2.2. What Is Algorithm Analysis?

It is very common for beginning computer science students to compare their programs with one another. You may also have noticed that it is common for computer programs to look very similar, especially the simple ones. An interesting question often arises. When two programs solve the same problem but look different, is one program better than the other?

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. As we stated in Chapter 1, 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.

To explore this difference further, consider the function shown in ActiveCode 1. This function solves a familiar problem, computing 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.

Run | Load History | Show CodeLens

```
1 def sumOfN(n):
2    theSum = 0
3    for i in range(1,n+1):
4        theSum = theSum + i
5
6    return theSum
7
8 print(sumOfN(10))
9
```

ActiveCode: 1 Summation of the First n Integers (active1)

Now look at the function in ActiveCode 2. At first glance it may look strange, but upon further inspection you can see that this function is essentially doing the same thing as the previous one. The reason this is not obvious is poor coding. We did not use good identifier names to assist with readability, and we used an extra assignment statement during the accumulation step that was not really necessary.

Run | Load History | Show CodeLens

ives.html)

```
1 def foo(tom):
2     fred = 0
3     for bill in range(1,tom+1):
4        barney = bill
```

```
      fred = fred + barney

   return fred

print(foo(10))

```

ActiveCode: 2 Another Summation of the First n Integers (active2)

The question we raised earlier asked whether one function is better than another. The answer depends on your criteria. The function `sumOfN` is certainly better than the function `foo` if you are concerned with readability. In fact, you have probably seen many examples of this in your introductory programming course since one of the goals there is to help you write programs that are easy to read and easy to understand. In this course, however, we are also interested in characterizing the algorithm itself. (We certainly hope that you will continue to strive to write readable, understandable code.)

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 essentially the same algorithm to solve the summation problem.

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.

Listing 1

ives.html)

```
import time

def sumOfN2(n):
   start = time.time()

   theSum = 0
   for i in range(1,n+1):
      theSum = theSum + i

   end = time.time()

   return theSum,end-start
```

Listing 1 shows the original `sumOfN` function with the timing calls embedded before and after the summation. The function returns a tuple consisting of the result and the amount of time (in seconds) required for the calculation. If we perform 5 invocations of the function, each computing the sum of the first 10,000 integers, we get the following:

```
>>>for i in range(5):
       print("Sum is %d required %10.7f seconds"%sumOfN(10000))
Sum is 50005000 required  0.0018950 seconds
Sum is 50005000 required  0.0018620 seconds
Sum is 50005000 required  0.0019171 seconds
Sum is 50005000 required  0.0019162 seconds
Sum is 50005000 required  0.0019360 seconds
```

We discover that the time is fairly consistent and it takes on average about 0.0019 seconds to execute that code. What if we run the function adding the first 100,000 integers?

```
>>>for i in range(5):
       print("Sum is %d required %10.7f seconds"%sumOfN(100000))
Sum is 5000050000 required  0.0199420 seconds
Sum is 5000050000 required  0.0180972 seconds
Sum is 5000050000 required  0.0194821 seconds
Sum is 5000050000 required  0.0178988 seconds
Sum is 5000050000 required  0.0188949 seconds
>>>
```

Again, the time required for each run, although longer, is very consistent, averaging about 10 times more seconds. For `n` equal to 1,000,000 we get:

```
>>>for i in range(5):
       print("Sum is %d required %10.7f seconds"%sumOfN(1000000))
Sum is 500000500000 required  0.1948988 seconds
Sum is 500000500000 required  0.1850290 seconds
Sum is 500000500000 required  0.1809771 seconds
Sum is 500000500000 required  0.1729250 seconds
Sum is 500000500000 required  0.1646299 seconds
>>>
```

In this case, the average again turns out to be about 10 times the previous.

Now consider ActiveCode 3, which shows a different means of solving the summation problem. This function, `sumOfN3`, takes advantage of a closed equation $\sum_{i=1}^{n} i = \frac{(n)(n+1)}{2}$ to compute the sum of the first `n` integers without iterating.

Run | Load History | Show CodeLens

```
1 def sumOfN3(n):
2    return (n*(n+1))/2
3
4 print(sumOfN3(10))
5
```

ActiveCode: 3 Summation Without Iteration (active3)

If we do the same benchmark measurement for `sumOfN3`, using five different values for `n` (10,000, 100,000, 1,000,000, 10,000,000, and 100,000,000), we get the following results:

```
Sum is 50005000 required 0.00000095 seconds
Sum is 5000050000 required 0.00000191 seconds
Sum is 500000500000 required 0.00000095 seconds
Sum is 50000005000000 required 0.00000095 seconds
Sum is 5000000050000000 required 0.00000119 seconds
```

There are two important things to notice about this output. First, the times recorded above are shorter than any of the previous examples. Second, they are very consistent no matter what the value of `n`. It appears that `sumOfN3` is hardly impacted by the number of integers being added.

But what does this benchmark really tell us? Intuitively, we can see that the iterative solutions seem to be doing more work since some program steps are being repeated. This is likely the reason it is taking longer. Also, the time required for the iterative solution seems to increase as we increase the value of `n`. However, there is a problem. If we ran the same function on a different computer or used a different programming language, we would likely get different results. It could take even longer to perform `sumOfN3` if the computer were older.

We 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.

(Objectives.html)

(BigONotation.html)

user not logged in

ives.html)