Sergio E. Garcia Tapia
Algorithms by Sedgewick and Wayne (4th edition) [SW11]
September 14th, 2024

1.2: Data Abstraction

Exercise 1. Write a `Point2D` client that takes an integer value n from the command line, generates n random points in the unit square, and computes the distance separating the *closest pair* of points.

Solution. See the `com.segarciat.algs4.ch1.sec2.ex01.ClosestPointPair` class. The `Point2D` objects can simply by stored in an array of size n. We can use `StdRandom.uniformDouble(` to generate random x and y coordinates for each point, and then leverage the `distanceTo()` method available in the `Point2D` API. I employed a nested `for` loop to compute the closest pair.

Exercise 2. Write an `Interval1D` client that takes an `int` value n as command-line argument, reads n intervals (each defined by a pair of `double` values) from standard input, and prints all pairs that intersect.

Solution. See the `com.segarciat.algs4.ch1.sec2.ex02.IntervalIntersection` class. It's much the same as in Exercise 1, but instead of using `StdRandom.uniformDouble()` to generate the coordinates, I use `StdIn.readDouble()` to read coordinate from standard input. Also, instead of the `distanceTo()` method from the `Point2D` API, I leveraged the `intersects()` method from the `Interval1D` API.

Exercise 3. Write an `Interval2D` client that takes command-line arguments `n`, `min`, and `max` and generates `n` random 2D intervals whose width and height are uniformly distributed between `min` and `max` in the unit square. Draw them on `StdDraw` and print the number of pairs of intervals that intersect and the number of intervals that are contained in one another.

Solution. See the `com.segarciat.algs4.ch1.sec2.ex03.IntersectingRectangles` class.

One important consideration is that since the widths and heights are generated uniformly between `min` and `max`, we must ensure the bottom left corner of each point isn't so large that it would exceed the dimensions of the unit square. That is, given `width` and `height`, each of the x and y coordinates of the bottom left vertex of all rectangles must not exceed `1 - width` and `1 - height`, respectively.

Another important consideration is that, to check if rectangle A contains rectangle B, we must check that the bottom-left and top-right vertices of rectangle B are contained in rectangle A.Since the `Interval2D` API does not expose methods for obtaining these quantities, it's necessary to save them while doing the computations to necessary to create the rectangles.

Exercise 4. What does the following code fragment print?

```
String string1 = "hello";
String string2 = string1;
```

```
string1 = "word";
StdOut.println(string1);
StdOut.println(string2);
```

Solution. When `string1` is assigned to`string2`, the `string2` variable receives a copy of the reference to the current value of `string1`. When `string1` is assigned the `String` with value `"word"`, the reference in `string2` is unchanged. Thus the output is:

```
world
hello
```

Exercise 5. What does the following code fragment print?

```
String s = "Hello World";
s.toUpperCase();
s.substring(6, 11);
StdOut.println(s);
```

Solution. `String` objects are immutable, so the calls to the `toUpperCase()` and the `substring()` methods do not change the object that `s` references; they return new `String` objects. In this case, those objects are not stored, so they are immediately available for garbage collection. Thus the output is simply:

```
Hello World
```

Exercise 6. A string `s` is a circular shift (or *circular rotation*) of a string `t` if it matches when the characters are circularly shifted by any number of positions; e.g., `ACTGACG` is a circular shift of `TGACGAC`, and viceversa. Detecting this condition is important in the study of genomic sequences. Write a program that checks whether two given strings `s` and `t` are circular shifts of one another. *Hint*: The solution is a one liner with `indexOf()`, `length()`, and string concatenation.

Solution. See the `com.segarciat.algs4.ch1.sec2.ex06.CircularShift` class. A prerequisite for `s` and `t` to be circular shifts of one another is that they have the same length. After establishing this, we can detect the condition by concatenating `s` with itself to create a new string, and then check whether `t` is a substring of this new string.

Exercise 7. What does the following recursive function return?

```
public static String mystery(String s)
{
   int n = s.length();
   if (n <= 1) return s;
   String a = s.substring(0, n/2);
   String b = s.substring(n/2, n);
   return mystery(b) + mystery(a);
}
```

Solution. It reverses the string `s`.

Exercise 8. Suppose `a[]` and `b[]` are each integer arrays consisting of millions of integers. What does the following code do? Is it reasonably efficient?

```
int[] t = a; a = b; b = t;
```

Solution. It swaps arrays `a` and `b`. It's very efficient because the array values are not copied.

Exercise 9. Instrument `BinarySearch` (page 47) to use a `Counter` to count the total number of keys examined during all searches and then print the total after all searches are complete. *Hint*: Create a `Counter` in `main()` and pass it as an argument to `indexOf()`.

Solution. See the See the `com.segarciat.algs4.ch1.sec2.ex09.BinarySearchCounter` class.

Exercise 10. Develop a class `VisualCounter` that allows both increment and decrement operations. Take two arguments `n` and `max` in the constructor, where `n` specifies the maximum number of operations and `max` specifies the maximum absolute value of the counter. As a side effect, create a plot showing the value of the counter each time its tally changes.

Solution. See the `com.segarciat.algs4.ch1.sec2.ex10.VisualCounter` class.

Exercise 11. Develop an implementation `SmartDate` of our `Date` API that raises an exception if the date is not legal.

Solution. See the `com.segarciat.algs4.ch1.sec2.ex11.SmartDate` class.

Exercise 12. Add a method `dayOfTheWeek()` to `SmartDate` that returns a `String` value `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday`, or `Sunday`, giving the appropriate day of the week for the date. You may assume that the date is in the 21st century.

Solution. See the `com.segarciat.algs4.ch1.sec2.ex12.SmartDate` class.

Exercise 13. Using our implementation of `Date` as a model (page 91), develop an implementation of `Transaction`.

Solution. See the `com.segarciat.algs4.ch1.sec2.ex13.Transaction` class.

Exercise 14. Using our implementation of `equals()` in `Date` as a model (page 103), develop an implementation of `equals()` for `Transaction`.

Solution. See the `com.segarciat.algs4.ch1.sec2.ex14.Transaction` class.

Exercise 16. *Rational Numbers.* Implement an immutable data type `Rational` for rational numbers that supports addition, subtraction, multiplication, and division.

```
public class Rational
         Rational(int numerator, int denominator)
Rational plus(Rational that)      /* sum of this number and that */
Rational minus(Rational that)     /* difference of this number and that */
Rational times(Rational that)     /* product of this number and that */
Rational dividedBy(Rational that) /* quotient of this number and that */
boolean  equals(Object that)      /* is this number equal to that? */
String   toString()               /* string representation */
```

You do not have to worry about testing for overflow (see Exercise 1.2.17), but use as instance variables two `long` values that represent the numerator and denominator to limit the possibility of overflow. Use Euclid's algorithm (see page 4) to ensure that the numerator and denominator never have any common factors. Include a test client that exercises all of your methods.

Solution. See the `com.segarciat.algs4.ch1.sec2.ex16.Rational` class. I used Euclid's algorithm from the constructor to ensure the denominator is in lowest terms. I declared the instance fields for the numerator and denominators as `final` to enforce immutability. I also decided that if the fraction is negative (that is, if either the numerator or denominator is negative), then I would account for this by always storing the numerator's sign to be the same as that of the fraction, while keeping the denominator as positive. That is, given $p, q \in \mathbb{Z}$, where $q \neq 0$, each fraction in `Rational` has the form

$$\frac{-|p|}{|q|}$$

Exercise 17. *Robust implementation of rational numbers.* Use assertions to develop an implementation of `Rational` (see Exercise 1.2.16) that is immune to overflow.

Solution. See the `com.segarciat.algs4.ch1.sec2.ex17.Rational` class. I did not add new tests to verify the overflow behavior, but I did ensure that the old tests still passed. I began by defining `minus()` in terms of the `sum()` operation, and `dividedBy()` in terms of the `times()` operation. This way, I could localize overflow detection to the `sum()` and `times()` methods, and benefit in all of the methods.

To detect overflow, I employed techniques that I learned in [BO16]. Moreover, I disallowed `Long.MIN` as a numerator or denominator because my implementation uses `Euclid.gcd()` to compute the greatest common denominator in order to express a fraction in lowest terms. The issue is that `Euclid.gcd()` requires non-negative arguments, and that `Math.abs(Long.MIN)` is `Long.MIN` (it remains negative).

I also did some other work to reduce the likelihood of overflow. For example, as in traditional algebra, I wrote code to cross-reduce before multiplying the fraction. Similarly, when computing the sum of two fractions, I wrote code to multiply the "other" fraction only by the "missing factor" necessary to obtain a common denominator.

These modifications make the algorithm implementation of each method more complex, and even slower, but I reckon it is more robust and less error-prone.

Exercise 18. *Variance for accumulator.* Validate the following code, which adds the methods `var()` and `stddev()` to `Accumulator`, computes both the sample mean, sample variance, and sample standard deviation of the numbers presented as arguments to `addDataValue()`:

```java
public class Accumulator
{
   private double mu = 0.0;
   private double sum = 0.0;
   private int n = 0;

   public void addDataValue(double value)
   {
      n++;
      double delta = value - mu;
      mu += delta / n;
      sum += (double) (n-1) / n * delta * delta;
   }

   public double mean()
   { return mu; }

   public double var()
   {
      if (n <= 1) return Double.NaN;
      return sum / (n - 1);
   }

   public double stddev()
   { return Math.sqrt(this.var()); }
}
```

Solution. When an instance of `Accumulator` is created, we initialize `mu` and `sum` to `0.0`. For each $n \in \mathbb{N}$, let μ_n correspond to `mu` and let t_n correspond to `sum`. Let x_n correspond to `value` passed to the `n`th invocation of `addDataValue()`. Then

$$\begin{aligned}
\mu_0 &= 0, \\
t_0 &= 0, \\
\mu_n &= \mu_{n-1} + \frac{x_n - \mu_{n-1}}{n}, \\
t_n &= t_{n-1} + \frac{n-1}{n} \cdot (x_n - \mu_{n-1})^2
\end{aligned}$$

First we verify that μ_n is indeed giving the mean of the n values seen so far by induction. Let $n = 1$. Then

$$\mu_1 = \mu_0 + \frac{x_1 - 0}{1} = x_1$$

Hence the base case holds. Suppose we proceed by induction, that for $n \in \mathbb{N}$ it is true that μ_n is the mean. That is, our inductive hypothesis is that

$$\mu_n = \frac{1}{n} \sum_{k=1}^{n} x_k$$

Then using the recurrence relation that defines μ_{n+1}, we have

$$\begin{aligned}
\mu_{n+1} &= \mu_n + \frac{x_{n+1} - \mu_n}{n+1} \\
&= \frac{(n+1)\mu_n + x_{n+1} - \mu_n}{n+1} \\
&= \frac{n\mu_n + x_{n+1}}{n+1} \\
&= \frac{n \cdot \frac{1}{n}\sum_{k=1}^{n} x_k + x_{n+1}}{n+1} \\
&= \frac{1}{n+1}\sum_{k=1}^{n+1} x_k
\end{aligned}$$

as we set out to show. By the mathematical induction, we conclude that μ_n is indeed the mean of the first n terms.

Similarly, we can prove by induction that $\frac{t_n}{n-1}$ is indeed the variance of the first n terms, for $n \in \mathbb{N}$ and $n \geq 2$. When $n = 1$, we have $t_1 = 0$ because $n - 1 = 1 - 1 = 0$. When $n = 2$, we have:

$$\begin{aligned}
\mu_1 &= x_1 \\
\mu_2 &= \frac{1}{2}(x_1 + x_2)
\end{aligned}$$

Then

$$\begin{aligned}
(x_1 - \mu_2)^2 + (x_2 - \mu_2)^2 &= \left(\frac{1}{2}x_1 - \frac{1}{2}x_2\right)^2 + \left(\frac{1}{2}x_2 - \frac{1}{2}x_1\right)^2 \\
&= 2 \cdot \left(\frac{x_2 - x_1}{2}\right)^2 \\
&= \frac{2-1}{2}\,(x_2 - x_1) \\
&= \frac{2-1}{2}(x_2 - \mu_1)^2 \\
&= t_2
\end{aligned}$$

This is our base case. Now suppose that $n \geq 2$. Then our inductive hypothesis is that

$$t_n = \sum_{k=1}^{n}(x_k - \mu_n)^2$$

We can simplify this to:

$$
\begin{aligned}
\sum_{k=1}^{n}(x_k-\mu_n)^2 &= \sum_{k=1}^{n}[x_k^2-2\mu_n x_k+\mu_n^2] \\
&= \sum_{k=1}^{n} x_k^2 - 2\cdot\mu_n\sum_{k=1}^{n} x_k + \sum_{k=1}^{n}\mu_n^2 \\
&= \sum_{k=1}^{n} x_k^2 - 2\cdot\mu_n\cdot n\cdot\mu_n + n\cdot\mu_n^2 \\
&= \sum_{k=1}^{n} x_k^2 - n\mu_n^2
\end{aligned}
$$

where we've used the fact that $\sum_{k=1}^{n} x_k = n\cdot\mu_n$. Moreover, note that

$$
\begin{aligned}
(n+1)\mu_{n+1} &= \sum_{k=1}^{n+1} x_k \\
&= x_{n+1} + \sum_{k=1}^{n} x_k \\
&= x_{n+1} + n\mu_n
\end{aligned}
$$

Now:

$$
\begin{aligned}
t_{n+1} - \sum_{k=1}^{n+1}(x_k-\mu_{n+1})^2 &= t_{n+1} - \left[\sum_{k=1}^{n+1} x_k^2 - (n+1)\mu_{n+1}^2\right] \\
&= \left[t_n + \frac{n}{n+1}(x_{n+1}-\mu_n)^2\right] - \left[\sum_{k=1}^{n+1} x_k^2 - (n+1)\mu_{n+1}^2\right] \\
&= \left[\sum_{k=1}^{n} x_k^2 - n\mu_n^2 + \frac{n}{n+1}(x_{n+1}-\mu_n)^2\right] - \left[\sum_{k=1}^{n+1} x_k^2 - (n+1)\mu_{n+1}^2\right] \\
&= \frac{n}{n+1}(x_{n+1}-\mu_n)^2 + (n+1)\mu_{n+1}^2 - n\mu_n^2 - x_{n+1}^2 \\
&= \frac{nx_{n+1}^2 - 2nx_{n+1}\mu_n + n\mu_n^2 + (n+1)^2\mu_{n+1}^2 - n(n+1)\mu_n^2 - (n+1)x_{n+1}^2}{n+1} \\
&= \frac{-(n^2\mu_n^2 + 2nx_{n+1}\mu_n + x_{n+1}^2) + (n+1)^2\mu_{n+1}^2}{n+1} \\
&= \frac{-(n\mu_n + x_{n+1})^2 + (n+1)^2\mu_{n+1}^2}{n+1} \\
&= \frac{-[(n+1)\mu_{n+1}]^2 + [(n+1)\mu_{n+1}]^2}{n+1} \\
&= 0
\end{aligned}
$$

References

[BO16] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 3rd ed. Pearson, 2016. ISBN: 9780134092669.

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