Introduction

An important question is: How efficient is an algorithm or piece of code? Efficiency covers lots of resources, including:

- CPU (time) usage
- memory usage
- disk usage
- network usage

All are important but we will mostly talk about CPU time in 367. Other classes will discuss other resources (e.g., disk usage may be an important topic in a database class).

Be careful to differentiate between:

1. Performance: how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code.
2. Complexity: how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger.

Complexity affects performance but not the other way around.

The time required by a method is proportional to the number of "basic operations" that it performs. Here are some examples of basic operations:

- one arithmetic operation (e.g., +, *).
- one assignment
- one test (e.g., x == 0)
- one read
- one write (of a primitive type)

Some methods perform the same number of operations every time they are called. For example, the *size* method of the *List* class always performs just one operation: return numItems; the number of operations is independent of the size of the list. We say that methods like this (that always perform a fixed number of basic operations) require **constant time**.

Other methods may perform different numbers of operations, depending on the value of a parameter or a field. For example, for the array implementation of the *List* class, the *remove* method has to move over all of the items that were to the right of the item that was removed (to fill in the gap). The number of moves depends both on the position of the removed item and the number of items in the list. We call the important factors (the parameters and/or fields whose values affect the number of operations performed) the **problem size** or the **input size**.

When we consider the complexity of a method, we don't really care about the **exact** number of operations that are performed; instead, we care about how the number of operations relates to the problem size. If the problem size doubles, does the number of operations stay the same? double? increase in some other way? For constant-time methods like the *size* method, doubling the problem size does not affect the number of operations (which stays the same).

Furthermore, we are usually interested in the **worst case**: what is the **most** operations that might be performed for a given problem size (other cases -- best case and average case -- are discussed below). For example, as discussed above, the *remove* method has to move all of the items that come after the removed item one place to the left in the array. In the worst case, **all** of the items in the array must be moved. Therefore, in the worst case, the time for *remove* is proportional to the number of items in the list, and we say that the worst-case time for *remove* is **linear** in the number of items in the list. For a linear-time method, if the problem size doubles, the number of operations also doubles.

Big-O Notation

We express complexity using **big-O notation**. For a problem of size N:

- a constant-time method is "order 1": O(1)
- a linear-time method is "order N": O(N)
- a quadratic-time method is "order N squared": O(N^2)

Note that the big-O expressions do not have constants or low-order terms. This is because, when N gets large enough, constants and low-order terms don't matter (a constant-time method will be faster than a linear-time method, which will be faster than a quadratic-time method). See below for an example.

Formal definition:

> A function T(N) is O(F(N)) if for some constant c and for all values of N greater than some value n_0:
>
> $$T(N) <= c * F(N)$$

The idea is that T(N) is the **exact** complexity of a method or algorithm as a function of the problem size N, and that F(N) is an upper-bound on that complexity (i.e., the actual time/space or whatever for a problem of size N will be no worse than F(N)). In practice, we want the smallest F(N) -- the **least** upper bound on the actual complexity.

For example, consider T(N) = 3 * N^2 + 5. We can show that T(N) is O(N^2) by choosing c = 4 and n_0 = 2. This is because for all values of N greater than 2:

$$3 * N^2 + 5 <= 4 * N^2$$

T(N) is **not** O(N), because whatever constant c and value n_0 you choose, I can always find a value of N greater than n_0 so that 3 * N^2 + 5 is greater than c * N.

How to Determine Complexities

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

1. Sequence of statements

   ```
   statement 1;
   statement 2;
     ...
   statement k;
   ```

   (Note: this is code that really is exactly k statements; this is **not** an unrolled loop like the N calls to *add* shown above.) The total time is found by adding the times for all statements:

   total time = time(statement 1) + time(statement 2) + ... + time(statement k)

   If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1). In the following examples, assume the statements are simple unless noted otherwise.

6. if-then-else statements

```
if (condition) {
    sequence of statements 1
}
  else {
      sequence of statements 2
  }
```

Here, either sequence 1 will execute, or sequence 2 will execute. Therefore, the worst-case time is the slowest of the two possibilities: max(time(sequence 1), time(sequence 2)). For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be O(N).

13. for loops

```
  for (i = 0; i < N; i++) {
      sequence of statements
  }
```

The loop executes N times, so the sequence of statements also executes N times. Since we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

17. Nested loops

First we'll consider loops where the number of iterations of the inner loop is independent of the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        sequence of statements
    }
}
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O(N * M). In a common special case where the stopping condition of the inner loop is `j < N` instead of `j < M` (i.e., the inner loop also executes N times), the total complexity for the two loops is O(N^2).

Now let's consider nested loops where the number of iterations of the inner loop depends on the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
    for (j = i+1; j < N; j++) {
        sequence of statements
    }
}
```

Now we can't just multiply the number of iterations of the outer loop times the number of iterations of the inner loop, because the inner loop has a different number of iterations each time. So let's think about how many iterations that inner loop has. That information is given in the following table:

Value of i	Number of iterations of inner loop
0	N
1	N-1

2	N-2
...	...
N-2	2
N-1	1

So we can see that the total number of times the sequence of statements executes is: N + N-1 + N-2 + ... + 3 + 2 + 1. We've seen that formula before: the total is $O(N^2)$.