UNIT V: Searching, Sorting, and Hashing

A Comprehensive Study Guide with Real-Time Examples

Searching Techniques

1. Linear Search:

Examines each element sequentially until finding the target.

Time Complexity: O(n)

Space Complexity: O(1)

Best Case: O(1) - element at first position

Worst Case: O(n) - element at end or not present

When to Use:

Small unsorted lists, linked lists, or when simplicity is preferred over performance.

Real-Time Example:

Library Book Search: Before computers, librarians used linear search physically examining shelf tags one by one. This was slow but necessary without alphabetical indexing.

2. Binary Search:

Efficiently searches sorted data by repeatedly dividing search space in half.

Time Complexity: O(log n)

Space Complexity: O(1) iterative, O(log n) recursive

Prerequisite: Array must be sorted

Real-Time Example:

Dictionary Lookup: When looking up a word in a physical dictionary, you don't read from page 1. You estimate the position (binary search), then recursively narrow down. First go to middle, if target word comes earlier, ignore second half, then bisect the remaining portion.

Another Real-Time Example:

Employee ID Lookup: A company database with 100,000 sorted employee IDs uses binary search for lookups. Instead of checking all 100,000 records (linear: 100,000 comparisons max), binary search requires only ~17 comparisons to locate any employee.

Sorting Techniques

1. Bubble Sort:

Compare adjacent elements repeatedly, swapping if needed. Slowest but easiest to understand. Stable sort.

Best Case: $O(n)$

Average Case: $O(n^2)$

Worst Case: $O(n^2)$

Space Complexity: $O(1)$

How it works: In each pass, the largest unsorted element "bubbles" to its correct position. After n passes, the array is sorted.

2. Selection Sort:

Find minimum element, place at beginning, repeat. Not stable.

Best Case: $O(n^2)$

Average Case: $O(n^2)$

Worst Case: $O(n^2)$

Space Complexity: $O(1)$

How it works: Divide array into sorted and unsorted parts. Repeatedly select minimum from unsorted part and move to sorted part.

3. Insertion Sort:

Build sorted array by inserting elements one by one. Stable sort. Good for small lists.

Best Case: $O(n)$

Average Case: $O(n^2)$

Worst Case: $O(n^2)$

Space Complexity: $O(1)$

How it works: Compare each element with those before it and insert in correct position. Similar to sorting playing cards in your hand.

4. Merge Sort:

Divide-and-conquer: divide list, sort halves, merge. Stable sort. Requires extra space.

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n log n)

Space Complexity: O(n)

How it works: Recursively divide array in half until single elements, then merge sorted subarrays back together.

5. Quick Sort:

Partition around pivot, recursively sort partitions. Fast in practice but unstable. Average best.

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: $O(n^2)$

Space Complexity: O(log n)

How it works: Select pivot, partition array so elements less than pivot are left and greater are right, recursively sort partitions.

6. Heap Sort:

Build heap, extract minimum repeatedly. Unstable sort. In-place.

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n log n)

Space Complexity: O(1)

How it works: Build a min heap from array, then repeatedly extract minimum and place at end. Result is sorted array.

Comparison Table:

Algorithm	Best Case	Average	Worst Case	Space
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
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	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

Real-Time Example:

E-commerce Product Sorting: When you filter products by price on Amazon, the system uses an efficient sorting algorithm. QuickSort is often chosen because it's fast on average and handles real-world data well. For recommendations requiring stability (maintaining product ratings when prices are tied), MergeSort is preferred.

Hashing

Hashing is a technique for fast data retrieval by computing an index (hash code) from a key using a hash function.

Hash Function Properties:

Deterministic: Same input always produces same output

Uniform Distribution: Keys distributed evenly across table

Fast Computation: Should run in O(1) time

Minimal Collisions: Different keys should map to different slots

Common Hash Functions:

Division Method: h(k) = k mod m (m = table size)

Multiplication Method: h(k) = floor(m × (k×A mod 1)), A = golden ratio constant

Folding Method: Divide key into parts, add and mod by table size

Real-Time Example:

Password Verification: When you create an online account, your password isn't stored as plain text. Instead, a hash function converts it to a fixed-size hash. When logging in, the system hashes your input and compares with stored hash. Even if database is compromised, hackers see hashes, not passwords.

Another Real-Time Example:

Duplicate File Detection: Cloud storage services like Google Drive use hashing to detect duplicate files. Each file is hashed, and the system quickly checks if a file with that hash exists, avoiding duplicate uploads.

Collision Resolution Techniques

When two keys hash to the same slot, a collision occurs. Various techniques resolve this.

1. Open Addressing:

Linear Probing: If slot is occupied, check next slot sequentially.

Advantage: Simple, uses hash table space efficiently

Disadvantage: Primary clustering (chains of occupied slots)

Quadratic Probing: Check slots at $h(k)$, $h(k)+1^2$, $h(k)+2^2$, $h(k)+3^2$, etc.

Advantage: Reduces clustering compared to linear probing

Double Hashing: Use second hash function $h2(k)$ for step size.

Advantage: Distributes probes well, minimal clustering

2. Closed Addressing (Chaining):

Each hash table slot maintains a linked list of elements with the same hash.

Advantage: Simple, handles high load factors, no clustering

Disadvantage: Extra memory for pointers, cache-unfriendly

Real-Time Example:

Dictionary Implementation: Hash tables implement dictionaries in Python. Keys hash to table slots. If multiple keys collide, chaining (linked lists) stores all values for that hash. Lookup is $O(1)$ average, $O(n)$ worst (all keys hash to one slot).