ASSIGNMENT 1: BIG-O NOTATION, EUCLIDEAN ALGORITHM

Contents

1. Big-O notation and its relatives 1
1.1. Exercises on big-O 3
2. The Euclidean algorithm and related algorithms 4
3. Useful information 8
3.1. Problems to hand in 8
3.2. Test cases for the programming part 8

This assignment consists of two rather different sections. The first part fleshes out the basics of big-O notation. The second part asks you to implement the Euclidean algorithm and its variants, so that you can have your computer compute gcds and the like for you. Note that you are asked to do some of these calculations by hand on your written assignment, and you should do them by hand – do not write that your computer program (even if you wrote it yourself) output whatever answer you got. (The reason for this is that being able to do calculations by hand is still useful, even if you have programs which can do those calculations, because ultimately it is desirable to be able to analyze programs by hand and perform calculations on paper if you don't have access to a computer or don't want to take the time to use a computer.)

1. Big-O notation and its relatives

We are often interested in estimating the rate of growth of a function $f(x)$ as $x \to \infty$. Big-O notation provides a convenient way of writing upper bounds on the growth of functions.

Let $f(x), g(x)$ be two functions defined for all positive integers or positive real x (or, more generally, for all sufficiently large x). We say that $f(x)$ is *big-O of* $g(x)$, and write $f(x) \in O(g(x))$ or $f(x) = O(g(x))$, if there exists a positive constant C such that

$$|f(x)| \leq C|g(x)|$$

for all sufficiently large x (in other words, for all $x \geq x_0$, for some constant x_0). In practice, $g(x)$ will be positive for all large x.

The intuitive way to think about big-O notation is that $g(x)$ has order of growth at least as big as $f(x)$, for large x. In practice, $f(x)$ might be a complicated function, but $g(x)$ will be a simpler function, and saying $f(x) = O(g(x))$ gives an upper bound on the growth of $f(x)$ in terms of a function $g(x)$ which we might be more familiar with.

This is all best illustrated with some examples.

Examples.

- $x = O(x^2)$, because $x \leq x^2$ for all large x (in particular, for all $x \geq 1$).
- $5x = O(x)$, because $5x \leq 5x$ for all x. In this case, the explicit constant C is equal to 5, although of course one could choose C to be any number ≥ 5.
- $x^2 \neq O(x)$, because x^2 is always larger than Cx for large x, regardless of how big C is. For example, if $C = 10^6$, then $Cx \geq x^2$ for $x \leq 10^6$, but once $x > 10^6$, $x^2 \geq Cx$. More generally, $x^2 \geq Cx$ for all $x \geq C$, regardless of the value of the explicit constant C. You can see this behavior if you graph Cx vs x^2; x^2 evidently grows far more rapidly than Cx when x is large.
- More generally, $x^a = O(x^b)$ if and only if $a \leq b$.
- $\sin x = O(1)$, and similarly, $\cos x = O(1)$. Indeed, $|\sin x|, |\cos x| \leq 1$ for all x.
- $x^a = O(e^x)$ for any positive power a. This statement says that the exponential function grows faster than any power of x.

Closely related to big-O notation is *little-o notation*. Suppose $g(x)$ is a function which is positive for all large x. We say that $f(x)$ is little-o of $g(x)$, and write $f(x) = o(g(x))$, if

$$\lim_{x\to\infty} \frac{|f(x)|}{g(x)} = 0.$$

In other words, $f(x)$ grows slower than $g(x)$, and becomes infinitesimally small compared to $g(x)$ as x gets large.

Proposition 1. *Suppose $f(x) = o(g(x))$. Then $f(x) = O(g(x))$.*

Proof. Since $\lim_{x\to\infty} |f(x)|/g(x) = 0$, this means that for any $\varepsilon > 0$, for all large x, one has $|f(x)| < \varepsilon g(x)$. In particular we can choose any positive constant C; it must be the case that $|f(x)| < Cg(x)$ for all large x. Therefore $f(x) = O(g(x))$. $\square$

Notice that the converse to this proposition can be false sometimes. For example, we saw that $x = O(x)$, but certainly $x \neq o(x)$, since $\lim_{x\to\infty} x/x = 1$. You should think of the property of being $o(g(x))$ as being stricter than $O(g(x))$, since $O(g(x))$ includes cases where a function $f(x)$ is of the same order of growth as $g(x)$, while $o(g(x))$ does not.

Another notation related to big-O notation is asymptotic notation. We say that two functions $f(x), g(x)$ are *asymptotic* to each other if

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1,$$

and write $f(x) \sim g(x)$ if this is the case. One can immediately check that if $f(x) \sim g(x)$, then both $f(x) = O(g(x))$ and $g(x) = O(f(x))$.

Examples.

- $x^2 + 2x \sim x^2$. More generally, two polynomials are asymptotic to each other if their leading terms are identical.
- If $f(x) \sim g(x)$, and $f(x), g(x)$ are both positive for large x, then we may write $f(x) = g(x) + R(x)$, where $R(x) = o(f(x)), o(g(x))$. Indeed, if we just define $R(x)$ as $f(x) - g(x)$, then we must have

$$\lim_{x\to\infty} \frac{R(x)}{g(x)} = \lim_{x\to\infty} \frac{f(x)-g(x)}{g(x)} = 1 - 1 = 0.$$

- (Exercise) The logarithmic integral is the function $\mathrm{li}(x)$, defined for $x > 1$, by the equation

$$\int_2^x \frac{1}{\log t}\, dt.$$

  One can show that $\mathrm{li}(x) \sim x/\log x$.

Complementary to big-O and little-O notation is big-omega and little-omega notation. Whereas big-O, little-O give us a notation for expressing upper bounds, the omega notation gives us a way to express lower bounds. Suppose $f(x), g(x)$ are positive for all large x. We say that $f(x)$ is $\Omega(g(x))$ if $g(x) = O(f(x))$. An alternate definition is that $f(x) = \Omega(g(x))$ if there exists some positive constant C such that $f(x) \geq C \cdot g(x)$ for all large x.

We say that for $f(x), g(x)$ functions which are positive for all large x, $f(x) = \omega(g(x))$ if $g(x) = o(f(x))$; ie, if

$$\lim_{x\to\infty} \frac{g(x)}{f(x)} = 0.$$

1.1. **Exercises on big-O.** The following exercises should be handed in on a piece of paper. You are expected to justify your assertions.

Problem 1 (5 points)**.** Let a be any positive number. Show that $a^n = o(n!)$.

Problem 2 (5 points)**.** Show that $n! = o(n^n)$.

As a matter of fact, $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$; this is called Stirling's Formula. Naturally, you are not allowed to use this fact in your solutions to either of the two above problems.

Problem 3 (5 points)**.** We will often say that $f(x) = O(\log x)$ in a variety of situations, without specifying the base of the logarithm in question. Show that the base of the logarithm is irrelevant; ie, show that if $b_1, b_2 > 1$ are arbitrary real numbers greater than 1, and if $f(x) = O(\log_{b_1} x)$, then $f(x) = O(\log_{b_2} x)$.

Problem 4 (5 points)**.** The following is a list of six functions of a natural number n. Arrange them in increasing order of growth; ie, arrange them in a list $f_1, \ldots, f_6$ so that $f_1 = O(f_2), f_2 = O(f_3), \ldots, f_5 = O(f_6)$. The functions listed below have been chosen so that there is exactly one correct way to do this.

$$5^n, (\log n)^3, n^2, \frac{n}{\log n}, 10^{100} \cdot n^{0.99}, n!$$

Explain why your answers are correct; you can cite previous problems and basic facts we learned in class (like $x^a = O(x^b)$ if $a \leq b$, or $\log x = o(x^\varepsilon)$ for any $\varepsilon > 0$).

2. The Euclidean algorithm and related algorithms

Problem 5 (10 points)**.** Implement a function gcd(a, b), which accepts as input two positive integers $a, b \geq 1$ and returns their greatest common divisor. Your program should run in under 1 second for a, b as large as several thousand digits.

The above function calculates the gcd of two integers. However, in practice, we might be interested in calculating the gcd of two or more integers. We would like to be able to use the same function in all such cases; that is, we want to write a function gcd2 which calculates appropriate gcds when we type in something like gcd2(8, 12), which has two arguments, or something like gcd2(18, 9, 32, 45), which has 4 arguments.

Fortunately, Python supports the simple-to-use feature of passing in a variable number of arguments in a function. If you define a function f as follows:

```
def f(*args):
    # Insert code here
```

then Python will take whatever arguments f has when called (there can be a variable number), and then pack them into a data structure known as a *tuple* with the name args, which you can then use as necessary.

A tuple is an ordered list of objects, and you access the objects by using index notation. For example, if s is a tuple of length 4; say s = (2, 5, -1, "red"), then s[0] is 2, s[2] is -1, and s[3] is "red". Notice that in Python, and many other programming languages, indexing of lists/tuples/arrays/strings/etc. start with 0, not 1.

A rudimentary function which uses the variable argument feature simply prints the tuple of arguments passed into the function:

```
def printargs(*args):
    print args
```

Then loading this function into the interpreter and executing it a few times yields output like the following:

```
>>> printargs(2, 5)
(2, 5)
>>> printargs("red", 3, "a")
('red', 3, 'a')
```

More generally, suppose you have a function in which you know certain parameters will be passed in, along with a variable number of other parameters. You can still use the *args feature by first listing the variables which must be passed in, and then using *args for the variable number of parameters. The *args must always appear after the names for variables which you know will come next, not before. For example, suppose we want to modify the printargs function above to print the variable parameter list n times, where n is a variable specified by the user of the function. Then we might write a function like the following:

```
def printargs2(n, *args):
    for i in xrange(n):
        print args
```

Using this function might result in the following:

```
>>> printargs(2, 3, "hi")
(3, "hi")
(3, "hi")
>>> printargs(3)
()
()
()
```

Unlike the original printargs, because printargs2 expects at least one argument (namely, n), calling printargs2() will result in an error. Finally, the actual name "args" in the parameter list can be changed; for instance, we could use "*stuff" instead of "*args", and the tuple containing the variable argument list would be called 'stuff' instead of 'args'. The '*' symbol is what Python needs in order to know that it should pack any remaining arguments into a tuple.

This explains how we pass in a variable number of arguments into a function, but how do we do something useful with this tuple if we don't know how many elements it has? For example, suppose we wanted to write a function prod, which accepts a variable number of integer arguments, and returns their product. The function definition might start something like this:

```
def prod(*args):
    ...
```

A good idea is to use a for loop to iterate over the numbers in the tuple args, and multiply them by a running tally which contains the product of the numbers that have been examined so far:

```
def prod(*args):
    total = 1 #Product of all numbers examined so far
    for number in args:
        total = total * number
    return total
```

Notice that in the for loop, we are iterating not over 'range' or 'xrange', but rather the actual tuple args. The line "for number in args:" indicates that the variable number will iterate over the elements of args, in sequence, as each loop of the block of the code belonging to the for loop is executed. Alternately, one could write

```
def prod(*args):
    total = 1
    for i in xrange(len(args)):
        total = total * args[i]
    return total
```

The function len will return the length of a tuple, list, or string. Notice that the 0-based indexing of tuples/lists/strings makes it easy to iterate over all permissible indices of that tuple/list/string, since range(n) and xrange(n) iterate through each integer starting at 0 up to but not including n.

In any case, using the function above (either version) yields results like the following:

```
>>> prod(2, 3, 5)
30
```

```
>>> prod(-4)
-4
>>> prod()
1
```

The last use of prod makes sense, because when called with no arguments, the parameter args inside the prod function is empty, so when we reach the for loop, the loop itself is skipped over, since there is nothing to iterate over inside args.

There is also an alternate way to write the prod function. First, define a function which multiples two integers and returns the result:

```
def times(a, b):
    return a*b
```

(As you might expect, such a function already exists inside Python, but accessing it takes about as much work as writing the times function, so we just write the times function.) Python has a function called 'reduce' whose first argument is a function which accepts exactly 2 arguments, and whose second argument is a list, tuple, string, etc. (Basically, we can use anything which supports iteration, although we will only use this for lists and tuples.) Reduce accepts two arguments, the first of which is a function which accepts exactly two arguments, and the second of which is a list/tuple/string or any other type of object which supports iteration.

Suppose we have a function f(x, y), and we call reduce(f, (2, 4, 6, 8)). Then what reduce does is it calculates the value of f with the first two elements in the tuple (2, 4, 6). The result is f(2, 4). It then calls f with f(2, 4) as its first argument, and 6 with as its second argument. The result is f(f(2, 4), 6). Finally, it calls reduce with f(f(2, 4), 6) as its first argument, and 8 as its second argument. The result is f(f(f(2, 4), 6), 8), and this is the value the original call reduce(f, (2, 4, 6, 8)) returns. Let's use reduce with the times function we wrote earlier:

```
>>> reduce(times, (2, 5, 4, 3))
120
```

When the first line is executed, Python computes times(2, 5) = 10, and then computes times(10, 4) = 40, and then computes times(40, 3) = 120. Since we have run out of elements in the original tuple, Python returns 120.

Problem 6 (10 points)**.** Write a function gcd2, which accepts at least two positive integers as arguments, and returns their gcd. The fact that $\gcd(a_1, \ldots, a_n) = \gcd(\gcd(a_1, a_2), a_3, \ldots, a_n)$ will come in handy in this problem. (This is Exercise 1.9 of the textbook; you should try it out.) You should probably use the gcd function you wrote for the last problem (assuming it works correctly). Your function should run in under 1 second for inputs consisting of a list of integers whose sum of number of digits is in the thousands.

In the next program, we will write a function which returns not a single number, but rather a tuple. This is easy to do in Python; for example, suppose we want to write a function f, which given an integer n, returns the tuple (n, n+1):

```
def f(n):
    return n, n+1
```

If you want, you can put parentheses around the tuple you are returning.

Problem 7 (10 points)**.** Write a function bezout(a, b), which accepts as input two nonzero integers a, b, and returns a tuple of integers x, y which satisfies the equation

$$ax + by = \gcd(a, b).$$

Your function should run in under 1 second for a, b as long as several thousand digits, and should also work if a, b are allowed to be negative. (However, a, b are guaranteed to be nonzero.) In particular, your implementation of this function should not use brute force. The abs function in Python returns the absolute value of an integer and should be helpful.

For the previous problem, it will be useful to know how to manipulate *lists* in Python. A list is a data structure in Python which is an ordered sequence of objects, much like a tuple. A list is delimited by [,], in contrast to a tuple, which is delimited by (,). Unlike a tuple, you can change the objects which live inside the list:

```
>>> l = [2, 5, 7]
>>> print l[0]
2
>>> l[0] = 4
>>> print l
[4, 5, 7]
```

Suppose you have a list l, and you want to make the list larger by tacking a new object onto the end of the list. You do so by calling the append *method* of a list object:

```
>>> l = [2, 4]
>>> print l
[2, 4]
>>> l.append(9)
>>> print l
[2, 4, 9]
```

The reason you want to use the append method, instead of writing something like l[2] = 9, is that the latter line of code is invalid, because l[2] references a non-existent index in l. There are also methods for inserting elements at arbitrary positions inside a list, although we will not need them. You can consult the Python documentation for more information if you are interested.

Another operation that might be helpful is reversing a list. This can be done by calling the reverse method of a list object:

```
>>> l = [2, 4]
>>> l.reverse()
>>> print l
[4, 2]
```

Make sure you call the reverse method with the parentheses; if you just type in l.reverse, Python will think you are interested in the actual function which performs the reversing, as opposed to executing that function.

The next problem is for extra credit and should only be attempted once you have completed all the regular homework assignments completely and correctly. For this question and any extra credit questions which appear in the future, answers with any significant component missing will receive 0 points. (In other words, getting partial credit is much harder for extra credit problems compared to regular problems.)

Problem 8 (3 points, extra credit)**.** Write a function bezout2, which accepts at least two nonzero integer arguments, say $a_1, a_2, \ldots, a_n$, and returns a tuple of integers $x_1, \ldots, x_n$ which satisfies the equation

$$a_1x_1 + \ldots + a_nx_n = \gcd(a_1, \ldots, a_n).$$

(As a matter of fact, that the above equation even has integer solutions might not be obvious. This is a good exercise.) The function should work for inputs whose sum of number of digits is in the thousands.

3. Useful information

3.1. **Problems to hand in.** In your file which contains the code you submit, make sure that you can run the file in IDLE without error and that all functions are loaded appropriately. In other words, if you open the .py file you submit and run it in IDLE (using the Run Module command under the Run submenu), there should be no error messages in the interactive interpreter, and you should be able to use the functions you wrote. Also make sure that you spell the names of the required functions correctly! (On the other hand, the names of the arguments passed into the function can be arbitrary, as long as the function accepts the correct number of arguments.) Finally, please put your name near the top of the file in a comment (use the # symbol to write comments; Python will ignore everything on that line which comes after the #).

- Turn in problems 5, 6, 7, and possibly 8 in a single file named [lastname]1.py (without brackets around your last name.)
- Turn in problems 1, 2, 3, 4, on a physical piece of paper, and remember to justify your answers!

3.2. **Test cases for the programming part.** The following are a listing of some sample inputs and the outputs that a correct program will return. The test cases used when grading your submissions will contain these sample test cases as well as many other, possibly larger, test cases.

3.2.1. *gcd test cases.*

Input	Output
2, 5	1
32, 244	4
53234, 63921	1

So, for example, when you load your functions into the Python interpreter, typing gcd(32, 244) should return 4.

3.2.2. *gcd2 test cases.*

Input	Output
2, 5	1
236, 144, 832	4
236234, 76321, 3876932, 32123, 632763	1

3.2.3. *bezout test cases.* For the bezout function, there are potentially many different answers (you only need to return one pair of integer solutions), so you can devise your own test cases. You can check whether your answer is correct by calculating whether your output (x, y) satisfies $ax + by = \gcd(a, b)$. Of course, this assumes that your gcd function returns correct answers!