

- LINEAR SEARCH

```
• //Source: https://www.programiz.com/dsa/linear-search
• //Big O Complexity:  $O(n)$ 
• //This means the time it takes increases as the array
  gets bigger.
•
• public class LinearSearch {
•     public static int linearSearch(int array[], int x) {
•         int n = array.length;
•
•         // Go through array one element at a time
•         for (int i = 0; i < n; i++) {
•             //if number is found, return
•             if (array[i] == x)
•                 return i;
•         }
•         //if we reach the end and don't find it, return
-1
•         return -1;
•     }
• }
```

- BINARY SEARCH

```
• // BinarySearchLab.java
• // Source: https://www.geeksforgeeks.org/dsa/binary-
  search/
• // Big O Complexity:  $O(\log n)$ 
• // Explanation: Binary search works on sorted arrays. It
  divides
• // the array in half each time to find the target
  number. Each step reduces the
• // search by half, making it much faster than linear
  search for larger arrays.
•
• import java.util.Random;
• import java.util.Arrays;
•
```

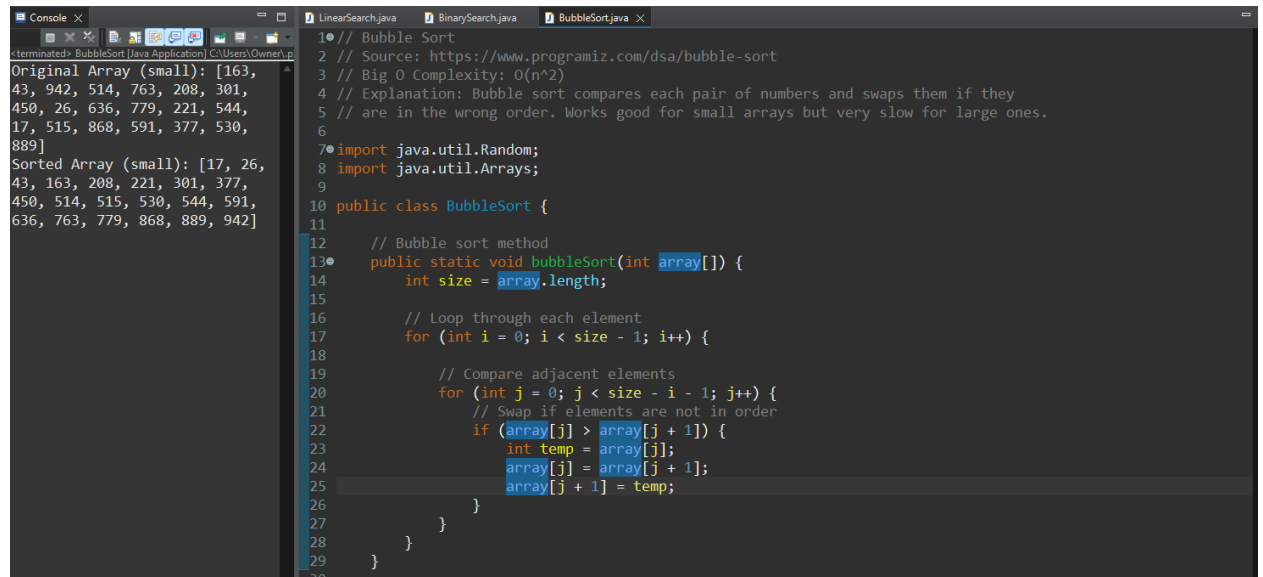
```

• public class BinarySearch {
•
•     // Binary Search method
•     public static int binarySearch(int arr[], int x) {
•         int low = 0, high = arr.length - 1;
•
•         // Keep looking while the search space is not
empty
•         while (low <= high) {
•             int mid = low + (high - low) / 2;
•
•             // Check if x is at mid
•             if (arr[mid] == x)
•                 return mid;
•
•             // If x is bigger, ignore left half
•             if (arr[mid] < x)
•                 low = mid + 1;
•
•             // If x is smaller, ignore right half
•             else
•                 high = mid - 1;
•         }
•
•         // If we reach here, x is not in the array
•         return -1;
•     }

```

- BUBBLE SORT

Screenshot 1:



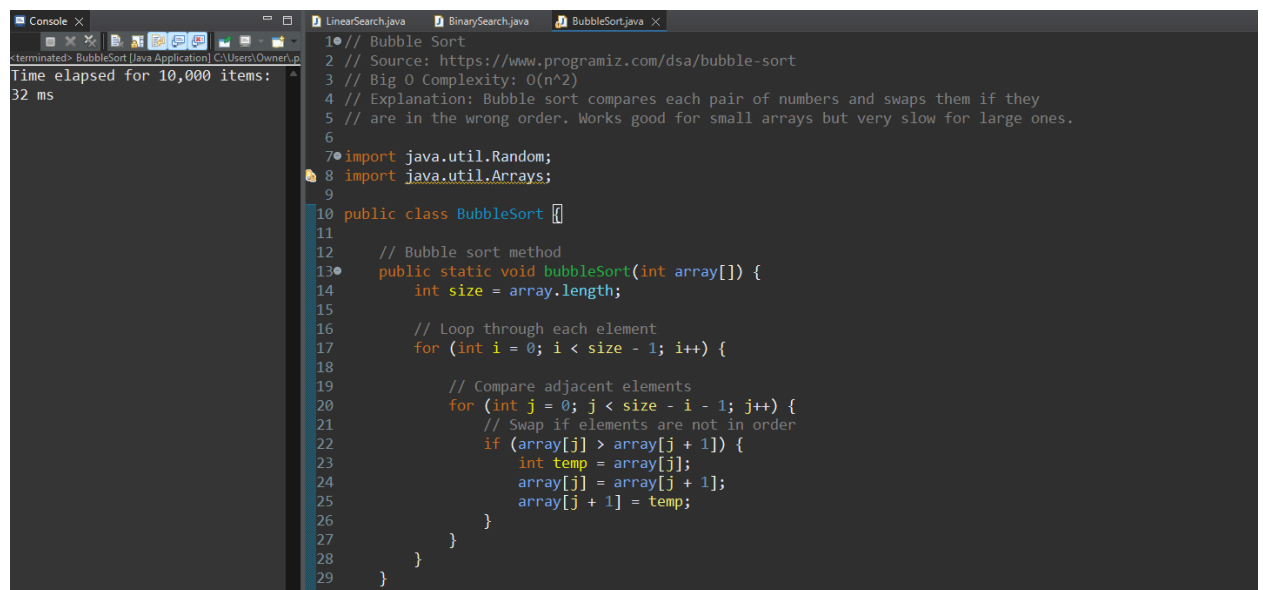
The screenshot shows an IDE with three tabs: LinearSearch.java, BinarySearch.java, and BubbleSort.java. The BubbleSort.java tab is active, displaying the following code:

```
1 // Bubble Sort
2 // Source: https://www.programiz.com/dsa/bubble-sort
3 // Big O Complexity: O(n^2)
4 // Explanation: Bubble sort compares each pair of numbers and swaps them if they
5 // are in the wrong order. Works good for small arrays but very slow for large ones.
6
7 import java.util.Random;
8 import java.util.Arrays;
9
10 public class BubbleSort {
11
12     // Bubble sort method
13     public static void bubbleSort(int array[]) {
14         int size = array.length;
15
16         // Loop through each element
17         for (int i = 0; i < size - 1; i++) {
18
19             // Compare adjacent elements
20             for (int j = 0; j < size - i - 1; j++) {
21                 // Swap if elements are not in order
22                 if (array[j] > array[j + 1]) {
23                     int temp = array[j];
24                     array[j] = array[j + 1];
25                     array[j + 1] = temp;
26                 }
27             }
28         }
29     }
30 }
```

The console on the left shows the output of the program:

```
Original Array (small): [163, 43, 942, 514, 763, 208, 301, 450, 26, 636, 779, 221, 544, 17, 515, 868, 591, 377, 530, 889]
Sorted Array (small): [17, 26, 43, 163, 208, 221, 301, 377, 450, 514, 515, 530, 544, 591, 636, 763, 779, 868, 889, 942]
```

Screenshot 2:



The screenshot shows the same IDE with the BubbleSort.java tab active. The code is identical to Screenshot 1. The console on the left shows the output of the program for a larger array:

```
Time elapsed for 10,000 items: 32 ms
```

Screenshot 3:



The screenshot shows an IDE with two tabs: 'BubbleSort.java' and 'Console'. The 'BubbleSort.java' tab is active, displaying the following code:

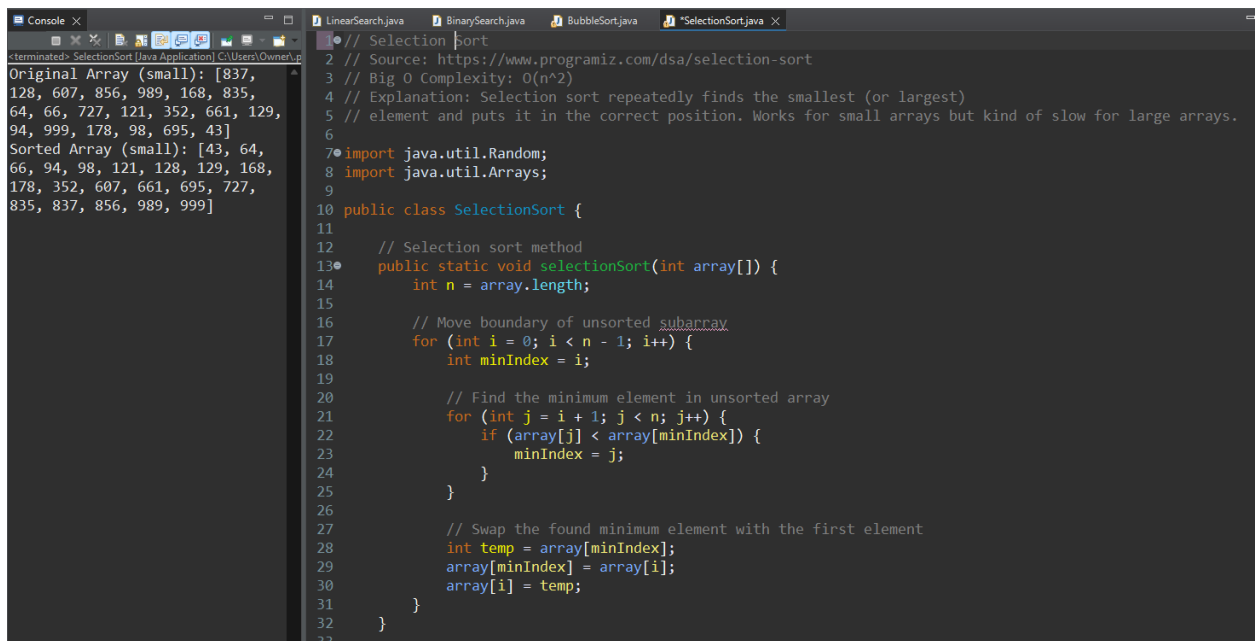
```
11 // Bubble sort method
12 public static void bubbleSort(int array[]) {
13     int size = array.length;
14
15     // Loop through each element
16     for (int i = 0; i < size - 1; i++) {
17
18         // Compare adjacent elements
19         for (int j = 0; j < size - i - 1; j++) {
20             // Swap if elements are not in order
21             if (array[j] > array[j + 1]) {
22                 int temp = array[j];
23                 array[j] = array[j + 1];
24                 array[j + 1] = temp;
25             }
26         }
27     }
28 }
29 }
```

The 'Console' tab shows the output of the program:

```
<terminated> BubbleSort [Java Application] C:\Users\Owner\p
Time elapsed for 20000 items:
177 ms
```

- SELECTION SORT

Screenshot 4:



The screenshot shows an IDE with two tabs: 'SelectionSort.java' and 'Console'. The 'SelectionSort.java' tab is active, displaying the following code:

```
1 // Selection sort
2 // Source: https://www.programiz.com/dsa/selection-sort
3 // Big O Complexity: O(n^2)
4 // Explanation: Selection sort repeatedly finds the smallest (or largest)
5 // element and puts it in the correct position. Works for small arrays but kind of slow for large arrays.
6
7 import java.util.Random;
8 import java.util.Arrays;
9
10 public class SelectionSort {
11
12     // Selection sort method
13     public static void selectionSort(int array[]) {
14         int n = array.length;
15
16         // Move boundary of unsorted subarray
17         for (int i = 0; i < n - 1; i++) {
18             int minIndex = i;
19
20             // Find the minimum element in unsorted array
21             for (int j = i + 1; j < n; j++) {
22                 if (array[j] < array[minIndex]) {
23                     minIndex = j;
24                 }
25             }
26
27             // Swap the found minimum element with the first element
28             int temp = array[minIndex];
29             array[minIndex] = array[i];
30             array[i] = temp;
31         }
32     }
33 }
```

The 'Console' tab shows the output of the program:

```
<terminated> SelectionSort [Java Application] C:\Users\Owner\p
Original Array (small): [837, 128, 607, 856, 989, 168, 835, 64, 66, 727, 121, 352, 661, 129, 94, 999, 178, 98, 695, 43]
Sorted Array (small): [43, 64, 66, 94, 98, 121, 128, 129, 168, 178, 352, 607, 661, 695, 727, 835, 837, 856, 989, 999]
```

Screenshot 5:

The screenshot shows an IDE with a console window on the left and a code editor on the right. The console window displays the output of a Java application: "Time elapsed for 10,000 items: 18 ms". The code editor shows the implementation of the Selection Sort algorithm in Java. The code includes comments explaining the algorithm's source, complexity, and operation. It imports `java.util.Random` and `java.util.Arrays`. The `SelectionSort` class contains a `selectionSort` method that iterates through the array, finding the minimum element and swapping it with the first element of the unsorted subarray.

```
1 // Selection Sort
2 // Source: https://www.programiz.com/dsa/selection-sort
3 // Big O Complexity: O(n^2)
4 // Explanation: Selection sort repeatedly finds the smallest (or largest)
5 // element and puts it in the correct position. Works for small arrays but kind of slow for large arrays.
6
7 import java.util.Random;
8 import java.util.Arrays;
9
10 public class SelectionSort {
11
12     // Selection sort method
13     public static void selectionSort(int array[]) {
14         int n = array.length;
15
16         // Move boundary of unsorted subarray
17         for (int i = 0; i < n - 1; i++) {
18             int minIndex = i;
19
20             // Find the minimum element in unsorted array
21             for (int j = i + 1; j < n; j++) {
22                 if (array[j] < array[minIndex]) {
23                     minIndex = j;
24                 }
25             }
26
27             // Swap the found minimum element with the first element
28             int temp = array[minIndex];
29             array[minIndex] = array[i];
30             array[i] = temp;
31         }
32     }
33 }
```

Screenshot 6:

The screenshot shows the same IDE as Screenshot 6, but with the console window displaying the output for 20,000 items: "Time elapsed for 20,000 items: 54 ms". The code editor shows the same Selection Sort implementation as in Screenshot 6.

```
1 // Selection Sort
2 // Source: https://www.programiz.com/dsa/selection-sort
3 // Big O Complexity: O(n^2)
4 // Explanation: Selection sort repeatedly finds the smallest (or largest)
5 // element and puts it in the correct position. Works for small arrays but kind of slow for large arrays.
6
7 import java.util.Random;
8 import java.util.Arrays;
9
10 public class SelectionSort {
11
12     // Selection sort method
13     public static void selectionSort(int array[]) {
14         int n = array.length;
15
16         // Move boundary of unsorted subarray
17         for (int i = 0; i < n - 1; i++) {
18             int minIndex = i;
19
20             // Find the minimum element in unsorted array
21             for (int j = i + 1; j < n; j++) {
22                 if (array[j] < array[minIndex]) {
23                     minIndex = j;
24                 }
25             }
26
27             // Swap the found minimum element with the first element
28             int temp = array[minIndex];
29             array[minIndex] = array[i];
30             array[i] = temp;
31         }
32     }
33 }
```

- INSERTION SORT

Screenshot 7:

This screenshot shows an IDE with a Java application named 'InsertionSort'. The console on the left displays the output of the program, showing an original array of 20 numbers and a sorted array. The code editor on the right shows the implementation of the InsertionSort algorithm. The code includes comments about the source and complexity, imports for Random and Arrays, and the main logic of the insertion sort method.

```
1 // Insertion Sort
2 // Source: https://www.geeksforgeeks.org/dsa/insertion-sort-algorithm/
3 // Big O Complexity: O(n^2)
4 // Explanation: Insertion sort builds the sorted array one element at a time.
5 // Works well for small arrays but becomes slow for large arrays.
6
7 import java.util.Random;
8 import java.util.Arrays;
9
10 public class InsertionSort {
11
12     // Insertion sort method
13     public static void insertionSort(int arr[]) {
14         int n = arr.length;
15         for (int i = 1; i < n; i++) {
16             int key = arr[i];
17             int j = i - 1;
18
19             // Move elements greater than key one position ahead
20             while (j >= 0 && arr[j] > key) {
21                 arr[j + 1] = arr[j];
22                 j--;
23             }
24             arr[j + 1] = key;
25         }
26     }
27 }
```

Console Output:

```
<terminated> InsertionSort [Java Application] C:\Users\Owner\p
Original Array (small): [619, 620, 272, 646, 499, 974, 554, 222, 909, 273, 172, 912, 66, 77, 786, 707, 544, 777, 814, 686]
Sorted Array (small): [66, 77, 172, 222, 272, 273, 499, 544, 554, 619, 620, 646, 686, 707, 777, 786, 814, 909, 912, 974]
```

Screenshot 8:

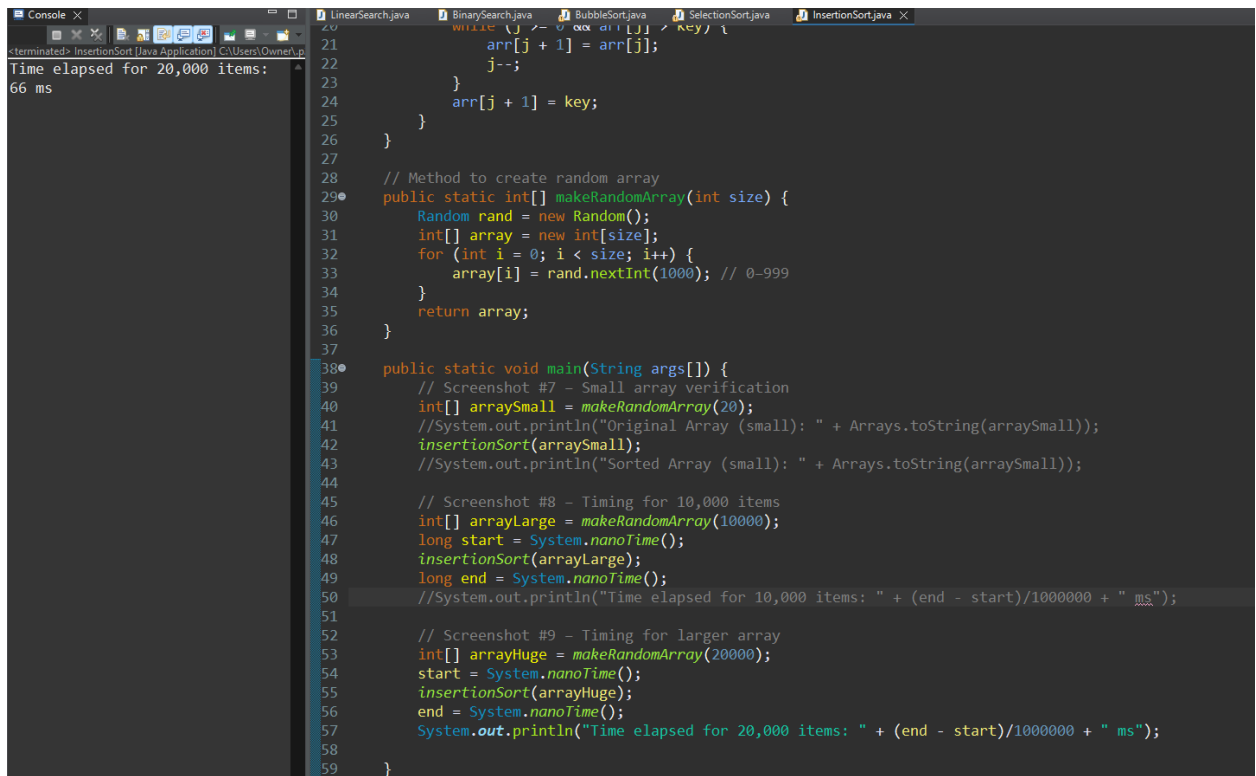
This screenshot shows the same IDE with the 'InsertionSort' application. The console now shows the time elapsed for 10,000 items. The code editor shows the continuation of the InsertionSort class, including a method to create a random array and a main method that tests the algorithm with different array sizes and measures execution time.

```
21         while (j >= 0 && arr[j] > key) {
22             arr[j + 1] = arr[j];
23             j--;
24         }
25         arr[j + 1] = key;
26     }
27
28     // Method to create random array
29     public static int[] makeRandomArray(int size) {
30         Random rand = new Random();
31         int[] array = new int[size];
32         for (int i = 0; i < size; i++) {
33             array[i] = rand.nextInt(1000); // 0-999
34         }
35         return array;
36     }
37
38     public static void main(String args[]) {
39         // Screenshot #7 - Small array verification
40         int[] arraySmall = makeRandomArray(20);
41         //System.out.println("Original Array (small): " + Arrays.toString(arraySmall));
42         insertionSort(arraySmall);
43         //System.out.println("Sorted Array (small): " + Arrays.toString(arraySmall));
44
45         // Screenshot #8 - Timing for 10,000 items
46         int[] arrayLarge = makeRandomArray(10000);
47         long start = System.nanoTime();
48         insertionSort(arrayLarge);
49         long end = System.nanoTime();
50         System.out.println("Time elapsed for 10,000 items: " + (end - start)/1000000 + " ms");
51
52         // Screenshot #9 - Timing for larger array
53         int[] arrayHuge = makeRandomArray(20000);
54         start = System.nanoTime();
55         insertionSort(arrayHuge);
56         end = System.nanoTime();
57         //System.out.println("Time elapsed for 20,000 items: " + (end - start)/1000000 + " ms");
58
59     }
60 }
```

Console Output:

```
<terminated> InsertionSort [Java Application] C:\Users\Owner\p
Time elapsed for 10,000 items:
19 ms
```

Screenshot 9:



The screenshot shows an IDE with several tabs: LinearSearch.java, BinarySearch.java, BubbleSort.java, SelectionSort.java, and InsertionSort.java. The InsertionSort.java tab is active, displaying the following code:

```
20
21     while (j > 0 && arr[j] < key) {
22         arr[j + 1] = arr[j];
23         j--;
24     }
25     arr[j + 1] = key;
26 }
27
28 // Method to create random array
29 public static int[] makeRandomArray(int size) {
30     Random rand = new Random();
31     int[] array = new int[size];
32     for (int i = 0; i < size; i++) {
33         array[i] = rand.nextInt(1000); // 0-999
34     }
35     return array;
36 }
37
38 public static void main(String args[]) {
39     // Screenshot #7 - Small array verification
40     int[] arraySmall = makeRandomArray(20);
41     //System.out.println("Original Array (small): " + Arrays.toString(arraySmall));
42     insertionSort(arraySmall);
43     //System.out.println("Sorted Array (small): " + Arrays.toString(arraySmall));
44
45     // Screenshot #8 - Timing for 10,000 items
46     int[] arrayLarge = makeRandomArray(10000);
47     long start = System.nanoTime();
48     insertionSort(arrayLarge);
49     long end = System.nanoTime();
50     //System.out.println("Time elapsed for 10,000 items: " + (end - start)/1000000 + " ms");
51
52     // Screenshot #9 - Timing for larger array
53     int[] arrayHuge = makeRandomArray(20000);
54     start = System.nanoTime();
55     insertionSort(arrayHuge);
56     end = System.nanoTime();
57     System.out.println("Time elapsed for 20,000 items: " + (end - start)/1000000 + " ms");
58 }
59 }
```

The console window on the left shows the output: "Time elapsed for 20,000 items: 66 ms".

- SUMMARY AND CONCLUSIONS

- For this lab, I tested bubble selection, and insertion sort. On small arrays, all three sorted correctly and quickly, but with larger arrays, timing shows big differences. Bubble sorts were the slowest, then selection then insertions being the fastest. All three algorithms have a Big O complexity of $O(n^2)$, meaning their time grows quickly as the array gets bigger. Using arrays with 10,000 elements or more was necessary to see some differences and see how the algorithms efficiency affects performance.