Sergio E. Garcia Tapia
Algorithms by Sedgewick and Wayne (4th edition) [SW11]
November 10th, 2024

2.5: Applications

Exercise 1. Consider the following implementation of the `complareTo()` method for `String`. How doe the third line help with efficiency?

```
public int compareTo(String that)
{
   if (this == that) return 0; // this line
   int n = Math.min(this.length(), that.length());
   for (int i = 0; i < n; i++)
   {
      if      (this.charAt(i) < that.charAt(i)) return -1;
      else if (this.charAt(i) > that.charAt(i)) return +1;
   }
   return this.length() - that.length();
}
```

Solution. In general, the method is linear in the length of the shortest of the two strings. However, it may be that the strings are aliased, so that effectively a string is being compared to itself. The indicated line detects this condition and reduces the duration of the compare to constant time.

Exercise 2. Write a program that reads a list of words from standard input and prints all two-word compound words in the list. For example, if `after`, `thought`, and `afterthought` are in the list, then `afterthought` is a compound word.

Solution. See `com.segarciat.algs4.ch2.sec5.ex02.TwoWordCompoundWords`.

Exercise 3. Criticize the following implementation of a class intended to represent account balances. Why is `compareTo()` a flawed implementation of the `Comparable` interface?

```
public class Balance implements Comparable<Balance>
{
   // ...
   private double amount;
   public int compareTo(Balance that)
   {
      if (this.amount < that.amount - 0.005) return -1;
      if (this.amount > that.amount + 0.005) return +1;
      return 0;
   }
   // ...
}
```

Describe a way to fix this problem.

Solution. It appears that the implementation is attempting to assert that the two `Balance` instances compare equal when their `amount` is within `0.005`. For example, this would certify that `0.10` and `0.104` are the same, presumably both 10 cents. However, numbers of type `double` are known to be subject to rounding errors. Moreover, such an implementation does not define a total ordering. For example, suppose we had objects `a`, `b`, and `c` of type `Balance`, such that

(i) `a.amount = 0.097`

(ii) `b.amount = 0.10`

(iii) `c.amount = 0.103`

Assuming no rounding errors, we would have `a.compareTo(b) == 0` and `b.compareTo(c) == 0`, but `a.compareTo(c) == -1`, so that we don't have transitivity.

To fix this, we can choose a different representation for the amount. We can use two instance variables: one for the amounts smaller than 1 (for example, the number of cents if we are speaking of dollars), and another for the amounts that are 1 or larger (like dollars bills). Then the `compareTo()` method can exactly compare these quantities.

Exercise 4. Implement a method `String[] dedup(String[] a)` that returns the objects in `a[]` in sorted order, with duplicates removed.

Solution. See `com.segarciat.algs4.ch2.sec5.ex04.DeduplicatedStrings`.

Exercise 5. Explain why selection sort is not stable.

Solution. [SW11] describes a sorting method as *stable* if "it preserves the relative order of equal keys in the array". The reason this is so is because at any point, the "next minimum" that the algorithm searches for could be anywhere in the array. If the two elements equal elements are adjacent to one another, and the "next minimum" is somewhere to the right of them, then they could end up not in relative order.

Considered, for example, the following array:

```
[2] 2 3 4 1
```

, and exchanges the *first* 2 to get:

```
1 [2] 3 4 2
```

Notice that the relative order of the 2's changed. On the next iteration, the `2` in the second place (which has not been subject to a swap) stays in place, because no other key in the array is smaller than it:

```
1 2 [3] 4 2
```

Next, the next smallest is the `2` at the end, which is swapped with the `3` to get:

```
1 2 2 [4] 3
```

The elements to the left of the scan pointers are not moved anymore, so the 2's do not end up in the same relative order they started with.

Exercise 6. Implement a recursive version of `select()`.

Solution. See `com.segarciat.algs4.ch2.sec5.ex06.RecursiveSelect`.

Exercise 7. About how many compares are required, on average, to find the smallest of n items using `select()`?

Solution. By `Proposition U`, the average number of compares to find the kth smallest is $\sim 2n + 2 \cdot k \ln(n/k) + 2(n-k) \cdot \ln(n/(n-k))$. As $k \to 0$, this quantity approaches $2n$, suggesting the average.

Exercise 8. Write a program `Frequency` that reads strings from standard input and prints the number of times each string occurs, in descending order of frequency.

Solution. See `com.segarciat.algs4.ch2.sec5.ex08.Frequency`. I have implemented this by using a minimum-oriented priority queue with `String` objects read from standard input, and a max-oriented priority queue with `StringCountNode` objects, a data type I defined that simply holds a `String` read from standard input and its frequency.

Exercise 9. Develop a data type that allows you to write a client that can sort a file such as the one shown on below:

```
# input (DJIA volumes for each day)
 1-Oct-28  3500000
 2-Oct-28  3850000
 3-Oct-28  4060000
 4-Oct-28  4330000
 5-Oct-28  4360000
...
30-Dec-99  554680000
31-Dec-99  374049984
 3-Jan-00  931800000
 4-Jan-00  1009000000
 5-Jan-00  1085500032

# output
  19-Aug-40 130000
  26-AUg-40 160000
  24-Jul-40 200000
  10-Aug-42 210000
  23-Jun-42 210000
  ...
  23-Jul-02 2441019904
  17-Jul-02 2566500096
  15-Jul-02 2574799872
  19-Jul-02 2654099968
  24-Jul-02 2775555936
```

Solution. See `com.segarciat.algs4.ch2.sec5.ex09.DJIAVolume`.

Exercise 10. Create a data type `Version` that represents a software version number, such as `115.1.1`, `115.10.1`, `115.10.2`. Implement the `Comparable` interface so that `115.1.1` is less than `115.10.1`, and so forth.

References

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th ed. Addison-Wesley, 2011. ISBN: 9780321573513.