Implementing an Algorithm for Boomerang Fraction Sequences in Python

Introduction

We've all encountered maze problems, where the challenge is to find a path through a labyrinth from a starting point to an ending point. Have you ever seen a maze in which the reverse path, from the end of the maze to the start, was easier to find than the normal (forward) path? Let's see if a similar strategy (one which we can then implement in Python) helps us with boomerang fractions. More specifically, let's use such a strategy as part of an approach to find boomerang fraction sequences that return to 1.

Let's start with the sequence for the fraction $(n-1)/n$, and see if we can use it to help us construct a method that works for some other fractions, as well.

Questions for discussion

Let's consider a few questions about boomerang fractions in general, and about the fraction $(n-1)/n$ in particular.

1. Do some of the boomerang fraction sequences you've found include multiple additions, one after the other? Do some include multiple inversions, one after the other? What would be effect of each of these two types of repeated operations?
2. Will any of the terms of a boomerang fraction sequence ever have a negative value? Will any of the terms be zero?
3. Just as we know that the first operation in a boomerang fraction sequence is always addition, what do we know (if anything) about the *last* operation in the sequence for $(n-1)/n$? Is this the case just for that fraction, or some others as well?
4. For the fraction $(n-1)/n$, do we also know something about the second-to-last operation? Can we be certain about the nature of that operation – and if so, what gives us this certainly? Is this the case for some other fractions, or just $(n-1)/n$?
5. If we traverse a boomerang fraction sequence in reverse order, from right to left, what operations would we perform to produce each term in the reversed sequence?

Nick Bennett & James Taylor
Math Circles Collaborative of New Mexico 6 January, 2017

Observations and preliminary conclusions

With some reflection, we might notice the following:

- The operations in the sequence will never include two inversions in a row – more accurately, performing two inversions in a row has the effect of returning to the value that we had before the two inversions. So let's assume that whatever strategy we use, it won't include two inversions in a row. Another way of saying this is that each inversion will be followed by one or more additions.
- If a boomerang fraction sequence returns to 1, we know that the last operation before the value returns to 1 must be addition; otherwise, the next-to-last term would also be 1. This is the case regardless of the fraction used.
- For the fraction $(n-1)/n$, if the last term is 1, then the second to last term is $1-(n-1)/n$, or $1/n$. From this, we can see that the second-to-last operation must be inversion; otherwise, the next preceding term would be negative. In fact, this will be the case for any fraction m/n, where $2m \geq n$.

  Thus, the sequence for $(n-1)/n$ will end with the terms

  $$\ldots \rightarrow n \circlearrowleft \frac{1}{n} \rightarrow 1.$$

  (Note that we're representing addition by $\rightarrow$, and inversion by $\circlearrowleft$. This symbol for inversion is intentionally different from the one used in the Math Circle activity; the reason for this will become clear later.)

  In fact, we can now see clearly one possible trajectory of this boomerang fraction, from start to finish:

  $$1 \rightarrow \left[1+\frac{(n-1)}{n}\right] \rightarrow \left[1+2\frac{(n-1)}{n}\right] \rightarrow \ldots \rightarrow \left[n-\frac{(n-1)}{n}\right] \rightarrow n \circlearrowleft \frac{1}{n} \rightarrow 1 \qquad (1)$$

  If we go from right to left, starting with the final value and moving to the initial value, then each $\rightarrow$ represents a subtraction, and each $\circlearrowleft$ is still an inversion.
- Since we now know that the first and last operations in a boomerang fraction sequence that returns to 1 are always addition operations, and any inversions will not be immediately repeated, we can break down the sequence of operations into 2 or more sub-sequences of additions, with single inversions between those sub-sequences.
- Similarly, the operations in the reverse sequence consist of 2 or more sub-sequences of subtractions, separated by single inversions.

- Further, since we know that a boomerang fraction sequence never has a negative or zero value, we have 2 natural stopping conditions for a sub-sequence of subtractions, if we are constructing the sequence in reverse order:
    - If a subtraction produces a value of 1, we've reached the end of the reverse-order sequence (i.e. the start of the original sequence), and we're done.
    - Otherwise, if another subtraction would result in a negative or zero result, no further subtractions are possible; we must perform an inversion, and start subtracting again.

  If we start with the value 1, moving from right to left, repeatedly subtract the fraction $(n-1)/n$, and applying the above conditions for performing an inversion and terminating the sequence, sequence (1) is the result. Will this approach work for other fractions as well? Let's try and see!

Expressing the solution approach as an algorithm

Restate the logic of the last few points as an *algorithm* (an explicit procedure that can be used to solve a general problem) that will construct – in reverse order – a boomerang fraction sequence starting and ending with 1 (assuming such a sequence is possible). For now, don't worry about whether the result is the shortest possible sequence. Describe all the steps clearly, with as little ambiguity as possible; using mathematical notation can often help produce this clarity and reduce ambiguity.

Check your algorithm by having someone else read it. (For best results, your reader should first read and understand the introduction to the boomerang fractions activity.) Is it clear to them? If necessary, modify your algorithm to make it easier for your reader to understand.

Using pencil and paper, test your algorithm by using it with the following fractions:

1. $1/2$
2. $2/3$
3. $1/4$
4. $4/5$
5. $2/5$

One possible algorithm

As you might imagine, there are multiple ways to express the algorithm we've been discussing; one approach is shown here. (Note that for consistency with normal mathematical conventions, this algorithm constructs 2 sequences: a sequence of values, and a separate sequence of operations.)

Hopefully, most of the mathematical notation used here is either familiar to you already, or reasonably self-explanatory. One possible exception is the $\circ$ symbol, used here to denote concatenation of 2 sequences into a single sequence.

```
1. Let $m$ and $n$ be the specified numerator and denominator (respectively), where $m, n \in \mathbb{N}$ and $m < n$.
2. Let $f = m/n$.
3. Let $v$ be the current value in the sequence. Initially, since we're starting at the end of the sequence, $v = 1$ (the target value for the end of the sequence).
4. Let $\boldsymbol{V}$ be the sequence of values. Initially, $\boldsymbol{V} = \{v\}$. That is, the sequence initially contains the current value (the target value for the end of the sequence).
5. Let $\boldsymbol{O}$ be the sequence of operations. Initially $\boldsymbol{O} = \{\}$ (no operations).
6. Repeat …
    a. Let $v = v - f$.
    b. Let $\boldsymbol{V} = \{v\} \circ \boldsymbol{V}$.
    c. Let $\boldsymbol{O} = \{\rightarrow\} \circ \boldsymbol{O}$.
    d. If $v \leq f$,
        i. Let $v = \frac{1}{v}$.
        ii. Let $\boldsymbol{V} = \{v\} \circ \boldsymbol{V}$. That is, insert $v$ at the start of $\boldsymbol{V}$.
        iii. Let $\boldsymbol{O} = \{\circlearrowright\} \circ \boldsymbol{O}$.
   … until $v = 1$.
```

Questions on the algorithm

1. Did your algorithm (or the one shown above) work for the test fractions that aren’t of the form $(n-1)/n$?
2. Did applying your algorithm to any of the test fractions result in a sequence that you noticed isn’t the shortest possible sequence for that fraction? Can you think of a way to modify your algorithm so that it produces a shorter sequence in those cases (and still works for the others)? It might help to think about how you might determine that a single inversion of the current value, followed by one or more subtractions, would result in the value 1.
3. Although the algorithm should work for all of the fractions on the previous list, there are others – for example, $7/9$ – for which the algorithm doesn’t seem to work. These might be values for which the sequence can never return to 1 – or it might be that this algorithm just isn’t effective for some fractions. So far, the only stopping condition for the algorithm occurs when a value of 1 is reached. What other stopping condition(s) should be added, to indicate that the algorithm will probably not succeed for the specified fraction?

Implementing the algorithm in Python

Before we make any changes to the algorithm, let's implement it in Python. Of course, there are usually multiple ways to express an algorithm, whatever the programming language. In any event, it may be helpful to keep the following points in mind:

1. Python does not have a *repeat…until* iteration construct, as such; it does, however, have a **`while`** iteration statement. (What is the difference?) For algorithms requiring the former, it's common to use the following in Python:

   ```
   while True:
       …
       if stop_condition:
           break
   ```

2. A Python list (a mutable sequence) can be constructed from back to front, by inserting new elements at the beginning of the list.

3. Multiple elements can be inserted into (or appended to the end of) an existing list by forming a second list containing just the new elements, and concatenating the 2 lists with the **`+`** operator.

4. Rather than dealing with potential numerical issues associated with decimal approximations to fractions, we can make use of the **`Fraction`** class in the **`fractions`** module, included in the standard Python library starting with version 2.6.

5. Python lists can be heterogeneous – that is, a list can contain elements of multiple types. While this can lead to confusion in some cases, it can be useful if we want to keep the operations and values of our sequence in a single list. (The operations might be expressed as strings, as references to the relevant functions in the **`operator`** module, etc.)

Write a Python implementation of your algorithm, or of the algorithm articulated in "One possible algorithm" (page 4). Debug and test the implementation, using the test values on page 3.

One possible implementation

Note that this implementation (tested in Python 3.5.x) includes some elements that are not actually part of the algorithm described on page 4, but are used for presenting the output in a (more or less) human-readable form. These include the **`ADDITION_OPERATION`** and **`INVERSION_OPERATION`** characters (which may not appear correctly on some systems), used to represent the respective operations; and the **`OUTPUT_TEMPLATE`** and **`SEQUENCE_DELIMITER`** formatting strings, used to concatenate the elements of the list produced by the algorithm into a familiar form.

Unlike the algorithm previously described, this implementation includes both the sequence values and the operations used to produce those values in a single list.

Finally, note that the code includes some docstrings and inline comments, intended to make clear the invocation and use of the **`boomerang`** function (which implements the algorithm), and a critical piece of logic in the algorithm.

```python
from fractions import Fraction

ADDITION_OPERATION = "\u2192"
INVERSION_OPERATION = "\u21bb"
OUTPUT_TEMPLATE = "{}/{}\n{}\n"
SEQUENCE_DELIMITER = " "

def boomerang(fraction):
    """Construct and return a boomerang fraction sequence.

    Arguments:
        fraction -- Additive term in boomerang fraction sequence.
    Return value:
        Sequence of interleaved values and operations.
    """
    value = Fraction(1)
    sequence = [value]
    while True:
        value -= fraction
        sequence = [value, ADDITION_OPERATION] + sequence
        if value == 1:
            break
        if value <= fraction:  # Subtraction not possible; invert.
            value = 1 / value
            sequence = [value, INVERSION_OPERATION] + sequence
    return sequence

if __name__ == '__main__':
    for m, n in [(1, 2), (2, 3), (1, 4), (4, 5), (2, 5)]:
        sequence = boomerang(Fraction(m, n))
        print(OUTPUT_TEMPLATE.format(m, n,
            SEQUENCE_DELIMITER.join(str(s) for s in sequence)))
```

Saving this code in a .py file, and then executing that file in Python 3.5.x, produces the following output:

```
1/2
1 → 3/2 → 2 ↻ 1/2 → 1

2/3
1 → 5/3 → 7/3 → 3 ↻ 1/3 → 1

1/4
1 → 5/4 → 3/2 → 7/4 → 2 → 9/4 → 5/2 → 11/4 → 3 → 13/4 → 7/2 → 15/4 →
4 ↻ 1/4 → 1/2 → 3/4 → 1

4/5
1 → 9/5 → 13/5 → 17/5 → 21/5 → 5 ↻ 1/5 → 1

2/5
1 → 7/5 → 9/5 → 11/5 → 13/5 → 3 → 17/5 → 19/5 → 21/5 → 23/5 → 5 ↻
1/5 → 3/5 → 1
```

Possible enhancements to the algorithm

To address questions 2 and 3 on page 5, we might augment our algorithm as follows. (Again, there are multiple approaches; this is just one.) For this updated algorithm, we need to introduce a new symbol: $|$, a vertical bar, used to denote divisibility. For example, the expression $a|b$ (read "a divides b") means that b is evenly divisible by a, with no remainder.

```
1. Let $m$ and $n$ be the specified numerator and denominator (respectively), where $m, n \in \mathbb{N}$
   and $m < n$.
2. Let $f = m/n$.
3. Let $L$ be the specified limit on the number of inversion operations performed before we
   conclude that the algorithm probably won't work for the given $f$.
4. Let $v$ be the current value in the sequence. Initially, $v = 1$.
5. Let $\boldsymbol{V}$ be the sequence of values. Initially, $\boldsymbol{V} = \{v\}$.
6. Let $\boldsymbol{O}$ be the sequence of operations. Initially $\boldsymbol{O} = \{\}$.
7. Let $k$ be the count of inversion operations performed so far. Initially, $k = 0$.
8. Repeat …
    a. Let $v = v - f$.
    b. Let $\boldsymbol{V} = \{v\} \circ \boldsymbol{V}$.
    c. Let $\boldsymbol{O} = \{\rightarrow\} \circ \boldsymbol{O}$.
    d. If $(v \le f)$ or $\left[(v < 1) \text{ and } f \,\middle|\, \left(\frac{1}{v} - 1\right)\right]$,
        i. If $k = L$,
            • Stop; the algorithm has failed to construct a sequence starting and
              ending with 1, for the given $f$.
        ii. Let $v = \frac{1}{v}$.
        iii. Let $\boldsymbol{V} = \{v\} \circ \boldsymbol{V}$.
        iv. Let $\boldsymbol{O} = \{\circlearrowright\} \circ \boldsymbol{O}$.
        v. Let $k = k + 1$.
… until $v = 1$.
```

Extending the implementation

If you have time (and interest), modify your implementation – to incorporate the changes described in the updated algorithm on page 9, to apply a different approach to address the same questions, or to address other issues you may have noticed.

Debug and test your updated implementation, this time including 7/9 in the test fractions. Does your implementation produce any shorter sequences for some of the test fraction? What does it produce for the fraction 7/9?