

Lab 8 Basic Sorts

Goal

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

Resources

- Chapter 8: An Introduction to Sorting

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 three basic sorting algorithms will be measured. Since better algorithms are known, these algorithms are of limited use. (The performance of two faster algorithms, merge sort and quick sort, will be examined in the next lab.)

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 $A_1, A_2, A_3, \dots A_k$. For each array, the number of comparisons made will be counted giving values $C_1, C_2, C_3, \dots C_k$. 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 may 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 $C_1, C_2, C_3, \dots, C_k$ 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?



Create an algorithm for `giveValue(c)`.




Test it on the sequence


`giveValue(-10)`

Test it on the sequence


`giveValue(7)`

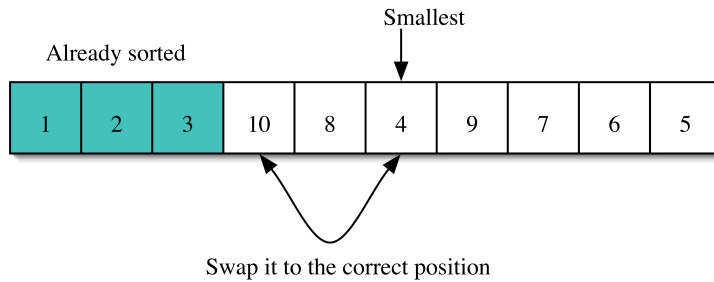
Test it on the sequence


`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?)



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?



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



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



What is the sequence of comparisons needed?



What is the sum of the sequence?



For $n=20$, how many comparisons are required?

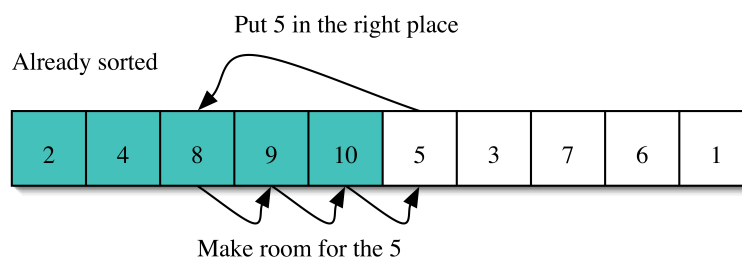


Remember this value for the lab.

Predicting the Average Performance of Insertion Sort

In insertion sort, the array is again divided into two parts. The first part has sorted values. The second part has values that are in arbitrary order. Unlike selection sort, the values in the first part may be larger than values in the second part. At each phase of the algorithm, the next value in the unsorted part is inserted into the correct place in the sorted part.

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



What is the largest number of values that the 5 must be checked against?



What is the smallest number of values that the 5 must be checked against?



If all values from the largest to smallest are equally possible, what is the average number of comparisons?



The general case of n values:

It is easier to think about the phases of insertion sort in reverse order.

What is the average number of comparisons needed in the last pass if there are n values in the array?



What is the average number of comparisons needed in the second to last pass if there are n values?



What is the average number of comparisons needed in the third to last pass if there are n values?



What is the average number of comparisons needed in the third pass if there are n values in the array?
(3 items already in the sorted part)



What is the average number of comparisons needed in the second pass if there are n values in the array?
(2 items already in the sorted part)



What is the average number of comparisons needed in the first pass if there are n values in the array?
(1 item already in the sorted part)



What is the sequence of comparisons needed?



What is the sum of the sequence?



For $n=20$, how many comparisons are required on average?



Remember this value for the lab.

Directed Lab Work

The basic sorts 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 sorts. The `SortDriver` class will generate the arrays, call the sorts, and then display the statistical results.

Adding Statistics to Selection Sort

Step 1. If you have not done so, look at the implementation of the sorts 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 sorts. Private variables of the class will be used to record the number of comparisons made.

Step 2. Create a new class name `SortArrayInstrumented`.

Step 3. Copy the contents of `SortArray` into `SortArrayInstrumented`. Change the name in the class declaration from `SortArray` to `SortArrayInstrumented`.

Step 4. Create a default constructor that does nothing. (It will have work to do later.)

Step 5. 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.

Step 6. In main of `sortDriver` declare and create a new instance of `SortArrayInstrumented` named `sai`.

Step 7. 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.

Step 8. Add a private variable `comparisons` of type `long` to the `SortArrayInstrumented` class. Initialize it to zero in the constructor.

Step 9. Add a public accessor method `getComparisons` to the `SortArrayInstrumented` class.

Step 10. 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.

Step 11. 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.

Step 12. In `SortDriver`, use the method `getInt()` to set the variable `trials`.

Step 13. 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.

Step 14. Add a private variable `totalComparisons` of type `long` to the `SortArrayInstrumented` class. Initialize it to zero in the constructor.

Step 15. Add a private variable `minComparisons` of type `long` to the `SortArrayInstrumented` class. Initialize it to `Long.MAX_VALUE` in the constructor.

Step 16. Add a private variable `maxComparisons` of type `long` to the `SortArrayInstrumented` class. Initialize it to zero in the constructor.

Step 17. 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 sorts, it is better to encapsulate it in a couple new methods.

Step 18. Add a private method `startStatistics()` to the `SortArrayInstrumented` class. It should initialize comparisons to zero.

Step 19. 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.

Step 20. Call `startStatistics()` at the beginning of the `selectionSort` method. Call `endStatistics()` at the end of the `selectionSort` method.

Step 21. 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.

Step 22. 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.)

Step 23. 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.

Step 24. 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	45	45	45
Size=50	1225	1225	1225
Size=100	4950	4950	4950
Size=200	19900	19900	19900
Size=300	44850	44850	44850
Size=400	79800	79800	79800
Size=500	124750	124750	124750
Size=750	280875	280875	280875
Size=1000	499500	499500	499500

Adding Statistics to Recursive Insertion Sort

Most of the work needed has been done before. It is now just a matter of adding the appropriate code to the insertion sort code.

Step 25. Add calls to `startStatistics()` and `endStatistics()` to the public, nonrecursive `insertionSort()` method.

Step 26. In the `insertInOrder()` method place code to add one to `comparisons` when `compareTo()` is invoked.

Step 27. In `main` in `SortDriver`, change the call from `selectionSort` to `insertionSort`.

Step 28. Uncomment the print statements in the `for` loop in `main` in `SortDriver`.

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 typically be in the range of 85 to 130 for each of the three with an average of approximately 107. If you get values that are outside this range, retry the test a few times. If you consistently get results outside the range, check the code you added for errors. Verify that the total, minimum, and maximum are correct for the reported number of comparisons.

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 approximately 28 for each of the three trials. Verify that the total, minimum, and maximum are correct for the reported number of comparisons.

Step 29. Recommend the print statements from the previous step.

Final checkpoint: Compile and run the program. Enter 20 for the array size. Enter 10000 for the number of trials. The average should be approximately 107.

Step 30. 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.

Warning: Depending on the computer you are using, there may be a limit on the number of recursive calls that can be made. If this happens you will get the error java.lang.StackOverflowError. While this often indicates that you have entered an infinite recursion, as long as the sort worked for a smaller array, that is not the case here. If we examine the pattern of calls, we see that we need to make one recursive call for each entry in the array. As you apply the algorithm to larger and larger arrays, the size of the stack must be larger and we can hit the upper limit. If this happens, find the limit and mark it on the table.

Comparisons for Insertion Sort

	MINIMUM COMPARISONS	AVERAGE COMPARISONS	MAXIMUM COMPARISONS
Size=10	17	27	40
Size=50	494	649	787
Size=100	2038	2547	2948
Size=200	8840	10053	11370
Size=300	20455	22764	25193
Size=400	36490	40297	43054
Size=500	58214	62822	68589
Size=750	132308	141569	149340
Size=1000	229081	249754	266346

Adding Statistics to Shell Sort

Step 1. Add calls to `startStatistics()` and `endStatistics()` to the public `shellSort()` method.

Step 2. In the `incrementalInsertionSort()` method place code to add one to `comparisons` when `compareTo()` is invoked. Since the comparison is in the condition of a `while` loop, this is a bit trickier to handle for than with the other two sorts. Certainly the `compareTo` method was invoked once for each time the body of the loop executed. `compareTo` will have been invoked one more time if the first clause in the condition (`index ≥ first`) was true and the result of the `compareTo` method caused the loop to finish. Make sure you count both possibilities.

Step 3. In `main` in `SortDriver`, change the call from `insertionSort` to `shellSort`.

Step 4. Uncomment the print statements in the `for` loop in `main` in `SortDriver`.

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 typically be in the range of 73 to 94 for each of the three with an average of approximately 85. Verify that the total, minimum, and maximum are correct for the reported number of comparisons.

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 approximately 28 for each of the three trials. Verify that the total, minimum, and maximum are correct for the reported number of comparisons.

Step 5. Recommend the print statements from the previous step.

Final checkpoint: Compile and run the program. Enter 20 for the array size. Enter 10000 for the number of trials. The average should be approximately 85.

Step 6. Fill in this table and record the average in the appropriate column in the table below.

Comparisons for Shell Sort

	MINIMUM COMPARISONS	AVERAGE COMPARISONS	MAXIMUM COMPARISONS
Size=10	25	31	38
Size=50	306	350	416
Size=100	811	890	1023
Size=200	2045	2207	2428
Size=300	3347	3572	3802
Size=400	5090	5389	5722
Size=500	6038	6439	6941
Size=750	10304	10792	11432
Size=1000	14577	16325	15391

Average Comparisons for All Three Sorts

	SELECTION SORT	INSERTION SORT	SHELL SORT
Size=10	45	27	31
Size=50	1225	649	350
Size=100	4950	2547	890
Size=200	19900	10053	2207
Size=300	44850	22764	3572
Size=400	44850	40297	5389
Size=500	124750	62822	6439
Size=750	124750	141569	10792
Size=1000	499500	249754	16325

Post-Lab Follow-Ups

1. Add a `reset()` method to `SortArrayInstrumented` that will set each of the variables as the constructor does. Modify `SortDriver` to compute the average, minimum, and maximum for each of the three sorts with the input array size and number of trials.
2. Add variables and methods to `SortArrayInstrumented` to compute the total of the squares of the number of comparisons. If the number of comparisons made by three calls were 3, 5, and 2, the sum of the squares would be $9 + 25 + 4$. The variance of k values is the average of the squares of the values minus the square of the average. For the given values, the average is $10/3$ and the variance is $38/3 - (10/3)^2$. The standard deviation is the square root of the variance. Use this to compute and display the standard deviation in `SortDriver`.
3. Another way of measuring the performance of a sort is by the amount of data movement it must do. Anytime an assignment is made using the array, add one to the number of moves. For example, a swap operation would add 3 to the number of moves. Add variables and methods to `SortArrayInstrumented` to compute the total, minimum, and maximum number of moves. Add code to `SortDriver` to display them.

Note: This measure is relatively unimportant in Java since the sorts work with arrays of references to objects. Because of this, only references are being moved and not the objects themselves. The time to complete a swap in Java will not depend on the size of object.

4. Bubble sort is an older sort whose performance is not competitive with the other basic sorts. Outside of this exercise, you should not use bubble sort. One variant of bubble sort that works on the first n entries in an array uses the following algorithm.

```
Set first position to 0
Set last position to  $n-2$ 
While the first position is less than the last position
    Set last swap location to first position
    Loop  $i$  from first position to last position
        If the entries at positions  $i$  and  $i+1$  are out of order
            Swap the entries in positions  $i$  and  $i+1$ 
            Set last swap to  $i$ 
    Set last position to last swap
```

Implement this version of bubble sort and add statistics to it in `SortArrayInstrumented`. Compute the minimum, maximum, and average number of comparisons done.

5. The given Shell sort works with increments of 1, 2, 4, 8, ... (written in reverse order). The performance of Shell's sort can be improved by allowing more varied overlap between the sequences. Implement two new versions of Shell's sort that use the sequence of increments

1, 4, 13, 40, 121 ($s_i = 3s_{i-1} + 1$)
and
1, 3, 7, 15, 31 ($s_i = 2s_{i-1} + 1$)

Compute the minimum, maximum, and average number of comparisons done by these two versions of Shell's sort and compare with the original.

6. Create a new sort that works in two phases. In the first phase, generate $n \log_2 n$ pairs of random positions in the array that are at least $(\log_2 n)^2$ apart. If the items in the pair of positions are out of order, switch them. In the second phase, do an insertion sort. Compute the minimum, maximum and average number of comparisons done by this sort and compare with the standard insertion sort. (The idea behind this sort is that insertion sort works much better on arrays that are nearly sorted. In the first phase we do a small amount of work to get the array closer to being in sorted order. Swapping values that are farther apart has a potentially better improvement so we don't want the positions to be too close to each other. Shell sort uses the same idea, but in a more regimented fashion with a decreasing distance between positions.
7. By counting the number of inversions in a array, you get a measure of how close the array is to being sorted. Consider every pair of values. (There are $n(n-1)/2$ pairs.) Each pair that is out of order contributes one to the number of inversions. Implement a method that counts the number of inversions in an array of `Comparable` objects.
8. While measuring the performance of a sort against randomly generated arrays is important, in real life data are often not random. Nearly sorted data are frequently encountered. Develop a method that generates random arrays that have at most k inversions. Use this method to compute the performance of the three sorts for $k = n/2$, $k = n$, and $k = 2n$.

