# Java Programming 3-4: Sorting and Searching

# Practice Activities

**Vocabulary Definitions**

1.

**Also called a linear search, an algorithm that searches through an array until a value is found. The array does not need to be sorted.**

2.

- **Linear Search**: A search algorithm that checks each element in the array sequentially until it finds the target value or reaches the end of the array.

3.

**An algorithm that finds the minimum value in an array and swaps that value with the first number in the array. The next smallest value is swapped with the second number in the array. The process is repeated until the array is sorted.**

4.

- **Selection Sort**: A sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted portion of the array and swaps it with the first unsorted element.

5.

**An algorithm that checks the value of the first two elements, then swaps them if necessary so that the larger of the two is the second number. Next, the second and third numbers are compared. The larger of those two are swapped, if necessary so that the larger of the two is the second number. The process continues until the largest number in the array is the last number in the array. Then the process is repeated until the array is sorted.**

6.

- **Bubble Sort**: A sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the array is sorted.

7. **A search algorithm that works with sorted data. The array is divided in half, searched in the correct half repeatedly, until the item is found.**

8.
- **Binary Search**: A search algorithm that divides a sorted array into halves and checks which half the target value lies in, repeating this process until the target is found or the search space is empty.

9. **An algorithm that divides an array in half repeatedly until all data is isolated. Then the isolated data is "merged" back together in correct order.**

10.
- **Merge Sort**: A sorting algorithm that divides the array into halves, sorts each half, and then merges the sorted halves back together.

11. **The ASCII sort order for data.**

12.
- **ASCII Order**: The order in which characters are arranged based on their ASCII values, with characters being compared based on their ASCII values.

## 1. Create a Program for Sorting and Searching

**a. Create a project named `sortAndSearch`.**

**b. Create an integer array named `numbers` that will hold 50 values.**

**c. Fill the array with random integer values between 0 and 100.**

**d. Display the contents of the array under the heading "Unordered list".**

**e. Get the number to be searched for from the user.**

**f. Use a sequential/linear search to identify if the value is in the array. If the number is found, return the position; otherwise, return -1.**

**g. Display the result of the search identifying the position it was found at or displaying a not found message.**

**h. Sort the array using a bubble sort.**

**i. Display the contents of the array under the heading "Ordered list".**

**j. Use a sequential/linear search to identify if the value is in the array. If the number is found, return the position; otherwise, return -1.**

Here's the complete Java program implementing the above steps:

```
import java.util.Random;
import java.util.Scanner;

public class SortAndSearch {

   public static void main(String[] args) {
      final int SIZE = 50;
      int[] numbers = new int[SIZE];
      Random rand = new Random();

      // Fill the array with random numbers between 0 and 100
      for (int i = 0; i < SIZE; i++) {
         numbers[i] = rand.nextInt(101);
      }

      // Display the unordered list
      System.out.println("Unordered list:");
      displayArray(numbers);

      // Get the number to search for from the user
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter a number to search for: ");
      int searchNumber = scanner.nextInt();

      // Linear search
      int position = linearSearch(numbers, searchNumber);
      if (position != -1) {
         System.out.println("Number found at position: " + position);
      } else {
         System.out.println("Number not found.");
      }

      // Bubble sort
      bubbleSort(numbers);

      // Display the ordered list
      System.out.println("Ordered list:");
      displayArray(numbers);

      // Linear search after sorting
      position = linearSearch(numbers, searchNumber);
```

```
        if (position != -1) {
            System.out.println("Number found at position: " + position);
        } else {
            System.out.println("Number not found.");
        }

        scanner.close();
    }

    public static void displayArray(int[] array) {
        for (int num : array) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    public static int linearSearch(int[] array, int value) {
        for (int i = 0; i < array.length; i++) {
            if (array[i] == value) {
                return i;
            }
        }
        return -1;
    }

    public static void bubbleSort(int[] array) {
        boolean swapped;
        for (int i = 0; i < array.length - 1; i++) {
            swapped = false;
            for (int j = 0; j < array.length - 1 - i; j++) {
                if (array[j] > array[j + 1]) {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                    swapped = true;
                }
            }
            if (!swapped) {
                break;
            }
        }
    }
}
```

## 2. Analyze Big-O Notation for Various Sorting Algorithms

| Algorithm | Worst | Average | Best | Notes |
|---|---|---|---|---|
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | Selection Sort always performs $O(n^2)$ comparisons regardless of the input data's in order. |
| Bubble Sort | $O(n^2)$ | $O(n^2)$ | O(n) | Worst and average case is $O(n^2)$, but best case (already sorted) is O(n) if optimized a swapped flag. |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) | Merge Sort consistently performs O(n log n) due to the divide-and-conquer appro |

- **Notes**:
- **Selection Sort**: Inefficient for large lists as it always performs $O(n^2)$ comparisons.
- **Bubble Sort**: Can be optimized for nearly sorted data but still generally performs $O(n^2)$ in the worst and average cases.
- **Merge Sort**: Generally more efficient and scales well due to its O(n log n) performance.

### 3. Difference Between Linear and Binary Search

- 

**Linear Search**:

- 
- **Description**: Checks each element sequentially from the beginning to the end of the array.
- **Time Complexity**: O(n).
- **Requirement**: Works on both sorted and unsorted arrays.
- **Example**: Searching for a name in a list where the names are not in any particular order.
- 

**Binary Search**:

- 
- **Description**: Repeatedly divides the search interval in half. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half.
- **Time Complexity**: O(log n).
- **Requirement**: Requires the array to be sorted.
- **Example**: Finding a name in a sorted list of names.

### 4. Sorting Order for Strings and Numbers

- 

**Strings**: Sorted lexicographically based on ASCII values of characters. For example, "apple" comes before "banana" because 'a' < 'b' in ASCII.

- 
-

**Numbers**: Sorted numerically in ascending order. For example, 2 comes before 10.

- **Sorting Order**:

- When comparing mixed types (strings and numbers), sorting is typically done based on the type of the elements. You can't directly sort mixed types using standard sorting methods as they require homogeneous data types. You would need to convert everything to a common type (e.g., strings) before sorting or handle them separately.

This comprehensive guide should help you understand and implement sorting and searching algorithms in Java, along with their complexities and practical applications.