# Lab 4 Selection Sort

## Goal

In this lab the performance of a basic sorting algorithm will be explored.

## Resources

* Chapter 10

## Java Files

* SortArray.java
* SortDriver.java

## Introduction

Sorting is an important basic operation used by many applications. A lot of effort has been spent creating fast sorting algorithms. In this lab, the performance of selection sorting algorithm will be measured. Since better algorithms are known, these algorithm is of limited use.

One of the difficulties in measuring the performance of the Fibonacci computation from the Recursion I Lab was that different computers execute programs at different speeds. One method for dealing with this problem is to find a core operation that the algorithm performs. The performance of the algorithm will be determined by counting the number of times that operation is performed. This will give a way of comparing two algorithms that is independent of the computer the algorithm runs on.

For general purpose sorting algorithms, the standard measure is the number of comparisons that are made.

### The Statistics

To get a fair view of the performance of a sort, it is not enough to just try it on one array. Instead its performance for all possible permutations of the data values in an array should be examined. For many sorts, the average can be computed mathematically. To do this experimentally, though is impractical for all but the smallest of arrays as the number of permutations grows exponentially. In practice, we will test the performance on a number of randomly generated arrays A1, A2, A3, ... Ak. For each array, the number of comparisons made will be counted giving values C1, C2, C3, ... Ck. From these data values, the average, minimum, and maximum number of comparisons will be computed.

The computed average number of comparisons will give an approximation to the true average number of comparisons over all array permutations for the algorithm. Assuming that the generation of the arrays is truly random, larger values of k will lead to a closer approximation to the true average.

The minimum and maximum give an indication of how consistent the performance is. They also give a rough indication of what the best and worst cases, respectively, are. Be aware, though, that the number of possible permutations of an array of size n is n factorial. If relatively few of the cases lead to best- or worst-case behavior, they are unlikely to show up in the randomly chosen test cases.

In characterizing the performance of an algorithm, usually only the worst and average cases are cited. The worst case allows you to guarantee the performance of an algorithm. The average gives the expected performance. We many even be willing to tolerate a bad worst case if the average is good, especially if the algorithm will be executed many times.

## Pre-Lab Visualization

### Computing Statistics

Consider an object, which is given the values C1, C2, C3, ... Ck one at a time. In other words, it has a method giveValue(c). It is not allowed to keep the values in an array, but it can have a limited number of private variables. After each call to the method giveValue(), the object is required to know the current minimum, maximum, and average.

What private variables should it use?

images:bighand.jpg

Create an algorithm for giveValue(c).

images:bighand.jpg

Test it on the sequence

images:bighand.jpg

giveValue(-10)

Test it on the sequence

images:bighand.jpg

giveValue(7)

Test it on the sequence

images:bighand.jpg

giveValue(3)

giveValue(2)

giveValue(-5)

giveValue(10)

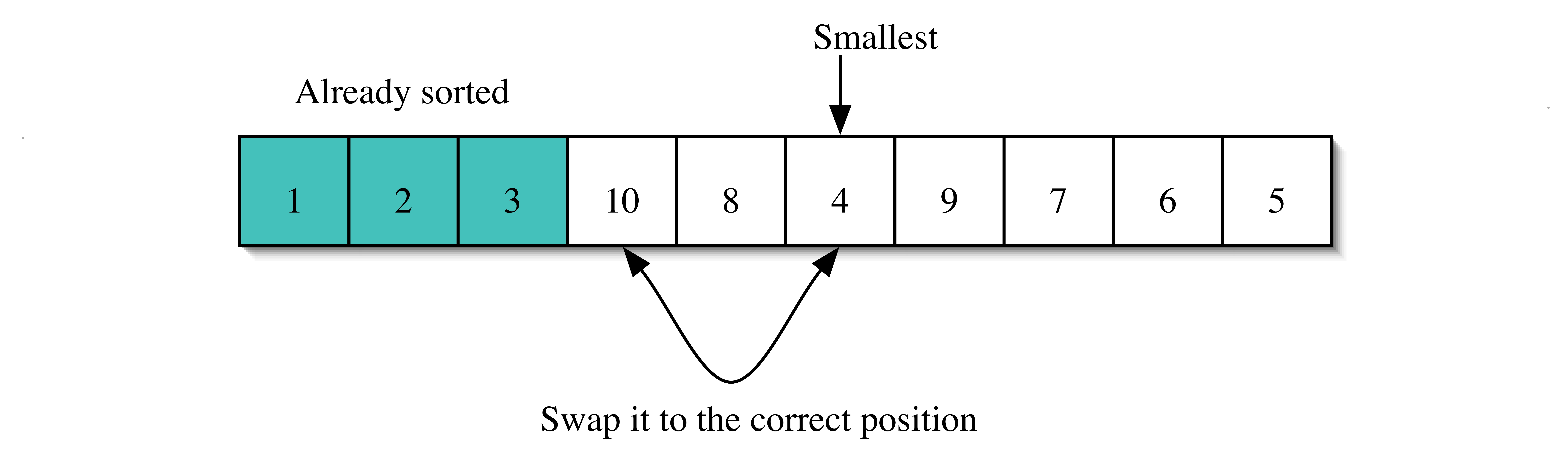
giveValue(33)

giveValue(1)

### Predicting the Average Performance of Selection Sort

In selection sort, the array is divided in two parts. The first part has sorted values. The second part has values that are in arbitrary order. All the values in the first part are less than the values in the second part. At each phase of the algorithm, the smallest value in the second part is found (selected) and swapped to the end of the first part.

The following picture shows the state of selection sort at an intermediate step.



Suppose someone told you the position of the smallest value in the second portion of the array. How many values must be checked to verify that 4 is the smallest value? (Does 4 need to be checked against itself?)

images:bighand.jpg

While this is not the way selection sort finds the 4, it does give a minimum for the number of comparisons required. In fact, selection sort will use that number of comparisons.

**The general case of n values:**

How many comparisons are needed in the first pass if there are n values in the array?

images:bighand.jpg

How many comparisons are needed in the second pass if there are n values in the array?

images:bighand.jpg

How many comparisons are needed in the third pass if there are n values in the array?

images:bighand.jpg

What is the sequence of comparisons needed?

images:bighand.jpg

What is the sum of the sequence?

images:bighand.jpg

For n=20, how many comparisons are required?

images:bighand.jpg

Remember this value for the lab.

***Directed Lab Work***

The selection sort have already been implemented in the SortArray class. You will make a new class SortArrayInstrumented that will be based on that class. It will allow you to gather statistics about the selection sort. The SortDriver class will generate the arrays, call the sort, and then display the statistical results.

### Adding Statistics to Selection Sort

1. If you have not done so, look at the implementation of the sort in *SortArray.java*. Look at the skeleton in *SortDriver.java*. Compile the classes SortArray, and SortDriver. Run the main method in SortDriver.

*Checkpoint: The program will ask you for an array size. Enter 20. An array of 20 random values between 0 and 20 should be generated and displayed. Selection sort will be applied to array and the sorted array will be displayed. Verify that this works correctly.*

*The first goal is to create a new class SortArrayInstrumented that will be used to collect statistics about the performance of the sort. Private variables of the class will be used to record the number of comparisons made.*

1. Create a new class name SortArrayInstrumented.
2. Copy the contents of SortArray into SortArrayInstrumented. Change the name in the class declaration from SortArray to SortArrayInstrumented.
3. Create a default constructor that does nothing. (It will have work to do later.)
4. Remove static from all the methods in the SortArrayInstrumented class.

*Checkpoint: You should be able to compile* SortArrayInstrumented *without errors.*

*Since the sort methods are no longer static, SortDriver must be changed to create an instance of SortArrayInstrumented and then invoke the sort method using the instance.*

1. In main of sortDriver declare and create a new instance of SortArrayInstrumented named sai.
2. Change SortArray.selectionSort(data, arraySize) to sai.selectionSort(data, arraySize).

*Checkpoint: Compile and run the program. Enter 20 for the array size. Verify that this works correctly.*

*The next goal is to add code to the selection sort to count the number of times that a comparison of data values is made. Methods will be added to the SortArrayInstrumented class to allow the number of comparisons to be recovered.*

1. Add a private variable comparisons of type long to the SortArrayInstrumented class. Initialize it to zero in the constructor.
2. Add a public accessor method getComparisons to the SortArrayInstrumented class.
3. In order to count the number of times that compareTo() is called by selection sort, put the line

comparisons++;

just before the if statement in indexOfSmallest(). If the code is inserted inside the then clause, only the comparisons that result in true will be counted.

1. In SortDriver, add the line

System.out.println(" comparison made: "+sai.getComparisons());

after the call to selection sort.

*Checkpoint: Compile and run the program. Enter 20 for the array size. Verify that the sort still works correctly. The number of comparisons should be 190.*

*The next goal is to compute the average number of comparisons made by the sort with many different arrays (all of the same size). Only SortDriver will be changed.*

1. In SortDriver, use the method getInt() to set the variable trials.
2. Starting with the call to generateRandomArray, wrap the remainder of the code in main in SortDriver with a for loop that runs the given number of trials.

*Checkpoint: Compile and run the program. Enter 20 for the array size. Enter 3 for the number of trials. Verify that each of the three trials sorted the values correctly and is for a different array of 20 values. The number of comparisons should be 190, 380, and 570.*

*Notice that the number of comparisons gives a running total for all calls. The next goal is to compute and report the minimum and maximum number of comparisons made over all the calls to the sort. To do this, the use of the comparisons variable will be changed slightly. It will only be the number of comparisons made by the last call to the sort. The total number of comparisons made by all calls will be held in a new variable. This aids in the computation of the maximum and minimum.*

1. Add a private variable totalComparisons of type long to the SortArrayInstrumented class. Initialize it to zero in the constructor.
2. Add a private variable minComparisons of type long to the SortArrayInstrumented class. Initialize it to Long.MAX\_VALUE in the constructor.
3. Add a private variable maxComparisons of type long to the SortArrayInstrumented class. Initialize it to zero in the constructor.
4. Add three public accessor methods (one for each of the new variables) to the SortArrayInstrumented class.

*To compute the minimum and maximum number of comparisons, code needs to be added at the beginning and end of the sort. While the needed code could be added directly to the sort, it is better to encapsulate it in a couple new methods.*

1. Add a private method startStatistics() to the SortArrayInstrumented class. It should initialize comparisons to zero.
2. Add a private method endStatistics() to the SortArrayInstrumented class. It should add comparisons to totalComparisons. It should compare comparisons to minComparisons and set minComparisons to whichever is smaller. It should also set maxComparisons in an analogous fashion.
3. Call startStatistics() at the beginning of the selectionSort method. Call endStatistics() at the end of the selectionSort method.
4. After the for loop in main of SortDriver, add in three statements that print the total, minimum, and maximum number of comparisons.

*Checkpoint: Compile and run the program. Enter 20 for the array size. Enter 3 for the number of trials. Verify that each of the three trials sorted the values correctly and is for a different array of 20 values. The number of comparisons should be 190 for each of the three trials. The total should be 570 and the minimum and maximum should both be 190. Refer to the pre-lab exercises and compare.*

*Enter 10 for the array size. Enter 3 for the number of trials. Verify that each of the three trials sorted the values correctly and is for a different array of 10 values. The number of comparisons should be 45 for each of the three calls. The total should be 135 and the minimum and maximum should both be 45.*

1. Compute the average number of comparisons made over the trials and print it. (The average is the total number of comparisons divided by the number of trials.)
2. In preparation for filling in the table, comment out the print statements inside the for loop in main.

*Final checkpoint: Compile and run the program. Enter 20 for the array size. Enter 1000 for the number of trials. The total should be 19000 and the average, minimum, and maximum should all be 190.*

1. Fill in this table and record the average in the appropriate column in the table at the end of the directed lab. Use 100 trials.

**Comparisons for Selection Sort**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Minimum Comparisons | Average Comparisons | maximum Comparisons |
| Size=10 |  |  |  |
| Size=50 |  |  |  |
| Size=100 |  |  |  |
| Size=200 |  |  |  |
| Size=300 |  |  |  |
| Size=400 |  |  |  |
| Size=500 |  |  |  |
| Size=750 |  |  |  |
| Size=1000 |  |  |  |