Searching and Sorting: Part 1

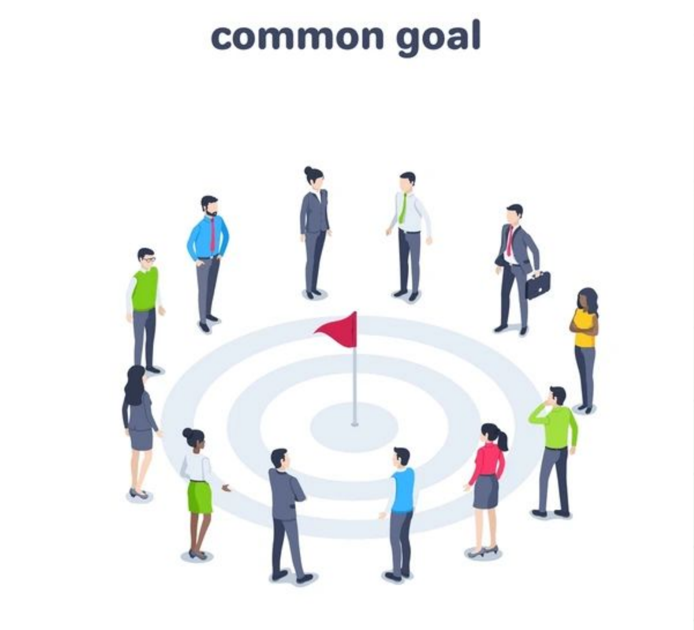
DSA

Presented by
Nikhil Nair

Website
www.guvi.com

Objectives

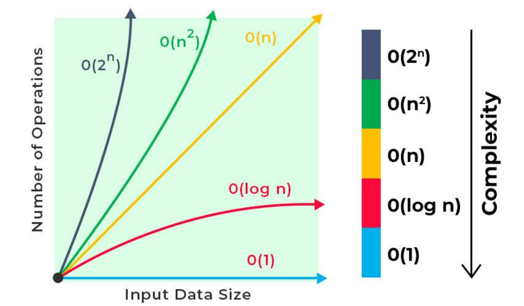
- What you will learn today
    - Linear & Binary Search
    - Binary Search on Answers
    - Bubble, Selection, Insertion Sort



Introduction to Searching

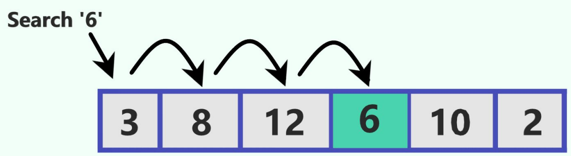
Why Search?

- Searching is the single most common operation a computer does
- Use Cases
  - Google: How does Google find the most relevant sites out of billions?
  - E-commerce: How does Amazon make it easy to explore ALL their products?
  - Your IDE: When you Ctrl/Cmd+Click on a method call, how does the IDE immediately allow you to visit the method definition?
  - Netflix: How do they find and suggest content to you from a library of tens of thousands?
- In such scenarios, search and optimal performance matters.
- Why does optimal matter? Consider a problem: Searching 1,000,000,000 (1 Billion) items
  - O(n): 1,000,000,000 operations (take several seconds)
  - O(log n): 30 operations (instant)



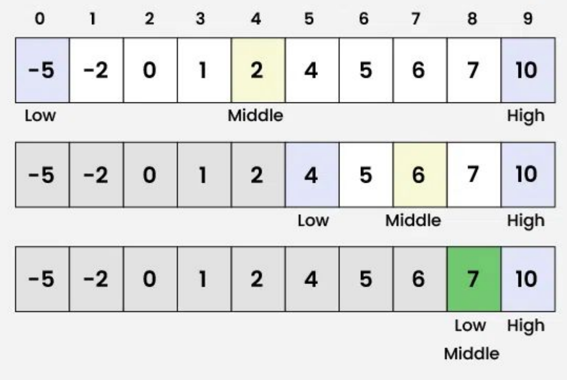
Linear Search

- Concept
    - The most basic, common-sense search
    - It sequentially checks every element in a collection, one by one, until the target is found
- Approach
    - Use a for loop to iterate from i = 0 to array.length - 1.
    - Inside the loop, check if array[i] == target.
    - If a match is found, return i (the index).
    - If the loop finishes without a match, return -1.
- Key Trait
    - Works on any collection, sorted or unsorted. This is its main advantage
- Time Complexity: O(n). In the worst case, we must check every single element
- Time Complexity: O(1). We only use a few variables for the loop; no extra memory is allocated



Binary Search

- The Requirement: The array MUST be sorted first
- Concept
    - A highly efficient "divide and conquer" algorithm.
    - Check the middle element (not the first).
    - If our target is smaller, we throw away the entire right half of the array.
    - If our target is larger, we throw away the entire left half.
- Approach
    - Set two pointers: low = 0 and high = array.length - 1.
    - Loop while (low <= high).
    - Find the middle: mid = low + (high - low) / 2.
    - If array[mid] == target: We found it! return mid.
    - If target < array[mid]: Throw away the right. high = mid - 1.
    - If target > array[mid]: Throw away the left. low = mid + 1.
- Time Complexity: O(log n). We cut the search space in half with every single guess
- Space Complexity: O(1). We only store three variables: low, high, and mid



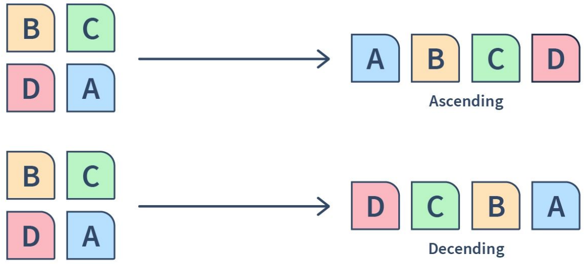
Binary Search on Answers

- Concept
  - This is a problem-solving pattern, not a direct array search.
  - We use the idea of binary search to find a specific answer that exists within a large, continuous range of possibilities
- Example Problem: "What is the square root of 2100?"
- Approach
  - We know the answer must be in a range, e.g., [0 ... 2100]. This is our "search space."
  - Let's binary search this answer range:
  - low = 0, high = 2100.
  - Guess mid = 1050. Is 1050 * 1050 > 2100? Yes. The answer must be smaller. New range: [0 ... 1049].
  - ...this continues until low and high converge on the answer
- You can use binary search for any answer where you need to quickly check if your "guess" is too high or too low

Introduction to Sorting

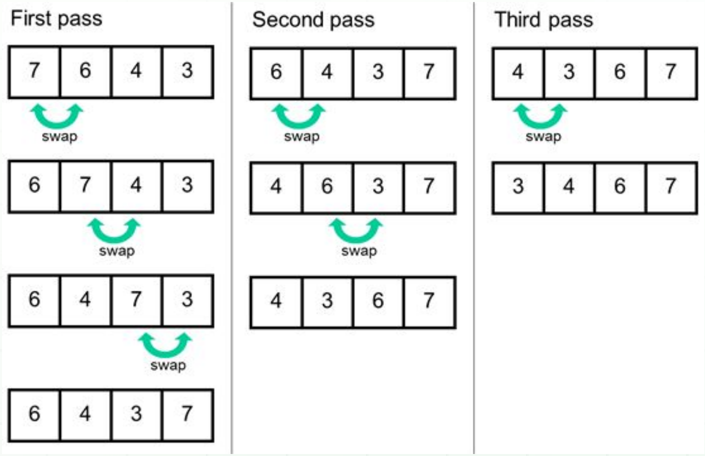
Why Sort?

- Concept:
    - The process of arranging items in a specific order (e.g., ascending).
    - We will start with three fundamental (but slow) sorts.
- Key Terminology
    - **In-Place**: The algorithm sorts within the original array, without creating a new one. (Uses O(1) space).
    - **Stable**: A sort is "stable" if elements with identical values (e.g., two 5s) are guaranteed to remain in their original relative order after the sort is complete



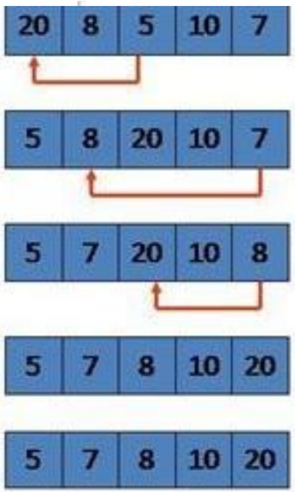
Bubble Sort

- Concept
    - Compares adjacent pairs, swapping them if they are in the wrong order.
    - This "bubbles" the largest element to the end of the array on each pass
- Approach
    - Use a nested for loop.
    - The outer loop (i) controls how many passes we do (or how many elements are "locked in" at the end).
    - The inner loop (j) does the comparisons and swaps, from 0 up to n-i-1.
    - if (nums[j] > nums[j+1]), we swap
- Time Complexity: O(n^2). A nested loop structure where we compare n elements n times
- Time Complexity: O(1). It's fully in-place; we only need a temp variable for swapping



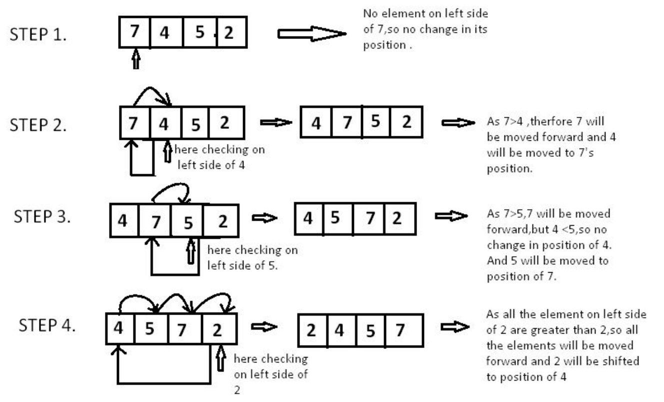
Selection Sort

- Concept
  - "Selects" the minimum element from the unsorted part of the list.
  - Swaps that minimum element with the first element of the unsorted part.
- Approach
  - The outer loop (i) iterates from 0 to n-1, tracking the start of the "unsorted" section.
  - In the inner loop (j), we don't swap. We just find the index of the minimum element (min_index).
  - After the inner loop finishes, we perform one swap: swap(nums[i], nums[min_index])
- Time Complexity: O(n^2). The "find min" operation (O(n)) is inside the main loop (O(n))
- Time Complexity: O(1). It's fully in-place



Insertion Sort

- Concept
  - Builds the final sorted array one item at a time.
  - It takes the next unsorted element (the key) and "inserts" it into its correct position within the already sorted part on the left.
- Approach
  - The outer loop (i) starts at 1 (the first "unsorted" element).
  - Store nums[i] as our key.
  - Use a while loop (or inner for loop) to "shift" all elements in the sorted part that are greater than our key one position to the right.
  - When the shifting is done, insert the key into the open slot.
- Time Complexity: O(n^2)
- Best Case: If the array is already sorted, it's O(n) because the inner while loop never runs.
- Time Complexity: O(1). It's fully in-place



That's for today!
Any questions?