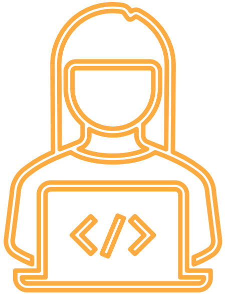
METROPOLITAN TIRANA UNIVERSITY

Course: Data Structures and Algorithms

Searching and Sorting Algorithms

Evis Plaku

Because life is too short for unsorted data



```
Arrays.sort(data)
Arrays.binarySearch(data, key)
```

Searching and sorting in real life

Searching and sorting in coding

Real world applications of searching and sorting

METROPOLITAN TIRANA UNIVERSITY

Google search results for PageRank

Amazon product listing by price and relevance

Bank fraud detection for suspicious activity

Netflix and YouTube recommendations

Stock market analysis by price changes

Job portals to filter, search and sort

Searching algorithms

Introduction to searching

Finding specific data efficiently in large datasets

- Search retrieves data from an unsorted or sorted collection

- Efficient searching reduces time complexity in large datasets

- Optimized search improves speed in real-time applications

Linear search

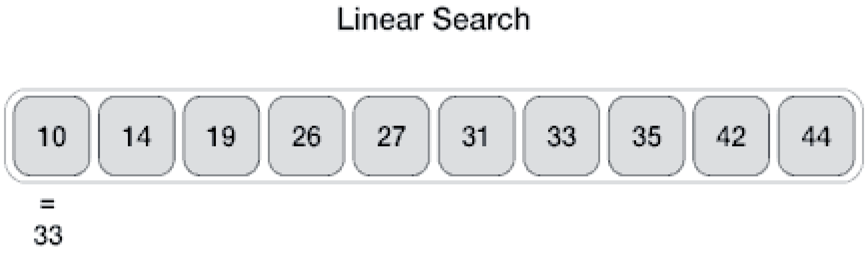
METROPOLITAN TIRANA UNIVERSITY

- Simple search method checking each element sequentially

- Iterates through elements one by one until a match is found

- Works on unsorted data but is inefficient for large datasets

- Used in small datasets, address books, and basic lookups




```java
public static int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) {
            return i;    // Return index if found
        }
    }
    return -1;          // Return -1 if not found
}
```

Click here to view full example

Real – word example of linear search

- Example: Finding an employee's name in a non-sorted employee collection

- Used when employee lists are not organized or indexed

```java
public static int linearSearch(Employee[] employees, String targetName) {
        for (int i = 0; i < employees.length; i++) {
            if (employees[i].name.equals(targetName)) {
                return i;    // Return index if the employee is found
            }
        }
        return -1;          // Return -1 if the employee is not found
    }
```

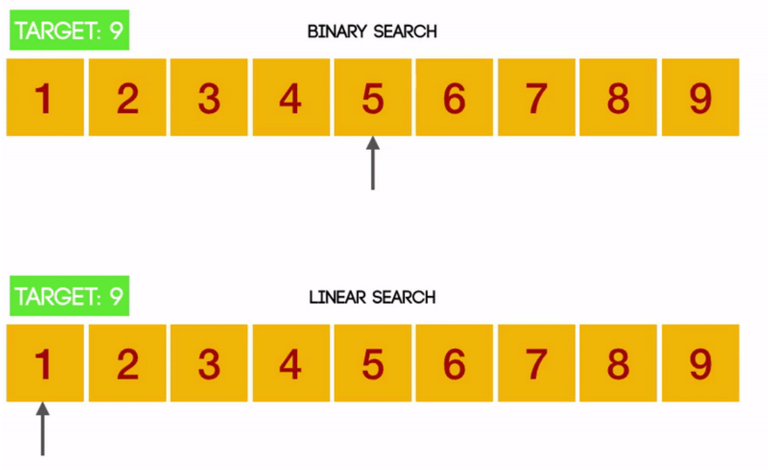
Click here to view full example

Binary search

METROPOLITAN TIRANA UNIVERSITY

Efficient search method on sorted arrays using divide and conquer

- Divides the search space in half with each iteration
- Requires sorted data to function correctly
- Faster than linear search for large datasets
- Used in search engines, databases, and problem-solving algorithms



Binary search algorithm

Divide and conquer: halves the search space with each iteration, reducing comparisons

```
function binarySearch(array, target):
    low = 0
    high = length(array) - 1

    while low <= high:
        mid = (low + high) / 2

        if array[mid] == target:
            return mid      // Target found at index mid

        else if array[mid] < target:
            low = mid + 1   // Search the right half

        else:
            high = mid - 1  // Search the left half

    return -1               // Target not found
```

- **Requires a sorted array** to function, offering faster searches than linear search
- Logarithmic Time: Time complexity is $O(\log n)$, making it optimal for large datasets

Complexity analysis of binary search

METROPOLITAN TIRANA UNIVERSITY

Time complexity logarithmically reduces the search space with each step

- Best Case: O(1) when the middle element is the target (first comparison)
- Average Case: O(log n) due to halving the search space each time
- Worst Case: O(log n) when the target is found in the last division

Proof sketch

- Start with n elements
- Divide by 2: each step halves search space ($n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4}$)
- Repeat: continue halving until one element remains
- Total steps: $O\ (\log n)$

Real world use cases of binary searches

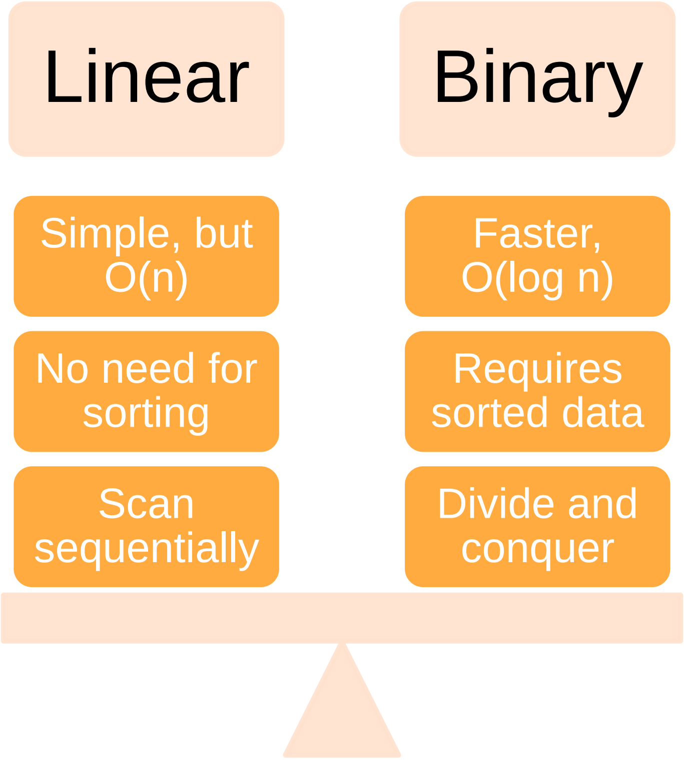
```java
public static int binarySearch(Employee[] employees, int targetId) {
        int low = 0;
        int high = employees.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;
            if (employees[mid].id == targetId) {
                return mid;     // Return index if found
            } else if (employees[mid].id < targetId) {
                low = mid + 1;  // Search the right half
            } else {
                high = mid - 1; // Search the left half
            }
        }
        return -1;              // Return -1 if not found
    }
```

Click here for full example

Searching for an employee in a sorted company directory

Summary of searching algorithms



Linear search scans each element,

while binary search halves the search

space, requiring sorted data

Sorting algorithms

Introduction to sorting

Sorting organizes data, making it easier to process and analyze

- Sorting algorithms
  - Slow
    - Bubble
    - Selection
    - Insertion
  - Fast
    - Quick
    - Merge

Bubble sort

Repeatedly swap adjacent elements to arrange in order

- Easy to understand and implement
- Inefficient for large datasets
- $O(n^2)$ time complexity in the worst case

6 5 3 1 8 7 2 4

```
function bubbleSort(array):
    n = length(array)
    for i from 0 to n-1:
        for j from 0 to n-i-2:
            if array[j] > array[j+1]:
                swap(array[j], array[j+1])
    return array
```

Selection sort

METROPOLITAN TIRANA UNIVERSITY

Builds a sorted array one element at a time by inserting each element in its correct position

- Efficient for small datasets
- Works well when the array is *nearly* sorted
- $O(n^2)$ complexity in the worst case

```
function insertionSort(array):
    n = length(array)
    for i from 1 to n-1:
        key = array[i]
        j = i - 1
        while j >= 0 and array[j] > key:
            array[j + 1] = array[j]
            j = j - 1
        array[j + 1] = key
    return array
```

6 5 3 1 8 7 2 4

Merge sort

Divide-and-conquer algorithm that splits the array and merges sorted subarrays

- Efficient with large datasets
- Stable sort (preserves equal elements' order)
- O($n \log n$) time complexity in all cases

6 5 3 1 8 7 2 4

Merge sort

```
function mergeSort(array):
    if length(array) <= 1:
        return array

    mid = length(array) / 2
    left = mergeSort(array[0...mid])
    right = mergeSort(array[mid+1...n])

    return merge(left, right)
```

- Recursively divides the array into two halves until subarrays of size 1 are reached

- Merges two sorted subarrays into a single sorted array
- Compares elements from both subarrays, appending the smaller one to the result

```
function merge(left, right):
    result = []
    while left and right are not empty:
        if left[0] < right[0]:
            append left[0] to result
            remove first element of left
        else:
            append right[0] to result
            remove first element of right
    append remaining elements of left and right to result
    return result
```

Quick sort

Efficient divide-and-conquer algorithm that partitions and recursively sorts subarrays

- Choose a pivot to partition the array into smaller and larger elements
- Efficient for large datasets with average time complexity of $O(n \log n)$
- In-place sorting, requiring no additional storage besides the input array

6 5 3 1 8 7 2 4

Quick sort

```
function quickSort(array, low, high):
    if low < high:
        pivotIndex = partition(array, low, high)
        quickSort(array, low, pivotIndex - 1)
        quickSort(array, pivotIndex + 1, high)
```

```
function partition(array, low, high):
    pivot = array[high]
    i = low - 1
    for j from low to high - 1:
        if array[j] < pivot:
            i = i + 1
            swap(array[i], array[j])
    swap(array[i + 1], array[high])
    return i + 1
```

- Recursively divides the array based on the pivot index
- Sorts subarrays on either side of the pivot

- Rearranges the array so smaller elements are left of the pivot
- Returns pivot index, placing the pivot in its sorted position

Comparison of sorting algorithms

Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
Quick sort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n^2)$	$O(\log n)$

When to use each sorting algorithm

METROPOLITAN TIRANA UNIVERSITY

Bubble sort

- Best for small datasets. Simplicity over efficiency

Selection sort

- Small datasets. Minimize memory and swaps

Insertion sort

- Performs well for nearly sorted datasets

Merge sort

- Perfect for large datasets and stable sorting. Excellent for external sorting

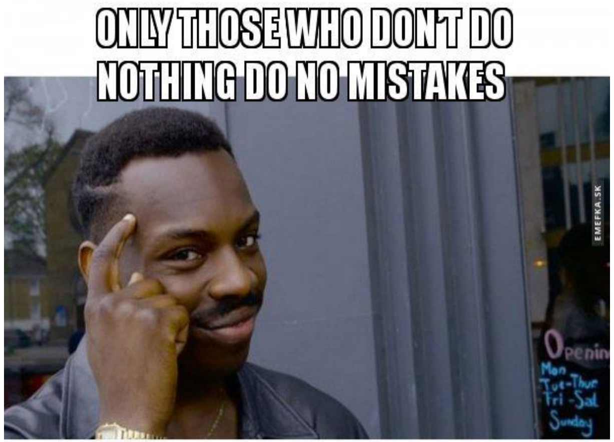
Quick sort

- Large datasets where speed is priority
- Ideal for in-place sorting

Common mistakes and optimization tips

- **Ignoring input size**: not all algorithms perform well on large datasets

- **Using linear search on sorted data**: always prefer binary search for sorted arrays



- **Neglecting in-place sorting**: when memory is limited, avoid merge sort

- **Skipping complexity analysis**: always consider time and space trade-offs before choosing an algorithm.

Java built-in searching and sorting

Sorting in java

Java provides optimized built-in sorting
for arrays and collections

- `Arrays.Sort(data)` sorts primitive and object arrays efficiently
- Algorithm used: uses dual-pivot Quicksort for primitives, Timsort for objects
- Performance: runs in $O(n \log n)$ on average, optimized for real-world cases
- Works with numbers, texts and custom objects with `comparable` interface

Sorting in java

Java provides optimized built-in sorting for arrays and collections

- `Collections.sort(list)` sorts `List<T>` elements in ascending order
- Algorithm Used: Uses Timsort, optimized for partially sorted data
- **Custom sorting:** use `Collections.sort(list, Comparator<T>)` for custom order. Ideal for sorting `ArrayLists`, `LinkedLists` and other `List` implementations

Sorting in java

```java
import java.util.Arrays;

public class ArraySortExample {
    public static void main(String[] args) {
        int[] numbers = {5, 2, 9, 1, 5, 6};
        Arrays.sort(numbers);
        System.out.println(Arrays.toString(numbers));
    }
}
```

```java
public class CustomSortExample {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
            new Employee("Alice", 50000),
            new Employee("Bob", 60000),
            new Employee("Charlie", 45000)
        );

        // Sorting by salary in descending order
        employees.sort(Comparator.comparingInt(e -> -e.salary));

        System.out.println(employees);
    }
}
```

```java
import java.util.*;

public class ListSortExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("John", "Alice", "Bob");
        Collections.sort(names);
        System.out.println(names);
    }
}
```

Sorting performance in Java

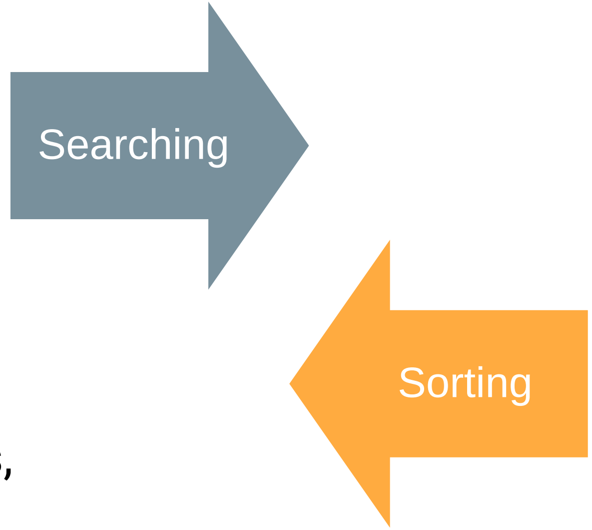
- Java optimizes sorting with efficient, adaptive algorithms
- Hybrid sorting: uses dual-pivot **quicksort** (primitives) and **timsort** (objects & lists)
- Adaptive behavior: timsort detects partially sorted data and optimizes performance
- Parallel execution `arrays.Parallelsort()` splits work across multiple CPU cores

Key Takeaways

METROPOLITAN TIRANA UNIVERSITY

Efficient searching & sorting are essential for scalable, real-world applications

- Search algorithms: linear search is simple, binary search is faster for sorted data
- Sorting techniques: Bubble, Insertion, Selection are simple but slow; MergeSort & QuickSort are efficient
- Real-world impact: used in databases, search engines, and big data processing



Helpful resources on arrays and linked lists

METROPOLITAN TIRANA UNIVERSITY

- 🔍 VisualAlgo: an interactive platform that visualizes searching and sorting techniques

- 🎥 Code with Conner tutorial on searching and sorting in Java

- 📚 Code Academy cheat sheet on searching and sorting algorithms

- 💻 GeekForGeeks on sorting algorithms and problems related to sorting

Quote of the Week

METROPOLITAN TIRANA UNIVERSITY

Life is a mix of searching and sorting:

find what matters, then put it in order