Sergio E. Garcia Tapia
Algorithms by Sedgewick and Wayne (4th edition) [SW11]
September 8th, 2024

1.1: Basic Programming Model

Exercise 1. Give the value of each of the following expressions:

(a) `( 0 + 15) / 2`

(b) `2.0e-6 * 100000000.1`

(c) `true && false || true && true`

Solution.

(a) `7` because integer division uses truncation.

(b) `2.000000002E-6`

(c) `true`

Exercise 2. Give the type and value of each of the following expressions:

(a) `(1 + 2.236) / 2`

(b) `1 + 2 + 3 + 4.0`

(c) `4.1 >= 4`

(d) `1 + 2 + "3"`

Solution.

(a) `double` with value `1.6.18`

(b) `double` with value `10.0`

(c) `boolean` with value `true`

(d) `String` with value `"33"`

Exercise 3. Write a program that takes three integer command-line arguments and prints `equal` if all three are equal, and `not equal` otherwise.

Solution. See `com.segarciat.algs4.ch1.sec.ex13.Compare3Integers`. The command-line argument are `String` objects, so they can be converted to integers with `Integer.parseInt()`. Assuming the three integers are `a`, `b`, and `c`, we now just verify the value of the boolean expression `a == b && b == c`.

Exercise 4. What (if anything) is wrong with each of the following statements?

(a) `if (a > b) then c = 0;`

(b) `if a > b { c = 0; }`

(c) `if (a > b) c = 0;`

(d) `if (a > b) c = 0 else b = 0;`

Solution.

(a) `then` is not a valid Java keyword. If we remove it, the code snippet will be valid.

(b) We need parentheses around the boolean condition of the `if` statement, in this case, around `a > b`. If we add this, the snippet will be valid.

(c) The snippet is valid.

(d) We need a semicolon to terminate the assignment statement `c = 0`. If we add this, the snippet will be valid.

Exercise 5. Write a code snippet that prints `true` if the `double` variables `x` and `y` are both strictly between `0` and `1` and `false` otherwise.

Solution.

```
System.out.println(x > 0 && x < 1 && y > 0 && y < 1);
```

Exercise 6. What does the following program print?

```
int f = 0;
int g = 1;
for (int i = 0; i <= 15; i++)
{
   StdOut.println(f);
   f = f + g;
   g = f - g;
}
```

Solution. Note we are given:

$$\begin{aligned} f_0 &= 0, \\ g_0 &= 1, \\ f_n &= f_{n-1} + g_{n-1}, \quad n \geq 1, \\ g_n &= f_n - g_{n-1}, \quad n \geq 1. \end{aligned}$$

Notice the recurrence of g_n follows because the value just computed in the current iteration of the `for` loop is used to compute the new value for `g`. Note that $f_0 = 0$, $f_1 = f_0 + g_0 =$

$0 + 1 = 1$, and

$$\begin{aligned} f_{n+1} &= f_n + g_n \\ &= (f_{n-1} + g_{n-1}) + g_n \\ &= f_{n-1} + (g_{n-1} + g_n) \\ &= f_{n-1} + f_n \end{aligned}$$

Altogether, we have:

$$\begin{aligned} f_0 &= 0, \\ f_1 &= 1, \\ f_{n+1} &= f_{n-1} + f_n, \quad n \geq 1. \end{aligned}$$

Hence $n \mapsto f_n$ is the Fibonacci sequence. The program will print the first 15 Fibonacci numbers:

```
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
```

Exercise 7. Give the value printed by each of the following code fragments.

(a)

```java
double t = 9.0;
while (Math.abs(t - 9.0/t) > 0.001)
   t = (9.0/t + t) / 2.0;
StdOut.printf("%.5f\n", t);
```

```java
int sum = 0;
for (int i = 1; i < 1000; i++)
   for (int j = 0; j < i; j++)
      sum++;
StdOut.println(sum);
```

(b)

```java
int sum = 0;
for (int i = 1; i < 1000; i *= 2)
```

```
    for (int j = 0; j < 1000; j++)
        sum++;
StdOut.println(sum);
```

Solution.

(a) The iterations are computed as:

$$t_0 = 9.0 \quad ; \quad t_0 - \frac{9.0}{t_0} = 1 \quad ; \quad \to \quad t_1 = \frac{9.0/t_0 + t_0}{2.0} = 5.0$$
$$t_1 = 5.0 \quad ; \quad t_1 - \frac{9.0}{t_1} = 3.2 \quad ; \quad \to \quad t_2 = \frac{9.0/t_1 + t_1}{2.0} = 3.4$$
$$t_2 = 3.4 \quad ; \quad t_2 - \frac{9.0}{t_2} \approx 0.75294 \quad ; \quad \to \quad t_3 = \frac{9.0/t_2 + t_2}{2.0} = 3.023529411764706$$
$$t_3 \approx 3.02352 \quad ; \quad t_3 - \frac{9.0}{t_3} \approx 0.04687 \quad ; \quad \to \quad t_4 \approx \frac{9.0/t_3 + t_3}{2.0} = 3.00009155413138$$
$$t_4 = 3.00009155413138 \quad ; \quad t_4 - \frac{9.0}{t_4} = 0.00018310546879263256$$

The iteration ends once t_4 has been computed because it's below the threshold of `0.004` that controls the `while` loop. Since the format specifier requires 5 places after the decimal, the output will be:

```
3.00009
```

(b) The outer `i` loop runs 999 times. The inner `j` loop runs `i` times, and each time, it increment `sum` by 1. The value of `sum` is given by:

$$\sum_{i=1}^{999} \sum_{j=0}^{i-1} = \sum_{i=1}^{999} i = \frac{999 \cdot (999+1)}{2} = 499500$$

Therefore the out will be:

```
499500
```

(c) In this case, we start with 1 and double `i` each time. When `i` reaches $2^{10} = 1024$, the `i` loops will end. Hence, the loop will run for $i = 2^0, i = 2^1, \ldots, i = 2^9$. Since the `j` loops increases `sum` a total of `i` times, we find that `sum` will now be:

$$\begin{aligned} \sum_k \sum_{i=1}^{1000} [i = 2^k] \sum_{j=0}^{i-1} &= \sum_k \sum_{1 \le i \le 1000} [i = 2^k] \cdot i \\ &= \sum_{1 \le 2^k \le 1000} 2^k \\ &= \sum_{0 \le k \le 9} 2^k \\ &= 2^{10} - 1 \\ &= 1023 \end{aligned}$$

Therefore the output will be:

```
1023
```

Exercise 8. What do each of the following print?

(a) `System.out.println('b');`

(b) `System.out.println('b' + 'c');`

(c) `System.out.println((char) ('a' + 4));`

Solution.

(a) Java will display the `char` as the corresponding symbol:

```
b
```

(b) When Java adds two `char` values, it will promote the result to an `int`. In Java, `char` values are 16-bit Unicode characters. Since `'b'` has decimal value 98 in Unicode, and `'c'` has decimal value `99`, the result is:

```
197
```

(c) The `char` value `'a'` has decimal value `97`, so when it is added to `4`, it becomes integer value `101`. The effect of `(char)` is to cast the result back to a `char`. The integer `101` fits into a `char`, and it corresponds to `'e'`:

```
e
```

Exercise 9. Write a code fragment that puts the binary representation of a positive integer n into a `String s`.

Solution. If we divide `n` by 2, then the remainder of the division is the least significant bit in the binary representation of `n`. If we were to divide by the resulting quotient by `2`, then the remainder of that division is the next most significant bit. Continuing this way, the value of `n` falls to `0` as we continue to divide by 2. The solution is actually given in [SW11]:

```
String s = "";
for (int k = n; n > 0; n /= 2)
   s = (k % 2) + s;
```

We could extend this to handle `0` by changing it to a `do {/*...*/} while(/*...*/);`.loop.

Exercise 10. What is wrong with the following code fragment?

```
int[] a;
for (int i = 0; i < 10; i++)
   a[i] = i * i;
```

Solution. It fails to use `new` to allocate memory for the array before using it in the `for` loop.

Exercise 11. Write a code fragment that prints the contents of a two-dimensional boolean array, using `*` to represent `true` and a space to represent `false`. Include row and column numbers.

Solution.

```
for (int i = 0; i < m; i++) {
   for(int j = 0; j < n; j++)
      System.out.printf("(%d,%d): %s ", i, j, (a[i][j]) ? "*" : " ");

   System.out.println();
}
```

Exercise 12. What does the following code fragment print?

```
int[] a = new int[10];
for (int i = 0; i < 10; i++)
   a[i] = 9 - i;
for (int i = 0; i < 10; i++)
   a[i] = a[a[i]];
for (int i = 0; i < 10; i++)
   System.out.println(a[i]);
```

Solution. The first lop sets `a` to {`9, 8, 7, 6, 5, 4, 3, 2, 1, 0`}. The second loop {`0, 1, 2, 3, 4, 4, 3, 2, 1, 0`}. Thus the output is:

```
0
1
2
3
4
4
3
2
1
0
```

Exercise 13. Write a code fragment to print the *transposition* (rows and columns changed) of a two-dimensional array with m rows and n columns.

Solution.

```
for (int i = 0; i < m; i++) {
   for (int j = 0; j < n; j++)
      System.out.printf("%d ", a[j][i]);
   System.out.println();
}
```

Exercise 14. Write a static method `lg()` that takes an `int` value `n` as argument and returns the largest `int` not larger than the base-2 logarithm of `n`. Do *not* use `Math`.

Solution. See the class `com.segarciat.algs4.ch1.sec.ex14.LgFloor`. Note that the logarithm is only defined for positive numbers, so we begin by throwing an exception if $n \leq 0$. Assuming now that $n > 0$, suppose that 2^m is the largest power of 2 in its base-2 (binary) representation. Since $\log_2$ is monotonic, we know that:

$$\begin{aligned} \log_2(2^m) &\leq \log_2(n) \\ m &\leq \log_2(n) \end{aligned}$$

Put another way, $m = \lfloor \log_2(n) \rfloor$, the *floor* of the base-2 logarithm; this is the number requested in this question. For example, we can make a table listing some sample values:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\lfloor \log_2(n) \rfloor$	0	1	1	2	2	2	2	3	3	3	3	3	3	3	3	4

Note that if 2^m is the largest power in the binary representation of n, then n can be represented by $m + 1$ bits. To determine the number of bits in the binary representation of n we can repeatedly divide by 2 (or perform logical right arithmetic shifts) until the quantity becomes 0. Subtracting 1 from this yields m. Equivalently, we can continue as long as the result of dividing by 2 is still greater than 1, and skip the subtraction.

Exercise 15. Write a static method `histogram()` that takes an array `a[]` of `int` values and an integer `m` as arguments and returns an array `m` whose `i`th entry is the number of times the integer `i` appeared in the argument array. If the values in `a[]` are all between `0` and `m-1`, the sum of the values in the returned array should equal to `a.length`.

Solution. See `com.segarciat.algs4.ch1.sec.ex19.Histogram`.

Exercise 16. Give the value of `exR1(6)`:

```java
public static String exR1(int n)
{
  if (n <= 0) return "";
  return exR1(n-3) + n + exR1(n-2) + n;
}
```

Solution. The first call is as follows:

```
exR1(6) -> exR1(3) + 6 + exR1(4) + 6
```

Now we look at `exR1(3)`:

```
exR1(3) -> exR1(0) + 3 + exR1(1) + 3
```

By the base case, `exR1(0)` is `""`. Meanwhile, we keep going for `exR1(1)`:

```
exR1(1) -> exR1(-2) + 1 + exR1(-1) + 1
```

Since `exR1(-2)` and `exR1(-1)` evaluate to `""` due to the base case, we get `exR1(1)` is `"11"`. Now `exR1(3)` is `"3113"`. Next we need `exR1(4)`:

```
exR1(4) -> exR1(1) + 4 + exR1(2) + 4
```

We already know that `exR1(1)` is `"11"`. For `exR1(2)`:

```
exR1(2) -> exR1(-1) + 2 + exR1(0) + 2
```

Hence `exR1(2)` is `"22"`. Altogether, we find that `exR1(4)` is `"114224"`. Finally, the value of `exR1(6)` is:

```
311361142246
```

Exercise 17. Criticize the following recursive function:

```
public static String exR2(int n)
{
   String s = exR2(n-3) + n + n + exR2(n-2) + n;
   if (n <= 0) return "";
   return s;
}
```

Solution. Because the base case comes after the recursive step, a program that invokes this function will crash with `StackOverflowError`.

Exercise 18. Consider the following recursive function:

```
public static int mystery(int a, int b)
{
   if (b == 0)   return 0;
   if (b % 2 == 0) return mystery(a+a, b/2);
   return mystery(a+a, b/2) + a;
}
```

What are the values of `mystery(2, 25)` and `mystery(3, 11)`? Given positive integers `a` and `b`, describe what `mystery(a, b)` computes. Answer the same question, but replace the three `+` operators with `*` and replace `return 0` with `return 1`.

Solution. Begin with `mystery(2, 25)`:

```
mystery(2, 25) -> 2 + mystery(4, 12):
mystery(4, 12) -> mystery(8, 6):
mystery(8, 6 ) -> mystery(16, 3):
mystery(16, 3) -> 16 + mystery(32, 1):
mystery(32, 1) -> 32 + mystery(64, 0):
mystery(64, 0) -> 0
```

Tracing back the calls, the result is $2 + 16 + 32 + 0 = 50 = 2 \cdot 25$. Similarly:

```
mystery(3, 11) -> 3 + mystery(6, 5):
mystery(6, 5 ) -> 6 + mystery(12, 2):
mystery(12, 2) -> mystery(24, 1):
mystery(24, 1) -> 24 + mystery(48, 0):
```

```
mystery(48, 0) -> 0
```

Tracing back the calls, the result is $3 + 6 + 24 + 0 = 33 = 3 \cdot 11$. It appears that the `mystery(a, b)` computes the product $a \cdot b$. In essence, we are using the binary representation of `b` decide which weights of the multiples of `a` we should add.

Next, we replace `+` with `*` and `return 0` with `return 1`:

```
mystery(2, 25) -> 2 * mystery(4, 12):
mystery(4, 12) -> mystery(16, 6):
mystery(16, 6) -> mystery(256, 3):
mystery(256, 3)-> 256 * mystery(65536, 1):
mystery(65536, 1) -> 65536 * mystery(4294967296, 0):
mystery(4294967296, 0) -> 1
```

The result is $2 \cdot 256 \cdot 65536 \cdot 1 = 33554432 = 2^{25}$. Meanwhile:

```
mystery(3, 11) -> 3 * mystery(9, 5):
mystery(9, 5 ) -> 9 * mystery(81, 2):
mystery(81, 2) -> mystery(6561, 1):
mystery(6561, 1) -> 6561 * mystery(43046721, 0):
mystery(43046721, 0) -> 1
```

The result is $3 \cdot 9 \cdot 6561 \cdot 1 = 177147$, which is 3^{11}. In this case, `mystery(a, b)` appears to be computing a^b (meaning a to the power of b).

Exercise 19. Run the following program on your computer (see Section 1.1 page 57 for the snippet). What is the largest value of `n` for which this program takes less than 1 hour to compute the value of `fibonacci(n)`? Develop a better implementation of `fibonacci(n)` that saves computed values in an array.

Solution. See the class `com.segarciat.algs4.ch1.sec.ex19.Fibonacci`. On my computer, when $n = 58$, it takes just under a hour, and during $n = 59$, it takes over an hour. See my `fibonacciFaster` method, which completes through $n = 90$ in less than a second.

Exercise 20. Write a recursive static method that computes the value of $\ln(n!)$.

Solution. See the class `com.segarciat.algs4.ch1.sec.ex20.FactorialLog`. The implementation is fairly trivial if we recall the power rule of logarithms. If x, y are any two positive real numbers, then

$$\ln(xy) = \ln(x) + \ln(y)$$

Since $n! = n \cdot (n-1)!$ for $n \geq 1$ and $0! = 1$, we have:

$$\ln(n!) = \ln(n \cdot (n-1)!) = \ln(n) + \ln[(n-1)!]$$

Exercise 21. Write a program that reads in lines from standard input with each line containing a name and two integers and then uses `printf()` to print a table with a column of the names, the integers, and the result of dividing the first by the second, accurate to three decimal places. You could use a program like this to tabulate batting averages for baseball players or grades for students.

Solution. See the class `com.segarciat.algs4.ch1.sec.ex21.StdinDivision`. Notice that this behaves much like the `cat` command in a UNIX-based system. Ideally, input comes from a while so that output will be redirected to a file so that it is not interspersed with the input.

Exercise 22. Write a version of `BinarySearch` that uses the recursive `indexOf()` given on page 25 and *traces* the method calls. Each time the recursive method is called, print the argument values `lo` and `hi`, indented by the depth of the recursion. *Hint*: Add an argument to the recursive method that keeps track of the depth.

Solution. See the class `com.segarciat.algs4.ch1.sec.ex23.BinarySearchTrace`. Since the purpose of the `depth` argument is to affect indentation, I decided to pass a `StringBuilder` object to control the indentation, which is appropriate because the recursion in this case is unidirectional.

Exercise 23. Add to the `BinarySearch` test client the ability to respond to a second argument: `+` to print numbers from standard input that `are not` in the whitelist, `-` to print numbers that *are* in the whitelist.

Solution. See the class `com.segarciat.algs4.ch1.sec.ex23.Filtering`.

Exercise 24. Give the sequence of values of p and q that are computed when Euclid's algorithm is used to compute the greatest common division of 105 and 24. Extend the code given on page 4 to develop a program `Euclid` that takes two integers from the command line and computes their greatest common divisor, printing out the two arguments for each call on the recursive method. Use your program to compute the greatest common divisor of 1111111 and 1234567.

Solution. Euclid's algorithm computes successive remainders. Hence if $p_0 = p$, $q_0 = q$, then $r_n = p_{n-1} \mod q_{n-1}$ for integer $n \geq 1$. If r_n is 0, then the greatest common divisor is r_n. Otherwise, we set $p_n = q_{n-1}$ and $q_n = r_n$.

p	q	r
105	24	9
24	9	6
9	6	3
6	3	0

See the class `com.segarciat.algs4.ch1.sec.ex24.Euclid`.

Exercise 25. Use mathematical induction to prove that Euclid's algorithm computes the greatest common divisor for any pair of nonnegative integers p and q.

Solution.

Proof. Suppose that $q = 0$. Then the algorithm returns p as the greatest common divisor and we are done.

Suppose now that p and q are nonnegative integers with $q > 0$. Let $k \in \mathbb{N}$, and define:

$$\begin{aligned} p_0 &:= p \\ q_0 &:= q \\ r_{k-1} &:= p_{k-1} \mod q_{k-1}, \\ p_k &:= q_{k-1} \\ q_k &:= r_{k-1} \end{aligned}$$

That is, r_{k-1} is the remainder of dividing p_{k-1} by q_{k-1}. If $r_{k-1} \neq 0$, then the algorithm continues to produce values for these sequences. The `p% q` operation in the algorithm translates into the $p_{k-1} \mod q_{k-1}$ operation above, which always produces an integer r_{k-1} satisfying $0 \leq r_{k-1} < q_{k-1}$. Hence, the sequence of successive quotients, $k \mapsto q_k$, form a strictly monotonically decreasing sequence of integers. Since it is strictly decreasing, the values decrease by at least 1 after each step, so the sequence must converge to 0 after at most q steps, meaning $q_k = 0$ for some $k \in \mathbb{N}$. By assumption we know $q_0 \neq 0$, so let N the smallest nonnegative integer such that $q_{N+1} = 0$, and set $d = q_N$. Then the algorithm will return d.

We must prove that:

(i) d is a common divisor of p and q.

(ii) d is the greatest among all the divisors of p and q.

To prove (i), note that $r_N = q_{N+1} = 0$, which means $r_N = 0$, and hence, q_N divides p_N. Since $d = q_N$, this implies that d is a common divisor of p_N and q_N. If $N = 0$, then $p_N = p$ and $q_N = q$, and we are done. Suppose that $N > 0$. Then $p_N = q_{N-1}$ and $q_N = r_{N-1}$, and d divides q_{N-1} and r_{N-1}. By the definition of r_{N-1}, there is a non-negative integer s_{N-1} such that

$$p_{N-1} = s_{N-1} \cdot q_{N-1} + r_{N-1}$$

Since d divides q_{N-1} and r_{N-1}, it follows that d divides p_{N-1}. Proceeding by induction, we conclude that d divides p_k and q_k for all $k \in \mathbb{N}$. Since $q_0 = p_1$ and $r_0 = q_1$, the definition of p_0 in terms of q_0 and r_0 implies that d divides p_0 also. Hence, and in turn, d divides $p_0 = p$ and $q_0 = q$, proving that d is a common divisor of p and q. This completes the proof of (i).

To prove (ii), suppose that d' is the greatest common divisor of p and q. Then $d \leq d'$, by definition. If we can prove that we also have $d' \leq d$, then both conditions together imply that $d = d'$. Recalling that $d = q_N$, suppose that $N = 0$. Since d' is the greatest common divisor, it must divide p and q. In particular, d' divides d because $q = q_0 = d$. Hence, we must have $d' \leq d$. Suppose now that $N > 0$, that $k \in, \mathbb{N}$, and d' divides p_{k-1} and q_{k-1}. By definition of r_{k-1}, we can write

$$r_{k-1} = p_{k-1} - s_{k-1} \cdot q_{k-1}$$

for some non-negative integer s_{k-1}. Since d' divides p_{k-1} and q_{k-1}, it must divide r_{k-1}. In particular, d' divides q_{k-1} and r_{k-1}. Since $p_k = q_{k-1}$ and $q_k = r_{k-1}$, this means d' divides p_k and q_k. By induction, d' divides p_k and q_k for all $k \in \mathbb{N}$. In particular, d' divides q_N, and since $d = q_N$, this means d' divides d. We conclude that $d' \leq d$. $\square$

Exercise 26. *Sorting three numbers.* Suppose that the variables `a`, `b`, `c`, and `c` are all of the same numeric primitive type. Show that the following code puts `a`, `b`, and `c` in ascending order:

```
if (a > b) { t = a; a = b; b = t; }
if (a > c) { t = a; a = c; c = t; }
if (b > c) { t = b; b = c; c = t; }
```

Solution.

Proof. Suppose that `a > b` so that the body of the first `if` runs. The effect is to swap `a` and `b`. After the first `if` statement, we have `a <= b`, but we do not yet know how `c` compares.

Suppose that `a > c`. This means that `b >= a` and `a >= c`. If the body of the second `if` runs, then it swaps `a` and `c`. At this point, we will have `b >= c` and `c >= a`. If the body of the `if` does not run, then we can certainly assure that `b >= a` and `c >= a`.

At this point we can be sure that `a` has the smallest value. The last `if` tests to see if `b` is greater than `c`, in which case it swaps them. After this, we will have `c >= b` and `b >= a`, so that `a`, `b`, and `c` are in ascending order. □

Exercise 27. *Binomial distribution.* Estimate the number of recursive calls that would be used by the code

```
public static double binomial(int n, int k, double p)
{
   if ((n == 0) && (k == 0)) return 1.0;
   if ((n < 0) || (k < 0 )) return 0.0;
   return (1 - p)*binomial(n-1, k, p) + p*binomial(n-1, k-1, p);
}
```

to compute `binomial(100, 50, 0.25)`. Develop a better implementation that is based on saving computed values in an array.

Solution. The sequence of invocations, top-to-bottom, is as follows:

```
                         binomial(100, 50, p)
binomial(99, 50, p),   binomial(99, 49, p)
                                          ⋮
```

Notice that each row corresponds to a separate value of n. Moreover, if r denotes the r-th row, starting at row $r = 0$ (the initial invocation), then i can be as large as 100, and the r-th row has 2^r such invocations. Therefore, a (not-so-tight) upper bound on the number of recursive invocations is:

$$\sum_{r=0}^{100} 2^r = 2^{101} - 1 - 1 = 2^{101} - 2$$

Notice the extra -1 because the initial invocation `binomial(100, 50, p)` is not a recursive call. Beyond row $i = 100$ we will have $n = -1$, which will not lead to a recursive invocation due to the guarding `if` statements. However, when $i = 50$, the value of k is 0 for the rightmost column, and 50 for the leftmost column. Hence, the rightmost instance of the call, namely `binomial(50, 0, p)`, will not lead to a recursive call thereafter, because of the `if` that guards when $k = 0$. Hence, starting at $i = 51$ (meaning $n = 49$), some recursive invocations will begin to return because of k reaching its threshold.

See `com.segarciat.algs4.ch1.sec1.ex27.BinomialDistribution`. I implemented my algorithm using recursion. Meanwhile, the solution by the authors employs a non-recursive technique that they call "memoization".

Exercise 28. *Remove duplicates.* Modify the test client in `BinarySearch` to remove any duplicate keys in the whitelist after the sort.

Solution. See the `com.segarciat.algs4.ch1.sec1.ex28.RemoveDuplicates` class. After devising a way to remove the duplicates, I used the `Array.copyOf()` method to create a new array large enough to hold the unique values.

Exercise 29. *Equal keys.* Add to `BinarySearch` a static method `rank()` that takes a sorted array of `int` values (some of which may be equal) and a key as arguments and returns the number of elements that are smaller than the key and a similar method `count()` that returns the number of elements equal to the key. *Note*: If `i` and `j` are the values returned by `rank(a, key)` and `count(a, key)`, respectively, then `a[i..i+j-1]` are the values in the array that are equal to key.

Solution. My `rank()` algorithm uses a modified version of the `indexOf()` algorithm. We can collapse the three `if-else` branches from `indexOf()` into two:

(i) If the target entry is larger than the middle key, increment `lo` to search starting at `mid + 1`.

(ii) Otherwise, decrement `hi` to end the search at `mid - 1`.

This guarantees that, at each step, `key` is larger than `a[lo]`. As usual, the search ends when `lo` exceeds `hi`. Because we never `break`, the algorithm takes all keys into consideration. This of course necessitates that the array be sorted to exploit transitivity.

The `count()` algorithm can the be implemented in a couple different ways. One way is as `rank(a, key + 1) - rank(a, key)`. Another way is to implement a modified version of `rank()`, call it `rankGe()` that makes a single change: replace `key > a[mid]` with `key >= a[mid]` in the `if` statement that controls the assignment operation on `lo`. Then we can implement `count()` as `rankGe(a, key) - rank(a, key)`. The former approach would work only for integer data, but the latter would work for other data types.

Exercise 30. *Array exercise.* Write a code fragment that creates an n-by-n boolean array `a[][]` such that `a[i][j]` is `true` if `i` and `j` are relatively prime (have no common factors), and `false` otherwise.

Solution. See the `com.segarciat.algs4.ch1.sec1.ex30.RelativelyPrimeArray` class.

Exercise 31. *Random Connections.* Write a program that takes as command-line arguments an integer `n` and a `double` value `p` (between 0 and 1), plots `n` equally spaced dots of size 0.05 on the circumference of a circle, and then, with probability `p` for each pair of points, draws a gray line connecting them.

Solution. See the `com.segarciat.algs4.ch1.sec1.ex31.RandomConnections` class.

References

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