Selection Sort in Java

Last modified: May 23, 2022

Written by: Nikunj Gandhi (https://www.baeldung.com/author/nikunj-gandhi)

Algorithms (https://www.baeldung.com/category/algorithms)

Java (https://www.baeldung.com/category/java) +

1. Introduction

In this tutorial, we'll **learn Selection Sort**, see its implementation in Java, and analyze its performance.

2. Algorithm Overview

Selection Sort **begins with the element in the 1st position of** an unsorted array and scans through subsequent elements to **find the smallest element**. Once found, the smallest element is swapped with the element in the 1st position.

The algorithm then moves on to the element in the 2nd position and scans through subsequent elements to find the index of the 2nd smallest element. Once found, the second smallest element is swapped with the element in the 2nd position.

This process goes on until we reach the n-1th element of the array, which puts the n-1th smallest element in the n-1th position. The last element automatically falls in place, in the n-1th iteration, thereby sorting the array.

We find **the largest element instead of the smallest element to sort the array in descending order.**

Let's see an example of an unsorted array and sort it in ascending order to visually understand the algorithm.

2.1. An Example

Consider the following unsorted array:

int[] arr = {5, 4, 1, 6, 2}

Iteration 1

Considering the above working of the algorithm, we start with the element in 1st position – 5 – and scan through all subsequent elements to find the smallest element – 1. We then swap the smallest element with the element in 1st position.

The modified array nows looks like:

{1, 4, 5, 6, 2}

Total comparisons made: 4

Iteration 2

In the second iteration, we move on to the 2nd element – 4 – and scan through subsequent elements to find the second smallest element – 2. We then swap the second smallest element with the element in 2nd position.

The modified array now looks like:

{1, 2, 5, 6, 4}

Total comparisons made: 3

Continuing similarly, we have the following iterations:

Iteration 3

{1, 2, 4, 6, 5}

Total comparisons made: 2

Iteration 4

{1, 2, 4, 5, 6}

Total comparisons made: 1

3. Implementation

Let's implement Selection Sort using a couple of *for* loops:

```
public static void sortAscending(final int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int minElementIndex = i;
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[minElementIndex] > arr[j]) {
                minElementIndex = j;
            }
        }

        if (minElementIndex != i) {
            int temp = arr[i];
            arr[i] = arr[minElementIndex];
            arr[minElementIndex] = temp;
        }
    }
}
```

Of course, to reverse it we could do something quite similar:

```java
public static void sortDescending(final int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int maxElementIndex = i;
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[maxElementIndex] < arr[j]) {
                maxElementIndex = j;
            }
        }

        if (maxElementIndex != i) {
            int temp = arr[i];
            arr[i] = arr[maxElementIndex];
            arr[maxElementIndex] = temp;
        }
    }
}
```

And with a bit more elbow grease, we could combine these using *Comparator*s (/java-comparator-comparable).

4. Performance Overview

4.1. Time

In the example that we saw earlier, **selecting the smallest element required a total of *(n-1)* comparisons** followed by swapping it to the 1st position. Similarly, **selecting the next smallest element required total *(n-2)*** comparisons followed by swapping in the 2nd position, and so on.

Thus, starting from index 0, we perform *n-1, n-2, n-3, n-4 .... 1* comparisons. The last element automatically falls in place due to previous iterations and swaps.

Mathematically, the **sum of the first *n-1* natural numbers** will tell us how many comparisons we need in order to sort an array of size *n* using Selection Sort.

The formula for the sum of *n* natural numbers is *n(n+1)/2*.

In our case, we need the sum of first *n-1* natural numbers. Therefore, we replace *n* with *n-1* in the above formula to get:

(n-1)(n-1+1)/2 = (n-1)n/2 = ***(n^2-n)/2***

As *n^2* grows prominently as *n* grows, we consider the higher power of *n* as the performance benchmark, making this algorithm have a **time complexity of *O(n^2)*.**

4.2. Space

In terms of auxiliary space complexity, Selection Sort requires one extra variable to hold the value temporarily for swapping. Therefore, Selection Sort's **space complexity is *O(1)*.**

5. Conclusion

Selection Sort is a very simple sorting algorithm to understand and implement. Unfortunately, **its quadratic time complexity makes it an expensive sorting technique**. Also, since the algorithm has to scan through each element, **the best case, average case, and worst-case time complexity is the same**.

Other sorting techniques like Insertion Sort (/java-insertion-sort) and Shell Sort (/java-shell-sort) also have quadratic worst-case time complexity, but they perform better in best and average cases.

Check out the complete code for Selection Sort over on GitHub (https://github.com/eugenp/tutorials/tree/master/algorithms-modules/algorithms-sorting).

Comments are closed on this article!