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. 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`. See text the class `com.segarciat.algs4e._03.Compare3Integers` under the `code` folder.

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)
```

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

Solution.

(a) The iterations are computed as:

$$t_0 = 9.0 \quad ; \quad t_0 - \frac{9.0}{t_0} = 1 \quad ; \quad \rightarrow \quad t_1 = \frac{9.0/t_0 + t_0}{2.0} = 5.0$$
$$t_1 = 5.0 \quad ; \quad t_1 - \frac{9.0}{t_1} = 3.2 \quad ; \quad \rightarrow \quad t_2 = \frac{9.0/t_1 + t_1}{2.0} = 3.4$$
$$t_2 = 3.4 \quad ; \quad t_2 - \frac{9.0}{t_2} \approx 0.75294 \quad ; \quad \rightarrow \quad t_3 = \frac{9.0/t_2 + t_2}{2.0} = 3.023529411764706$$
$$t_3 \approx 3.02352 \quad ; \quad t_3 - \frac{9.0}{t_3} \approx 0.04687 \quad ; \quad \rightarrow \quad t_4 \approx \frac{9.0/t_3 + t_3}{2.0} = 3.00009155413138$$
$$t_4 = 3.00009155413138 \quad ; \quad t_4 - \frac{9.0}{t_4} = 0.000183105468792632 56$$

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.algs4e._14.LgFloor` in the `code` folder. 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`.

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

```
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.

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

Solution. 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:

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

The implementation for this is in the `factorialLog` method in the `com.segarciat.algs4e._20.FactorialLog` class.

References

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