Searching Algorithms

Find Elements Efficiently Within Collections

Minseok Jeon
DGIST

November 2, 2025

Table of Contents

Introduction

What is Searching?

Searching: Finding a specific element in a collection

Why Searching Matters:

- Fundamental operation in computer science
- Used in databases, search engines, file systems
- Performance-critical in many applications
- Basis for more complex algorithms

Key Questions:

- Is the data sorted?
- How large is the dataset?
- How frequently do we search?
- Can we preprocess the data?

Classification of Search Algorithms

Based on Data Structure:

- **Array-based**: Linear, Binary, Interpolation, Jump
- **Tree-based**: BST, AVL, Red-Black Trees
- **Hash-based**: Hash Tables, Hash Maps

Based on Preconditions:

- **Unsorted data**: Linear Search, Hash Tables
- **Sorted data**: Binary Search, Interpolation, Jump
- **Special structures**: Tree Search

Based on Complexity:

- $O(1)$: Hash Table (average)
- $O(\log \log n)$: Interpolation (average, uniform data)
- $O(\log n)$: Binary Search, BST
- $O(\sqrt{n})$: Jump Search
- $O(n)$: Linear Search

Linear vs Binary Search

Linear Search: Overview

Sequential Search Through Elements

Algorithm:

1. Start from first element
2. Check each element sequentially
3. Return index when found
4. Return -1 if not found

Characteristics:

- **Time**: $O(n)$
- **Space**: $O(1)$
- **Requirement**: None (works on unsorted data)

When to Use:

- Small datasets ($n < 100$)
- Unsorted data
- Simple implementation needed

Linear Search: Implementation

```python
def linear_search(arr, target):
    """Search for target in array"""
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return index
    return -1  # Not found

# Example
arr = [64, 34, 25, 12, 22, 11, 90]
result = linear_search(arr, 22)  # Returns 4
```

Analysis:

- **Best case**: $O(1)$ - element at first position
- **Average case**: $O(n/2) = O(n)$ - element in middle
- **Worst case**: $O(n)$ - element at end or not present

Binary Search: Overview

Divide and Conquer on Sorted Array

Algorithm:

1. Find middle element
2. Compare with target
3. If equal: found!
4. If target $<$ middle: search left half
5. If target $>$ middle: search right half
6. Repeat until found or exhausted

Characteristics:

- **Time**: $O(\log n)$
- **Space**: $O(1)$ iterative, $O(\log n)$ recursive
- **Requirement**: Array must be sorted

Key Advantage:

- Eliminates half of search space in each step

Binary Search: Iterative Implementation

```python
def binary_search(arr, target):
    """Search for target in sorted array"""
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2  # Avoid overflow

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1  # Search right half
        else:
            right = mid - 1  # Search left half

    return -1  # Not found

# Example (array must be sorted!)
arr = [11, 12, 22, 25, 34, 64, 90]
result = binary_search(arr, 22)  # Returns 2
```

Note: `mid = left + (right - left) // 2` avoids integer overflow

Binary Search: Recursive Implementation

```python
def binary_search_recursive(arr, target, left, right):
    if left > right:
        return -1

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

    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, target,
                                       mid + 1, right)
    else:
        return binary_search_recursive(arr, target,
                                       left, mid - 1)

# Usage
arr = [11, 12, 22, 25, 34, 64, 90]
```

Binary Search: Visual Example

Search for 22 in [11, 12, 22, 25, 34, 64, 90]

Step 1: left=0, right=6, mid=3

[11, 12, 22, 25, 34, 64, 90]

arr[3]=25 $>$ 22, search left half

Step 2: left=0, right=2, mid=1

[11, 12, 22]

arr[1]=12 $<$ 22, search right half

Step 3: left=2, right=2, mid=2

[22]

arr[2]=22, found at index 2!

Result: Only 3 comparisons for 7 elements

Linear vs Binary Search: Comparison

Feature	Linear	Binary
Time Complexity	$O(n)$	$O(\log n)$
Space Complexity	$O(1)$	$O(1)$ iterative
Sorted Required	No	Yes
Best For	Small/unsorted	Large/sorted
Avg comparisons (n=1000)	500	10
Implementation	Simple	Moderate
Linked List Support	Yes	No

Performance Comparison:

- $n = 10$: Linear=5, Binary=3
- $n = 100$: Linear=50, Binary=7
- $n = 1,000,000$: Linear=500,000, Binary=20

Binary Search on Answers (Parametric)

Concept: Search the Answer Space

Binary Search on Answers (Parametric Search)

Idea:

- Binary search on the **solution space**, not the array
- Find minimum/maximum value satisfying a condition
- Requires monotonic property

Monotonic Property:

- If value x works, then all values $> x$ also work (or vice versa)
- Creates a boundary: [false, false, ..., true, true, ...]
- Binary search finds the boundary

Common Patterns:

- **Minimize maximum**: Find smallest value where all constraints satisfied
- **Maximize minimum**: Find largest value where all constraints satisfied
- **Find threshold**: Find boundary where condition changes

Template: Binary Search on Answers

```python
def binary_search_answer(condition_func, low, high):
    """
    Find minimum value in [low, high] satisfying condition
    """
    result = -1

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

        if condition_func(mid):
            result = mid
            high = mid - 1  # Try to find smaller
        else:
            low = mid + 1

    return result
```

Example 1: Integer Square Root

```python
def sqrt(x):
    """Find integer square root of x"""
    if x < 2:
        return x

    left, right = 1, x // 2

    while left <= right:
        mid = left + (right - left) // 2
        square = mid * mid

        if square == x:
            return mid
        elif square < x:
            left = mid + 1
        else:
            right = mid - 1

    return right  # Largest integer whose square <= x

# Example: sqrt(8) = 2 (since 2^2 = 4 <= 8 < 3^2 = 9)
print(sqrt(8))   # Output: 2
print(sqrt(16))  # Output: 4
print(sqrt(17))  # Output: 4
```

Analysis: Search space is [1, x/2], binary search in $O(\log x)$

Example 2: Minimum Ship Capacity

Problem: Ship packages in D days, find minimum capacity

```python
def ship_within_days(weights, days):
    """Find minimum ship capacity"""
    def can_ship(capacity):
        """Check if we can ship with given capacity"""
        days_needed = 1
        current_load = 0

        for weight in weights:
            if current_load + weight > capacity:
                days_needed += 1
                current_load = weight
            else:
                current_load += weight

        return days_needed <= days

    # Binary search on capacity
    left = max(weights)  # Min: heaviest package
    right = sum(weights)  # Max: all at once

    while left < right:
        mid = left + (right - left) // 2

        if can_ship(mid):
            right = mid  # Try smaller capacity
        else:
            left = mid + 1

    return left
```

Ship Capacity Example

Input: weights = [1,2,3,4,5,6,7,8,9,10], days = 5

Search Space:

- Minimum capacity: max(weights) = 10
- Maximum capacity: sum(weights) = 55
- Search range: [10, 55]

Binary Search Process:

- Try capacity=32: Can ship in 3 days $\rightarrow$ too large
- Try capacity=21: Can ship in 4 days $\rightarrow$ too large
- Try capacity=15: Can ship in 5 days $\rightarrow$ works!
- Try capacity=12: Cannot ship in 5 days $\rightarrow$ too small

Answer: 15
Distribution: [1,2,3,4,5], [6,7], [8], [9], [10]

Example 3: Koko Eating Bananas

Problem: Finish all banana piles in H hours, minimize eating speed

```python
def min_eating_speed(piles, h):
    """Find minimum eating speed"""
    def can_finish(speed):
        """Check if can finish with given speed"""
        hours = 0
        for pile in piles:
            hours += (pile + speed - 1) // speed  # Ceiling
        return hours <= h

    left, right = 1, max(piles)

    while left < right:
        mid = left + (right - left) // 2

        if can_finish(mid):
            right = mid
        else:
            left = mid + 1

    return left

# Example: piles=[3,6,7,11], h=8
# Answer: 4 (eat 4 bananas/hour)
```

Search Trees and Hashing

Binary Search Tree (BST)

Tree-Based Search Structure

Properties:

- Each node has left (smaller) and right (larger) children
- Left subtree $<$ node $<$ right subtree
- Enables $O(\log n)$ search on average

Characteristics:

- **Search**: $O(\log n)$ average, $O(n)$ worst
- **Insert/Delete**: $O(\log n)$ average
- **Space**: $O(n)$

Advantages:

- Dynamic: supports insertions/deletions
- Maintains sorted order
- Range queries efficient

BST Search Implementation

```python
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def search_bst(root, target):
    """Search in BST - Recursive"""
    if not root or root.val == target:
        return root

    if target < root.val:
        return search_bst(root.left, target)
    else:
        return search_bst(root.right, target)

def search_bst_iterative(root, target):
```

Hash Table Search

Constant-Time Lookup via Hashing

Concept:

- Map keys to array indices using hash function
- Direct access to value via computed index
- Handle collisions (chaining or open addressing)

Characteristics:

- **Search**: $O(1)$ average, $O(n)$ worst
- **Insert/Delete**: $O(1)$ average
- **Space**: $O(n)$

Advantages:

- Fastest average-case search: $O(1)$
- Simple interface (key-value pairs)
- Good for large datasets with unique keys

Hash Table Implementation

```python
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)
                return
        self.table[index].append((key, value))

    def search(self, key):
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        return None

# Python's built-in dict is a hash table
hash_table = {"apple": 5, "banana": 3}
print(hash_table["apple"])  # O(1) lookup
```

Comparison: Array, BST, Hash Table

Structure	Average	Worst	Sorted
Sorted Array + Binary	$O(\log n)$	$O(\log n)$	Yes
BST	$O(\log n)$	$O(n)$	Yes
Balanced BST	$O(\log n)$	$O(\log n)$	Yes
Hash Table	$O(1)$	$O(n)$	No

When to Use:

- **Sorted Array**: Static data, range queries
- **BST**: Dynamic data, need sorted order
- **Balanced BST**: Guaranteed performance, sorted
- **Hash Table**: Fast lookup, no order needed

Interpolation and Exponential Search

Interpolation Search

Position Estimation Based on Value

Idea:

- Like looking up a name in phone book
- Estimate position based on value distribution
- Works best with uniformly distributed data

Position Formula:

$$pos = left + \frac{(target - arr[left])}{(arr[right] - arr[left])} \times (right - left)$$

Characteristics:

- **Time**: $O(\log \log n)$ average, $O(n)$ worst
- **Requirement**: Sorted + uniformly distributed
- **Better than binary**: For uniform data

Example: Array [10, 20, 30, 40, 50, 60, 70, 80, 90], target=70
Estimate: $pos \approx 0 + \frac{70-10}{90-10} \times 8 = 6$ (directly finds it!)

Interpolation Search Implementation

```python
def interpolation_search(arr, target):
    """Search in uniformly distributed sorted array"""
    left, right = 0, len(arr) - 1

    while left <= right and target >= arr[left] and target <= arr[right]:
        if left == right:
            if arr[left] == target:
                return left
            return -1

        # Estimate position
        pos = left + int(((target - arr[left]) /
                          (arr[right] - arr[left])) * (right - left))

        if arr[pos] == target:
            return pos
        elif arr[pos] < target:
            left = pos + 1
        else:
            right = pos - 1

    return -1

# Example: uniformly distributed data
arr = [10, 20, 30, 40, 50, 60, 70, 80, 90]
result = interpolation_search(arr, 70)  # Returns 6
```

Exponential Search

Find Range, Then Binary Search

Algorithm:

1. Check if element at position 0
2. Find range $[2^{k-1}, 2^k]$ where element exists
3. Perform binary search in that range

Characteristics:

- **Time**: $O(\log n)$
- **Best for**: Element near beginning, unbounded arrays

Why Useful:

- Don't need to know array size
- Better than binary search when target near start
- Good for infinite/unbounded lists

Example: Search for 10 in sorted array

Exponential Search Implementation

```python
def exponential_search(arr, target):
    """Search in sorted array, efficient when target near start"""
    n = len(arr)

    # If target at first position
    if arr[0] == target:
        return 0

    # Find range for binary search
    i = 1
    while i < n and arr[i] <= target:
        i *= 2

    # Binary search in range [i//2, min(i, n-1)]
    return binary_search_range(arr, target, i // 2, min(i, n - 1))

def binary_search_range(arr, target, left, right):
    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1
```

Jump Search

Jump Fixed Steps, Then Linear

Algorithm:

1. Jump ahead by $\sqrt{n}$ steps
2. Find block containing target
3. Linear search within block

Characteristics:

- **Time**: $O(\sqrt{n})$
- **Jump size**: Optimal is $\sqrt{n}$
- **Space**: $O(1)$

Why Use:

- Simpler than binary search
- Better than linear search
- Good for systems where backward jumps expensive

Jump Search Implementation

```python
import math

def jump_search(arr, target):
    """Jump search in sorted array"""
    n = len(arr)
    step = int(math.sqrt(n))
    prev = 0

    # Jump to find block
    while arr[min(step, n) - 1] < target:
        prev = step
        step += int(math.sqrt(n))
        if prev >= n:
            return -1

    # Linear search in block
    while arr[prev] < target:
```

Complexity and Preconditions

Time Complexity Summary

Algorithm	Best	Avg	Worst	Space
Linear	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Binary	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Interpolation	$O(1)$	$O(\log \log n)$	$O(n)$	$O(1)$
Exponential	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Jump	$O(1)$	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(1)$
BST	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$
Hash Table	$O(1)$	$O(1)$	$O(n)$	$O(n)$

Performance Comparison (n=1,000,000):

- Linear: up to 1,000,000 comparisons
- Jump: $\sqrt{1,000,000} = 1,000$ comparisons
- Binary: $\log_2(1,000,000) \approx 20$ comparisons
- Interpolation: $\log \log(1,000,000) \approx 4$ comparisons
- Hash: 1 comparison (average)

Preconditions: Binary Search

MUST be sorted!

```python
# Correct usage
arr = [1, 3, 5, 7, 9, 11, 13]  # Sorted
result = binary_search(arr, 7)  # Works correctly

# Incorrect usage
arr = [3, 1, 5, 9, 7, 11, 13]  # NOT sorted
result = binary_search(arr, 7)  # May fail!

# Check if sorted
def is_sorted(arr):
    return all(arr[i] <= arr[i+1]
               for i in range(len(arr)-1))

# Safe usage
if is_sorted(arr):
    result = binary_search(arr, target)
```

Preconditions: Other Algorithms

Interpolation Search:

```
# MUST be sorted AND uniformly distributed
arr = [10, 20, 30, 40, 50]          # Good: uniform
arr = [1, 2, 3, 100, 1000]          # Bad: not uniform
```

Hash Table:

```
# Keys must be hashable (immutable)
hash_table[42] = "value"             # OK: int
hash_table["key"] = "value"          # OK: str
hash_table[(1, 2)] = "value"         # OK: tuple
hash_table[[1, 2]] = "value"         # ERROR: list!
```

BST:

```
# Elements must be comparable
bst.insert(5)     # OK: int
```

Choosing the Right Algorithm

Decision Factors:

1. Data Characteristics:
- Sorted? $\rightarrow$ Binary, Interpolation, Jump
- Uniformly distributed? $\rightarrow$ Interpolation
- Small size (< 100)? $\rightarrow$ Linear
- Large size? $\rightarrow$ Binary or Hash

2. Operation Frequency:
- Many searches? $\rightarrow$ Hash Table or BST
- One-time search? $\rightarrow$ Linear
- Frequent insertions/deletions? $\rightarrow$ Hash or BST

3. Requirements:
- Need sorted order? $\rightarrow$ BST or Sorted Array
- Range queries? $\rightarrow$ BST
- Fastest possible? $\rightarrow$ Hash Table

Applications and Patterns

Pattern 1: Finding Boundaries

Find first and last occurrence in sorted array

```python
def find_first_occurrence(arr, target):
    """Find first occurrence of target"""
    left, right = 0, len(arr) - 1
    result = -1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            result = mid
            right = mid - 1  # Continue searching left
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return result

def find_last_occurrence(arr, target):
    """Find last occurrence of target"""
    left, right = 0, len(arr) - 1
    result = -1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            result = mid
            left = mid + 1  # Continue searching right
```

Pattern 2: Rotated Array Search

Search in rotated sorted array

```python
def search_rotated(arr, target):
    """
    Search in rotated sorted array
    Example: [4,5,6,7,0,1,2] (rotated at index 4)
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid

        # Determine which half is sorted
        if arr[left] <= arr[mid]:
            # Left half is sorted
            if arr[left] <= target < arr[mid]:
                right = mid - 1
            else:
                left = mid + 1
        else:
            # Right half is sorted
            if arr[mid] < target <= arr[right]:
                left = mid + 1
            else:
                right = mid - 1

    return -1
```

Pattern 3: Peak Finding

```python
def find_peak_element(arr):
    """
    Find a peak element (greater than neighbors)
    """
    left, right = 0, len(arr) - 1

    while left < right:
        mid = left + (right - left) // 2

        if arr[mid] < arr[mid + 1]:
            left = mid + 1  # Peak is on right
        else:
            right = mid  # Peak is on left or at mid

    return left

# Example: [1, 3, 20, 4, 1, 0]
```

Pattern 4: 2D Matrix Search

```python
def search_matrix(matrix, target):
    """
    Search in row-wise and column-wise sorted matrix
    Example: [[1,4,7,11],
              [2,5,8,12],
              [3,6,9,16],
              [10,13,14,17]]
    """
    if not matrix or not matrix[0]:
        return False

    rows, cols = len(matrix), len(matrix[0])

    # Start from top-right corner
    row, col = 0, cols - 1

    while row < rows and col >= 0:
        if matrix[row][col] == target:
            return True
        elif matrix[row][col] > target:
            col -= 1  # Move left
        else:
            row += 1  # Move down

    return False
```

Time: $O(m + n)$, **Strategy**: Eliminate row or column each step

Pattern 5: Finding Missing Number

```python
def find_missing(arr):
    """
    Find missing number in [0, n]
    Example: [0,1,3,4,5] -> missing 2
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == mid:
            left = mid + 1  # Missing is on right
        else:
            right = mid - 1  # Missing is on left

    return left
```

Common Search Patterns Summary

1. **Find Exact Match:**
   - Standard binary search
   - Return index or -1
2. **Find First/Last Occurrence:**
   - Binary search with boundary tracking
   - Continue searching after finding match
3. **Find Insertion Position:**
   - Binary search returning left pointer
   - Used for maintaining sorted order
4. **Search in Rotated Array:**
   - Identify which half is sorted
   - Search appropriate half
5. **Find Peak/Valley:**
   - Compare with neighbors

Summary

Key Takeaways

Search Algorithm Categories:

- **Linear Search**: $O(n)$, works on any data
- **Binary Search**: $O(\log n)$, requires sorted data
- **Interpolation**: $O(\log \log n)$, uniform distribution
- **Hash Table**: $O(1)$ average, no order
- **BST**: $O(\log n)$, dynamic with sorted order

Key Concepts:

- Binary search eliminates half each step
- Parametric search: search answer space
- Hash tables trade space for speed
- Preconditions matter (sorted, uniform, etc.)

Choosing Algorithm:

- Consider: data size, sorted, distribution, frequency
- Small/unsorted: Linear
- Large/sorted: Binary

Practical Recommendations

Use Built-in Functions:

- Python: `list.index()`, `bisect` module, `dict`
- Java: `Arrays.binarySearch()`, `HashMap`
- C++: `std::binary_search()`, `std::map`

Common Mistakes to Avoid:

- Forgetting to sort before binary search
- Integer overflow in mid calculation
- Off-by-one errors in loop conditions
- Using linear search on large sorted data

Optimization Tips:

- Preprocess data (sort, build index)
- Cache frequent queries
- Use appropriate data structure
- Consider trade-offs: time vs space vs complexity

Practice Problems

Problem 1: Binary Search Variants

- Implement finding first and last occurrence
- Find insertion position for sorted array

Problem 2: Parametric Search

- Koko eating bananas (LeetCode 875)
- Capacity to ship packages (LeetCode 1011)
- Split array largest sum (LeetCode 410)

Problem 3: Rotated Array

- Search in rotated sorted array
- Find minimum in rotated array

Problem 4: 2D Search

- Search 2D matrix (LeetCode 74, 240)
- Find peak in 2D array

Resources

Books:

- "Introduction to Algorithms" (CLRS) - Chapters 2, 11, 12
- "The Algorithm Design Manual" (Skiena)

Online Resources:

- Visualizations: visualgo.net
- LeetCode Binary Search problems
- GeeksforGeeks search algorithm tutorials

Practice Platforms:

- LeetCode: Binary Search tag
- HackerRank: Search challenges
- Codeforces: Binary search problems

Advanced Topics:

- Ternary search
- Fractional cascading