Searching and Sorting

Fundamental Algorithms

Prof. Antonio Khalil Moretti

SCIS 313: Data Structures and Algorithm Analysis

Spelman College

Today's Topics

Two fundamental problems in computer science:

1. **Searching:** Find an element in a collection
2. **Sorting:** Arrange elements in order

Why these matter:

- Used everywhere in practice
- Great examples for algorithm analysis
- Demonstrate recursion and divide-and-conquer
- Show importance of choosing the right algorithm

Today's focus:

- Binary search (the most important searching algorithm)
- One simple sort (for comparison)
- Merge sort and Quick sort (the practical algorithms)

The Searching Problem

Problem: Given an array and a target value, find the target's index (or -1 if not found)

Example:

Array: `[3, 7, 12, 18, 23, 31, 45]`

Target: `18`

Answer: Index 3

Two approaches:

- **Linear Search:** Check each element sequentially - $O(n)$
- **Binary Search:** If array is sorted, divide and conquer - $O(\log n)$

Key insight: Binary search requires a sorted array!

Linear Search - The Simple Approach

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

Analysis:

- **Best case:** Target is first element - $O(1)$
- **Worst case:** Target is last or not present - $O(n)$
- **Average case:** Check half the array - $O(n)$

When to use: Unsorted arrays, small arrays, or when simplicity matters

Binary Search - The Smart Approach

Works only on sorted arrays

Key idea: If the array is sorted, we can eliminate half the search space each step!

Algorithm:

1. Look at the middle element
2. If it's the target, done!
3. If target is smaller, search the left half
4. If target is larger, search the right half
5. Repeat until found or search space is empty

Example: Search for 18 in `[3, 7, 12, 18, 23, 31, 45]`

- Mid = 18 $\rightarrow$ Found it in 1 step!

Example: Search for 31

- Mid = 18, target $>$ 18 $\rightarrow$ search right half `[23, 31, 45]`
- Mid = 31 $\rightarrow$ Found it in 2 steps!

Binary Search - Iterative Implementation

```
int binarySearch(int arr[], int n, int target
    ) {
    int left = 0;
    int right = n - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;
        }
        else if (arr[mid] < target) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }
    return -1;
}
```

Key points:

- Loop while search space exists
- Check middle
- Narrow search space
- Return -1 if not found

Overflow trick:

`mid = left + (right - left) / 2`
safer than
`(left + right) / 2`

Binary Search - Recursive Implementation

```
int binarySearchRec(int arr[], int left,
                    int right, int target) {
    if (left > right) {
        return -1;  // not found
    }
    int mid = left + (right - left) / 2;
    if (arr[mid] == target) {
        return mid;  // found!
    }
    else if (arr[mid] < target) {
        return binarySearchRec(arr, mid + 1,
                               right, target);
    }
    else {
        return binarySearchRec(arr, left,
                               mid - 1, target);
    }
}
```

Base cases:

- `left > right` → not found
- `arr[mid] == target` → found!

Recursive case:
Search left or right half

Call with:
`binarySearchRec(arr, 0, n-1, target)`

Binary Search - Complexity Analysis

Recurrence relation:

$$T(1) = c$$
$$T(n) = T(n/2) + c$$

Solving by expansion:

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= [T(n/4) + c] + c = T(n/4) + 2c \\ &= T(n/8) + 3c \\ &= T(n/2^k) + kc \end{aligned}$$

Base case when $n/2^k = 1$, so $k = \log_2 n$

$$T(n) = T(1) + c \log n = \Theta(\log n)$$

Space complexity: $O(\log n)$ for recursive, $O(1)$ for iterative

Why Binary Search is Remarkable

The power of logarithmic growth:

Array Size	Linear Search	Binary Search
100	100	7
1,000	1,000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30

Example: Searching a billion elements

- Linear search: up to 1 billion comparisons
- Binary search: at most 30 comparisons

This is why sorting is worth it! If you search many times, paying $O(n \log n)$ to sort once gives you $O(\log n)$ searches forever.

Binary Search Variations

Classic binary search finds any **occurrence of the target.**

Useful variations:

- **Find first occurrence:** In array with duplicates
  Example: `[1, 2, 2, 2, 5, 7]` find first 2 $\rightarrow$ index 1
- **Find last occurrence:** In array with duplicates
  Example: `[1, 2, 2, 2, 5, 7]` find last 2 $\rightarrow$ index 3
- **Find insertion point:** Where to insert to keep sorted
  Example: `[1, 3, 5, 7]` insert 4 $\rightarrow$ between index 1 and 2
- **Find closest element:** Element nearest to target

All still $O(\log n)$ with slight modifications to the basic algorithm!

Why We Need Sorting

Binary search requires sorted data. So how do we sort?

The Sorting Problem:

Given an array, rearrange elements in non-decreasing order.

Example:

Input: `[23, 3, 45, 12, 7, 18, 31]`

Output: `[3, 7, 12, 18, 23, 31, 45]`

Many sorting algorithms exist:

- Simple but slow: Insertion, Selection, Bubble - $O(n^2)$
- Fast but complex: Merge, Quick, Heap - $O(n \log n)$
- Special cases: Counting, Radix - $O(n)$ for specific data

Today: One simple sort, two fast sorts

Insertion Sort - The Simple One

Like sorting playing cards in your hand

Idea: Build sorted array one element at a time by inserting each new element into its correct position.

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        // Shift elements greater than key to the right
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;  // Insert key in correct position
    }
}
```

Insertion Sort - Example Trace

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

Step	Array
Initial	`[5, 2, 4, 1, 3]`
i=1 (insert 2)	`[2, 5, 4, 1, 3]`
i=2 (insert 4)	`[2, 4, 5, 1, 3]`
i=3 (insert 1)	`[1, 2, 4, 5, 3]`
i=4 (insert 3)	`[1, 2, 3, 4, 5]`

Analysis:

- **Best case:** Already sorted - $O(n)$
- **Worst case:** Reverse sorted - $O(n^2)$
- **Average case:** $O(n^2)$

When to use: Small arrays (n $<$ 10-20), nearly sorted data, online sorting

Merge Sort - Divide and Conquer

The first $O(n \log n)$ algorithm

Strategy:

1. **Divide:** Split array into two halves
2. **Conquer:** Recursively sort each half
3. **Combine:** Merge the two sorted halves

Example: `[38, 27, 43, 3, 9, 82, 10]`

1. Split: `[38, 27, 43, 3]` and `[9, 82, 10]`
2. Split again: `[38, 27]`, `[43, 3]`, `[9, 82]`, `[10]`
3. Split again: `[38]`, `[27]`, `[43]`, `[3]`, etc.
4. Merge back up: `[27, 38]`, `[3, 43]`, `[9, 82]`, `[10]`
5. Merge: `[3, 27, 38, 43]`, `[9, 10, 82]`
6. Final merge: `[3, 9, 10, 27, 38, 43, 82]`

The Key: Merging Two Sorted Arrays

Given two sorted arrays, merge them into one sorted array in $O(n)$ time.

```
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];  // temporary arrays
    // Copy data to temp arrays
    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];
    // Merge back into arr[left..right]
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];  // copy remaining
    while (j < n2) arr[k++] = R[j++];
}
```

Merge Sort - Complete Implementation

```
void mergeSort(int arr[], int left, int right) {
    if (left >= right) {
        return;  // base case: single element or empty
    }

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

    mergeSort(arr, left, mid);      // sort left half
    mergeSort(arr, mid + 1, right); // sort right half
    merge(arr, left, mid, right);   // merge them
}

// Call with: mergeSort(arr, 0, n-1)
```

This is the recursive structure we analyzed in the last lesson!

Merge Sort - Complexity Analysis

Recurrence relation:

$$T(1) = c$$
$$T(n) = 2T(n/2) + cn$$

Using Master Theorem:

- $a = 2$, $b = 2$, $f(n) = cn$
- $n^{\log_b a} = n^{\log_2 2} = n$
- $f(n) = cn = \Theta(n)$
- Case 2: $T(n) = \Theta(n \log n)$

Space complexity: $O(n)$ for the temporary arrays

Properties:

- Always $O(n \log n)$ - no worst case!
- Stable (preserves relative order of equal elements)
- Requires extra space

Quick Sort - Another Divide and Conquer

The most-used sorting algorithm in practice

Strategy:

1. **Partition:** Choose a pivot, rearrange so all elements $<$ pivot are on left, all $>$ pivot are on right
2. **Conquer:** Recursively sort left and right subarrays
3. **Combine:** Nothing! Already in place.

Example: `[10, 7, 8, 9, 1, 5]` with pivot $= 5$

1. Partition around 5: `[1, 5, 8, 9, 10, 7]`
2. Left of 5: `[1]` - sorted
3. Right of 5: `[8, 9, 10, 7]` - recurse
4. Partition around 7: `[7, 9, 10, 8]`
5. Continue until done: `[1, 5, 7, 8, 9, 10]`

Key difference from Merge Sort: Work is in partitioning, not merging

Quick Sort - The Partition Function

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // choose last element as pivot
    int i = low - 1;  // index of smaller element

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

    swap(arr[i + 1], arr[high]);  // put pivot in correct place
    return i + 1;  // return pivot position
}
```

Partition is $O(n)$: Single pass through the array

Quick Sort - Complete Implementation

```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partition the array
        int pi = partition(arr, low, high);

        // Recursively sort before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Call with: quickSort(arr, 0, n-1)
```

Notice: No merge step! The partitioning does all the work.

Quick Sort - Complexity Analysis

Best/Average case: Pivot divides array evenly

$$T(n) = 2T(n/2) + cn = \Theta(n \log n)$$

Worst case: Pivot is always smallest or largest (already sorted!)

$$T(n) = T(n-1) + cn = \Theta(n^2)$$

Why use Quick Sort despite worst case?

- Average case is $O(n \log n)$ and very fast in practice
- Worst case is rare with good pivot selection
- In-place (no extra space like merge sort)
- Cache-friendly (good locality of reference)

Improvements: Random pivot, median-of-three, hybrid with insertion sort for small subarrays

Sorting Algorithms Comparison

Algorithm	Best	Average	Worst	Space
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

Additional properties:

- **Stable:** Insertion Sort, Merge Sort
- **Unstable:** Quick Sort (standard implementation)
- **In-place:** Insertion Sort, Quick Sort
- **Not in-place:** Merge Sort

Stability: Preserves relative order of equal elements

Example: Sorting students by grade, then by name - stability preserves name order within same grade

Which Sorting Algorithm to Use?

Insertion Sort:

- Small arrays (n < 20)
- Nearly sorted data
- Online sorting
- Simplicity matters

Merge Sort:

- Guaranteed $O(n \log n)$
- Need stable sort
- Extra memory OK
- Sorting linked lists

Quick Sort:

- General purpose (most common)
- Memory limited (in-place)
- Average case matters
- Cache performance matters

In practice:

Libraries use hybrids:

- C++: introsort
- Python: timsort

The Sorting Lower Bound

Can we do better than $O(n \log n)$?

Theorem: Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Why?

- There are $n!$ possible permutations of n elements
- Each comparison gives binary (yes/no) information
- Need enough comparisons to distinguish between all $n!$ permutations
- $\log_2(n!) = \Theta(n \log n)$ by Stirling's approximation

What this means:

- Merge sort and quick sort are *asymptotically optimal*
- Can't do better with comparisons alone
- Can do better with non-comparison sorts (counting, radix) for special cases

We won't prove this formally, but it's an important result!

Real-World Sorting

C++ `std::sort`**:**

- **Introsort** (introspective)
- Starts with quick sort
- Switches to heap sort if too deep
- Insertion for small arrays
- Guaranteed $O(n \log n)$

Python `sorted()`**:**

- **Timsort** (Tim Peters, 2002)
- Hybrid merge + insertion
- Exploits existing order
- Stable $O(n \log n)$

Takeaway: Production code uses sophisticated hybrids, but they're all based on the algorithms we studied!

Summary

Searching:

- Linear search: $O(n)$ - simple, works on unsorted data
- Binary search: $O(\log n)$ - requires sorted data, incredibly fast
- Binary search shows the power of divide-and-conquer

Sorting:

- Insertion sort: $O(n^2)$ - simple, good for small/nearly-sorted arrays
- Merge sort: $O(n \log n)$ always - stable, needs extra space
- Quick sort: $O(n \log n)$ average - in-place, most used in practice
- $\Omega(n \log n)$ is the lower bound for comparison sorts

Key insights:

- Recursion and divide-and-conquer are powerful paradigms
- The right algorithm makes a huge difference
- Trade-offs matter: time vs space, worst vs average, stable vs unstable

What's Next

Today we covered:

- Binary search - the foundation of efficient searching
- Three sorting algorithms with different trade-offs
- Analysis using recurrence relations

Next class: Data Structures Begin!

- Arrays and amortized analysis
- Dynamic arrays (vectors)
- Why `push_back` is $O(1)$ amortized
- Introduction to amortized analysis techniques

Homework: Implementing and analyzing sorting algorithms (posted on Canvas)

Make sure you understand binary search - it's one of the most important algorithms in computer science!