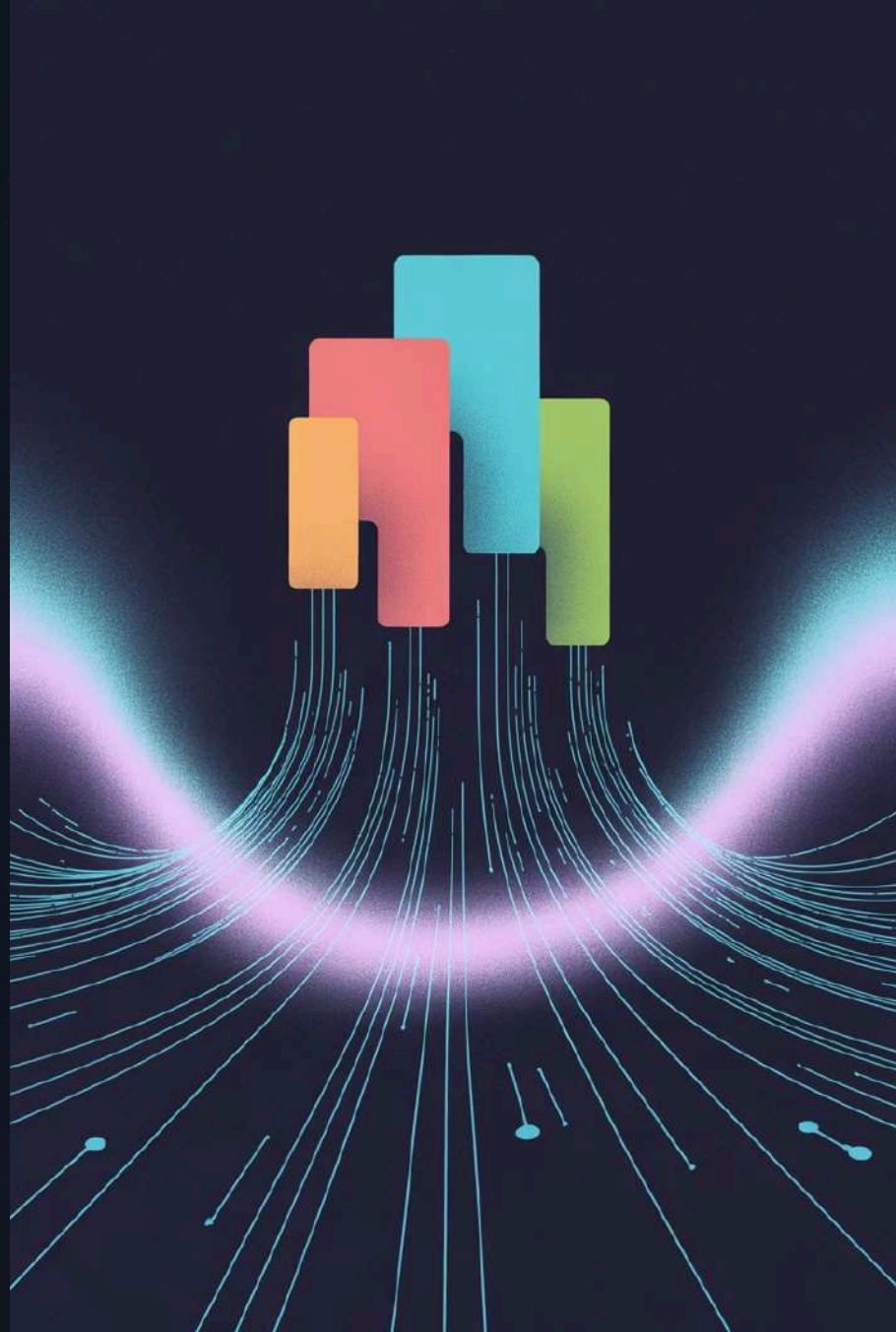
Sorting Algorithms II

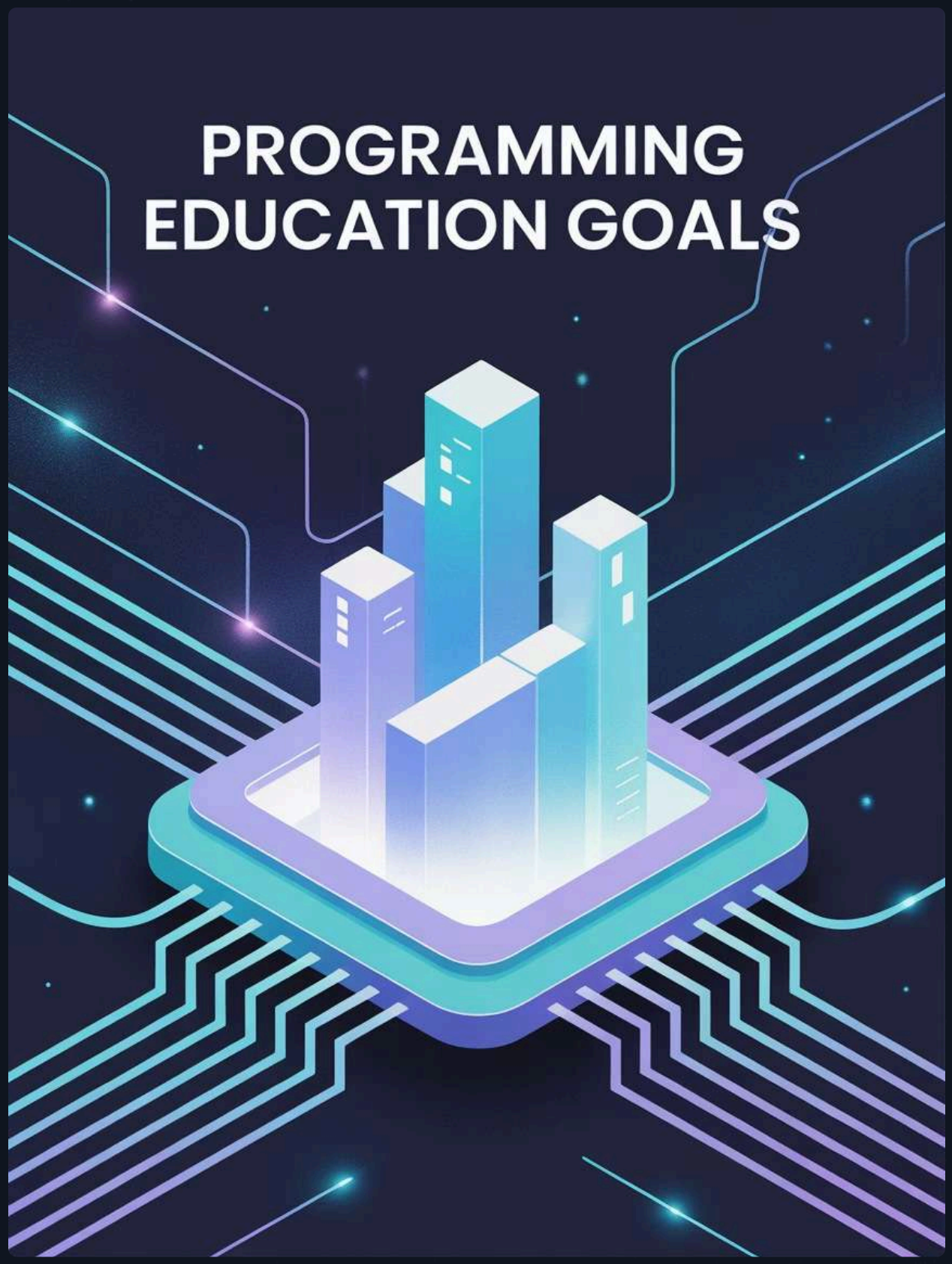
Quicksort and Mergesort are efficient algorithms with O(n log n) time complexity



Lecture: Sorting Algorithms II

Lab 13: Quicksort, Mergesort

Welcome to the lecture dedicated to studying two fundamental sorting algorithms that form the basis of efficient data processing in modern programming.



Learning Objectives

Understanding Principles

Explore the fundamentals of QuickSort and MergeSort algorithms, based on the "divide and conquer" strategy

Practical Implementation

Learn to implement efficient sorting algorithms in C++

Complexity Analysis

Understand the time and space complexity of O(n log n) algorithms

Motivation for Learning

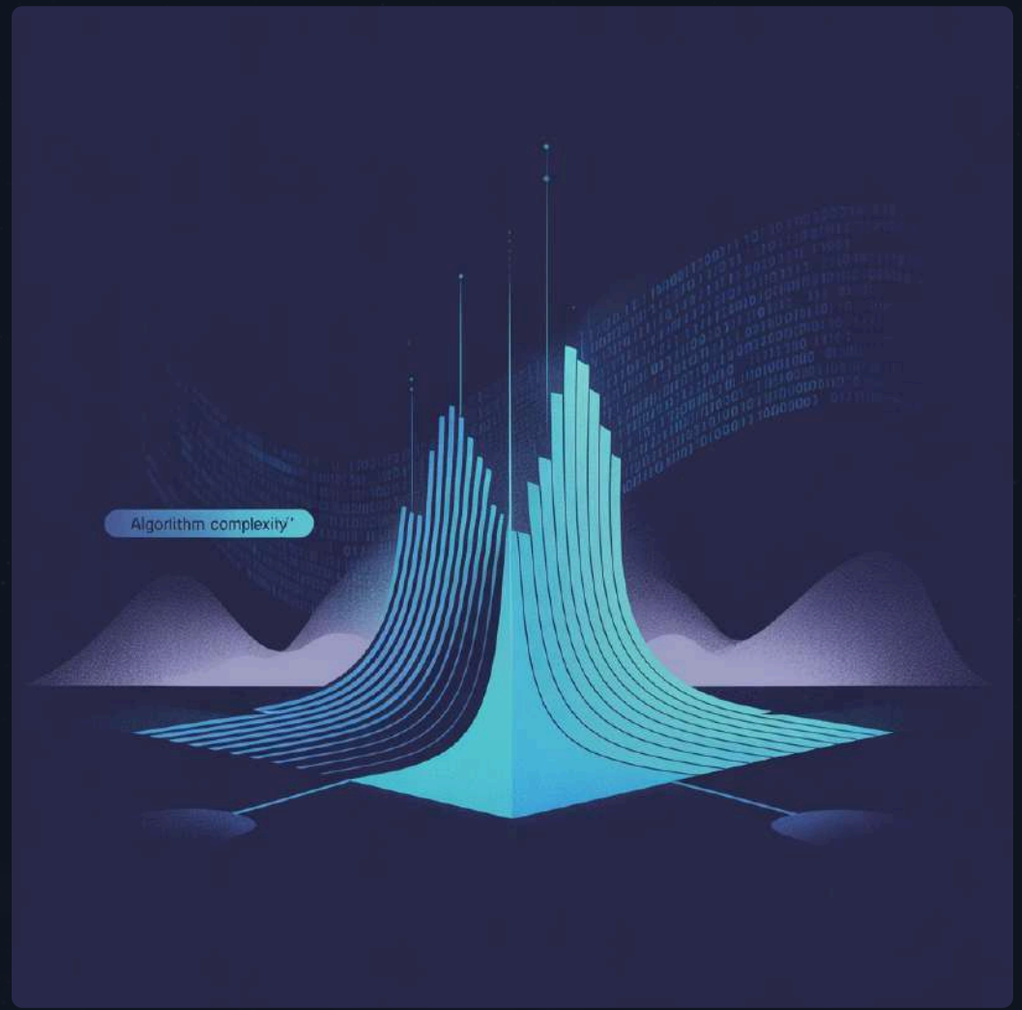
Problem with Simple Algorithms

Simple sorting algorithms (Bubble Sort, Selection Sort, Insertion Sort) have quadratic complexity $O(n^2)$, which makes them inefficient for processing large volumes of data.

- Bubble Sort — $O(n^2)$
- Selection Sort — $O(n^2)$
- Insertion Sort — $O(n^2)$

Solution: O(n log n) Algorithms

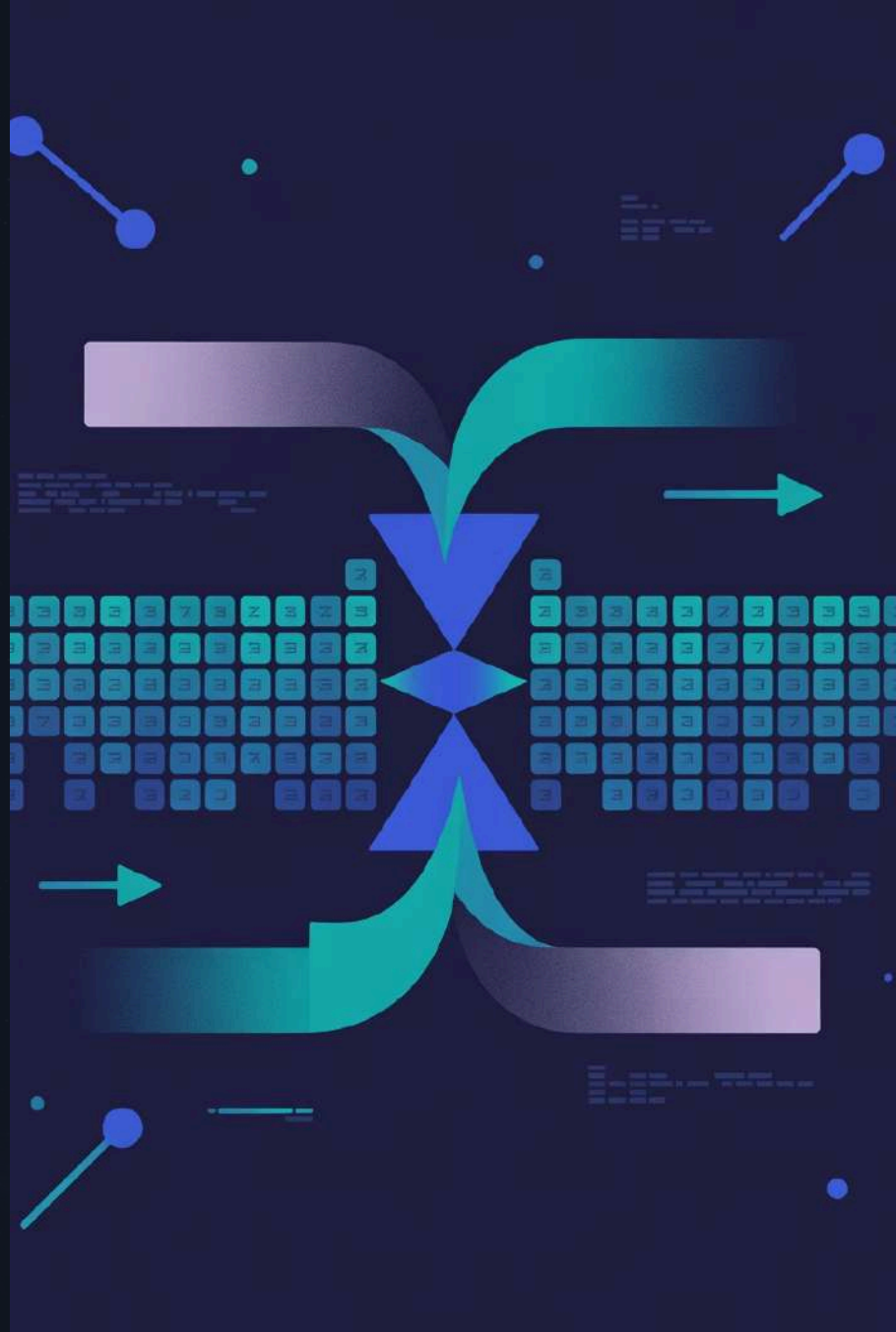
QuickSort and MergeSort use the "divide and conquer" principle to achieve significantly better performance on large data arrays.



QuickSort

Quick Sort

Quick Sort is one of the most efficient and widely used sorting algorithms, based on the "divide and conquer" strategy.



How QuickSort Works

01

Choosing a Pivot Element

Select an element from the array as the pivot. This can be the first, last, a random element, or the median.

02

Partitioning the Array

Rearrange the array so that elements smaller than the pivot are to its left, and larger elements are to its right.

03

Recursive Sorting

Recursively apply the algorithm to the left and right sub-arrays until fully sorted.

> **Real-world analogy:** Imagine seating people at a table — those under 30 sit on the left, those over 30 sit on the right. Then repeat the process for each group.

QuickSort Example

Original array: [5, 3, 7, 2, 6]

1. **Choose pivot = 5**

   Take the last element as the pivot

2. **Partitioning**

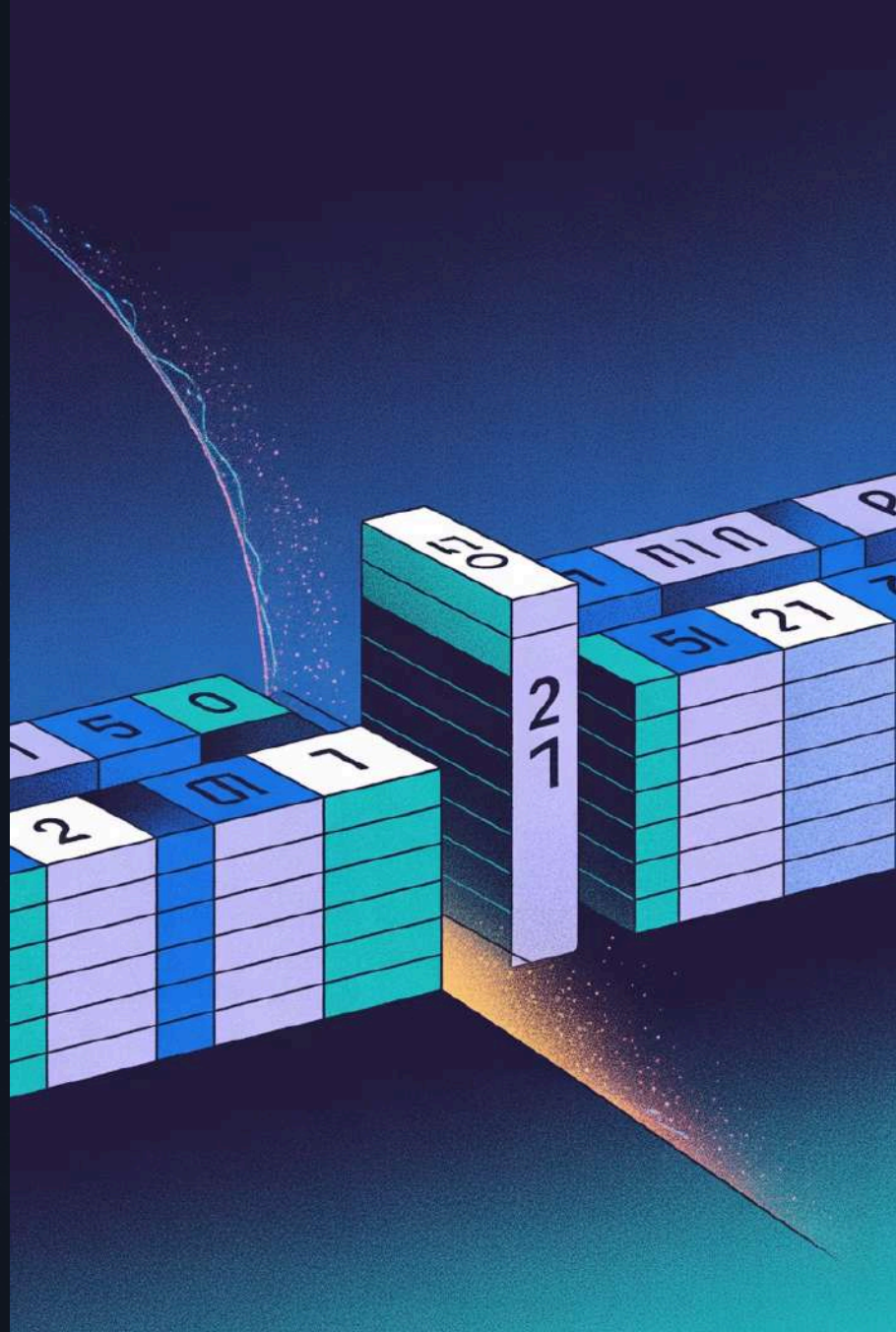
   Left: [3, 2] < 5, Right: [7, 6] > 5

3. **Recursive sorting**

   Sort [3, 2] → [2, 3] and [7, 6] → [6, 7]

4. **Result**

   [2, 3] + [5] + [6, 7] = [2, 3, 5, 6, 7]



QuickSort Implementation in C++

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // Выбираем последний элемент как pivot
    int i = low - 1;        // Индекс меньшего элемента

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);    // Сортируем левую часть
        quickSort(arr, pi + 1, high);   // Сортируем правую часть
    }
}
```

Important: The partition function performs the main task of partitioning the array relative to the pivot element.

QuickSort Complexity

Average Case

With random pivot selection and even partitioning

Worst Case

When the array is already sorted and pivot is chosen poorly

Memory

Recursion depth averages log n

QuickSort demonstrates excellent performance in the average case, but it's important to consider the possibility of degradation to quadratic complexity in the worst case.

QuickSort Optimizations

Pivot Selection

- Random element
- Median of three elements
- First or last element

Hybridization

Switching to Insertion Sort for small subarrays (usually < 10 elements) to improve efficiency

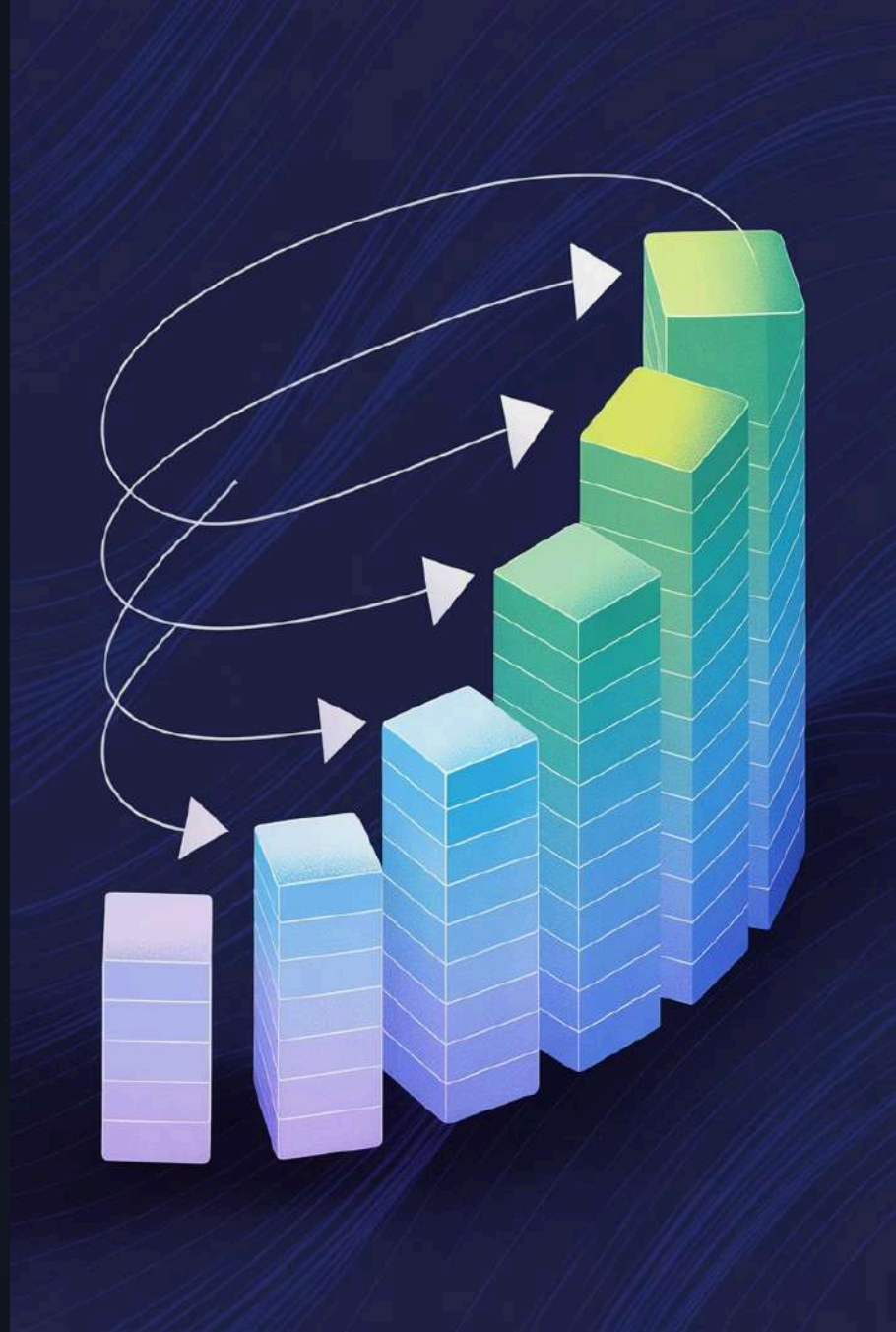
Practical Application

Used in standard libraries: std::sort in C++, Arrays.sort() in Java

MergeSort

Merge Sort

Merge Sort is a stable sorting algorithm that guarantees a time complexity of O(n log n) in all cases.



How MergeSort Works

Divide

Divide the array into two equal halves until arrays of a single element are obtained

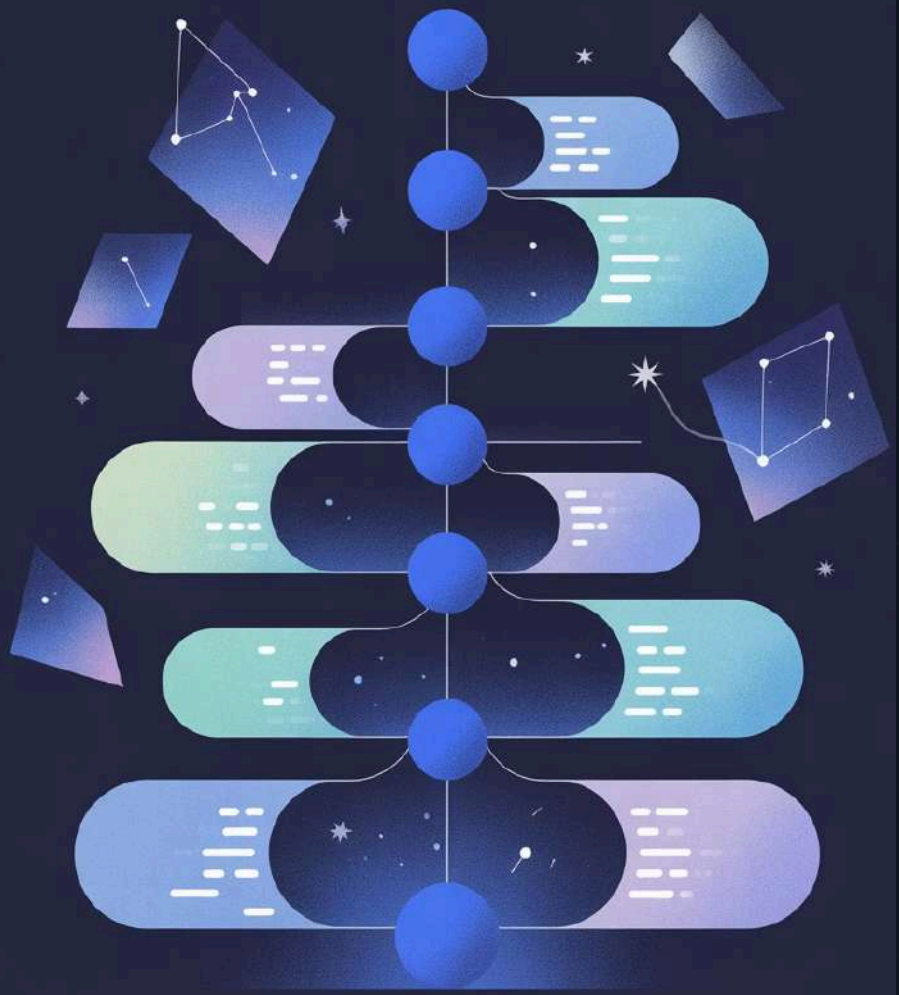
Sort

Recursively sort each half of the array, applying the same algorithm

Merge

Combine the sorted parts into a single ordered array

> **Real-life analogy:** Merging two already sorted stacks of playing cards into one ordered stack.



MergeSort Example

Initial array: [5, 2, 4, 1]

1

Divide

[5, 2, 4, 1] → [5, 2] and [4, 1]

2

Further Division

[5, 2] → [5], [2]; [4, 1] → [4], [1]

3

Merge Pairs

[5], [2] → [2, 5]; [4], [1] → [1, 4]

4

Final Merge

[2, 5] + [1, 4] → [1, 2, 4, 5]

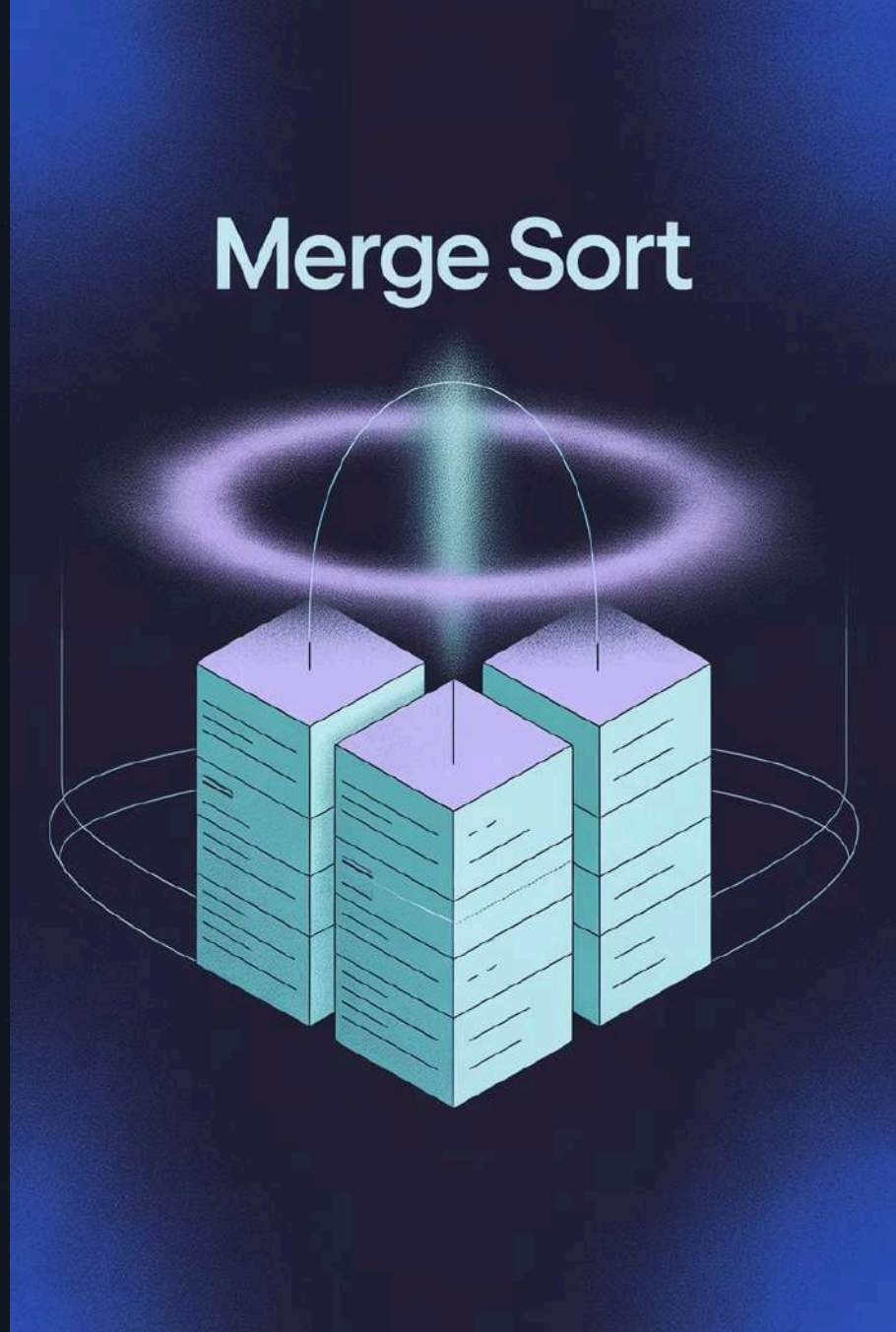
MergeSort Implementation in C++

```
void merge(int arr[], int l, int m, int r) {
 int n1 = m - l + 1;
 int n2 = r - m;
 vector L(n1), R(n2);

 // Copy data to temporary arrays
 for (int i = 0; i < n1; i++) L[i] = arr[l + i];
 for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

 // Merge the temporary arrays back into arr[l..r]
 int i = 0, j = 0, k = l;
 while (i < n1 && j < n2) {
 if (L[i] <= R[j]) arr[k++] = L[i++];
 else arr[k++] = R[j++];
 }

 // Copy the remaining elements
 while (i < n1) arr[k++] = L[i++];
 while (j < n2) arr[k++] = R[j++];
}

void mergeSort(int arr[], int l, int r) {
 if (l < r) {
 int m = l + (r - l) / 2;
 mergeSort(arr, l, m);
 mergeSort(arr, m + 1, r);
 merge(arr, l, m, r);
 }
}
```

MergeSort Complexity

O(n lo...	O(n)	100%
Time Complexity	**Space Complexity**	**Stability**
Guaranteed in all cases: best, average, and worst	Requires additional memory for temporary arrays	Preserves the relative order of equal elements

MergeSort is characterized by predictable performance and stability, making it an ideal choice for mission-critical applications.



Comparison of QuickSort and MergeSort

Algorithm	Average Complexity	Worst Case	Memory	Stability
QuickSort	O(n log n)	$O(n^2)$	O(log n)	No
MergeSort	O(n log n)	O(n log n)	O(n)	Yes

The choice between algorithms depends on specific requirements: QuickSort is faster in practice, while MergeSort is more predictable and stable.

Practical Applications

QuickSort

- Standard libraries (std::sort)
- Real-time systems
- Embedded systems with limited memory
- General purpose sorting

MergeSort

- External sorting of large files
- Stable sorting of critical data
- Parallel computing
- Linked lists

External Sorting: When working with data that does not fit into RAM, MergeSort performs particularly well due to sequential data access.

Conclusion and Questions for Reflection

Key Takeaways

- Both algorithms are significantly more efficient than simple quadratic methods
- QuickSort is fast in practice, but can degrade
- MergeSort is stable and predictable
- Often used in combination with other algorithms

Discussion Questions:

1. **What is the fundamental difference in the approaches of QuickSort and MergeSort?**
2. **Why is QuickSort often faster than MergeSort, even though they have the same average complexity?**
3. **In what situations does MergeSort outperform QuickSort?**