SOURCE CODES WITH RESULTS

1) Main Function:

```
// Author: Enes Denizli

// This program performs calculations on most-known sorting algorithms and
// analyzes their time efficiencies depending on specific types and
// characteristics of inputs


package apackage;

import java.util.Arrays;
import java.util.Random;

public class SortingAlgorithms {

	public static void main(String[] args) {

		// number will be changed depending on the array size
		int[] arr = arrayCreator(number);

		// Will be uncommented when almost sorted array is needed
		// and number will be changed depending on the array size
		int[] arr = sortedArrayCreator(number);

		// Will be uncommented when almost sorted array is needed
		// and number will be changed depending on the array size
		int[] arr = almostSortedArrayCreator(number);

		// Function call timer starts
		long startSelection = System.nanoTime();
		selectionSort(arr);
		// Function call timer stops
		long endSelection = System.nanoTime() - startSelection;
		System.out.println("Total time for selection sort: " +
		endSelection + "ns");

		// Function call timer starts
		long startBubble = System.nanoTime();
		bubbleSort(arr);
		// Function call timer stops
		long endBubble = System.nanoTime() - startBubble;
		System.out.println("Total time for bubble sort: " +
		endBubble + "ns");

		// BSSC stands for bubble sort.
		// Function call timer starts
		long startBSSC = System.nanoTime();
		bubbleSortSwapsCount(arr);
		// Function call timer stops
		long endBSSC = System.nanoTime() - startBSSC;
		System.out.println("Total time for BSSC sort: " +
		endBSSC + "ns");
```

```
            // Function call timer starts
            long startInsertion = System.nanoTime();
            insertionSort(arr);
            // Function call timer stops
            long endInsertion = System.nanoTime() - startInsertion;
            System.out.println("Total time for insertion sort: " +
            endInsertion + "ns");

            // Function call timer starts
            long startQuick = System.nanoTime();
            quickSort(arr, 0, arr.length - 1);
            // Function call timer stops
            long endQuick = System.nanoTime() - startQuick;
            System.out.println("Total time for quick sort: " +
            endQuick + "ns");

            // Function call timer starts
            long startMerge = System.nanoTime();
            mergeSort(arr, 0, arr.length - 1);
            // Function call timer stops
            long endMerge = System.nanoTime() - startMerge;
            System.out.println("Total time for merge sort: " +
            endMerge + "ns");
    }
```

2) sortedArray Function:

```
            // This function creates sorted array and its size is
            // between 0-"num" inclusive
            public static int[] sortedArrayCreator(int number) {
                    Random random = new Random();

                    int[] arr = new int[number];

                    for (int i = 0; i < number; i++) {
                            arr[i] = i + 1;
                    }
                    return arr;
            }
```

3) almostSortedArrayCreater Funtion:

```
    // This function creates sorted array but every tenth
    // element is a random number between 0-314 inclusive
    public static int[] almostSortedArrayCreator(int number) {
            Random random = new Random();
```

```
        int[] arr = new int[number];

        for (int i = 0; i < number; i++) {
                arr[i] = i + 1;
                if (arr[i] % 10 == 0) {
                        arr[i] = random.nextInt(314);
                }
        }
        return arr;
}
```

4) arrayCreator Function:

```
// This function creates random array of
// integer objects between 0-"num" inclusive
public static int[] arrayCreator(int number) {
        Random rd = new Random();

        int[] arr = new int[number];

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

                // Functions' parameter is the range
                // of numbers
                arr[i] = rd.nextInt(10000);
        }
        return arr;
}
```

5) Selection Sort:

```
// Sorts array using selection sort algorithm
public static void selectionSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
                int min = i;
                for (int j = i + 1; j < arr.length; j++) {
                        if (arr[j] < arr[min])
                                min = j;
                }
                int temp = arr[i];
                arr[i] = arr[min];
                arr[min] = temp;
}
```

6) Insertion Sort:

```
// Sorts the array using insertion sort algorithm
public static void insertionSort(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
                int v = arr[i];
                int j = i - 1;
```

```
                while (j >= 0 && arr[j] > v) {
                        arr[j + 1] = arr[j];
                        j = j - 1;
                }
                arr[j + 1] = v;
        }
}
```

7) Bubble Sort:

```
// Sorts array using bubble sort algorithm
public static void bubbleSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
                for (int j = 0; j < arr.length - 1 - i; j++) {
                        if (arr[j] > arr[j + 1]) {
                                int temp = arr[j];
                                arr[j] = arr[j + 1];
                                arr[j + 1] = temp;
                        }
                }
        }
}
```

8) Bubble Sort with Swaps Count:

```
// Sorts the array using bubble sort algorithm
// and counts swaps
public static void bubbleSortSwapsCount(int[] arr) {
        int swapCount = 0; // Swap counter
        for (int i = 0; i < arr.length - 1; i++) {
                for (int j = 0; j < arr.length - 1 - i; j++) {
                        if (arr[j] > arr[j + 1]) {
                                int temp = arr[j];
                                arr[j] = arr[j + 1];
                                arr[j + 1] = temp;
                                // If swap is made,
                                // increment swapCount
                                swapCount++;
                        }
                }
        }
}
```

9) Quick Sort

```
// Author: Matt Timmermans
// Cited from: https://stackoverflow.com/a/33884601/11780836
// Function has 3 parameter where last two is array's first
// end last indexes and calls 'partitionHoare' function and
// itself recursively
```

```
public static void quickSort(int[] arr, int begin, int end) {
        while (begin < end) {
                int q = partitionHoare(arr, begin, end);
                if (q - begin <= end - (q + 1)) {
                        quickSort(arr, begin, q);
                        begin = q + 1;
                } else {
                        quickSort(arr, q + 1, end);
                        end = q;
                }
        }
```

partitionHoare Function:

```
// Hoare partition but slightly improved where one undo swap
// is removed before the last swap
public static int partitionHoare(int[] arr, int begin, int end) {
        int pivot = arr[begin];
        int i = begin - 1;
        int j = end + 1;

        while (true) {
                do {
                        i++;
                } while (pivot > arr[i]);
                do {
                        j--;
                } while (pivot < arr[j]);

                if (i >= j)
                        return j;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
        }
}
```

10) Merge Sort:

```
// Author: Rajat Mishra
// Cited from: https://www.geeksforgeeks.org/merge-sort/
// Function has 3 parameters last two is the first and last
// indexes and calls 'merge' function and itself recursively
public static void mergeSort(int arr[], int l, int r) {
```

```
        if (l < r) {
                int m = (l + r) / 2;

                mergeSort(arr, l, m);
                mergeSort(arr, m + 1, r);
                merge(arr, l, m, r);
        }
    }
```

Merge Function:

```
    // Merges two temporarily made arrays into one array
    public 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;
        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
```

Snippets for each output with the array sizes 1000, 10 000, and 100 000

1) **Array Size = 1000**
    a) **Random array:**

```
<terminated> SortingAlgorithms (1) [Java Application] C:
Total time for selection sort: 20696600ns
Total time for bubble sort: 24519200ns
Total time for BSSC sort: 20416200ns
Total time for insertion sort: 194200ns
Total time for quick sort: 14182400ns
Total time for merge sort: 3916900ns
```

b) Sorted array:

```
<terminated> SortingAlgorithms (1) [Java Application] C:
Total time for selection sort: 8892700ns
Total time for bubble sort: 6911400ns
Total time for BSSC sort: 6865100ns
Total time for insertion sort: 10200ns
Total time for quick sort: 3417300ns
Total time for merge sort: 1021400ns
```

c) Almost-sorted array:

```
<terminated> SortingAlgorithms (1) [Java Application] C
Total time for selection sort: 6526600ns
Total time for bubble sort: 6839500ns
Total time for BSSC sort: 7381800ns
Total time for insertion sort: 52600ns
Total time for quick sort: 3534400ns
Total time for merge sort: 1044700ns
```

2) Array Size = 10000

a) Random array:

```
<terminated> SortingAlgorithms (1) [Java Application] C
Total time for selection sort: 77317000ns
Total time for bubble sort: 36209700ns
Total time for BSSC sort: 36536000ns
Total time for insertion sort: 463700ns
Total time for quick sort: 33944300ns
Total time for merge sort: 3162800ns
```

b) Sorted array:

```
<terminated> SortingAlgorithms (1) [Java Application] C:
Total time for selection sort: 28028600ns
Total time for bubble sort: 34738600ns
Total time for BSSC sort: 34727700ns
Total time for insertion sort: 102100ns
Total time for quick sort: 40636200ns
Total time for merge sort: 3144800ns
```

c) Almost-sorted array:

```
<terminated> SortingAlgorithms (1) [Java Application] C:\
Total time for selection sort: 77938000ns
Total time for bubble sort: 34699700ns
Total time for BSSC sort: 37311500ns
Total time for insertion sort: 457800ns
Total time for quick sort: 33274200ns
Total time for merge sort: 3261100ns
```

3) Array Size = 100000

a) Random array:

```
Total time for selection sort: 6100310800ns
Total time for bubble sort: 2359701000ns
Total time for BSSC sort: 2460869000ns
Total time for insertion sort: 3214300ns
Total time for quick sort: 873569100ns
Total time for merge sort: 21569200ns
```

b) Sorted array:

```
Total time for selection sort: 2182176900ns
Total time for bubble sort: 2547266600ns
Total time for BSSC sort: 2351235000ns
Total time for insertion sort: 3005400ns
Total time for quick sort: 1387680400ns
Total time for merge sort: 19516600ns
```

c) Almost-sorted array:

```
Total time for selection sort: 3291461000ns
Total time for bubble sort: 2638755800ns
Total time for BSSC sort: 2579682000ns
Total time for insertion sort: 3612500ns
Total time for quick sort: 1638928500ns
Total time for merge sort: 24997300ns
```