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 $k$th 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.

**Exercise 12.** *Scheduling.* Write a program `SPT.java` that reads job names and processing times from standard input and prints a schedule that minimizes average completion time using the shortest processing time first rule, as described on page 349.

**Solution.** See `com.segarciat.algs4.ch2.sec5.ex12`.

**Exercise 13.** *Load balancing.* Write a program `LPT.java` that takes an integer $m$ as a command-line argument, reads job names and processing times from standard input and prints a schedule assigning the jobs to $m$ processors that approximately minimizes the time when the last job completes using the longest processing time first rule, as described on page 349.

**Solution.** See `com.segarciat.algs4.ch2.sec5.ex13`.

**Exercise 14.** *Sort by reverse domain.* Write a data type `Domain` that represents domain names, including an appropriate `compareTo()` where the natural order is in order of the *reverse* domain name. For example, the reverse domain name of `cs.princeton.edu` is `edu.princeton.cs`. This is useful for web log analysis. *Hint*: Use `s.split("\\.")` to split the string `s` into tokens, delimited by dots. Write a client that reads domain names from standard input and prints the reverse domains in sorted order.

**Solution.** See `com.segarciat.algs4.ch2.sec5.ex14`.

**Exercise 15.** *Spam campaign.* To initiate an illegal spam campaign, you have a list of email addresses from various domains (the part of the email address that follows the `@` symbol). To better forge the return addresses, you want to send the email from another user at the same domain. For example, you might want to forge an email from `wayne@princeton.edu` to `rs@princeton.edu`. How would you process the email list to make this an efficient task?

**Solution.** I would use a priority queue to sort the list by domain. Then I would begin by taking address from the queue, which is the first forge senders. I would take other address off and as long as the the it is in the same domain as the current sender, I would set its sender as the first sender. I would keep track of the last sender removed at each step. As soon as the domain of the current domain differs from that of the last sender removed, I would arrange for the first sender to receive an email from the last sender removed. Then I would update the first sender to be the email just removed, the one from the new domain.

**Exercise 16.** *Unbiased election.* In order to thwart bias against candidates whose names appear toward the end of the alphabet, California sorted the candidates appearing on its 2003 gubernational ballot by using the following order of characters:

```
R W Q O J M V A H B S G Z X N T C I E K U P D Y F L
```

Create a data type where this is the natural order and write a client `California` with a single static method `main()` that sorts strings according to this ordering. Assume that each string is composed solely of uppercase letters.

**Exercise 17.** *Check stability.* Extend your `check()`method from **Exercise 2.1.16** to call `sort()` for a given array and return `true` if `sort()` sorts the array *in a stable manner*, `false` otherwise. Do not assume that `sort()` is restricted to move data only with `exchange()`.

**Solution.** See `com.segarciat.algs4.ch2.sec5.ex17.Stability`.

**Exercise 18.** *Force stability.* Write a wrapper method that makes any sort stable by creating a new key type that allows you to append each key's index to the key, call `sort()`, then restore the original key after the sort.

**Solution.** See `com.segarciat.algs4.ch2.sec5.ex18.ForceStability`.

**Exercise 19.** *Kendall tau distance.* Write a program `KendallTau.java` that computes the Kendall tau distance between two permutations in linearithmic time.

**Solution.** From page 345, we know the following:

- A permutation (rankings) is an array of `n` integers where each of the integers between `0` and `n-1` appears exactly once.
- The Kendall tau distance between two permutations is the number of pairs that are in different order in the two rankings.
- The number of inversions in an array is the Kendall tau distance between the array and the identity permutation.

Given permutations `a` and `b`, we can have the order of the elements in `b` define the "sort order". For example, if we have arrays:

```
a: 0 3 1 6 2 5 4
b: 1 0 3 6 4 2 5
```

Then, according to array `b`, the sort order is:

```
s: [ 1, 0, 5, 2, 4, 6, 3 ]
```

Notice that if we consider `b` as a mapping from the integer indices to its array values, then `s` is the inverse.

Now we can use mergesort to sort `a` according to the order define by `b` (according to `s`), counting inversions along the way.

# References

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