Computational Complexity

Sebastian Jeon, Sameer Pai

September 26, 2018

"anything that's not polynomial is $\mathcal{O}(no)$"

§1 Introduction

For each test case in a USACO problem, your program must run under a predetermined time limit (default 1 second). Although computers are very fast, they can "only" do about 10^9 computations per second, and it is surprisingly easy to exceed this limit and receive a TLE (time limit exceeded) verdict.

In order to predict how quickly a program executes, we will use a tool called **computational complexity**.

§2 Complexity Analysis

The computational complexity (also referred to as "time complexity" or just "complexity") of an algorithm counts the number of computations as a function of the input variable(s) (i.e. N). For example, consider the following (simple) code:

```
for(int i = 0; i < N; i++) {
    ans++;
}
```

Let's analyze how many fundamental operations this program involves:

- The variable `i` is compared to N,
- The variable `i` is incremented,
- The variable `ans` is incremented.
- The above procedure is repeated N times (until `i` is assigned the value of N).

Thus, we see that the computer makes $3N$ operations. As the number of operations is approximately proportional to N, we say that the time complexity of this algorithm is $\mathcal{O}(N)$. To be mathematically precise,

Definition 2.1 (Big-O Notation)**.** Let $f(N)$ be the number of computations that a program requires as a function of N. We say that $f(N) = O(g(N))$ for a function $g(N)$ if there are constants c for which $f(x) < cg(x)$ for sufficiently large x. Here, g is called an **asymptotic upper bound** for f.

This definition ensures that the number of computations doesn't get "too big" as N gets big.

An immediate consequence of the definition is that the complexity of an algorithm depends on its slowest step. For example, assume we want to print a list of N numbers in increasing order. As we will later see, sorting the numbers can be done in $\mathcal{O}(N \log N)$, while printing the sorted list is obviously $\mathcal{O}(N)$. But as $n \log N$ overtakes N for large values of N as we will soon see, we can say that the entire algorithm performs in $\mathcal{O}(N \log N)$.

Of course, if an algorithm performs in $\mathcal{O}(N)$, we can also say it is $\mathcal{O}(N \log N)$ because $N \log N$ is larger than N as N gets large. However, we should note that our complexity is an upper bound, and smaller upper bounds give more information. Indeed, we will see instances where careful analysis establishes better upper bounds that lead to solutions that pass the time limit.

Exercise 2.2. Generalize the definition of Big-O notation to functions with multiple inputs.

Exercise 2.3. Find the time complexity of the following algorithm:

```
for(int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        cout << i+j << " ";
    }
    cout << "\n";
}
```

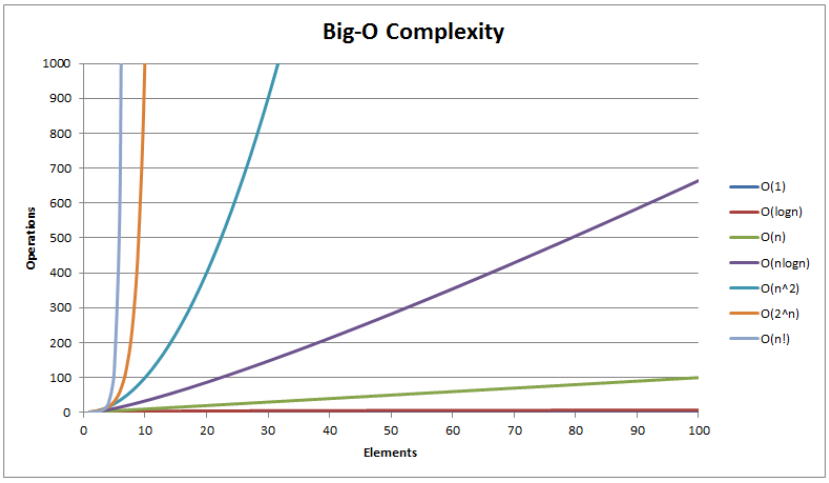

Figure 1: The behavior of common algorithms for large N

Each of the lines on this graph represents the complexity of the algorithm for many standard tasks. Here are each of the lines in order from least to greatest:

- The smallest possible runtime is $\mathcal{O}(1)$, or **constant time**. Examples of $\mathcal{O}(1)$ algorithms are adding two numbers and outputting a string.
- Next is $\mathcal{O}(\log N)$. This complexity is very small compared to N, and you usually don't have to worry about adding a log factor to your code. An example of a

$\mathcal{O}(\log N)$ algorithm is searching through a sorted list (to be addressed in a future handout).

- $\mathcal{O}(N)$ algorithms are possibly the most common complexity you will see. We saw an example of one above (i.e. the for loop), but other examples of $\mathcal{O}(N)$ algorithms are looping through an array.
- $\mathcal{O}(N \log N)$ algorithms arise when you do an $\mathcal{O}(\log N)$ algorithm N times. This is also a very common complexity to see in higher-level problems. An example of one of these algorithms is sorting an array.
- $\mathcal{O}(N^2)$ algorithms take significantly longer than $\mathcal{O}(N \log N)$ algorithms, as you can see from the graph. Still, for smaller values of N, this will be fast enough. A double for loop is the most common occurrence of a $\mathcal{O}(N^2)$ algorithm.
- $\mathcal{O}(2^N)$ algorithms are almost always far too slow to be viable. They grow extremely quickly. Iterating over all subsets of an array would be $\mathcal{O}(2^n)$.
- $\mathcal{O}(N!)$ is bad for any $N > 12$. Don't do this.

§3 Applications to Runtime

Other than the time complexity of the algorithm, many other factors affect the speed of the algorithm. Certain operations like %, in/output are generally much slower than, say, the standard arithmetic operations. Using `long int` instead of `int` can also cause extra runtime, as the computer has to manipulate more bits.

These "constant factor" issues can have a large impact on runtime in practice, especially when time-consuming operations involving STL data structures (covered in a later lecture) are at play. However, complexity analysis is still a very useful tool to check whether a solution has even a chance at passing. To this end, we propose this rule of thumb:

Theorem 3.1 (Estimate)

The time complexity function of an algorithm evaluated the maximum value of N (and other variables if present) should be less than 50 million for each second in the time limit.