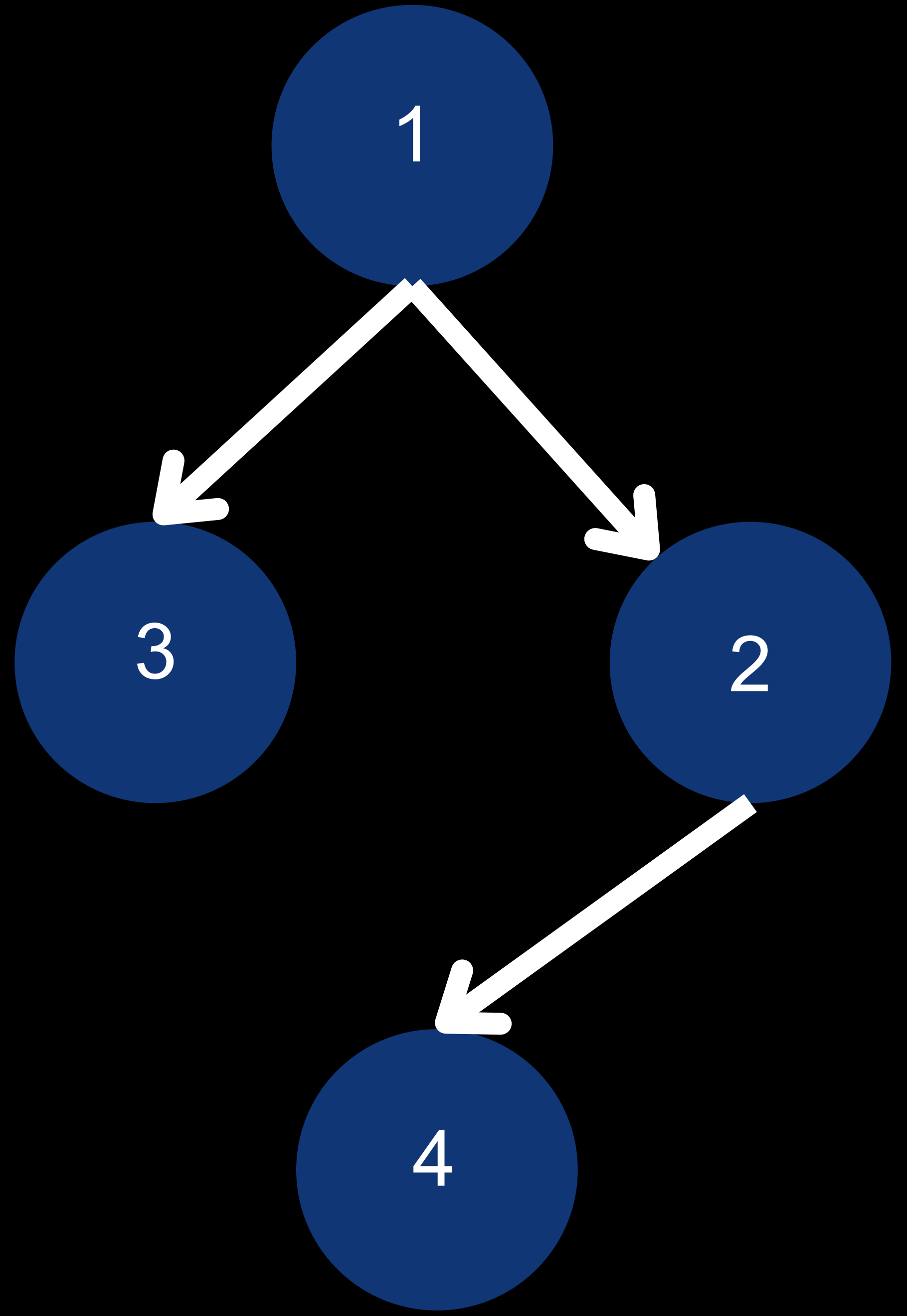
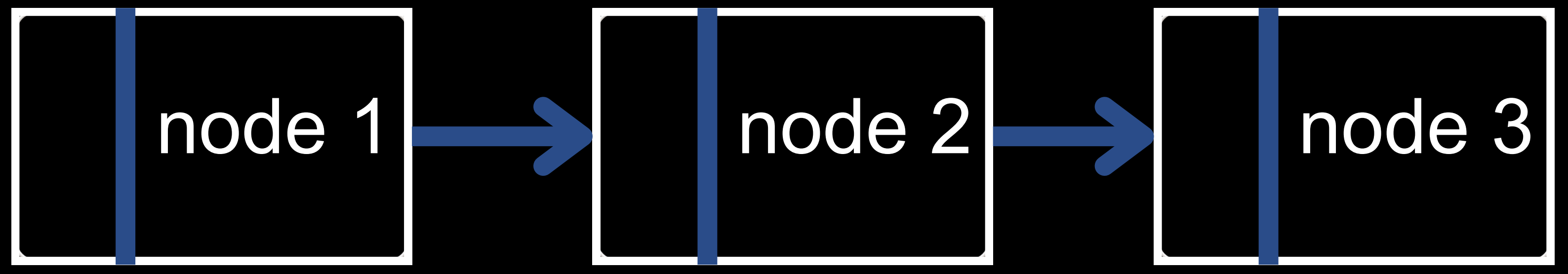
DATA STRUCTURE





essammohamedst1@gmail.com

Complexity analysis

What is Complexity Analysis?

Complexity analysis helps us understand how much time or space an algorithm will need as the size of the input grows.

Asymptotic Notations

There are several types of asymptotic notations, including Big O, Big Omega, and Big Theta. Each notation provides a different perspective on the growth rate of a function.

Big-Oh (O) Notation:

It represents the maximum amount of time or space required by an algorithm considering all input values. It represents the upper limit of an algorithm’s execution time or space, offering insight into its worst-case complexity.

Big-Omega (Ω) notation:

It represents the minimum amount of time or space required by an algorithm considering all input values.

Big-Theta (Θ) notation

It represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

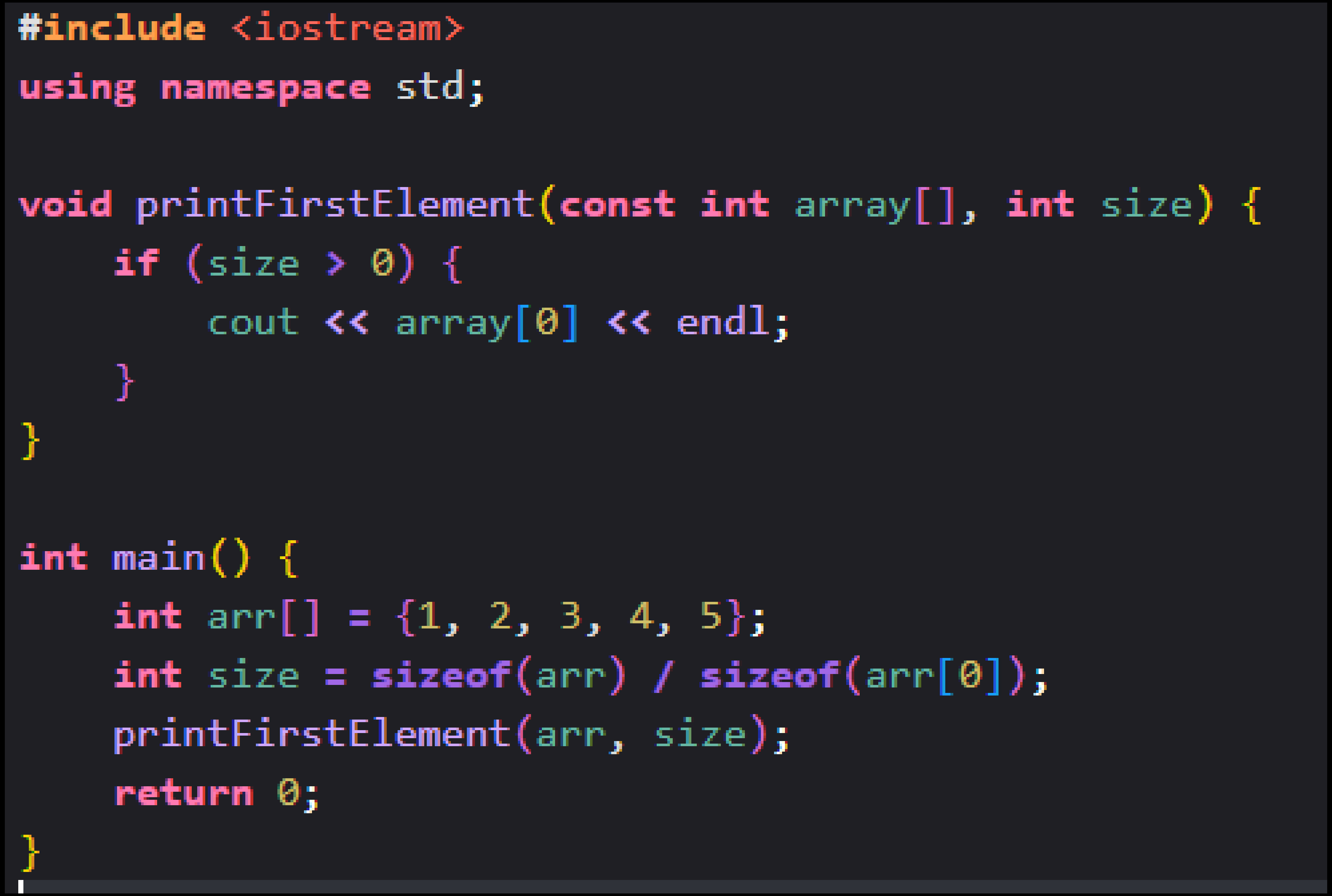
Time Complexity?

Time complexity measures the amount of time an algorithm takes to run as a function of the input size.

Algorithms with lower time complexity are generally more desirable as they can handle larger input sizes with reasonable runtimes.

Constant Time: O(1)

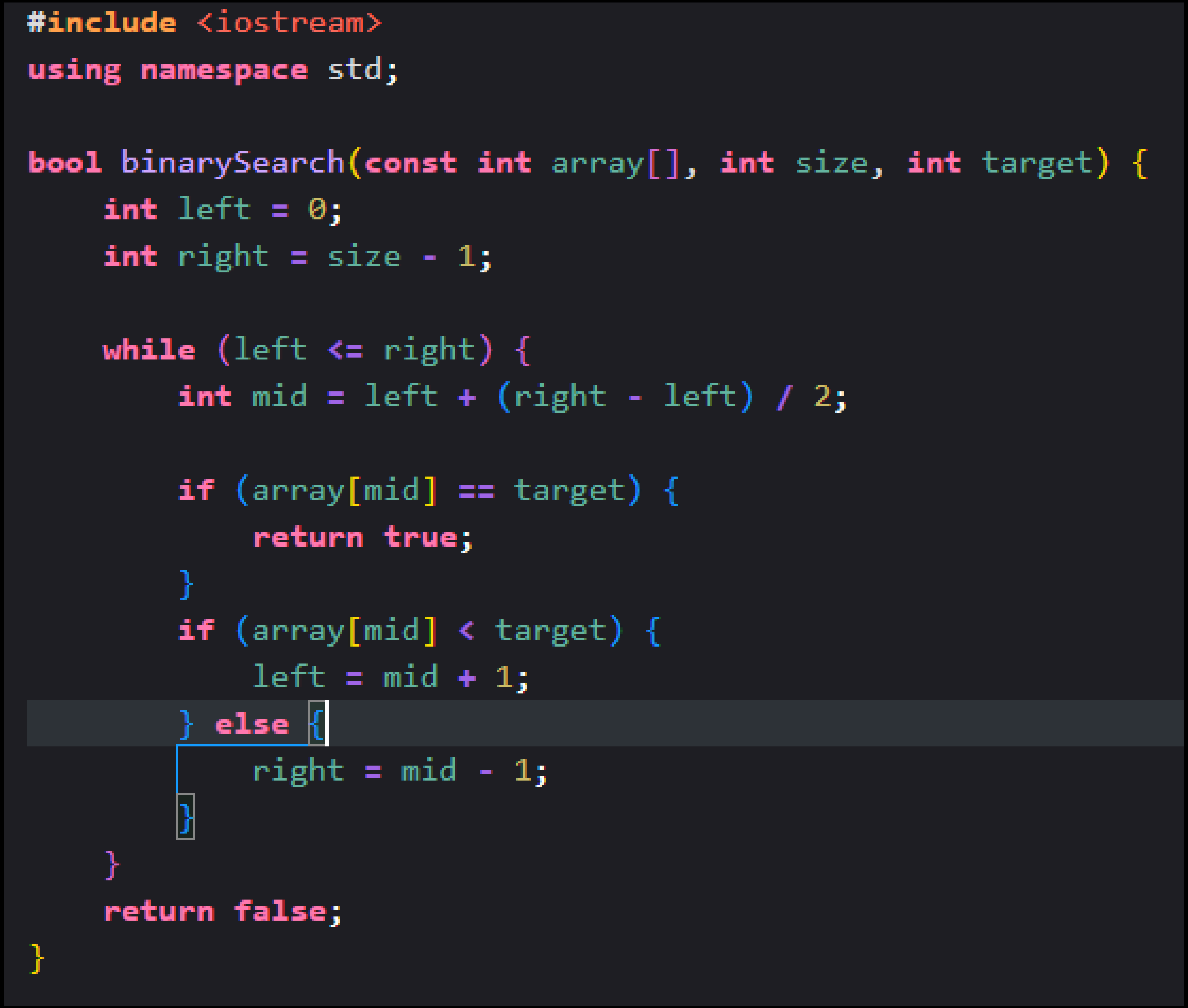
The execution time does not depend on the size of the input. No matter how large the input is, the algorithm takes the same amount of time to complete.



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

void printFirstElement(const int array[], int size) {
    if (size > 0) {
        cout << array[0] << endl;
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    printFirstElement(arr, size);
    return 0;
}
```

Logarithmic Time: O(log n)

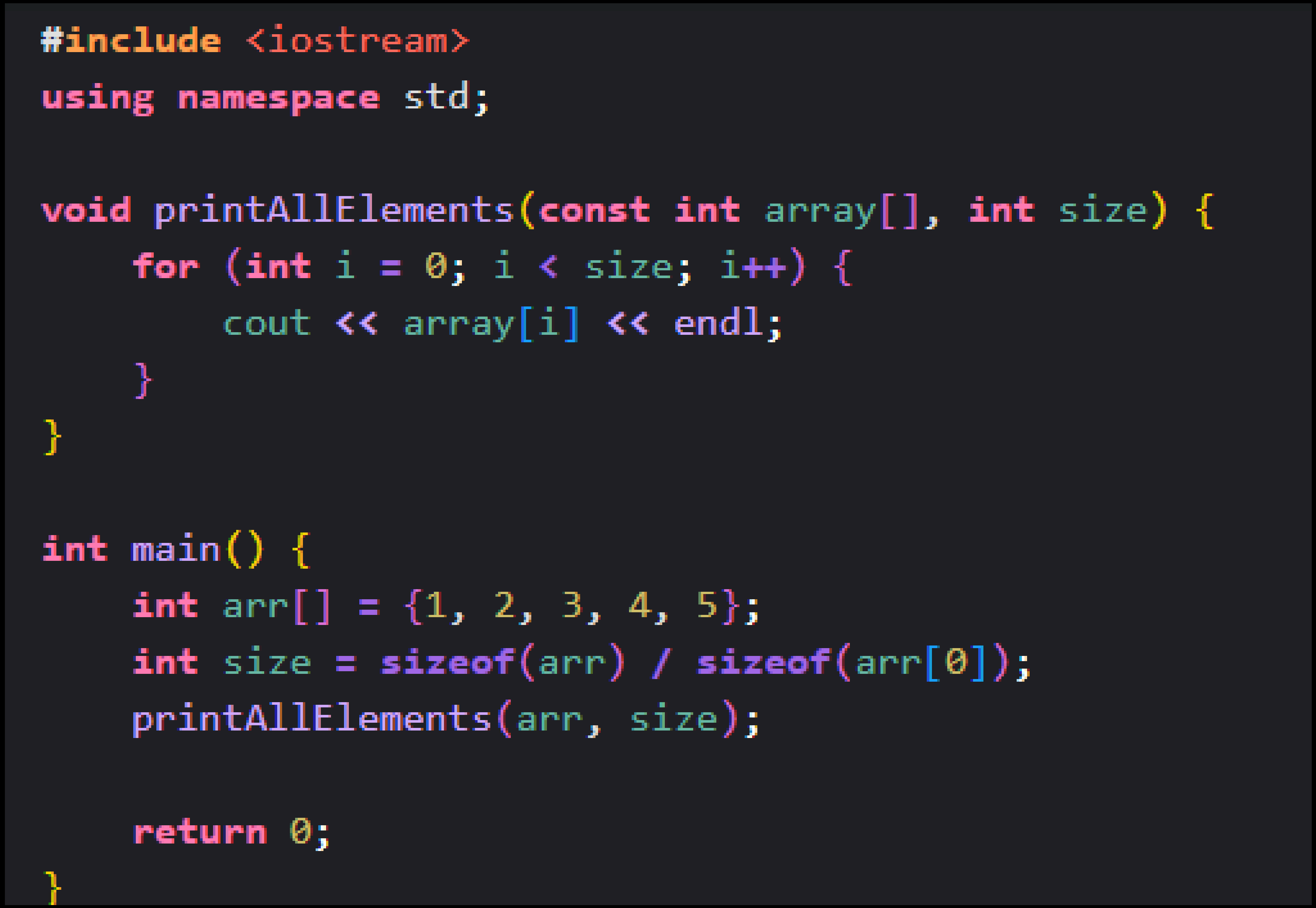
The execution time grows logarithmically with the input size.



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

bool binarySearch(const int array[], int size, int target) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (array[mid] == target) {
            return true;
        }
        if (array[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return false;
}
```

Linear Time: O(n)

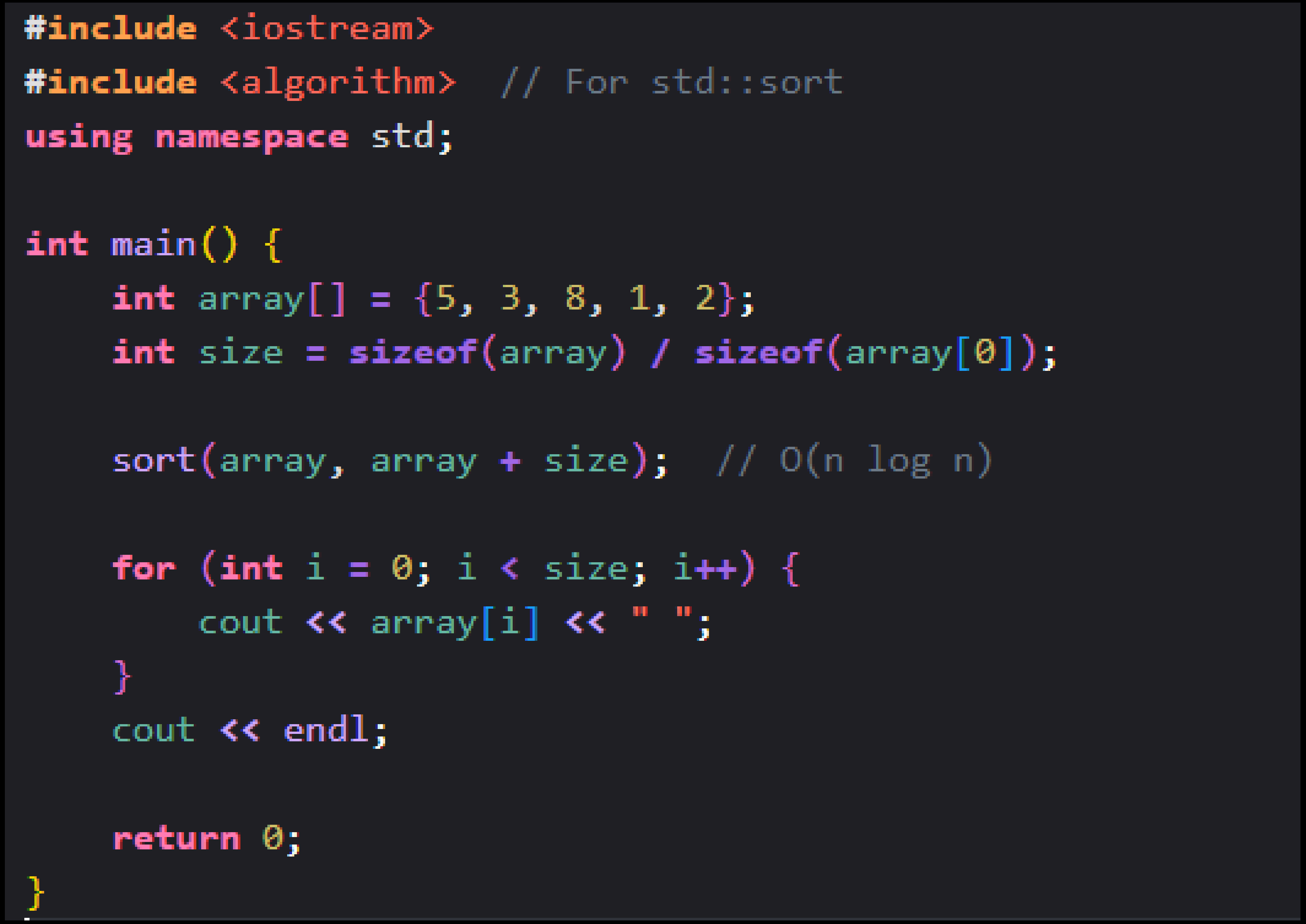
The execution time grows linearly with the input size. For each additional element in the input, the algorithm requires a proportional amount of time to process it.



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

void printAllElements(const int array[], int size) {
    for (int i = 0; i < size; i++) {
        cout << array[i] << endl;
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    printAllElements(arr, size);

    return 0;
}
```

Linearithmic Time: O(n log n)

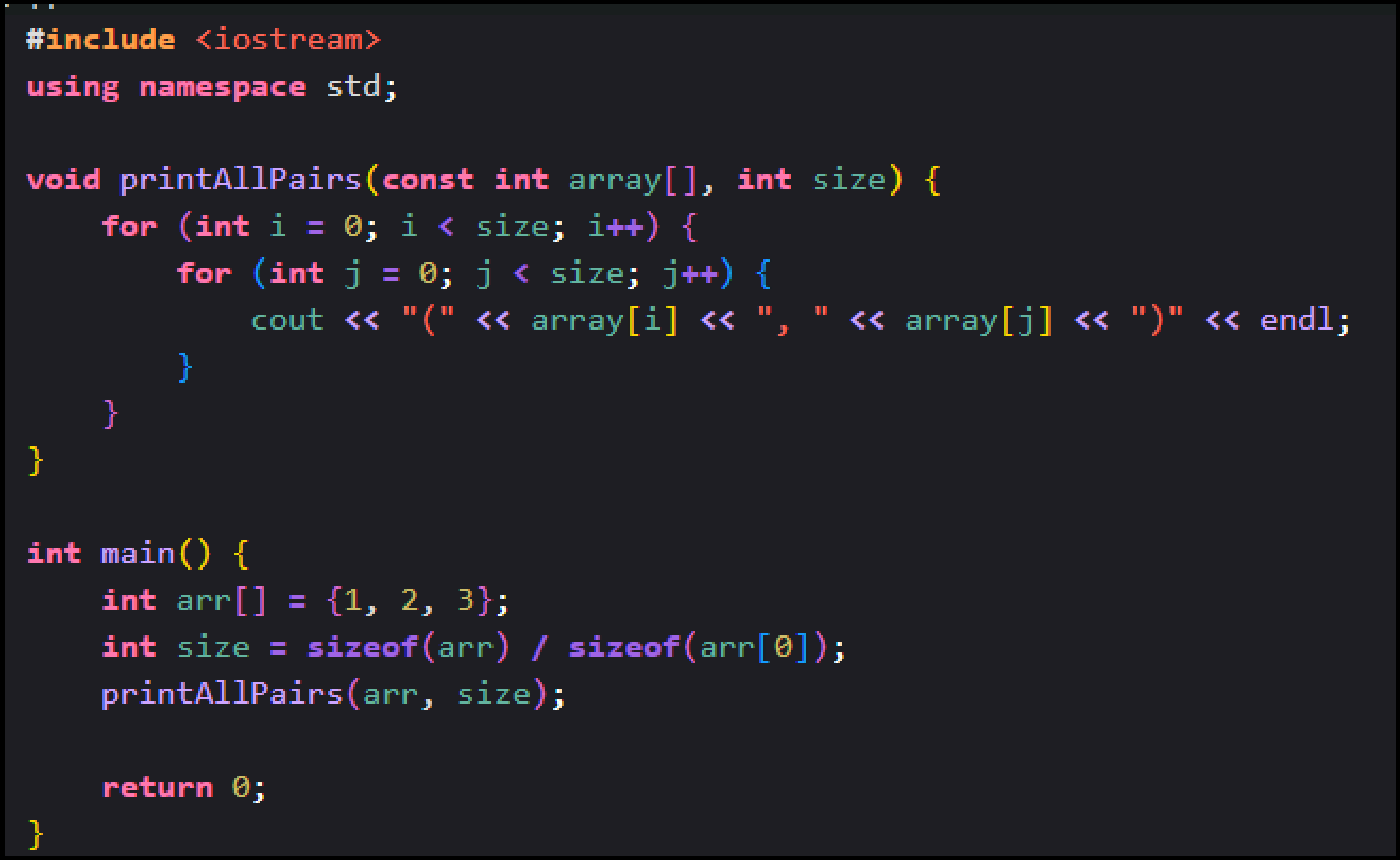
The algorithm's runtime grows in proportion to the size of the input multiplied by the logarithm of the input size. Many efficient sorting algorithms, like Merge Sort and Quick Sort, have linearithmic time complexity.



```cpp
#include <iostream>
#include <algorithm>  // For std::sort
using namespace std;

int main() {
    int array[] = {5, 3, 8, 1, 2};
    int size = sizeof(array) / sizeof(array[0]);

    sort(array, array + size);  // O(n log n)

    for (int i = 0; i < size; i++) {
        cout << array[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Quadratic Time: $O(n^2)$

The execution time grows quadratically with the input size. This is typical of algorithms with nested loops.



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

void printAllPairs(const int array[], int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            cout << "(" << array[i] << ", " << array[j] << ")" << endl;
        }
    }
}

int main() {
    int arr[] = {1, 2, 3};
    int size = sizeof(arr) / sizeof(arr[0]);
    printAllPairs(arr, size);

    return 0;
}
```