Project Euler with SIAM

at ASU WEST — lead by David Pratt, November 6, 2019

Introduction

In this session, we will continue to use Project Euler challenges to practice building and improving algorithms. In particular, we will develop an algorithm to generate Fibonacci numbers, recognize its flaws, and improve upon the algorithm to decrease its runtime from multiple lifespans of the universe to 2ms.

The goal of this session is to let the participants do most of the coding, ergo walk-throughs for problems #25 and #57 are withheld. All of what is needed to program the solutions to these two problems is given in the following walk-through.

Project Euler #2

Even Fibonacci Numbers *Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms are 1, 2, 3, 5, 8, 13, 21, 34, 55, and 89. By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.*

You may already be familiar with the Fibonacci sequence, in which case you know that the sequence is defined recursively by[1]

$$F_1 = 1,\ F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

for $n > 2$. Let's write a function in Python that gives us the n^{th} Fibonacci number. We can transcribe the above definition into code almost exactly as it is written.

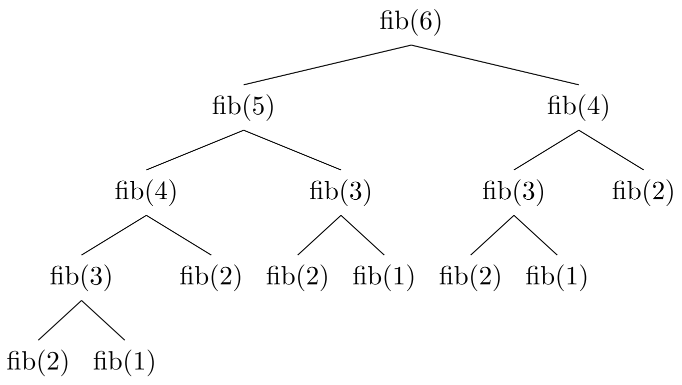
```
1 def fib(n):
2     if n in (1, 2):
3         return 1
4     return fib(n-1) + fib(n-2)
```

If you are new to programming, you may want to take a moment to fully understand each line of the function. The `if` statement in line 2 could be written as `if n == 1 or n == 2`, but you may find the form written in the listing to

[1]Typically, the sequence is defined with $F_0 = 0$ and $F_1 = 1$, but we will adopt Project Euler's definition (from Problem 25) as it makes no difference.

be useful if you have many disjuncts[2]. On line 4, take note that the function is calling itself recursively. The recursive calls necessitate the initial `if` statement, commonly called the base case (what would happen if there was no base case?). Mathematically, the recursive behavior is natural, but the devil is in the ~~details~~ computations.

Let's run through the calculation of fib(6) to better understand our function. Below is a tree that represents the call to fib(6). First, fib(6) calls fib(5) and fib(4), both of which again call the function numerous times.



It looks like the calculation of fib(6) takes a considerable amount of work. From a mathematical context, all that must be known to calculate F_6 is just the previous two terms, but from a computational perspective, all previous terms must be known[3]. The keen-eyed readers will notice, though, that *more than all* of the previous terms must be known, as some values are computed more than once, e.g. fib(3).

In the example above, there are 15 total calls to the function. Computing fib(10) takes a total of 109 calls, and fib(15) takes a total of 1219 calls[4]. The function is simply adding numbers, but has a runtime of over one second starting at fib(33) (and a total of 7,049,155 function calls, wow!).

We should strive to improve this algorithm much like we did previously[5] with Project Euler #1. As mentioned above, when calculating fib(6), there are multiple duplicate function calls, e.g. fib(4) and fib(3). Instead of repeatedly computing these values, let's store them in memory for use whenever they are required again.

[2]An operand of a logical disjunction is called a disjunct.

[3]That is the case in this *particular* algorithm.

[4]If you are curious, find a recursive formula that gives the total number of calls to the function given an input n.

[5]. . . which was a blast, by the way!

The task of programming this feature[6] from scratch is not trivial (if you want to do it correctly, that is). Luckily for us mathematicians, Python provides this feature in its standard library. The `lru_cache` decorator is incredibly useful if you wish to speed up recursive functions[7].

```python
from functools import lru_cache

@lru_cache(maxsize=None)
def fib(n):
    if n in (1, 2):
        return 1
    return fib(n-1) + fib(n-2)
```

With this new and improved function, a call to fib(n) will only run the function n times, as opposed to the monstrosity of a function we had previously. Now, we could answer the original question posed to us incredibly quickly.

But I would be remiss if I let the opportunity pass to share a ubiquitous feature of Python: *generators*. Generator functions are functions that make use of the `yield` keyword as opposed to the `return` keyword. At a high level, generator functions return a *series of values*, one at a time, as opposed to a single value or a list of values.

To illustrate generators, let's first write a normal function that computes the n^{th} Fibonacci number using iteration as opposed to recursion.

```python
def fib(n):
    # set small_fib = 1 and big_fib = 1
    small_fib, big_fib = 1, 1
    # repeat n-1 times
    for _ in range(n-1):
        small_fib, big_fib = big_fib, small_fib + big_fib
    return small_fib
```

This function is nearly equivalent to the function above that makes use of `lru_cache` in the sense that both functions perform the same number of additions. Also take note of the fantastic feature of *simultaneous assignment*[8]!

But the problem requires that we calculate multiple Fibonacci numbers, meaning that if we were to, for example, calculate the first 50 Fibonacci numbers, we would end up using F_{10} only once, but it would be calculated 41 times. Though this is not as extreme as the original function, we can still make improvements.

[6]It is commonly called *memoization* (not *memorization*).

[7]This comes at the cost of storing results in memory. The programmer can set a limit on the number of results stored with the `maxsize` parameter. If a `maxsize` is specified, results will be *forgotten* according to the least-recently-used (LRU) algorithm.

[8]This is an absolute gem of Python, and allows us to avoid creating temporary variables in situations like this. If you are curious, read up on simultaneous assignment and/or iterable unpacking.

Instead of writing a function that `return`s a Fibonacci number, let's write a generator that `yield`s Fibonacci *numbers*. The generator function will look nearly identical to the previously defined function, but with one key difference.

```python
def fib():
    # set small_fib = 1 and big_fib = 1
    small_fib, big_fib = 1, 1
    while True:
        small_fib, big_fib = big_fib, small_fib + big_fib
        yield small_fib
```

This function will generate **all** Fibonacci numbers one at a time, yielding values only when the programmer desires. Below is a tabulation of runtimes and function calls for each algorithm when used to solve this Project Euler problem.

Algorithm	Calls	Runtime
naive	29,860,668	2.59s
lru	34	16.9μs
iterative	34	28.6μs
generator	1	4.74μs

Clearly, there is a *best* algorithm out of these four. Still not convinced that generators are amazing? There are two huge advantages to using generators that can be demonstrated through two examples.

Consider this very Project Euler problem in which you are asked to sum all even Fibonacci numbers less than four million. Instead of calculating Fibonacci numbers individually as with the previous functions (meaning to calculate fib(10) requires that you also calculate all previous Fibonacci numbers), you instead only keep track of the two largest Fibonacci numbers you have calculated so that the next Fibonacci number can be obtained through a simple addition. In short, generators save time.

Consider a problem in which you had to iterate over one million Fibonacci numbers. Do you think it would be more efficient to calculate the first one million Fibonacci numbers before you began to use them, or would it be more efficient to calculate a single Fibonacci number at a time, performing your calculations, then moving on to the next Fibonacci number? In short, generators save space.

Want to learn more about efficient code in Python from a Python core developer? Check out Raymond Hettinger's talk *Transforming Code into Beautiful, Idiomatic Python* available on YouTube[9].

[9]Although the code in the talk is Python 2, the ideas carry over to Python 3. Plus, Raymond Hettinger is a fantastic speaker and this talk contains one of my all-time favorite quotes.

Project Euler #25

1000-digit Fibonacci Number *The 12th term of the Fibonacci sequence, $F_{12} = 144$, is the first term to contain three digits. What is the index of the first term in the Fibonacci sequence to contain 1000 digits?*

If you attempt this problem with the naive Fibonacci algorithm, you will be waiting for an answer for approximately $1.12 \cdot 10^{985}$ years[10]. I have homework due this Friday, so unfortunately I don't have the time. Fortunately, though, I have a Fibonacci generator that will produce the correct result in 2ms.

For this problem, we need a formula that gives the number of digits in a given number. From a programming perspective, we could convert a number to a string, then find the length of the string. But this is lame (and ever so slightly less efficient than using a mathematical formula).

Project Euler #57

Square Root Convergents *It is possible to show that $\sqrt{2}$ can be expressed as an infinite continued fraction,*

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cdots}}}$$

Evaluating the first two iterations, we get

$$1 + \frac{1}{2} = \frac{3}{2} = 1.5$$

and

$$1 + \frac{1}{2 + \frac{1}{2}} = \frac{7}{5} = 1.4.$$

The next five iterations are $17/12$, $41/29$, $99/70$, $239/169$, and $577/408$. But the eighth iteration, $1393/985$ is the first in which the number of digits in the numerator exceeds the number of digits in the denominator. In the first one-thousand iterations, how many fractions contain a numerator with more digits than the denominator?

Dealing with fractions will be incredibly difficult and will require the use of a specialized module capable of working with rational numbers[11]. Ironically, a solution to this problem is only difficult to program if you think exclusively in a programming context. You are a mathematician, and you can greatly reduce the effort required to program a solution.

[10] Bonus points if you figure out how much RAM would be required.

[11] Python provides the `fractions` module in the standard library.