CMPUT 275

Topic 1: Asymptotic Notation

Rob Hackman and Xiao-Bo Li

Winter 2024

Reference

These slides are based on material from previous CMPUT 275 courses, and also the textbook "Introduction to Algorithms, 3rd edition" by Cormen, Leiserson, Rivest, and Stein.

Definition: Abstract Data Type vs Data Structure

An **abstract data type** (ADT) is an algebraic structure in mathematics. It consists of a domain, a collection of operations, and a set of constraints the operations must satisfy, such as pre-conditions and post-conditions.

A **data structure** is a way to store and organize data to facilitate access and modifications. A data structure is how a computer stores an ADT. See wikipedia: data structure.

For example,

- a list is an ADT.
- an array is one data structure used by a computer to store a list, another data structure is a linked-list.

Data structures provide a means to manage large amounts of data efficiently, and are key to designing efficient *algorithms*.

Data Structures in C++

In object oriented languages like C++, data structures are implemented as classes.

Definition: Algorithm

An **algorithm** is a well-defined **computational procedure** that takes some value, or set of values, as **input**, and produces some value, or set of values, as **output** in a finite amount of time. It is a sequence of computational steps that transform the input into the output.

An algorithm is a *tool* for solving a well-specified **computational problem**. The problem statement specifies the desired input/output relationship for problem instances of arbitrarily large size. The algorithm describes a specific procedure for achieving that relationship for all problem instances.

Correctness of Algorithms

An algorithm is **correct** if for every input instance, it halts and gives the correct output. A correct algorithm **solves** the given computational problem.

An incorrect algorithm either does not halt on some input instances, or it might halt with an incorrect answer.

Analyzing Algorithms

Analyzing an algorithm in this course means to measure its running time as a function of the input size.

Other resources used by the algorithm, such as memory, can also be analyzed.

Input Size

The input size depends on the problem.

- For sorting, it is the number of items to sort.
- For multiplying two integers, it may be the total number of bits.
- For a graph, it can be two numbers such as the number of vertices and edges.

Running Time

Running time of an algorithm on a particular input is the number of “steps” executed. The notion of “step” is **machine-independent**. It is the number of instructions and data accesses executed.

Pseudocode

Since algorithms are machine-independent, they are presented in **pseudocode**. It differs from real code because it can use any method that can clearly and concisely specify an algorithm, such as math or English. Also pseudocode typically ignores aspects of software engineering, such as data abstraction and error handling.

We will assume that line i of the pseudocode take constant time c_i to execute. Then use asymptotic notation to ignore c_i.

Note that sometimes, one line of pseudocode is actually a procedure call, such as "sort a list of numbers" whose running time is *not* constant.

- Calling a procedure is assumed to take constant time.
- Executing a procedure can take more than constant time, and is a function of the size of the input.

Best Case, Average Case, Worst Case

Algorithm running times have a best case, average case, and worst case.

We usually concentrate on finding the **worst case** running time for the following reasons.

- The worst case running time of an algorithm provides an upper bound on the running time.
- The worst case occurs fairly often.
- The average case is often roughly as bad as the worst case.

Asymptotic Running Time

The running time of an algorithm for an input of size n is denoted $T(n)$. We are interested in the running time as $n \to \infty$. This is called the **rate of growth** or **order of growth** of $T(n)$, and describes the **asymptotic efficiency** of an algorithm.

Aymptotic notation are used to provide bounds on the asymptotic efficiency of an algorithm.

For example, if the run time is

$$T(n) = an^2 + bn + c$$

for constants a, b, c. The order of growth is n^2 since lower-order terms and constants are insignificant for large n. An asymptotically tight bound on the order of growth of $T(n)$ is n^2, this is written as $\Theta(n^2)$.

Asymptotic Space Use

Asymptotic notation can also be used to describe space use, in addition to running time.

Big Θ Definition

For a given function $g(n)$, $\Theta(g(n))$ is the **set** of functions:

$$\Theta(g(n)) = \{f(n) : \exists \text{ constants } c_1 > 0, c_2 > 0, n_0 > 0 \text{ such that}$$
$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \ \forall n \geq n_0\}$$

In other words, we need to find c_1 and c_2, and some n_0 such that when $n > n_0$ the function $f(n)$ is sandwiched between $c_1 g(n)$ and $c_2 g(n)$. We say $g(n)$ is an **asymptotically tight bound** for $f(n)$.

Since $\Theta(g(n))$ is a set, the notation

$$f(n) \in \Theta(g(n))$$

describes one of the functions in this set. However, the notation

$$f(n) = \Theta(g(n))$$

is also used.

Big Θ Example 1

Show that $f(n) = n^2 - n = \Theta(n^2)$.

Solution:
We need to show there exist c_1, c_2, and n_0 such that when $n \geq n_0$,

$$c_1 n^2 \leq n^2 - n \leq c_2 n^2.$$

Divide by n:

$$c_1 \leq 1 - \frac{1}{n} \leq c_2.$$

Break up this inequality. Consider the inequality for c_2 first:

$$1 - \frac{1}{n} \leq c_2.$$

This inequality holds for $n_0 = 1$ and any $c_2 \geq 1$.

Big Θ Example 1 (cont.)

Now consider the inequality for c_1:

$$c_1 \leq 1 - \frac{1}{n}$$

Since the definition requires $c_1 > 0$, we do not want to choose $n_0 = 1$. Consider $n_0 = 6$, then when $n \geq 6$

$$c_1 \leq 1 - \frac{1}{n} \leq 1 - \frac{1}{6} = \frac{5}{6}$$

So we can choose

$$0 < c_1 \leq \frac{5}{6}.$$

Two n_0's were used in the above inequalities, if we max $6 = \max\{1, 6\}$ then the inequalities for both c_1 and c_2 will still be true.

Big Θ Example 2

Show that $n^3 \neq \Theta(n^2)$.

Solution:

Use proof by contradiction. Suppose $n^3 = \Theta(n^2)$. This means there exists constants c_2 and n_0 such that $n^3 \leq c_2 n^2$ for all $n \geq n_0$. Divide by n^2:

$$n \leq c_2$$

But since c_2 is a constant, this cannot hold for arbitrarily large n. Therefore, $n^3 \neq \Theta(n^2)$.

Big Θ: Ignoring Constants and Lower-Order Terms

Any lower-order terms in a function can be ignored because for large n they become insignificant relative to the highest-order term.

For example,

$$f(n) = an^2 + bn + c = \Theta(n^2).$$

Big Θ: Constant Functions

Constant functions are $\Theta(n^0) = \Theta(1)$.

- The $\Theta(n^0)$ notation emphasizes the variable n.
- However, $\Theta(1)$ is preferred.

Big O Definition

The Big Θ notation bounds a function from above and below. The Big O-notation provides an **asymptotic upper bound**.

For a given function $g(n)$, $O(g(n))$ is the **set** of functions:

$$O(g(n)) = \{f(n) : \exists \text{ constants } c > 0, n_0 > 0 \text{ such that } \\ 0 \leq f(n) \leq cg(n)\ \forall n \geq n_0\}.$$

As with Big $\Theta(g(n))$, $O(g(n))$ is a set. However

$$f(n) = O(g(n))$$

is the notation used instead of

$$f(n) \in O(g(n))$$

for a member of this set

Big O Example

Show that $f(n) = 5n + 3 = O(n^2)$.

Solution:

According to the definition, we need to find a constant $c > 0$ and $n_0 > 0$ such that

$$0 \leq 5n + 3 \leq cn^2 \ \forall n \geq n_0$$

This inequality holds when $c = 5 + 3 = 8$, and $n_0 = 1$.

This example shows a linear function is $O(n^2)$.

Big $\Theta \subseteq$ Big O

If $f(n) = \Theta(g(n))$, then this implies $f(n) = O(g(n))$. The big Θ-notation is a stronger notion than the big O-notation.

- As the previous example showed, $f(n) = O(n^2)$ does not imply $f(n) = \Theta(n^2)$ when $f(n)$ is a linear function.

Big O and big Θ can both be used for an asymptotically tight bound. For example, $n^2 = O(n^2)$, means the same as $n^2 = \Theta(n^2)$.

Big Ω Definition

The Big Ω-notation provides an **asymptotic lower bound**. For a given function $g(n)$, $\Omega(g(n))$ is the **set** of functions:

$$\Omega(g(n)) = \{f(n) : \exists \text{ constants } c > 0, n_0 > 0 \text{ such that } \\ 0 \leq cg(n) \leq f(n) \ \forall n \geq n_0\}.$$

Theorem for the Relationship Between Ω, Θ, and O

Let $f(n)$ and $g(n)$ be two functions.

$$f(n) = \Theta(g(n))$$

if and only if

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

Asymptotic Notation in Equations

An equality sign means the left equation is a member of the set on the right hand side.

$$n = O(n^2) \text{ means } n \in O(n^2).$$

If asymptotic notation is used as part of a formula on the right hand side

$$f(n) = n^3 + O(n^2)$$

then it stands for any arbitrary function from that set.

- This is a convenient way to emphasize a part of a function, and ignore any inessential detail or clutter, such as lower order terms.

Asymptotic Notation in Equations (cont. 1)

The notation

$$\sum_{i=1}^{n} O(i)$$

means there is only a single anonymous function, a function of i

Asymptotic Notation in Equations (cont. 2)

If asymptotic notation appears *before* another asymptotic notation, for example

$$n^2 + \Theta(n) = \Theta(n^2)$$

or

$$\begin{aligned} n^2 + 2n + 1 &= n^2 + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

then this means for *any* function in the first set, there is some function in the second set to make the equation true.

Abuse of Asymptotic Notation

If asymptotic notation is used for n *not* approaching ∞, it is meant to hide the constant. For example,

$$T(n) = \Theta(1) \text{ for } n < 3$$

means $T(n)$ is bounded by unknown constants $c_1 \leq T(n) \leq c_2$.

Little o Definition

The expression $n^2 = O(n^2)$ means the same as $n^2 = \Theta(n^2)$.

Similarly, the expression $n = O(n^2)$ means the same as $n = o(n)$, which is the little o notation. The little o-notation is used to denote an upper bound that is not asymptotically tight.

For a given function $g(n)$, $o(g(n))$ is the set:

$$o(g(n)) = \{f(n) : \forall \text{ constants } c > 0, \exists n_0 > 0 \text{ such that } \\ 0 \leq f(n) < cg(n) \ \forall n \geq n_0\}.$$

The difference with big O is that

- big O holds for *some* $c > 0$, whereas
- little o holds for *all* $c > 0$.
- Little o is defined using a **strict less than** $(<)$.

Little o Definition (cont.)

Another definition for little o is for functions $f(n)$ and $g(n)$, $f(n) = o(g(n))$ means $f(n)$ becomes insignificant relative to $g(n)$ as $n \to \infty$.

Mathematically: $f(n) = o(g(n))$ implies:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

For example, $n = o(n^2)$, but $n^2 \neq o(n^2)$.

Little ω Definition

Just like Big O has a corresponding little o-notation, Big Ω has a corresponding little ω-notation.

The little ω-notation defines a lower bound that is not asymptotically tight. For a function $g(n)$, $\omega(g(n))$ the set:

$$\omega(g(n)) = \{f(n) : \forall \text{ constants } c > 0, \exists n_0 > 0 \text{ such that } \\ 0 \le cg(n) < f(n)\ \forall n \ge n_0\}\,.$$

Similar to little o, an alternative definition is $f(n) = \omega(g(n))$ implies:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$

$f(n)$ becomes arbitrarily large relative to $g(n)$ as $n \to \infty$.

Relational Properties: Transpose Symmetry

Note the following relationship between little o and little ω:

$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)).$$

This is called **transpose symmetry**.

There is also a corresponding relationship for big O and big Ω:

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)).$$

Relational Properties: Symmetry

The following relationship for big Θ is called **symmetry**.

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)).$$

Relational Properties: Reflexivity

Big O, big Ω, and big Θ all have reflexive properties:

$$
\begin{aligned}
f(n) &= \Theta(f(n)) \\
f(n) &= O(f(n)) \\
f(n) &= \Omega(f(n))
\end{aligned}
$$

Relational Properties: Transitivity

All asymptotic notations have transitive properties. For big Θ, it is:

$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \text{ imply } f(n) = \Theta(h(n)).$$

Big Θ in the above relation can be replaced with big O, big Ω, little o, and little ω.

Trichotomy Does Not Hold

The relational properties of asymptotic notations resemble those for real numbers. However, one property that's true for real numbers is not true for asymptotic notations, trichotomy.

The **trichotomy** property of real numbers says for any two real numbers a and b, exactly one of the following must hold:

$$a < b$$
$$a = b$$
$$a > b$$

Not all functions can be compared. For example $f(n) = n$ and $g(n) = n^{1+sin(n)}$ because the exponent of $g(n)$ oscillates between 0 and 2.

Monotonicity

A function $f(n)$ is **monotonically increasing** if $m \leq n$ implies $f(m) \leq f(n)$.

A function $f(n)$ is **monotonically decreasing** if $m \leq n$ implies $f(m) \geq f(n)$.

- A constant function, and a function with jump discontinuities are both monotone.

A function $f(n)$ is **strictly increasing** if $m < n$ implies $f(m) < f(n)$.

A function $f(n)$ is **strictly decreasing** if $m < n$ implies $f(m) > f(n)$.

Floor and Ceiling

The **floor** of a real number x, denoted $\lfloor x \rfloor$, is the greatest integer less than or equal to x.

The **ceiling** of a real number x, denoted $\lceil x \rceil$, is the least integer greater than or equal to x.

The floor and ceiling functions are both monotonically increasing functions.

The following relations hold:

$$x - 1 < \lfloor x \rfloor < x < \lceil x \rceil < x + 1.$$

For any integer x,

$$\left\lfloor \frac{x}{2} \right\rfloor + \left\lceil \frac{x}{2} \right\rceil = x.$$

Modular Arithmetic

For any integer a, and any positive integer n,

$$a \bmod n$$

is the remainder when a is divided by n

$$a \bmod n = a - n\left\lfloor \frac{a}{n} \right\rfloor.$$

If $(a \bmod n) = b \bmod n$, then this is written as:

$$a \equiv b \pmod{n}$$

Polynomially Bounded

A **polynomial** in n of **degree** d is a function of the form:

$$f(n) = \sum_{i=0}^{d} a_i n^i.$$

The constants $a_0, \ldots, a_d$ are called **coefficients** and $a_d \neq 0$.

$f(n)$ is **asymptotically positive** if and only if $a_d > 0$. In this case, $f(n) = \Theta(n^d)$.

Any function $f(n)$ is **polynomially bounded** if $f(n) = O(n^k)$ for some constant k.

Exponential function

Let $a, m, n \in \mathbb{R}$ be real numbers and $a > 0$. A function of the form:

$$f(n) = a^n,$$

is an **exponential** function. If $a > 1$ then a^n is monotonically increasing. If $0 < a < 1$, then a^n is monotonically decreasing.

The exponent has the following properties:

$$\begin{aligned} a^0 &= 1 \text{ and } 0^0 = 1 \\ a^1 &= a \\ a^{-1} &= \frac{1}{a} \end{aligned}$$

In addition, the exponent can be added or multiplied:

$$\begin{aligned} (a^m)^n &= a^{mn} = (a^n)^m \\ a^m a^n &= a^{m+n} \end{aligned}$$

Comparing Exponentials and Polynomials

For all real constants $a > 1$ and b

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0.$$

This result can be shown using l'Hopital's Rule, repeatedly take derivatives and the numerator will be $b!$ which is independent of n.

Therefore,

$$n^b = o(a^n).$$

Euler's Number e

e is the real number 2.71828... and is the base of the natural log.

e^x is the limit:

$$\lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n = e^x.$$

e^x is approximated by the sum:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \sum_{i=0}^{\infty}\frac{x^i}{i!}.$$

When x is very small

$$e^x = 1 + x + \Theta(x^2)$$

uses asymptotic notation to ignore higher order terms.

Logarithms

The **logarithm function** is the inverse of the exponential function. It is defined by: if $b^y = x$ then $y = log_b(x)$. b is the base and y is the exponent. The following notations are common:

$$\lg(n) = \log_2(n)$$
$$\ln(n) = \log_e(n)$$

The bracket around the parameter is often omitted, in which case the logarithm function apll to the next term:

$$\lg n + m = \lg(n) + m.$$

The logarithm function can be exponentiated or composed:

$$\lg^k n = (\lg n)^k$$
$$\lg \lg n = \lg(\lg n)$$

Logarithm Identities

For real numbers $a > 0$, $b > 0$, $c > 0$, and n, and where the logarithm's base is greater than 1:

$$\begin{aligned}
a &= b^{log_b a} \\
\log_b(cd) &= \log_b(c) + \log_b(d) \\
\log_b(a^n) &= n \log_b(a) \\
\log_c(a) &= \frac{\log_b(a)}{\log_b(c)} \\
\log_b\left(\frac{1}{a}\right) &= -\log_b(a)
\end{aligned}$$

$$\log_b(a) = \frac{1}{\log_a(b)}$$

$$a^{\log_b(c)} = c^{\log_b(a)}$$

Logarithm Base 2

The identity:

$$\log_c(a) = \frac{\log_b(a)}{\log_b(c)}$$

shows changing the base of the logarithm changes the logarithm by a constant factor.

- Therefore, we can choose a base that is convenient and use asymptotic notation to *ignore the constant*.

Base 2 is the most common base in computer science, and the notation is

$$\log_2(a) = \lg(a).$$

Polylog Bounded

A function $f(n)$ is polylogarithmically bounded (polylog) if

$$f(n) = O(\lg^k n)$$

Recall the following limit:

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0.$$

Replace a with 2^k and n with $\lg m$:

$$\begin{aligned} 0 = \lim_{n\to\infty} \frac{n^b}{a^n} &= \lim_{n\to\infty} \frac{(\lg m)^b}{(2^k)^{\lg m}} \\ &= \lim_{n\to\infty} \frac{(\lg m)^b}{m^k}. \end{aligned}$$

Therefore,

$$\lg^k m = o(m^k).$$

Factorial

The factorial function, $f(n) = n!$, is defined recursively as:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{if } n > 0 \end{cases}$$

It can be approximated with Stirling's approximation:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right).$$

The factorial function has the following asymptotic behaviour:

$$\begin{aligned} n! &= o(n^n) \\ n! &= \omega(2^n) \\ \lg(n!) &= \Theta(n \lg n) \end{aligned}$$

Function Iteration

Let $f(n)$ be a function over the real numbers, $n \in \mathbb{R}$. Let i be a non-negative interger.

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 \\ f(f^{(i-1)}(n)) & \text{if } i > 0 \end{cases}$$

This expression means the function $f(\cdot)$ is applied i times to the initial argument n.

Iterated Logarithm

The iterated logarithm, $\lg^* n$ ("log star of n"), is the number of iterations of the log function until the result is 1 or less:

$$\lg^* n = \min\left\{ i \geq 0 : \lg^{(i)} n \leq 1 \right\},$$

where $\lg^{(i)} n$ is defined only if $\lg^{(i-1)} n > 0$.

For example, $\lg^* 16 = 3$ because $16 = 2^{2^2}$ so

$$\lg^{(3)} 16 = \lg\lg\lg\left(2^{2^2}\right) = \lg\lg\left(2^2\right) = \lg(2) = 1$$

$\lg^*$ is a **very slowly** growing function.

$$\begin{aligned} \lg^* 2 &= 1 \\ \lg^* 4 &= 2 \\ \lg^* 16 &= 3 \\ \lg^* 65536 &= 4 \\ \lg^* 2^{65536} &= 5 \end{aligned}$$

Fibonacci Numbers

The **Fibonacci numbers** is defined by the following recurrence:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \end{aligned}$$

Each number is the sum the previous two numbers. It also has a closed form expression:

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor .$$

This expression is $\frac{\phi^i}{\sqrt{5}}$ rounded to the nearest integer, where

$$\phi = \frac{1+\sqrt{5}}{2} \text{ and } \widehat{\phi} = \frac{1-\sqrt{5}}{2}.$$