Algorithmic Analysis

It is common for us to try to deduce, "some code is better than other code". It can be but how do we properly analyze the difference and determine which algorithms are better?

In order to answer this question, we need to remember that there is an important difference between a program and the underlying algorithm that the program is representing. An **algorithm** is a generic, step-by-step list of instructions for solving a problem. It is a method for solving any instance of the problem such that given a particular input, the algorithm produces the desired result. A **program**, on the other hand, is an algorithm that has been encoded into some programming language. There may be many programs for the same algorithm, depending on the programmer and the programming language being used.

How do we define "better"?

To explore this difference further, consider the functions shown in Example 1 and 2.

Example 1 solves for the sum of the first *n* integers. The algorithm uses the idea of an accumulator variable that is initialized to 0. The solution then iterates through the *n* integers, adding each to the accumulator.

Example 2 looks different from Example 1. It is doing the same computation but is just poorly coded. I did not use good identifier names to assist with readability, and there is an extra assignment statement during the accumulation step that was not really necessary.

The question we raised before asked which of these functions was better. This really depends on what we define as "better." The first *sumOfN* is better in terms of readability but we want to analyze the algorithm itself. **Algorithm analysis** is concerned with comparing algorithms based upon the amount of computing resources that each algorithm uses. We want to be able to consider two algorithms and say that one is better than the other because it is more efficient in its use of those resources or perhaps because it simply uses fewer. From this perspective, the two functions above seem very similar. They both use

3.1 Example Functions

Example 1

```
def sumOfN(n):
theSum = 0
for i in range(i, n+1):
theSum = theSum + i
return theSum
```

```
print(sumOfN(10))
```

Example 2

```
def ted(rick):
morty = 0
for bill in range(1,rick+
1):
jerry = bill
morty = morty + jerry
return morty
```

```
print(ted(10))
```

essentially the same algorithm to solve the summation problem.

Resource Use - Space and Time

At this point, it is important to think more about what we really mean by computing resources. There are two different ways to look at this. One way is to consider the amount of space or memory an algorithm requires to solve the problem. The amount of space required by a problem solution is typically dictated by the problem instance itself. Every so often, however, there are algorithms that have very specific **space requirements**, and in those cases we will be very careful to explain the variations.

As an alternative to space requirements, we can analyze and compare algorithms based on the amount of time they require to execute. This measure is sometimes referred to as the **"execution time"** or "running time" of the algorithm. One way we can measure the execution time for the function *sumOfN* is to do a benchmark analysis. This means that we will track the actual time required for the program to compute its result. In Python, we can benchmark a function by noting the starting time and ending time with respect to the system we are using. In the *time* module there is a function called *time* that will return the current system clock time in seconds since some arbitrary starting point. By calling this function twice, at the beginning and at the end, and then computing the difference, we can get an exact number of seconds (fractions in most cases) for execution.

However, we would need a better way to characterize these algorithms with respect to execution time. The benchmark technique computes the actual time to execute. It does not really provide us with a useful measurement because it is dependent on a particular machine, program, time of day, compiler, and programming language. Instead, we would like to have a characterization that is independent of the program or computer being used. This measure would then be useful for judging the algorithm alone and could be used to compare algorithms across implementations.

Source: Problem Solving and Algorithms in Python (http://interactivepython.org/runestone/static/pythonds/index.html#) from Bradley Miller on www.interactivepython.org (http://interactivepython.org/runestone/static/pythonds/index.html#).