Introduction to Algorithms and Implementations: Three Simple Problems

Nicholas Bennett
nickbenn@g-r-c.com

March 2017

Copyrights and Licenses

Document

Python, Java, and NetLogo Source Code

Last modified: 2 March 2017.

Preface

Objective

This document is intended as preparation, background, or additional material for the accompanying OpenOffice/LibreOffice presentation by the same name. While this document and the accompanying presentation aim to introduce some important programming and algorithmic concepts through their application to some well-known problems, they're only an introduction: this is neither a broad programming tutorial nor an in-depth lesson on algorithms.

Audience

Instructors should have enough programming experience to be able to find and correct syntax errors reported by a compiler or interpreter, and to correct logical errors in simple Python and Java programs. College-level algebra experience (at least) is highly recommended for instructors.

This lesson should not be a student's first hands-on experience with programming – though previous Python or Java experience isn't strictly necessary. Minimally, students should be comfortable writing assignment, conditional, and iteration statements, as well as subroutines (functions, procedures, or methods), in some programming language. Previous completion of first-year algebra and first-year geometry is assumed, and previous completion of – or current enrollment in – second-year algebra is highly recommended.

Implementation Languages

In the algorithm implementations in this lesson, we'll focus on concepts that aren't tied to any single programming language, or family of languages. However, we have to pick some specific languages for implementations; we'll be writing ours in Python [2] and Java [3]. Additionally, though it's not presented or discussed in this document, NetLogo implementations of The Tower of Hanoi accompany this document [4]; for more information on the NetLogo implementations, along with curriculum content roughly corresponding to the Tower of Hanoi material in this document, see the NetLogo models' **Info** pages.

Python was invented in the late 1980s, but in the last 10 years its user base and applications have increased dramatically; by some measures it has overtaken Java, C, and C++ as the most popular programming language [5]. For our purposes, it has two key benefits: Its relatively simple syntax for basic operations make it a very good language for beginners, while placing few limits on experienced programmers who want to use its advanced features. In addition, it lends itself to multiple fundamentally different approaches to many problems. These attributes make Python an excellent language not only for doing programming work, but also for exploring many important concepts in computer science.

Apart from its broad user base and its extensive ecosystem of third-party open source and commercial software, Java has important educational and productivity benefits: it shares much of its syntax with several other languages that are – like Java – closely related to C, helping us leverage what we learn across multiple languages. At the same time, its strictness in data typing, syntax, and memory handling can help beginning programmers (and experienced ones as well) catch and avoid many errors that can complicate the programming task in some of the other C-derived languages.

NetLogo is an agent-based modeling and simulation language developed by Uri Wilensky at The Center for Connected Learning and Computer-Based Modeling of Northwestern University. In this case, we're using NetLogo for its visual capabilities, and for the ease it offers in building and presenting a user interface that includes not only graphical controls, but also a REPL (read-evaluate-print loop) console.

Language Versions

The Python code in this lesson was tested successfully using several Python versions from 2.6.8 to 3.6.0, and should run with all Python 2.6 through 3.6 releases.

The Java code in the lesson was tested successfully with several JDK versions from 1.6 update 45 to 1.8 update 121; it should compile and run with all JDK 1.6, 1.7, and 1.8 releases.

The NetLogo code accompanying this document is provided in 2 versions: one compatible with NetLogo v5.x, and another compatible with NetLogo 6.0.

Introduction

Recurrence and Algorithms

In learning to program (or in acquiring almost any set of skills), much of our time is spent learning to perform tasks or solve problems that are *recurrent*. In other words, even if we're not completely aware of it, we're learning to handle problems that we face over and over. The problems aren't duplicated exactly each time, but they're similar enough that what we've learned in the past helps us now and in the future.

One of the ways we learn to perform a recurrent task is by learning a repeatable *procedure*, a series of steps that can be applied to it. Often, these procedures are general enough that we can adjust for some variation in the circumstances or in the task itself. Sometimes, these steps are written down, as they often are in cooking recipes and driving directions; sometimes they're more a matter of acquired habits or routines.

If we take a step back and look at these procedures we're following, we might recognize some of them as *algorithms*. Paraphrasing Donald Knuth's definition in *The Art of Computer Programming* [1], an algorithm is a procedure with a definite precisely defined, finite sequence of steps, effective for solving a generalized problem, which can be applied to instances of the problem that vary in their inputs, and produce output accordingly. While such procedures are common in virtually all domains of expertise, the word "algorithm" isn't often used outside of mathematics and computer science. Nonetheless, we're learning, creating, and using procedures for performing recurrent tasks all the time – and some of these procedures can be considered algorithms.

In this lesson, we'll explore three simple problems – a couple of which you might have encountered already as puzzles or games. Two of the problems include specific questions for you to answer, but *how* you go about finding those answers is much more interesting than *what* the answers are. For each problem, we'll explore an approach for solving it in its general form. In other words, we'll construct and implement algorithms. We'll then apply these algorithms to answer the specific questions for each problem.

Initially, we won't worry too much about whether our algorithms – or our implementations of them – are the most efficient or the most elegant forms possible. But in some cases, we'll discuss ways the efficiency might be improved.

Greatest Common Divisor of Two Positive Integers

Description

Most of us are familiar with the problem of finding the greatest common divisor (also called the greatest common factor) of two integers, where at least one of them is not zero. This is the largest positive integer that evenly divides (with no remainders) both of the original numbers. For example, both 28 and 21 are divisible by 7, and there isn't a larger integer that divides both 28 and 21; thus, 7 is the greatest common divisor of 28 and 21.

So far, so good. Now let's make it more difficult.

Question

- What's the greatest common divisor of 553,237 and 448,043?

If your instinct is to look for the factors of one or both numbers by hand – or even using a general purpose calculator or spreadsheet program to do so – you'll probably have a lot of work ahead of you. Can we do better than that? Is there a method that can solve the problem efficiently for *any* pair of positive integers?

Let's begin by introducing some useful notation:

$\vert$	Divides. When some integer *a* is divisible by a nonzero integer *c*, with no remainder left over, we say that *c* divides *a*, written $c \vert a$.
GCD	Greatest common divisor, used as both an initialism and a function. The greatest common of two integers, *a* and *b*, is written as the GCD of *a* and *b*, or $\mathrm{GCD}(a,b)$.

Based on the definitions of GCD and divisibility, here are a few simple statements that will come in handy (we won't prove these; that's left as an exercise for the mathematically inclined reader):

If *a* is a non-zero integer, then $a|a$ and $a|0$. (1)

If $c|a$ and $c|b$, then $c|(a+b)$ and $c|(a-b)$. (2)

If *c* and *a* are integers, with $c > 0$ and $c|a$, then $\mathrm{GCD}(a,c) = c$. (3)

If $\mathrm{GCD}(a,b) = c$, then $c|a$ and $c|b$. (4)

If $c|a$ and $c|b$, then $\mathrm{GCD}(a,b) \geq c$. (5)

Note the inequality in (5): there are many values of *a*, *b*, and *c* for which $c|a$ and $c|b$, but $\mathrm{GCD}(a,b) > c$. For example, $5|15$ and $5|45$, but $\mathrm{GCD}(45,15) = 15$.

In searching for a general method for finding the GCD, can you think of a way to take advantage of the above statements? Is there a way to express the problem in visual terms that will help?

Visualization and Analysis

In Figure 1, there are two rectangles: one of height a, and the other of height b, with $a > b$. Assume that both have unit width, so that their areas are also a and b Assume that $\text{GCD}(a, b) = c$; thus, we can completely fill up the a and b rectangles with rectangles of height c and width 1.

Note that the breaks in the rectangles indicate that some unknown portion of the rectangles are left out. So all we know about the relative sizes of a, b, and c is what we've been given, and what we can deduce from it. For example, from $\text{GCD}(a, b) = c$ we know that $c|a$ and $c|b$.

On the other hand, the illustration does help us see clearly that if $c|a$ and $c|b$, then $c|(a-b)$. That's an important piece of the general solution – but it would be even better if we could prove that $\text{GCD}(a-b, b) = c$. However, we need to be careful: how do we know that there isn't another integer d, larger than c, that divides both $(a-b)$ and b? To answer that, let's assume that there is such an integer d, and see if that assumption leads us to a contradiction.[1]

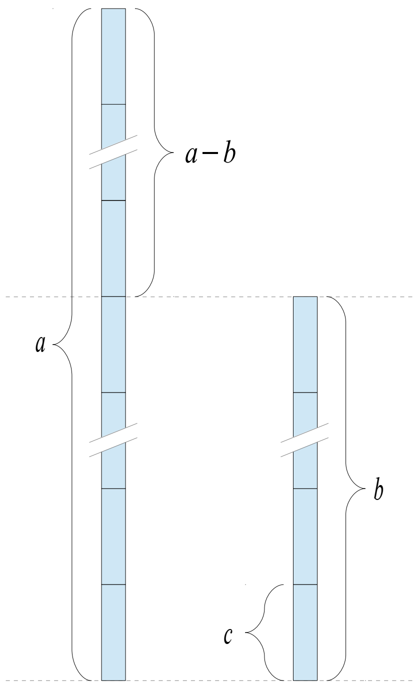


Figure 1: GCD with rectangles

Given positive integers[2] a, b, and c, with

$$\text{GCD}(a, b) = c\text{, and } a > b > 0. \tag{6}$$

Assume

$$\text{GCD}(a-b, b) = d\text{, and } d > c. \tag{7}$$

By (4), assumption (7) implies that

$$d|(a-b)\text{, and } d|b. \tag{8}$$

But according to (2), it must then be true that

$$d|[(a-b)+b]\text{; that is, } d|a. \tag{9}$$

According to (8) and (9), d divides both a and b. However, this contradicts (6) and the definition of GCD, which say that there can be no positive integer greater than c that divides both a and b. Therefore assumption (7) must be wrong.

1 Here, we're using a method of proof known as *reductio ad absurdum* (Latin: "reduction to the absurd"). To prove a statement, we may start by assuming the opposite. If the line of reasoning (with no further assumptions) leads to a contradiction, then the assumption must be wrong – i.e. the original statement is proved.

2 Limiting our argument to positive integers fits the visualization, but the restriction is easy to remove later.

In summary, if

$$\text{GCD}\left(a,b\right) = c \text{ and } a > b > 0,$$

then

$$\text{GCD}\left(a-b,b\right) = c. \tag{10}$$

By showing that $\text{GCD}\left(a-b,b\right) = c$, we've reduced the problem of finding $\text{GCD}\left(a,b\right)$ to that of finding $\text{GCD}\left(a-b,b\right)$. We can continue subtracting the smaller of the two numbers from the larger, to get smaller and smaller pairs of integers – but how long can we keep that up?

When we reach a point where the two integers are equal, we can stop: According to (1) and (3), if $a = b$, then $\text{GCD}\left(a,b\right) = a = b$. So we could start by examining *a* and *b*: if they're equal, then the GCD is equal to both, and we're done; otherwise, we subtract the smaller of *a* and *b* from the larger, and repeat the process – including testing for equality – with the new values. This is how we'll find the GCD.

Algorithm

Let's describe the algorithm more formally. This will refine our understanding of it; also, by expressing it in unambiguous, structured terms, we'll make the subsequent implementation a relatively straightforward process.

```
To find the GCD of positive integers $a$ and $b$:
    1. While $a \neq b$, repeat the following:
        • If $a > b$, then
            ◦ let $a = a - b$;
          otherwise,
            ◦ let $b = b - a$.
    2. Done. The current value of $a$ is the GCD of the original values of $a$ and $b$.
```

Algorithm 1: Finding the GCD of two positive integers

Implementation

To translate from this description of the algorithm to an implementation in some programming language is mostly a matter of understanding how to translate the essential elements – e.g. variable assignment, conditional execution, *conditional iteration* (repeated execution of one or more statements as long as a given condition is true), function definition, and returning function results. We'll write our implementation in Python, which makes some of this translation relatively easy. For example, in Python – as in several widely used programming languages – the syntax for conditional execution and iteration is quite close to standard English.

The function below implements our GCD algorithm. After the code making up the function definition (lines 1-7), the code that starts with **`if __name__ == '__main__'`** (lines 9-13) invoke the function – i.e. execute the algorithm – for the pair of numbers given on page 7. By changing the values assigned to **`a`** and **`b`** on lines 11-12, you can run the algorithm to find the GCD of any pair of positive integers. Alternatively, you can save the code as a Python module, and run the algorithm from the Python command prompt (or another script or module file) by importing and invoking the function for the desired values of **`a`** and **`b`**.

Using the editor of your choice, type and save the code on Listing 1 (without line numbers). The lesson materials include the **`Python/euclid.py`** file; it has several of the lines already written, including a *stub* (an incomplete – though often syntactically valid – function, used as a placeholder for the code to be written) of the **`gcd`** function, and can be used as a starting point. (If you start with that file, be sure not to duplicate the code that's already there. Also, note that the code you add should replace any **`TODO`** comments and **`pass`** statements.)

Be careful to preserve the indentation and character casing shown in the listing. Python is a case-sensitive language (note that all keywords are written in lower case characters), and indentation is crucial in Python syntax. (Most Python-aware text editors will help you maintain correct indentation.)

```
 1 def gcd(a, b):
 2     while a != b:
 3         if a > b:
 4             a -= b
 5         else:
 6             b -= a
 7     return a
 8
 9
10 if __name__ == '__main__':
11     a = 553237
12     b = 448043
13     c = gcd(a, b)
14     print("GCD({0}, {1}) = {2}".format(a, b, c))
```

Listing 1: Python implementation of GCD algorithm with invocation

There are a few important things to notice in the code – especially if this is your first experience with Python.

- Function definitions in Python start with the keyword **`def`** (as seen on line 1), followed by the function name, then the parenthesized parameters (even if a function takes no parameters, the parentheses are required), and finally a colon. The body of the function follows on the subsequent lines, indented to the right (this indentation is required).
- Unlike languages more closely related to C, semicolons aren't used to terminate simple statements in Python. Instead, such statements usually end when the line ends.

- In the compound statements for conditional execution (**`if`**) and conditional iteration (**`while`**), used on lines 3 and 2 (respectively), the keyword is followed by a *Boolean* (true or false) expression that controls execution, and then a colon; the *suite* (one or more statements that are executed under control of the compound statement) follow on the subsequent lines, indented to the right. (Alternatively, a simple suite may be written directly after the colon, on the same line.) When an **`if`** statement includes an **`else`** clause, this follows the first suite, and is itself followed by a suite. In any case, without the colon, or the suite that follows, or proper indentation of the suite, the compound statement isn't syntactically correct.
- The inequality comparison is written as **`!=`** (with no space between **`!`** and **`=`**), and the equality comparison as **`==`**. (This is also the case in Java and many other languages.)
- In Python (and most C-derived languages), **`a = a - b`** and **`a -= b`** are equivalent statements. The second form is one of the many *augmented assignment* statements supported in these languages; in some cases, they're executed more efficiently than the longer forms, but many programmers prefer them primarily for their compactness.
- The predefined **`__name__`** variable and **`__main__`** text value (on line 10) both include 2 underscores at the beginning and 2 more underscores at the end.
- In line 14, **`format`** is used to substitute parameter values into a text string. Here, **`{0}`**, **`{1}`**, and **`{2}`** are *replacement fields* with *positional arguments:* the first, second, and third parameters in the invocation of **`format`** will be substituted in place of these fields to produce the resulting string. (As in many programming languages, counting in Python usually starts with 0, instead of 1.)

Solution

Run your code (correcting any errors preventing it from being interpreted or executed), and check the answer. If you're using a development environment like Spyder, Eclipse with PyDev, or PyScripter (or, to a lesser extent, the IDLE environment included with most Python installations), you can edit and run directly from within that environment. However you execute your program, you should see this console output:

```
GCD(553237, 448043) = 149
```

Output 1: GCD of 553,247 and 448,043

Enhancements and Alternatives

You probably noticed that our description of the algorithm states explicitly that *a* and *b* are positive integers. However, our implementation doesn't catch violations of that condition; nor does it handle negative or zero values in way that makes sense for the GCD. Because of this, we can easily break our code by invoking the **`gcd`** function with invalid parameters. This might or might not be a problem – it mostly depends on how the implementation will be used. In any case, input validity is an important consideration in programming.

The description and implementation of our algorithm both include the possibility of repeated subtractions. For example, if a is much larger than b, we'll have to subtract b from a several times, until $a \leq b$. With some adjustments to the algorithm, we can get the same result more efficiently (in most cases) by instead computing the remainder when a is divided by b. This computation of the division remainder, called the *modulo operation* (or *remainder operation*, or *modular division*) and written as a mod b, is supported directly in most programming languages, and gives (essentially) the same result as repeated subtractions. Note that modifying our implementation to use modular division would require that we also modify the iteration condition: instead of repeating while $a \neq b$, we would need to repeat while the division remainder is non-zero.

We might also consider extending the algorithm description and implementation to include negative integers as inputs.[3] (Note that the definition of the GCD itself doesn't change for negative integers; it's still defined as the largest *positive* integer that divides both input values.) We could also extend it to finding the GCD of more than 2 integers.

Of course, the GCD algorithm can be implemented in virtually any programming language; each has its own strengths and weaknesses. For example, in languages that require declaration of variables and their types before use (such as Java), some errors caused by invalid inputs (noted above) would be caught earlier; on the other hand, the syntax of those languages can be more difficult for some to learn.

Historical Note

Even if you didn't know the algorithm for finding the GCD before now, you might suspect that others have known about it for some time. If so, you're right: the Greek mathematician Euclid invented the algorithm, describing it (almost exactly as we have) in Propositions 1 and 2 of Book VII of his *Elements* [6], around 2,300 years ago. Euclid's algorithm (as it's known, and as we'll refer to it from now on) is thus one of the oldest numerical algorithms still in use today.

3 We can also extend our algorithm to find the GCD where one or more inputs are equal to zero. It's fairly easy to show – and understand – that $GCD(a, 0) = |a|$. It's not quite as easy to see why we might define $GCD(0,0) = 0$, but doing so has certain theoretical benefits.

Fizz Buzz

Description

Fizz Buzz (also known as *Bizz Buzz*) is a simple counting game [7]. Players are arranged in a circle, and one player begins the count at 1. The turn moves around the circle, the count increasing by 1 with each player. Each player in turn says the current count out loud, except in these cases:

- When the count is divisible by 3, the player whose turn it is says “fizz”.
- When the count is divisible by 5, the current player says “buzz”.
- When the count is divisible by both 3 and 5, the current player says “fizz buzz”.

Any player who fails to say the count (or the appropriate fizz buzz combination) in his or her turn loses the round, and a new round begins with the next player restarting the count at 1.

A couple of rounds might play out like this:

- 1, 2, fizz, 4, buzz, fizz, 7, 8, fizz, buzz, 11, 12. (The last count should have been “fizz”, since 12 is divisible by 3; this error ends the round.)
- 1, 2, fizz, 4, buzz, fizz, 7, 8, fizz, buzz, 11, fizz, 13, 14, fizz buzz, fizz. (The last count was 16, which isn't divisible by 3; thus, "fizz" was incorrect, and the round is over.)

Aside from being a rather silly game, in recent years Fizz Buzz has been used in some companies as a minimum-competence test when hiring programmers [8]: An interviewer might ask a candidate to write a program that prints out the numbers from 1 to 100 in order, substituting “fizz”, “buzz”, and “fizzbuzz” for the appropriate values. A candidate who's unable to do this in a few minutes will presumably have even more difficulty when faced with a real world programming task in the workplace.

Task

For this problem, we must come up with an algorithm that can be used to perform the task described above: Output the integers from 1 to 100 in order – but replace all multiples of 3 with “fizz”, replace all multiples of 5 with “buzz”, and all multiples of both 3 and 5 with “fizzbuzz” (but not “buzzfizz”). Note that this is a different task than playing the game as one of several players: essentially, our algorithm will be acting as every player in the game.

Begin by writing an algorithm for this task on paper. There are several effective procedures for Fizz Buzz; whichever you choose, try to make it sufficiently complete and unambiguous that you could give it to someone else for implementation. Among other things, this means that you should specify exactly how multiples of 3 or 5 should be generated or tested.

Algorithm

As noted above, there are many algorithms that can be used for Fizz Buzz. The one described here isn't necessarily the most efficient, but it's easy to express and understand – once we add a few new symbols to our vocabular:

$\leftarrow$	Assignment of the value on the right-hand side of the arrow to the variable on the left-hand side. Using this symbol helps us distinguish between assignment and equality testing (for which we're using the equal sign, =).
$\circ$	Concatenation (combination) of one string of characters with another. For example, if *s* has the value “fizz”, and *t* has the value “buzz”, then the value of $s \circ t$ is “fizzbuzz”.
mod	*a* mod *b* performs integer division of *a* by *b*, and returns the remainder. This is the *modulo operation* mentioned in the “Enhancements and Alternatives” for Euclid's algorithm (page 11).

```
To produce correct Fizz Buzz output for the first $N$ positive integers:
  • Repeat the following with $i$ taking each of the values 1, 2, 3, …, $N$ (in order):
      ▫ If $i \bmod 3 = 0$:
          ▪ $s \leftarrow$ "fizz";
        otherwise,
          ▪ $s \leftarrow$ "" (an empty text string).
      ▫ If $i \bmod 5 = 0$:
          ▪ $s \leftarrow s \circ$ "buzz".
      ▫ If $s =$ "":
          ▪ $s \leftarrow i$.
      ▫ Print $s$.
```

Algorithm 2: Fizz Buzz

Implementation

This time, we'll use Java for our implementation. Java requires that all code be part of a *class* – a structure for grouping related code and data – so we'll have a **`fizzBuzz`** method (by convention, functions defined in a class are called *methods*) that implements the algorithm, and a **`main`** method that invokes it, all contained in a class named **`FizzBuzz`**. The implementation shown below corresponds to the algorithm above; if you came up with a different algorithm, and if you're feeling ambitious, try implementing your algorithm.

As you type the code, you'll probably notice that Java seems to require more lines of code than Python does for similar operations. For example, rather than using indentation to group a suite of statements together, Java (along with C/C++/C#, JavaScript, and PHP) uses curly braces to enclose a *statement block*, and we usually write the closing brace (at least) on its own line. Also, defining a class adds extra lines of code to even very simple Java programs. However, these additional lines of "housekeeping" code quickly become almost automatic for experienced programmers, and most Java-aware text editors generate some of these lines automatically.

Note that this algorithm, like Euclid's algorithm, includes *iteration* – repeated execution of a group of statements. In Euclid's algorithm, we used *conditional iteration*: we executed some statements repeatedly while a specified condition was true. Here, we need to repeat the execution of a group of statements for every integer value in a specified range; instead of **`while`**, we'll use the **`for`** statement, which is specifically intended for iteration over a range of numbers or over the items in a collection.

When typing your code, make sure you have balanced braces, correct character casing (like Python, Java is case-sensitive, and all keywords are written in lower case letters), and the required semicolons at the end of all simple statements. Also, while Java doesn't require indentation, it's a good idea to use it for your own benefit – as well as the benefit of others who might want to read and understand your code. As before, there's a file for you to start with: **`Java/FizzBuzz.java`** doesn't include the algorithm implementation, but it includes the **`FizzBuzz`** class structure and the completed **`main`** method, along with a stub of the **`fizzBuzz`** method.

```java
public class FizzBuzz {

    public static void fizzBuzz(int limit) {
        for (int i = 1; i <= limit; i++) {
            String output = (i % 3 == 0) ? "fizz" : "";
            if (i % 5 == 0) {
                output += "buzz";
            }
            if (output.isEmpty()) {
                output = Integer.toString(i);
            }
            System.out.println(output);
        }
    }

    public static void main(String[] args) {
        int limit = 100;
        fizzBuzz(limit);
    }

}
```

Listing 2: Java implementation of Fizz Buzz algorithm with invocation

In some ways, the rules and conventions of Java are more demanding than those of Python, so there are several important details to note in the code:

- A Java source file can contain only one class declared as **`public`**; that class *must* have *exactly* the same name as the source file, minus the file extension. Because we're creating a **`public`** class called **`FizzBuzz`**, it must be saved in a file called **`FizzBuzz.java`**; from that, the Java compiler will create a compiled file called **`FizzBuzz.class`**.
- By convention, Java class names start with upper-case letters, method and variable names start with lower-case letters, and all of these names use upper-case initial letters for subsequent words in multiple-word names. Thus, we have a class named **`FizzBuzz`** (line 1), a method named **`fizzBuzz`** (line 3), etc.
- In Java, all variables must be declared – with their data types – before use. A value can be assigned to a variable in the same statement as its declaration, as in line 5.
- All Java methods (except for *constructors*, a special kind of method we won't be dealing with here) must also be declared with a type; this is the data type of the result returned by the method. If a method doesn't return a value, the declared type is **`void`**.
- Each Java method has a *scope*, which determines the extent to which it is visible and can be invoked. A **`public`** method can be called by code inside or outside the class; a **`private`** method can only be called from inside the class containing it; a **`protected`** method can be called from within the class and any subclasses (other classes based on that class). Because our **`fizzBuzz`** method is **`public`**, it can be invoked from code in the the **`FizzBuzz`** class, or from code in another class.
- In addition to being a container for related methods, a class can be used to define a new data type – i.e. a new type of variable whose behaviors are the methods of the class. When we're defining methods that are intended primarily for the first purpose, we generally declare them with the **`static`** keyword. A specialized data type probably wouldn't be very useful here, so we've declared our **`fizzBuzz`** method **`static`**.
- The form of the Java **`for`** statement used in line 4 starts with that keyword, followed by 3 semi-colon separated fragments enclosed in a single pair of parentheses: the *initialization*, in which a variable is generally declared and assigned an initial value; the *iteration condition*, evaluated before each iteration; and the *update*, in which one or more variables are updated. Finally, the parentheses are followed by the statement – either a simple statement or a statement block enclosed in braces – to be iteratively executed.
- The update fragment of the **`for`** statement on line 4 uses the **`++`** operator to increment a variable – that is, to update its value by adding 1. (Similarly, **`--`** can be used to decrement a variable.) For iterating over a range of integer values, the statement pattern **`for (int i = `*`initialValue`*`; i < `*`limitValue`*`; i++) {…}`** is very widely used.
- Line 5 uses the Java *conditional* or *ternary* operator. This is an expression following the form ***`condition`*`  ?  `*`value1`*`  :  `*`value2`***; the computed value of the expression is ***`value1`*** if **`condition`** is **`true`**, and ***`value2`*** otherwise.

- In Java conditional statements (as well as those in most other C-derived languages), the Boolean expression following `if` (or `while`) must be enclosed in parentheses.
- The modulo operation ("*a* mod *b*" in the algorithm description) is written as `a % b` (see line 6). This is true not just in Java, but in Python and many other languages.
- String concatenation in Java uses the `+` symbol; also, the *compound assignment* (called *augmented assignment* in Python) operator `+=` is supported for strings. So the statement in line 7 appends the text string `"buzz"` to the text string stored in the variable `output`, and assigns the result to `output`.
- The `printf` method (used in line 12) is similar to the C function of the same name: it formats and outputs one or more values according to a format string.
- Every Java program must have at least one `public static void` method named `main`, taking an array of `String` as a parameter. This method serves as the program's starting point (much like `if __name__ == '__main__'` in Python).

Verification

For this problem, we didn't have a specific question to answer; our task was to write the algorithm description and implementation, and we've done that. However, we need to execute the program and examine its output to verify the correctness of our implementation.

Compile and run your program, fixing any errors reported by the compiler or runtime output. If you're using an integrated development environment (e.g. NetBeans or Eclipse), you can compile and run your code directly from within that environment. (In addition, most syntax errors are caught and flagged before compilation in these environments.)

The first 15 lines of program output should look like this:

```
1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
11
fizz
13
14
fizzbuzz
```

Output 2: Fizz Buzz output for 1 through 15

You probably don't need to worry about checking every line of output; verifying these key features should be sufficient:

- There should be no blank output lines: non-blank output (not always numeric) must be produced for every value from 1 to 100.
- If the lines corresponding to the first few multiples of 3 – i.e. 3, 6, 9, 12 – are displayed as “fizz”, then you probably don't need to verify all the multiples of 3. The same goes for the display of “buzz” in place of the first few multiples of 5. (But don't forget about multiples of 15; see the next point.)
- The lines corresponding to 15, 30, 45, 60, 75, and 90 – all divisible by both 3 and 5 – must be displayed as “fizzbuzz”.
- There should always be a gap of 2 non-“fizz” values between each “fizz” pair, and a gap of 4 non-“buzz” values between each “buzz” pair.
- If one of the “fizz”, “buzz”, or “fizzbuzz” substitutions is displayed, a number must not be displayed on the same line.

Enhancements and Alternatives

Because Java is a *strongly typed language* (i.e. variables and methods are declared with associated types, and operations involving implicit type coercion or mixed types are limited), the compiler won't accept code that invokes the **`fizzBuzz`** method with a non-integer value (which would obviously be invalid); however, it can still be invoked with an invalid integer value (e.g. zero or a negative integer). As before, we need to be aware of this, and – if appropriate – address it with input validity checks in the code.

As stated on page 13, there are many different algorithms for Fizz Buzz, and even more ways to implement those algorithms. We invite you to come up with a few different algorithms and implementations (including implementations in other languages besides Java). For specific challenges, try these:

- Even though the modulo operation is relatively efficient (vs. floating-point division, for example), it still requires more processing time than a single addition or subtraction operation – which in turn requires more than an increment or decrement operation. Is there a way to implement a Fizz Buzz algorithm without any modulo operations?
- How might we measure the running time of alternative implementations, to compare their efficiency? Try to measure and interpret the timing results for a few different implementations. After doing that, do you think it's worth it to spend much time improving the efficiency of Fizz Buzz? Are there other computational tasks you can think of, for which efficiency is much more (or less) important than it is for Fizz Buzz?

The Tower of Hanoi

Description

This puzzle was invented by Édouard Lucas, a 19th-century French mathematician and the author of the puzzle (which he called the "Tower of Brahma") [9]. Lucas introduced the puzzle by relating a legend from India (though it's possible he invented the legend as well): In a certain temple in India, there's a room with 3 stout wooden posts, and 64 gilded disks of graduated sizes, each with a hole in the center. In the beginning, all of the disks were stacked on one post, with the largest at the bottom, decreasing in size to the top. For centuries, the Brahmin priests of the temple have been engaged in the task of rearranging the disks, with the aim of placing them all in a single stack on one of the other two posts. However, there are very strict rules about how the disks may be moved:

1. A move consists of lifting the top disk from a stack, sliding it off its post, then moving it to another post, where it becomes the top disk in that post's stack of disks.
2. No more than one disk may be moved at a time.
3. No disk may be stacked on top of a smaller disk.

According to the legend, when all 64 disks are transferred to a single stack on one of the other posts, the world will end.

Questions

1. What's the minimal *sequence* of moves required to transfer a stack of 5 disks from post 1 to post 2?
2. What's the minimum *number* of moves that must be performed by the Brahmin priests, to transfer all 64 disks from the starting post to a single stack on one of the other posts?

In thinking about the questions above, and thinking about the problem in general terms, it might be helpful to start with the following questions:

- What are the moves required to solve the problem for 1 disk? 2 disks? 3 disks?
- How are the sequences and numbers of moves required for 1, 2, and 3 disks related?
- In general, how does the sequence and number of moves required for n disks relate to the sequence and number required for (n + 1) disks?

Note that regardless of the number of disks, we will use no more than 3 posts for stacking them. (When 4 posts are used, the problem is sometimes called Reve's puzzle [10]. Can the puzzle be solved using fewer than 3 posts?)

Notation Conventions

For convenience and clarity, we'll refer to the posts and the disks by number while formulating our algorithm.

- The posts will be numbered (and labeled in our diagrams) 1, 2, and 3. These specific numbers will have additional importance when we create and implement our algorithm.
- When referring to an unknown post (for example, when writing a general algorithm for transferring a stack from one post to another), we'll use the capital letters A, B, and C. We mustn't assume, however, that $A = 1$, $B = 2$, and $C = 3$; these are variables, and any one of them can refer to any one of the three posts.
- Even though we won't show this in the diagrams, assume that the disks are numbered as well, with disk 1 being the smallest, disk 2 being the next larger in size, and disk n being the n^{th} smallest disk.
- We'll frequently refer to the n smallest disks, the $(n - 1)$ smallest disks, etc. Note that this usage doesn't necessarily refer to the total number of disks in the puzzle: as we will see, solving a puzzle for some number of disks requires that we solve it for smaller numbers of disks along the way.

Key Corollaries to the Rules

There are a few consequences of the rules which are easily overlooked. Getting them clear in our minds will help us avoid being overwhelmed by the problem.

If we need to transfer the n smallest disks – already arranged in a single stack – from one post to another, the rules guarantee the following:

- Those n smallest disks are already stacked in the proper order: disk 1 is on top, then disk 2 beneath it, increasing in size all the way to disk n (which may or may not be the bottom disk stacked on the post; there could be disks larger than disk n stacked beneath it).
- No disk larger than the n^{th} smallest is stacked on top of – or interspersed with – the stack we're transferring. Such larger disks can only be beneath disk n, or on the other two posts.
- If there are disks larger than the n smallest disks – i.e. if there are more than n disks – their current positions have no effect on the sequence of moves needed to transfer the n smallest. Any of the n smallest disks can be stacked – directly or indirectly – on top of disks larger than disk n, without violating the rules.
- The number of moves required for the transfer is dependent only on n, and not on the starting or destination post.

Visualization and Analysis

One of the first things we should notice when exploring this puzzle is that solving it for some number of disks (greater than 1) always requires solving it for the next smaller number of disks along the way. For example, solving it for 2 disks requires solving it for 1 disk – twice, in fact.

Let's make sure that we understand these smaller, embedded problems very well, because they're the key to solving the larger problem. First, we need to introduce some new notation: Let's write the move of a disk from one post to another using the right arrow symbol, ➔. For example, if our posts are numbered 1, 2, and 3, then a move from post 1 to post 2 is written as $1 \rightarrow 2$. Also, let's refer to the sequence of moves required to transfer a stack of n disks from some post A (which may be 1, 2, or 3) to another post B as $\boldsymbol{S}(n, A, B)$. For example, we'll write the sequence of moves required to transfer a stack of 3 disks from post 1 to post 3 as $\boldsymbol{S}(3,1,3)$.

With this notation, let's look at $\boldsymbol{S}(2,1,2)$ – i.e. the sequence of moves required to transfer a stack of 2 disks from post 1 to post 2:

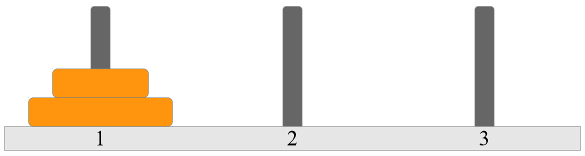
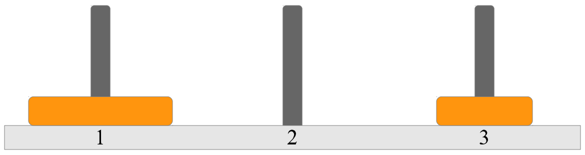
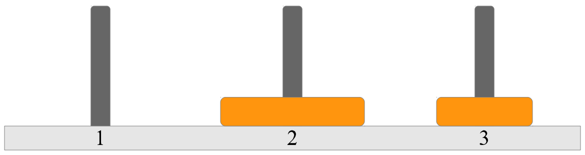
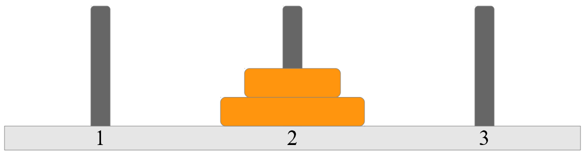
#	Move	Position After Move
0		
1	$1 \rightarrow 3$	
2	$1 \rightarrow 2$	
3	$3 \rightarrow 2$	

Table 1: Tower of Hanoi solution $\boldsymbol{S}(2, 1, 2)$

Can you find a shorter sequence? Some thought and experimentation should convince us that 3 moves is the minimum required for transferring a stack of 2 disks from one post to another.

A solution for 1 disk may seem trivial, but it's actually important to notice that move #1 in the table is the same sequence we would use for a 1-disk solution from post 1 to post 3 – that is, $\boldsymbol{S}(1,1,3)$ – and move #3 is another 1-disk solution, $\boldsymbol{S}(1,3,2)$.

Now let's look at the problem with 3 disks, and see if we notice a similar pattern. This time, let's work out the solution for transferring those disks from post 1 to post 3, or $\boldsymbol{S}(3,1,3)$.

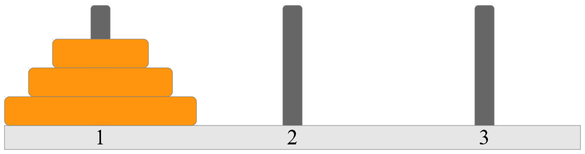
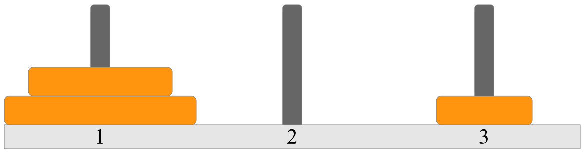
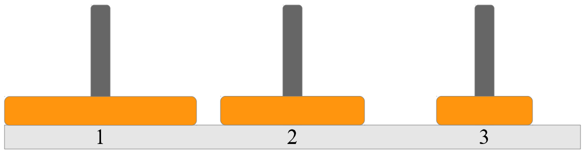
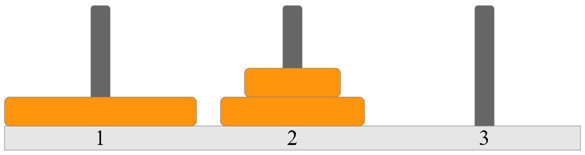
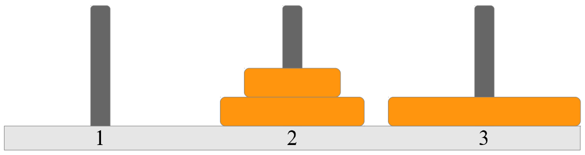
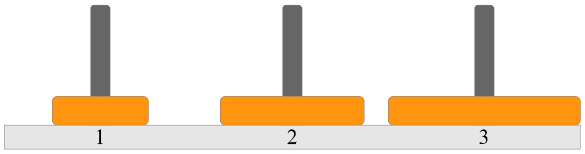
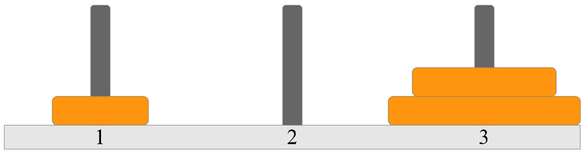
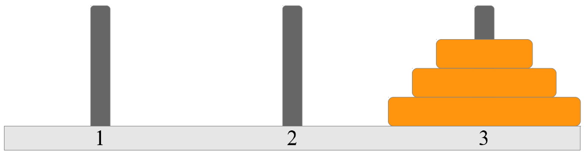
#	Move	Position After Move
0		
1	$1 \rightarrow 3$	
2	$1 \rightarrow 2$	
3	$3 \rightarrow 2$	
4	$1 \rightarrow 3$	
5	$2 \rightarrow 1$	
6	$2 \rightarrow 3$	
7	$1 \rightarrow 3$	

Table 2: Tower of Hanoi solution ***S***(3, 1, 3)

A quick inspection shows us that moves #1-3 in Table 2 are the same as the moves making up ***S***(2, 1, 2) in Table 1. Further, moves #5-7 are the moves required to move a stack of 2 disks from post 2 to post 3 – i.e. ***S***(2, 2, 3).

When we think about it, this makes perfect sense. In order to move a stack from a starting post to a destination post, at some point we have to move the largest disk in the stack to that destination post, and then stack the smaller disks on top of it. Working from this, we observe the following:

1. The only way to move the largest of *n* disks to its destination (assuming $n > 1$) is first to move all of the smaller disks to a stack on the post that is neither the starting nor the destination post.
2. We can then move the disk *n* to its destination post.
3. Finally, we can transfer the stack of smaller disks from its current post to the destination post, stacking them on top of disk *n*.

Step 1 includes all of the moves to solve the next smaller version of the puzzle – i.e. for $(n - 1)$ disks. Step 2 is a move of a single disk. Step 3, like step 1, includes the moves to solve the puzzle for $(n - 1)$.

We now have enough understanding of the problem to create an algorithm for finding the *sequence* of moves required to transfer *n* disks, and to create a formula for computing the *number* of moves required to transfer *n* disks.

Algorithm

In some ways, this is a very simple algorithm to describe, but we do need some new notation. First, we'll define our algorithm as if we were defining a function in an implementation. We can name our algorithm **solve**, and construct it to return a sequence of moves: $\textbf{solve}(n, A, B)$ will return the sequence of moves required to transfer *n* disks from post *A* to post *B*. To describe the algorithm, we'll need the following symbols:

$\{\ldots\}$	Encloses elements of a set. For example, the set of post numbers is $\{1, 2, 3\}$.
$\in$	"is in", "is an element of" – denotes membership in a set.
$\rightarrow$	Denotes a move from one post to another – e.g. $1 \rightarrow 2$ represents the move of a single disk from post 1 to post 2.
$[\ldots]$	Encloses a sequence of moves. For example, $[1 \rightarrow 2]$ is a sequence made up of a single move (from post 1 to post 2), and $[1 \rightarrow 2,\ 1 \rightarrow 3,\ 2 \rightarrow 3]$ is a sequence of three moves. $[]$ is an empty sequence – i.e. one with zero moves.
$\circ$	Returns the concatenation (combination) of one sequence of moves with another. The value of $S_1 \circ S_2$ is a sequence containing all of the moves from S_1 (in order), followed by all of the moves from S_2 (in order). Concatenation with an empty sequence has no effect:

$$\begin{aligned} S \circ [] &= S \\ [] \circ S &= S \\ [] \circ [] &= [] \end{aligned}$$

```
To solve$(n, A, B)$:
  • Where:
      ▫ $n$ is the number of disks to be transferred, $n \in \{0, 1, 2, \ldots\}$.
      ▫ $A$ is the starting post, $A \in \{1, 2, 3\}$.
      ▫ $B$ is the destination post, $B \in \{1, 2, 3\}$, $B \neq A$.
      ▫ $C$ is the other (non-starting, non-destination) post, $C \in \{1, 2, 3\}$, $C \neq A$, $C \neq B$.
      ▫ It is assumed that disks 1, 2, …, $n$ are initially stacked on post $A$; disks larger than
        disk $n$ might be stacked on one or more posts, but we can safely ignore them.
      ▫ The return value is the sequence of moves required for the transfer.
  • If $n < 1$:
      ▫ Return $[\,]$.
    Otherwise:
      ▫ Return solve$(n-1, A, C) \circ [A \rightarrow B] \circ$ solve$(n-1, C, B)$.
```
Algorithm 3: Tower of Hanoi solution sequence

That's it – that's our Tower of Hanoi solution algorithm.

Surely it can't be that simple, can it? Well, let's try **solve**(2, 1, 2). First, our algorithm tests the number of disks, to see if it's less than 1; it's not, so the algorithm returns the value of $\textbf{solve}(1, 1, 3) \circ [1 \rightarrow 2] \circ \textbf{solve}(1, 3, 2)$. In other words, in order to execute the algorithm to produce the sequence of moves for transferring 2 disks from post 1 to post 2, we execute the algorithm for transferring 1 disk from post 1 to post 3, then append to that result a single move from post 1 to post 2, and finally append the result of executing the algorithm for transferring 1 disks from post 3 to post 2. Extending this logic to its conclusion, we have

$$\begin{aligned}
\textbf{solve}(2, 1, 2) &= \textbf{solve}(1, 1, 3) \circ [1 \rightarrow 2] \circ \textbf{solve}(1, 3, 2) \\
&= \textbf{solve}(0, 1, 2) \circ [1 \rightarrow 3] \circ \textbf{solve}(0, 2, 3) \\
&\quad \circ [1 \rightarrow 2] \\
&\quad \circ \textbf{solve}(0, 3, 1) \circ [3 \rightarrow 2] \circ \textbf{solve}(0, 1, 2) \\
&= [\,] \circ [1 \rightarrow 3] \circ [\,] \circ [1 \rightarrow 2] \circ [\,] \circ [3 \rightarrow 2] \circ [\,] \\
&= [1 \rightarrow 3, 1 \rightarrow 2, 3 \rightarrow 2]
\end{aligned}$$

This is exactly the sequence shown in Table 1.

When an algorithm is defined in terms of itself, as this one is, we call it a *recursive* algorithm. Many useful and important algorithms can be described in recursive terms (for example, there's a recursive form of Euclid's algorithm); sometimes it's the simplest way to describe them.

Can the same approach help us answer question 2 – i.e. can it help us compute the number of moves required for 64 disks?

Let's use M_n to refer to the number of moves required to transfer a stack of *n* disks. We've observed that in order to transfer *n* disks, we need to transfer (*n* – 1) disks, then transfer a single disk, then transfer (*n* – 1) disks again. Therefore,

$$\begin{aligned} M_n &= M_{n-1} + 1 + M_{n-1} \\ &= 2M_{n-1} + 1 \end{aligned} \tag{11}$$

Once again, we see a recursive definition: M_n is defined in terms of M_{n-1}. When recursion is used in a formula relating the value of an item in a mathematical sequence to previous values in the same sequence, we call it a *recurrence relation*.

An equivalent way of expressing (11), which will come in handy below, is

$$M_{n+1} = 2M_n + 1. \tag{12}$$

We know that it takes 1 move to transfer a stack consisting of a single disk; that is,

$$M_1 = 1. \tag{13}$$

From (12) and (13), we can generate the sequence *M* consisting of the number of moves required for 1, 2, 3, etc. disks.

$$M = 1, 3, 7, 15, \ldots \tag{14}$$

We could easily extend sequence (14) to find the number of moves for 64 disks. But maybe there's a formula for M_n that doesn't require computing all the intermediate values. Since each term in *M* doubles the previous term, and then adds 1, let's see if we can express M_n using powers of 2. First, we notice that $M_1 = 1 = 2^1 - 1$; we also see that $M_2 = 3 = 2^2 - 1$, and $M_3 = 7 = 2^3 - 1$. Is this just coincidence, or are we on to something?

For the moment, let's assume that

$$M_n = 2^n - 1. \tag{15}$$

Does (15) fit the recurrence relation shown in (12)? To answer that, we can substitute our tentative expression for M_n from (15) into (12), to compute M_{n+1}:

$$\begin{aligned} M_{n+1} &= 2M_n + 1 \\ &= 2\left(2^n - 1\right) + 1 \\ &= 2^{n+1} - 2 + 1 \\ &= 2^{n+1} - 1 \end{aligned}$$

The result is simply (15), expressed in terms of (n + 1). Though a discussion of *mathematical induction* is beyond the scope of this lesson, we've just used it to prove that $M_n = 2^n - 1$, by showing that (15) holds for M_1 and satisfies the recurrence relation (12).

Implementation

So now we have an algorithm for $\textbf{Solve}(n, A, B)$, and we have a formula for M_n. Are we ready to implement them both? Almost, but there's a detail we've glossed over: given A and B (the numbers of the starting and destination posts), how will we compute C? Arguably the simplest method is to take the known sum of A, B, and C (6), and subtract A and B to obtain C; this is what we'll do in our code.

For this problem, Python's features make implementation in that language much easier than in Java, so that's the language we'll use.

This time, we need to implement two functions, along with a main script that will invoke them. (In the accompanying materials, **`Python/tower.py`** can be used as a starting point; it already has the main script completed, and stubs for the two functions.)

```python
def solve(num_disks, orig, dest):
    stage = 6 - orig - dest
    return [] if num_disks < 1 else solve(num_disks - 1, orig, stage) \
                                 + [[orig, dest]] \
                                 + solve(num_disks - 1, stage, dest)


def num_moves(num_disks):
    return (1 << num_disks) - 1;


if __name__ == '__main__':
    move_disks = 5
    move_orig = 1
    move_dest = 2
    move_sequence = solve(move_disks, move_orig, move_dest)
    count_disks = 64
    count = num_moves(count_disks)
    print("Sequence to transfer {0} disks between posts {1} and {2} = {3}"
          .format(move_disks, move_orig, move_dest, move_sequence))
    print("Number of moves required for {0} disks = {1}"
          .format(count_disks, count))
```

Listing 3: Python implementation of Tower of Hanoi algorithm with invocation

Like the algorithm itself, the implementation is very simple; however, there are still some important details to note about the Python code we're using:

- Several of the native data types in Python are *sequences* – ordered collections of items. Text strings, for example, are sequences of characters. One type of sequence of particular interest to us here is the *list*. A list can contain items of virtually any type; in this case, we're representing a move from one post to another as a list of two numbers. A sequence of moves are represented as a list of moves.

In lines 3 and 4 of the code, we use the Python syntax of enclosing items in a list with square brackets. So **`[orig, dest]`** (in line 4) is a list representing a move from **`orig`** to **`dest`**. When we put an additional set of square brackets around this list, **`[[orig, dest]]`**, the result is a sequence of moves consisting of a single move – i.e. the move of the **`num_disks`**th disk from one post to another, after moving all of the smaller disks off it, and before re-stacking all of those smaller disks on top of it.

- The recursion in our implementation looks very much like it did in the algorithm description. In particular, the implementation of the **`solve`** function includes two invocations of the same function (on lines 3 and 5), just as the definition of **Solve** in the algorithm description includes two invocations of **Solve**.
- One notable difference between the algorithm and its implementation is seen in the use of a conditional expression (lines 3-7). In Python, the conditional expression takes the form ***`value1`*** **`if`** ***`condition`*** **`else`** ***`value2`***. Similar to the Java conditional expression, the computed value of the expression is ***`value1`*** if ***`condition`*** is **`True`** (after coercion to a Boolean value if necessary), and ***`value2`*** otherwise. Essentially, what we're doing here is using a single **`return`** statement with a conditionally computed value, rather than using a conditional (**`if`**) statement to execute one of two **`return`** statements. (These are equivalent, but the latter would follow the algorithm more literally.)
- As noted previously, Python simple statements usually terminate at the end of the physical line. However, line continuation can be used to extend a statement over multiple lines. In lines 3-5, the **`\`** character tells Python that the statement on the current line is continued to the next. Python treats that next line as an exception to the indentation rules: we can indent it in any way that makes sense to us. Here, we're indenting it so that the recursive calls to the **`solve`** function line up neatly.

  An implicit form of line continuation is used in lines 19-20 and 21-22. If there are parentheses or brackets that are not yet closed when a line ends, Python assumes that the statement continues to the next line.
- An efficient way to compute powers of 2 is shown in line 9. The **`<<`** operator shifts the base-2 form of a number to the left by adding zeros to the right of the number; the effect of this is to multiply the number by a power of 2. For example, the value of **`1 << 1`** is 10_2, or $1 \cdot 2^1 + 0 \cdot 2^0$, or 2; the value of **`1 << 2`** is 100_2, or $1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$, or 4; and the value of **`1 << disks`** is 2^{disks}. (This is analogous to multiplying a base-10 integer by a power of 10 by adding zeros to the right of the number.)

Answer

This time, the output should look something like this:

```
Sequence to transfer 5 disks between posts 1 and 2 = [[1, 2], [1, 3],
[2, 3], [1, 2], [3, 1], [3, 2], [1, 2], [1, 3], [2, 3], [2, 1], [3, 1],
[2, 3], [1, 2], [1, 3], [2, 3], [1, 2], [3, 1], [3, 2], [1, 2], [3, 1],
[2, 3], [2, 1], [3, 1], [3, 2], [1, 2], [1, 3], [2, 3], [1, 2], [3, 1],
[3, 2], [1, 2]]
Number of moves required for 64 disks = 18446744073709551615
```

Output 3: Tower of Hanoi solution for 5 disks and number of moves required for 64 disks

It looks like the Brahmin priests in the legend have a lot of work ahead of them.

Enhancements and Alternatives

Unlike Python, Java doesn't have an intrinsic list data type; arrays and strings are the only complex data types with direct language support in Java – and arrays can't be re-sized dynamically, which makes their use in a Tower of Hanoi implementation tricky (though not impossible). However, the **`java.util`** package of the standard Java library includes an extensive set of dynamically re-sizable collection classes – including lists, sets, queues, stacks, vectors, and hash tables. Implementing the Tower of Hanoi algorithm in Java would be a good exercise in learning how to use some of these types of collections.

Another area where a Tower of Hanoi implementation in Java would need extra work is in the handling of very large numbers – at least if we intend to compute the number of moves required for 64 or more disks, as we did with the Python implementation.

The standard integer type in both Python and Java supports values in the range $\left[-2^{31}, 2^{31}-1\right]$. As we've found, the number of moves required for 64 disks is $2^{64}-1$, which is far outside that range. Fortunately, Python has a long integer data type that permits computations with much higher and lower integer values (practically without limit), with no explicit conversion required; in other words, Python uses the standard integer type when it can, and switches automatically to the long integer type when necessary.

Java also has an intrinsic **`long`** integer type – but that type is limited to the range $\left[-2^{63}, 2^{63}-1\right]$, which doesn't include the value we had to compute for this problem. Are computations at that scale possible in Java? Absolutely – but they aren't as simple in Java as they are in Python. The **`java.math`** package of the standard Java library includes the **`BigInteger`** class, which has the same lack of practical limits as long integers in Python. However, **`BigInteger`** is a library class, rather than an intrinsic data type. The strong typing of Java doesn't permit us to treat **`BigInteger`** objects as if they were intrinsic integers, so we have to do any necessary conversions explicitly, and use methods of the **`BigInteger`** class for computations on objects of that type.

Historical Note

The numbers in sequence (14) – defined as $M_n = 2^n - 1$, where *n* is a positive integer – are called *Mersenne numbers*. Mersenne numbers are named for Marin Mersenne, a French monk of the 16th and 17th centuries [11].

Some Mersenne numbers are also prime; these are called *Mersenne primes*.4 If *n* is prime, then M_n may be (but isn't necessarily) prime, as well; if *n* isn't prime, then M_n cannot be prime. For clarity, Mersenne numbers with prime values of *n* are usually written as M_p, where *p* denotes a prime number.

Actually, very few Mersenne numbers are prime (49 have been found so far [12]); nonetheless, the fact that M_p is so much larger than *p* helps make M_p an attractive candidate in the search for ever-larger primes.5 Making M_p even more attractive is the availability of specialized primality tests for M_p candidates – tests that are more efficient than general primality tests. The most commonly used of these specialized tests is the Lucas-Lehmer primality test [13], originally developed in 1856 by none other than Édouard Lucas, the inventor of the Tower of Hanoi puzzle.

4 Besides their theoretical appeal, Mersenne primes have practical uses as well, including applications in pseudorandom number generators such as the Mersenne twister (the default random number generator in MATLAB, Python, and NetLogo).

5 As of March 2017, 11 of the 12 largest known primes are Mersenne primes.

References

[1] D. E. Knuth, “Basic Concepts,” in *The Art of Computer Programming*, Volume 1, 3rd ed. Boston: Addison-Wesley, 1997, ch. 1, sec. 1, pp. 4-5.

[2] Python Software Foundation. (2017, Jan. 17). *Python* [Online]. Available: http://www.python.org [Mar. 2, 2017].

[3] Oracle Corporation. (2017, Feb. 28). *Java* [Online]. Available: http://www.oracle.com/technetwork/java/ [Mar. 2, 2017].

[4] U. Wilensky. (2016). *NetLogo* [Online]. Available: http://ccl.northwestern.edu/netlogo/ [Mar. 2, 2017].

[5] K. Finley. (2012, Jun. 5). “5 Ways to Tell Which Programming Languages are Most Popular,” *readwrite* [Online]. Available: http://readwrite.com/2012/06/05/5-ways-to-tell-which-programming-lanugages-are-most-popular/ [Mar. 2, 2017].

[6] Euclid (D.E. Joyce, ed.). (c. 300 BCE, ed. 1997). *Elements* [Online]. Available: http://aleph0.clarku.edu/~djoyce/java/elements/toc.html [Mar. 2, 2017].

[7] Wikimedia Foundation. (2017, Feb. 23). “Bizz buzz,” *Wikipedia* [Online]. Available: http://en.wikipedia.org/wiki/Bizz_buzz [Mar. 2, 2017].

[8] I. Ghory. (2007, Jan. 24). “Using FizzBuzz to Find Developers who Grok Coding,” *Imran on Tech* [Online]. Available: http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/ [Mar. 2, 2017].

[9] É. Lucas. “Deuxième Récréation: Le Calcul Et Les Machines A Calculer,” in *Récréations mathématiques*, Vol. 3. Paris: Gauthier-Villars, 1893, pp. 55-59.

[10] H. E. Dudeney. “The Canterbury Puzzles,” in *The Canterbury Puzzles and Other Curious Problems*, 2nd ed. London: Thomas Nelson and Sons, 1919, pp. 24-25.

[11] J.J. O'Conner, E.F. Robertson. (2005, Aug.) “Marin Mersenne,” *MacTutor History of Mathematics archive* [Online]. Available: http://www-history.mcs.st-and.ac.uk/Biographies/Mersenne.html [Mar. 2, 2017].

[12] Mersenne Research, Inc. (2016, Sep. 2). “List of Known Mersenne Prime Numbers,” *The Great Internet Prime Search* [Online]. Available: https://www.mersenne.org/primes [Mar. 2, 2017].

[13] Wikimedia Foundation. (2017, Feb. 27). “Lucas-Lehmer primality test,” *Wikipedia* [Online]. Available: http://en.wikipedia.org/wiki/Lucas%E2%80%93Lehmer_primality_test [Mar. 2, 2017].