Tick Tock, Tick Tock

Step 1

During World War II, Cambridge mathematics alumnus Alan Turing, the "Father of Computer Science", was recruited by British intelligence to crack the German naval Enigma, a rotor cipher machine that encrypted messages in a way that was previously impossible to decrypt by the Allied forces.

In 2014, a film titled *The Imitation Game* was released to pay homage to Turing and his critical work in the field of cryptography during World War II. In the film, Turing's leader, Commander Denniston, angrily critiques the Enigma-decrypting machine for not working, to which Turing replies:

"It works... It was just... Still working."

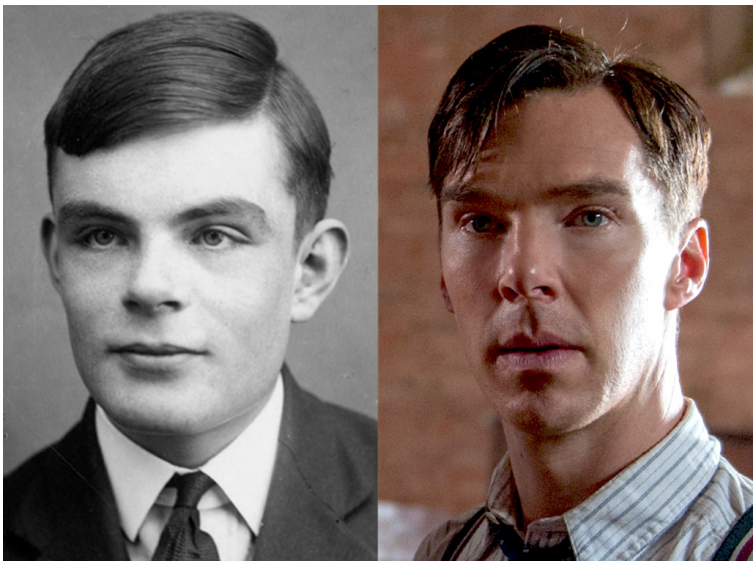


Figure: Alan Turing (left) and Benedict Cumberbatch's portrayal of him in *The Imitation Game*

Step 2

In the world of mathematics, a solution is typically either correct or incorrect, and all correct solutions are typically equally "good". In the world of algorithms, however, this is certainly not the case: even if two algorithms produce identical solutions for a given problem, there is the possibility that one algorithm is "better" than the other.

In the aforementioned film, the initial version of Turing's Enigma-cracking machine would indeed produce the correct solution if given enough time to run, but because the Nazis would change the encryption cipher every 24 hours, the machine would only be useful if it were able to complete the decryption process far quicker than this. However, at the end of the movie, Turing updates the machine to omit possible solutions based on knowledge of the unchanging greeting appearing at the end of each message. After this update, the machine is able to successfully find a solution fast enough to be useful to the British intelligence agency.

What Turing experienced can be described by **time complexity**, a way of quantifying the time taken by an algorithm to run as a function of its input size by effectively counting the number of elementary operations performed by the algorithm. Essentially, the time complexity of a given algorithm tells us roughly how fast the algorithm performs as the input size grows.

Figure: A World War II Nazi "Enigma" machine

Step 3

There are three main notations to describe time complexity: **Big-O** ("Big-Oh"), **Big-Ω** ("Big-Omega"), and **Big-Θ** ("Big-Theta"). **Big-O** notation provides an **upper-bound** on the number of operations performed, **Big-Ω** provides a **lower-bound**, and **Big-Θ** provides both **an upper-bound *and* a lower-bound**. In other words, **Big-Θ** implies *both* **Big-O *and* Big-Ω**. For our purposes, we will only consider Big-O notation because, in general, we only care about upper-bounds on our algorithms.

Let $f(n)$ and $g(n)$ be two functions defined for the real numbers. One would write

$$f(n) = \mathcal{O}(g(n))$$

if and only if there exists some constant c and some real n_0 such that the absolute value of $f(n)$ is at most c multiplied by the absolute value of $g(n)$ for all values of n greater than or equal to n_0. In other words, such that

$$|f(n)| \leq c|g(n)| \text{ for all } n \geq n_0$$

Also, note that, for all three notations, we drop all lower functions when we write out the complexity. Mathematically, say we have two functions $g(n)$ and $h(n)$, where

$$g(n) > h(n) \text{ for all } n \geq n_0$$

If we have a function $f(n)$ that is $O(g(n) + h(n))$, we would just write that $f(n)$ is $O(g(n))$ to simplify.

Note that we only care about time complexities with regard to large values of n, so when you think of algorithm performance in terms of any of these three notations, you should really only think about how the algorithm scales as n approaches infinity.

Step 4

For example, say we have a list of n numbers and, for each number, we want to print out the number and its opposite. Assuming the "print" operation is a single operation and the "opposite" operation is also a single operation, we will be performing 3 operations for each number (print itself, return the opposite of it, and print the opposite).

If we have n elements and we will be performing exactly 3 operations on each element, we will be performing exactly $3n$ operations. Therefore, we would say that this algorithm is $O(n)$, because there exists some constant (3) for which the number of operations performed by our algorithm is always bounded by the function $g(n) = n$.

Note that, although this algorithm is O(n), it is also technically O(n^2) and any other larger function of n, because technically, these are all functions that are upper-bounds on the function $f(n) = 3n$. However, when we describe algorithms, we always want to describe them using the tightest possible upper-bound.

Here's another example: say I have an algorithm that, given a list of n students, prints out 5 header lines (independent of n) and then prints the n students' names and grades on separate lines. This algorithm will perform $2n + 5$ operations total: 2 operations per student (print name, print grade) and 5 operations independent of the number of students (print 5 header lines). The time complexity of this algorithm in Big-O notation would be O($2n + 5$), which we would simplify to O(n) because we drop the constant ($2n$ becomes n) and drop all lower terms ($5 < n$ as n becomes large).

Step 5

EXERCISE BREAK: What is the tightest upper-bound for the time complexity of the following segment of C++ code (in Big-O notation)?

```
void print(vector<int> a) {
    int n = a.size();
    float avg = 0.0;
    for(int i = 0; i < n; i++) {
        cout << "Element #" << i << " is " << a[i] << endl;
        avg += a[i];
    }
    avg /= n;
    cout << "Average is " << avg << endl;
}
```

To solve this problem please visit https://stepik.org/lesson/26053/step/5

Step 6

EXERCISE BREAK: What is the tightest upper-bound for the time complexity of the following segment of C++ code (in Big-O notation)?

```
void dist(vector<int> a) {
    int n = a.size();
    for(int i = 0; i < n-1; i++) {
        for(int j = i+1; j < n; j++) {
            cout << a[j] << " - " << a[i] << " = " << (a[j]-a[i]) << endl;
        }
    }
}
```

To solve this problem please visit https://stepik.org/lesson/26053/step/6

Step 7

EXERCISE BREAK: What is the tightest upper-bound for the time complexity of the following segment of C++ code (in Big-O notation)?

```
void tricky(int n) {
    int operations = 0;
    while(n > 0) {
        for(int i = 0; i < n; i++) {
            cout << "Operations: " << operations++ << endl;
        }
        n /= 2;
    }
}
```

HINT: Does the sum representing the number of operations performed converge to some function of *n*?

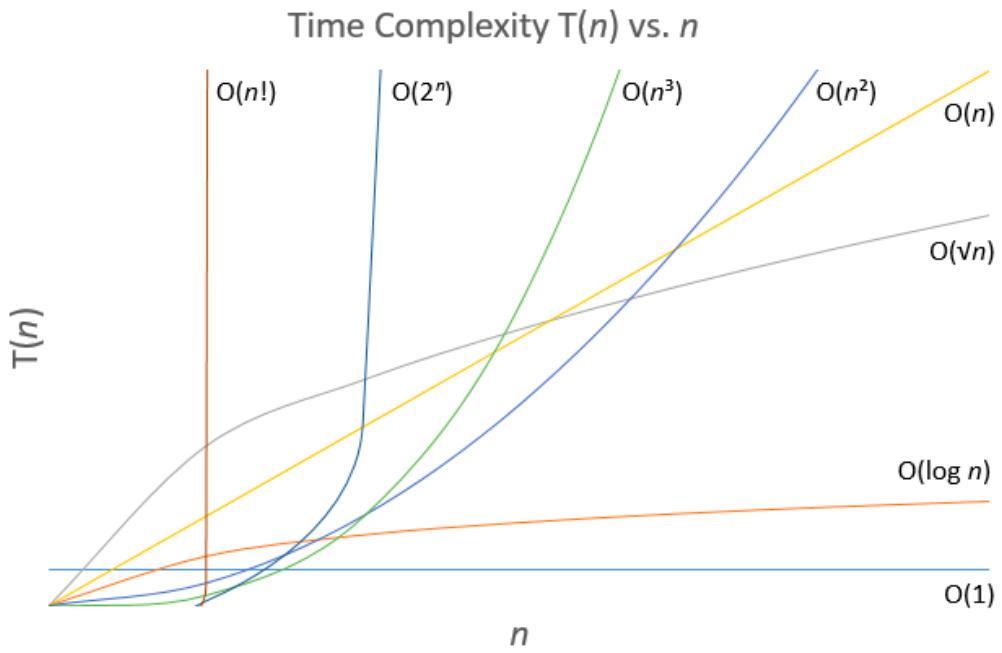
To solve this problem please visit https://stepik.org/lesson/26053/step/7

Step 8

When we discuss algorithms colloquially, we don't always explicitly say "Big-O of *n*" or "Big-O of *n*-cubed", etc. Instead, we typically use more colloquial terms to refer to Big-O time complexities. Some of the most common of these colloquial terms (along with their corresponding Big-O time complexities) have been listed below ("good" time complexities have been colored green, and "bad" time complexities have been colored red):

- **"Constant Time"** = **O(1)**
- **"Logarithmic Time"** = "Scales/Increases Logarithmically" = **O(log *n*)**
- **"Polynomial Time"** = "Scales/Increases Polynomially" = **O(n^k) (for any constant *k*)**
- **"Exponential Time"** = "Scales/Increases Exponentially" = **O(k^n) (for any constant *k*)**
- **"Factorial Time"** = **O(*n*!)**

Below is a plot of various common time complexities, and it should be clear why the "bad" time complexities are considered so "bad".



Step 9

A **data structure**, as implied by the name, is a particular structured way of storing data in a computer so that it can be used efficiently. Throughout this text, we will discuss various important data structures, both simple and complex, that are used by Computer Scientists every day. Specifically, we will cover what these data structures are, how they function (i.e., the algorithms that will be running behind-the-scenes for insertions, deletions, and look-ups), and when to appropriately use each one.

When dealing with data storage, in addition to worrying about time complexity, we will also be worrying about **space complexity**, which measures the amount of working storage needed for an input of size n. Just like time complexity, space complexity is described using Big-O notation.

For example, say we have n cities and we want to store information about the amount of gas and time needed to go from city i to city j (which is not necessarily equivalent to the amount of gas and time needed to go from city j to city i). For a given city i, for each of the other n cities, we need to represent 2 numbers: the amount of gas needed to go from city i to each of the n cities (including itself), and the amount of time needed to go from city i to each of the n cities (including itself). Therefore, we need to store $n \times n \times 2 = 2n^2$ numbers. Therefore, the space complexity is $O(n^2)$.

Step 10

The lasting goal of this text is to provide you with a solid and diverse mental toolbox of data structures so that, when you work on your own projects in the future, you will be able to recognize the appropriate data structure to use.

Much like how a brain surgeon probably shouldn't use a butter knife to perform a surgery (even though it could technically work), you should always use the appropriate data structure, even if a less appropriate one would "technically work". In the following chapters, you will begin learning about the data structures covered in this text.