Sorting and Searching Techniques (Python & Java)

1. Linear Search

Definition:

Linear Search is the simplest searching algorithm where each element in a list is checked one by one until the target element is found or the list ends.

Process:

1. Start from the first element.
2. Compare each element with the target.
3. If a match is found, return the index.
4. If not, move to the next element.
5. Continue until the element is found or the list ends.

Python Code:

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

Java Code:

```java
class LinearSearch {
    static int search(int arr[], int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target)
                return i;
        }
        return -1;
    }
}
```

Time Complexity:

Best: O(1)
Average: O(n)
Worst: O(n)

Space Complexity:

O(1)

2. Binary Search

Definition:

Binary Search is an efficient algorithm that works only on sorted arrays. It repeatedly divides the search interval in half.

Process:

1. Start with the middle element.
2. If the target equals the middle element, return its index.
3. If the target is smaller, search the left half.
4. If larger, search the right half.
5. Repeat until found or the search space is empty.

Python Code:

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Java Code:

```
class BinarySearch {
    static int search(int arr[], int target) {
        int low = 0, high = arr.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] == target)
                return mid;
            else if (arr[mid] < target)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return -1;
    }
}
```

Time Complexity:
Best: O(1)
Average: O(log n)
Worst: O(log n)

Space Complexity:
O(1)

3. Insertion Sort

Definition:
Insertion Sort builds the sorted array one element at a time by inserting elements into their correct position.

Process:
1. Start from the second element.
2. Compare with previous elements.
3. Shift all larger elements to the right.
4. Insert the current element in its correct position.
5. Repeat until sorted.

Python Code:
```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

Java Code:
```
class InsertionSort {
    static void sort(int arr[]) {
        for (int i = 1; i < arr.length; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }
}
```

Time Complexity:

Best: O(n)
Average: O(n^2)
Worst: O(n^2)

Space Complexity:

O(1)

4. Bubble Sort

Definition:

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order.

Process:

1. Start from the first element.
2. Compare adjacent pairs.
3. Swap if necessary.
4. Repeat for all elements until sorted.

Python Code:

```python
def bubble_sort(arr):
    for i in range(len(arr)-1):
        for j in range(0, len(arr) - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

Java Code:

```java
class BubbleSort {
    static void sort(int arr[]) {
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = 0; j < arr.length - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
}
```

Time Complexity:

Best: O(n)
Average: O(n^2)
Worst: O(n^2)

Space Complexity:

O(1)

5. Selection Sort

Definition:

Selection Sort repeatedly finds the smallest element from the unsorted portion and moves it to the sorted portion.

Process:

1. Find the smallest element.
2. Swap it with the first unsorted element.
3. Move to the next position.
4. Repeat until sorted.

Python Code:

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

Java Code:

```
class SelectionSort {
    static void sort(int arr[]) {
        for (int i = 0; i < arr.length - 1; i++) {
            int min_idx = i;
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[j] < arr[min_idx]) {
                    min_idx = j;
                }
            }
            int temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
    }
}
```

Time Complexity:

Best: $O(n^2)$
Average: $O(n^2)$
Worst: $O(n^2)$

Space Complexity:

O(1)

6. Quick Sort

Definition:

Quick Sort is a divide-and-conquer algorithm that partitions the array around a pivot element.

Process:

1. Choose a pivot element.
2. Partition the array into two halves: less than pivot and greater than pivot.
3. Recursively sort both halves.
4. Combine results.

Python Code:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr)//2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

Java Code:

```
class QuickSort {
    static void sort(int arr[], int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            sort(arr, low, pi - 1);
            sort(arr, pi + 1, high);
        }
    }
    static int partition(int arr[], int low, int high) {
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        return i + 1;
    }
}
```

Time Complexity:

Best: O(n log n)

Average: O(n log n)

Worst: O(n^2)

Space Complexity:

O(log n)

7. Merge Sort

Definition:

Merge Sort divides the array into halves, sorts each half recursively, and then merges them back together.

Process:

1. Divide the list into two halves.
2. Sort each half recursively.
3. Merge the sorted halves into a single sorted list.

Python Code:

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        merge_sort(L)
        merge_sort(R)
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
        k += 1
    while i < len(L):
        arr[k] = L[i]
        i += 1
        k += 1
    while j < len(R):
        arr[k] = R[j]
        j += 1
        k += 1
```

--

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
merge_sort([10,3,12,9,7,140])
```

Java Code:

```
class MergeSort {
    static void sort(int arr[], int l, int r) {
        if (l < r) {
            int m = (l + r) / 2;
            sort(arr, l, m);
            sort(arr, m + 1, r);
            merge(arr, l, m, r);
        }
    }
    static void merge(int arr[], int l, int m, int r) {
        int n1 = m - l + 1;
        int n2 = r - m;
        int L[] = new int[n1];
        int R[] = new int[n2];
        for (int i = 0; i < n1; i++)
            L[i] = arr[l + i];
        for (int j = 0; j < n2; j++)
            R[j] = arr[m + 1 + j];
        int i = 0, j = 0, k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j])
                arr[k++] = L[i++];
            else
                arr[k++] = R[j++];
        }
        while (i < n1)
            arr[k++] = L[i++];
        while (j < n2)
            arr[k++] = R[j++];
    }
}
```

Time Complexity:

Best: O(n log n)
Average: O(n log n)
Worst: O(n log n)

Space Complexity:

O(n)

8. Heap Sort

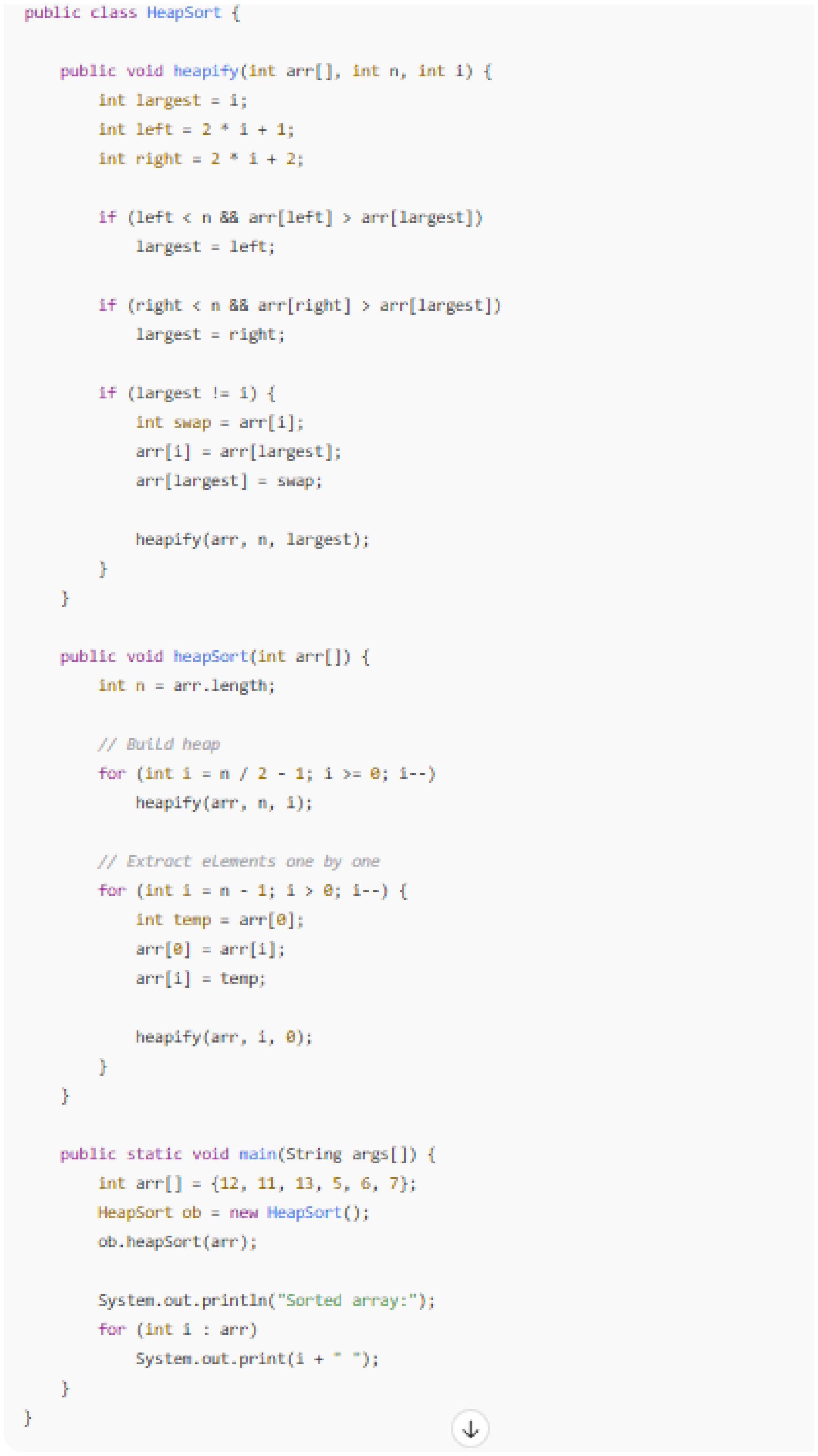
Heap Sort is a **comparison-based sorting algorithm** that uses a **Binary Heap** data structure (usually a **Max Heap**) to sort elements.

It first builds a heap from the input data and then repeatedly extracts the largest element from the heap and rebuilds the heap until the array is sorted.

Process / Algorithm Steps

1. **Build a Max Heap** from the given array.
   (The largest element will be at the root.)
2. **Swap** the root (maximum value) with the last element in the heap.
3. **Reduce the heap size** by 1 and **heapify** the root to restore heap property.
4. Repeat step 2 and 3 until the heap size becomes 1.

```python
def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    left = 2 * i + 1
    right = 2 * i + 2

    # Check if left child is greater than root
    if left < n and arr[left] > arr[largest]:
        largest = left

    # Check if right child is greater than largest
    if right < n and arr[right] > arr[largest]:
        largest = right

    # If largest is not root, swap and continue heapifying
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def heap_sort(arr):
    n = len(arr)

    # Step 1: Build a Max Heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Step 2: Extract elements from heap one by one
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]  # swap
        heapify(arr, i, 0)

    return arr


# Example usage
arr = [12, 11, 13, 5, 6, 7]
heap_sort(arr)
print("Sorted array:", arr)
```



```java
public class HeapSort {

    public void heapify(int arr[], int n, int i) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < n && arr[left] > arr[largest])
            largest = left;

        if (right < n && arr[right] > arr[largest])
            largest = right;

        if (largest != i) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

            heapify(arr, n, largest);
        }
    }

    public void heapSort(int arr[]) {
        int n = arr.length;

        // Build heap
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        // Extract elements one by one
        for (int i = n - 1; i > 0; i--) {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            heapify(arr, i, 0);
        }
    }

    public static void main(String args[]) {
        int arr[] = {12, 11, 13, 5, 6, 7};
        HeapSort ob = new HeapSort();
        ob.heapSort(arr);

        System.out.println("Sorted array:");
        for (int i : arr)
            System.out.print(i + " ");
    }
}
```

Time Complexity:

Best: O(n log n)
Average: O(n log n)
Worst: O(n log n)

Space Complexity:

O(1)