Lab 9: Sorting
12:00 PM, Mar 21, 2018

Contents

1 Heapsort 2
2 Quicksort 2
3 Bubble Sort 3
4 Merge Sort 3
5 Mirror Mirror on the Wall, Who is Fastest of Them All? 4
6 Just For Fun: Stable & In-Place Sorting 4

Objectives

By the end of this lab, you will be able to:

- sort numbers quickly
- run empirical tests to compare the run times of various sorting algorithms

Setup

Task: Before beginning this assignment, copy the support code to the ‘lab09’ directory inside your scalaproject:

cp /course/cs0180/src/lab09/src/* ~ /course/cs0180/workspace/scalaproject/src/lab09/src/

You will be referring to this source code throughout the lab. It contains:

ISorter, an interface containing the sort method and the implicit coerce method which enforces the requirement that only Ordered (sortable) types can be used as this interface’s type parameter.

AbsSorter, an abstract class that inherits the same requirement and is extended by the concrete sorting classes (sorters), some of which you will write.

MergeSorter, a concrete class that implements the sort method using merge sort.

BubbleSorter, a concrete class that implements the sort method using bubble sort.

CompareSorters, a class that allows you to run the sorting algorithms you implement on randomly generated datasets, so you can compare their runtimes empirically.
You will be implementing the sort method using heapsort and quicksort, in concrete classes named HeapSorter and QuickSorter, respectively.

Testing

To facilitate testing, the support code also includes an ISorterTest class, which is extended by traits that contain methods you can use to test your various implementations of ISorter. To use these test cases, you should first create an ISorterTest object by passing it an ISorter. Then, you can mix in whichever test traits you want to use. When you mix in a test trait, it will include the tests from that trait. For example, if you want to run the tests in the IntSorterTest trait on the provided MergeSorter, you would do the following:

```scala
object BuiltInQuickSorter extends App {
  Tester.run(new ISorterTest[Int](new MergeSorter) with IntSorterTest)
}
```

Note: If the syntax is still unclear, check out the tests under BubbleSorter and MergeSorter provided in the source code.

1 Heapsort

Given a set of comparable items, how can you use a heap to sort the set? The answer to this question leads to the sorting algorithm called heapsort.

Task: Write a HeapSorter class that extends the AbsSorter class. Implement heapsort by overriding the sort method!

Hint: The constructor for HeapSorter should look like this:

```scala
class HeapSorter[T <% Ordered[T]] extends AbsSorter[T]
```

Hint: You should override the sort method like this:

```scala
override def sort(toSort: Seq[T]): Seq[T]
```

Hint: Feel free to use Scala’s built-in priority queue:

```scala
import scala.collection.mutable.PriorityQueue
```

Hint: You can iterate through the input Seq with the following syntax:

```scala
for (elem <- toSort)
  // do stuff
```

Task: Create an Object that extends App in your HeapSorter file. In this Object, use the test cases provided in the IntSorterTest trait to test your HeapSorter implementation.

Task: Given that each insertion and deletion from a heap takes $O(\log n)$ time, what is the big-O runtime of heapsort?

Hint: Try figuring out the runtime of the inserting and removing from the priority queue.
2 Quicksort

Quicksort is another fast sorting algorithm. It works by first choosing an element in the given list, called a pivot. Then, it moves all items in the list with value less than the pivot before the pivot, and all items in the list with value greater than the pivot after the pivot. At this point, the pivot is in the correct spot for the final sorted list.

The algorithm then recursively repeats the process on the sub-list before the pivot and the sub-list after the pivot, sorting both. This type of algorithm is called a divide-and-conquer algorithm, because the two halves of the list are being sorted independently, and eventually are simply concatenated.

In short: one element is picked, and moved to its correct (sorted) index, then the left half and right half of the list are also sorted recursively.

Task: Write a QuickSorter class that extends the AbsSorter class. Implement quicksort by overriding the sort method!

Note: In class you looked at an in-place version of this algorithm, which is more challenging to implement. You are encouraged to instead implement a recursive solution, or any solution which you find easiest to understand.

Hint: You’ll need to keep track of the sublist you’re working with, since you’ll be calling the algorithm recursively on smaller and smaller slices of the list. How can you accomplish this? Think about using a helper function!

Hint: Think about the base case(s)!

Task: Create an Object that extends App in your QuickSorter file. In this Object, use the test cases provided in the IntSorterTest trait to test your QuickSorter implementation.

Task: What is the big-O runtime of this algorithm? What is the average-case runtime?

Hint: Consider the effect of the pivot value on the runtime. What happens if the pivot is near the middle of the list? What happens if it isn’t?

Hint: Try coming up with a recurrence relation, and see if you recognize it! If not, call over a TA to help jog your memory.

3 Bubble Sort

The bubble sort algorithm is another sorter that we have implemented for you in BubbleSorter. It works by starting at the beginning of the list, comparing the first two elements. If they are out of order, they get swapped. Then, it compares the second and third elements, again swapping these two if they are out of order. This process continues with the third and fourth elements, etc., until the last two elements of the list are compared and swapped.

That process is one single pass through the list. The algorithm functions by making such passes through the list until no swaps occur, at which point the list is sorted.

Task: Take a look at our implementation of this algorithm. Using this and the description of bubble sort, what is the big-O runtime of this sorting algorithm?
4 Merge Sort

Merge sort is another sorting algorithm we have already implemented for you. The first part of this algorithm splits the list in half, and sorts each half separately, recursively. At the end of this part, we end up with several small (base-case, can be of length 1) sorted lists which we need to merge together. Simply appending these lists to one another would not result in a sorted result!

Thus, the second part of the algorithm works as follows: we inspect a pair of adjacent sorted lists, and begin with an empty list res to store the result of their merge. We compare the list heads and remove the smaller list head, appending it to res. Then, we repeat the process of comparing the two list heads and removing the smaller one, appending it to res. This continues until either one list is empty, where we simply append the non-empty list to res, or both lists are empty.

This second step repeats with the newly merged sub-lists until we have merged the full list.

**Task:** Take a brief look at our implementation of merge sort, in MergeSorter. Using our description and our code, what is the big-O runtime of this algorithm?

**Hint:** If the merge sort description is still confusing, check out the mergesort slide from the presentation, which shows what happens in the algorithm visually!

5 Mirror Mirror on the Wall, Who is Fastest of Them All?

In the CompareSorters class, you will find a compareSorters method that takes as input a dictionary (map) of Strings and ISorters and a testSize (an integer representing the size of lists to sort), and then reports to the console the results of running each ISorter in the dictionary on a sequence of testSize randomly-generated integers. The Map that you provide should have sorter names (Strings) as keys, and sorter objects as values. For example:

```scala
val sorters = Map("Heapsort" -> new HeapSorter[Int],
                  "Quicksort" -> new QuickSorter[Int])

(new CompareSorters).compareSorters(sorters, 200000)
```

would invoke your HeapSorter and QuickSorter on 200000 randomly-generated integers.

**Task:** Create a new Scala file, containing Object SorterTest extends App. In it, you will compare the run times of the sorting algorithms you have implemented, and all of the sorting algorithms we have provided.

**Task:** Run the SorterTest with a testSize of 500, 1000, and 2000 to see that our implementation of bubble sort and merge sort matches your asymptotic (big O) analysis. That is, show that if you increase n by a factor of 4, the runtime increases by at most a factor of 4 for an O(n) algorithm, a factor of 8 for an O(n log n) algorithm, a factor of 16 for an O(n^2) algorithm, etc. Record the runtime for each sorting algorithm at each list size.

Note how long BubbleSort took! Now we want to run the different sorts on really large sizes, but we don’t want BubbleSort to take ages.

**Task:** Remove BubbleSort from the map. Run CompareSorters a few times on at least 3 very large (ex: 1000000) testSizes. Record the runtimes for each of the algorithms at each size and see that the implementations of the various sorts match your asymptotic (big O) analyses. That is, show that if you increase n by a factor of 4, the runtime increases by at most a factor of 4 for an O(n) algorithm, a factor of 8 for an O(n log n) algorithm, a factor of 16 for an O(n^2) algorithm, etc.
6 Just For Fun: Stable & In-Place Sorting

In CS, a stable sort is a sorting algorithm where if there are multiple copies of the same element (or, elements with the same sorted value) in the original list, then in the sorted list, they appear in the same order as in the original list. For example, suppose we are sorting a list of number pairs:

\{<1, 5>, <6, 3>, <1, 2>, <4, 10>, <1, 10>\}

Let’s say the criteria is that you want to sort by the first number. For example, \(<1, 5>\) should be before \(<2, 4>\); and \(<2, 5>\) and \(<2, 3>\) are considered “equal.” A stable sort would produce the following:

\{<1, 5>, <1, 2>, <1, 10>, <4, 10>, <6, 3>\}

Note that the three pairs with 1 as their first number are in the same order as they were in the original list. This is what makes it a stable sort. In contrast, this:

\{<1, 2>, <1, 5>, <1, 10>, <4, 10>, <6, 3>\}

Is still correctly sorted, but is unstable, because the three elements with “equal” ranking were reordered.

**Task:** Is your implementation of heapsort a stable sort? What about your quicksort?

**Hint:** It depends on the way you did it! Try walking through your algorithms with two “equally ranked” items, and see if it’s possible for the algorithm to swap them.

**Hint:** Priority queues in Java do not have a stable insertion method! That is, when inserting values, the queue may not keep values of equal “rank” in the order of insertion.

**Task:** If your quicksort is not a stable sort, how could you modify it to make it stable? You don’t need to implement this, just be able to describe the necessary modifications with a good level of detail.

**Hint:** How did you handle the case where you find an element in the list of equal value to the pivot?

In CS, an in-place sorting implementation is one that modifies the original data structure to be sorted, rather than returning a sorted copy. In-place sorts require no extra space, which can be an important consideration!

**Task:** Which, if any, of the sorts that you implemented today or that we provided to you are implemented in-place? What type of data structure (i.e. mutable vs immutable) is the input required to be in order to perform an in-place sort?

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [https://cs.brown.edu/courses/cs018/feedback](https://cs.brown.edu/courses/cs018/feedback)