Time Complexity

Authors: Darren Yao, Benjamin Qi

Contributors: Ryan Chou, Qi Wang

Measuring the number of operations an algorithm performs.

Language: C++ Edit This Page

TABLE OF CONTENTS

RESOURCES

IUSACO	**3 - Algorithm Analysis**	module is based off this
CPH	**2 - Time Complexity**	Intro and examples
PAPS	**5 - Time Complexity**	More in-depth. In particular, 5.2 gives a formal definition of Big O.
YouTube	**Introduction to Big-O**	If you prefer watching a video instead

In programming contests, your program needs to finish running within a certain timeframe in order to receive credit. For USACO, this limit is 2 seconds for C++ submissions, and 4

seconds for Java/Python submissions. A conservative estimate for the number of operations the grading server can handle per second is 10^8, but it could be closer to $5 \cdot 10^8$ given good constant factors*.

Complexity Calculations

We want a method to calculate how many operations it takes to run each algorithm, in terms of the input size n. Fortunately, this can be done relatively easily using **Big O notation**, which expresses worst-case time complexity as a function of n as n gets arbitrarily large. Complexity is an upper bound for the number of steps an algorithm requires as a function of the input size. In Big O notation, we denote the complexity of a function as $\mathcal{O}(f(n))$, where constant factors and lower-order terms are generally omitted from $f(n)$. We'll see some examples of how this works, as follows.

The following code is $\mathcal{O}(1)$, because it executes a constant number of operations.

Copy CPP

```cpp
int a = 5;
int b = 7;
int c = 4;
int d = a + b + c + 153;
```

Input and output operations are also assumed to be $\mathcal{O}(1)$. In the following examples, we assume that the code inside the loops is $\mathcal{O}(1)$.

The time complexity of loops is the number of iterations that the loop runs. For example, the following code examples are both $\mathcal{O}(n)$.

Copy CPP

```cpp
for (int i = 1; i <= n; i++) {
    // constant time code here
}
```

Copy CPP

```cpp
int i = 0;
while (i < n) {
    // constant time code here
    i++;
}
```

Because we ignore constant factors and lower order terms, the following examples are also $\mathcal{O}(n)$:

Copy CPP

```cpp
for (int i = 1; i <= 5 * n + 17; i++) {
    // constant time code here
}
```

Copy CPP

```
for (int i = 1; i <= n + 457737; i++) {
    // constant time code here
}
```

We can find the time complexity of multiple loops by multiplying together the time complexities of each loop. This example is $\mathcal{O}(nm)$, because the outer loop runs $\mathcal{O}(n)$ iterations and the inner loop $\mathcal{O}(m)$.

Copy CPP

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        // constant time code here
    }
}
```

In this example, the outer loop runs $\mathcal{O}(n)$ iterations, and the inner loop runs anywhere between 1 and n iterations (which is a maximum of n). Since Big O notation calculates worst-case time complexity, we treat the inner loop as a factor of n.* Thus, this code is $\mathcal{O}(n^2)$.

Copy CPP

```
for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j++) {
        // constant time code here
    }
}
```

If an algorithm contains multiple blocks, then its time complexity is the worst time complexity out of any block. For example, the following code is $\mathcal{O}(n^2)$.

Copy CPP

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // constant time code here
    }
}
for (int i = 1; i <= n + 58834; i++) {
    // more constant time code here
}
```

Prev Next

The following code is $\mathcal{O}(n^2 + m)$, because it consists of two blocks of complexity $\mathcal{O}(n^2)$ and $\mathcal{O}(m)$, and neither of them is a lower order function with respect to the other.

Copy CPP

```cpp
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // constant time code here
    }
}
for (int i = 1; i <= m; i++) {
    // more constant time code here
}
```

Common Complexities and Constraints

Complexity factors that come from some common algorithms and data structures are as follows:

Warning!

Don't worry if you don't recognize most of these! They will all be introduced later.

- Mathematical formulas that just calculate an answer: $\mathcal{O}(1)$
- Binary search: $\mathcal{O}(\log n)$
- Sorted set/map or priority queue: $\mathcal{O}(\log n)$ per operation
- Prime factorization of an integer, or checking primality or compositeness of an integer naively: $\mathcal{O}(\sqrt{n})$
- Reading in n items of input: $\mathcal{O}(n)$
- Iterating through an array or a list of n elements: $\mathcal{O}(n)$
- Sorting: usually $\mathcal{O}(n \log n)$ for default sorting algorithms (mergesort, `Collections.sort`, `Arrays.sort`)
- Java Quicksort `Arrays.sort` function on primitives: $\mathcal{O}(n^2)$
  - See Introduction to Data Structures for details.
- Iterating through all subsets of size k of the input elements: $\mathcal{O}(n^k)$. For example, iterating through all triplets is $\mathcal{O}(n^3)$.
- Iterating through all subsets: $\mathcal{O}(2^n)$
- Iterating through all permutations: $\mathcal{O}(n!)$

Here are conservative upper bounds on the value of n for each time complexity. You might get away with more than this, but this should allow you to quickly check whether an algorithm is viable.

n	**Possible complexities**
$n \le 10$	$\mathcal{O}(n!)$, $\mathcal{O}(n^7)$, $\mathcal{O}(n^6)$
$n \le 20$	$\mathcal{O}(2^n \cdot n)$, $\mathcal{O}(n^5)$
$n \le 80$	$\mathcal{O}(n^4)$
$n \le 400$	$\mathcal{O}(n^3)$
$n \le 7500$	$\mathcal{O}(n^2)$
$n \le 7 \cdot 10^4$	$\mathcal{O}(n\sqrt{n})$
$n \le 5 \cdot 10^5$	$\mathcal{O}(n \log n)$
$n \le 5 \cdot 10^6$	$\mathcal{O}(n)$
$n \le 10^{18}$	$\mathcal{O}(\log^2 n)$, $\mathcal{O}(\log n)$, $\mathcal{O}(1)$

Warning!

A significant portion of Bronze problems will have $n \le 100$. This doesn't give much of a hint regarding the intended time complexity. The intended solution could still be $\mathcal{O}(n)$!

Constant Factor

Constant factor refers to the idea that different operations with the same complexity take slightly different amounts of time to run. For example, three addition operations take a bit longer than a single addition operation. Another example is that although binary search on an array and insertion into an ordered set are both $\mathcal{O}(\log n)$, binary search is noticeably faster.

Constant factor is entirely ignored in Big O notation. This is fine most of the time, but if the time limit is particularly tight, you may receive time limit exceeded (TLE) with the intended complexity. When this happens, it is important to keep the constant factor in mind. For example, a piece of code that iterates through all *ordered* triplets runs in $\mathcal{O}(n^3)$ time might be sped up by a factor of 6 if we only need to iterate through all *unordered* triplets.

For now, don't worry about optimizing constant factors -- just be aware of them.

Formal Definition of Big O notation

Let f and g be non-negative functions from $\mathbb{R}_{\ge 0}$ to $\mathbb{R}_{\ge 0}$. If there exist positive constants n_0 and c such that $f(n) \le c \cdot g(n)$ whenever $n \ge n_0$, we say that $f(n) = \mathcal{O}(g(n))$.

Therefore, we *could* say that the time complexity of a linear function, $\mathcal{O}(n)$, is also $\mathcal{O}(n/2)$, $\mathcal{O}(2n)$, $\mathcal{O}(n^2)$, $\mathcal{O}(2^n)$, $\mathcal{O}(n^n)$, etc. However, we usually just write the simplest function out of those that are the most restrictive, which in the case of our linear function above, is $\mathcal{O}(n)$.

Optional: P vs. NP

P refers to the class of problems that can be solved within polynomial time ($\mathcal{O}(n^2), \mathcal{O}(n^3), \mathcal{O}(n^{100}), \ldots$). NP, short for nondeterministic polynomial time, is the set of problems with solutions that can be verified in polynomial time.

A common example of a problem in NP is a generalized version of Sudoku, where a solution is easily verifiable in polynomial time, but it is unknown whether a solution is computable in polynomial time. "P vs. NP" is the classic unsolved problem that asks whether every problem that can be verified in polynomial time can also be solved in polynomial time.

If you're interested in learning more about P vs. NP, check out this YouTube video.

Quiz

What's time complexity?

1. Measuring the amount of memory an algorithm consumes.
2. Measuring the number of operations an algorithm performs.
3. The time it takes you to solve a problem.

← Previous Question 1 of 4 Skip →

Module Progress: Not Started

Prev Next