

Asymptotic Analysis

CLRS 1.1

Definition 1 Algorithm: *A well-defined computational procedure which takes some value or set of values as input and produces some value or set of values as output.*

- “well-defined computational procedure” — synonymous with program running on generic computer
- usually used to solve a “computational problem”
- can “compose” algorithms

Algorithm is *correct* — for every input instance, it halts and produces the correct output.

Example 1 “*Sorting Problem*”

Input: $\langle a_1, a_2, \dots, a_n \rangle$

Output: $\langle a_1', a_2', \dots, a_n' \rangle$

$\langle 31, 41, 59, 26, 41, 28 \rangle \rightarrow \langle 26, 31, 41, 41, 58, 59 \rangle$. Will refer to $\langle 31, 41, 59, 26, 41, 28 \rangle$ as an “instance”

Concrete example of an algorithm: **insertion sort**. Pseudo-code in Figure 1

Two fundamental issues:

1. Analysis **CLRS 2.2**
2. Design **CLRS 2.3**

Analysis — predict “computational resources”, e.g., **time**, space, communication, logic gates, etc.

```

insertion_sort( A )
  for j <- 2 to length[A]
    do key <- A[j]
      // insert A[j] into the sorted sequence A[1..j-1]
      i <- j - 1
      while i > 0 and A[i] > key
        do A[j+1] <- A[j]
          i <- i - 1
      A[i+1] <- key

```

Figure 1: Pseudo-code for insertion sort. See CLRS 2.1 for details, notation.

- can be **very** difficult!

We'll be examining the problem of determining the run time needed: ignore the development time, compile time.

Consider insertion sort —

1. runtime is a function of input “sortedness”
2. runtime is also a function of array size

General fact —

- time taken grows with input size

Need to formalize the notion of runtime, size of input

Size: depends on problem being solved

1. sorting – array length
2. multiplying large binary integers — total number of bits

Sometimes two components to size, e.g., $m \times n$ matrix, vertices/edges in graph.

Running time: for an algorithm on a particular input

- would like to make “machine independent”
 - count number of “primitive steps” executed (think of as machine instructions)
 - * take care with function calls

Example 2 insertion-sort

1. *array already sorted* — $a \cdot n + b$

2. *reverse sorted* — $c \cdot n^2 + d \cdot n + e$

VERY IMPORTANT: We will always focus on the “worst case” runtime as a function of the input size.

- gives us a guarantee
- commonly seen:
 - worst case happens
 - average case same as worst case

Will examine the “rate of growth”

- impossible to predict exact runtimes (need very elaborate experimental methodology)

Useful to compare algorithms for same problem, predict growth.

Recall insertion sort worst case: $c \cdot n^2 + d \cdot n + e$ — focus on the “dominant term”

CLRS 3.1

Mathematical foundations: will be looking at real-valued functions on $N = \{0, 1, 2, \dots\}$

Definition 2 Given $g(n)$, denote by $\Theta(g(n))$ the set

$$\{f(n) \mid \exists c_1, c_2 \text{ and } \exists n_0 \text{ such that } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0\}$$

Conceptually — f is “sandwiched” between c_1g and c_2g for large n .

Common to abuse notation and say $f(n) = \Theta(g(n))$ when $f(n) \in \Theta(g(n))$.

Example 3 $n^2/2 - 3n = \Theta(n^2)$

Proof: Need to show c_1, c_2, n_0 such that $\forall n \geq n_0 \quad c_1 n^2 \leq n^2/2 - 3n \leq c_2 n^2$.

Equivalently, need to show: $\forall n \geq n_0 \quad c_1 \leq 1/2 - 3/n \leq c_2$.

Take $c_2 \geq 1/2 \Rightarrow 1/2 - 3/n \leq c_2$ if $n \geq 1$.

Take $c_1 \leq 1/14 \Rightarrow 1/14 \leq 1/2 - 3/n$ if $n \geq 7$.

So $c_1 = 1/14, c_2 = 1/2, n_0 = 7$ works. ■

Note there are many other choices for c_1, c_2, n_0 ; similarly for other functions in $\Theta(n^2)$ we may need different c_1, c_2, n_0 .

Example 4 $6n^3 \neq \Theta(n^2)$

Suppose $6n^3 = \Theta(n^2)$. Then there exists c_1, c_2 , and n_0 such that $6n^3 \leq c_2 n^2 \quad \forall n \geq n_0$; equivalently, $n \leq c_2/6 \quad \forall n \geq n_0$

This is impossible! Hence the supposition $6n^3 = \Theta(n^2)$ must be false.

***O*-notation — “Asymptotic Lower Bound”**

Definition 3

$O(g(n)) = \{f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0\}$

As with Θ notation, we'll write $f(n) = O(g(n))$ when we really mean $f(n) \in O(g(n))$.

Conceptually — Θ -notation specifies upper and lower bounds; the O -notation specifies only upper bound. Look at graphs of $f(n)$ and $g(n)$ for more intuition. Advantage of O notation: usually easier to come up with by inspecting the algorithm.

Straightforward fact: if $f(n) = \Theta(g(n))$ it must be that $f(n) = O(g(n))$.

Example 5 $a \cdot n + b$ is in $O(n^2)$. (Reason: take $c = |a| + |b|$ and $n_0 = 1$. Check — $a \cdot n + b \leq (|a| + |b|) \cdot n^2$ holds whenever $n \geq 1$.)

Warnings:

1. Oldtimers use O where we use Θ .
2. We'll often say “running time of insertion sort is $O(n^2)$ ” when it would be more precise to say “the worst case running time of insertion sort is $O(n^2)$.”

Ω -notation — “Asymptotic Upper Bound”

$\Omega(g(n)) = \{f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0\}$

Easy fact:

Theorem 1 For any two functions $f(n)$ and $g(n)$

$$f(n) = \Theta(g(n)) \text{ iff } f(n) = O(g(n)) \text{ and } g(n) = O(f(n))$$

We’ll find this theorem useful when we want to prove that two functions are “asymptotically equivalent.”

Notation: thus far have been writing things like $n = O(n^2)$. Later will find it convenient to write $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$; by this we mean $2n^2 + 3n + 1 = 2n^2 + g(n)$, where $g(n) = \Theta(n)$.

 o -notation

First the idea: $2n^2 = O(n^2)$ and $2n = O(n^2)$; however, first bound is “tight,” the second isn’t. With this in mind we define the “little-oh” notation:

Definition 4

$o(g(n)) = \{f(n) \mid \text{for each constant } c > 0 \text{ there exists } n_0 > 0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0\}$

Example 6 $2n = o(n^2)$ but $2n^2 \neq O(n^2)$.

Note the difference between “little-oh” and “big-Oh”

- Big Oh — $f(n) \leq c \cdot g(n)$ for some c
- Little Oh — $f(n) \leq c \cdot g(n)$ for every c

Conceptually: “ $f(n)$ is (asymptotically) insignificant with respect to $g(n)$.”

Theorem 2 $f(n) = o(g(n))$ iff $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$

Proof: Use the definition of $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ ■

Comparing functions

Analogs of the ordering properties of the real numbers hold for functions:

$$\begin{aligned}
 f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) & \text{ imply } f(n) = \Theta(h(n)) \\
 f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) & \text{ imply } f(n) = O(h(n)) \\
 f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) & \text{ imply } f(n) = \Omega(h(n)) \\
 f(n) & = \Theta(f(n)) \\
 f(n) = \Theta(g(n)) & \text{ iff } g(n) = \Theta(f(n)) \\
 f(n) = O(g(n)) & \text{ iff } g(n) = \Omega(f(n))
 \end{aligned}$$

As such, the following analogies can be made:

$$\begin{aligned}
 f(n) = O(g(n)) & \sim a \leq b \\
 f(n) = \Omega(g(n)) & \sim a \geq b \\
 f(n) = \Theta(g(n)) & \sim a = b \\
 f(n) = o(g(n)) & \sim a < b \\
 f(n) = \omega(g(n)) & \sim a > b
 \end{aligned}$$

However, the analogy can break down: for any real numbers a and b , exactly one of the following must be true: $a < b$ or $a = b$ or $a > b$. However given two functions $f(n)$ and $g(n)$, it is not always the case that $f(n) = O(g(n))$ or $f(n) = \Theta(g(n))$ or $f(n) = \Omega(g(n))$.

Counter example: $f(n) = n$, $g(n) = n^{1+\sin n}$.

Remember — we are interested in program performance.

Why is it content free to say “the running time of algorithm A is at least $O(n^2)$ ”?

CLRS 3.2

Standard notation and terminology: you should be familiar with monotonicity, strict monotonicity, $\lceil x \rceil$, $\lfloor y \rfloor$, polynomials, a^n , $\log_a b$, \lg , \ln , $n!$.

We will define $\lg^*(n)$ when we need it; will never encounter Fibonacci numbers.