CS 240E – Data Structures and Data Management (Enriched)

Module 1: Introduction and Asymptotic Analysis

Therese Biedl

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2025

version 2025–01–04 03:40

Outline

Outline

1. Introduction and Asymptotic Analysis
   - How to “Solve a Problem”
   - Asymptotic Notation
   - Rules for asymptotic notation
   - Analysis of Algorithms Revisited

Algorithms and Problems (review)

Problem: Description of possible input and desired output.
Example: Sorting problem.

Algorithm: *Step-by-step process*, works on any **instance** I.
Example: *insertion-sort*

1) Describe the overall idea
"Keep part of array sorted, and repeatedly add more to sorted part."

2) Give **pseudo-code** or detailed description.

```
insertion-sort(A, n)
A: array of size n
    for (i ← 1; i < n; i++) do
        for (j ← i; j > 0 and A[j−1] > A[j]; j−−) do
            swap A[j] and A[j − 1]
```

Pseudo-code: designed for a person, not for a computer.

Algorithms and Problems (review)

3) Argue correctness.

- Typically state loop-invariants, or other key-ingredients, but no need for a formal (CS245-style) proof by induction.
- Sometimes obvious enough from idea-description and comments.

4) Analyze the algorithm.

- We want to bound the number of **primitive operations**
- We want to bound the **auxiliary space**
- We need a computer model: **Random Access Model** (RAM)
  - unlimited set of memory cells
  - any number fits into a cell (but do not abuse)
  - standard arithmetic operations, but no $\sqrt{\cdot}$, sin, ...
  - all operations take the same amount of time
- We do not count exactly, instead use **asymptotic analysis** (big-O)

Outline

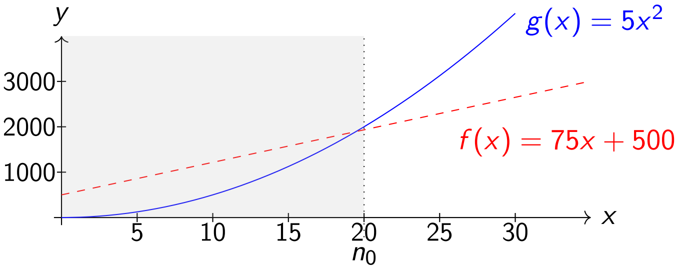
Order Notation overview

Symbol and acronyms			picture / definition	Typical use
O	big-O	asymptotic upper bound		*merge-sort* takes $O(n \log n)$ time.
Ω	big-Omega	asymptotic lower bound		*insertion-sort* may take $\Omega(n^2)$ time.
Θ	Theta	asymptotically the same/tight		*insertion-sort* has worst-case run-time $\Theta(n^2)$
o	little-o	asymptotically strictly smaller		*merge-sort* asymp. faster than *insertion-sort* in worst case.
ω	little-omega	asymptotically strictly bigger		*merge-sort* uses asymp. more space than *insertion-sort*.

Order Notation

Study relationships between *functions*.

Example: $f(x) = 75x + 500$ and $g(x) = 5x^2$ (e.g. $c = 1, n_0 = 20$)



***O*-notation:** $f(x) \in O(g(x))$ (f is *asymptotically upper-bounded* by g) if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. $|f(x)| \leq c\,|g(x)|$ for all $x \geq n_0$.

In CS240: Parameter is usually an integer (write n rather than x). $f(n), g(n)$ usually positive for sufficiently big n (omit absolute value signs).

Asymptotic Lower Bound

- We have $2n^2 + 3n + 11 \in O(n^2)$.
- But we also have $2n^2 + 3n + 11 \in O(n^{10})$.
- We want a *tight* asymptotic bound.

Ω-notation: $f(x) \in \Omega(g(x))$ (f is *asymptotically lower-bounded* by g) if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. $c\,|g(x)| \leq |f(x)|$ for all $x \geq n_0$.

Example: Prove that $f(n) = 2n^2 + 3n + 11 \in \Omega(n^2)$ from first principles.

Example: Prove that $\frac{1}{2}n^2 - 5n \in \Omega(n^2)$ from first principles.

Asymptotic Tight Bound

Θ-notation: $f(x) \in \Theta(g(x))$ (f is *asymptotically tightly-bounded* by g) if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ such that

$$c_1 |g(x)| \leq |f(x)| \leq c_2 |g(x)| \text{ for all } x \geq n_0.$$

Equivalently: $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

We also say that *the growth rates of f and g are the same*. Typically, $f(x)$ may be "complicated" and $g(x)$ is chosen to be a very simple function.

Example: Prove that $\log_b(n) \in \Theta(\log n)$ for all $b > 1$ from first principles.

Common Growth Rates

Commonly encountered growth rates in analysis of algorithms include the following:

- $\Theta(1)$ (*constant*),
- $\Theta(\log n)$ (*logarithmic*),
- $\Theta(n)$ (*linear*),
- $\Theta(n \log n)$ (*linearithmic*),
- $\Theta(n \log^k n)$, for some constant k (*quasi-linear*),
- $\Theta(n^2)$ (*quadratic*),
- $\Theta(n^3)$ (*cubic*),
- $\Theta(2^n)$ (*exponential*).

These are sorted in *increasing order* of growth rate.

Common Growth Rates

Commonly encountered growth rates in analysis of algorithms include the following:

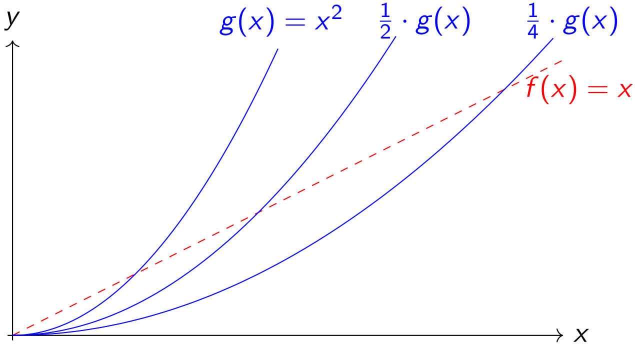
- $\Theta(1)$ (*constant*),
- $\Theta(\log n)$ (*logarithmic*),
- $\Theta(n)$ (*linear*),
- $\Theta(n \log n)$ (*linearithmic*),
- $\Theta(n \log^k n)$, for some constant k (*quasi-linear*),
- $\Theta(n^2)$ (*quadratic*),
- $\Theta(n^3)$ (*cubic*),
- $\Theta(2^n)$ (*exponential*).

These are sorted in *increasing order* of growth rate.

How do we define 'increasing order of growth rate'?

Strictly smaller asymptotic bounds

- We have $f(n) = n \in \Theta(n)$.
- How to express that $f(n)$ grows slower than n^2?



***o*-notation:** $f(x) \in o(g(x))$ (f is *asymptotically strictly smaller* than g) if for all constants $c > 0$, there exists a constant $n_0 \geq 0$ such that $|f(x)| \leq c\,|g(x)|$ for all $x \geq n_0$.

Strictly smaller/larger asymptotic bounds

Example: Prove that $n \in o(n^2)$ from first principles.

Strictly smaller/larger asymptotic bounds

Example: Prove that $n \in o(n^2)$ from first principles.

- Main difference between o and O is the quantifier for c.
- n_0 will depend on c, so it is really a function $n_0(c)$.
- We also say 'the growth rate of f is *less than* the growth rate of g'.
- Rarely proved from first principles (instead use limit-rule $\rightsquigarrow$ later).

ω-notation: $f(x) \in \omega(g(x))$ (f is *asymptotically strictly larger* than g) if for all constants $c > 0$, there exists a constant $n_0 \geq 0$ such that $|f(x)| \geq c\,|g(x)|$ for all $x \geq n_0$.

- Symmetric, the growth rate of f is *more than* the growth rate of g.

Outline

1. Introduction and Asymptotic Analysis
   - How to "Solve a Problem"
   - Asymptotic Notation
   - Rules for asymptotic notation
   - Analysis of Algorithms Revisited

The Limit Rule

Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Suppose that

$$L = \lim_{n\to\infty} \frac{f(n)}{g(n)} \quad \text{(in particular, the limit exists).}$$

Then

$$f(n) \in \begin{cases} o(g(n)) & \text{if } L = 0 \\ \Theta(g(n)) & \text{if } 0 < L < \infty \end{cases}$$

If the fraction tends to infinity then $f(n) \in \omega(g(n))$.

The required limit can often be computed using *l'Hôpital's rule*. Note that this result gives *sufficient* (but not necessary) conditions for the stated conclusion to hold.

Application 1: Logarithms vs. polynomials

Compare the growth rates of $f(n) = \log n$ and $g(n) = n$.

Now compare the growth rates of $f(n) = (\log n)^c$ and $g(n) = n^d$ (where $c > 0$ and $d > 0$ are arbitrary numbers).

Application 2: Polynomials

Let $f(n)$ be a polynomial of degree $d \geq 0$:

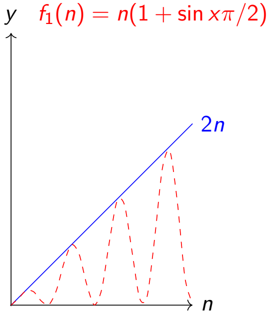
$$f(n) = c_d n^d + c_{d-1} n^{d-1} + \cdots + c_1 n + c_0$$

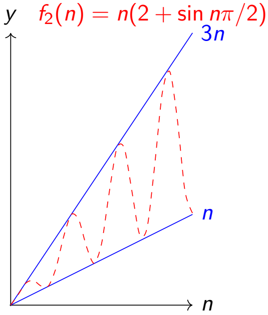
for some $c_d > 0$.

Then $f(n) \in \Theta(n^d)$:

Example: Oscillating functions

Consider two oscillating functions f_1, f_2 for which $\lim_{n\to\infty} \frac{f_i(n)}{n}$ does not exist. Are they in $\Theta(n)$?





So no limit $\rightsquigarrow$ must use other methods to prove asymptotic bounds.

Relationships between Order Notations

- $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$
- $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
- $f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$

- $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \notin \Omega(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \notin O(g(n))$

Example: Fill the following table with TRUE or FALSE:

		Is $f(n) \in \ldots(g(n))$?			
$f(n)$	$g(n)$	o	O	Ω	ω
$\log n$	$\sqrt{n}$				

Asymptotic Notation and Arithmetic

- Normally, we say $f(n) \in \Theta(g(n))$ because $\Theta(g(n))$ is a set.
- Avoid doing arithmetic with asymptotic notations.
  Do **not** write $O(n) + O(n) = O(n)$.
  (CS136 allowed you to be sloppy here. CS240 does not, mostly because it can go badly wrong with recursions.)
- Instead, when you do arithmetic, replace '$\Theta(f(n))$' by '$c \cdot f(n)$ for some constant $c > 0$'
  (That's still a bit sloppy (why?), but less dangerous.)

Asymptotic Notation and Arithmetic

- Normally, we say $f(n) \in \Theta(g(n))$ because $\Theta(g(n))$ is a set.
- Avoid doing arithmetic with asymptotic notations.
  Do **not** write $O(n) + O(n) = O(n)$.
  (CS136 allowed you to be sloppy here. CS240 does not, mostly because it can go badly wrong with recursions.)
- Instead, when you do arithmetic, replace '$\Theta(f(n))$' by '$c \cdot f(n)$ for some constant $c > 0$'
  (That's still a bit sloppy (why?), but less dangerous.)
- There are some (very limited) exceptions:
  - $f(n) = n^2 + \Theta(n)$ means "$f(n)$ is n^2 plus a linear term"
    - nicer to read than "$n^2 + n + \log n$"
    - more precise about constants than "$\Theta(n^2)$"
  - But use this very sparingly (typically only for stating the final result)

Asymptotic Notation and Arithmetic

- Normally, we say $f(n) \in \Theta(g(n))$ because $\Theta(g(n))$ is a set.

- Avoid doing arithmetic with asymptotic notations.
  Do **not** write $O(n) + O(n) = O(n)$.
  (CS136 allowed you to be sloppy here. CS240 does not, mostly because it can go badly wrong with recursions.)

- Instead, when you do arithmetic, replace '$\Theta(f(n))$' by '$c \cdot f(n)$ for some constant $c > 0$'
  (That's still a bit sloppy (why?), but less dangerous.)

- There are some (very limited) exceptions:
  - $f(n) = n^2 + \Theta(n)$ means "$f(n)$ is n^2 plus a linear term"
    - nicer to read than "$n^2 + n + \log n$"
    - more precise about constants than "$\Theta(n^2)$"
  - But use this very sparingly (typically only for stating the final result)
  - Similarly $f(n) = n^2 + o(1)$ means "n^2 plus a vanishing term."

Outline

Complexity of Algorithms

- To measure run-time, count primitive operations, sum up over loops, bound asymptotically.
- Run-time $T(n)$ is always a function of the **input size** n.
- Algorithm can have different running times on two instances of the same size.

insertion-sort$(A,\ n)$
A: array of size n
1. **for** $(i \leftarrow 1; i < n; i{+}{+})$ **do**
2. **for** $(j \leftarrow i; j > 0$ and $A[j{-}1] > A[j]; j{-}{-})$ **do**
3. swap $A[j]$ and $A[j-1]$

Let $T_{\mathcal{A}}(I)$ denote the running time of an algorithm $\mathcal{A}$ on instance I.

Study this value for the worst-possible, best-possible and 'typical' (average) instance I.

Complexity of Algorithms

Worst-case (best-case) complexity of an algorithm: The *worst-case (best-case) running time* of an algorithm $\mathcal{A}$ is a function $T : \mathbb{Z}^+ \to \mathbb{R}$ mapping n (the input size) to the *longest (shortest)* running time for any input instance of size n:

$$T_{\mathcal{A}}^{\text{worst}}(n) = \max_{I \in \mathcal{I}_n} \{ T_{\mathcal{A}}(I) \}$$

$$T_{\mathcal{A}}^{\text{best}}(n) = \min_{I \in \mathcal{I}_n} \{ T_{\mathcal{A}}(I) \}$$

To prove a lower bound on the worst-case run-time: Pick one especially bad example, and bound its run-time (using Ω-notation).

Average-case complexity of an algorithm: The average-case running time of an algorithm $\mathcal{A}$ is a function $T : \mathbb{Z}^+ \to \mathbb{R}$ mapping n (the input size) to the *average* running time of $\mathcal{A}$ over all instances of size n:

$$T_{\mathcal{A}}^{avg}(n) = \sum_{I \in \mathcal{I}_n} T_{\mathcal{A}}(I) \cdot (\text{relative frequency of } I)$$

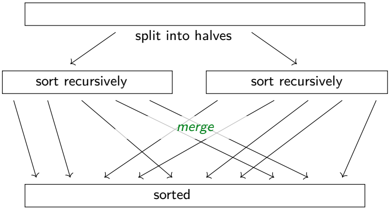
Analysis of recursive algorithms

We illustrate this here on *merge-sort*.

Step 1: Describe the overall idea

Input: Array A of n integers

1. We split A into two subarrays A_L and A_R that are roughly half as big.
2. *Recursively* sort A_L and A_R
3. After A_L and A_R have been sorted, use a function *merge* to merge them into a single sorted array.



Explaining the solution of a problem

Step 2: Give pseudo-code or detailed description.

```
merge-sort(A, n)
A: array of size n
    if (n ≤ 1) then return
    else
        m = ⌊(n − 1)/2⌋
        merge-sort(A[0..m], m + 1)
        merge-sort(A[m + 1..n−1], r)
        merge(A[0..m], A[m + 1..n−1])
```

Improvements to this, and pseudo-code for *merge* $\rightsquigarrow$ course notes

Analysis of merge-sort

Step 3: Argue correctness.

- Typically state loop-invariants, or other key-ingredients, but no need for a formal (CS245-style) proof by induction.
- Sometimes obvious enough from idea-description and comments.

Step 4: Analyze the run-time.

- First analyze work done outside recursions.
- If applicable, analyze subroutines separately.
- If there are recursions: how big are the subproblems? The run-time then becomes a recursive function.

Let $T(n)$ denote the time to run *merge-sort* on an array of length n.

1. (initialize array) takes time $\Theta(n)$
2. (recursively call *merge-sort*) takes time $T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right)$
3. (call *merge*) takes time $\Theta(n)$

The run-time of *merge-sort*

- The **recurrence relation** for $T(n)$ is as follows (constant factor c replaces Θ):

$$T(n) = \begin{cases} T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + c\,n & \text{if } n > 1 \\ c & \text{if } n = 1. \end{cases}$$

- The following is the corresponding **sloppy recurrence** (it has floors and ceilings removed):

$$T(n) = \begin{cases} 2\,T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \\ c & \text{if } n = 1. \end{cases}$$

- When n is a power of 2, then the exact and sloppy recurrences are *identical* and can easily be solved by various methods. E.g. prove by induction that $T(n) = cn\log(2n) \in \Theta(n\log n)$.
- It is possible to show that $T(n) \in \Theta(n\log n)$ *for all n* by analyzing the exact recurrence.

Some Recurrence Relations

Recursion	resolves to	example
$T(n) \leq T(n/2) + O(1)$	$T(n) \in O(\log n)$	binary-search
$T(n) \leq 2T(n/2) + O(n)$	$T(n) \in O(n \log n)$	merge-sort
$T(n) \leq 2T(n/2) + O(\log n)$	$T(n) \in O(n)$	heapify (*)
$T(n) \leq cT(n-1) + O(1)$ for some $c < 1$	$T(n) \in O(1)$	avg-case analysis (*)
$T(n) \leq 2T(n/4) + O(1)$	$T(n) \in O(\sqrt{n})$	range-search (*)
$T(n) \leq T(\sqrt{n}) + O(\sqrt{n})$	$T(n) \in O(\sqrt{n})$	interpol. search (*)
$T(n) \leq T(\sqrt{n}) + O(1)$	$T(n) \in O(\log \log n)$	interpol. search (*)

- Once you know the result, it is (usually) easy to prove by induction.
- These bounds are tight if the upper bounds are tight.
- Many more recursions, and some methods to find the result, in CS341.

(*) These may or may not get used later in the course.