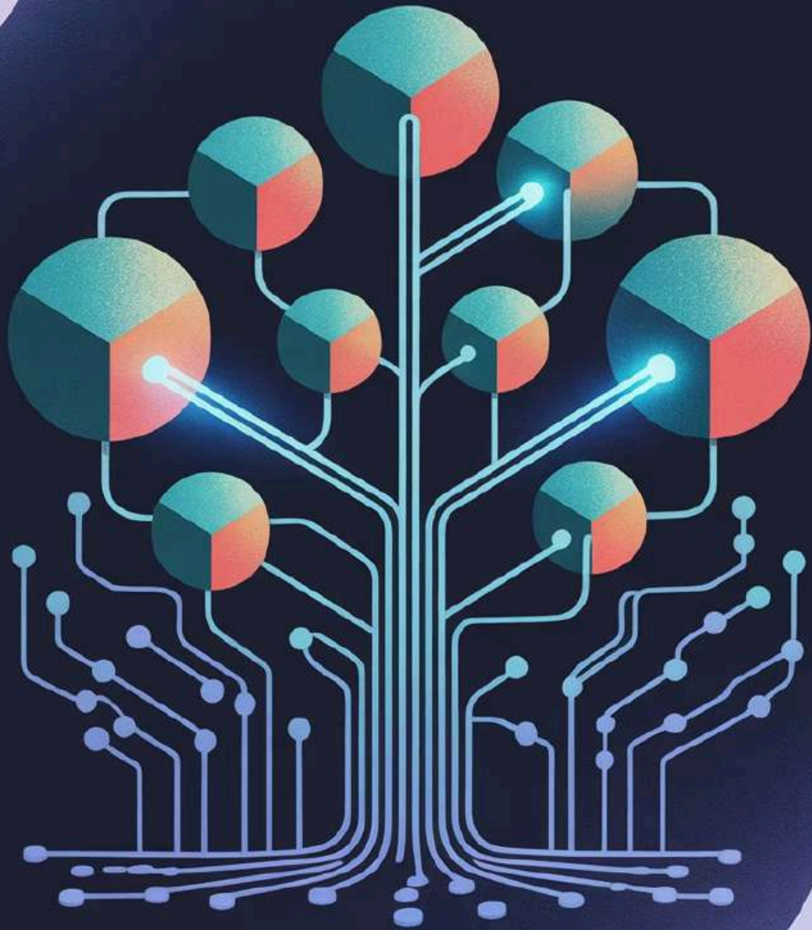
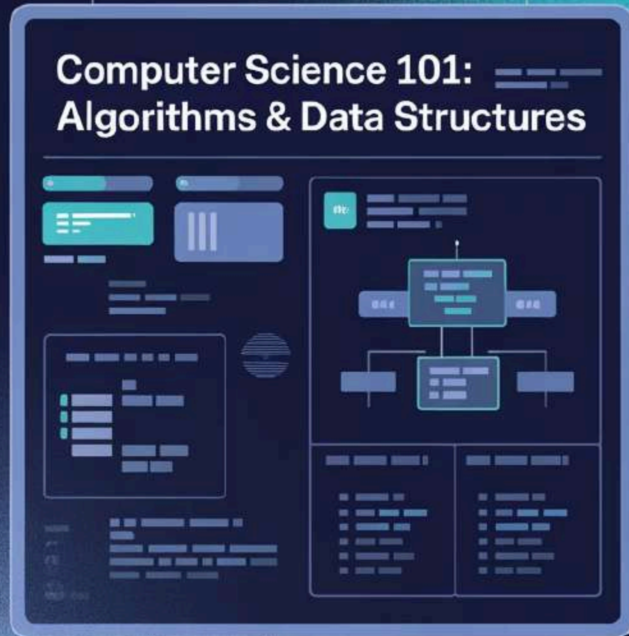


Search Algorithms

Exploring linear and binary search: from theory to practice in C++



Lecture Plan: "Search Algorithms"

01

Motivation and Examples

Why search algorithms are needed in everyday life and programming

02

Classification of Methods

Linear vs. binary search, iterative and recursive approaches

03

Practical Implementation

Writing C++ code, complexity and performance analysis

04

Comparison and Conclusions

When to use which algorithm, practical recommendations

Lecture Title

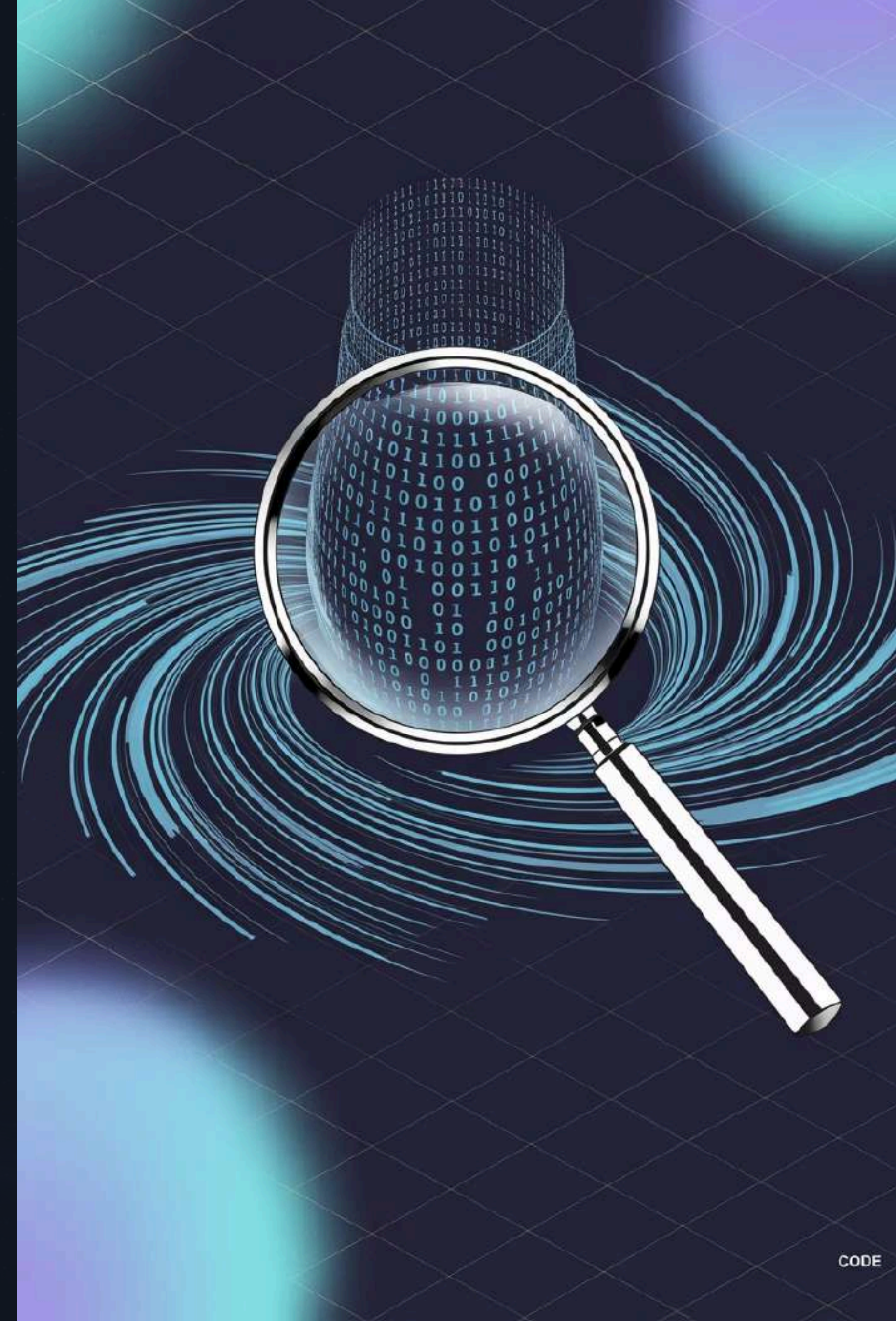
Search Algorithms

Topics to Cover

- Linear Search
- Binary Search
- Iterative Approach
- Recursive Approach

Lecture Goal

Understand the principles of search algorithms, learn to implement them in C++, and choose the optimal approach for specific tasks. We will explore how computers efficiently find necessary data in arrays and other structures.



Motivation and Real-Life Examples

Searching for a book on a shelf

We browse books one by one until we find the desired one. This is a classic example of linear search — a simple, but not always fast approach.

Searching for a word in a dictionary

We open the dictionary in the middle and decide whether to search in the first or second half. We repeat until the word is found — this is the principle of binary search.

Computer Systems

Searching for a contact on a phone, a file in a folder, a record in a database — search algorithms are used everywhere to quickly retrieve the necessary information.

Search algorithms are the foundation of many software solutions. Understanding their principles will help create more efficient applications.

Main Types of Algorithm Complexity

Asymptotic notations are used to evaluate algorithm performance, describing how the execution time or memory usage of an algorithm grows with an increase in input data size.

Big-O (O-notation)

The upper bound of function growth – the worst-case scenario for execution.

Big-Omega (Ω-notation)

The lower bound of function growth – the best-case scenario for execution.

Theta (Θ-notation)

A precise asymptotic estimate, covering both worst and best cases.

Complexity

Main Types of Algorithm Complexity

O(1) - Constant

Time does not depend on data size. Example: accessing an array element by index

O(log n) - Logarithmic

Time grows slowly. Example: binary search

O(n) - Linear

Time is proportional to the number of elements. Example: linear search

4

O(n log n) - Linearithmic

Efficient sorting algorithms. Example: MergeSort

5

$O(n^2)$ - Quadratic

Each element is compared with all others. Example: bubble sort

O(2^n) - Exponential

Time doubles with each increment. Example: subset enumeration

Visual Comparison of Complexity Growth

Comparison table for different data sizes:

n	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	O(2^n)
10	1	3	10	33	100	1024
100	1	7	100	664	10000	huge number
1000	1	10	1000	9966	1000000	impossible

100x

$O(n^2)$ vs O(n) (n=100)

For 100 elements, quadratic complexity requires 100 times more operations than linear.

100x

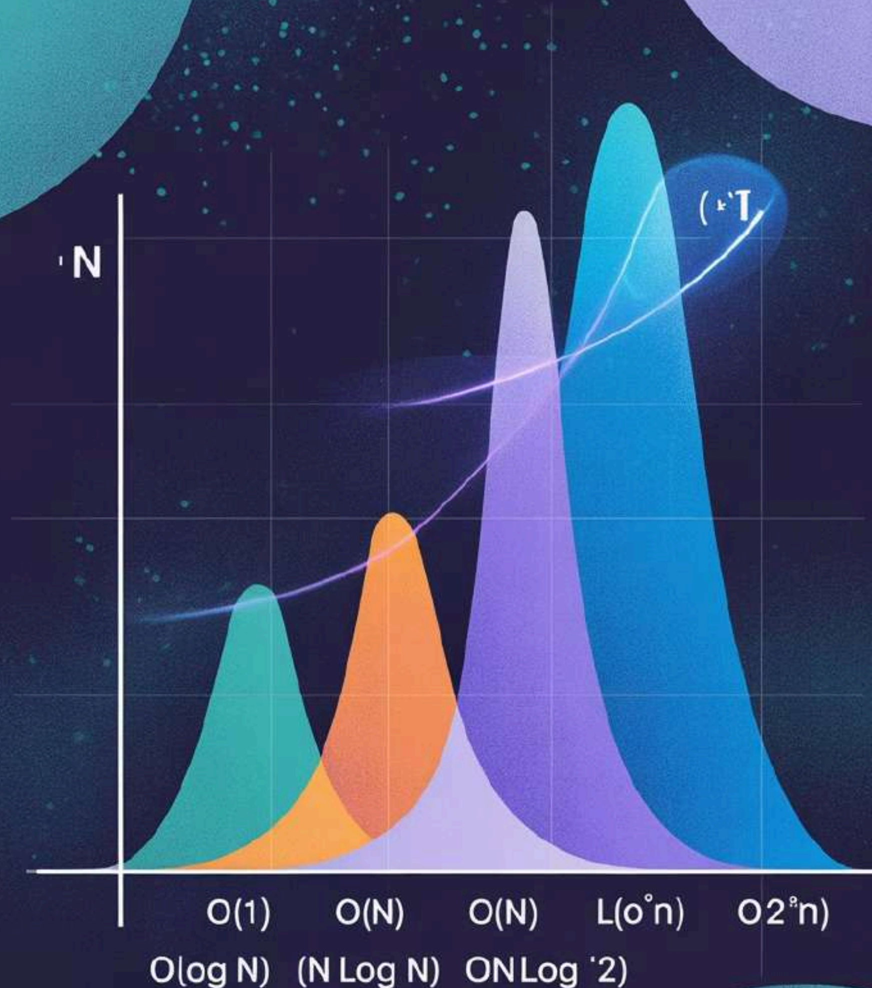
O(n) vs O(log n) (n=1000)

With 1000 elements, linear search is significantly slower than logarithmic.

~1M

O(2^n) (n=10)

Exponential complexity quickly becomes infeasible even for small n (1024 operations for n=10).



Practical Complexity Analysis

1	**Time Complexity** • Count the number of basic operations • Consider the worst case • Ignore constants and lower-order terms
2	**Space Complexity** • Additional memory used by the algorithm • Do not count input data • Important for large data volumes
3	**Practical Considerations** • Data size in real-world problems • Frequency of operation execution • Available system resources

Let's add examples of analyzing simple algorithms to reinforce understanding of these concepts.

Algorithm Complexity Evaluation

In the world of programming, algorithm efficiency is key to creating fast and scalable solutions. Complexity evaluation allows us to predict how an algorithm will perform as data volume increases, and to choose the most suitable tool for each task.

Performance

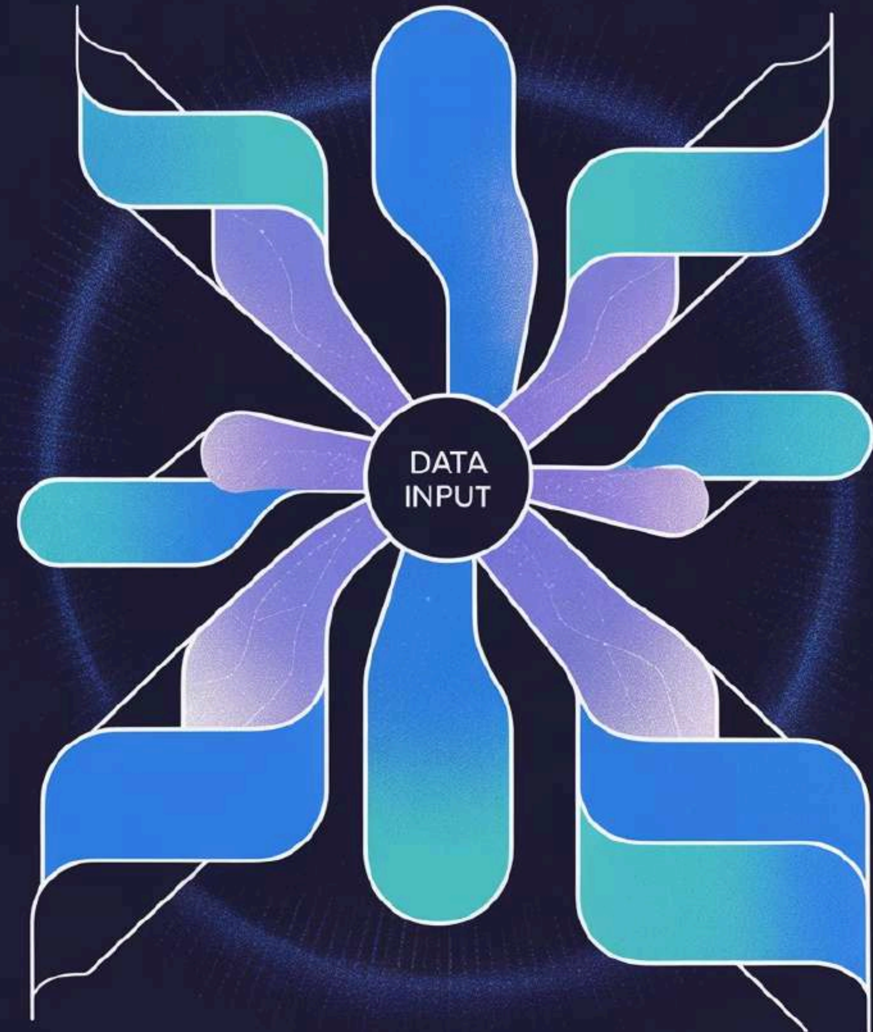
How quickly an algorithm completes a task depending on the size of the input data.

Scalability

The algorithm's ability to efficiently process large volumes of information.

Resource Optimization

Choosing the best solution considering execution time and memory usage.



Classification of Search Algorithms

Linear Search

Sequentially checks each element. Works with any data but is slow for large volumes.

- Versatility
- Simplicity of implementation
- Does not require sorting

Binary Search

Divides the search area in half at each step. Very fast, but requires sorted data.

- High speed
- Logarithmic complexity
- Requires sorting

Implementation variants: **iterative** (using loops) and **recursive** (function calls itself)

Linear Search (Sequential Search)

How it works

Checks each element of an array in order until the desired value is found or all elements have been examined.

- Suitable for unsorted data
- Time complexity: O(n)
- Space complexity: O(1)

Real-life example

Searching for a lost key in a messy pile of items — checking each item until the key is found.

```
int linearSearch(int arr[], int n, int x) {
 for (int i = 0; i < n; i++) {
 if (arr[i] == x)
 return i; // found
 }
 return -1; // not found
}
```

Advantages and Disadvantages of Linear Search

✅ Advantages

- **Simplicity of implementation** — just a few lines of code
- **Versatility** — works with any data
- **No pre-processing required** — can be applied immediately
- **Efficient for small arrays** — for $n < 100$, the difference is insignificant

❌ Disadvantages

- **Slow on large data sets** — time grows linearly
- **Inefficient for frequent searches** — checks all elements every time
- **Does not utilize order** — ignores data sorting

> Linear search is optimal when data is small or constantly changing

Binary Search

Divide and Conquer Principle

We divide the problem in half at each step, discarding half of the elements.

Requires Sorting

Works only with a sorted data array.

Logarithmic Complexity

O(log n) — very fast even for millions of elements.

Real-life example: searching for a word in a dictionary or a surname in an alphabetical list. We intuitively use the binary principle by opening the dictionary roughly in the middle.

How Binary Search Works

Step 1: Find the Middle

Calculate the middle index of the array and compare the element at this position with the target value

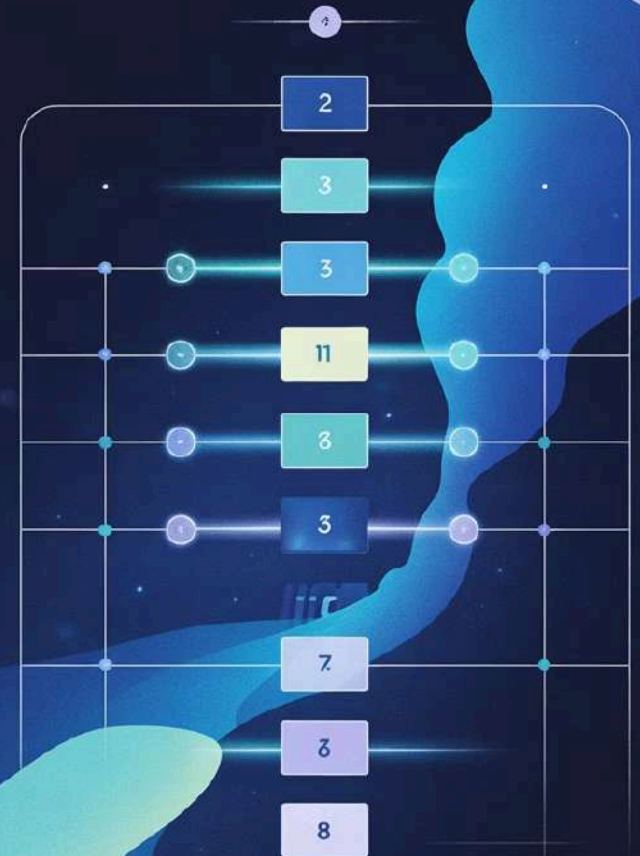
Step 2: Choose Direction

If the target is less than the middle element, go to the left half; if greater, go to the right half of the array

Step 3: Repeat the Process

Repeat the steps for the selected half until the element is found or the search area becomes empty

With each iteration, the number of elements to check is halved!



Iterative Binary Search

Implementation using a `while` loop. We use pointers for the left and right boundaries of the search area.

```
int binarySearchIterative(int arr[], int n, int x) {
    int left = 0, right = n - 1;

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

        if (arr[mid] == x)
            return mid;
        else if (arr[mid] < x)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;  // not found
}
```

Advantages

- Saves memory
- Faster execution
- Easier to understand

Important detail

The formula `left + (right - left) / 2` prevents overflow when working with large numbers

Recursive Binary Search

An implementation where the function calls itself with a reduced search range.

```
int binarySearchRecursive(int arr[], int left, int right, int x) {
    if (left > right)
        return -1;  // base case

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

    if (arr[mid] == x)
        return mid;
    else if (arr[mid] > x)
        return binarySearchRecursive(arr, left, mid - 1, x);
    else
        return binarySearchRecursive(arr, mid + 1, right, x);
}
```

Base Case

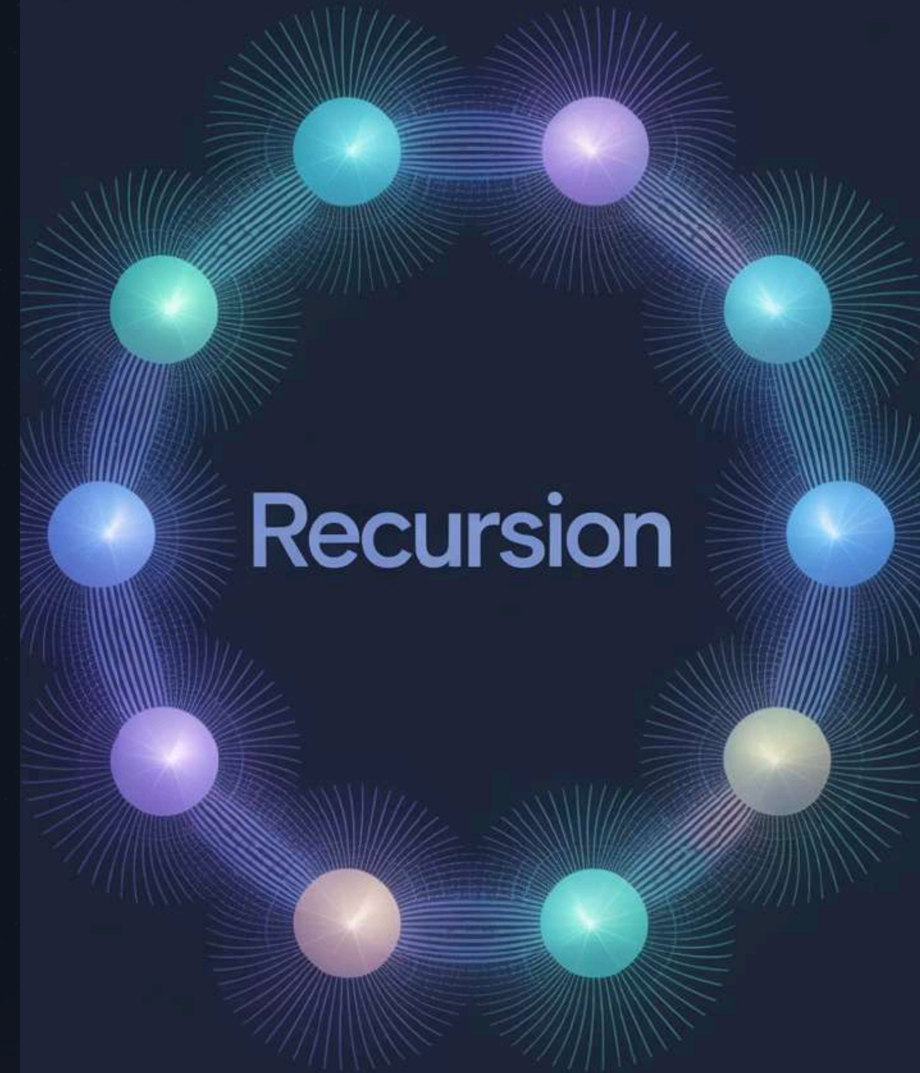
Condition for stopping recursion: `left > right`

Recursive Call

The function calls itself with new search boundaries

Code Elegance

A more natural expression of the "divide and conquer" algorithm



Comparison of Linear and Binary Search

Algorithm	Requirements	Complexity	Applicability
Linear	None	O(n)	Small or unsorted arrays
Binary	Sorted	O(log n)	Large arrays, frequent searches

1000

Linear: 1000 operations

for an array of 1000 elements

10

Binary: 10 operations

for the same array of 1000 elements

100X

Speed Difference

binary search is 100 times faster

Real-World Examples

Linear Search

- **Attendance Check**

  A teacher reads the list of students in order and marks those present

- **Email Search**

  Reviewing emails in an inbox one by one to find the desired one

- **Searching an Unordered List**

  Finding an item in a randomly arranged warehouse

Binary Search

- **Phone Book Search**

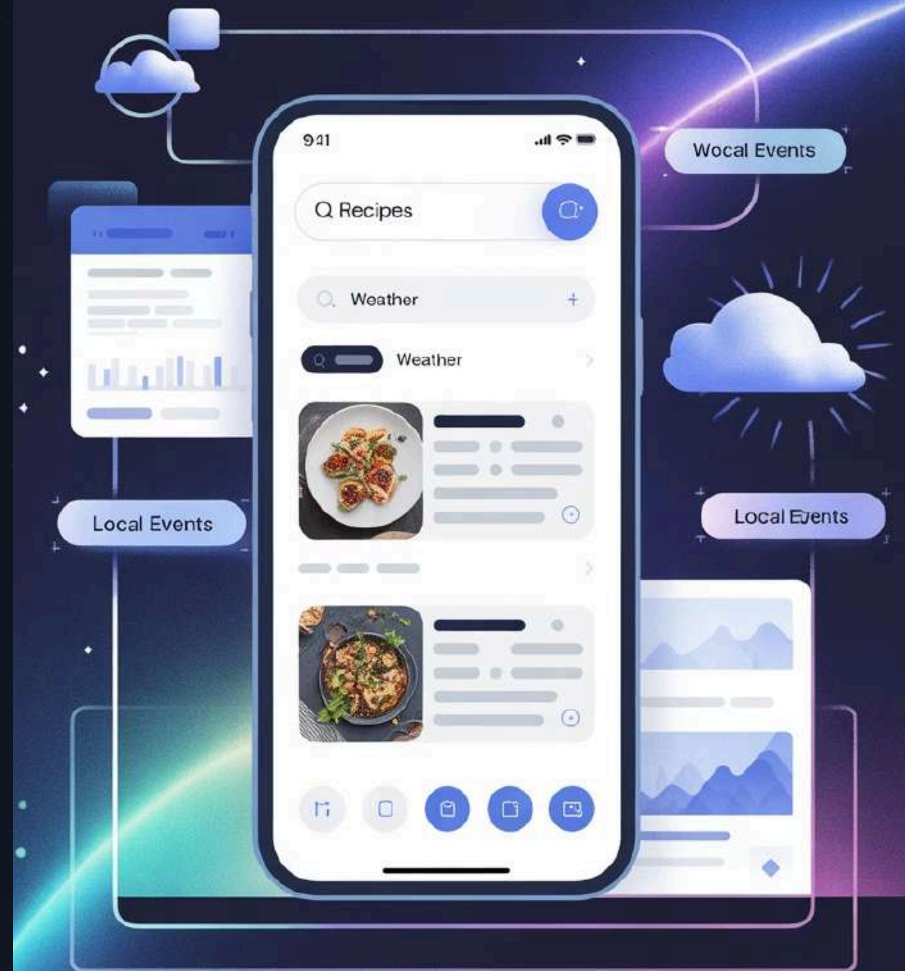
  Finding a last name in an alphabetical directory — opening approximately to the correct letter

- **Number Guessing Game**

  Playing "guess the number from 1 to 100" — each time halving the range

- **Optimal Weight Search**

  Determining the maximum load a bridge can withstand



Practical Nuances

Sorting Cost

Binary search requires prior sorting of data, which takes O(n log n) time. This is only justified for multiple searches.

Data Size Matters

For arrays with fewer than 50-100 elements, the performance difference is insignificant. Linear search might even be faster due to its simplicity.

Built-in Functions in C++

The standard library provides: `std::find` (linear), `std::binary_search`, `std::lower_bound`, `std::upper_bound`.

Choosing Data Structure

For frequent searches, consider using hash tables (O(1)) or balanced search trees.

Collecttive Takeaways

Lecture Outcomes

Fundamental Nature of Algorithms

Search algorithms underpin the operation of most software systems — from search engines to databases.

Balance of Simplicity and Efficiency

Linear search is universal and simple, but slow. Binary search is fast, but requires additional conditions.

Implementation Choice

Iterative and recursive versions of algorithms have their own advantages depending on the context of use.

> Understanding the principles of search algorithms is key to creating efficient software solutions

Questions for Discussion

1 When is linear search preferable?

Consider situations where simplicity and universality are more important than speed. Discuss cases with small arrays, frequent data changes, or when sorting is too expensive.

2 Recursion vs iteration in binary search

Compare code readability, performance, and memory usage. In which cases is the recursive version more natural to understand?

3 Why use ready-made solutions?

Discuss the advantages of library functions: optimization, testing, standardization. When is it worthwhile to implement algorithms yourself?

Additional Materials

Practical Exercises

- Implement both search algorithms
- Measure execution time on arrays of different sizes
- Compare the performance of iterative and recursive versions
- Analyze the impact of data type on search speed

Topics for Further Study

- Interpolation search
- Exponential search
- Searching in multidimensional arrays
- Hash tables and their applications

Next Lecture: **Sorting Algorithms** — we will learn how to prepare data for efficient binary search