Lecture 18

Classic Problems

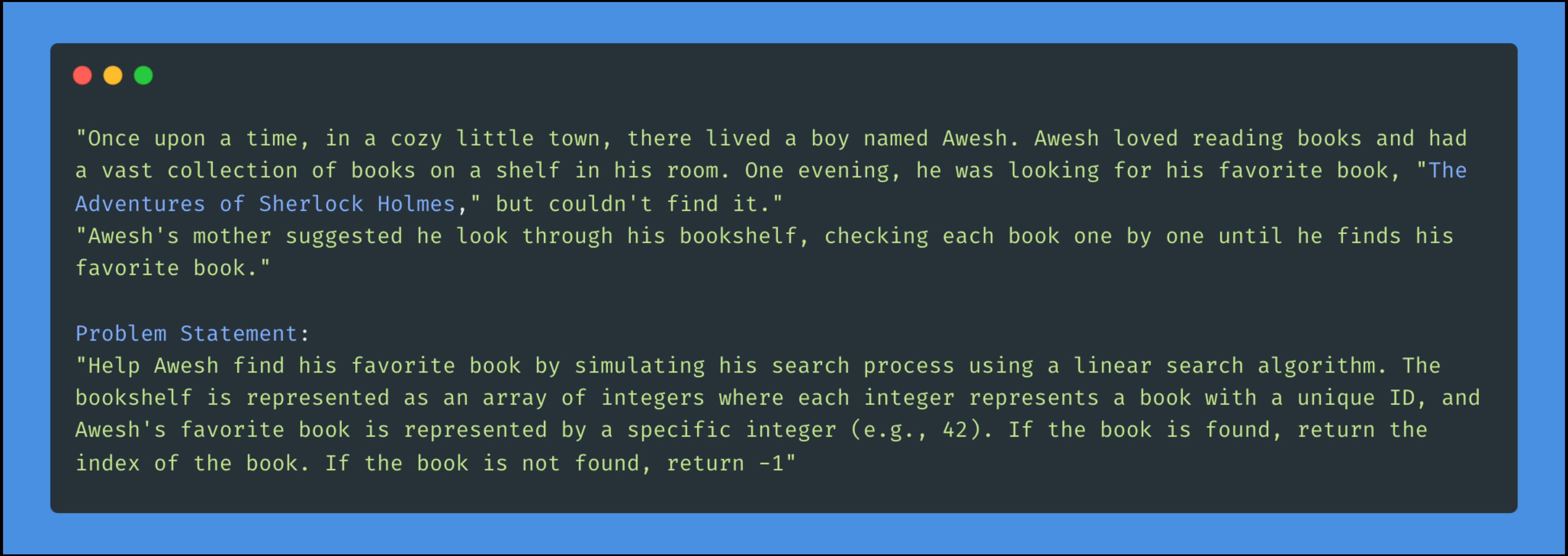
RECURSION 23

SIMPLIFIED CSE COURSE FOR ALL DEPARTMENTS

C & C++

REGISTER NOW

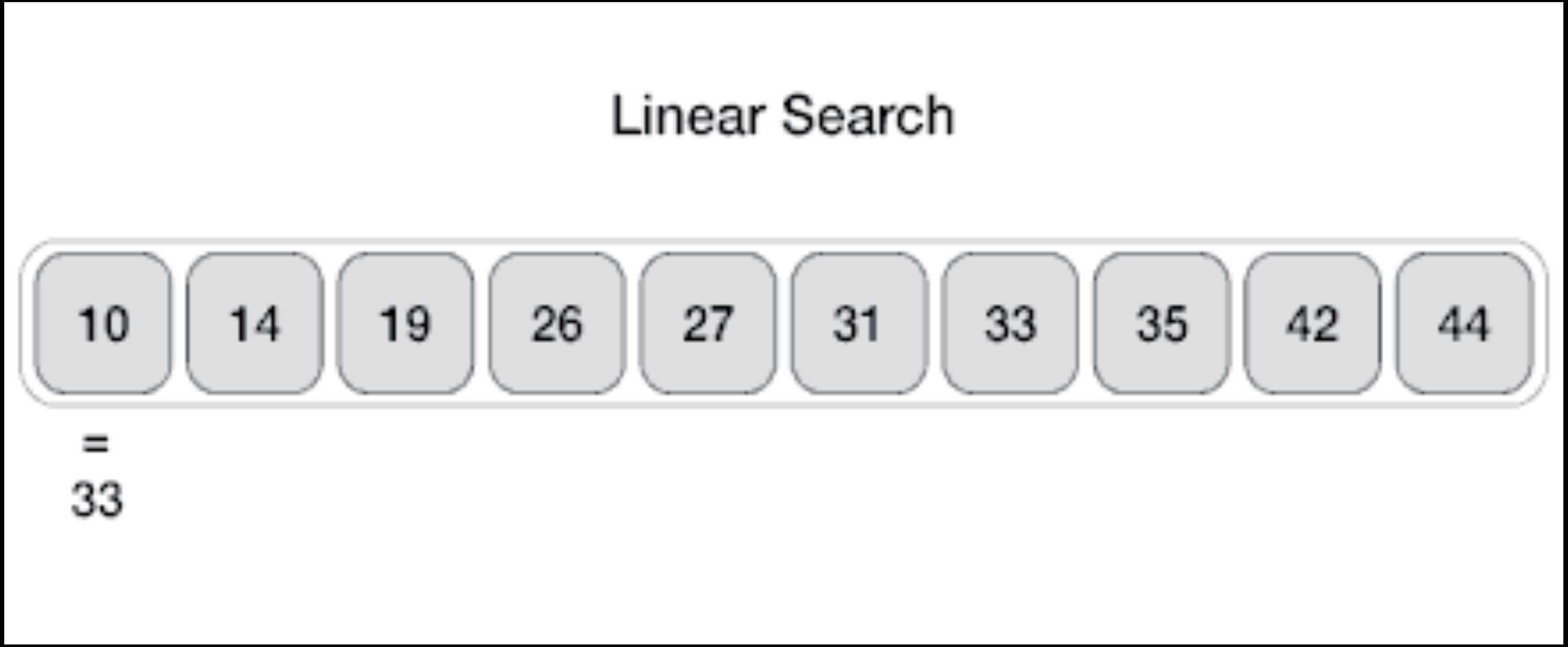
Problem-The Missing Book



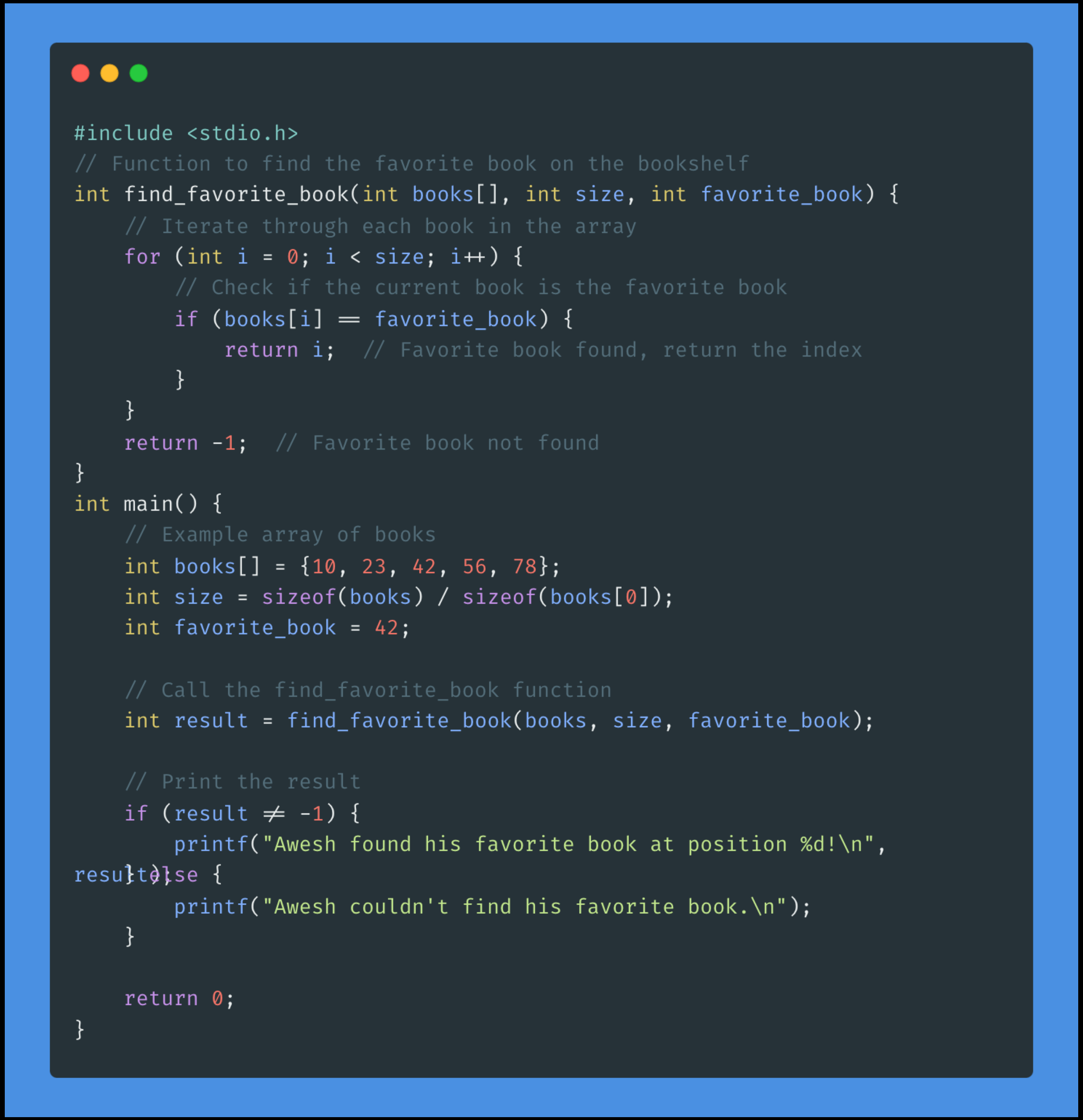
"Once upon a time, in a cozy little town, there lived a boy named Awesh. Awesh loved reading books and had a vast collection of books on a shelf in his room. One evening, he was looking for his favorite book, "The Adventures of Sherlock Holmes," but couldn't find it."
"Awesh's mother suggested he look through his bookshelf, checking each book one by one until he finds his favorite book."

Problem Statement:
"Help Awesh find his favorite book by simulating his search process using a linear search algorithm. The bookshelf is represented as an array of integers where each integer represents a book with a unique ID, and Awesh's favorite book is represented by a specific integer (e.g., 42). If the book is found, return the index of the book. If the book is not found, return -1"

Linear Search

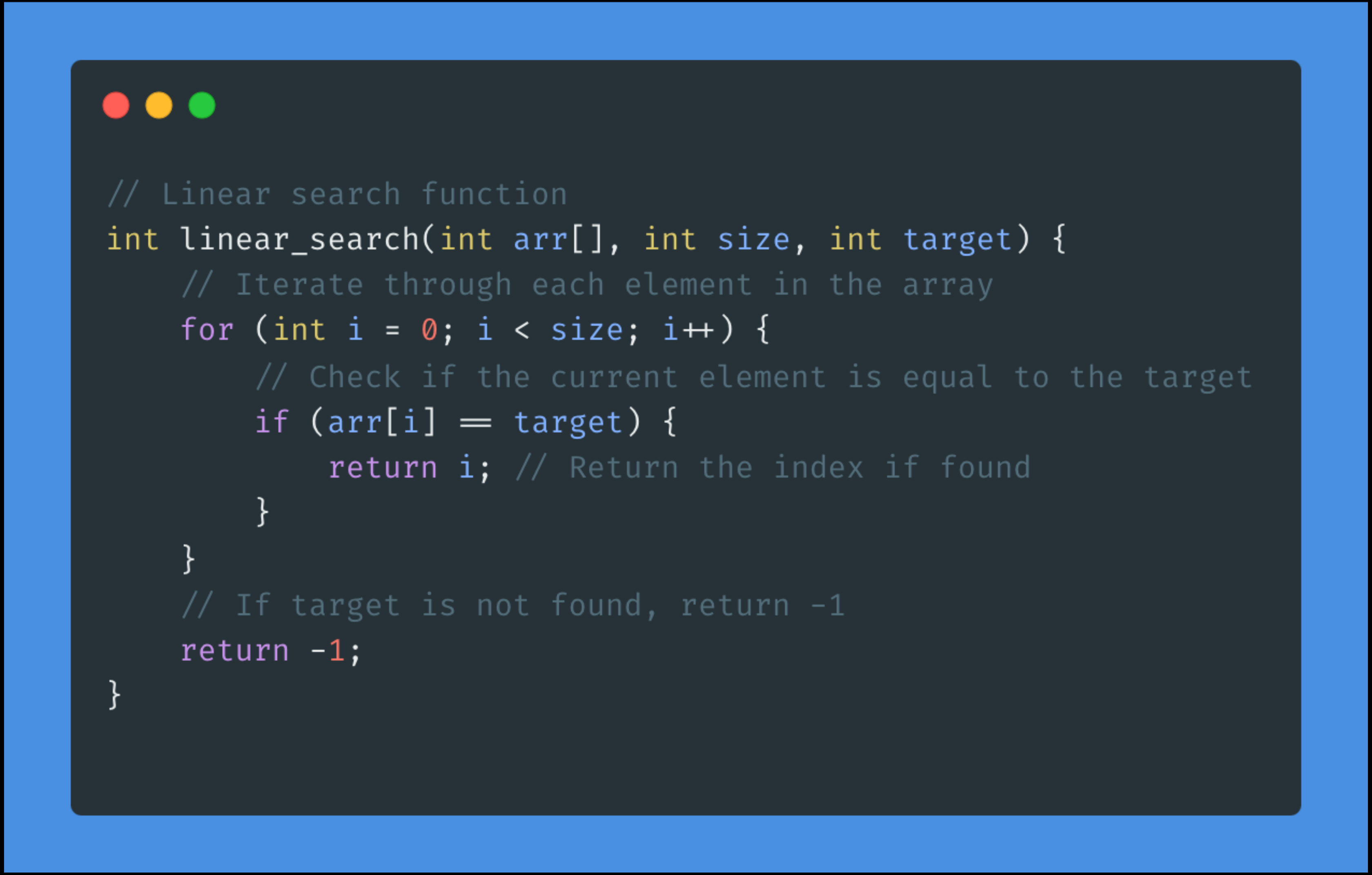


Solve



```c
#include <stdio.h>
// Function to find the favorite book on the bookshelf
int find_favorite_book(int books[], int size, int favorite_book) {
    // Iterate through each book in the array
    for (int i = 0; i < size; i++) {
        // Check if the current book is the favorite book
        if (books[i] == favorite_book) {
            return i;  // Favorite book found, return the index
        }
    }
    return -1;  // Favorite book not found
}
int main() {
    // Example array of books
    int books[] = {10, 23, 42, 56, 78};
    int size = sizeof(books) / sizeof(books[0]);
    int favorite_book = 42;

    // Call the find_favorite_book function
    int result = find_favorite_book(books, size, favorite_book);

    // Print the result
    if (result != -1) {
        printf("Awesh found his favorite book at position %d!\n",
result);
    } else {
        printf("Awesh couldn't find his favorite book.\n");
    }

    return 0;
}
```

Linear Search Code



```c
// Linear search function
int linear_search(int arr[], int size, int target) {
    // Iterate through each element in the array
    for (int i = 0; i < size; i++) {
        // Check if the current element is equal to the target
        if (arr[i] == target) {
            return i; // Return the index if found
        }
    }
    // If target is not found, return -1
    return -1;
}
```

Pros of this algorithm

Simplicity: Linear search is straightforward and easy to understand. It involves sequentially checking each element in the array until the target value is found or the end of the array is reached.

Universal Applicability: Linear search can be applied to any type of array, whether it's sorted or unsorted. This makes it versatile and suitable for a wide range of scenarios.

Ease of Implementation: Linear search doesn't require any pre-processing or special data structures. It can be implemented using basic programming constructs like loops and conditional statements, making it accessible to beginners.

Intuitive: The concept of linear search mimics how we might search for an item in a list manually, making it intuitive for beginners to grasp.

Problems of this algorithm

Inefficiency for Large Datasets: Linear search can be inefficient for large datasets because it checks each element one by one until it finds the target value or reaches the end of the array. This can result in longer search times, especially for arrays with many elements.

Performance Degradation: As the size of the array increases, the time taken to perform linear search also increases linearly. This can lead to performance degradation, particularly for applications that require quick search times.

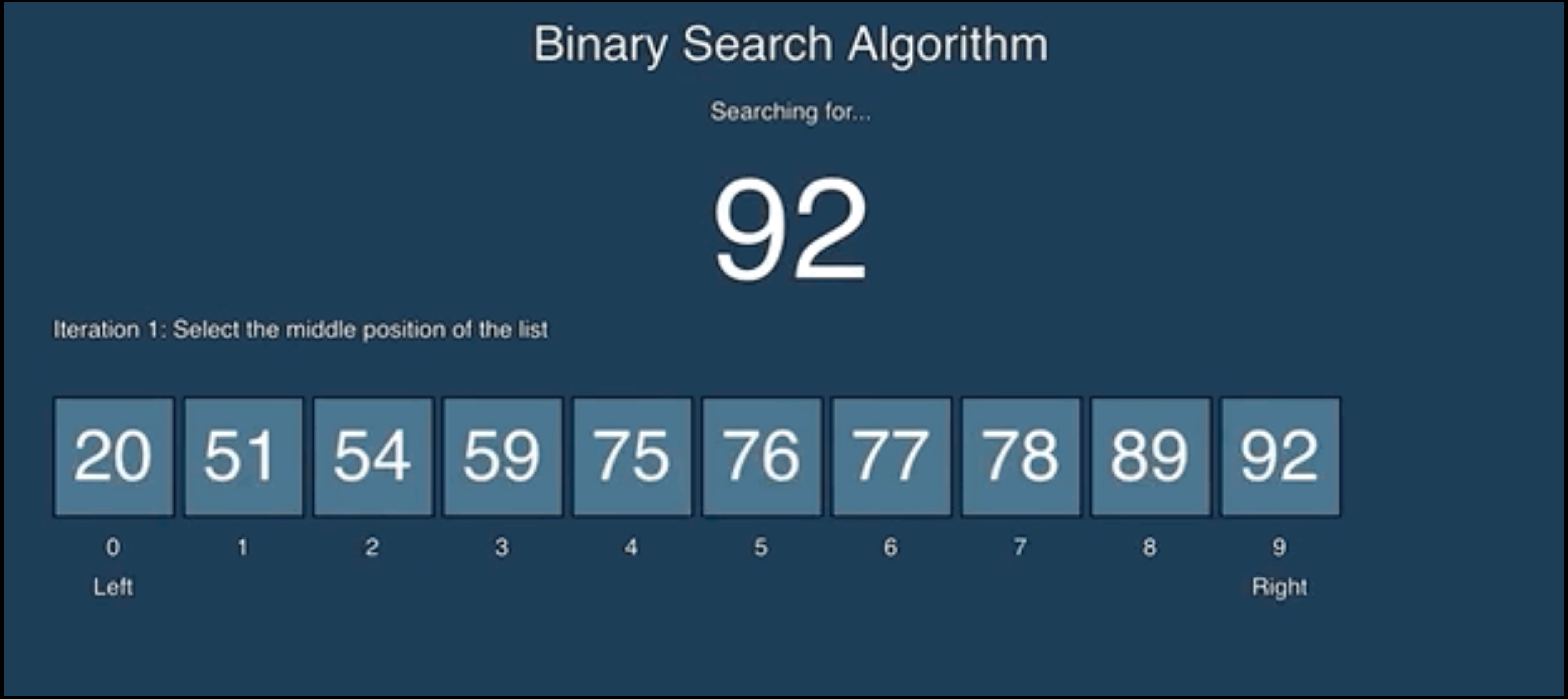
Not Suitable for Sorted Arrays: While linear search works for both sorted and unsorted arrays, it is particularly ineffective for searching in sorted arrays. Other search algorithms like binary search offer better performance for sorted arrays.

Limited Scalability: Linear search may not scale well for applications with large or rapidly growing datasets. In such cases, more efficient search algorithms may be required to maintain acceptable performance levels.

Binary Search

LET’S PLAY A GAME!

Binary Search


Binary Search Algorithm

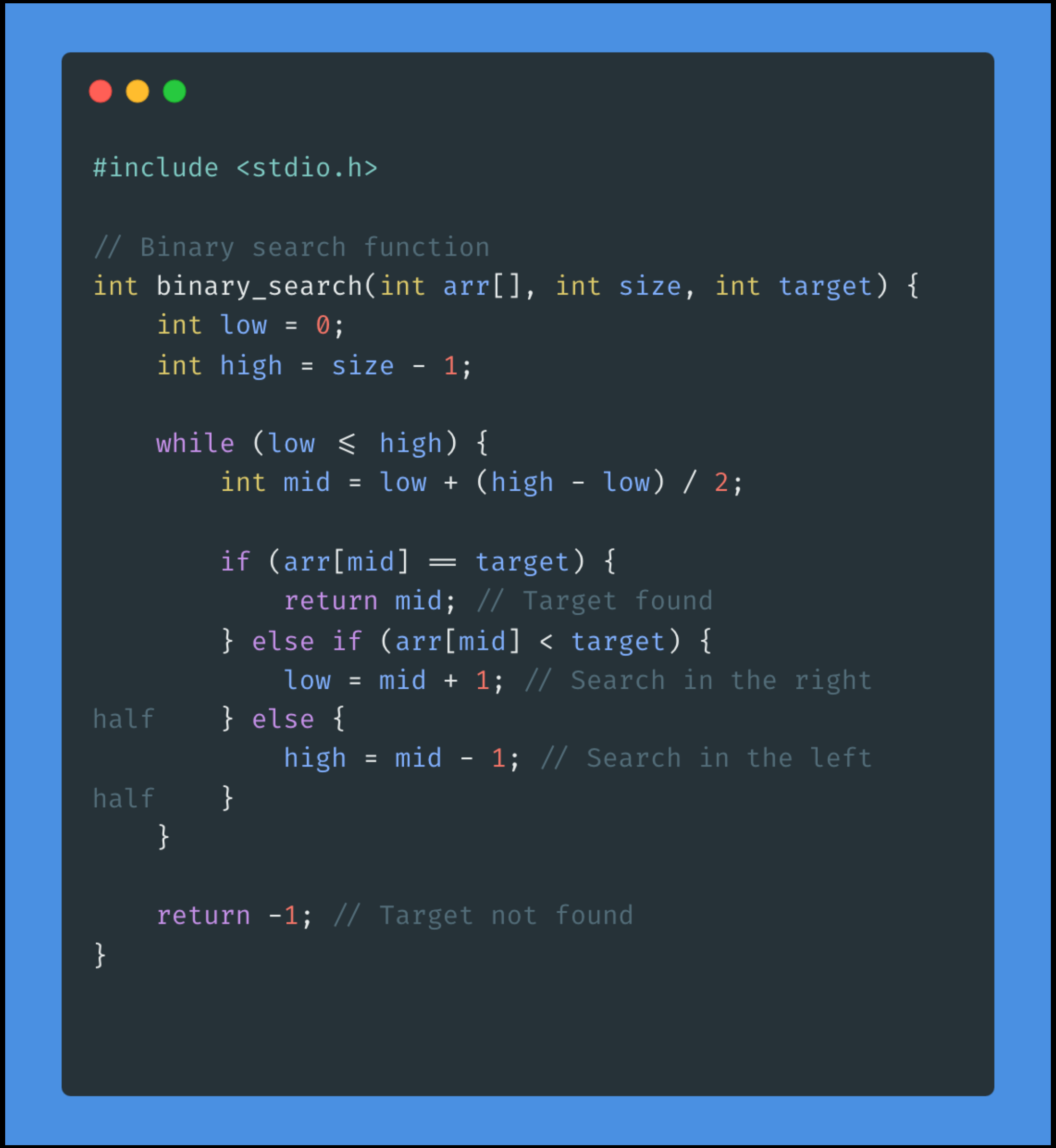
Searching for...

92

Iteration 1: Select the middle position of the list

20	51	54	59	75	76	77	78	89	92
0	1	2	3	4	5	6	7	8	9
Left									Right

Binary Search



```c
#include <stdio.h>

// Binary search function
int binary_search(int arr[], int size, int target) {
    int low = 0;
    int high = size - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == target) {
            return mid; // Target found
        } else if (arr[mid] < target) {
            low = mid + 1; // Search in the right
half    } else {
            high = mid - 1; // Search in the left
half    }
    }

    return -1; // Target not found
}
```

Pros of this algorithm

Efficiency: Binary search has a time complexity of O(log n), making it highly efficient for searching in sorted arrays or lists.

Optimal Performance: It achieves the best possible time complexity for searching in sorted arrays without additional information.

Simple Implementation: Binary search follows a straightforward approach, making it easy to understand and implement.

Space Efficiency: It operates on the original array without requiring additional space for auxiliary data structures, ensuring memory efficiency.

Versatility: Binary search can be applied to various data structures, including arrays, lists, and trees, as long as the elements are sorted, enhancing its usefulness in a wide range of applications.

Problems of this algorithm

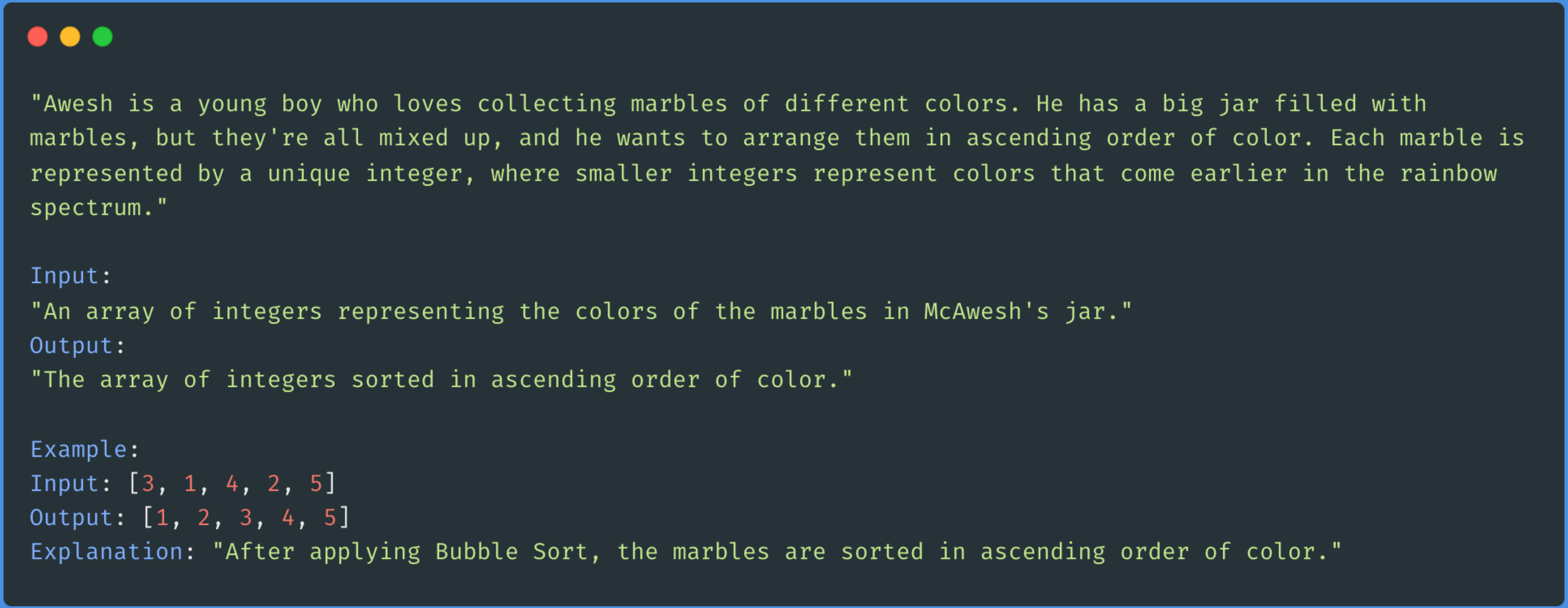
Requirement of Sorted Data: Binary search requires the array to be sorted in ascending or descending order. If the array is unsorted or if sorting is not feasible, binary search cannot be directly applied, and preprocessing steps are required.

Limited Applicability: Binary search is not suitable for all types of data structures. It works best for arrays and lists where elements are accessible via indices. It may not be suitable for complex data structures like trees or graphs.

Memory Overhead: Binary search typically requires more memory overhead than linear search, as it involves additional variables to keep track of the search space and indices. However, the memory overhead is generally minimal and not a significant concern in most cases.

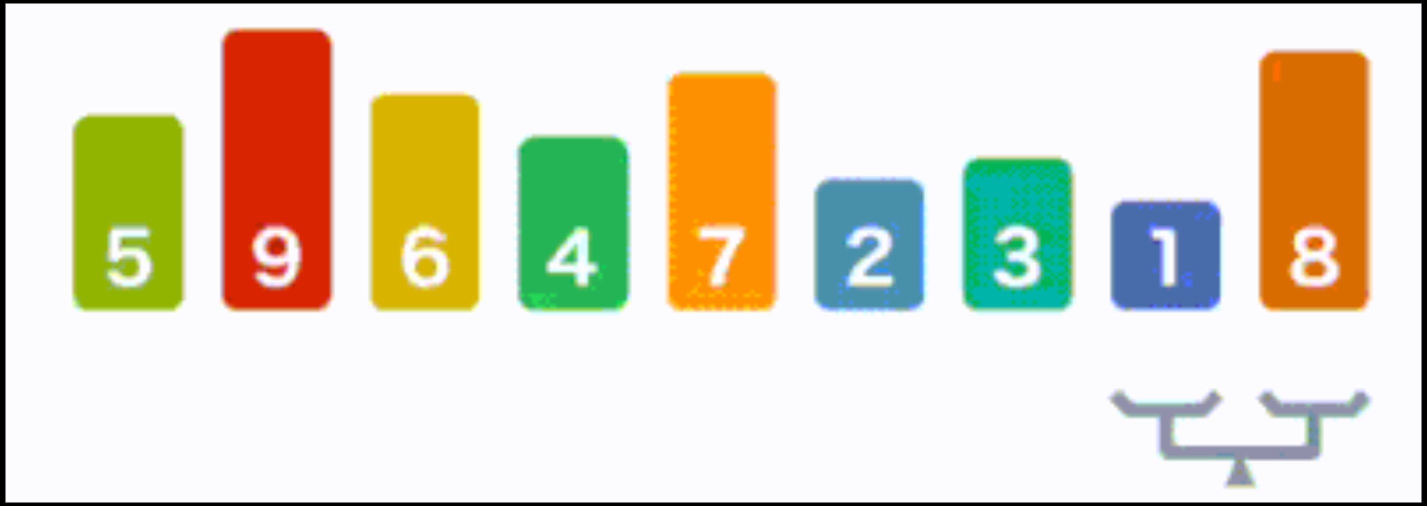
Lack of Flexibility: While binary search is highly efficient for searching in sorted arrays, it may not be the best choice for dynamic or frequently changing datasets. Maintaining the sorted order of the array can be challenging in such scenarios.

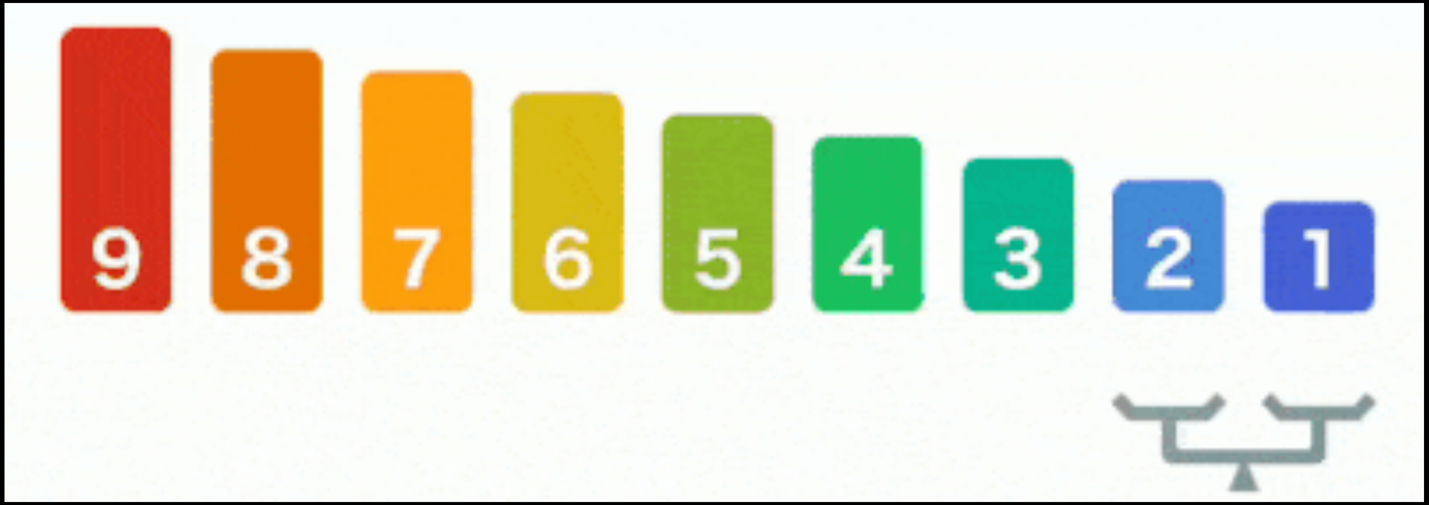
Problem: Sorting Marbles



```
"Awesh is a young boy who loves collecting marbles of different colors. He has a big jar filled with
marbles, but they're all mixed up, and he wants to arrange them in ascending order of color. Each marble is
represented by a unique integer, where smaller integers represent colors that come earlier in the rainbow
spectrum."

Input:
"An array of integers representing the colors of the marbles in McAwesh's jar."
Output:
"The array of integers sorted in ascending order of color."

Example:
Input: [3, 1, 4, 2, 5]
Output: [1, 2, 3, 4, 5]
Explanation: "After applying Bubble Sort, the marbles are sorted in ascending order of color."
```

Bubble Sort





Pros and Cons

Pros:

Simple Implementation: Bubble Sort is one of the simplest sorting algorithms to implement, making it easy to understand for beginners.

No Extra Space: Bubble Sort sorts the elements in place without requiring any additional space, which can be advantageous for memory-constrained environments.

Stable Sorting: Bubble Sort is stable, meaning that equal elements maintain their relative order after sorting.

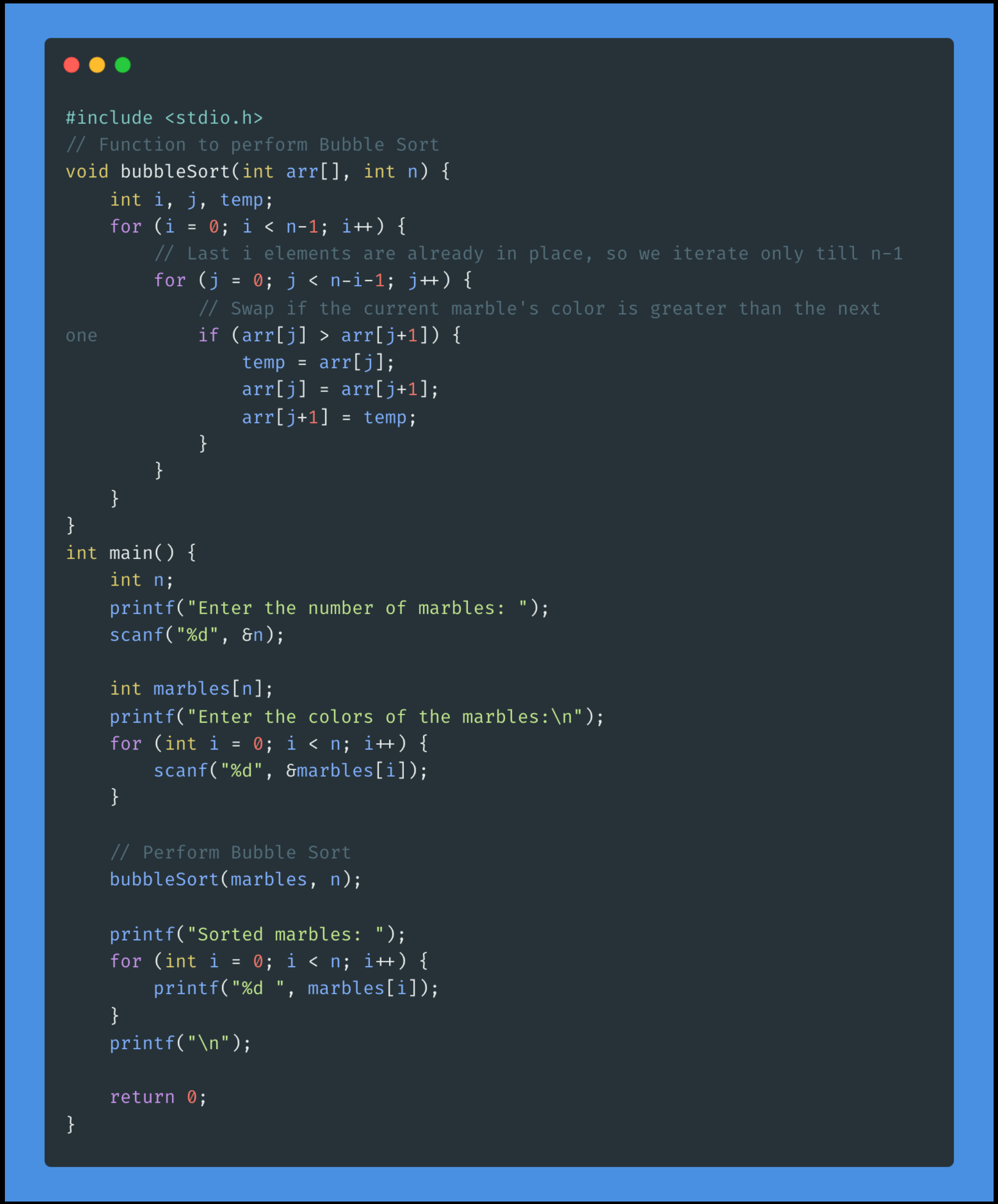
Cons:

Inefficiency: Bubble Sort is generally inefficient for large lists or arrays. Its time complexity is O(n^2), which means it becomes slow as the number of elements increases.

Poor Performance: It has poor performance even for small lists compared to more efficient sorting algorithms like Merge Sort and Quick Sort.

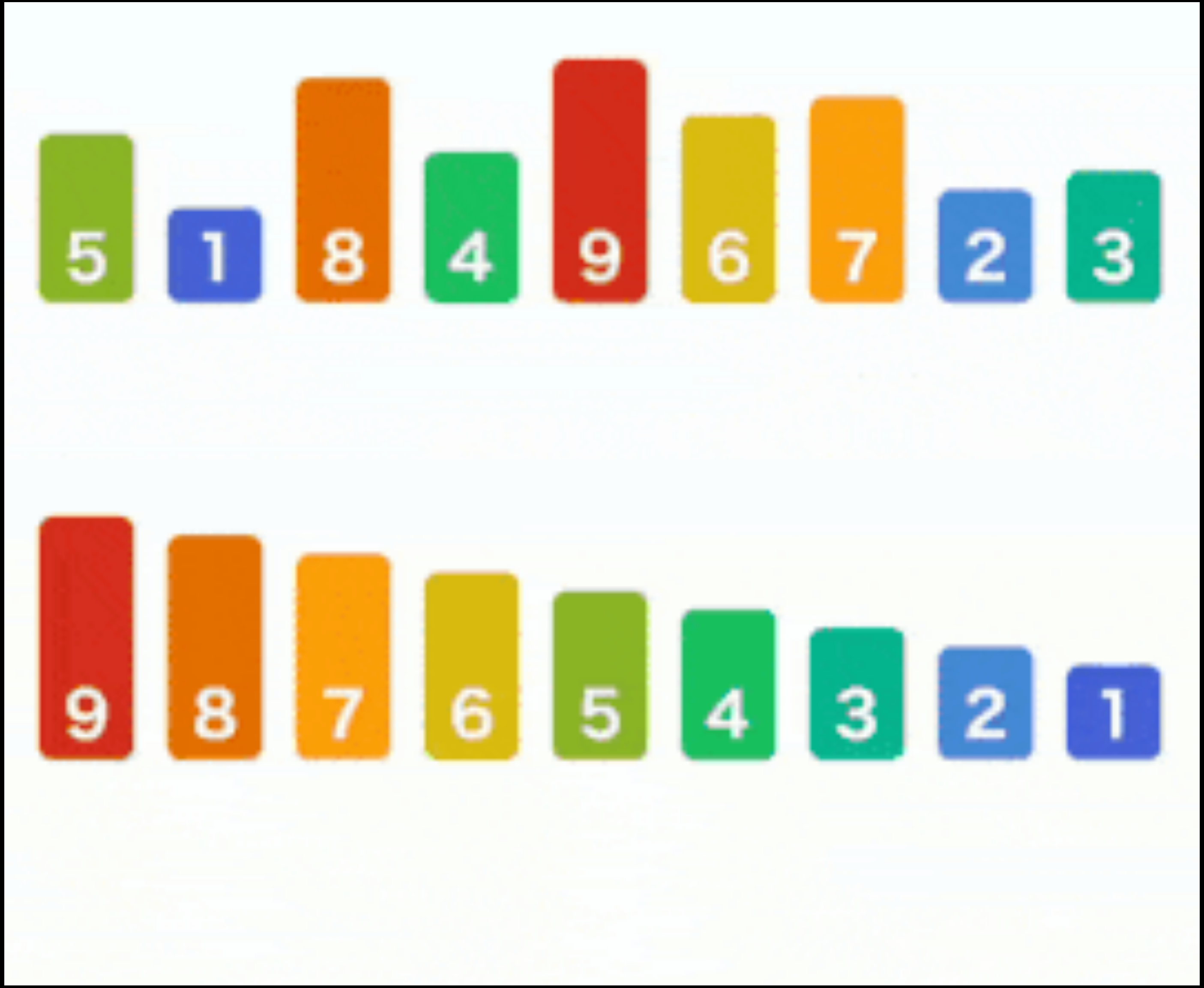
Not Adaptive: Bubble Sort doesn't adapt to the input data; it always performs the same number of comparisons and swaps regardless of the initial order of elements.

Solve

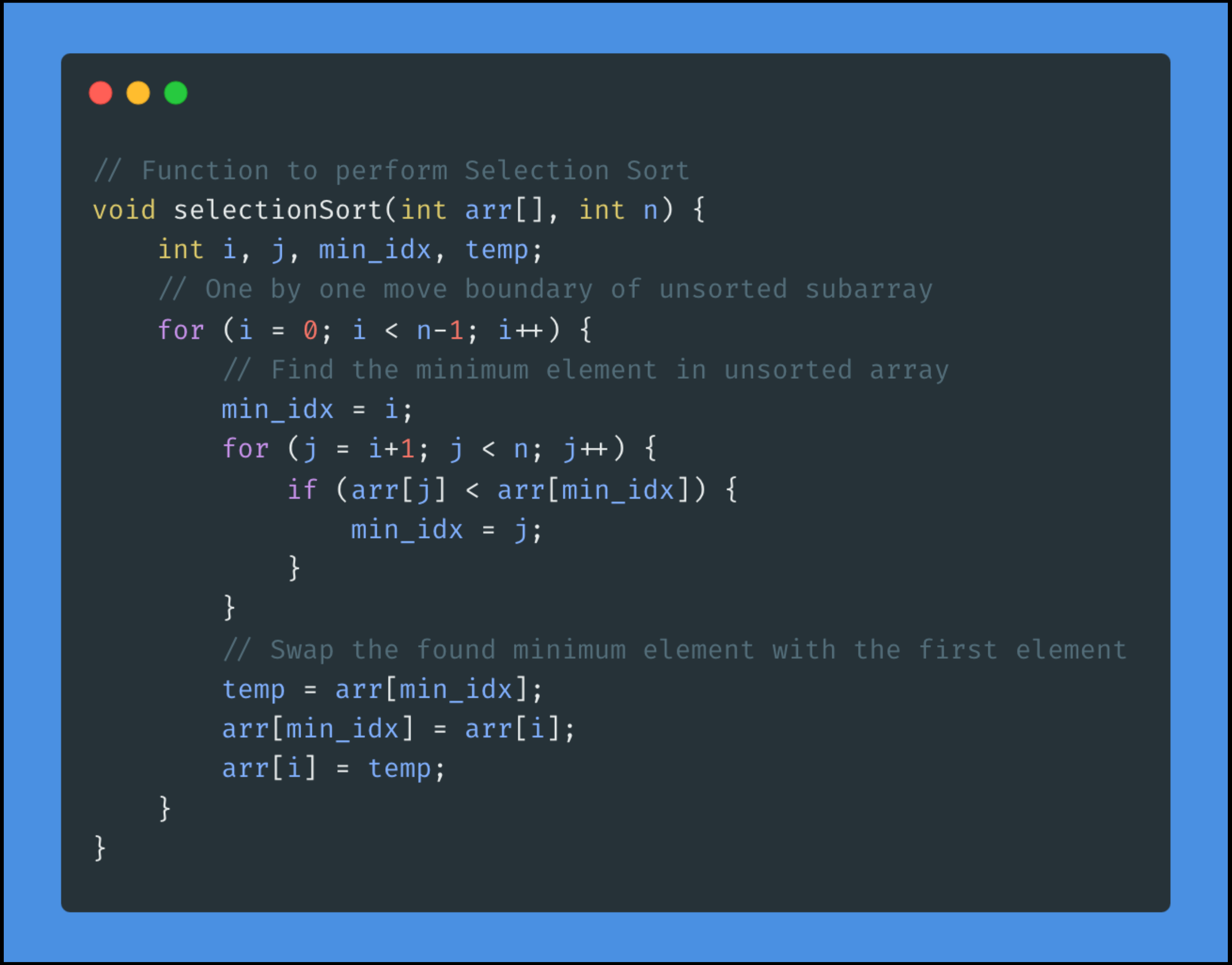


```c
#include <stdio.h>
// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n-1; i++) {
        // Last i elements are already in place, so we iterate only till n-1
        for (j = 0; j < n-i-1; j++) {
            // Swap if the current marble's color is greater than the next
one         if (arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
int main() {
    int n;
    printf("Enter the number of marbles: ");
    scanf("%d", &n);

    int marbles[n];
    printf("Enter the colors of the marbles:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &marbles[i]);
    }

    // Perform Bubble Sort
    bubbleSort(marbles, n);

    printf("Sorted marbles: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", marbles[i]);
    }
    printf("\n");

    return 0;
}
```

Selection Sort



Solve



```c
// Function to perform Selection Sort
void selectionSort(int arr[], int n) {
    int i, j, min_idx, temp;
    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++) {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        // Swap the found minimum element with the first element
        temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Pros and Cons

Pros:

Simple Implementation: Selection Sort is also relatively simple to implement, making it suitable for educational purposes or situations where simplicity is preferred over performance.

In-Place Sorting: Similar to Bubble Sort, Selection Sort sorts the elements in place without requiring additional space for temporary storage.

Good for Small Lists: Selection Sort performs reasonably well for small lists or arrays, especially when compared to inefficient algorithms like Bubble Sort.

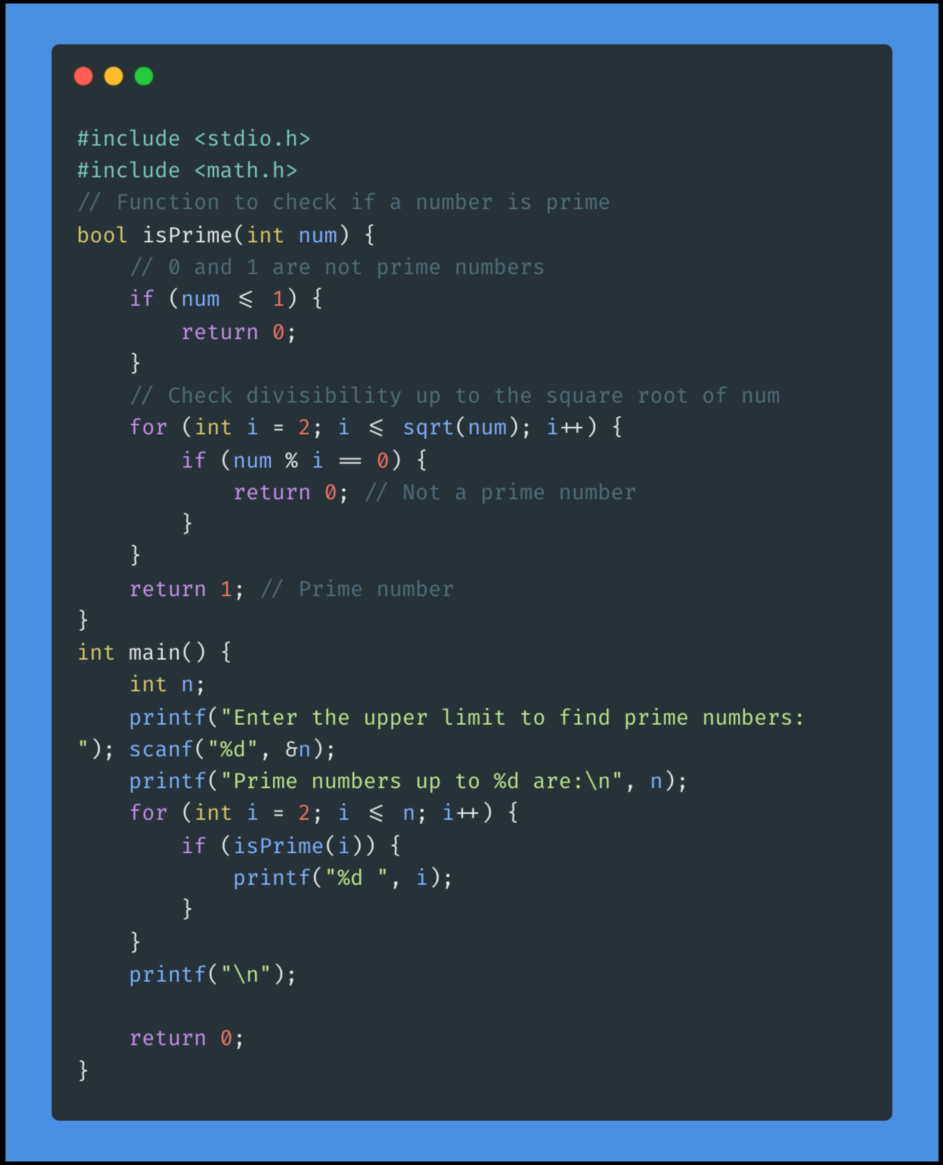
Cons:

Inefficiency: Like Bubble Sort, Selection Sort has a time complexity of O(n^2), which makes it inefficient for large lists.

Not Adaptive: Selection Sort doesn't adapt to the input data; it always performs the same number of comparisons and swaps regardless of the initial order of elements.

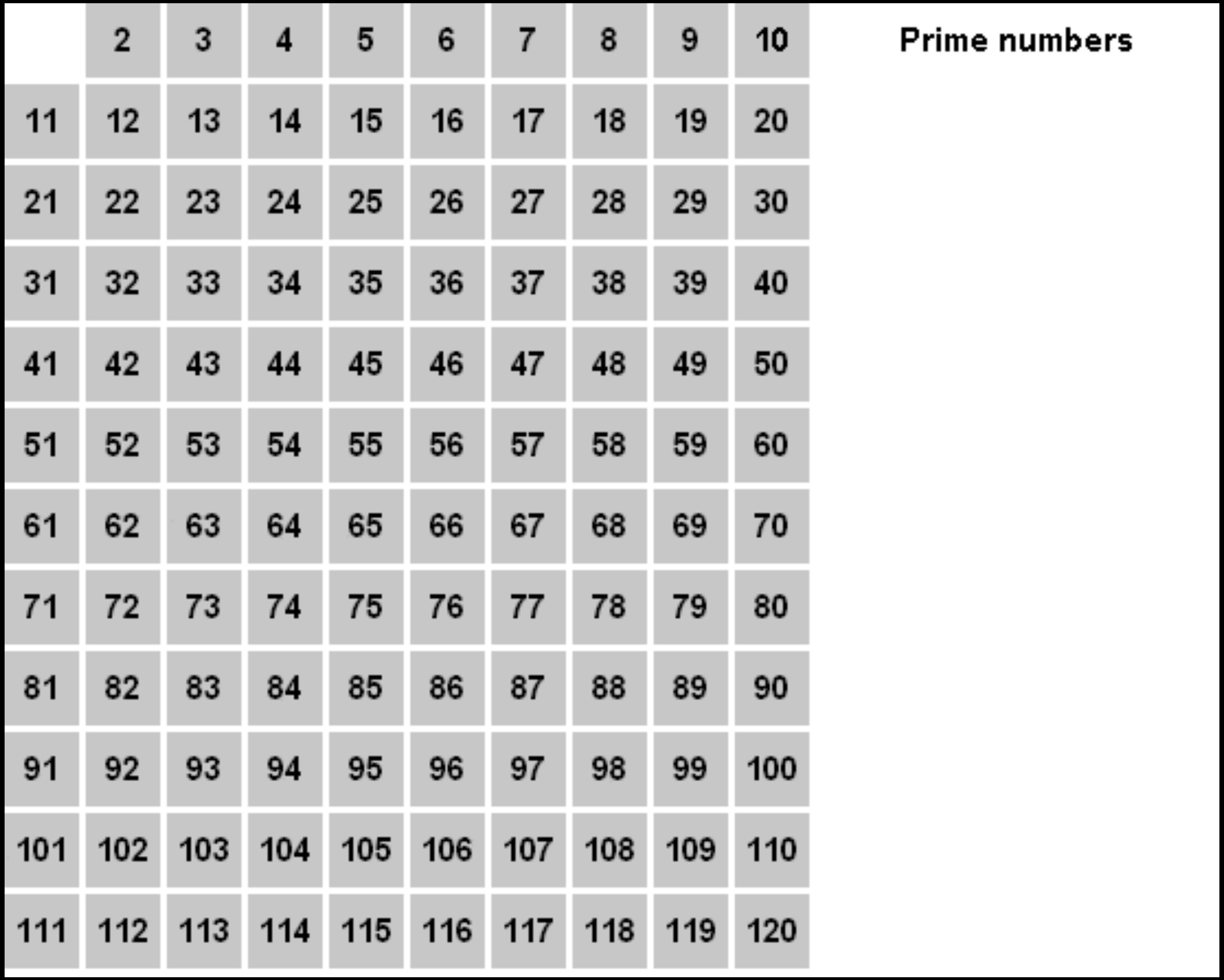
Not Stable: Selection Sort is not stable; it may change the relative order of equal elements during sorting.

Let’s find prime numbers



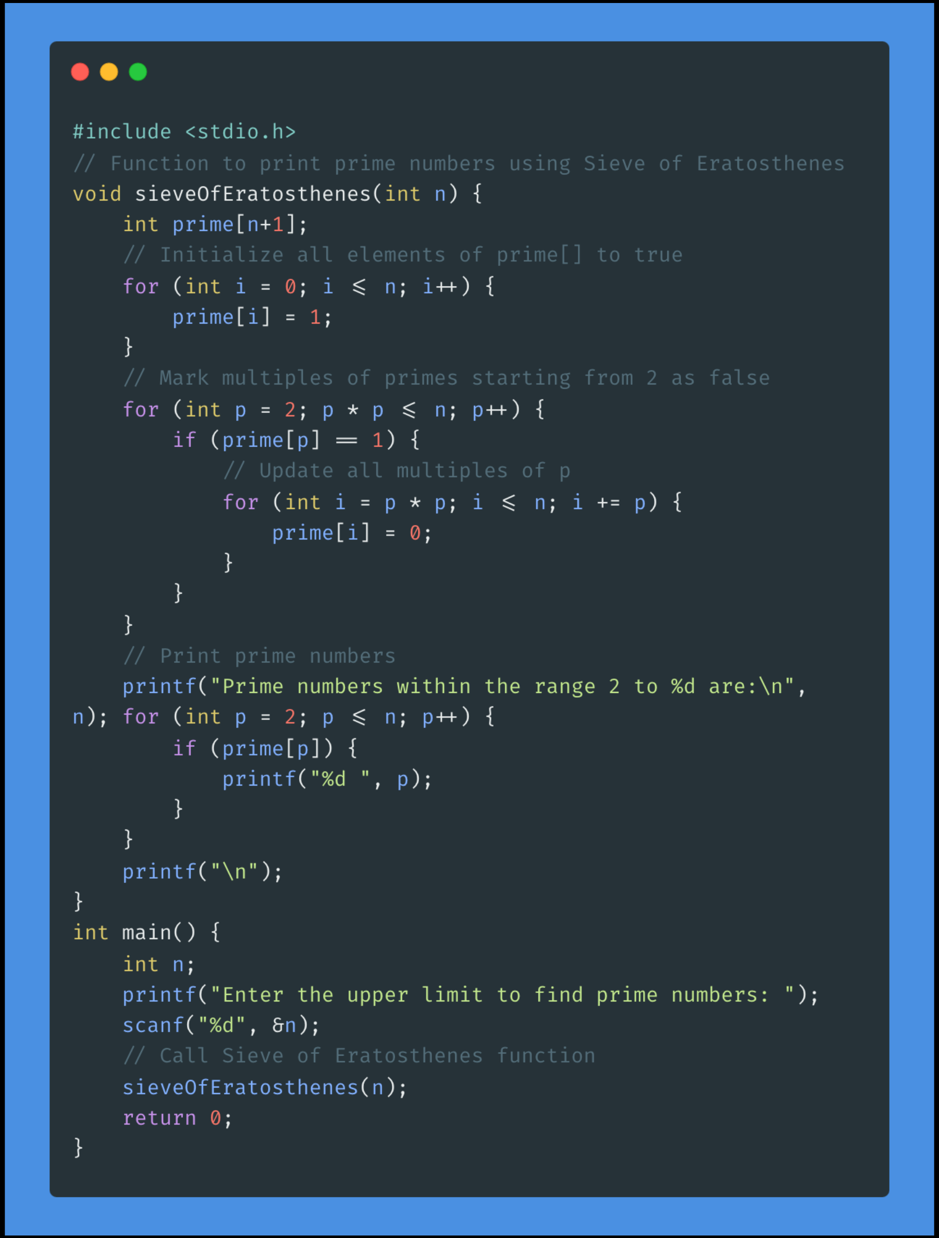
```c
#include <stdio.h>
#include <math.h>
// Function to check if a number is prime
bool isPrime(int num) {
    // 0 and 1 are not prime numbers
    if (num <= 1) {
        return 0;
    }
    // Check divisibility up to the square root of num
    for (int i = 2; i <= sqrt(num); i++) {
        if (num % i == 0) {
            return 0; // Not a prime number
        }
    }
    return 1; // Prime number
}
int main() {
    int n;
    printf("Enter the upper limit to find prime numbers:
"); scanf("%d", &n);
    printf("Prime numbers up to %d are:\n", n);
    for (int i = 2; i <= n; i++) {
        if (isPrime(i)) {
            printf("%d ", i);
        }
    }
    printf("\n");

    return 0;
}
```

Sieve of Eratosthenes



	2	3	4	5	6	7	8	9	10	Prime numbers
11	12	13	14	15	16	17	18	19	20	
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	
41	42	43	44	45	46	47	48	49	50	
51	52	53	54	55	56	57	58	59	60	
61	62	63	64	65	66	67	68	69	70	
71	72	73	74	75	76	77	78	79	80	
81	82	83	84	85	86	87	88	89	90	
91	92	93	94	95	96	97	98	99	100	
101	102	103	104	105	106	107	108	109	110	
111	112	113	114	115	116	117	118	119	120	

Code

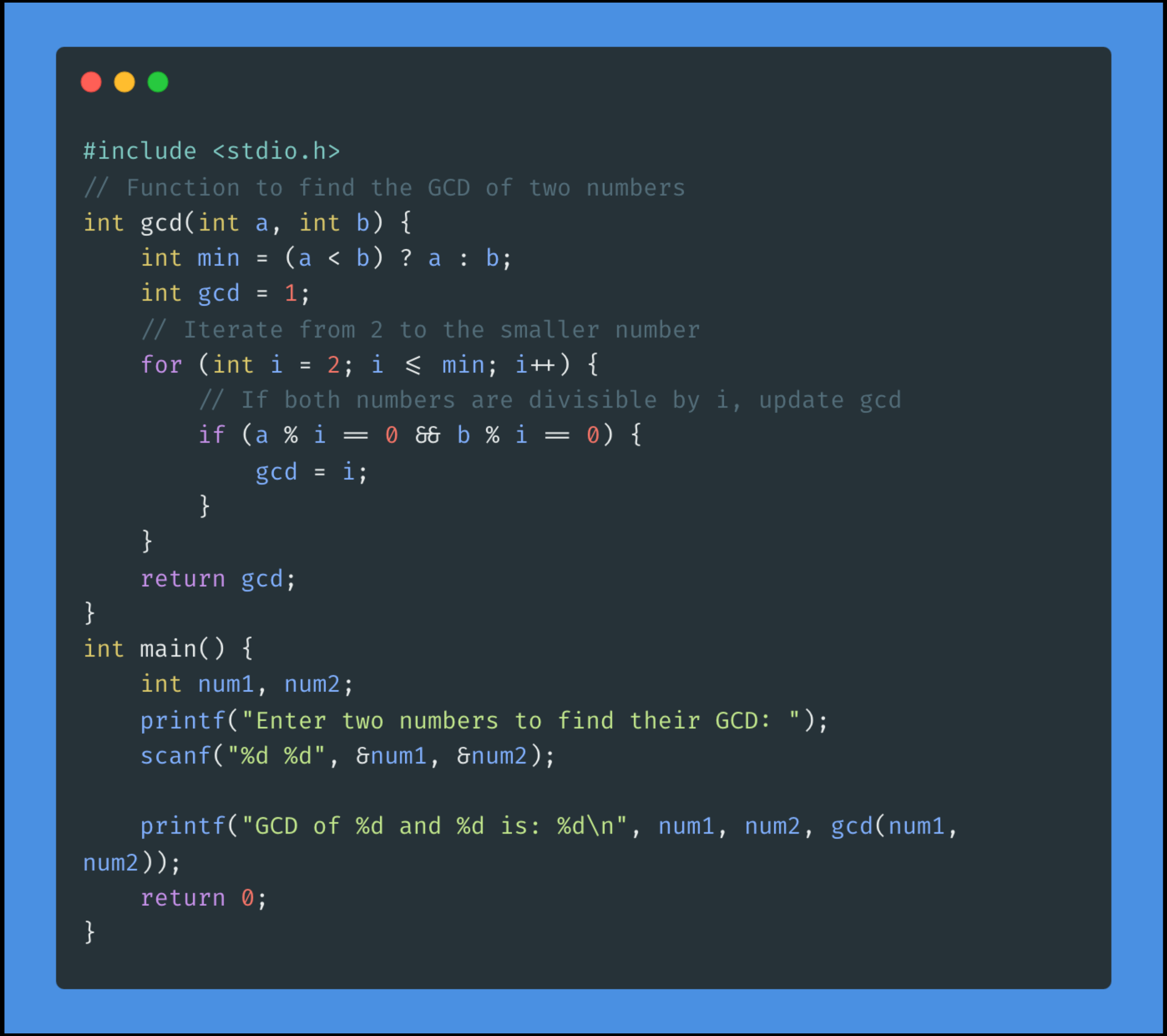


```c
#include <stdio.h>
// Function to print prime numbers using Sieve of Eratosthenes
void sieveOfEratosthenes(int n) {
    int prime[n+1];
    // Initialize all elements of prime[] to true
    for (int i = 0; i <= n; i++) {
        prime[i] = 1;
    }
    // Mark multiples of primes starting from 2 as false
    for (int p = 2; p * p <= n; p++) {
        if (prime[p] == 1) {
            // Update all multiples of p
            for (int i = p * p; i <= n; i += p) {
                prime[i] = 0;
            }
        }
    }
    // Print prime numbers
    printf("Prime numbers within the range 2 to %d are:\n",
n); for (int p = 2; p <= n; p++) {
        if (prime[p]) {
            printf("%d ", p);
        }
    }
    printf("\n");
}
int main() {
    int n;
    printf("Enter the upper limit to find prime numbers: ");
    scanf("%d", &n);
    // Call Sieve of Eratosthenes function
    sieveOfEratosthenes(n);
    return 0;
}
```

Why is Eratosthenes better?

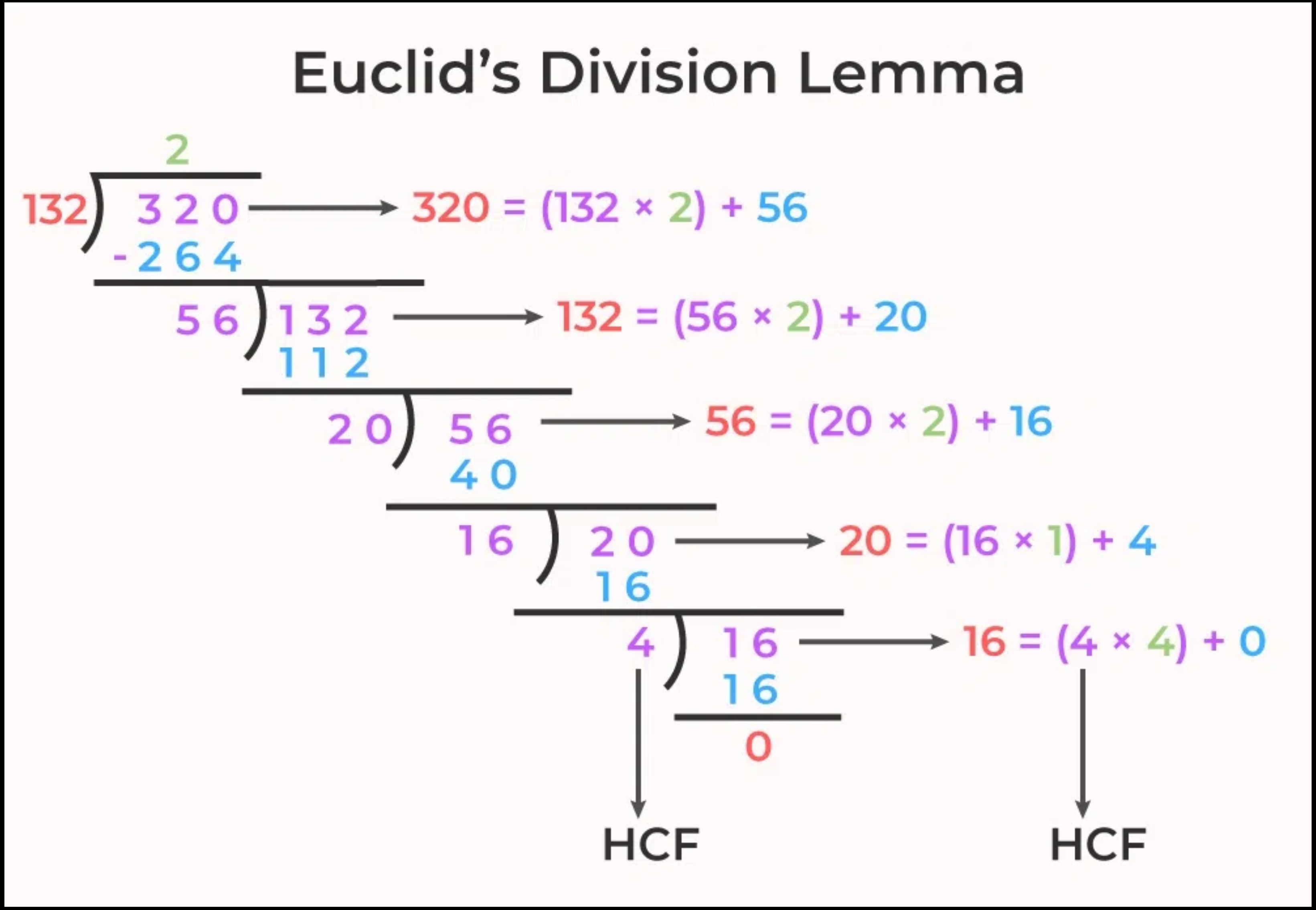
1. **Efficiency**: The Sieve of Eratosthenes is significantly more efficient than the basic method of checking divisibility, especially for large ranges of numbers. Its time complexity is O(n log log n), where n is the upper limit of the range. In contrast, the basic method has a time complexity of O(n*sqrt(n)), which becomes slower as the range increases.
2. **Optimality**: The Sieve of Eratosthenes is an optimal algorithm for finding prime numbers within a given range. It efficiently eliminates multiples of prime numbers, ensuring that each composite number is only marked once. This results in an optimal number of iterations required to sieve out composite numbers.
3. **Space Complexity**: While the Sieve of Eratosthenes requires additional memory to store the sieve array, the space complexity is still considered reasonable. It only requires O(n) additional space, where n is the upper limit of the range. In contrast, the basic method of checking divisibility does not require additional space beyond the input numbers.
4. **Ease of Implementation**: The Sieve of Eratosthenes is relatively easy to implement and understand. It follows a simple, step-by-step procedure of marking multiples of primes, making it accessible even to beginner programmers. In contrast, implementing the basic method of checking divisibility for large ranges can be more complex and error-prone.
5. **Versatility**: The Sieve of Eratosthenes can be adapted and extended for various related problems, such as finding prime factors, prime factorization, and checking primality efficiently for multiple numbers within a given range.

Let’s find gcd



```c
#include <stdio.h>
// Function to find the GCD of two numbers
int gcd(int a, int b) {
    int min = (a < b) ? a : b;
    int gcd = 1;
    // Iterate from 2 to the smaller number
    for (int i = 2; i <= min; i++) {
        // If both numbers are divisible by i, update gcd
        if (a % i == 0 && b % i == 0) {
            gcd = i;
        }
    }
    return gcd;
}
int main() {
    int num1, num2;
    printf("Enter two numbers to find their GCD: ");
    scanf("%d %d", &num1, &num2);

    printf("GCD of %d and %d is: %d\n", num1, num2, gcd(num1,
num2));
    return 0;
}
```

Euclid’s Algorithm



$320 = (132 \times 2) + 56$

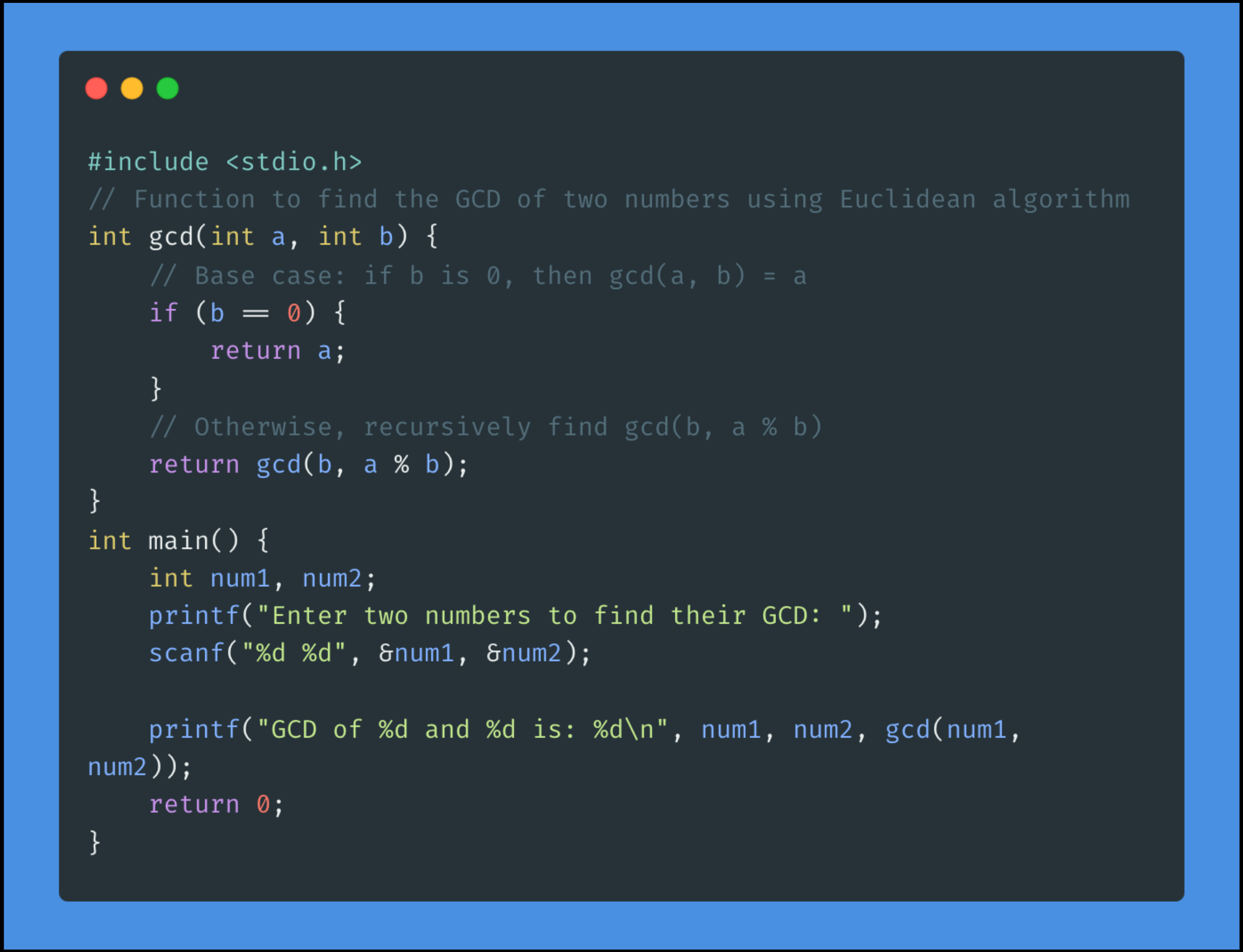
$132 = (56 \times 2) + 20$

$56 = (20 \times 2) + 16$

$20 = (16 \times 1) + 4$

$16 = (4 \times 4) + 0$

Code

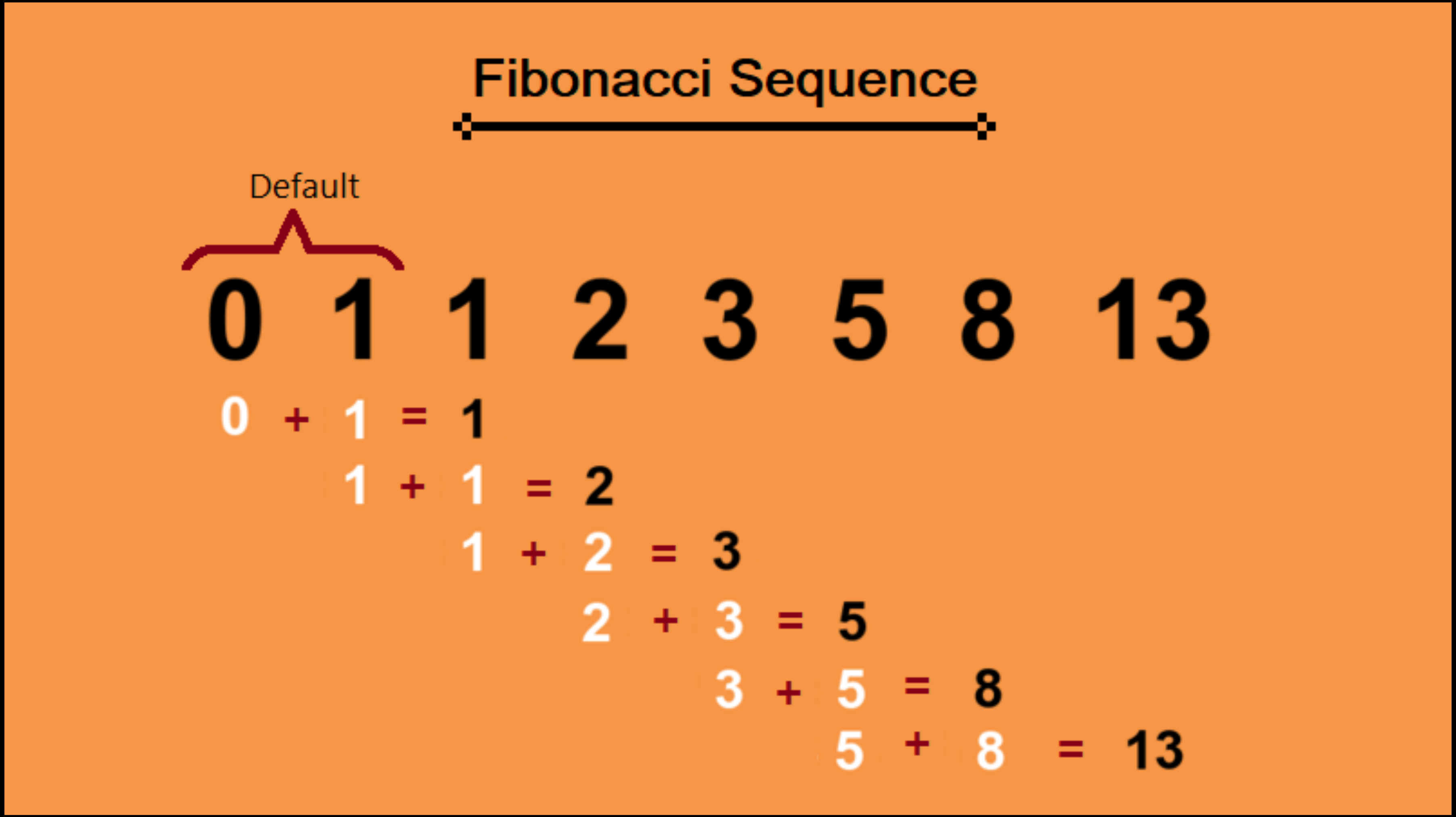


```c
#include <stdio.h>
// Function to find the GCD of two numbers using Euclidean algorithm
int gcd(int a, int b) {
    // Base case: if b is 0, then gcd(a, b) = a
    if (b == 0) {
        return a;
    }
    // Otherwise, recursively find gcd(b, a % b)
    return gcd(b, a % b);
}
int main() {
    int num1, num2;
    printf("Enter two numbers to find their GCD: ");
    scanf("%d %d", &num1, &num2);

    printf("GCD of %d and %d is: %d\n", num1, num2, gcd(num1,
num2));
    return 0;
}
```

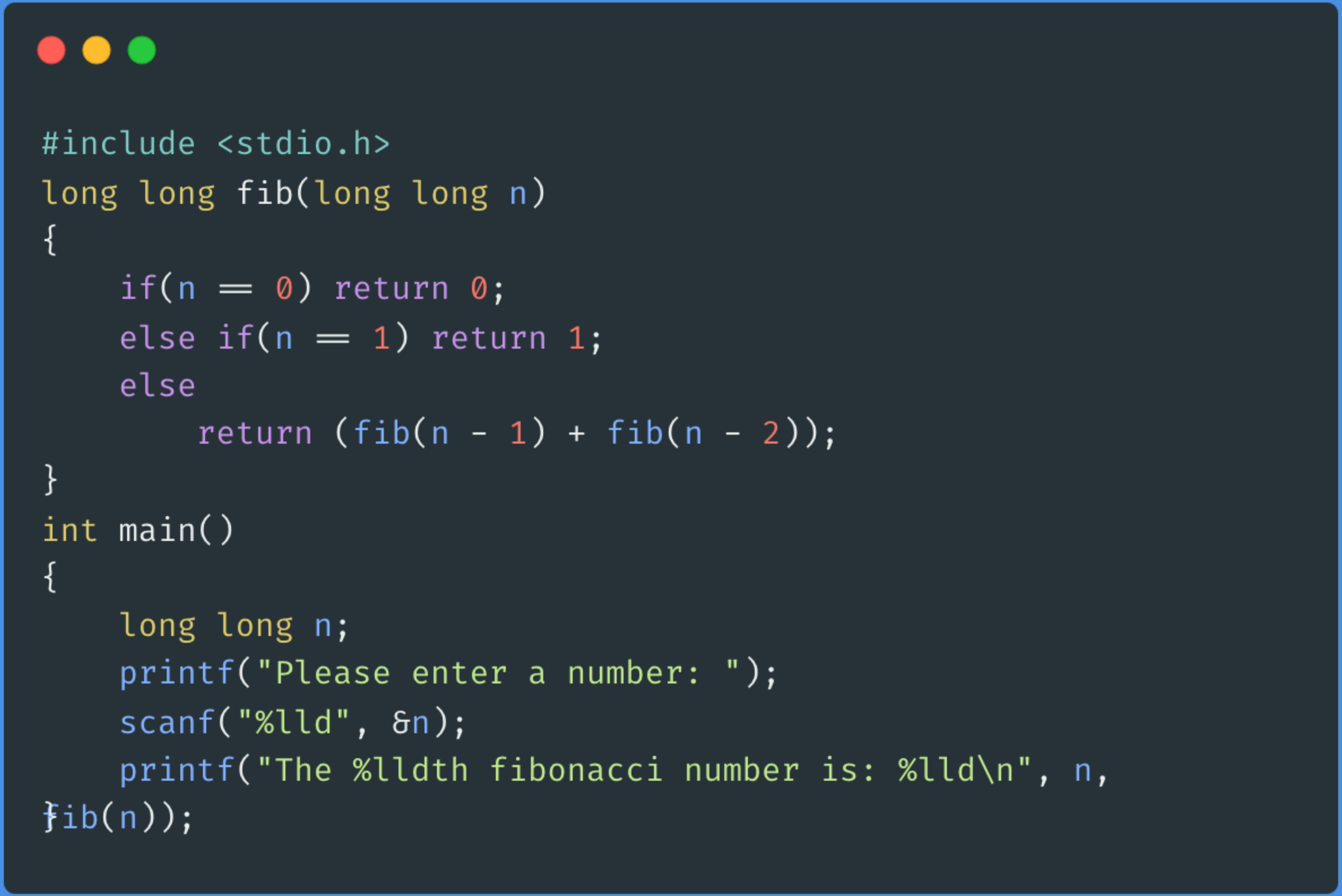
Why is Euclid better?

1. **Efficiency**: The Euclidean algorithm is more efficient than the basic iteration approach, especially for large numbers. Its time complexity is O(log(min(a, b))), where a and b are the two input numbers. This makes it significantly faster than the basic iteration approach, which has a time complexity of O(min(a, b)).
2. **Optimality**: The Euclidean algorithm is optimal in terms of the number of steps required to find the GCD. It achieves this optimality by repeatedly reducing the problem to a smaller instance until a base case is reached. This contrasts with the basic iteration approach, which checks divisibility for all numbers up to the smaller of the two input numbers.
3. **Simplicity**: The Euclidean algorithm is conceptually simpler and more elegant than the basic iteration approach. It leverages the fundamental properties of divisibility and remains straightforward to understand even as the input numbers increase in magnitude.
4. **Versatility**: The Euclidean algorithm extends naturally to other mathematical problems, such as finding the least common multiple (LCM) or solving linear Diophantine equations. Its recursive nature lends itself well to solving a variety of related problems efficiently.

Fibonacci Number

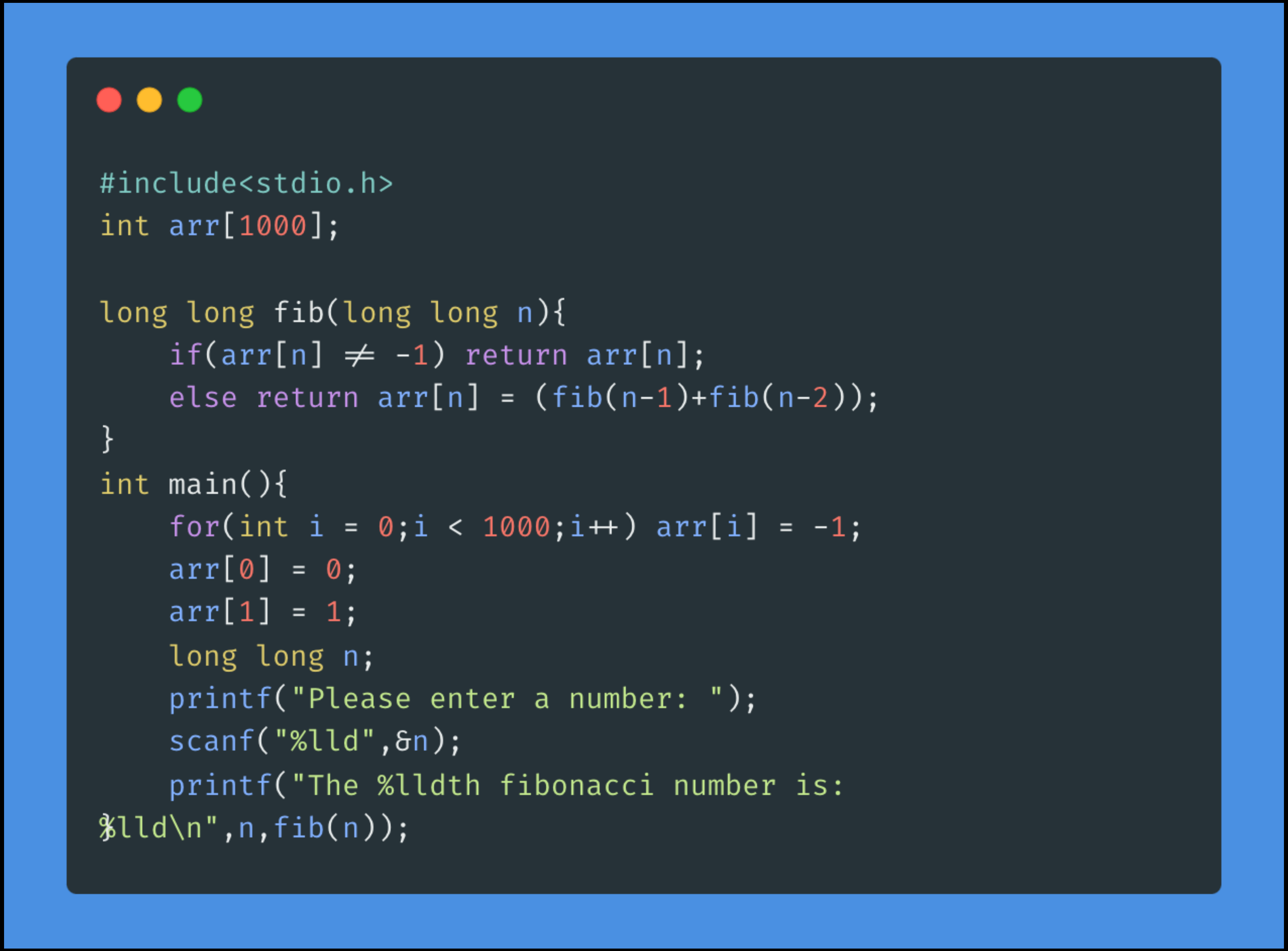


Fibonacci Number Code



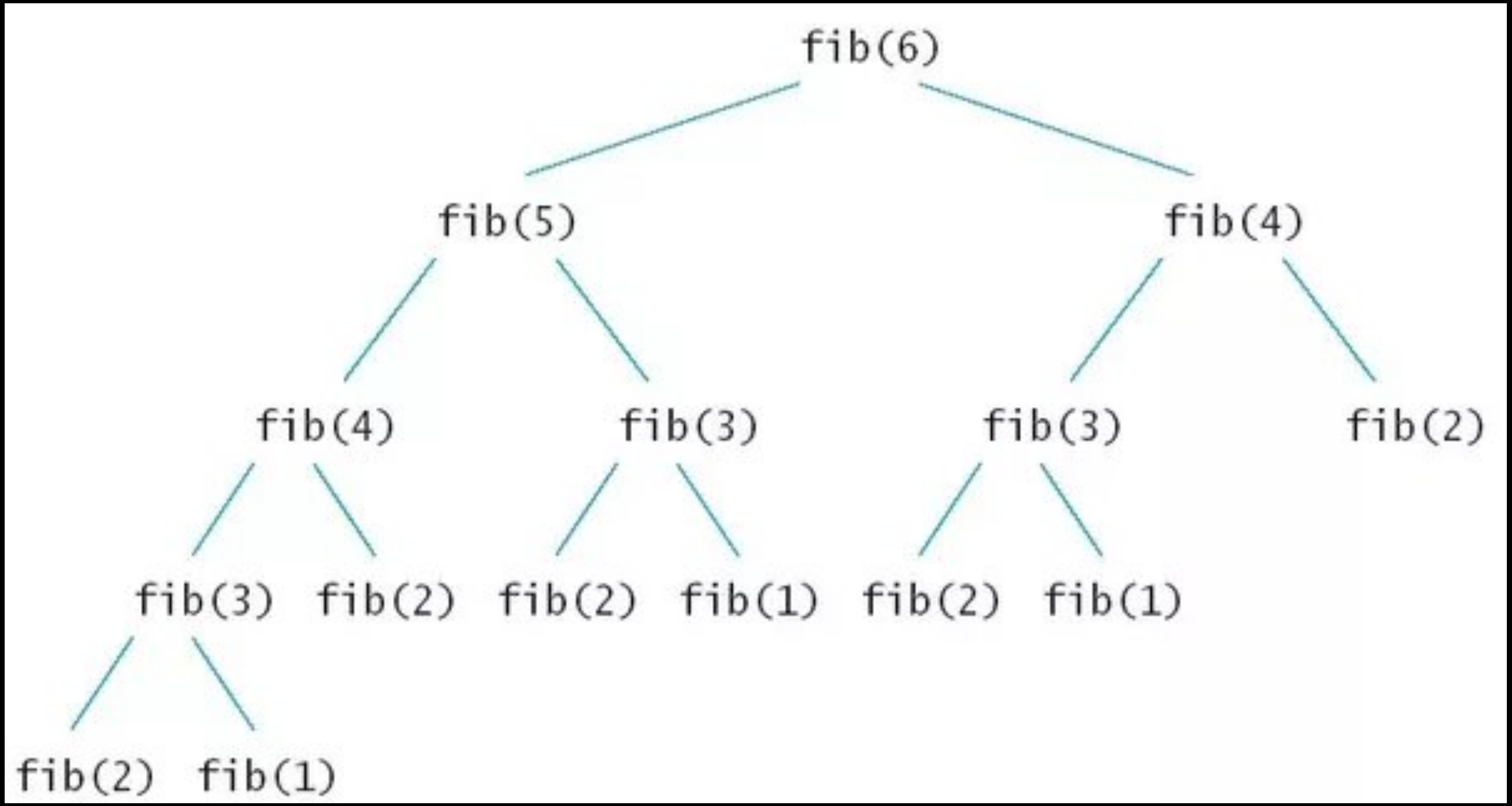
```c
#include <stdio.h>
long long fib(long long n)
{
    if(n == 0) return 0;
    else if(n == 1) return 1;
    else
        return (fib(n - 1) + fib(n - 2));
}
int main()
{
    long long n;
    printf("Please enter a number: ");
    scanf("%lld", &n);
    printf("The %lldth fibonacci number is: %lld\n", n,
fib(n));
}
```

Code with memoization

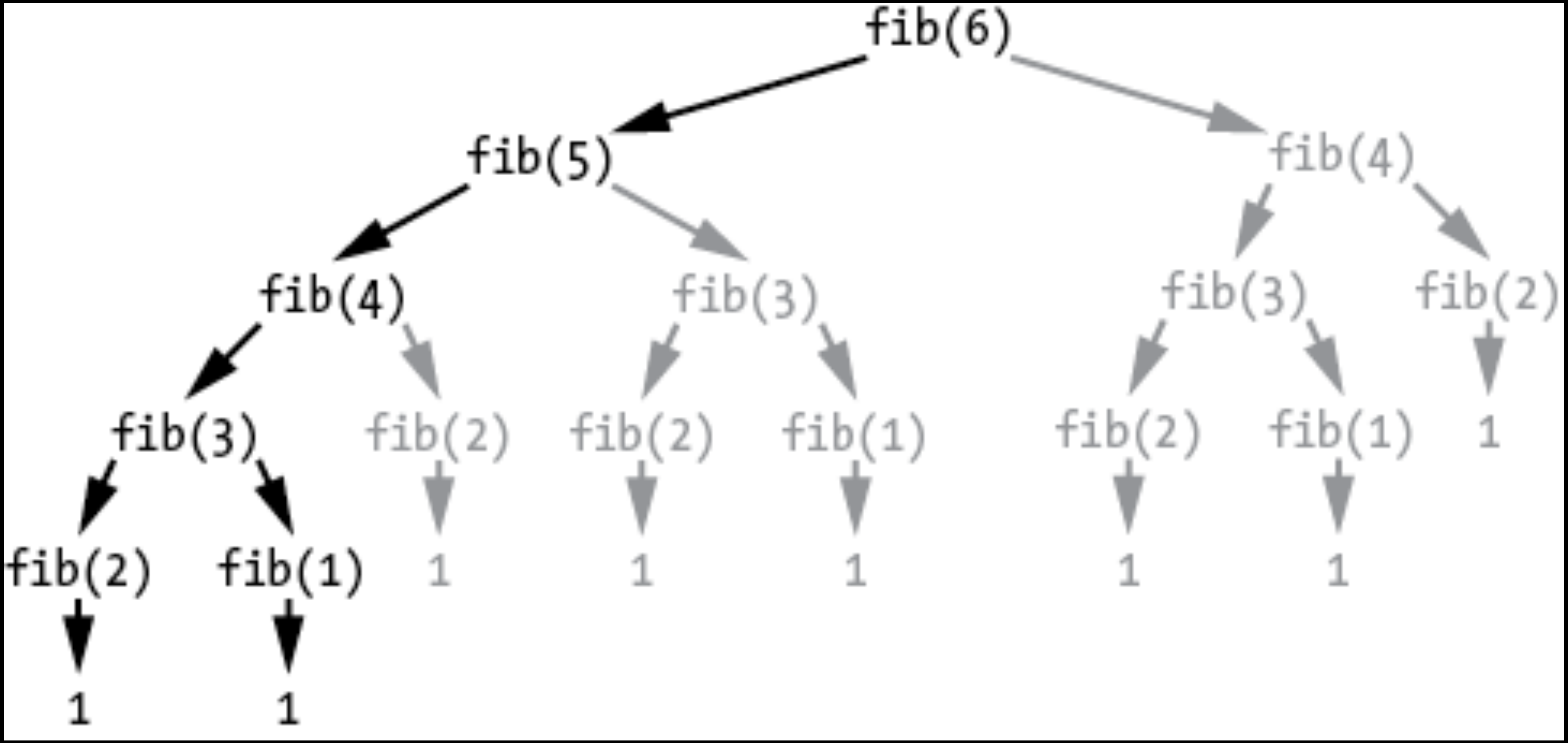


```c
#include<stdio.h>
int arr[1000];

long long fib(long long n){
    if(arr[n] ≠ -1) return arr[n];
    else return arr[n] = (fib(n-1)+fib(n-2));
}
int main(){
    for(int i = 0;i < 1000;i++) arr[i] = -1;
    arr[0] = 0;
    arr[1] = 1;
    long long n;
    printf("Please enter a number: ");
    scanf("%lld",&n);
    printf("The %lldth fibonacci number is:
%lld\n",n,fib(n));
```

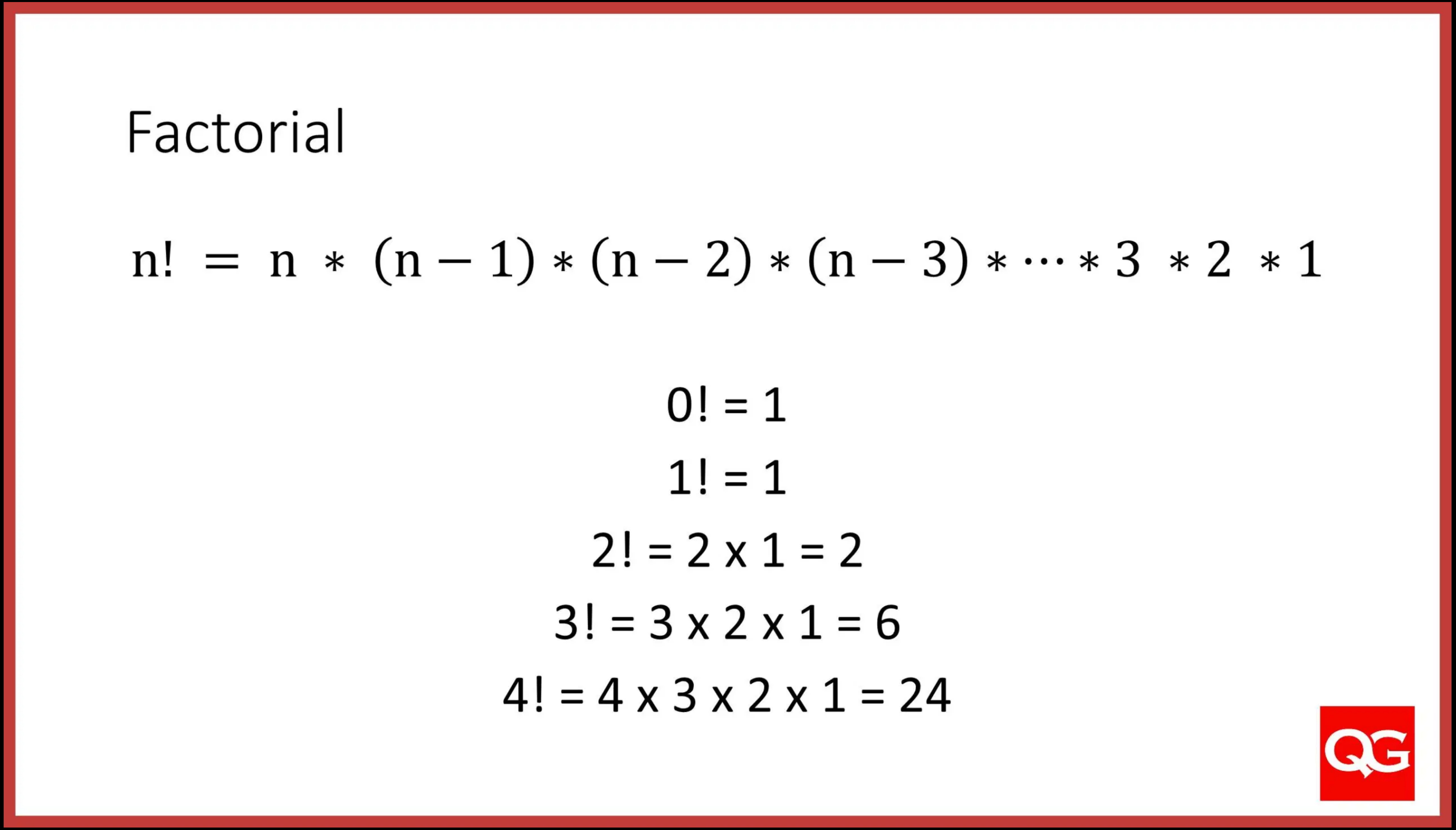
Recursive Tree without memoization



R.T. with memoization



Factorial



Factorial

$$n! = n * (n-1) * (n-2) * (n-3) * \cdots * 3 * 2 * 1$$

0! = 1

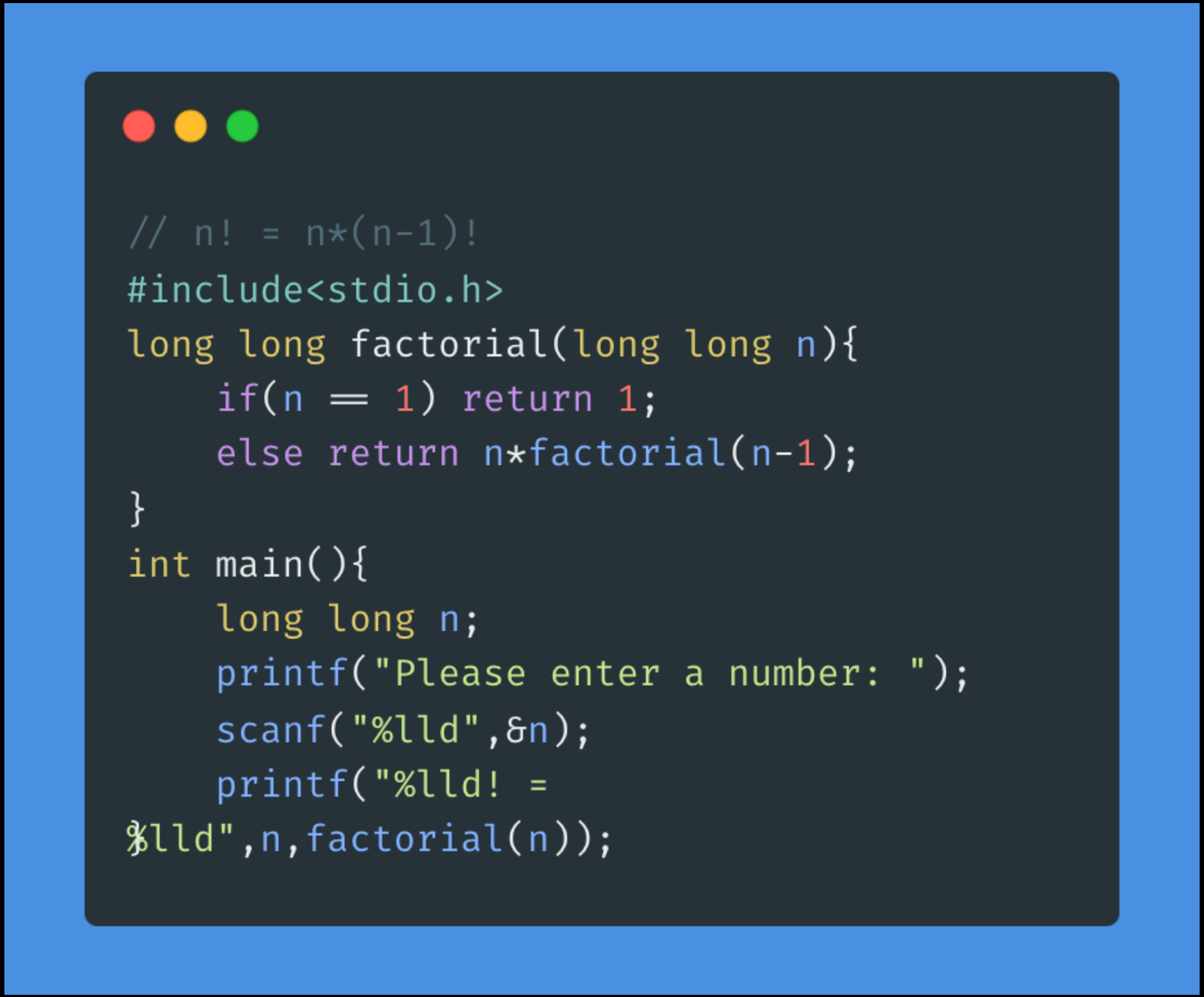
1! = 1

2! = 2 x 1 = 2

3! = 3 x 2 x 1 = 6

4! = 4 x 3 x 2 x 1 = 24

Factorial Recursive Code



```
// n! = n*(n-1)!
#include<stdio.h>
long long factorial(long long n){
    if(n == 1) return 1;
    else return n*factorial(n-1);
}
int main(){
    long long n;
    printf("Please enter a number: ");
    scanf("%lld",&n);
    printf("%lld! =
%lld",n,factorial(n));
```