CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Search and Sort

Department of Computer Science
University of Maryland, College Park

Linear Search

- **Idea:**
  - **Brute-force approach**: Check each element one by one.
  - Works on **both sorted and unsorted** arrays.
  - **Best case**: The target is found at the first position.
  - **Worst case**: The target is at the last position or not in the array at all.
  - **Time Complexity**: O(n) in the worst case.
- **Advantages:**
  - Simple to implement.
  - No need for sorting.
  - Works on all types of data.
- **Disadvantages:**
  - Inefficient for large datasets.
  - Takes linear time in the worst case.
    See: `linearSearch`

Binary Search

- **Idea:**
  - **Divide and conquer**: Repeatedly divide the search space in half.
  - Only works on **sorted arrays**.
  - **Best case**: The target is found in the middle.
  - **Worst case**: The target is not found, requiring log(n) comparisons.
  - **Time Complexity**: O(log n) in the worst case.
- **Process:**
  1. Find the middle element.
  2. If the target is equal to the middle element, return it.
  3. If the target is smaller, search the left half.
  4. If the target is larger, search the right half.
  5. Repeat until the element is found or the search space is empty.
- **Advantages:**
  - Much faster than linear search for large datasets.
  - Efficient with sorted data.
- **Disadvantages:**
  - Only works on sorted data.
  - Requires extra steps to maintain a sorted list.
  - More complex to implement compared to linear search.
  - .
    - See: `BinarySearch and BinarySearchRecursive`

When to Use Which?

- Use **Linear Search** when:
  - The array is **small**.
  - The array is **unsorted** and sorting is not feasible.
  - Searching needs to be applied just once or infrequently.
- Use **Binary Search** when:
  - The array is **large** and already **sorted**.
  - Fast lookups are needed frequently.
  - Sorting overhead is acceptable.

Feature	Linear Search	Binary Search
Data must be sorted?	No	Yes
Time Complexity (Worst)	O(n)	O(log n)
Best case time	O(1)	O(1)
Use case	Small or unsorted data	Large sorted data
Implementation complexity	Easy	Moderate

Bubble Sort– Part 1

- **Idea**
  - **Comparison-based sorting algorithm**.
  - Repeatedly swaps adjacent elements if they are in the wrong order.
  - The largest element "bubbles up" to its correct position after each pass.
- **Time Complexity**:
  - **Worst case**: $O(n^2)$ (when the array is reversed).
  - **Best case**: O(n) (if the array is already sorted, with an optimized version).
- **Process**
  1. Compare adjacent elements.
  2. Swap them if they are in the wrong order.
  3. Repeat for all elements, pushing the largest unsorted element to its correct position.
  4. Reduce the range and repeat until the array is sorted.
- **Example**
- Sorting **[5, 3, 8, 4, 2]**
- **Pass 1**
  - **(5,3) → swap → [3, 5, 8, 4, 2]**
  - **(5,8) → no swap**
  - **(8,4) → swap → [3, 5, 4, 8, 2]**
  - **(8,2) → swap → [3, 5, 4, 2, 8]** // Largest element **8** is now in place.
- **Pass 2**
  - **(3,5) → no swap**
  - **(5,4) → swap → [3, 4, 5, 2, 8]**
  - **(5,2) → swap → [3, 4, 2, 5, 8].** // Largest remaining element **5** is now in place.

- Repeats until sorted: **[2, 3, 4, 5, 8]**

Bubble Sort – Part 2

- **Advantages**
  - **Simple and easy to understand**
    In-place sorting (no extra memory needed)
    Stable sort (preserves order of equal elements)
- **Disadvantages**
  - **Inefficient for large datasets** $O(n^2)$
    Too many swaps compared to other sorting algorithms
    Slower than Selection Sort and Insertion Sort for large inputs
- **Optimized Bubble Sort**
  - If no swaps occur in a pass, the array is already sorted.
  - **Improves best case to O(n)** when the array is already sorted.

- See: `BubbleSort and OptimizedBubbleSort`

Selection Sort – Part 1

- **Idea**
  - **Comparison-based sorting algorithm**.
  - Repeatedly finds the **smallest (or largest) element** and moves it to the correct position.
  - **In-place algorithm** (doesn't require extra memory).
  - **Time Complexity**: $O(n^2)$ in all cases.
- **Process**
  1. Start from the first position.
  2. Find the **smallest** element in the remaining array.
  3. Swap it with the element at the current position.
  4. Move to the next position and repeat until the array is sorted.
- **Example**
  - **Sorting: [5, 3, 8, 4, 2]**
  - **Step 1**: Find **2** → Swap with **5** → **[2, 3, 8, 4, 5]**
  - **Step 2**: Find **3** (already in place) → **[2, 3, 8, 4, 5]. //but it still has to do the work of scan the whole array**
  - **Step 3**: Find **4** → Swap with **8** → **[2, 3, 4, 8, 5]**
  - **Step 4**: Find **5** → Swap with **8** → **[2, 3, 4, 5, 8]**
  - **Step 5**: Array is now sorted!

Selection Sort – Part 2

- **Advantages**

  Simple and easy to understand.
  Works well for small datasets.
  Fewer swaps compared to Bubble Sort.

- **Disadvantages**

  Inefficient for large datasets due to $O(n^2)$ complexity.
  Not adaptive (does not take advantage of partial sorting).
  Slower than other O(n log n) sorting algorithms like QuickSort or MergeSort.

See: `SelectionSort`