Advanced Algorithm Analysis

Searching and Sorting Algorithms

PART A: SEARCHING ALGORITHMS

1. LINEAR SEARCH

Introduction

Linear Search is the simplest searching algorithm that checks every element in a list sequentially until the target element is found or the list ends. It works on both sorted and unsorted arrays. This algorithm is used in small datasets, unsorted data, or when simplicity is more important than speed.

Real-world use: Searching for a contact in an unsorted phone list, finding a book on an unorganized shelf.

Step-by-Step Working

1. Start from the first element of the array
2. Compare the current element with the target value
3. If they match, return the current index
4. If they don't match, move to the next element
5. Repeat steps 2-4 until element is found or array ends
6. If array ends without finding the element, return -1 (not found)

Example: Search for 7 in array [3, 8, 1, 7, 5]

- Check index 0: 3 ≠ 7
- Check index 1: 8 ≠ 7
- Check index 2: 1 ≠ 7
- Check index 3: 7 = 7 → Found at index 3!

Code with Line-by-Line Complexity

```python
```

```
def linear_search(arr, target):
    n = len(arr)                # O(1) - Getting length is constant

    for i in range(n):          # O(n) - Loop runs n times
        if arr[i] == target:    # O(1) - Single comparison
            return i            # O(1) - Return statement

    return -1                   # O(1) - Return if not found
```

Asymptotic Analysis

Case	Time Complexity	Explanation
Best Case	O(1)	Element found at first position
Average Case	O(n)	Element found in middle on average
Worst Case	O(n)	Element at end or not present
Space Complexity	O(1)	Only uses few variables

Detailed Explanation:

Time Complexity: Linear Search has O(n) complexity because in the worst scenario, we must check every single element in the array. If we have 100 elements, we might need 100 comparisons. If we have 1000 elements, we might need 1000 comparisons. The time grows linearly with input size.

Why this complexity? The algorithm doesn't skip any elements—it checks them one by one. There's no way to know where the target is without looking.

Practical meaning: For a list of 1 million items, you might need 1 million comparisons in the worst case. This makes linear search slow for large datasets but perfectly fine for small ones (under 100 elements).

Space Complexity: O(1) means constant space—we only use a few variables (n, i, target) regardless of array size. Whether the array has 10 or 10,000 elements, our memory usage stays the same.

---

2. BINARY SEARCH

Introduction

Binary Search is an efficient algorithm that finds an element in a **sorted array** by repeatedly dividing the search interval in half. It's much faster than linear search but requires the data to be sorted first. Used in phone directories, dictionaries, database indexing, and any system where data is pre-sorted.

Key requirement: Array must be sorted!

Step-by-Step Working

1. Start with two pointers: left = 0, right = n-1
2. Find the middle element: mid = (left + right) / 2
3. Compare the middle element with target
4. If middle element equals target, return mid (found!)
5. If target is smaller, search in left half (right = mid - 1)
6. If target is larger, search in right half (left = mid + 1)
7. Repeat steps 2-6 until element is found or left > right
8. If left > right, element doesn't exist, return -1

Example: Search for 7 in sorted array [1, 3, 5, 7, 9, 11, 13]

- Step 1: left=0, right=6, mid=3 → arr[3]=7 → Found at index 3!

Another example: Search for 6 in [1, 3, 5, 7, 9, 11, 13]

- Step 1: left=0, right=6, mid=3 → arr[3]=7 > 6, search left half
- Step 2: left=0, right=2, mid=1 → arr[1]=3 < 6, search right half
- Step 3: left=2, right=2, mid=2 → arr[2]=5 < 6, search right half
- Step 4: left=3, right=2 → left > right → Not found!

Code with Line-by-Line Complexity

python

```python
def binary_search(arr, target):
    left = 0                    # O(1) - Initialize left pointer
    right = len(arr) - 1        # O(1) - Initialize right pointer

    while left <= right:        # O(log n) - Loop runs log n times
        mid = (left + right) // 2   # O(1) - Calculate middle

        if arr[mid] == target:      # O(1) - Compare with target
            return mid              # O(1) - Return if found

        elif arr[mid] < target:     # O(1) - Check if target is larger
            left = mid + 1          # O(1) - Search right half

        else:                       # Target is smaller
            right = mid - 1         # O(1) - Search left half

    return -1                   # O(1) - Not found
```

Asymptotic Analysis

Case	Time Complexity	Explanation
Best Case	O(1)	Element found at middle position
Average Case	O(log n)	Element requires log divisions
Worst Case	O(log n)	Element at end or not present
Space Complexity	O(1)	Only uses pointer variables

Detailed Explanation:

Time Complexity: Binary Search has O(log n) complexity because we cut the search space in half with each step. For 1000 elements, we need at most 10 comparisons (2^10 = 1024). For 1 million elements, we need only 20 comparisons (2^20 ≈ 1 million).

Why this complexity? Each comparison eliminates half the remaining elements. This "divide and conquer" approach is extremely efficient.

Mathematical proof: After k steps, array size becomes n/2^k. When this equals 1, we're done: n/2^k = 1 → 2^k = n → k = $\log_2(n)$

Practical meaning: Doubling the data size only adds one more step! This makes binary search incredibly fast for large sorted datasets. Searching through 1 billion elements requires only 30 comparisons maximum.

Space Complexity: O(1) iterative version uses only a few pointer variables. (Note: Recursive version would use O(log n) space for call stack)

---

3. JUMP SEARCH

Introduction

Jump Search is a searching algorithm for sorted arrays that works by jumping ahead by fixed steps and then performing a linear search in the identified block. It's faster than linear search but slower than binary search. Best used when jumping backward is costly (like in tape storage systems) or when binary search overhead is too much for the system.

Key requirement: Array must be sorted!

Step-by-Step Working

1. Determine jump size: step = √n (square root of array length)
2. Start at index 0
3. Jump ahead by 'step' until arr[current] ≥ target or reach end
4. Once you've jumped past the target (or reached end), go back one jump

5. Perform linear search in that block (from previous jump to current position)
6. If element found in linear search, return index; else return -1

Example: Search for 7 in [1, 2, 3, 4, 5, 6, 7, 8, 9]

- n = 9, step = √9 = 3
- Jump 1: Check arr[3]=4 < 7, continue
- Jump 2: Check arr[6]=7 = 7 → Found at index 6!

Another example: Search for 6 in same array

- Jump 1: arr[3]=4 < 6
- Jump 2: arr[6]=7 > 6, jump back
- Linear search from index 3 to 6: Find 6 at index 5

Code with Line-by-Line Complexity

python

```python
import math

def jump_search(arr, target):
    n = len(arr)                    # O(1) - Get array length
    step = int(math.sqrt(n))        # O(1) - Calculate jump size
    prev = 0                        # O(1) - Previous jump position

    # Jump phase
    while arr[min(step, n) - 1] < target:  # O(√n) - At most √n jumps
        prev = step                 # O(1) - Store previous position
        step += int(math.sqrt(n))   # O(1) - Next jump

        if prev >= n:               # O(1) - Check if beyond array
            return -1               # O(1) - Not found

    # Linear search phase
    while arr[prev] < target:       # O(√n) - Linear search in block
        prev += 1                   # O(1) - Move to next element

        if prev == min(step, n):    # O(1) - Check block end
            return -1               # O(1) - Not found

    if arr[prev] == target:         # O(1) - Final check
        return prev                 # O(1) - Return position

    return -1                       # O(1) - Not found
```

Asymptotic Analysis

Case	Time Complexity	Explanation
Best Case	O(1)	Element at first jump position
Average Case	O(√n)	Combination of jumps and linear search
Worst Case	O(√n)	Element at end or not present
Space Complexity	O(1)	Only uses few pointer variables

Detailed Explanation:

Time Complexity: Jump Search has O(√n) complexity because we make approximately √n jumps and then perform √n comparisons in linear search within a block.

Why this complexity? We divide the array into √n blocks of size √n each. Worst case: we jump through all √n blocks (√n operations) and then search through one full block (√n operations). Total: √n + √n = 2√n = O(√n)

Mathematical reasoning: If jump size is k, we make n/k jumps and k linear comparisons. To minimize total work (n/k + k), we differentiate and find optimal k = √n.

Practical meaning: For 10,000 elements, we need only 100 jumps + 100 comparisons = 200 operations maximum, compared to 10,000 for linear search. However, binary search would need only 14 operations, making it still more efficient.

When to use Jump Search? In systems where backward movement is expensive (tape drives, certain data structures), jump search is better than binary search because it only moves forward.

Space Complexity: O(1) because we only use a few variables (n, step, prev) regardless of input size.

PART B: SORTING ALGORITHMS

4. BUBBLE SORT

Introduction

Bubble Sort is the simplest sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they're in wrong order. The largest element "bubbles up" to its correct position after each pass. Used primarily for teaching purposes and very small datasets.

Real-world use: Educational purposes, nearly sorted data with few elements.

Step-by-Step Working

1. Start from the first element of array
2. Compare current element with next element
3. If current element is greater, swap them
4. Move to next pair and repeat steps 2-3
5. After one complete pass, the largest element reaches the end
6. Repeat the process for remaining n-1 elements
7. Continue until no swaps are needed (array is sorted)

Example: Sort [5, 2, 8, 1, 9]

- Pass 1: [2, 5, 1, 8, 9] - 9 bubbles to end
- Pass 2: [2, 1, 5, 8, 9] - 8 in position
- Pass 3: [1, 2, 5, 8, 9] - 5 in position
- Pass 4: [1, 2, 5, 8, 9] - Already sorted

Code with Line-by-Line Complexity

```python
def bubble_sort(arr):
    n = len(arr)                    # O(1) - Get array length

    for i in range(n):              # O(n) - n passes
        swapped = False             # O(1) - Flag for optimization

        for j in range(0, n - i - 1):   # O(n) - Compare adjacent elements
            if arr[j] > arr[j + 1]:     # O(1) - Comparison
                # Swap elements
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  # O(1) - Swap
                swapped = True          # O(1) - Mark swap occurred

        if not swapped:             # O(1) - Check if sorted
            break                   # O(1) - Exit early if sorted

    return arr                      # O(1) - Return sorted array
```

Asymptotic Analysis

Case	Time Complexity	Explanation
Best Case	O(n)	Array already sorted, one pass needed
Average Case	$O(n^2)$	Random order, multiple passes needed
Worst Case	$O(n^2)$	Reverse sorted, maximum swaps needed
Space Complexity	O(1)	Sorts in-place, no extra space

Detailed Explanation:

Time Complexity: Bubble Sort has $O(n^2)$ complexity because of nested loops. Outer loop runs n times, inner loop runs n times on average, giving $n \times n = n^2$ operations.

Why this complexity? First pass: n-1 comparisons, Second pass: n-2 comparisons, continuing this pattern gives us: $(n-1) + (n-2) + ... + 1 = n(n-1)/2 = O(n^2)$

Best case O(n): If array is already sorted, the optimized version (with swapped flag) detects this in one pass and stops early. Only n comparisons needed.

Worst case $O(n^2)$: If array is reverse sorted [5, 4, 3, 2, 1], every element must be compared and swapped with every other element, requiring maximum number of operations.

Practical meaning: For 1000 elements in worst case, we need approximately 1,000,000 operations. This makes bubble sort impractical for large datasets. However, it's simple to understand and implement, making it great for

learning.

Space Complexity: O(1) means it sorts in-place by swapping elements within the original array. No additional arrays or data structures needed.

---

5. INSERTION SORT

Introduction

Insertion Sort builds the sorted array one element at a time by repeatedly picking the next element and inserting it into its correct position among the previously sorted elements. It works like sorting playing cards in your hand—you pick one card and place it in the correct position. Used for small datasets, nearly sorted data, and online sorting (when data arrives one piece at a time).

Real-world use: Sorting cards while playing, organizing files by date, small datasets.

Step-by-Step Working

1. Assume the first element is already sorted
2. Pick the next element (call it 'key')
3. Compare 'key' with elements in the sorted portion (moving backward)
4. Shift all larger elements one position to the right
5. Insert 'key' at the correct position
6. Repeat steps 2-5 for all remaining elements
7. Array is sorted when all elements are processed

Example: Sort [5, 2, 8, 1, 9]

- Start: [5 | 2, 8, 1, 9] - 5 is sorted
- Step 1: [2, 5 | 8, 1, 9] - Insert 2 before 5
- Step 2: [2, 5, 8 | 1, 9] - 8 already in place
- Step 3: [1, 2, 5, 8 | 9] - Insert 1 at beginning
- Step 4: [1, 2, 5, 8, 9] - 9 already in place

Code with Line-by-Line Complexity

```
python
```

```
def insertion_sort(arr):
    n = len(arr)                    # O(1) - Get array length

    for i in range(1, n):           # O(n) - Process each element
        key = arr[i]                # O(1) - Current element to insert
        j = i - 1                   # O(1) - Start of sorted portion

        # Shift elements larger than key
        while j >= 0 and arr[j] > key:  # O(n) worst - Compare & shift
            arr[j + 1] = arr[j]         # O(1) - Shift element right
            j -= 1                      # O(1) - Move backward

        arr[j + 1] = key            # O(1) - Insert key at position

    return arr                      # O(1) - Return sorted array
```

Asymptotic Analysis

Case	Time Complexity	Explanation
Best Case	O(n)	Array already sorted, no shifting
Average Case	$O(n^2)$	Random order, some shifting needed
Worst Case	$O(n^2)$	Reverse sorted, maximum shifting
Space Complexity	O(1)	Sorts in-place, no extra space

Detailed Explanation:

Time Complexity: Insertion Sort has $O(n^2)$ worst case complexity because for each element, we might need to compare and shift it past all previously sorted elements.

Why this complexity? For element at position i, worst case requires i comparisons and i shifts. Total work: $1 + 2 + 3 + ... + n = n(n+1)/2 = O(n^2)$

Best case O(n): When array is already sorted, the inner while loop never executes. We just compare each element once with its previous element, giving us n comparisons total.

Average case $O(n^2)$: On average, we need to shift each element halfway through the sorted portion, giving $(n^2)/4$ operations, which is still $O(n^2)$.

Worst case $O(n^2)$: When array is reverse sorted [5, 4, 3, 2, 1], each element must be moved to the very beginning, causing maximum shifts.

Practical meaning: Despite $O(n^2)$ complexity, insertion sort is often faster than other $O(n^2)$ algorithms for small arrays ($n < 50$) because:

- Simple operations (few comparisons per iteration)

- Good cache performance (sequential access)
- Very efficient for nearly sorted data

When to use: Best choice when data is nearly sorted, array is small, or you're receiving data one element at a time and need to keep it sorted.

Space Complexity: O(1) because sorting happens in-place using only one extra variable (key) regardless of array size.

---

6. MERGE SORT

Introduction

Merge Sort is an efficient, divide-and-conquer sorting algorithm that divides the array into two halves, recursively sorts them, and then merges the sorted halves back together. It guarantees O(n log n) performance regardless of input. Used in external sorting (sorting large files that don't fit in memory), linked list sorting, and when stable sorting is required.

Real-world use: Database sorting, sorting large files, version control systems (like Git).

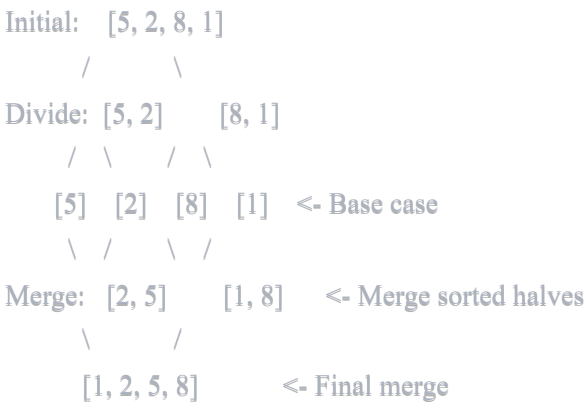
Step-by-Step Working

1. Divide the array into two equal halves
2. Recursively sort the left half
3. Recursively sort the right half
4. Merge the two sorted halves into one sorted array
5. Base case: Array of size 1 is already sorted

Merging process:

- Create two pointers, one for each sorted half
- Compare elements at both pointers
- Add smaller element to result array
- Move that pointer forward
- Repeat until one array is exhausted
- Copy remaining elements from other array

Example: Sort [5, 2, 8, 1]



Code with Line-by-Line Complexity

python

```
def merge_sort(arr):
    if len(arr) <= 1:                # O(1) - Base case check
        return arr                   # O(1) - Return if size 1

    # Divide phase
    mid = len(arr) // 2              # O(1) - Find middle
    left = arr[:mid]                 # O(n) - Create left subarray
    right = arr[mid:]                # O(n) - Create right subarray

    # Conquer phase (recursive calls)
    left = merge_sort(left)          # T(n/2) - Sort left half
    right = merge_sort(right)        # T(n/2) - Sort right half

    # Combine phase
    return merge(left, right)        # O(n) - Merge sorted halves

def merge(left, right):
    result = []                      # O(1) - Initialize result
    i = j = 0                        # O(1) - Initialize pointers

    # Merge while both arrays have elements
    while i < len(left) and j < len(right):  # O(n) - Compare all
        if left[i] <= right[j]:      # O(1) - Compare elements
            result.append(left[i])   # O(1) - Add to result
            i += 1                   # O(1) - Move left pointer
        else:
            result.append(right[j])  # O(1) - Add to result
            j += 1                   # O(1) - Move right pointer

    # Copy remaining elements
    result.extend(left[i:])          # O(n) - Remaining left
    result.extend(right[j:])         # O(n) - Remaining right

    return result                    # O(1) - Return merged array
```

Asymptotic Analysis

Case	Time Complexity	Explanation
Best Case	O(n log n)	Always divides and merges same way
Average Case	O(n log n)	Consistent performance
Worst Case	O(n log n)	Guaranteed performance
Space Complexity	O(n)	Requires extra arrays for merging

Detailed Explanation:

Time Complexity: Merge Sort has O(n log n) complexity in all cases because it always divides the array in half (log n levels) and merges all elements at each level (n work per level).

Why this complexity?

- **Divide step:** Splitting array in half creates $\log_2(n)$ levels in the recursion tree. For 8 elements: 8→4→2→1 = 3 levels = $\log_2(8)$
- **Merge step:** At each level, we merge all n elements across all subarrays
- **Total work:** n work per level × log n levels = n log n

Recurrence relation: $T(n) = 2T(n/2) + O(n)$

- $2T(n/2)$: Two recursive calls on half-sized arrays
- $O(n)$: Linear work to merge

Why O(n log n) is optimal: For comparison-based sorting, O(n log n) is the best possible worst-case performance. Merge sort achieves this theoretical optimum.

Practical meaning: For 1 million elements, merge sort needs approximately 20 million operations ($1M \times \log_2(1M) \approx 1M \times 20$), while insertion sort would need 1 trillion operations. This makes merge sort practical for large datasets.

Advantages:

- Guaranteed O(n log n) performance (no worst cases)
- Stable sort (preserves relative order of equal elements)
- Predictable performance
- Excellent for linked lists (no random access needed)

Disadvantages:

- Requires O(n) extra space for temporary arrays
- Slower than quicksort on small arrays
- Not in-place sorting

Space Complexity: O(n) because we create temporary arrays during merging. At any point, we need space for the original array plus temporary merge arrays totaling n elements. Additionally, recursive call stack uses O(log n) space, but O(n) dominates, so overall space is O(n).

When to use: Choose merge sort when you need guaranteed performance, stable sorting, or are working with linked lists or external storage where sequential access is better than random access.

SUMMARY COMPARISON TABLE

Algorithm	Best Case	Average Case	Worst Case	Space	Stable?
Linear Search	O(1)	O(n)	O(n)	O(1)	-
Binary Search	O(1)	O(log n)	O(log n)	O(1)	-
Jump Search	O(1)	O(√n)	O(√n)	O(1)	-
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes

Note: Stable sorting means equal elements maintain their relative order from the input.

KEY TAKEAWAYS

Searching:

- Linear Search: Simple but slow, works on unsorted data
- Binary Search: Very fast but requires sorted data
- Jump Search: Middle ground, moves forward only

Sorting:

- Bubble Sort: Simple, good for teaching, slow for large data
- Insertion Sort: Efficient for small/nearly sorted data
- Merge Sort: Guaranteed fast performance, uses extra space

General Rule: For large datasets, use O(n log n) algorithms (merge sort). For small datasets ($n < 50$), simpler $O(n^2)$ algorithms (insertion sort) can be faster due to lower overhead.

End of Document