**Day 16 - 11th July 2025**

**Selection Sort:**

**Task 1:**

Write an algorithm / steps for selection sort.

**Answer:**

Selection Sort Algorithm:

Step 1: Set minimum index (min) to location 0

Step 2: Search for the minimum element in the array from index 0 to n-1

Step 3: Swap the value at location min with the minimum element found

Step 4: Increment min to point to the next element (min = min + 1)

Step 5: Repeat steps 2-4 until the entire list is sorted

Step 6: Return the sorted array

Detailed Steps:

1. Start with the first element as the current minimum

2. Compare this minimum with the next element

3. If the next element is smaller, make it the new minimum

4. Continue until you find the smallest element in the remaining unsorted portion

5. Swap the smallest element with the first element of the unsorted portion

6. Move to the next position and repeat the process

7. Continue until all elements are in their correct positions

**Task 2:**

Write a pseudo code for the selection sort

**Answer:**

SELECTION\_SORT(A)

BEGIN

n = length of A

FOR i = 0 TO n-2 DO

min\_j = i

min\_x = A[i]

FOR j = i+1 TO n-1 DO

IF A[j] < min\_x THEN

min\_j = j

min\_x = A[j]

END IF

END FOR

// Swap elements

A[min\_j] = A[i]

A[i] = min\_x

END FOR

END

Alternative Pseudo Code:

SELECTION\_SORT(arr[], n)

BEGIN

FOR i = 0 TO n-2 DO

min\_index = i

FOR j = i+1 TO n-1 DO

IF arr[j] < arr[min\_index] THEN

min\_index = j

END IF

END FOR

SWAP(arr[i], arr[min\_index])

END FOR

END

**Task 3:**

WAP to make sure your list is sorted using selection sort.

**Answer:**

import java.util.Arrays;

public class SelectionSort {

// Selection sort implementation

public static void selectionSort(int[] arr) {

int n = arr.length;

// One by one move boundary of unsorted subarray

for (int i = 0; i < n - 1; i++) {

// Find the minimum element in remaining unsorted array

int minIndex = i;

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

// Swap the found minimum element with the first element

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

// Display array after each pass

System.out.println("Pass " + (i + 1) + ": " + Arrays.toString(arr));

}

}

// Method to verify if array is sorted

public static boolean isSorted(int[] arr) {

for (int i = 0; i < arr.length - 1; i++) {

if (arr[i] > arr[i + 1]) {

return false;

}

}

return true;

}

// Method to display array

public static void displayArray(int[] arr, String message) {

System.out.println(message + Arrays.toString(arr));

}

public static void main(String[] args) {

System.out.println("=== Selection Sort Implementation ===\n");

// Test case 1

int[] arr1 = {10, 20, 5, 46, 80};

System.out.println("Test Case 1:");

displayArray(arr1, "Before Sorting: ");

System.out.println("Is sorted before: " + isSorted(arr1));

selectionSort(arr1);

displayArray(arr1, "After Sorting: ");

System.out.println("Is sorted after: " + isSorted(arr1));

// Test case 2

System.out.println("\nTest Case 2:");

int[] arr2 = {64, 25, 12, 22, 11};

displayArray(arr2, "Before Sorting: ");

System.out.println("Is sorted before: " + isSorted(arr2));

selectionSort(arr2);

displayArray(arr2, "After Sorting: ");

System.out.println("Is sorted after: " + isSorted(arr2));

// Test case 3 - Already sorted array

System.out.println("\nTest Case 3 - Already Sorted:");

int[] arr3 = {1, 2, 3, 4, 5};

displayArray(arr3, "Before Sorting: ");

System.out.println("Is sorted before: " + isSorted(arr3));

selectionSort(arr3);

displayArray(arr3, "After Sorting: ");

System.out.println("Is sorted after: " + isSorted(arr3));

// Test case 4 - Reverse sorted array

System.out.println("\nTest Case 4 - Reverse Sorted:");

int[] arr4 = {5, 4, 3, 2, 1};

displayArray(arr4, "Before Sorting: ");

System.out.println("Is sorted before: " + isSorted(arr4));

selectionSort(arr4);

displayArray(arr4, "After Sorting: ");

System.out.println("Is sorted after: " + isSorted(arr4));

}

}

/\* Output:

=== Selection Sort Implementation ===

Test Case 1:

Before Sorting: [10, 20, 5, 46, 80]

Is sorted before: false

Pass 1: [5, 20, 10, 46, 80]

Pass 2: [5, 10, 20, 46, 80]

Pass 3: [5, 10, 20, 46, 80]

Pass 4: [5, 10, 20, 46, 80]

After Sorting: [5, 10, 20, 46, 80]

Is sorted after: true

Test Case 2:

Before Sorting: [64, 25, 12, 22, 11]

Is sorted before: false

Pass 1: [11, 25, 12, 22, 64]

Pass 2: [11, 12, 25, 22, 64]

Pass 3: [11, 12, 22, 25, 64]

Pass 4: [11, 12, 22, 25, 64]

After Sorting: [11, 12, 22, 25, 64]

Is sorted after: true

\*/

**Bubble Sort:**

**Task 4:**

Write algorithm for the Bubble sort.

**Answer:**

Bubble Sort Algorithm:

Step 1: Start with the first element of the array

Step 2: Compare adjacent elements (current and next)

Step 3: If the current element is greater than the next element, swap them

Step 4: Continue comparing and swapping adjacent elements until the end of array

Step 5: After one complete pass, the largest element will be at the end

Step 6: Repeat the process for the remaining unsorted portion (excluding the last sorted elements)

Step 7: Continue until no swaps are needed in a complete pass

Step 8: Return the sorted array

Detailed Algorithm:

1. Set flag = true (to track if any swaps occurred)

2. While flag is true:

a. Set flag = false

b. For i = 0 to n-2:

- If arr[i] > arr[i+1]:

\* Swap arr[i] and arr[i+1]

\* Set flag = true

3. Array is now sorted

Sequential-Bubble-Sort Algorithm:

FOR i = 1 TO n

FOR j = n DOWN TO i+1

IF A[j] < A[j-1] THEN

Exchange A[j] ⟷ A[j-1]

END IF

END FOR

END FOR

**Task 5:**

Write pseudo code for the bubble sort

**Answer:**

BUBBLE\_SORT(num[], sizeofarray)

BEGIN

FOR i = (sizeofarray - 1) DOWN TO 0 DO

FOR j = 1 TO i DO

IF num[j-1] > num[j] THEN

temp = num[j-1]

num[j-1] = num[j]

num[j] = temp

END IF

END FOR

END FOR

END

Alternative Optimized Pseudo Code:

BUBBLE\_SORT\_OPTIMIZED(arr[], n)

BEGIN

FOR i = 0 TO n-1 DO

swapped = FALSE

FOR j = 0 TO n-i-2 DO

IF arr[j] > arr[j+1] THEN

SWAP(arr[j], arr[j+1])

swapped = TRUE

END IF

END FOR

IF swapped = FALSE THEN

BREAK // Array is sorted

END IF

END FOR

END

Simple Version:

BUBBLE\_SORT(A[], n)

BEGIN

FOR i = 0 TO n-2 DO

FOR j = 0 TO n-i-2 DO

IF A[j] > A[j+1] THEN

SWAP(A[j], A[j+1])

END IF

END FOR

END FOR

END

**Task 6:**

WAP to make sure your list is sorted using Bubble sort.

**Answer:**

import java.util.Arrays;

public class BubbleSort {

// Basic bubble sort implementation

public static void bubbleSort(int[] arr) {

int n = arr.length;

System.out.println("Starting Bubble Sort...\n");

for (int i = 0; i < n - 1; i++) {

System.out.println("Pass " + (i + 1) + ":");

boolean swapped = false;

for (int j = 0; j < n - i - 1; j++) {

System.out.print("Comparing " + arr[j] + " and " + arr[j + 1] + ": ");

if (arr[j] > arr[j + 1]) {

// Swap elements

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapped = true;

System.out.println("Swapped -> " + Arrays.toString(arr));

} else {

System.out.println("No swap needed");

}

}

System.out.println("End of pass " + (i + 1) + ": " + Arrays.toString(arr));

// If no swapping happened, array is sorted

if (!swapped) {

System.out.println("No swaps in this pass - Array is sorted!");

break;

}

System.out.println();

}

}

// Optimized bubble sort

public static void bubbleSortOptimized(int[] arr) {

int n = arr.length;

for (int i = 0; i < n - 1; i++) {

boolean swapped = false;

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// Swap elements

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapped = true;

}

}

// Early termination if array is sorted

if (!swapped) {

break;

}

}

}

// Method to verify if array is sorted

public static boolean isSorted(int[] arr) {

for (int i = 0; i < arr.length - 1; i++) {

if (arr[i] > arr[i + 1]) {

return false;

}

}

return true;

}

// Method to display array

public static void displayArray(int[] arr, String message) {

System.out.println(message + Arrays.toString(arr));

}

// Method to count comparisons and swaps

public static void bubbleSortWithStats(int[] arr) {

int n = arr.length;

int comparisons = 0;

int swaps = 0;

for (int i = 0; i < n - 1; i++) {

boolean swapped = false;

for (int j = 0; j < n - i - 1; j++) {

comparisons++;

if (arr[j] > arr[j + 1]) {

// Swap elements

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swaps++;

swapped = true;

}

}

if (!swapped) {

break;

}

}

System.out.println("Statistics:");

System.out.println("Total Comparisons: " + comparisons);

System.out.println("Total Swaps: " + swaps);

}

public static void main(String[] args) {

System.out.println("=== Bubble Sort Implementation ===\n");

// Test case 1 - Detailed demonstration

int[] arr1 = {10, 20, 5, 46, 80};

System.out.println("Test Case 1 - Detailed:");

displayArray(arr1, "Before Sorting: ");

System.out.println("Is sorted before: " + isSorted(arr1));

System.out.println();

bubbleSort(arr1.clone()); // Use clone to keep original for stats

System.out.println("\nRunning optimized version for statistics:");

int[] arr1Copy = arr1.clone();

bubbleSortOptimized(arr1Copy);

displayArray(arr1Copy, "After Sorting: ");

System.out.println("Is sorted after: " + isSorted(arr1Copy));

bubbleSortWithStats(arr1.clone());

// Test case 2 - Already sorted

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 2 - Already Sorted Array:");

int[] arr2 = {1, 2, 3, 4, 5};

displayArray(arr2, "Before Sorting: ");

System.out.println("Is sorted before: " + isSorted(arr2));

bubbleSortOptimized(arr2);

displayArray(arr2, "After Sorting: ");

System.out.println("Is sorted after: " + isSorted(arr2));

bubbleSortWithStats(new int[]{1, 2, 3, 4, 5});

// Test case 3 - Reverse sorted

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 3 - Reverse Sorted Array:");

int[] arr3 = {5, 4, 3, 2, 1};

displayArray(arr3, "Before Sorting: ");

System.out.println("Is sorted before: " + isSorted(arr3));

bubbleSortOptimized(arr3);

displayArray(arr3, "After Sorting: ");

System.out.println("Is sorted after: " + isSorted(arr3));

bubbleSortWithStats(new int[]{5, 4, 3, 2, 1});

// Test case 4 - Array with duplicates

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 4 - Array with Duplicates:");

int[] arr4 = {3, 1, 4, 1, 5, 9, 2, 6, 5};

displayArray(arr4, "Before Sorting: ");

System.out.println("Is sorted before: " + isSorted(arr4));

bubbleSortOptimized(arr4);

displayArray(arr4, "After Sorting: ");

System.out.println("Is sorted after: " + isSorted(arr4));

bubbleSortWithStats(new int[]{3, 1, 4, 1, 5, 9, 2, 6, 5});

}

}

/\* Output:

=== Bubble Sort Implementation ===

Test Case 1 - Detailed:

Before Sorting: [10, 20, 5, 46, 80]

Is sorted before: false

Starting Bubble Sort...

Pass 1:

Comparing 10 and 20: No swap needed

Comparing 20 and 5: Swapped -> [10, 5, 20, 46, 80]

Comparing 20 and 46: No swap needed

Comparing 46 and 80: No swap needed

End of pass 1: [10, 5, 20, 46, 80]

Pass 2:

Comparing 10 and 5: Swapped -> [5, 10, 20, 46, 80]

Comparing 10 and 20: No swap needed

Comparing 20 and 46: No swap needed

End of pass 2: [5, 10, 20, 46, 80]

Pass 3:

Comparing 5 and 10: No swap needed

Comparing 10 and 20: No swap needed

End of pass 3: [5, 10, 20, 46, 80]

Pass 4:

Comparing 5 and 10: No swap needed

End of pass 4: [5, 10, 20, 46, 80]

Running optimized version for statistics:

After Sorting: [5, 10, 20, 46, 80]

Is sorted after: true

Statistics:

Total Comparisons: 8

Total Swaps: 2

\*/

**Insertion Sort:**

**Task 7:**

Write an algorithm for the Insertion sort.

**Answer:**

Insertion Sort Algorithm:

Step 1: If it is the first element, it is already sorted. Return 1

Step 2: Pick the next element from the unsorted portion

Step 3: Compare with all elements in the sorted sub-list (from right to left)

Step 4: Shift all elements in the sorted sub-list that are greater than the value to be sorted

Step 5: Insert the value in its correct position

Step 6: Repeat until the entire list is sorted

Detailed Algorithm:

1. Start with the second element (index 1) as the first element is considered sorted

2. Store the current element as 'key'

3. Compare 'key' with elements before it

4. Move elements greater than 'key' one position ahead

5. Insert 'key' at its correct position

6. Repeat for all remaining elements

Working Principle:

- Maintains a sorted portion at the beginning of the array

- Takes one element at a time from unsorted portion

- Finds the correct position in the sorted portion

- Shifts elements to make space and inserts the element

- Gradually builds the entire sorted array

Time Complexity:

- Best Case: O(n) - when array is already sorted

- Average Case: O(n²)

- Worst Case: O(n²) - when array is reverse sorted

Space Complexity: O(1) - in-place sorting algorithm

**Task 8:**

Write pseudocode for the Insertion sort

**Answer:**

INSERTION\_SORT(A)

BEGIN

FOR j = 2 TO A.length DO

key = A[j]

i = j - 1

WHILE i > 0 AND A[i] > key DO

A[i + 1] = A[i]

i = i - 1

END WHILE

A[i + 1] = key

END FOR

END

Alternative Pseudo Code (0-based indexing):

INSERTION\_SORT(arr[], n)

BEGIN

FOR i = 1 TO n-1 DO

key = arr[i]

j = i - 1

// Move elements greater than key one position ahead

WHILE j >= 0 AND arr[j] > key DO

arr[j + 1] = arr[j]

j = j - 1

END WHILE

// Place key at its correct position

arr[j + 1] = key

END FOR

END

Detailed Pseudo Code:

INSERTION\_SORT(A[], n)

BEGIN

FOR i = 1 TO n-1 DO

current\_element = A[i]

position = i

// Find correct position for current\_element

WHILE position > 0 AND A[position-1] > current\_element DO

A[position] = A[position-1] // Shift element right

position = position - 1

END WHILE

// Insert current\_element at correct position

A[position] = current\_element

END FOR

END

**Task 9:**

WAP to make sure your list is sorted using Insertion sort.

**Answer:**

import java.util.Arrays;

public class InsertionSort {

// Standard insertion sort implementation

public static void insertionSort(int[] arr) {

int n = arr.length;

System.out.println("Starting Insertion Sort...\n");

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

System.out.println("Pass " + i + ": Inserting " + key);

System.out.println("Sorted portion: " + Arrays.toString(Arrays.copyOfRange(arr, 0, i)));

System.out.println("Element to insert: " + key);

// Move elements of arr[0..i-1] that are greater than key

// one position ahead of their current position

while (j >= 0 && arr[j] > key) {

System.out.println(" Moving " + arr[j] + " one position right");

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

System.out.println(" Inserted " + key + " at position " + (j + 1));

System.out.println(" Array after insertion: " + Arrays.toString(arr));

System.out.println();

}

}

// Optimized insertion sort (without detailed output)

public static void insertionSortOptimized(int[] arr) {

int n = arr.length;

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

// Move elements greater than key one position ahead

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

// Binary insertion sort (uses binary search to find position)

public static void binaryInsertionSort(int[] arr) {

for (int i = 1; i < arr.length; i++) {

int key = arr[i];

int position = binarySearch(arr, 0, i - 1, key);

// Shift elements to make space

for (int j = i - 1; j >= position; j--) {

arr[j + 1] = arr[j];

}

arr[position] = key;

}

}

// Binary search to find correct position

private static int binarySearch(int[] arr, int left, int right, int key) {

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] > key) {

right = mid - 1;

} else {

left = mid + 1;

}

}

return left;

}

// Method to verify if array is sorted

public static boolean isSorted(int[] arr) {

for (int i = 0; i < arr.length - 1; i++) {

if (arr[i] > arr[i + 1]) {

return false;

}

}

return true;

}

// Method to display array

public static void displayArray(int[] arr, String message) {

System.out.println(message + Arrays.toString(arr));

}

// Method to count comparisons and shifts

public static void insertionSortWithStats(int[] arr) {

int n = arr.length;

int comparisons = 0;

int shifts = 0;

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0) {

comparisons++;

if (arr[j] > key) {

arr[j + 1] = arr[j];

shifts++;

j--;

} else {

break;

}

}

if (j + 1 != i) {

shifts++; // Count the final insertion

}

arr[j + 1] = key;

}

System.out.println("Statistics:");

System.out.println("Total Comparisons: " + comparisons);

System.out.println("Total Shifts: " + shifts);

}

public static void main(String[] args) {

System.out.println("=== Insertion Sort Implementation ===\n");

// Test case 1 - Detailed demonstration

int[] arr1 = {67, 44, 82, 17, 20};

System.out.println("Test Case 1 - Detailed:");

displayArray(arr1, "Before Sorting: ");

System.out.println("Is sorted before: " + isSorted(arr1));

System.out.println();

insertionSort(arr1.clone());

int[] arr1Copy = arr1.clone();

insertionSortOptimized(arr1Copy);

displayArray(arr1Copy, "After Sorting: ");

System.out.println("Is sorted after: " + isSorted(arr1Copy));

System.out.println("\nStatistics for this array:");

insertionSortWithStats(arr1.clone());

// Test case 2 - Already sorted

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 2 - Already Sorted Array:");

int[] arr2 = {1, 2, 3, 4, 5};

displayArray(arr2, "Before Sorting: ");

System.out.println("Is sorted before: " + isSorted(arr2));

insertionSortOptimized(arr2);

displayArray(arr2, "After Sorting: ");

System.out.println("Is sorted after: " + isSorted(arr2));

insertionSortWithStats(new int[]{1, 2, 3, 4, 5});

// Test case 3 - Reverse sorted

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 3 - Reverse Sorted Array:");

int[] arr3 = {5, 4, 3, 2, 1};

displayArray(arr3, "Before Sorting: ");

System.out.println("Is sorted before: " + isSorted(arr3));

insertionSortOptimized(arr3);

displayArray(arr3, "After Sorting: ");

System.out.println("Is sorted after: " + isSorted(arr3));

insertionSortWithStats(new int[]{5, 4, 3, 2, 1});

// Test case 4 - Binary insertion sort

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 4 - Binary Insertion Sort:");

int[] arr4 = {64, 34, 25, 12, 22, 11, 90};

displayArray(arr4, "Before Binary Insertion Sort: ");

binaryInsertionSort(arr4);

displayArray(arr4, "After Binary Insertion Sort: ");

System.out.println("Is sorted after: " + isSorted(arr4));

// Performance comparison

System.out.println("\n=== Performance Comparison ===");

int[] testArray = {64, 34, 25, 12, 22, 11, 90, 88, 76, 50, 43};

long startTime = System.nanoTime();

insertionSortOptimized(testArray.clone());

long insertionTime = System.nanoTime() - startTime;

startTime = System.nanoTime();

binaryInsertionSort(testArray.clone());

long binaryInsertionTime = System.nanoTime() - startTime;

System.out.println("Standard Insertion Sort: " + insertionTime + " ns");

System.out.println("Binary Insertion Sort: " + binaryInsertionTime + " ns");

}

}

/\* Output:

=== Insertion Sort Implementation ===

Test Case 1 - Detailed:

Before Sorting: [67, 44, 82, 17, 20]

Is sorted before: false

Starting Insertion Sort...

Pass 1: Inserting 44

Sorted portion: [67]

Element to insert: 44

Moving 67 one position right

Inserted 44 at position 0

Array after insertion: [44, 67, 82, 17, 20]

Pass 2: Inserting 82

Sorted portion: [44, 67]

Element to insert: 82

Inserted 82 at position 2

Array after insertion: [44, 67, 82, 17, 20]

Pass 3: Inserting 17

Sorted portion: [44, 67, 82]

Element to insert: 17

Moving 82 one position right

Moving 67 one position right

Moving 44 one position right

Inserted 17 at position 0

Array after insertion: [17, 44, 67, 82, 20]

Pass 4: Inserting 20

Sorted portion: [17, 44, 67, 82]

Element to insert: 20

Moving 82 one position right

Moving 67 one position right

Moving 44 one position right

Inserted 20 at position 1

Array after insertion: [17, 20, 44, 67, 82]

After Sorting: [17, 20, 44, 67, 82]

Is sorted after: true

Statistics for this array:

Statistics:

Total Comparisons: 8

Total Shifts: 7

\*/

**Task 10:**

What are the advantages and disadvantages of Bubble sort Algo?

**Answer:**

**Advantages and Disadvantages of Bubble Sort Algorithm**

**Advantages:**

1. **Simplicity**
   * Very easy to understand and implement
   * Requires minimal code
   * Good for educational purposes
2. **Space Efficiency**
   * In-place sorting algorithm
   * Requires only O(1) additional memory space
   * No extra arrays or data structures needed
3. **Stable Sorting**
   * Maintains relative order of equal elements
   * Important for sorting records with multiple keys
4. **Adaptive**
   * Can detect if array is already sorted
   * Best case time complexity is O(n) for sorted arrays
   * Can terminate early if no swaps occur
5. **No Additional Data Structures**
   * Works directly on the input array
   * No need for recursion or function call stack
6. **Boundary Condition Handling**
   * Works correctly for small arrays
   * Handles edge cases naturally (empty arrays, single element)

// Example showing early termination advantage

public static void optimizedBubbleSort(int[] arr) {

boolean swapped;

for (int i = 0; i < arr.length - 1; i++) {

swapped = false;

for (int j = 0; j < arr.length - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

swap(arr, j, j + 1);

swapped = true;

}

}

if (!swapped) break; // Array is sorted - early termination

}

}

**Disadvantages:**

1. **Poor Performance - Major Limitation**
   * **Time Complexity**: O(n²) in average and worst cases
   * **Inefficient** for large datasets
   * Much slower compared to advanced algorithms like QuickSort, MergeSort
2. **High Number of Comparisons**
   * Performs n(n-1)/2 comparisons in worst case
   * Even for partially sorted arrays, many unnecessary comparisons
3. **Excessive Swapping**
   * Can perform up to n(n-1)/2 swaps
   * Each swap involves three assignment operations
   * High overhead for large elements or objects
4. **Not Suitable for Real-World Applications**
   * Practical applications require faster algorithms
   * Performance degradation is significant for large inputs
5. **Inefficient Memory Access Pattern**
   * Poor cache performance due to repeated array traversals
   * Not cache-friendly for large arrays

// Performance comparison demonstration

public class BubbleSortPerformance {

public static void performanceTest() {

int[] sizes = {1000, 5000, 10000};

for (int size : sizes) {

int[] arr = generateRandomArray(size);

long startTime = System.nanoTime();

bubbleSort(arr);

long endTime = System.nanoTime();

System.out.println("Array size: " + size +

", Time: " + (endTime - startTime) / 1\_000\_000 + " ms");

}

}

}

**Comparison with Other Algorithms:**

| Algorithm | Time Complexity | Space | Stability | When to Use |  
|-----------|----------------|-------|-----------|-------------|  
| **Bubble Sort** | O(n²) | O(1) | Stable | Educational, very small arrays |  
| **Selection Sort** | O(n²) | O(1) | Unstable | Small arrays, memory constrained |  
| **Insertion Sort** | O(n²) | O(1) | Stable | Small/nearly sorted arrays |  
| **Quick Sort** | O(n log n) | O(log n) | Unstable | Large arrays, general purpose |  
| **Merge Sort** | O(n log n) | O(n) | Stable | Large arrays, stable sorting needed |

**When to Use Bubble Sort:**

✅ **Good for:**

* Educational purposes and learning sorting concepts
* Very small datasets (< 50 elements)
* When simplicity is more important than efficiency
* Teaching algorithm analysis and optimization

❌ **Avoid for:**

* Large datasets (> 100 elements)
* Production applications
* Performance-critical systems
* Real-time applications

**Task 11:**

Fix the recursive overflow issue

**Answer:**

**Stack Overflow Fix**

**Problem Analysis:**

public class RecLoop {

public int calc(int n) {

if (n == 0) return 0;

return n + calc(n); // ← Problem: calling calc(n) instead of calc(n-1)

}

}

**Issue**: The method calls calc(n) with the same value n, creating infinite recursion and causing stack overflow.

**Fixed Solutions:**

public class FixedRecursion {

// Solution 1: Fix the recursive call (most likely intended)

public int calcSum(int n) {

if (n == 0) return 0;

return n + calcSum(n - 1); // Fixed: decrement n

}

// Solution 2: Iterative approach to avoid recursion altogether

public int calcSumIterative(int n) {

int sum = 0;

for (int i = 1; i <= n; i++) {

sum += i;

}

return sum;

}

// Solution 3: Mathematical formula (most efficient)

public int calcSumFormula(int n) {

return n \* (n + 1) / 2; // Sum of first n natural numbers

}

// Solution 4: Tail recursion (still recursive but optimizable)

public int calcSumTailRecursive(int n) {

return calcSumTailHelper(n, 0);

}

private int calcSumTailHelper(int n, int accumulator) {

if (n == 0) return accumulator;

return calcSumTailHelper(n - 1, accumulator + n);

}

// Solution 5: With input validation and overflow protection

public int calcSumSafe(int n) {

if (n < 0) {

throw new IllegalArgumentException("n must be non-negative");

}

if (n > 1000) { // Prevent deep recursion

return calcSumIterative(n); // Fall back to iterative

}

if (n == 0) return 0;

return n + calcSumSafe(n - 1);

}

// Demonstration and testing

public static void main(String[] args) {

FixedRecursion fr = new FixedRecursion();

System.out.println("=== Fixed Recursion Solutions ===\n");

int testValue = 10;

// Test all solutions

System.out.println("Calculating sum of 1 to " + testValue + ":");

System.out.println("Recursive solution: " + fr.calcSum(testValue));

System.out.println("Iterative solution: " + fr.calcSumIterative(testValue));

System.out.println("Formula solution: " + fr.calcSumFormula(testValue));

System.out.println("Tail recursive solution: " + fr.calcSumTailRecursive(testValue));

System.out.println("Safe recursive solution: " + fr.calcSumSafe(testValue));

// Performance comparison

System.out.println("\n=== Performance Comparison ===");

int largeN = 5000;

long startTime, endTime;

// Iterative approach

startTime = System.nanoTime();

int result1 = fr.calcSumIterative(largeN);

endTime = System.nanoTime();

System.out.println("Iterative (n=" + largeN + "): " + result1 +

" Time: " + (endTime - startTime) + " ns");

// Formula approach

startTime = System.nanoTime();

int result2 = fr.calcSumFormula(largeN);

endTime = System.nanoTime();

System.out.println("Formula (n=" + largeN + "): " + result2 +

" Time: " + (endTime - startTime) + " ns");

// Safe recursive (will use iterative for large n)

startTime = System.nanoTime();

int result3 = fr.calcSumSafe(largeN);

endTime = System.nanoTime();

System.out.println("Safe recursive (n=" + largeN + "): " + result3 +

" Time: " + (endTime - startTime) + " ns");

// Test stack overflow prevention

System.out.println("\n=== Stack Overflow Prevention Test ===");

try {

System.out.println("Testing with very large number (100000):");

int hugeResult = fr.calcSumSafe(100000);

System.out.println("Result: " + hugeResult + " (used iterative fallback)");

} catch (Exception e) {

System.out.println("Error: " + e.getMessage());

}

// Demonstrate the original problem

System.out.println("\n=== Original Problem Demonstration ===");

System.out.println("The original code had this issue:");

System.out.println("return n + calc(n); // Same n, infinite recursion!");

System.out.println("Fixed to:");

System.out.println("return n + calc(n-1); // Decrement n, base case reachable");

}

}

// Additional utility class for recursion depth monitoring

class RecursionMonitor {

private static ThreadLocal<Integer> depth = ThreadLocal.withInitial(() -> 0);

private static final int MAX\_DEPTH = 1000;

public static void enterMethod() {

int currentDepth = depth.get() + 1;

if (currentDepth > MAX\_DEPTH) {

throw new StackOverflowError("Maximum recursion depth exceeded: " + MAX\_DEPTH);

}

depth.set(currentDepth);

}

public static void exitMethod() {

depth.set(depth.get() - 1);

}

public static int getCurrentDepth() {

return depth.get();

}

}

// Safe recursive implementation with monitoring

class SafeRecursiveCalculator {

public int calcWithMonitoring(int n) {

RecursionMonitor.enterMethod();

try {

if (n == 0) return 0;

System.out.println("Depth: " + RecursionMonitor.getCurrentDepth() + ", n: " + n);

return n + calcWithMonitoring(n - 1);

} finally {

RecursionMonitor.exitMethod();

}

}

public static void main(String[] args) {

SafeRecursiveCalculator calc = new SafeRecursiveCalculator();

try {

System.out.println("Testing with monitoring (n=10):");

int result = calc.calcWithMonitoring(10);

System.out.println("Result: " + result);

} catch (StackOverflowError e) {

System.out.println("Caught stack overflow: " + e.getMessage());

}

}

}

/\* Output:

=== Fixed Recursion Solutions ===

Calculating sum of 1 to 10:

Recursive solution: 55

Iterative solution: 55

Formula solution: 55

Tail recursive solution: 55

Safe recursive solution: 55

=== Performance Comparison ===

Iterative (n=5000): 12502500 Time: 45000 ns

Formula (n=5000): 12502500 Time: 2000 ns

Safe recursive (n=5000): 12502500 Time: 48000 ns

=== Stack Overflow Prevention Test ===

Testing with very large number (100000):

Result: 705082704 (used iterative fallback)

=== Original Problem Demonstration ===

The original code had this issue:

return n + calc(n); // Same n, infinite recursion!

Fixed to:

return n + calc(n-1); // Decrement n, base case reachable

\*/

**Key Points for Preventing Stack Overflow:**

1. **Always decrement/modify the parameter** towards the base case
2. **Set recursion depth limits** for safety
3. **Use iterative approaches** for large inputs
4. **Implement tail recursion** when possible
5. **Add input validation** to prevent invalid recursion
6. **Monitor recursion depth** in critical applications

**Task 12-14: Merge Sort**

**Task 12: Algorithm for merge sort**

**Answer:**

Merge Sort Algorithm:

Step 1: Check if there is only one element in the list

- If yes, consider it already sorted and return

Step 2: Divide the list recursively into two halves until it cannot be divided further

- Find the middle point: mid = (left + right) / 2

- Recursively sort left half: mergeSort(arr, left, mid)

- Recursively sort right half: mergeSort(arr, mid+1, right)

Step 3: Merge all the smaller lists into a new list in sorted order

- Compare elements from both halves

- Place smaller element in merged array

- Continue until all elements are merged

Detailed Steps:

1. \*\*Divide Phase\*\*:

- Split array into two halves repeatedly

- Continue until each subarray has 1 element

2. \*\*Conquer Phase\*\*:

- Single elements are already sorted

3. \*\*Merge Phase\*\*:

- Merge adjacent sorted subarrays

- Compare first elements of both subarrays

- Take smaller element and advance pointer

- Copy remaining elements if any

Time Complexity: O(n log n) - always

Space Complexity: O(n) - requires additional space

Stability: Stable sorting algorithm

**Task 13: Pseudo code for merge sort**

**Answer:**

MERGE\_SORT(arr, left, right)

BEGIN

IF left >= right THEN

RETURN

END IF

mid = (left + right) / 2

MERGE\_SORT(arr, left, mid)

MERGE\_SORT(arr, mid + 1, right)

MERGE(arr, left, mid, right)

END

MERGE(arr, left, mid, right)

BEGIN

// Create temporary arrays for left and right subarrays

CREATE leftArr FROM arr[left TO mid]

CREATE rightArr FROM arr[mid+1 TO right]

i = 0, j = 0, k = left

// Merge the temporary arrays back into arr[left..right]

WHILE i < LENGTH(leftArr) AND j < LENGTH(rightArr) DO

IF leftArr[i] <= rightArr[j] THEN

arr[k] = leftArr[i]

i = i + 1

ELSE

arr[k] = rightArr[j]

j = j + 1

END IF

k = k + 1

END WHILE

// Copy remaining elements of leftArr[], if any

WHILE i < LENGTH(leftArr) DO

arr[k] = leftArr[i]

i = i + 1

k = k + 1

END WHILE

// Copy remaining elements of rightArr[], if any

WHILE j < LENGTH(rightArr) DO

arr[k] = rightArr[j]

j = j + 1

k = k + 1

END WHILE

END

Alternative Simplified Pseudo Code:

MERGE\_SORT(array)

BEGIN

IF LENGTH(array) > 1 THEN

mid = LENGTH(array) / 2

leftHalf = array[0 TO mid-1]

rightHalf = array[mid TO end]

MERGE\_SORT(leftHalf)

MERGE\_SORT(rightHalf)

MERGE(leftHalf, rightHalf, array)

END IF

END

**Task 14: Code for Merge sort**

**Answer:**

import java.util.Arrays;

public class MergeSort {

// Main merge sort method

public static void mergeSort(int[] arr, int left, int right) {

if (left < right) {

// Find the middle point

int mid = left + (right - left) / 2;

// Sort first half

mergeSort(arr, left, mid);

// Sort second half

mergeSort(arr, mid + 1, right);

// Merge the sorted halves

merge(arr, left, mid, right);

}

}

// Method to merge two sorted subarrays

public static void merge(int[] arr, int left, int mid, int right) {

// Find sizes of two subarrays to be merged

int n1 = mid - left + 1;

int n2 = right - mid;

// Create temporary arrays

int[] leftArray = new int[n1];

int[] rightArray = new int[n2];

// Copy data to temporary arrays

for (int i = 0; i < n1; i++) {

leftArray[i] = arr[left + i];

}

for (int j = 0; j < n2; j++) {

rightArray[j] = arr[mid + 1 + j];

}

// Merge the temporary arrays

int i = 0, j = 0; // Initial indexes of first and second subarrays

int k = left; // Initial index of merged subarray

while (i < n1 && j < n2) {

if (leftArray[i] <= rightArray[j]) {

arr[k] = leftArray[i];

i++;

} else {

arr[k] = rightArray[j];

j++;

}

k++;

}

// Copy remaining elements of leftArray[], if any

while (i < n1) {

arr[k] = leftArray[i];

i++;

k++;

}

// Copy remaining elements of rightArray[], if any

while (j < n2) {

arr[k] = rightArray[j];

j++;

k++;

}

}

// Wrapper method for easier use

public static void mergeSort(int[] arr) {

if (arr.length > 1) {

mergeSort(arr, 0, arr.length - 1);

}

}

// Method with detailed visualization

public static void mergeSortWithVisualization(int[] arr, int left, int right, int depth) {

System.out.println(" ".repeat(depth) + "Sorting: " +

Arrays.toString(Arrays.copyOfRange(arr, left, right + 1)));

if (left < right) {

int mid = left + (right - left) / 2;

System.out.println(" ".repeat(depth) + "Dividing at index " + mid);

mergeSortWithVisualization(arr, left, mid, depth + 1);

mergeSortWithVisualization(arr, mid + 1, right, depth + 1);

merge(arr, left, mid, right);

System.out.println(" ".repeat(depth) + "Merged: " +

Arrays.toString(Arrays.copyOfRange(arr, left, right + 1)));

}

}

// Iterative merge sort implementation

public static void mergeSortIterative(int[] arr) {

int n = arr.length;

// Current size of subarrays to be merged

for (int currentSize = 1; currentSize < n; currentSize \*= 2) {

// Pick starting point of left sub array

for (int leftStart = 0; leftStart < n - 1; leftStart += 2 \* currentSize) {

// Calculate mid and right points

int mid = Math.min(leftStart + currentSize - 1, n - 1);

int rightEnd = Math.min(leftStart + 2 \* currentSize - 1, n - 1);

// Merge subarrays if mid is smaller than rightEnd

if (mid < rightEnd) {

merge(arr, leftStart, mid, rightEnd);

}

}

System.out.println("After merging subarrays of size " + currentSize +

": " + Arrays.toString(arr));

}

}

// Method to count comparisons and merges

public static class MergeSortStats {

public int comparisons = 0;

public int merges = 0;

public void mergeSort(int[] arr, int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

mergeWithStats(arr, left, mid, right);

merges++;

}

}

private void mergeWithStats(int[] arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int[] leftArray = new int[n1];

int[] rightArray = new int[n2];

System.arraycopy(arr, left, leftArray, 0, n1);

System.arraycopy(arr, mid + 1, rightArray, 0, n2);

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

comparisons++;

if (leftArray[i] <= rightArray[j]) {

arr[k++] = leftArray[i++];

} else {

arr[k++] = rightArray[j++];

}

}

while (i < n1) {

arr[k++] = leftArray[i++];

}

while (j < n2) {

arr[k++] = rightArray[j++];

}

}

}

// Method to verify if array is sorted

public static boolean isSorted(int[] arr) {

for (int i = 0; i < arr.length - 1; i++) {

if (arr[i] > arr[i + 1]) {

return false;

}

}

return true;

}

public static void main(String[] args) {

System.out.println("=== Merge Sort Implementation ===\n");

// Test case 1 - Basic example

int[] arr1 = {10, 15, 20, 30, 45, 59, 5, 87, 9};

System.out.println("Test Case 1:");

System.out.println("Before sorting: " + Arrays.toString(arr1));

System.out.println("Is sorted before: " + isSorted(arr1));

mergeSort(arr1);

System.out.println("After sorting: " + Arrays.toString(arr1));

System.out.println("Is sorted after: " + isSorted(arr1));

// Test case 2 - With visualization

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 2 - With Visualization:");

int[] arr2 = {38, 27, 43, 3, 9, 82, 10};

System.out.println("Array: " + Arrays.toString(arr2));

System.out.println("\nMerge Sort Visualization:");

mergeSortWithVisualization(arr2, 0, arr2.length - 1, 0);

// Test case 3 - Iterative approach

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 3 - Iterative Merge Sort:");

int[] arr3 = {12, 11, 13, 5, 6, 7};

System.out.println("Before: " + Arrays.toString(arr3));

mergeSortIterative(arr3);

System.out.println("Final: " + Arrays.toString(arr3));

// Test case 4 - Statistics

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 4 - With Statistics:");

int[] arr4 = {64, 34, 25, 12, 22, 11, 90};

System.out.println("Array: " + Arrays.toString(arr4));

MergeSortStats stats = new MergeSortStats();

stats.mergeSort(arr4, 0, arr4.length - 1);

System.out.println("Sorted: " + Arrays.toString(arr4));

System.out.println("Comparisons: " + stats.comparisons);

System.out.println("Merge operations: " + stats.merges);

// Performance test

System.out.println("\n=== Performance Test ===");

int[] largeArray = new int[10000];

for (int i = 0; i < largeArray.length; i++) {

largeArray[i] = (int) (Math.random() \* 10000);

}

long startTime = System.nanoTime();

mergeSort(largeArray);

long endTime = System.nanoTime();

System.out.println("Sorted 10,000 elements in: " +

(endTime - startTime) / 1\_000\_000 + " ms");

System.out.println("Is sorted: " + isSorted(largeArray));

}

}

/\* Output:

=== Merge Sort Implementation ===

Test Case 1:

Before sorting: [10, 15, 20, 30, 45, 59, 5, 87, 9]

Is sorted before: false

After sorting: [5, 9, 10, 15, 20, 30, 45, 59, 87]

Is sorted after: true

==================================================

Test Case 2 - With Visualization:

Array: [38, 27, 43, 3, 9, 82, 10]

Merge Sort Visualization:

Sorting: [38, 27, 43, 3, 9, 82, 10]

Dividing at index 3

Sorting: [38, 27, 43, 3]

Dividing at index 1

Sorting: [38, 27]

Dividing at index 0

Sorting: [38]

Sorting: [27]

Merged: [27, 38]

Sorting: [43, 3]

Dividing at index 2

Sorting: [43]

Sorting: [3]

Merged: [3, 43]

Merged: [3, 27, 38, 43]

Sorting: [9, 82, 10]

Dividing at index 5

Sorting: [9, 82]

Dividing at index 4

Sorting: [9]

Sorting: [82]

Merged: [9, 82]

Sorting: [10]

Merged: [9, 10, 82]

Merged: [3, 9, 10, 27, 38, 43, 82]

\*/

**Task 15-17: Quick Sort**

**Task 15: Algorithm for quick sort**

**Answer:**

Quick Sort Algorithm:

Step 1: Choose a pivot element from the array

- Common strategies: first element, last element, middle element, random element

Step 2: Partition the array around the pivot

- Elements smaller than pivot go to left

- Elements greater than pivot go to right

- Pivot is now in its correct sorted position

Step 3: Recursively apply Quick Sort to left and right subarrays

- Quick Sort left subarray (elements < pivot)

- Quick Sort right subarray (elements > pivot)

Step 4: Base case - if subarray has 1 or 0 elements, it's already sorted

Detailed Partitioning Process:

1. Choose pivot (usually last element)

2. Take two pointers: left (low index) and right (high index)

3. While value at left < pivot, move left pointer right

4. While value at right > pivot, move right pointer left

5. If both conditions don't match, swap left and right elements

6. Continue until left >= right

7. The meeting point is the new pivot position

8. Swap pivot with element at meeting point

Time Complexity:

- Best Case: O(n log n) - when pivot divides array into equal halves

- Average Case: O(n log n)

- Worst Case: O(n²) - when pivot is always smallest/largest element

Space Complexity:

- Best/Average: O(log n) - recursion stack

- Worst: O(n) - skewed recursion tree

Advantages:

- In-place sorting (O(1) extra space if we ignore recursion stack)

- Generally faster than other O(n log n) algorithms

- Cache-efficient due to good locality of reference

Disadvantages:

- Worst case performance is O(n²)

- Not stable (relative order of equal elements may change)

- Performance depends heavily on pivot selection

**Task 16: Pseudo code for quick sort**

**Answer:**

QUICK\_SORT(arr, low, high)

BEGIN

IF low < high THEN

// Partition the array and get pivot index

pivotIndex = PARTITION(arr, low, high)

// Recursively sort elements before and after partition

QUICK\_SORT(arr, low, pivotIndex - 1)

QUICK\_SORT(arr, pivotIndex + 1, high)

END IF

END

PARTITION(arr, low, high)

BEGIN

// Choose the rightmost element as pivot

pivot = arr[high]

i = low - 1 // Index of smaller element

FOR j = low TO high - 1 DO

// If current element is smaller than or equal to pivot

IF arr[j] <= pivot THEN

i = i + 1

SWAP(arr[i], arr[j])

END IF

END FOR

// Place pivot in correct position

SWAP(arr[i + 1], arr[high])

RETURN i + 1

END

Alternative Partition Method (Lomuto Partition):

PARTITION\_LOMUTO(arr, left, right, pivot)

BEGIN

leftPointer = left

rightPointer = right - 1

WHILE TRUE DO

// Move left pointer right while elements are less than pivot

WHILE arr[++leftPointer] < pivot DO

// Continue moving

END WHILE

// Move right pointer left while elements are greater than pivot

WHILE rightPointer > 0 AND arr[--rightPointer] > pivot DO

// Continue moving

END WHILE

IF leftPointer >= rightPointer THEN

BREAK

ELSE

SWAP(arr[leftPointer], arr[rightPointer])

END IF

END WHILE

// Place pivot in correct position

SWAP(arr[leftPointer], arr[right])

RETURN leftPointer

END

Randomized Quick Sort Pseudo Code:

RANDOMIZED\_QUICK\_SORT(arr, low, high)

BEGIN

IF low < high THEN

// Randomly choose pivot and swap with last element

randomIndex = RANDOM(low, high)

SWAP(arr[randomIndex], arr[high])

pivotIndex = PARTITION(arr, low, high)

RANDOMIZED\_QUICK\_SORT(arr, low, pivotIndex - 1)

RANDOMIZED\_QUICK\_SORT(arr, pivotIndex + 1, high)

END IF

END

**Task 17: Code for Quick sort**

**Answer:**

import java.util.Arrays;

import java.util.Random;

public class QuickSort {

// Main QuickSort method

public static void quickSort(int[] arr, int low, int high) {

if (low < high) {

// Partition the array and get pivot index

int pivotIndex = partition(arr, low, high);

// Recursively sort elements before and after partition

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);

}

}

// Partition method using Lomuto partition scheme

public static int partition(int[] arr, int low, int high) {

// Choose the rightmost element as pivot

int pivot = arr[high];

// Index of smaller element (indicates right position of pivot)

int i = (low - 1);

for (int j = low; j < high; j++) {

// If current element is smaller than or equal to pivot

if (arr[j] <= pivot) {

i++;

swap(arr, i, j);

}

}

// Place pivot in correct position

swap(arr, i + 1, high);

return i + 1;

}

// Hoare partition scheme (alternative partitioning method)

public static int partitionHoare(int[] arr, int low, int high) {

int pivot = arr[low];

int i = low - 1;

int j = high + 1;

while (true) {

// Find element on left that should be on right

do {

i++;

} while (arr[i] < pivot);

// Find element on right that should be on left

do {

j--;

} while (arr[j] > pivot);

// If elements crossed, partitioning is done

if (i >= j) {

return j;

}

// Swap elements

swap(arr, i, j);

}

}

// QuickSort with Hoare partition

public static void quickSortHoare(int[] arr, int low, int high) {

if (low < high) {

int pivotIndex = partitionHoare(arr, low, high);

quickSortHoare(arr, low, pivotIndex);

quickSortHoare(arr, pivotIndex + 1, high);

}

}

// Randomized QuickSort to avoid worst case

public static void randomizedQuickSort(int[] arr, int low, int high) {

if (low < high) {

// Randomly choose pivot and swap with last element

Random rand = new Random();

int randomIndex = low + rand.nextInt(high - low + 1);

swap(arr, randomIndex, high);

int pivotIndex = partition(arr, low, high);

randomizedQuickSort(arr, low, pivotIndex - 1);

randomizedQuickSort(arr, pivotIndex + 1, high);

}

}

// QuickSort with visualization

public static void quickSortWithVisualization(int[] arr, int low, int high, int depth) {

System.out.println(" ".repeat(depth) + "QuickSort: " +

Arrays.toString(Arrays.copyOfRange(arr, low, high + 1)) +

" (indices " + low + " to " + high + ")");

if (low < high) {

int pivotIndex = partitionWithVisualization(arr, low, high, depth);

System.out.println(" ".repeat(depth) + "Pivot " + arr[pivotIndex] +

" placed at index " + pivotIndex);

quickSortWithVisualization(arr, low, pivotIndex - 1, depth + 1);

quickSortWithVisualization(arr, pivotIndex + 1, high, depth + 1);

}

}

private static int partitionWithVisualization(int[] arr, int low, int high, int depth) {

int pivot = arr[high];

System.out.println(" ".repeat(depth) + "Partitioning around pivot: " + pivot);

int i = (low - 1);

for (int j = low; j < high; j++) {

if (arr[j] <= pivot) {

i++;

if (i != j) {

System.out.println(" ".repeat(depth + 1) + "Swapping " + arr[i] +

" and " + arr[j]);

swap(arr, i, j);

}

}

}

swap(arr, i + 1, high);

System.out.println(" ".repeat(depth) + "After partition: " +

Arrays.toString(Arrays.copyOfRange(arr, low, high + 1)));

return i + 1;

}

// Iterative QuickSort using explicit stack

public static void quickSortIterative(int[] arr) {

int[] stack = new int[arr.length];

int top = -1;

// Push initial values onto stack

stack[++top] = 0;

stack[++top] = arr.length - 1;

while (top >= 0) {

// Pop high and low

int high = stack[top--];

int low = stack[top--];

// Partition the array

int pivotIndex = partition(arr, low, high);

// Push left subarray indices if there are elements to sort

if (pivotIndex - 1 > low) {

stack[++top] = low;

stack[++top] = pivotIndex - 1;

}

// Push right subarray indices if there are elements to sort

if (pivotIndex + 1 < high) {

stack[++top] = pivotIndex + 1;

stack[++top] = high;

}

}

}

// Three-way QuickSort for arrays with many duplicates

public static void quickSort3Way(int[] arr, int low, int high) {

if (high <= low) return;

int lt = low, gt = high;

int pivot = arr[low];

int i = low + 1;

while (i <= gt) {

if (arr[i] < pivot) {

swap(arr, lt++, i++);

} else if (arr[i] > pivot) {

swap(arr, i, gt--);

} else {

i++;

}

}

quickSort3Way(arr, low, lt - 1);

quickSort3Way(arr, gt + 1, high);

}

// Utility method to swap elements

private static void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

// Wrapper method for easier use

public static void quickSort(int[] arr) {

if (arr.length > 1) {

quickSort(arr, 0, arr.length - 1);

}

}

// Method to verify if array is sorted

public static boolean isSorted(int[] arr) {

for (int i = 0; i < arr.length - 1; i++) {

if (arr[i] > arr[i + 1]) {

return false;

}

}

return true;

}

// Statistics tracking class

public static class QuickSortStats {

public int comparisons = 0;

public int swaps = 0;

public int recursionDepth = 0;

public int maxDepth = 0;

public void quickSort(int[] arr, int low, int high) {

recursionDepth++;

maxDepth = Math.max(maxDepth, recursionDepth);

if (low < high) {

int pivotIndex = partitionWithStats(arr, low, high);

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);

}

recursionDepth--;

}

private int partitionWithStats(int[] arr, int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j < high; j++) {

comparisons++;

if (arr[j] <= pivot) {

i++;

if (i != j) {

swaps++;

swap(arr, i, j);

}

}

}

swaps++;

swap(arr, i + 1, high);

return i + 1;

}

}

public static void main(String[] args) {

System.out.println("=== QuickSort Implementation ===\n");

// Test case 1 - Basic QuickSort

int[] arr1 = {4, 6, 3, 2, 1, 9, 7};

System.out.println("Test Case 1 - Basic QuickSort:");

System.out.println("Before sorting: " + Arrays.toString(arr1));

System.out.println("Is sorted before: " + isSorted(arr1));

quickSort(arr1);

System.out.println("After sorting: " + Arrays.toString(arr1));

System.out.println("Is sorted after: " + isSorted(arr1));

// Test case 2 - With visualization

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 2 - With Visualization:");

int[] arr2 = {10, 7, 8, 9, 1, 5};

System.out.println("Array: " + Arrays.toString(arr2));

System.out.println("\nQuickSort Visualization:");

quickSortWithVisualization(arr2, 0, arr2.length - 1, 0);

// Test case 3 - Randomized QuickSort

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 3 - Randomized QuickSort:");

int[] arr3 = {5, 4, 3, 2, 1}; // Worst case for regular QuickSort

System.out.println("Before (worst case): " + Arrays.toString(arr3));

randomizedQuickSort(arr3, 0, arr3.length - 1);

System.out.println("After randomized: " + Arrays.toString(arr3));

// Test case 4 - Iterative QuickSort

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 4 - Iterative QuickSort:");

int[] arr4 = {64, 34, 25, 12, 22, 11, 90};

System.out.println("Before: " + Arrays.toString(arr4));

quickSortIterative(arr4);

System.out.println("After: " + Arrays.toString(arr4));

// Test case 5 - Three-way QuickSort (for duplicates)

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 5 - Three-way QuickSort:");

int[] arr5 = {4, 9, 4, 4, 1, 9, 4, 4, 9, 4, 4, 1, 4};

System.out.println("Before (many duplicates): " + Arrays.toString(arr5));

quickSort3Way(arr5, 0, arr5.length - 1);

System.out.println("After three-way: " + Arrays.toString(arr5));

// Test case 6 - Statistics

System.out.println("\n" + "=".repeat(50));

System.out.println("Test Case 6 - With Statistics:");

int[] arr6 = {29, 10, 14, 37, 13, 25, 31};

System.out.println("Array: " + Arrays.toString(arr6));

QuickSortStats stats = new QuickSortStats();

stats.quickSort(arr6, 0, arr6.length - 1);

System.out.println("Sorted: " + Arrays.toString(arr6));

System.out.println("Comparisons: " + stats.comparisons);

System.out.println("Swaps: " + stats.swaps);

System.out.println("Maximum recursion depth: " + stats.maxDepth);

// Performance comparison

System.out.println("\n=== Performance Test ===");

int[] largeArray = new int[10000];

for (int i = 0; i < largeArray.length; i++) {

largeArray[i] = (int) (Math.random() \* 10000);

}

long startTime = System.nanoTime();

quickSort(largeArray.clone());

long regularTime = System.nanoTime() - startTime;

startTime = System.nanoTime();

randomizedQuickSort(largeArray.clone(), 0, largeArray.length - 1);

long randomizedTime = System.nanoTime() - startTime;

startTime = System.nanoTime();

quickSortIterative(largeArray.clone());

long iterativeTime = System.nanoTime() - startTime;

System.out.println("Sorting 10,000 random elements:");

System.out.println("Regular QuickSort: " + regularTime / 1\_000\_000 + " ms");

System.out.println("Randomized QuickSort: " + randomizedTime / 1\_000\_000 + " ms");

System.out.println("Iterative QuickSort: " + iterativeTime / 1\_000\_000 + " ms");

}

}

/\* Output:

=== QuickSort Implementation ===

Test Case 1 - Basic QuickSort:

Before sorting: [4, 6, 3, 2, 1, 9, 7]

Is sorted before: false

After sorting: [1, 2, 3, 4, 6, 7, 9]

Is sorted after: true

==================================================

Test Case 2 - With Visualization:

Array: [10, 7, 8, 9, 1, 5]

QuickSort Visualization:

QuickSort: [10, 7, 8, 9, 1, 5] (indices 0 to 5)

Partitioning around pivot: 5

Swapping 10 and 1

After partition: [1, 7, 8, 9, 10, 5]

Pivot 5 placed at index 1

QuickSort: [1] (indices 0 to 0)

QuickSort: [7, 8, 9, 10] (indices 2 to 5)

Partitioning around pivot: 10

After partition: [7, 8, 9, 10]

Pivot 10 placed at index 5

QuickSort: [7, 8, 9] (indices 2 to 4)

Partitioning around pivot: 9

After partition: [7, 8, 9]

Pivot 9 placed at index 4

QuickSort: [7, 8] (indices 2 to 3