COMP301
Data Structures & Algorithms [C++]

Dr. N. B. Gyan

Central University, Miotso. Ghana

The Big ‘O’/Big-Oh Notation

Relative Growth Rates

The analysis required to estimate the resource use of an algorithm is generally a *theoretical* issue, and therefore a formal framework is required. For example...

- Although $1000N$ is larger than N^2 for small values of N, N^2 *grows at a faster rate*, and thus N^2 will eventually be the larger function. (When do you think this will happen?)
- The turning point is $N = 1000$ in this case. Thus we are led to a number of mathematical *definitions*.

Relative Growth Rates: Definition 2.1 - Big-Oh

$T(N) = O(f(N))$ if there are positive *constants* c and n_0 such that $T(N) \leq c \cdot f(N)$ when $N \geq n_0$.

This says that eventually there is some point n_0 past which $c \cdot f(N)$ is always at least as large as $T(N)$, so that if constant factors are ignored, $f(N)$ is at least as big as $T(N)$.

Big-Oh

- In the example given, $T(N) = 1000N, f(N) = N^2, n_0 = 1000$ and $c = 1$.
- Note also that n_0 could be 10 and $c = 100$.
- Thus, it can be said that $1000N = O(N^2)$ (**order** N-Squared) and instead of saying "order ...," say "Big-Oh...".
- The idea of these definitions is to establish a *relative order* among functions.

Big-Oh

- Given two functions, there are usually points where one function is smaller than the other.
- So it does not make sense to claim, for instance, $f(N) < g(N)$.
- Thus, we compare their **relative rates of growth**.
- When we apply this to the analysis of algorithms, we shall see why this is the important measure.

Relative Growth Rates: Definition 2.2 - Big-Omega

A similar line of reasoning leads us to

$T(N) = \Omega(g(N))$ if there are positive *constants* c and n_0 such that $T(N) \geq c \cdot g(N)$ when $N \geq n_0$.

The second definition, $T(N) = \Omega(g(N))$ (pronounced “omega”), says that the growth rate of $T(N)$ is greater than or equal to ($\geq$) that of $g(N)$.

Relative Growth Rates: Definition 2.3 - Big-Theta

$T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

This third definition, $T(N) = \Theta(h(N))$ (pronounced “theta”), says that the growth rate of $T(N)$ equals (=) the growth rate of $h(N)$.

Relative Growth Rates: Definition 2.4 - Small-Oh

$T(N) = o(p(N))$ if, for all positive *constants* c, there exists an n_0 such that $T(N) < c \cdot p(N)$ when $N > n_0$.

Small-Oh

Less formally,
$T(N) = o(p(N))$ if $T(N) = O(p(N))$ and $T(N) \neq \Theta(p(N))$.

Small-Oh

$T(N) = o(p(N))$ (pronounced "little-oh"), says that the growth rate of $T(N)$ is less than ($<$) the growth rate of $p(N)$.

This is different from Big-Oh, because Big-Oh allows the possibility that the growth rates are the same.

Implications of Definitions

- When we say that $T(N) = O(f(N))$, we are *guaranteeing* that the function $T(N)$ grows at a rate no faster than $f(N)$; thus $f(N)$ is an **upper bound** on $T(N)$.
- This then implies that $f(N) = \Omega(T(N))$, and can therefore be said that $T(N)$ is a **lower bound** on $f(N)$.
- Examples:

Implications of Definitions

1. N^3 grows faster than N^2 , so we can say that $N^2 = O(N^3)$ or $N^3 = \Omega(N^2)$.
2. $f(N) = N^2$ and $g(N) = 2N^2$ grow at the same rate, so both $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$ are true.

   When two functions grow at the same rate, then the decision of whether or not to signify this with $\Theta()$ can depend on the *particular context*.

Implications of Definitions

Intuitively, if $g(N) = 2N^2$, then $g(N) = O(N^4)$, $g(N) = O(N^3)$, and $g(N) = O(N^2)$ are all technically correct, but the last option is the best answer.

3. Writing $g(N) = \Theta(N^2)$ says not only that $g(N) = O(N^2)$ but also that the result is as good (*tight*) as possible.

Rules of Thumb: Rule 1

The definitions then lead us to important set of rules:

If $T1(N) = O(f(N))$ and $T2(N) = O(g(N))$, then

1. $T1(N) + T2(N) = O(f(N) + g(N))$
2. $T1(N) * T2(N) = O(f(N) * g(N))$

Rules of Thumb: Rule 2

If $T(N)$ is a polynomial of degree k, then $T(N) = \Theta(N^k)$.

Rules of Thumb: Rule 3

$log^k N = O(N)$ for any constant k.

This tells us that logarithms grow very slowly.

Rules of Thumb

This information is sufficient to arrange most of the common functions by growth rate indicated below:

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Rules of Thumb

- Note that it is very bad style to include constants or low-order terms inside a Big-Oh. Do not say $T(N) = O(2N^2)$ or $T(N) = O(N^2 + N)$.
- In both cases, the correct form is $T(N) = O(N^2)$.
- This means that in any analysis that will require a Big-Oh answer, all sorts of shortcuts are possible.

Rules of Thumb

- Lower-order terms can generally be ignored, and constants can be thrown away.
- It is bad to say $f(N) \leq O(g(N))$, because the inequality is implied by the definition.
- And it is wrong to write $f(N) \geq O(g(N))$, because it does not make sense.

File Download Example Again

- Consider the problem of downloading a file over the Internet.
- Suppose there is an initial 3-sec delay (to set up a connection), after which the download proceeds at 1.5M(bytes)/sec.
- Then it follows that if the file is N megabytes, the time to download is described by the formula $T(N) = N/1.5 + 3$. This is a linear function.
- Notice that the time to download a 1,500M file (1,003 sec) is approximately (but not exactly) twice the time to download a 750M file (503 sec).

File Download Example Again

- Notice, also, that if the speed of the connection doubles, both times decrease, but the 1,500M file still takes approximately twice the time to download as a 750M file.
- This is the typical characteristic of linear-time algorithms, and it is the reason we write $T(N) = O(N)$, ignoring constant factors.
- (Although using big-theta would be more precise, Big-Oh answers are typically given.)

Algorithms NOT Programs

- Notice also that, although we analyze C++ code, these bounds are really bounds for the algorithms rather than programs.
- Programs are an implementation of the algorithm in a particular programming language, and almost always the details of the programming language do not affect a Big-Oh answer.

Algorithms NOT Programs

- This can occur in C++ when arrays are inadvertently copied in their entirety, instead of passed with references.
- If a program is running much more slowly than the algorithm analysis suggests, there may be an implementation inefficiency.

Simple Example Analysis 1

We wish to analyze the code for $\sum_{i=1}^{N} i^3$.

```
1 int sum( n )
2 {
3  int partialSum = 0;
4
5  for( int i = 1; i <= n; ++i )
6    partialSum += i * i * i;
7  return partialSum;
8 }
```


Simple Example Analysis 1

The analysis of this fragment is as follows.

- The declarations count for no time (line 1).
- Lines 3 and 7 count for one unit each.
- Line 5 has the hidden costs of initializing i, testing $i \leq N$, and incrementing i. The total cost of all these is 1 to initialize, $N + 1$ for all the tests, and N for all the increments, which is $2N + 2$.
- Line 6 counts for four units per time executed (two multiplications, one addition, and one assignment) and is executed N times, for a total of $4N$ units.

But...

- We ignore the costs of calling the function and returning, for a total of $6N + 4$. Thus, we say that this function is $O(N)$.
- If we had to perform all this work every time we needed to analyze a program, the task would quickly become infeasible.
- Fortunately, since we are giving the answer in terms of Big-Oh, there are lots of shortcuts that can be taken without affecting the final answer.

But...

- For instance, line 7 is obviously an $O(1)$ statement (per execution), so it is silly to count precisely whether it is two, three, or four units; it does not matter.
- Line 3 is obviously insignificant compared with the for loop, so it is useless to waste time here.
- This leads to several general rules.

General Rules for Algorithm Analysis

- **Rule 1—FOR loops**

  The running time of a `for` loop is at most the running time of the statements inside the `for` loop (including tests) times the number of iterations.

- **Rule 2—Nested loops**

  Analyze these inside out. The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

General Rules for Algorithm Analysis

As an example, the following program fragment is $O(N^2)$:

```
for( i = 0; i < n; ++i )
  for( j = 0; j < n; ++j )
    ++k;
```

General Rules

- **Rule 3—Consecutive Statements**

  These just add (which means that the maximum is the one that counts.)

General Rules

As an example, the following program fragment, which has $O(N)$ work followed by $O(N^2)$ work, is also $O(N^2)$:

```
for( i = 0; i < n; ++i )
  a[ i ] = 0;

for( i = 0; i < n; ++i )
  for( j = 0; j < n; ++j )
    a[ i ] += a[ j ] + i + j
```

General Rules

- **Rule 4—If/Else**

  For the fragment

```
if ( condition )
    S1
else
    S2
```

  the running time of an `if/else` statement is never more than the running time of the test plus the larger of the running times of $S1$ and $S2$.

Logarithms in the Running Time

- The most confusing aspect of analyzing algorithms probably centers around the logarithm.
- Some **divide-and-conquer** algorithms will run in $O(N \log N)$ time.
- Besides divide-and-conquer algorithms, the most frequent appearance of logarithms centers around the following general rule:

  An algorithm is $O(\log N)$ if it takes constant ($O(1)$) time to cut the problem size by a fraction (which is usually $\frac{1}{2}$).

Logarithms in the Running Time

- On the other hand, if constant time is required to merely reduce the problem by a constant amount (such as to make the problem smaller by 1), then the algorithm is O(N).
- Only special kinds of problems can be $O(log\ N)$, e.g **Binary Search**, **Euclid's Algorithm**, etc.

Simple Analysis Example 2

```
int a = 5; int b = 6; int c = 10;

for(int i = 0; i < n; ++i){
  for(int j = 0; j < n; ++j)
  {
    int x = i * j;
    int y = j * j;
    int z = i * j;
  }
}
```

```
for(int k = 0; k < n; ++k){
  w = a * k + 45;
  v = b * b;
}

b = 33;
```


Simple Analysis Example 2

- The number of assignment operations is the sum of four terms. The first term is the constant 3, representing the three assignment statements at the start of the fragment.
- The second term is $3n^2$, since there are three statements that are performed n^2 times due to the nested iteration.
- The third term is $2n$, two statements iterated n times.
- Finally, the fourth term is the constant 1, representing the final assignment statement.

Simple Analysis Example 2

- This gives us
  $T(n) = 3 + 3n^2 + 2n + 1 = 3n^2 + 2n + 4$
- By looking at the exponents, we can easily see that the n^2 term will be dominant and therefore this fragment of code is $O(n^2)$.
- Note that all of the other terms as well as the coefficient on the dominant term can be ignored as n grows larger.

Exercises

Find the running times of the following functions using Big 'O' notation:

1. $5n^2 + 3nlogn + 2n + 5$
2. $10nlogn + 5 + 20n^3$
3. $3logn + 2$
4. 2^{n+2}
5. $2n + 100logn$

Exercises

6. Order the following functions by growth rate: N, $\sqrt{N}$, $N^{1.5}$, N^2, $NlogN$, $Nlog\ log\ N$, $Nlog^2\ N$, $Nlog(N^2)$, $2/N$, 2^N, $2^N/2$, 37, $N^2 log\ N$, N^3.

   Indicate which functions grow at the same rate.
7. Read about algorithms which run in logarithmic times.

See you next week, God willing 🙏🏿