Sorting Algorithms

Notes (Nishant)

November 27, 2024

Contents

1 Selection Sort

1.1 Intuition

The selection sort algorithm sorts an array by repeatedly finding the minimum element from the unsorted part and putting it at the beginning. The largest element will end up at the last index of the array.

1.2 Approach

1. Select the starting index of the unsorted part using a loop with i from 0 to $n-1$.
2. Find the smallest element in the range from i to $n-1$ using an inner loop.
3. Swap this smallest element with the element at index i.
4. Repeat the process for the next starting index.

1.3 Code

1.3.1 Java Code

```java
import java.util.Arrays;

class Solution {

    public int[] selectionSort(int[] nums) {
        // Loop through unsorted part of the array (0 to n-2)
        for (int i = 0; i < nums.length - 1; i++) {
            /* Assume current element
               is the minimum */
            int minIndex = i;

            // Find actual minimum in unsorted part (i+1 to n-1)
            for (int j = i + 1; j < nums.length; j++) {
                if (nums[j] < nums[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap only if minIndex changed
            if (minIndex != i) {
                int temp = nums[i];
                nums[i] = nums[minIndex];
                nums[minIndex] = temp;
            }
        }

        return nums;
    }

    public static void main(String[] args) {
        int[] arr = {7, 5, 9, 2, 8};

        System.out.print("Original array: ");
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();

        // Create an instance of Solution class
        Solution solution = new Solution();

        // Function call for selection sort
        int[] sortedArr = solution.selectionSort(arr);

        System.out.print("Sorted array: ");
        for (int num : sortedArr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```

Listing 1: Java: Selection Sort Solution

1.4 Complexity Analysis

- **Time Complexity**: $O(n^2)$
- **Space Complexity**: $O(1)$

1.5 Limitations

Selection Sort is inefficient for large datasets due to its quadratic time complexity. It performs poorly on large lists and is generally not suitable for practical use compared to more efficient algorithms like Quick Sort or Merge Sort.

2 Bubble Sort

2.1 Intuition

The bubble sort algorithm sorts an array by repeatedly swapping adjacent elements if they are in the wrong order. The largest elements ”bubble” to the end of the array with each pass.

2.2 Approach

1. Run a loop i from 0 to $n-1$.
2. Run a nested loop from j from 0 to $n-i-1$.

3. If `arr[j]` > `arr[j+1]`, swap them.
4. Continue until the array is sorted.

2.3 Code

2.3.1 Java Code

```java
import java.util.Arrays;

class Solution {

    // Bubble Sort
    public int[] bubbleSort(int[] arr) {
        int n = arr.length;
        boolean swapped;

        // Traverse through all array elements
        for (int i = 0; i < n - 1; i++) {
            swapped = false;

            // Last i elements are already in place
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap arr[j] and arr[j+1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }

            // If no two elements were swapped by inner loop, then break
            if (!swapped)
                break;
        }

        return arr;
    }

    public static void main(String[] args) {
        int[] arr = {7, 5, 9, 2, 8};

        System.out.print("Original␣array:␣");
        for (int num : arr) {
            System.out.print(num + "␣");
        }
        System.out.println();

        // Create an instance of Solution class
        Solution solution = new Solution();

        // Function call for bubble sort
        int[] sortedArr = solution.bubbleSort(arr);

        System.out.print("Sorted␣array:␣");
        for (int num : sortedArr) {
            System.out.print(num + "␣");
        }
        System.out.println();
    }
}
```

Listing 2: Java: Bubble Sort Solution

2.4 Complexity Analysis

- **Time Complexity**: $O(n^2)$
- **Space Complexity**: $O(1)$

2.5 Advantages

Bubble Sort is simple to implement and understand. It can be efficient for small datasets and can detect if the array is already sorted, allowing for early termination.

2.6 Limitations

Bubble Sort is inefficient for large datasets due to its quadratic time complexity. It generally performs worse than more efficient algorithms like Quick Sort or Merge Sort.

3 Insertion Sort

3.1 Intuition

Insertion sort builds a sorted array one element at a time by repeatedly picking the next element and inserting it into its correct position within the already sorted part of the array.

3.2 Approach

1. In each iteration, select an element from the unsorted part of the array using an outer loop.
2. Place this element in its correct position within the sorted part of the array.
3. Use an inner loop to shift the remaining elements as necessary to accommodate the selected element. This involves swapping elements until the selected element is in its correct position.
4. Continue this process until the entire array is sorted.

3.3 Code

3.3.1 Java Code

```java
import java.util.Arrays;

class Solution {
    // Insertion Sort
    public int[] insertionSort(int[] nums) {
        int n = nums.length;
        // Traverse through the array
        for (int i = 1; i < n; i++) {
            int key = nums[i];
            int j = i - 1;

            // Move elements of nums[0..i-1], that are greater than key, to one position ahead
            while (j >= 0 && nums[j] > key) {
                nums[j + 1] = nums[j];
                j = j - 1;
            }
            nums[j + 1] = key;
        }
        return nums;
    }

    public static void main(String[] args) {
        // Create an instance of Solution class
        Solution solution = new Solution();

        int[] nums = {13, 46, 24, 52, 20, 9};

        System.out.println("Before␣Using␣Insertion␣Sort:␣" + Arrays.toString(nums));

        // Function call for insertion sort
        nums = solution.insertionSort(nums);

        System.out.println("After␣Using␣Insertion␣Sort:␣" + Arrays.toString(nums));
    }
}
```

Listing 3: Java: Insertion Sort Solution

3.4 Complexity Analysis

- **Time Complexity**: $O(n^2)$
- **Space Complexity**: $O(1)$

3.5 Advantages

Insertion Sort is efficient for small datasets and nearly sorted arrays. It is stable and in-place, requiring only a constant amount of additional memory space.

3.6 Limitations

Insertion Sort is inefficient for large datasets due to its quadratic time complexity. It performs poorly compared to more advanced algorithms like Quick Sort or Merge Sort.

4 Understanding Recursion in Data Structures and Algorithms

4.1 Function Overview

A function is a reusable block of code designed to perform a specific task. It can take input (parameters) and may return a result. Functions allow us to break down complex problems into smaller, manageable pieces.

4.2 Recursion

Recursion occurs when a function calls itself directly or indirectly to solve a problem. It is an elegant approach to handle problems that can be broken down into smaller, similar subproblems.

4.3 Infinite Recursion

Infinite recursion happens when a function does not have a base condition to stop the recursive calls. This leads to the function calling itself indefinitely, eventually causing a stack overflow.

```java
class Main {
    static void infiniteRecursion() {
        System.out.println("Calling␣function");
        infiniteRecursion();
    }

    public static void main(String[] args) {
        // Uncommenting this will cause infinite recursion
        // infiniteRecursion();
    }
}
```

Listing 4: Java: Infinite Recursion Example

4.4 Base Case/Condition

The base case is a stopping condition in recursive functions that prevents infinite recursion. It defines the simplest instance of the problem that can be solved without further recursion.

Note: Writing effective recursion involves defining a base case or condition that ensures the recursion terminates. Without a base case, recursion will continue indefinitely, leading to stack overflow.

4.5 Recursive Stack Space

Each time a function calls itself, a new frame is added to the function call stack. The stack keeps track of the current function execution. When a base condition is met, the stack starts unwinding, returning the results in reverse order.

4.6 Program Flow in Recursion

When a recursive function is called, a new instance of that function is created and the control is passed to it until it hits the base case. Each recursive call adds a new frame to the stack. Once the base case is reached, the stack starts unwinding, returning the results step by step.

4.7 Types of Recursion

4.7.1 Head Recursion

In head recursion, the recursive call occurs before any other processing in the function. The function waits for the recursive call to return before proceeding with any operation.

Example: The following code prints numbers from 1 to 5 using head recursion.

```java
class Main {
    static void headRecursion(int n) {
        if (n > 0) {
            headRecursion(n - 1);  // Recursive call before processing
            System.out.print(n + "␣");  // Processing after recursion
        }
    }

    public static void main(String[] args) {
        headRecursion(5);
    }
}
```

Listing 5: Java: Head Recursion Example

4.7.2 Tail Recursion

In tail recursion, the recursive call is the last operation in the function. Once the function calls itself, there is no need to retain the current function's state, allowing the compiler to optimize tail recursion.

Example: The following code prints numbers from 5 to 1 using tail recursion.

```java
class Main {
    static void tailRecursion(int n) {
        if (n == 0)
            return;
        System.out.print(n + "␣");  // Processing before recursion
        tailRecursion(n - 1);       // Recursive call is the last action
    }

    public static void main(String[] args) {
        tailRecursion(5);
    }
}
```

Listing 6: Java: Tail Recursion Example

4.8 Stack Overflow

Any local machine has limited resources. Stack overflow occurs when too many recursive calls are made without a base case, or the recursion depth exceeds the system's call stack limit. This causes the program to crash as the system runs out of stack space.

4.9 Recursion Tree

A recursion tree is a visual representation that helps understand the flow of recursive calls. It shows how the problem is divided into smaller subproblems at each recursive step.

1. **Recursion Tree for Head Recursion**: In head recursion, the tree grows downward as the function waits for each recursive call to complete before executing the remaining operations.
2. **Recursion Tree for Tail Recursion**: In tail recursion, the recursion tree is simpler since each recursive call is the last operation, leading to more straightforward unwinding of the stack.

4.10 Time Complexity

The time complexity of a recursive function is generally based on the number of recursive calls made. If a function makes one recursive call, the time complexity is $O(n)$, where n is the depth of the recursion.

4.11 Space Complexity

The space complexity of a recursive function is determined by the maximum depth of the recursive call stack. If the function reaches a maximum recursion depth of n, the space complexity is $O(n)$.

4.12 Recursion Concepts with Parameters

[Details to be filled based on specific content or examples.]

5 Recursion Example: Factorial

5.1 Intuition

Finding the factorial of a number N using recursion involves visualizing the process of multiplying the number by each smaller positive integer down to 1. The approach is to repeatedly call the function, reducing the number by 1 each time. This continues until reaching the base case of 1. Imagine the factorial as a series of multiplications: starting with N, then multiplying by $N-1$, then by $N-2$, and so on, until multiplying by 1.

5.2 Approach

1. Define a function that calculates the factorial of a number.
2. In the function, if the number is 0 or 1, return 1 (base case).
3. Otherwise, return the number multiplied by the factorial of the number minus 1 (recursive case).

5.3 Code

5.3.1 Java Code

```java
class Solution {
    /**
     * Calculates the factorial of a number using recursion.
     *
     * @param n The number to calculate the factorial for.
     * @return The factorial of the number.
     */
    public long factorial(int n) {
        // Base case: factorial of 0 or 1 is 1
        if (n <= 1) return 1;
        // Recursive case: n * factorial of n-1
        return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int N = 5; // Example input
        System.out.println("Factorial␣of␣" + N + "␣is␣" + solution.factorial(N));
    }
}
```

Listing 7: Java: Recursive Factorial Solution

5.3.2 Python Code

```python
class Solution:
    def factorial(self, n):
        """
␣␣␣␣␣␣␣␣Calculates␣the␣factorial␣of␣a␣number␣using␣recursion.

␣␣␣␣␣␣␣␣:param␣n:␣int␣-␣The␣number␣to␣calculate␣the␣factorial␣for.
␣␣␣␣␣␣␣␣:return:␣int␣-␣The␣factorial␣of␣the␣number.
␣␣␣␣␣␣␣␣"""
        # Base case: factorial of 0 or 1 is 1
        if n <= 1:
            return 1
        # Recursive case: n * factorial of n-1
        return n * self.factorial(n - 1)

if __name__ == "__main__":
    solution = Solution()
    N = 5  # Example input
    print(f"Factorial␣of␣{N}␣is␣{solution.factorial(N)}")
```

Listing 8: Python: Recursive Factorial Solution

5.4 Complexity Analysis

- **Time Complexity**: $O(n)$
- **Space Complexity**: $O(n)$

5.5 Limitations

Recursion can lead to stack overflow if the recursion depth is too high. Iterative solutions are often preferred for large inputs to avoid excessive stack usage.

6 Merge Sort

6.1 Intuition

Merge Sort is a powerful sorting algorithm that follows the divide-and-conquer approach. The array is divided into two equal halves until each sub-array contains only one element. Each pair of smaller sorted arrays is then merged into a larger sorted array.

The algorithm consists of two main functions:

- **merge()**: This function merges the two halves of the array, assuming both parts are already sorted.
- **mergeSort()**: This function divides the array into 2 parts: low to mid and mid+1 to high, where low is the leftmost index of the array, high is the rightmost index of the array, and mid is the middle index of the array.

By repeating these steps recursively, Merge Sort efficiently sorts the entire array.

6.2 Approach

To implement Merge Sort, we will create two functions: `mergeSort()` and `merge()`.

mergeSort(arr[], low, high)

1. **Divide the Array**: Split the given array into two halves by splitting the range. For any range from low to high, the splits will be low to mid and mid+1 to high, where mid = $\lfloor \frac{low+high}{2} \rfloor$. This process continues until the range size is 1.
2. **Recursive Division**: In `mergeSort()`, divide the array around the middle index by making recursive calls: `mergeSort(arr, low, mid)` for the left half and `mergeSort(arr, mid+1, high)` for the right half.

3. **Base Case**: Define the base case where if low >= high, the function returns as the array is already sorted.

merge(arr[], low, mid, high)

1. **Merge the Two Halves**: Use a temporary array to store the elements of the two sorted halves after merging. The range of the left half is from low to mid and the range of the right half is from mid+1 to high.

2. **Two Pointers**: Use two pointers, left starting from low and right starting from mid+1. Compare the elements from each half and insert the smaller one into the temporary array.

3. **Copy Remaining Elements**: After the loop, copy any remaining elements from both halves into the temporary array.

4. **Transfer to Original Array**: Transfer the elements from the temporary array back to the original array in the range low to high.

This approach ensures that the array is efficiently sorted using the divide-and-conquer strategy of Merge Sort.

6.3 Code

6.3.1 Java Code

```java
import java.util.*;

class Solution {
    // Function to merge two sorted halves of the array
    public void merge(int[] arr, int low, int mid, int high) {
        // Temporary array to store merged elements
        List<Integer> temp = new ArrayList<>();
        int left = low;
        int right = mid + 1;

        // Loop until one of the subarrays is exhausted
        while (left <= mid && right <= high) {
            // Compare left and right elements
            if (arr[left] <= arr[right]) {
                // Add left element to temp
                temp.add(arr[left]);
                // Move left pointer
                left++;
            } else {
                // Add right element to temp
                temp.add(arr[right]);
                // Move right pointer
                right++;
            }
        }

        // Adding the remaining elements of left half
        while (left <= mid) {
            temp.add(arr[left]);
            left++;
        }

        // Adding the remaining elements of right half
        while (right <= high) {
            temp.add(arr[right]);
            right++;
        }

        // Transferring the sorted elements to arr
        for (int i = low; i <= high; i++) {
            arr[i] = temp.get(i - low);
        }
    }

    // Helper Function to perform the recursive merge sort
    public void mergeSortHelper(int[] arr, int low, int high) {
        /* Base case: If the array has only one element */
        if (low >= high)
            return;

        // Find the middle index
        int mid = (low + high) / 2;

        // Recursively sort the left half
        mergeSortHelper(arr, low, mid);
        // Recursively sort the right half
        mergeSortHelper(arr, mid + 1, high);
        // Merge the sorted halves
        merge(arr, low, mid, high);
    }

    // Function to perform merge sort on the given array
    public int[] mergeSort(int[] nums) {
        int n = nums.length; // Size of array

        // Perform Merge sort on the whole array
        mergeSortHelper(nums, 0, n - 1);

        // Return the sorted array
        return nums;
    }

    public static void main(String[] args) {
        int[] arr = {9, 4, 7, 6, 3, 1, 5};
        int n = arr.length;

        System.out.println("Before␣Sorting␣Array:");
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + "␣");
        System.out.println();

        // Create an instance of Solution class
        Solution sol = new Solution();
        // Function call to sort the array using merge sort
        int[] sortedArr = sol.mergeSort(arr);

        System.out.println("After␣Sorting␣Array:");
        for (int i = 0; i < n; i++)
            System.out.print(sortedArr[i] + "␣");
        System.out.println();
    }
}
```

Listing 9: Java: Merge Sort Solution

6.4 Complexity Analysis

- **Time Complexity**: $O(n \log n)$. At each step, we divide the whole array, which takes $\log n$ steps, and we assume n steps are taken to sort the array.
- **Space Complexity**: $O(n)$. We are using a temporary array to store elements in sorted order.

6.5 Advantages

Merge Sort is efficient and has a predictable time complexity of $O(n \log n)$. It is stable and works well for large datasets. Additionally, it performs well on linked lists and can be easily parallelized.

6.6 Limitations

Merge Sort requires additional space proportional to the size of the input array, which can be a drawback for large datasets. It is also not an in-place sorting algorithm.

7 Quick Sort

7.1 Intuition

Quick Sort is a divide-and-conquer algorithm like Merge Sort. However, unlike Merge Sort, Quick Sort does not use an extra array for sorting (though it uses auxiliary stack space). This makes Quick Sort slightly better than Merge Sort from a space perspective.

This algorithm follows two simple steps repeatedly:

1. **Pick a Pivot**: Select a pivot element and place it in its correct position in the sorted array.
2. **Partition**: Move smaller elements (i.e., smaller than the pivot) to the left of the pivot and larger ones to the right.

The main goal is to place the pivot at its final position in each recursion call, where it should be in the final sorted array.

7.2 Approach

To implement Quick Sort, we will create two functions: `quickSort()` and `partition()`.

quickSort(arr[], low, high)

1. **Initial Setup**: The low pointer points to the first index, and the high pointer points to the last index of the array.
2. **Partitioning**: Use the `partition()` function to get the index where the pivot should be placed after sorting. This index, called the partition index, separates the left and right unsorted subarrays.
3. **Recursive Calls**: After placing the pivot at the partition index, recursively call `quickSort()` for the left and right subarrays. The range of the left subarray will be [low to partition index - 1] and the range of the right subarray will be [partition index + 1 to high].
4. **Base Case**: The recursion continues until the range becomes 1.

partition(arr[], low, high)

1. **Select Pivot**: Choose the first element as pivot (arr[low]).
2. **Initialize Pointers**: Use pointers i (low) and j (high).
3. **Rearrange Elements**:
   - Move i forward until an element greater than the pivot is found.
   - Move j backward until an element smaller than the pivot is found.
   - If $i < j$, swap arr[i] and arr[j].
4. **Place Pivot in Correct Position**: Swap pivot (arr[low]) with arr[j] and return j as the partition index.

This approach ensures that Quick Sort efficiently sorts the array using the divide-and-conquer strategy.

7.3 Code

7.3.1 Java Code

```java
import java.util.Arrays;

class Solution {
    // Function to partition the array
    public int partition(int[] arr, int low, int high) {

        // Choosing the first element as pivot
        int pivot = arr[low];
        // Starting index for left subarray
        int i = low;
        // Starting index for right subarray
        int j = high;

        while (i < j) {
            /* Move i to the right until we find an
               element greater than the pivot */
            while (arr[i] <= pivot && i <= high - 1) {
                i++;
            }
            /* Move j to the left until we find an
               element smaller than the pivot */
            while (arr[j] > pivot && j >= low + 1) {
                j--;
            }
            /* Swap elements at i and j if i is still
               less than j */
            if (i < j) {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // Pivot placed in correct position
        int temp = arr[low];
        arr[low] = arr[j];
        arr[j] = temp;
        return j;
    }

    // Helper Function to perform the recursive quick sort
    public void quickSortHelper(int[] arr, int low, int high) {
        /* Base case: If the array has one or no
           elements, it's already sorted */
        if (low < high) {
            // Get the partition index
            int pIndex = partition(arr, low, high);
            // Sort the left subarray
            quickSortHelper(arr, low, pIndex - 1);
            // Sort the right subarray
            quickSortHelper(arr, pIndex + 1, high);
        }
    }

    // Function to perform quick sort on given array
    public int[] quickSort(int[] nums) {
        // Get the size of array
        int n = nums.length;

        // Perform quick sort
        quickSortHelper(nums, 0, n - 1);

        // Return sorted array
        return nums;
    }

    public static void main(String[] args) {
        int[] arr = {4, 6, 2, 5, 7, 9, 1, 3};
        int n = arr.length;

        System.out.println("Before␣Sorting␣Array:");
        for (int i = 0; i < n; i++) {
            System.out.print(arr[i] + "␣");
        }
        System.out.println();

        // Create an instance of Solution class
        Solution solution = new Solution();

        // Function call to sort the array using quick sort
        int[] sortedArr = solution.quickSort(arr);

        System.out.println("After␣Sorting␣Array:");
        for (int i = 0; i < n; i++) {
            System.out.print(sortedArr[i] + "␣");
        }
        System.out.println();
    }
}
```

Listing 10: Java: Quick Sort Solution

7.4 Complexity Analysis

- **Time Complexity**: $O(n \log n)$ on average, $O(n^2)$ in the worst case.
- **Space Complexity**: $O(\log n)$ due to recursive stack space.

7.5 Advantages

Quick Sort is generally faster in practice compared to other $O(n \log n)$ algorithms like Merge Sort and Heap Sort due to better cache performance and low constant factors. It is an in-place sorting algorithm with efficient average-case performance.

7.6 Limitations

Quick Sort has a worst-case time complexity of $O(n^2)$, which occurs when the smallest or largest element is always chosen as the pivot. This can be mitigated by choosing a random pivot or using the median-of-three method. Additionally, Quick Sort is not stable by default.

8 Conclusion/Summary

In this section, we explored five fundamental sorting algorithms: **Selection Sort**, **Bubble Sort**, **Insertion Sort**, **Merge Sort**, and **Quick Sort**. Each algorithm follows a different approach to sorting, with varying time and space complexities.

1. **Selection Sort**:
    - **Time Complexity**: $O(n^2)$
    - **Space Complexity**: $O(1)$
    - **Pros**: Simple to implement.
    - **Cons**: Inefficient for large datasets.
2. **Bubble Sort**:
    - **Time Complexity**: $O(n^2)$
    - **Space Complexity**: $O(1)$
    - **Pros**: Simple and can detect already sorted arrays.
    - **Cons**: Highly inefficient for large datasets.
3. **Insertion Sort**:
    - **Time Complexity**: $O(n^2)$
    - **Space Complexity**: $O(1)$
    - **Pros**: Efficient for small or nearly sorted datasets.
    - **Cons**: Inefficient for large datasets.
4. **Merge Sort**:
    - **Time Complexity**: $O(n \log n)$
    - **Space Complexity**: $O(n)$
    - **Pros**: Efficient and stable; works well for large datasets.
    - **Cons**: Requires additional space.
5. **Quick Sort**:
    - **Time Complexity**: $O(n \log n)$ on average, $O(n^2)$ in the worst case.
    - **Space Complexity**: $O(\log n)$
    - **Pros**: Generally faster in practice; in-place.
    - **Cons**: Not stable by default; worst-case performance.

8.1 Summary Tables of Solutions

Table 1: Comparison of Sorting Algorithms

Algorithm	Time Complexity	Space Complexity	Pros & Cons
Selection Sort	$O(n^2)$	$O(1)$	Simple to implement; Inefficient for large datasets
Bubble Sort	$O(n^2)$	$O(1)$	Simple; Can detect already sorted arrays; Highly inefficient
Insertion Sort	$O(n^2)$	$O(1)$	Efficient for small/nearly sorted datasets; Inefficient for large datasets
Merge Sort	$O(n \log n)$	$O(n)$	Efficient and stable; Requires additional space
Quick Sort	$O(n \log n)$ average	$O(\log n)$	Fast in practice; In-place; Not stable; Worst-case $O(n^2)$

8.2 Summary Table of Comparison between Java and Python Implementations

Table 2: Comparison of Java and Python Implementations for Sorting Algorithms

Feature	Java	Python
Data Structures Used	Arrays	Lists
Loop Constructs	For loops	For loops with range or iteration over elements
Handling Edge Cases	Conditional checks for array boundaries	Conditional checks for array boundaries
Function Definition	Methods inside Class	Methods inside Class with docstrings
Recursive Calls	Explicit recursive functions	Explicit recursive functions

8.3 Additional Insights

- **Early Termination**: In algorithms like Bubble Sort, if no swaps are made during a pass, the algorithm can terminate early, enhancing efficiency.
- **In-Place Sorting**: Quick Sort and Selection Sort are in-place sorting algorithms, requiring minimal additional memory.
- **Stability**: Merge Sort is a stable sort, meaning it preserves the relative order of equal elements, whereas Quick Sort is not stable by default.
- **Recursion Stack Space**: Recursive algorithms like Merge Sort and Quick Sort use additional stack space, which can be optimized in some cases.
- **Pivot Selection in Quick Sort**: Choosing a good pivot is crucial for Quick Sort's performance. Randomized pivot selection or the median-of-three method can help avoid worst-case scenarios.

9 References

- GeeksforGeeks: Selection Sort
- GeeksforGeeks: Bubble Sort
- GeeksforGeeks: Insertion Sort
- GeeksforGeeks: Merge Sort

- GeeksforGeeks: Quick Sort
- Wikipedia: Recursion (Computer Science)