LECTURE 29

Searching Algorithms

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

WHAT THIS LECTURE COVERS

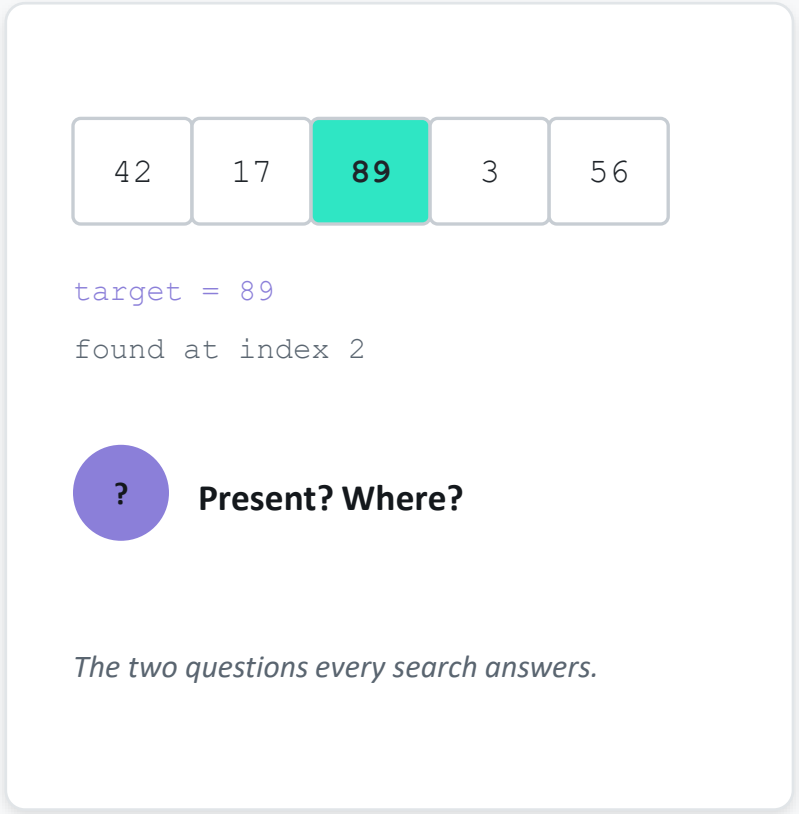
Agenda

1. The searching problem, formally
2. Linear search, and its complexity
3. Sentinel linear search — a small optimization on the same idea
4. Binary search — iterative and recursive — and its complexity
5. Binary search variants: first and last occurrence of a duplicate value
6. Common pitfalls, and a direct comparison of the two strategies

FOUNDATIONS

The Searching Problem

- Given n elements and a target value, searching answers: does the target exist, and if so, at what position?
- Every searching algorithm trades off how much work is needed per search against how much preparation (like sorting) is needed beforehand.
- This lecture compares the two fundamental strategies precisely, and shows why one needs the data sorted first.



The two questions every search answers.

STRATEGY 1

Linear Search

- Checks every element in order until it finds the target or exhausts the collection — no assumptions about data order required.

Searching for 56:

checked: 42, 17, 89, 3 → then found 56 at index 4 (5 comparisons)

O(n)
worst case: target absent or last

Best case (target first): O(1) • **Worst case (last / absent): O(n)**

IMPLEMENTATION

Linear Search — Code

searching_algorithms.cpp

```
int linearSearch(const vector<int>& data, int target)
{
    for (int i = 0; i < data.size(); i++) {
        if (data[i] == target) return i;
    }
    return -1;
}
```

- One comparison per element, checked in order.
- Returns the index on the first match.
- Returns -1 if the loop finishes without finding the target.

```
Linear search for 56 in unsorted data:
index 4
Linear search for 99 (absent):
index -1
```

A SMALL OPTIMIZATION

Sentinel Linear Search

- The plain loop does TWO comparisons per iteration: the bounds check (i < n) and the actual value check.
- Sentinel search plants the target at the very last slot, guaranteeing the loop always finds a match — only ONE comparison per iteration remains.
- Still O(n) worst case — same growth rate — but a real, measurable constant-factor speedup.

sentinel_linear_search.cpp

```
int last = data[n - 1];
data[n - 1] = target;   // plant sentinel

int i = 0;
while (data[i] != target) i++;
// only ONE comparison / iteration
```

Array with target 8 (the real last element) as sentinel-safe search:

½

the comparison count per iteration

STRATEGY 2 — REQUIRES SORTED DATA

Binary Search

- Requires sorted data — in exchange, eliminates HALF the remaining search space with every comparison.

Step 1: full range

mid=42. 56 > 42 → search RIGHT half

Step 2: right half

mid=71. 56 < 71 → search LEFT half

Step 3: found

mid=56. Found! (3 comparisons)

3 comparisons found 56 in a 7-element array — linear search would have needed 5 (checking 3, 8, 17, 42 first).

ITERATIVE IMPLEMENTATION

Binary Search — Code

searching_algorithms.cpp

```
int low = 0, high = sortedData.size() - 1;
while (low <= high) {
    int mid = low + (high - low) / 2;
    if (sortedData[mid] == target) return mid;
    else if (sortedData[mid] < target) low = mid + 1;
    else high = mid - 1;
}
return -1;
```

- `low + (high-low)/2` avoids integer overflow that `(low+high)/2` risks on large arrays.
- Each comparison halves the search space by moving low or high.
- A recursive version follows the exact same logic.

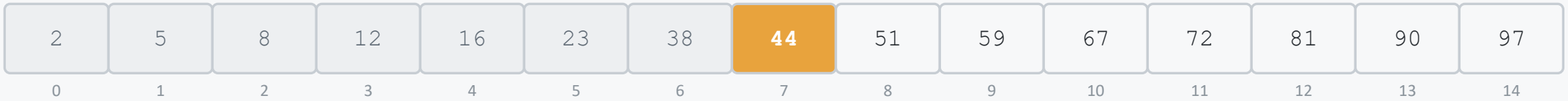
O(1)
best case: target is the middle element

O(log n)
worst case: target at an edge, or absent

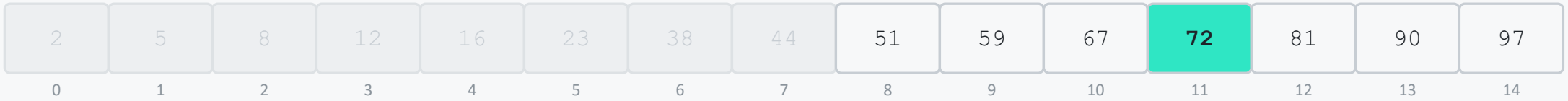
FEEL THE O(LOG N) PATTERN

Tracing Binary Search on 15 Elements

Step 1: low=0, high=14, mid=7 → 44. 72 > 44 → search right half



Step 2: low=8, high=14, mid=11 → 72. 72 == 72 → FOUND at index 11



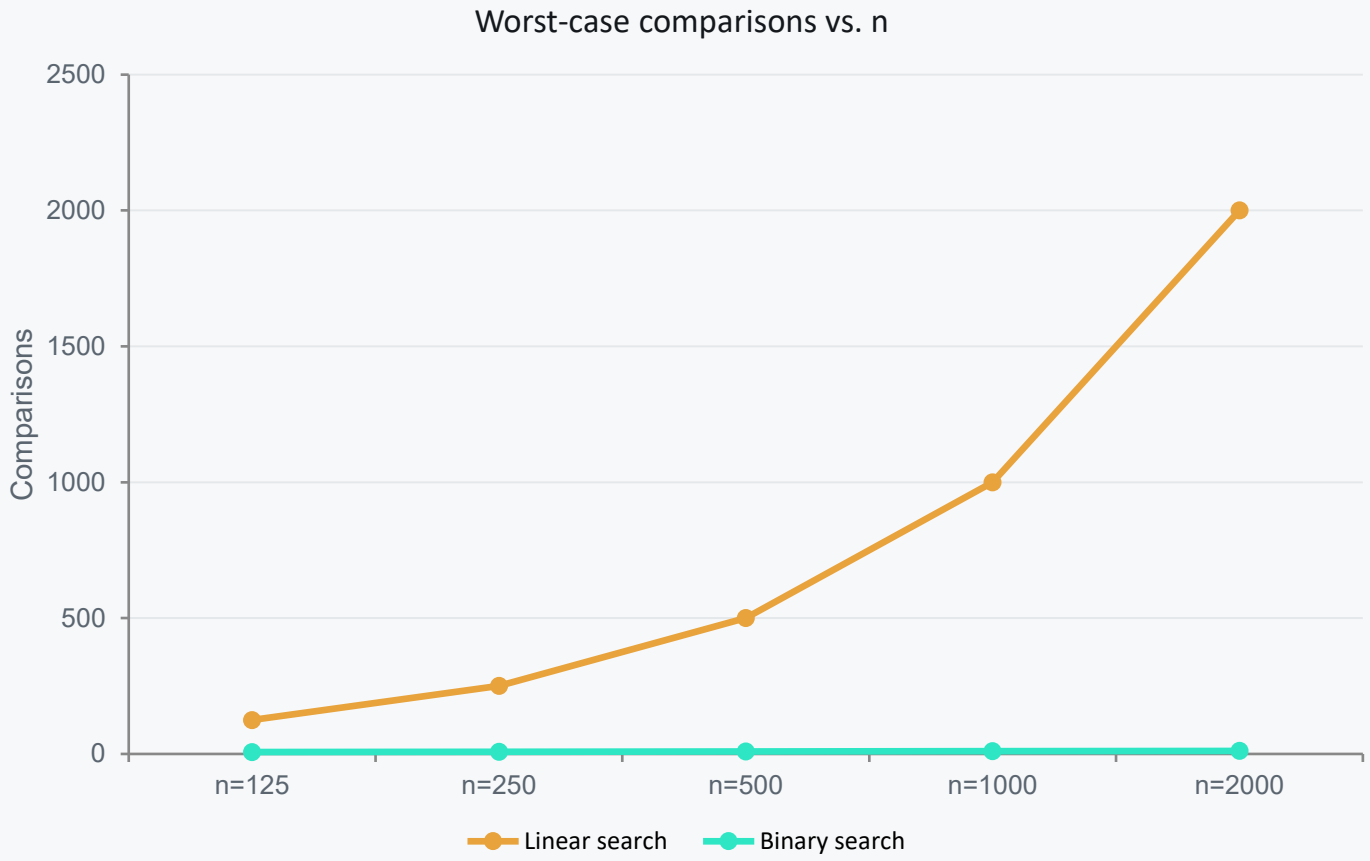
Two comparisons located the target among 15 elements. The gap only widens as n grows.

2

comparisons vs. 12 for linear search

A 2,000-ELEMENT WORKED EXAMPLE

Measuring the Advantage



2000 vs 11

comparisons: linear vs. binary at n=2000

- Linear search's worst case scales linearly with n.
- Binary search needs only ceil($\log_2 n$) comparisons — one more per doubling of n.
- This is exactly why O(log n) is called "logarithmic."

BINARY SEARCH VARIANTS

First and Last Occurrence

- On finding a match, don’t stop — record it, then keep narrowing toward an earlier (or later) one.

findFirst — the ONE line that differs

```
if (sortedData[mid] == target) {
    result = mid;
    high = mid - 1;  // keep searching LEFT
}
```

findLast — the ONE line that differs

```
if (sortedData[mid] == target) {
    result = mid;
    low = mid + 1;   // keep searching RIGHT
}
```

SHORT CODE, WELL-EARNED REPUTATION FOR BUGS

Common Pitfalls in Binary Search

Integer overflow

mid = (low+high)/2 can overflow on large arrays; use low + (high-low)/2 instead.

Off-by-one loop condition

while (low < high) silently skips the single-element case; use low <= high.

Unenforced sorted precondition

Binary search doesn’t fail loudly on unsorted data — it just returns wrong answers.

Infinite loops

low = mid instead of low = mid + 1 can leave the search space unchanged forever.

Jon Bentley’s Programming Pearls noted most published implementations, across decades, contained bugs.

NOT JUST SPEED — SHAPE

Linear vs. Binary Search

	Linear Search	Binary Search
Requires sorted data?	No	Yes
Worst-case complexity	O(n)	O(log n)
Works on a linked list?	Yes (sequential access)	Poorly (needs random access)

- A one-time search of unsorted data rarely justifies sorting first (which itself costs O(n log n)) plus a binary search.
- Searching the same data repeatedly: sorting once and reusing binary search every time wins decisively.

CLOSING

Key Takeaways

- Linear search makes no assumptions about order (O(n) worst case); sentinel search halves the per-iteration comparisons.
- Binary search needs sorted, random-access data in exchange for O(log n) — confirmed with a 15-element trace and a 2,000-element count.
- findFirst / findLast generalize binary search to locate the boundaries of a run of duplicates.
- Short code hides real pitfalls: midpoint overflow, off-by-one loops, unenforced sort order, infinite loops.
- The right choice depends on how the data is organized and how often it’s searched.