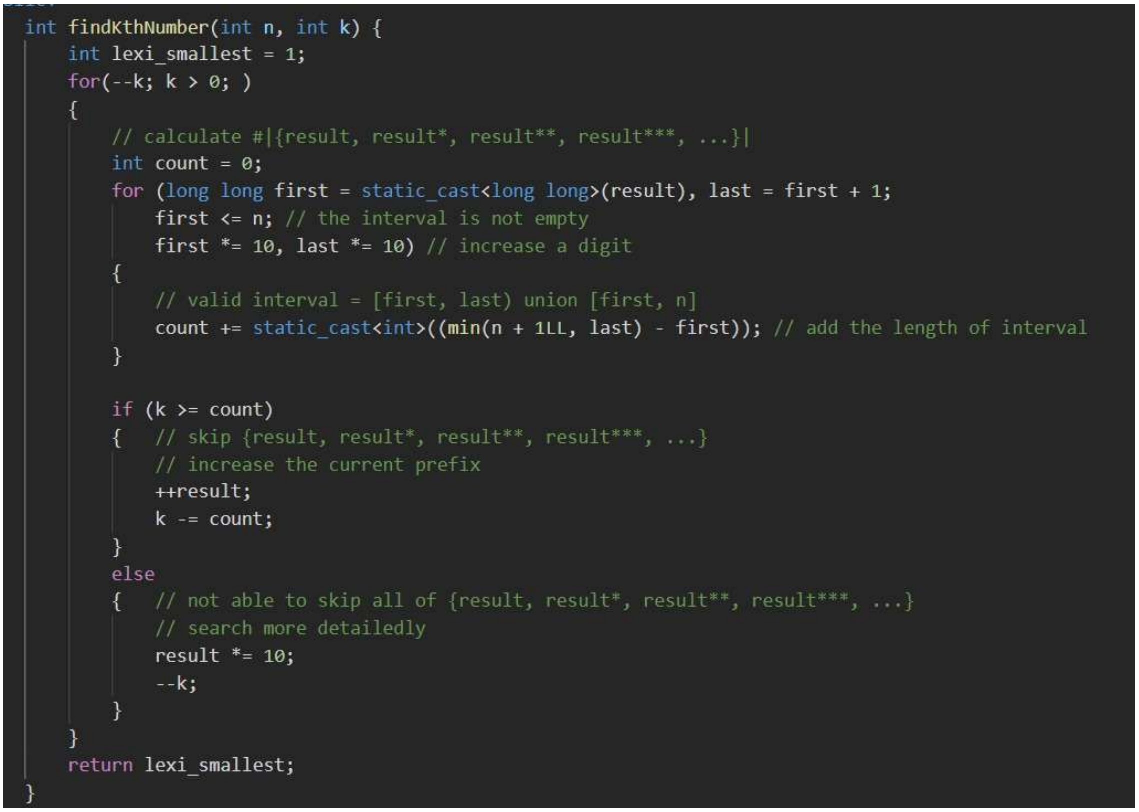
n=9

k=2

Output:

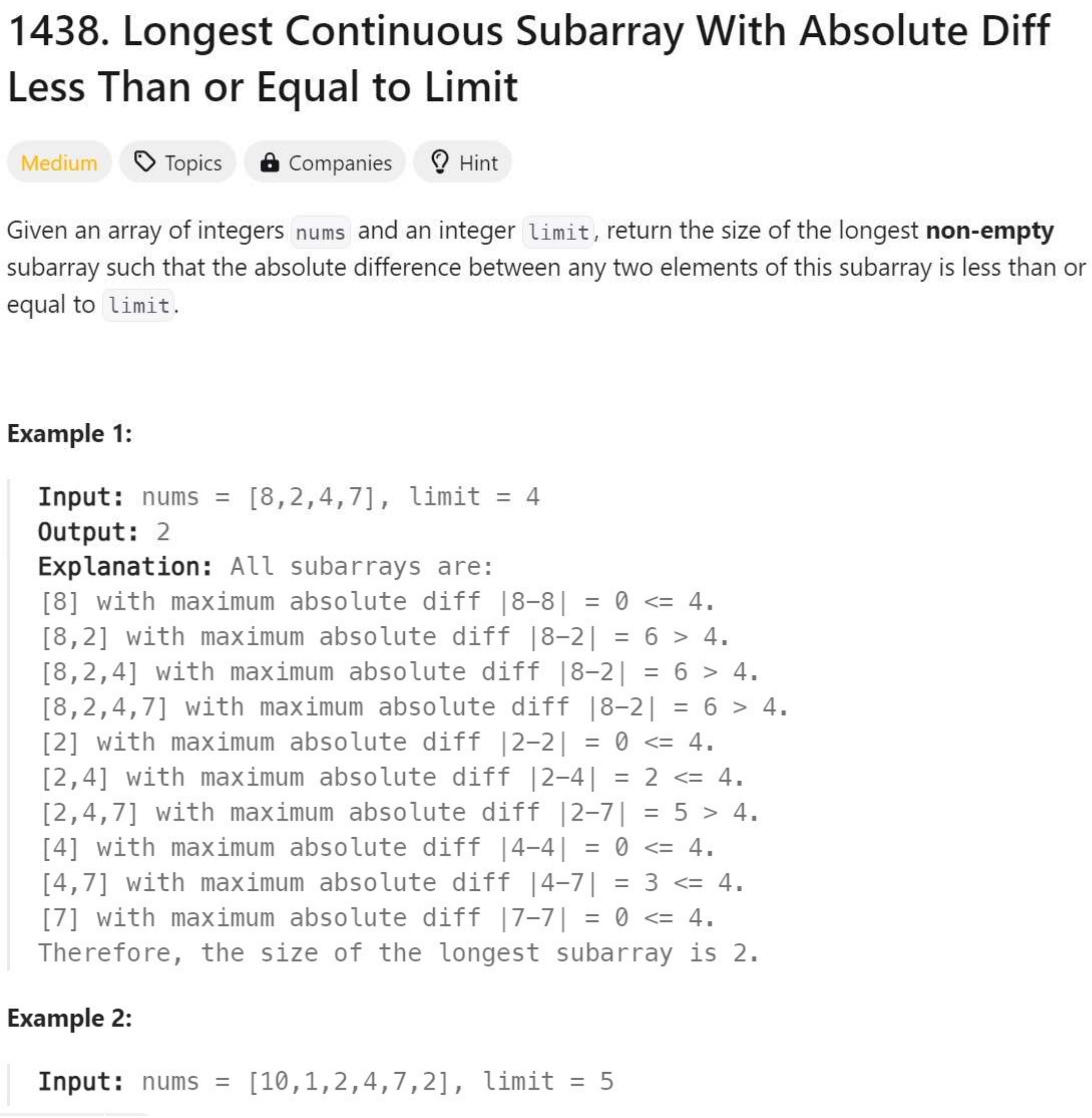
lexi_smallest = 2

Example 2:



```
int findKthNumber(int n, int k) {
    int lexi_smallest = 1;
    for(--k; k > 0; )
    {
        // calculate #|{result, result*, result**, result***, ...}|
        int count = 0;
        for (long long first = static_cast<long long>(result), last = first + 1;
            first <= n; // the interval is not empty
            first *= 10, last *= 10) // increase a digit
        {
            // valid interval = [first, last) union [first, n]
            count += static_cast<int>((min(n + 1LL, last) - first)); // add the length of interval
        }

        if (k >= count)
        {   // skip {result, result*, result**, result***, ...}
            // increase the current prefix
            ++result;
            k -= count;
        }
        else
        {   // not able to skip all of {result, result*, result**, result***, ...}
            // search more detailedly
            result *= 10;
            --k;
        }
    }
    return lexi_smallest;
}
```

8. Longest Subarray



1438. Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit

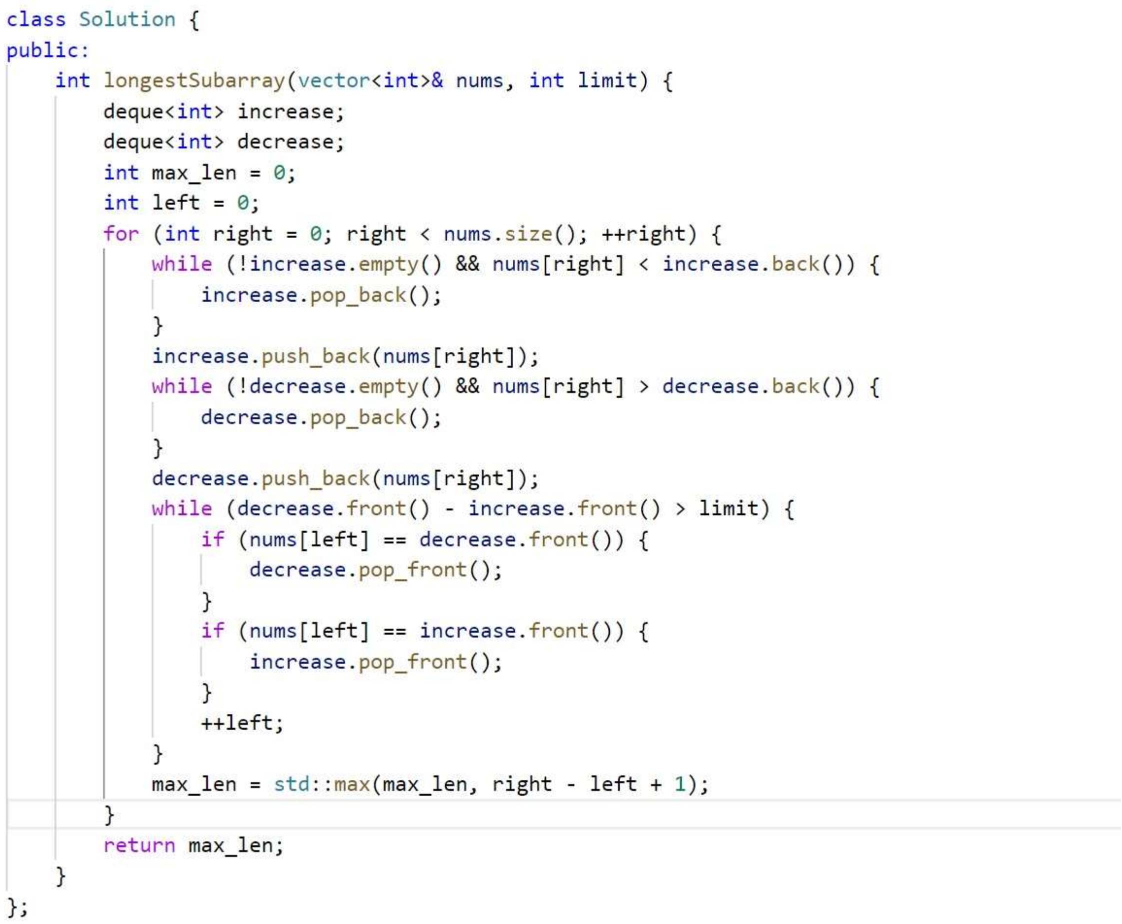
Given an array of integers `nums` and an integer `limit`, return the size of the longest **non-empty** subarray such that the absolute difference between any two elements of this subarray is less than or equal to `limit`.

Example 1:

```
Input: nums = [8,2,4,7], limit = 4
Output: 2
Explanation: All subarrays are:
[8] with maximum absolute diff |8-8| = 0 <= 4.
[8,2] with maximum absolute diff |8-2| = 6 > 4.
[8,2,4] with maximum absolute diff |8-2| = 6 > 4.
[8,2,4,7] with maximum absolute diff |8-2| = 6 > 4.
[2] with maximum absolute diff |2-2| = 0 <= 4.
[2,4] with maximum absolute diff |2-4| = 2 <= 4.
[2,4,7] with maximum absolute diff |2-7| = 5 > 4.
[4] with maximum absolute diff |4-4| = 0 <= 4.
[4,7] with maximum absolute diff |4-7| = 3 <= 4.
[7] with maximum absolute diff |7-7| = 0 <= 4.
Therefore, the size of the longest subarray is 2.
```

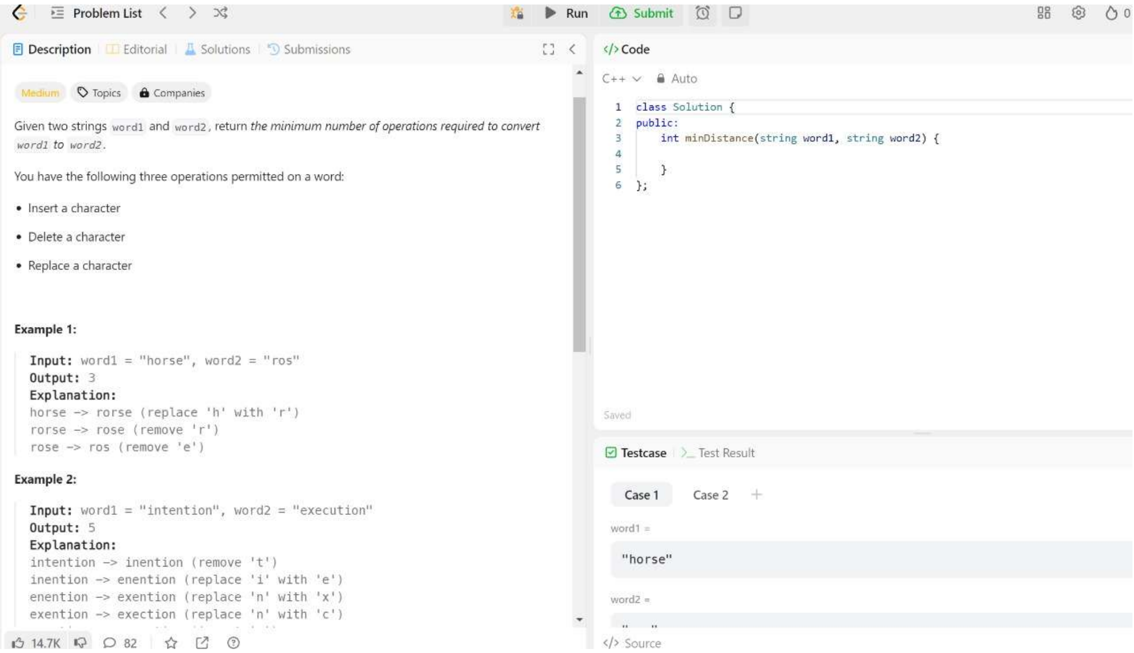
Example 2:

```
Input: nums = [10,1,2,4,7,2], limit = 5
```



```
class Solution {
public:
    int longestSubarray(vector<int>& nums, int limit) {
        deque<int> increase;
        deque<int> decrease;
        int max_len = 0;
        int left = 0;
        for (int right = 0; right < nums.size(); ++right) {
            while (!increase.empty() && nums[right] < increase.back()) {
                increase.pop_back();
            }
            increase.push_back(nums[right]);
            while (!decrease.empty() && nums[right] > decrease.back()) {
                decrease.pop_back();
            }
            decrease.push_back(nums[right]);
            while (decrease.front() - increase.front() > limit) {
                if (nums[left] == decrease.front()) {
                    decrease.pop_front();
                }
                if (nums[left] == increase.front()) {
                    increase.pop_front();
                }
                ++left;
            }
            max_len = std::max(max_len, right - left + 1);
        }
        return max_len;
    }
};
```

9. Minimum Distance



Given two strings `word1` and `word2`, return *the minimum number of operations required to convert `word1` to `word2`*.

You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Example 1:

```
Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')
```

Example 2:

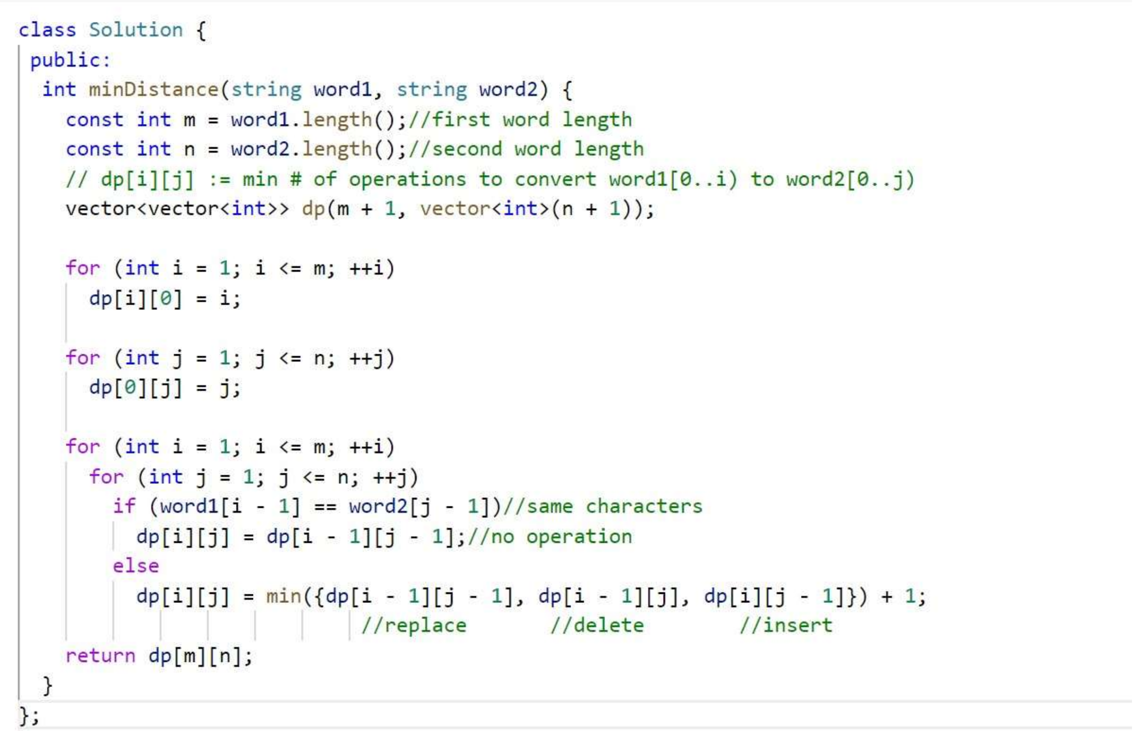
```
Input: word1 = "intention", word2 = "execution"
Output: 5
Explanation:
intention -> inention (remove 't')
inention -> enention (replace 'i' with 'e')
enention -> exention (replace 'n' with 'x')
exention -> exection (replace 'n' with 'c')
```

```
class Solution {
public:
    int minDistance(string word1, string word2) {

    }
};
```

word1 =

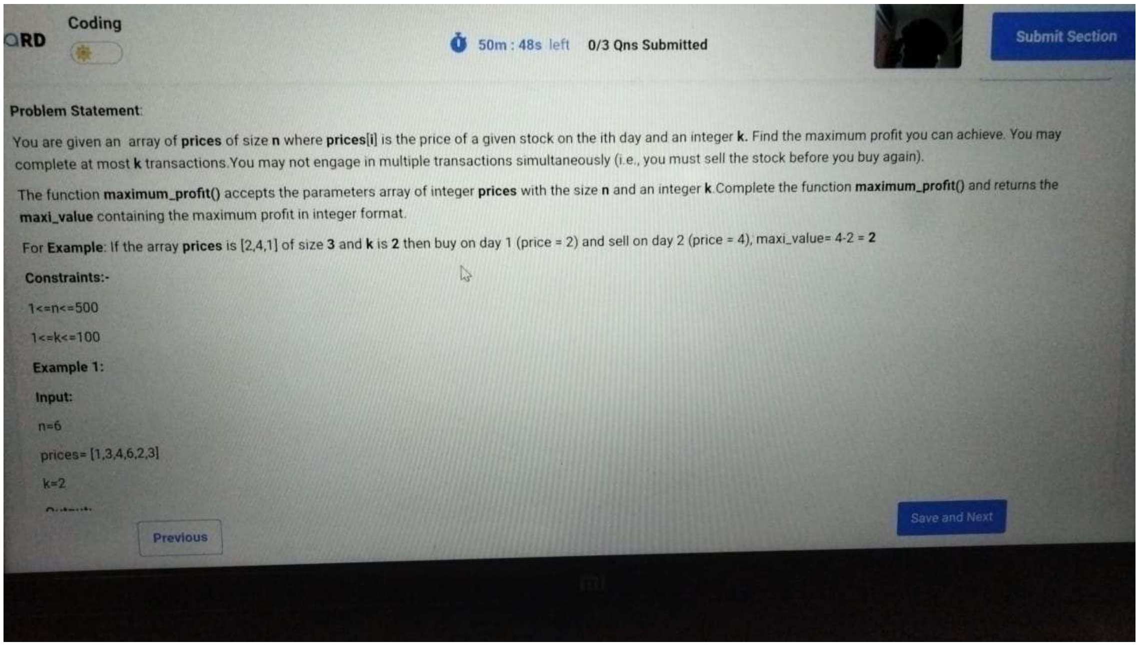
"horse"

word2 =



```
class Solution {
 public:
  int minDistance(string word1, string word2) {
    const int m = word1.length();//first word length
    const int n = word2.length();//second word length
    // dp[i][j] := min # of operations to convert word1[0..i) to word2[0..j)
    vector<vector<int>> dp(m + 1, vector<int>(n + 1));

    for (int i = 1; i <= m; ++i)
      dp[i][0] = i;

    for (int j = 1; j <= n; ++j)
      dp[0][j] = j;

    for (int i = 1; i <= m; ++i)
      for (int j = 1; j <= n; ++j)
        if (word1[i - 1] == word2[j - 1])//same characters
          dp[i][j] = dp[i - 1][j - 1];//no operation
        else
          dp[i][j] = min({dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]}) + 1;
                             //replace       //delete        //insert
    return dp[m][n];
  }
};
```

10. Maximum Profit



Problem Statement:

You are given an array of **prices** of size **n** where **prices[i]** is the price of a given stock on the ith day and an integer **k.** Find the maximum profit you can achieve. You may complete at most **k** transactions.You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

The function **maximum_profit()** accepts the parameters array of integer **prices** with the size **n** and an integer **k**.Complete the function **maximum_profit()** and returns the **maxi_value** containing the maximum profit in integer format.

For **Example**: If the array **prices** is [2,4,1] of size **3** and **k** is **2** then buy on day 1 (price = 2) and sell on day 2 (price = 4), maxi_value= 4-2 = **2**

Constraints:-

1<=n<=500

1<=k<=100

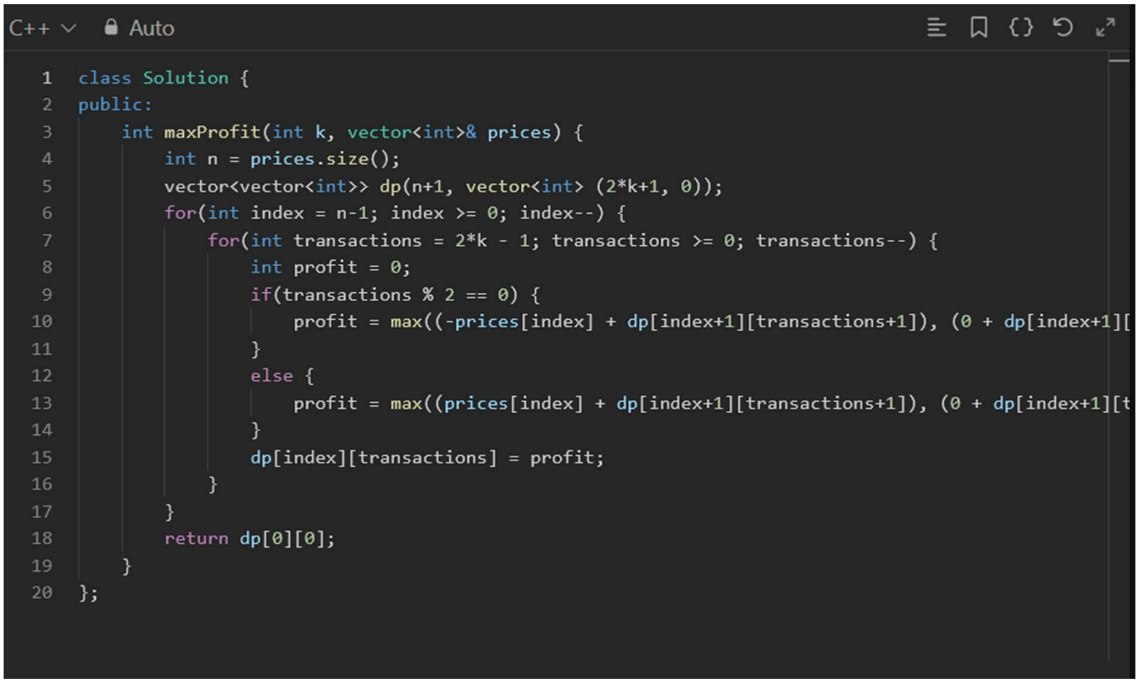
Example 1:

Input:

n=6

prices= [1,3,4,6,2,3]

k=2



```cpp
class Solution {
public:
    int maxProfit(int k, vector<int>& prices) {
        int n = prices.size();
        vector<vector<int>> dp(n+1, vector<int> (2*k+1, 0));
        for(int index = n-1; index >= 0; index--) {
            for(int transactions = 2*k - 1; transactions >= 0; transactions--) {
                int profit = 0;
                if(transactions % 2 == 0) {
                    profit = max((-prices[index] + dp[index+1][transactions+1]), (0 + dp[index+1][
                }
                else {
                    profit = max((prices[index] + dp[index+1][transactions+1]), (0 + dp[index+1][t
                }
                dp[index][transactions] = profit;
            }
        }
        return dp[0][0];
    }
};
```