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 $O(N^2)$.

Cheat Sheet for Algorithm Analysis

Following are the properties of asymptotic notations:-

1. Transitive

- If f(n) = Θ(g(n)) and g(n) = Θ(h(n)), then f(n) = Θ(h(n))
- If f(n) = O(g(n)) and g(n) = O(h(n)), then f(n) = O(h(n))
- If f(n) = o(g(n)) and g(n) = o(h(n)), then f(n) = o(h(n))
- If f(n) = Ω(g(n)) and g(n) = Ω(h(n)), then f(n) = Ω(h(n))
- If f(n) = ω(g(n)) and g(n) = ω(h(n)), then f(n) = ω(h(n))

2. Reflexivity

- f(n) = Θ(f(n))
- f(n) = O(f(n))
- f(n) = Ω(f(n))

3. Symmetry

- f(n) = Θ(g(n)) if and only if g(n) = Θ(f(n))

4. Transpose Symmetry

- f(n) = O(g(n)) if and only if g(n) = Ω(f(n))
- f(n) = o(g(n)) if and only if g(n) = ω(f(n))

5. Some other properties of asymptotic notations are as follows:

- If f (n) is O(h(n)) and g(n) is O(h(n)), then f (n) + g(n) is O(h(n)).

6. Properties of Logarithms:

- *Definition:* $\log_b a = c$ means $b^c = a$. We refer to b as the *base* of the logarithm.
- *Special cases:* $\log_b b = 1,\ \log_b 1 = 0$
- *Inverse of exponential:* $b^{\log_b x} = x$
- *Product:* $\log_b(x \times y) = \log_b x + \log_b y$
- *Division:* $\log_b(x \div y) = \log_b x - \log_b y$
- *Finite product:* $\log_b(x_1 \times x_2 \times \ldots \times x_n) = \log_b x_1 + \log_b x_2 + \ldots + \log_b x_n$
- *Changing bases:* $\log_b x = \log_c x \ / \ \log_c b$
- *Rearranging exponents:* $x^{\log_b y} = y^{\log_b x}$
- *Exponentiation:* $\log_b(x^y) = y \log_b x$

7. Useful formulas and approximations. Here are some useful formulas for approximations that are widely used in the analysis of algorithms.

- *Harmonic sum:* $1 + 1/2 + 1/3 + \ldots + 1/n \sim \ln n$
- *Triangular sum:* $1 + 2 + 3 + \ldots + n = n\,(n+1)\,/\,2 \sim n^2\,/\,2$
- *Sum of squares:* $1^2 + 2^2 + 3^2 + \ldots + n^2 \sim n^3\,/\,3$
- *Geometric sum:* If $r \neq 1$, then $1 + r + r^2 + r^3 + \ldots + r^n = (r^{n+1} - 1)\,/\,(r - 1)$
- *Geometric sum (r = 1/2):* $1 + 1/2 + 1/4 + 1/8 + \ldots + 1/2^n \sim 2$
- *Geometric sum (r = 2):* $1 + 2 + 4 + 8 + 16 + \ldots + n = 2n - 1 \sim 2n$, when n is a power of 2
- *Stirling's approximation:* $\lg(n!) = \lg 1 + \lg 2 + \lg 3 + \ldots + \lg n \sim n \lg n$
- *Exponential:* $(1 - 1/n)^n \sim 1/e$
- *Binomial coefficients:* $\binom{n}{k} \sim n^k\,/\,k!$ when k is a small constant
- *Approximate sum by integral:* If $f(x)$ is a monotonically increasing function, then $\int_0^n f(x)\,dx \;\leq\; \sum_{i=1}^{n} f(i) \;\leq\; \int_1^{n+1} f(x)\,dx$