Problem.1

You are given an integer array **nums** and an integer **k**. Determine whether there exists a contiguous subarray of length exactly **k** that contains at least two equal elements.If such a subarray exists, return true; otherwise, return false.

Example 1:

Input: nums =[1,2,3,2,4,5], k = 3

Output: true

Explanation: The subarray [2,3,2] (indices 1–3) contains the duplicate element 2

Example 2:

Input: nums = [1,2,3,4,5,6], k = 3

Output: false

Explanation: All contiguous subarrays of length 3 contain distinct elements.

Problem.2

Given an integer array **nums** and two integers **k** and **t**, determine whether there exist two distinct indices **i** and **j** such that:

abs(i - j) <= k

abs(nums[i] - nums[j]) <= t

Example 1:

Input: nums = [1,5,9,3,1], k = 2, t = 2

Output: true

Explanation: Indices 1 and 3: abs(1-3) = 2 ≤ 2, abs(5-3) = 2 ≤ 2

Example 2:

Input: nums = [1,10,20,30,40], k = 2, t = 5

Output: false

Explanation: Within any sliding window of length 3, the difference between any two elements is greater than 5.

Tips:

1 <= nums.length <= 10^4

-10^9 <= nums[i] ,nums[j]<= 10^9

0 <= k <= 10000

0 <= t <= 10^9

Problem.3

You are given an integer array **nums** and two integers **k** and **f**. Determine whether there exists a contiguous subarray of length at most **k** in which some element appears at least **f** times.

Example 1:

Input: nums = [1,2,2,3,2,4], k = 4, f = 3

Output:true

Explanation: The contiguous subarray [2,2,3,2] (indices 1–4, length 4) contains the element 2 appearing 3 times.

Example 2:

Input: nums = [1,1,1,2,2,2], k = 2, f = 3

Output: false

Explanation: In any contiguous subarray of length ≤ 2, the maximum frequency of any element is 2.

Tips:

1 <= nums.length <= 10^4

-10^9 <= nums[i] <= 10^9

1 <= k <= nums.length

2 <= f <= k

Problem 4

The Catalan numbers are a sequence of natural numbers that appear in various counting problems in combinatorics.

They can be defined recursively as:

$$C_0 = 1, \quad C_{n+1} = \sum_{i=0}^{n} C_i \times C_{n-i}$$

Tasks

For each implementation below:

1. **Correctness –** Is the implementation correct?

   If not, what needs to be fixed?

2. **Characteristics and Use Case –**

   Describe the characteristics of this implementation, its advantages, and the types of situations it is most suitable for.

3. **New Knowledge –**

   Identify any C/C++ programming syntax or algorithmic details you were not familiar with before, and briefly explain them.

4. **Code Style –**

   Add comments to each function following the **Google C++ Coding Style**.

5. **Your Version –**

   After analyzing all ten versions, write your own implementation of the Catalan number function that you consider the **most elegant and effective**.

10 Implementations of the Catalan Number

1	<pre>// Version 1: Simple recursive implementation
	int catalan_recursive(int n) {
	    if (n <= 1) return 1;
	    int res = 0;
	    for (int i = 0; i < n; ++i) {
	        res += catalan_recursive(i) * catalan_recursive(n - 1 - i);
	    }
	    return res;
	}</pre>

2	<pre>// Version 2: Iterative dynamic programming
	#include <vector>
	int catalan_iterative(int n) {
	    if (n <= 1) return 1;
	    std::vector<int> dp(n + 1, 0);
	    dp[0] = dp[1] = 1;
	    for (int i = 2; i <= n; ++i) {
	        for (int j = 0; j < i; ++j)
	            dp[i] += dp[j] * dp[i - 1 - j]; }
	    return dp[n];}</pre>

3	<pre>// Version 3: Recursive implementation with memoization
	#include <unordered_map>
	
	long long catalan_memoization(int n, std::unordered_map<int, long long>& memo) {
	    if (n <= 1) return 1;
	    if (memo.find(n) != memo.end()) return memo[n];
	    long long res = 0;
	    for (int i = 0; i < n; ++i)
	        res += catalan_memoization(i, memo) * catalan_memoization(n - 1 - i, memo);
	    return memo[n] = res;
	}</pre>

4	<pre>// Version 4: Using Boost multiprecision for large n
	#include <boost/multiprecision/cpp_int.hpp>
	#include <vector>
	
	boost::multiprecision::cpp_int catalan_bigint(int n) {
	    using boost::multiprecision::cpp_int;
	    if (n <= 1) return 1;
	    std::vector<cpp_int> dp(n + 1);
	    dp[0] = dp[1] = 1;</pre>

<table>
<tr>
<td></td>
<td>

```
    for (int i = 2; i <= n; ++i) {
        dp[i] = 0;
        for (int j = 0; j < i; ++j)
            dp[i] += dp[j] * dp[i - 1 - j];
    }
    return dp[n];
}
```

</td>
</tr>
<tr>
<td>5</td>
<td>

```
// Version 5: Using factorial and combinatorial formula
#include <cmath>

long long factorial(int n) {
    long long res = 1;
    for (int i = 2; i <= n; ++i) res *= i;
    return res;
}

long long catalan_formula(int n) {
    if (n <= 1) return 1;
    return factorial(2 * n) / (factorial(n + 1) * factorial(n));
}
```

</td>
</tr>
<tr>
<td>6</td>
<td>

```
// Version 6: Compile-time computation using constexpr
constexpr long long catalan_constexpr(int n) {
    if (n <= 1) return 1;
    long long dp[64] = {0}; // supports n <= 63
    dp[0] = dp[1] = 1;
    for (int i = 2; i <= n; ++i) {
        dp[i] = 0;
        for (int j = 0; j < i; ++j)
            dp[i] += dp[j] * dp[i - 1 - j];
    }
    return dp[n];
}
```

</td>
</tr>
<tr>
<td>7</td>
<td>

```
// Version 7: Parallel recursive computation using async
#include <future>

long long catalan_parallel(int n) {
    if (n <= 1) return 1;
    std::vector<std::future<long long>> futures;
    long long res = 0;
    for (int i = 0; i < n; ++i) {
        futures.push_back(std::async(std::launch::async, catalan_parallel, i));
```

</td>
</tr>
</table>

	}
	for (int i = 0; i < n; ++i) {
	res += futures[i].get() * catalan_parallel(n - 1 - i);
	}
	return res;
	}

8	// Version 8: Safe implementation using std::optional
	#include <optional>
	#include <vector>
	
	std::optional<long long> catalan_safe(int n) {
	if (n < 0) return std::nullopt;
	if (n <= 1) return 1;
	std::vector<long long> dp(n + 1);
	dp[0] = dp[1] = 1;
	for (int i = 2; i <= n; ++i)
	for (int j = 0; j < i; ++j)
	dp[i] += dp[j] * dp[i - 1 - j];
	return dp[n];
	}

9	// Version 9: Template metaprogramming version
	template<int N>
	struct Catalan {
	static constexpr long long value = ([]() constexpr {
	long long sum = 0;
	for (int i = 0; i < N; ++i)
	sum += Catalan<i>::value * Catalan<N - 1 - i>::value;
	return sum;
	})();
	};
	
	template<> struct Catalan<0> { static constexpr long long value = 1; };
	template<> struct Catalan<1> { static constexpr long long value = 1; };

10	// Version 10: Conceptual matrix-based recurrence demonstration
	#include <array>
	
	using Matrix2x2 = std::array<std::array<long long, 2>, 2>;
	
	Matrix2x2 matrix_multiply(const Matrix2x2& a, const Matrix2x2& b) {
	Matrix2x2 result = {{{0, 0}, {0, 0}}};
	for (int i = 0; i < 2; ++i)

	for (int j = 0; j < 2; ++j)
	    for (int k = 0; k < 2; ++k)
	        result[i][j] += a[i][k] * b[k][j];
	return result;
	}
	
	long long catalan_matrix(int n) {
	    if (n <= 1) return 1;
	    // This version illustrates the concept of matrix recurrence
	    Matrix2x2 base = {{{1, 1}, {1, 0}}};
	    Matrix2x2 result = {{{1, 0}, {0, 1}}};
	    for (int i = 0; i < n; ++i)
	        result = matrix_multiply(result, base);
	    return result[0][0];
	}