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.

2. 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).

3. 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.

4. 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 $O(N^2)$.

Best-case and Average-case Complexity

Some methods may require different amounts of time on different calls, even when the problem size is the same for both calls. For example, consider the *add* method that adds an item to the end of the list. In the worst case (the array is full), that method requires time proportional to the number of items in the list (because it has to copy all of them into the new, larger array). However, when the array is not full, *add* will only have to copy one value into the array, so in that case its time is independent of the length of the list; i.e., constant time.

In general, we may want to consider the **best** and **average** time requirements of a method as well as its worst-case time requirements. Which is considered the most important will depend on several factors. For example, if a method is part of a time-critical system like one that controls an airplane, the worst-case times are probably the most important (if the plane is flying towards a mountain and the controlling program can't make the next course correction until it has performed a computation, then the best-case and average-case times for that computation are not relevant -- the computation needs to be guaranteed to be fast enough to finish before the plane hits the mountain).

On the other hand, if occasionally waiting a long time for an answer is merely inconvenient (as opposed to life-threatening), it may be better to use an algorithm with a slow worst-case time and a fast average-case time, rather than one with so-so times in both the average and worst cases.

Note that calculating the average-case time for a method can be tricky. You need to consider all possible values for the important factors, and whether they will be distributed evenly.

When do Constants Matter?

Recall that when we use big-O notation, we drop constants and low-order terms. This is because when the problem size gets sufficiently large, those terms don't matter. However, this means that two algorithms can have the **same** big-O time complexity, even though one is always faster than the other. For example, suppose algorithm 1 requires N^2 time, and algorithm 2 requires $10 * N^2 + N$ time. For both algorithms, the time is $O(N^2)$, but algorithm 1 will always be faster than algorithm 2. In this case, the constants and low-order terms **do** matter in terms of which algorithm is actually faster.

However, it is important to note that constants do *not* matter in terms of the question of how an algorithm "scales" (i.e., how does the algorithm's time change when the problem size doubles). Although an algorithm that requires N^2 time will always be faster than an algorithm that requires $10*N^2$ time, for **both** algorithms, if the problem size doubles, the actual time will quadruple.

When two algorithms have **different** big-O time complexity, the constants and low-order terms only matter when the problem size is small. For example, even if there are large constants involved, a linear-time algorithm will always eventually be faster than a quadratic-time algorithm. This is illustrated in the following table, which shows the value of $100*N$ (a time that is linear in N) and the value of $N^2/100$ (a time that is quadratic in N) for some values of N. For values of N less than 10^4, the quadratic time is smaller than the linear time. However, for all values of N greater than 10^4, the linear time is smaller.

N	**100*N**	**N^2/100**
10^2	10^4	10^2
10^3	10^5	10^4
10^4	10^6	10^6
10^5	10^7	10^8
10^6	10^8	10^{10}
10^7	10^9	10^{12}