Advanced Topics on Algorithms

Search & Sorting

Author: Eng. Carlos Andrés Sierra, M.Sc.
cavirguezs@udistrital.edu.co

Full-time Adjunct Professor
Computer Engineering Program
School of Engineering
Universidad Distrital Francisco José de Caldas

2026-I

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Course Outline

Outline

Linear Search Algorithm and Analysis

Definition

Linear search is a sequential algorithm that checks each element in a data structure until the target element is found or all elements have been examined.

Algorithm $\text{LinearSearch}(A, x)$
// *Input: Array A[1..n], search value x*
// *Output: Index of x in A, or -1 if not found*
for $i = 1$ to $length(A)$ **do**
  if $A[i] = x$ **then**
    return i
  end if
end for
return -1 // *Element not found*

Linear Search Algorithm and Analysis

Definition

Linear search is a sequential algorithm that checks each element in a data structure until the target element is found or all elements have been examined.

```
Algorithm LinearSearch(A, x)
// Input: Array A[1..n], search value x
// Output: Index of x in A, or -1 if not found
for i = 1 to length(A) do
    if A[i] = x then
        return i
    end if
end for
return −1 // Element not found
```

Sequential Access Principles

Characteristics:

- **Order Independence:** Works on unsorted data
- **Memory Efficiency:** Constant space complexity $O(1)$
- **Simple Implementation:** Easy to understand and code
- **No Preprocessing:** Can search immediately

Advantages

- Works on any data structure
- No sorting requirement
- Simple to implement
- Guaranteed to find element (if exists)

Disadvantages

- Slow for large datasets
- Cannot skip elements
- Time increases linearly with size
- No early termination optimization

Sequential Access Principles

Characteristics:

- **Order Independence:** Works on unsorted data
- **Memory Efficiency:** Constant space complexity $O(1)$
- **Simple Implementation:** Easy to understand and code
- **No Preprocessing:** Can search immediately

Advantages

- Works on any data structure
- No sorting requirement
- Simple to implement
- Guaranteed to find element (if exists)

Disadvantages

- Slow for large datasets
- Cannot skip elements
- Time increases linearly with size
- No early termination optimization

Sequential Access Principles

Characteristics:

- **Order Independence:** Works on unsorted data
- **Memory Efficiency:** Constant space complexity $O(1)$
- **Simple Implementation:** Easy to understand and code
- **No Preprocessing:** Can search immediately

Advantages

- Works on any data structure
- No sorting requirement
- Simple to implement
- Guaranteed to find element (if exists)

Disadvantages

- Slow for large datasets
- Cannot skip elements
- Time increases linearly with size
- No early termination optimization

Study Case: Searching in Phone Directory

Problem

Find a person's phone number in a phone directory containing 1000 entries.

Example

Linear Search Approach:

Algorithm $\textsc{FindPhoneNumber}(directory, name)$
for $i = 1$ to $length(directory)$ **do**
  if $directory[i].name = name$ **then**
    return $directory[i].phone$
  end if
end for
return "Not Found"

Study Case: Searching in Phone Directory

Problem

Find a person's phone number in a phone directory containing 1000 entries.

Example

Linear Search Approach:

Algorithm $\textsc{FindPhoneNumber}(directory, name)$
for $i = 1$ to $length(directory)$ **do**
  if $directory[i].name = name$ **then**
    return $directory[i].phone$
  end if
end for
return "Not Found"

Binary Search Algorithm and Requirements

Definition

Binary search is a divide-and-conquer algorithm that finds an element in a sorted array by repeatedly dividing the search interval in half.

Prerequisites:

1. **Sorted Data:** Array must be sorted in ascending or descending order
2. **Random Access:** Ability to access any element directly (arrays, not linked lists)
3. **Comparison Operation:** Elements must be comparable

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Binary Search Algorithm and Requirements

Definition

Binary search is a divide-and-conquer algorithm that finds an element in a sorted array by repeatedly dividing the search interval in half.

Prerequisites:

1. **Sorted Data:** Array must be sorted in ascending or descending order
2. **Random Access:** Ability to access any element directly (arrays, not linked lists)
3. **Comparison Operation:** Elements must be comparable

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Binary Search Algorithm

Algorithm $\textsc{BinarySearch}(A, x, low, high)$
while $low \leq high$ **do**
  $mid \leftarrow \lfloor \frac{low+high}{2} \rfloor$
  if $A[mid] = x$ **then**
    return mid
  else if $A[mid] < x$ **then**
    $low \leftarrow mid + 1$
  else
    $high \leftarrow mid - 1$
  end if
end while
return -1

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Study Case: Library Book Location System

Problem

Locate a book in a library catalog system with 100,000 books organized by ISBN.

Study Case: Library Book Location System

Example

Algorithm $\textsc{FindBook}(catalog, target\_isbn)$
$low \leftarrow 1$
$high \leftarrow length(catalog)$
while $low \leq high$ **do**
  $mid \leftarrow \lfloor \frac{low+high}{2} \rfloor$
  if $catalog[mid].isbn = target\_isbn$ **then**
    return $catalog[mid]$ // *Return book information*
  else if $catalog[mid].isbn < target\_isbn$ **then**
    $low \leftarrow mid + 1$
  else
    $high \leftarrow mid - 1$
  end if
end while
return "Book not found"

Interpolation Search Introduction

Definition

Interpolation search improves upon binary search by making educated guesses about where the target element might be located, based on the values at the endpoints.

Key Concept

Instead of always choosing the middle element, interpolation search estimates the position using:

$$pos = low + \frac{(x - A[low])}{(A[high] - A[low])} \times (high - low)$$

Interpolation Search Introduction

Definition

Interpolation search improves upon binary search by making educated guesses about where the target element might be located, based on the values at the endpoints.

Key Concept

Instead of always choosing the middle element, interpolation search estimates the position using:

$$pos = low + \frac{(x - A[low])}{(A[high] - A[low])} \times (high - low)$$

Interpolation Search Algorithm

```
Algorithm INTERPOLATIONSEARCH(A, x, low, high)
while low ≤ high and x ≥ A[low] and x ≤ A[high] do
    pos ← low + (x − A[low]) / (A[high] − A[low]) × (high − low)
    if A[pos] = x then
        return pos
    else if A[pos] < x then
        low ← pos + 1
    else
        high ← pos − 1
    end if
end while
return −1
```

Search Algorithm Selection Criteria

Decision Framework

- **Data Size:** How many elements to search?
- **Data Organization:** Is the data sorted?
- **Search Frequency:** One-time or repeated searches?
- **Data Distribution:** Uniformly distributed values?
- **Memory Constraints:** Available space for preprocessing?

Algorithm Comparison:

Algorithm	Prerequisites	Best Use Case
Linear Search	None	Small, unsorted data
Binary Search	Sorted array	Large, sorted data
Interpolation	Sorted, uniform	Large, uniform data

Search Algorithm Selection Criteria

Decision Framework

- **Data Size:** How many elements to search?
- **Data Organization:** Is the data sorted?
- **Search Frequency:** One-time or repeated searches?
- **Data Distribution:** Uniformly distributed values?
- **Memory Constraints:** Available space for preprocessing?

Algorithm Comparison:

Algorithm	Prerequisites	Best Use Case
Linear Search	None	Small, unsorted data
Binary Search	Sorted array	Large, sorted data
Interpolation	Sorted, uniform	Large, uniform data

Study Case: Guess the Number (Large Space)

Problem

Guess a secret number between 1 and 1,000,000 with minimum guesses.

Outline

Bubble Sort Algorithm and Mechanism

Definition

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

```
Algorithm BubbleSort(A)
n ← length(A)
for i = 1 to n − 1 do
    for j = 1 to n − i do
        if A[j] > A[j + 1] then
            Swap(A[j], A[j + 1])
        end if
    end for
end for
```

Large elements "bubble up" to their correct position, just like air bubbles rising to the surface of water.

Bubble Sort Algorithm and Mechanism

Definition

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Algorithm $\textsc{BubbleSort}(A)$
$n \leftarrow length(A)$
for $i = 1$ to $n - 1$ **do**
  for $j = 1$ to $n - i$ **do**
    if $A[j] > A[j+1]$ **then**
      $\textsc{Swap}(A[j], A[j+1])$
    end if
  end for
end for

Large elements "bubble up" to their correct position, just like air bubbles rising to the surface of water.

Bubble Sort Algorithm and Mechanism

Definition

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Algorithm $\textsc{BubbleSort}(A)$
$n \leftarrow length(A)$
for $i = 1$ to $n - 1$ **do**
    for $j = 1$ to $n - i$ **do**
        if $A[j] > A[j+1]$ **then**
            $\textsc{Swap}(A[j], A[j+1])$
        end if
    end for
end for

Large elements "bubble up" to their correct position, just like air bubbles rising to the surface of water.

Adjacent Element Comparison Approach

Core Mechanism

Bubble sort works by making multiple passes through the array, comparing each pair of adjacent elements and swapping them if they're out of order.

Example

Step-by-step example: Sort [64, 34, 25, 12, 22, 11, 90]

Pass 1:

- Compare 64, 34 → Swap → [34, 64, 25, 12, 22, 11, 90]
- Compare 64, 25 → Swap → [34, 25, 64, 12, 22, 11, 90]
- Compare 64, 12 → Swap → [34, 25, 12, 64, 22, 11, 90]
- Compare 64, 22 → Swap → [34, 25, 12, 22, 64, 11, 90]
- Compare 64, 11 → Swap → [34, 25, 12, 22, 11, 64, 90]
- Compare 64, 90 → No swap → [34, 25, 12, 22, 11, 64, 90]

After Pass 1: Largest element (90) is in correct position!

Adjacent Element Comparison Approach

Core Mechanism

Bubble sort works by making multiple passes through the array, comparing each pair of adjacent elements and swapping them if they're out of order.

Example

Step-by-step example: Sort [64, 34, 25, 12, 22, 11, 90]

Pass 1:

- Compare 64, 34 → Swap → [34, 64, 25, 12, 22, 11, 90]
- Compare 64, 25 → Swap → [34, 25, 64, 12, 22, 11, 90]
- Compare 64, 12 → Swap → [34, 25, 12, 64, 22, 11, 90]
- Compare 64, 22 → Swap → [34, 25, 12, 22, 64, 11, 90]
- Compare 64, 11 → Swap → [34, 25, 12, 22, 11, 64, 90]
- Compare 64, 90 → No swap → [34, 25, 12, 22, 11, 64, 90]

After Pass 1: Largest element (90) is in correct position!

Study Case: Sorting Playing Cards

Real-World Application

You have a hand of playing cards and want to sort them by value. How would you naturally do this?

Selection Sort Algorithm and Approach

Definition

Selection sort sorts by repeatedly finding the minimum element from the unsorted portion and placing it at the beginning.

Key Characteristics:

- **Invariant:** After i iterations, first i elements are sorted
- **Selections:** Makes exactly $n-1$ swaps
- **Comparisons:** Always $\frac{n(n-1)}{2}$ comparisons

Selection Sort Algorithm and Approach

Definition

Selection sort sorts by repeatedly finding the minimum element from the unsorted portion and placing it at the beginning.

Key Characteristics:

- **Invariant:** After i iterations, first i elements are sorted
- **Selections:** Makes exactly $n-1$ swaps
- **Comparisons:** Always $\frac{n(n-1)}{2}$ comparisons

Selection Sort Algorithm

```
Algorithm SelectionSort(A)
n ← length(A)
for i = 1 to n − 1 do
    min_index ← i
    for j = i + 1 to n do
        if A[j] < A[min_index] then
            min_index ← j
        end if
    end for
    Swap(A[i], A[min_index])
end for
```

Minimum/Maximum Selection Strategy

Core Strategy

Selection sort maintains two regions:

- **Sorted region:** Elements already in final position
- **Unsorted region:** Elements yet to be processed

Example

Sorting [64, 25, 12, 22, 11]:

Pass	Array State	Action
Initial	[64, 25, 12, 22, 11]	Find min in [64,25,12,22,11]
1	[11, 25, 12, 22, 64]	Min=11, swap with 64
2	[11, 12, 25, 22, 64]	Min=12, swap with 25
3	[11, 12, 22, 25, 64]	Min=22, swap with 25
4	[11, 12, 22, 25, 64]	Min=25, no swap needed

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Minimum/Maximum Selection Strategy

Core Strategy

Selection sort maintains two regions:

- **Sorted region:** Elements already in final position
- **Unsorted region:** Elements yet to be processed

Example

Sorting [64, 25, 12, 22, 11]:

Pass	Array State	Action
Initial	[64, 25, 12, 22, 11]	Find min in [64,25,12,22,11]
1	[11, 25, 12, 22, 64]	Min=11, swap with 64
2	[11, 12, 25, 22, 64]	Min=12, swap with 25
3	[11, 12, 22, 25, 64]	Min=22, swap with 25
4	[11, 12, 22, 25, 64]	Min=25, no swap needed

Study Case: Student Grades Enrollment Schedule

Problem

A professor needs to organize student grades for enrollment priority. Students with higher GPAs get priority registration.

Example

Student Records:

Student Name	GPA
Alice	3.8
Bob	2.5
Charlie	3.9
Diana	2.1
Eve	3.7

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Study Case: Student Grades Enrollment Schedule [Solution]

Algorithm $\textsc{OrganizeByGPA}(students)$
for $i = 1$ to $length(students) - 1$ **do**
  $max\_gpa\_index \leftarrow i$
  for $j = i + 1$ to $length(students)$ **do**
    if $students[j].gpa > students[max\_gpa\_index].gpa$ **then**
      $max\_gpa\_index \leftarrow j$
    end if
  end for
  $\textsc{Swap}(students[i], students[max\_gpa\_index])$
end for

Result: Priority Order

Charlie (3.9) → Alice (3.8) → Eve (3.7) → Bob (2.5) → Diana (2.1)

Study Case: Student Grades Enrollment Schedule [Solution]

```
Algorithm OrganizeByGPA(students)
for i = 1 to length(students) − 1 do
    max_gpa_index ← i
    for j = i + 1 to length(students) do
        if students[j].gpa > students[max_gpa_index].gpa then
            max_gpa_index ← j
        end if
    end for
    Swap(students[i], students[max_gpa_index])
end for
```

Result: Priority Order

Charlie (3.9) → Alice (3.8) → Eve (3.7) → Bob (2.5) → Diana (2.1)

Insertion Sort Algorithm and Methodology

Definition

Insertion sort builds the final sorted array one element at a time by repeatedly taking an element from the unsorted portion and inserting it into its correct position in the sorted portion.

Algorithm $\textsc{InsertionSort}(A)$
for $i = 2$ to $length(A)$ **do**
  $key \leftarrow A[i]$
  $j \leftarrow i - 1$
  while $j \geq 1$ **and** $A[j] > key$ **do**
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j - 1$
  end while
  $A[j+1] \leftarrow key$
end for

Insertion Sort Algorithm and Methodology

Definition

Insertion sort builds the final sorted array one element at a time by repeatedly taking an element from the unsorted portion and inserting it into its correct position in the sorted portion.

Algorithm $\textsc{InsertionSort}(A)$
for $i = 2$ to $length(A)$ **do**
  $key \leftarrow A[i]$
  $j \leftarrow i - 1$
  while $j \geq 1$ **and** $A[j] > key$ **do**
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j - 1$
  end while
  $A[j+1] \leftarrow key$
end for

Incremental Sorting Approach I

Insertion sort works like organizing playing cards in your hand:

1. Start with first card (trivially sorted)
2. Pick next card from unsorted pile
3. Find correct position in sorted portion
4. Shift other cards as needed
5. Insert card in correct position
6. Repeat until all cards are sorted

Incremental Sorting Approach I

Example

Sorting [5, 2, 4, 6, 1, 3]:

Step	Array State	Action
Initial	[5, 2, 4, 6, 1, 3]	Start with first element
1	[2, 5, 4, 6, 1, 3]	Insert 2 before 5
2	[2, 4, 5, 6, 1, 3]	Insert 4 between 2 and 5
3	[2, 4, 5, 6, 1, 3]	6 already in position
4	[1, 2, 4, 5, 6, 3]	Insert 1 at beginning
5	[1, 2, 3, 4, 5, 6]	Insert 3 between 2 and 4

Study Case: Alphabetical Name Sorting

Problem

Sort a class roster alphabetically for easy lookup during attendance.

Example

Class Roster: [David, Alice, Charlie, Bob, Eve]

Insertion Sort Process:

Step	Roster State
Initial	[David, Alice, Charlie, Bob, Eve]
1	[Alice, David, Charlie, Bob, Eve]
2	[Alice, Charlie, David, Bob, Eve]
3	[Alice, Bob, Charlie, David, Eve]
4	[Alice, Bob, Charlie, David, Eve]

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Study Case: Alphabetical Name Sorting

Problem

Sort a class roster alphabetically for easy lookup during attendance.

Example

Class Roster: [David, Alice, Charlie, Bob, Eve]

Insertion Sort Process:

Step	**Roster State**
Initial	[David, Alice, Charlie, Bob, Eve]
1	[Alice, David, Charlie, Bob, Eve]
2	[Alice, Charlie, David, Bob, Eve]
3	[Alice, Bob, Charlie, David, Eve]
4	[Alice, Bob, Charlie, David, Eve]

Study Case: UD Students Code Sorting (Sort Battle)

Programming Challenge!

Universidad Distrital students are identified by codes like "20192578001". Sort student codes efficiently for registration processing.

Challenge Rules

- Input: 1000 student codes in random order
- Goal: Sort in ascending order
- Competition: Which algorithm performs best?
- Test different algorithms with same dataset

Example

Sample Student Codes: [20192578001, 20202589123, 20171098765]
Sorted Result: [20171098765, 20192578001, 20202589123]

Study Case: UD Students Code Sorting (Sort Battle)

Programming Challenge!

Universidad Distrital students are identified by codes like "20192578001". Sort student codes efficiently for registration processing.

Challenge Rules

- Input: 1000 student codes in random order
- Goal: Sort in ascending order
- Competition: Which algorithm performs best?
- Test different algorithms with same dataset

Example

Sample Student Codes: [20192578001, 20202589123, 20171098765]
Sorted Result: [20171098765, 20192578001, 20202589123]

Quick Sort Algorithm (Divide & Conquer Application) I

Definition

Quick sort is a divide-and-conquer algorithm that sorts by selecting a pivot element and partitioning the array around it, then recursively sorting the subarrays.

```
Algorithm QuickSort(A, low, high)
if low < high then
    pivot_index ← Partition(A, low, high)
    QuickSort(A, low, pivot_index − 1)
    QuickSort(A, pivot_index + 1, high)
end if
```

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Quick Sort Algorithm (Divide & Conquer Application) II

Algorithm $\textsc{Partition}(A, low, high)$
$pivot \leftarrow A[high]$ // *Choose last element as pivot*
$i \leftarrow low - 1$ // *Index of smaller element*
for $j = low$ to $high - 1$ **do**
  if $A[j] \leq pivot$ **then**
    $i \leftarrow i + 1$
    $\textsc{Swap}(A[i], A[j])$
  end if
end for
$\textsc{Swap}(A[i+1], A[high])$
return $i + 1$

Partition Strategy and Pivot Selection

Partitioning Process

The partition operation rearranges the array so that:

- Elements smaller than pivot are on the left
- Elements greater than pivot are on the right
- Pivot is in its final sorted position

Pivot Selection Strategies:

1. **First Element:** Simple but poor for sorted data
2. **Last Element:** Common choice, same issue
3. **Random Element:** Good average performance
4. **Median-of-Three:** Choose median of first, middle, last
5. **True Median:** Best but expensive to compute

Partition Strategy and Pivot Selection

Partitioning Process

The partition operation rearranges the array so that:

- Elements smaller than pivot are on the left
- Elements greater than pivot are on the right
- Pivot is in its final sorted position

Pivot Selection Strategies:

1. **First Element:** Simple but poor for sorted data
2. **Last Element:** Common choice, same issue
3. **Random Element:** Good average performance
4. **Median-of-Three:** Choose median of first, middle, last
5. **True Median:** Best but expensive to compute

Study Case: UD Students Code Sorting (The Final Challenge)

Ultimate Sorting Challenge!

Now we face the final boss: Sort 100,000 UD student codes using Quick Sort. Can it handle the massive dataset?

Challenge Specifications

- **Dataset Size:** 100,000 student codes
- **Format:** 11-digit codes (e.g., 20192578001)
- **Goal:** Sort in under 1 second
- **Memory Limit:** In-place sorting preferred

Study Case: UD Students Code Sorting (The Final Challenge)

Ultimate Sorting Challenge!

Now we face the final boss: Sort 100,000 UD student codes using Quick Sort. Can it handle the massive dataset?

Challenge Specifications

- **Dataset Size:** 100,000 student codes
- **Format:** 11-digit codes (e.g., 20192578001)
- **Goal:** Sort in under 1 second
- **Memory Limit:** In-place sorting preferred

Merge Sort Algorithm (Divide & Conquer Application) I

Definition

Merge sort is a stable, divide-and-conquer algorithm that divides the array into halves, recursively sorts them, and then merges the sorted halves.

Algorithm $\textsc{MergeSort}(A, left, right)$
if $left < right$ **then**
    $mid \leftarrow \lfloor \frac{left+right}{2} \rfloor$
    $\textsc{MergeSort}(A, left, mid)$
    $\textsc{MergeSort}(A, mid+1, right)$
    $\textsc{Merge}(A, left, mid, right)$
end if

Merge Sort Algorithm (Divide & Conquer Application) I

Definition

Merge sort is a stable, divide-and-conquer algorithm that divides the array into halves, recursively sorts them, and then merges the sorted halves.

Algorithm $\textsc{MergeSort}(A, left, right)$
if $left < right$ **then**
  $mid \leftarrow \lfloor \frac{left+right}{2} \rfloor$
  $\textsc{MergeSort}(A, left, mid)$
  $\textsc{MergeSort}(A, mid + 1, right)$
  $\textsc{Merge}(A, left, mid, right)$
end if

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Merge Sort Algorithm (Divide & Conquer Application) II

Algorithm $\textsc{Merge}(A, left, mid, right)$
Create temporary arrays $L[left..mid]$ and $R[mid+1..right]$
Copy data to temporary arrays
$i \leftarrow 0, j \leftarrow 0, k \leftarrow left$
while $i < len(L)$ **and** $j < len(R)$ **do**
    if $L[i] \leq R[j]$ **then**
        $A[k] \leftarrow L[i]; i \leftarrow i + 1$
    else
        $A[k] \leftarrow R[j]; j \leftarrow j + 1$
    end if
    $k \leftarrow k + 1$
end while
Copy remaining elements of L and R to A

Merge Strategy and Stability

Merge Strategy

The merge operation combines two sorted subarrays into one sorted array:

1. Compare the first elements of both subarrays
2. Select the smaller element and add to result
3. Move pointer in that subarray
4. Repeat until one subarray is exhausted
5. Copy remaining elements from other subarray

Stability Property

Merge sort is stable because when elements are equal, we always take from the left array first, preserving the original relative order.

Merge Strategy and Stability

Merge Strategy

The merge operation combines two sorted subarrays into one sorted array:

1. Compare the first elements of both subarrays
2. Select the smaller element and add to result
3. Move pointer in that subarray
4. Repeat until one subarray is exhausted
5. Copy remaining elements from other subarray

Stability Property

Merge sort is stable because when elements are equal, we always take from the left array first, preserving the original relative order.

Study Case: Real-time Sorting

Problem

A live streaming platform needs to sort viewer comments by timestamp in real-time while new comments continuously arrive.

Example

Scenario: Comments arrive every millisecond during a popular stream

Requirements:

- Sort existing comments while new ones arrive
- Maintain chronological order (stability important)
- Handle high volume (100,000+ comments/minute)
- Provide sorted output for display

When to Use Which Sorting Algorithm

Decision Matrix

Algorithm	Time Complexity	Stable	Best Use Case
Bubble Sort	$O(n^2)$	Yes	Educational/tiny datasets
Selection Sort	$O(n^2)$	No	Minimizing swaps
Insertion Sort	$O(n^2)$	Yes	Nearly sorted data
Quick Sort	$O(n \log n)$	No	General purpose
Merge Sort	$O(n \log n)$	Yes	Stability required

Choose Based on Requirements:

- **Stability needed:** Merge sort or Insertion sort
- **Memory limited:** Quick sort or Insertion sort
- **Partially sorted:** Insertion sort
- **Random data:** Quick sort
- **Guaranteed performance:** Merge sort

Performance Trade-offs Analysis

Comprehensive Performance Comparison

Let's analyze the trade-offs between different sorting algorithms across multiple dimensions.

Key Insights

- **Simplicity vs. Efficiency:** $O(n^2)$ algorithms are simpler but slower
- **Memory vs. Speed:** Faster algorithms often use more memory
- **Stability Cost:** Stable algorithms may be slightly slower
- **Adaptive Algorithms:** Some perform better on partially sorted data

Summary: Advanced Sorting & Searching I

Search Algorithms Summary

- **Linear Search:** Simple, works on any data, $O(n)$ time
- **Binary Search:** Efficient for sorted data, $O(\log n)$ time
- **Interpolation Search:** Best for uniform data, $O(\log \log n)$ average

Sorting Algorithms Summary

- **Simple Sorts ($O(n^2)$):** Bubble, Selection, Insertion - good for small data
- **Advanced Sorts ($O(n \log n)$):** Quick Sort (fast), Merge Sort (stable)
- **Algorithm Choice:** Depends on data size, memory, stability requirements

Summary: Advanced Sorting & Searching I

Search Algorithms Summary

- **Linear Search:** Simple, works on any data, $O(n)$ time
- **Binary Search:** Efficient for sorted data, $O(\log n)$ time
- **Interpolation Search:** Best for uniform data, $O(\log \log n)$ average

Sorting Algorithms Summary

- **Simple Sorts ($O(n^2)$):** Bubble, Selection, Insertion - good for small data
- **Advanced Sorts ($O(n \log n)$):** Quick Sort (fast), Merge Sort (stable)
- **Algorithm Choice:** Depends on data size, memory, stability requirements

Summary: Advanced Sorting & Searching II

Key Takeaways

- **Understand your data:** Size, order, distribution affect algorithm choice
- **Know your constraints:** Memory, time, stability requirements matter
- **Practice implementation:** Understanding helps with optimization
- **Benchmark performance:** Theory vs. practice can differ

Outline

Thanks!

Questions?

Repo: *https://github.com/EngAndres/ud-public/tree/main/courses/computer-sciences-i*

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS