CS 240 – Data Structures and Data Management

Module 1: Introduction and Asymptotic Analysis

Armin Jamshidpey, Éric Schost

Based on lecture notes by many previous cs240 instructors

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

Spring 2026

version 2026−05−07 16:32

Outline

1. Introduction and Asymptotic Analysis
   - CS240 overview
   - Algorithm design
   - Analysis of algorithms I
   - Rules for asymptotic notation
   - Analysis of algorithms revisited

Outline

1. Introduction and Asymptotic Analysis
   - CS240 overview
   - Algorithm design
   - Analysis of algorithms I
   - Rules for asymptotic notation
   - Analysis of algorithms revisited

Course objectives: What is this course about?

- Much of Computer Science is *problem solving*: Write a program that converts the given input to the expected output.
- When first learning to program, we emphasize *correctness*: does your program output the expected results?
- Starting with this course, we will also be concerned with *efficiency*: is your program using the computer's resources (typically processor time) efficiently?
- We will study efficient methods of *storing*, *accessing*, and *organizing* large collections of data.

  Motivating examples: Dictionary (e.g., as in Python)

  Typical operations include: *inserting* new data items, *deleting* data items, *searching* for specific data items, *sorting*.

Course objectives: What is this course about?

- We will consider various **abstract data types** (ADTs) and how to realize them efficiently using appropriate **data structures**.
- We will some problems in **data management** (sorting, pattern matching, compression) and how to solve them with efficient **algorithms**.
- Main objective: efficiency. We may briefly discuss naive solutions, but the main focus is on fast algorithms.
- There is a strong emphasis on mathematical analysis in the course.
- Algorithms are presented using pseudo-code and analyzed using order notation (big-Oh, etc.).

Course topics

1. background, big-Oh analysis
2. priority queues and heaps
3. efficient sorting, selection
4. binary search trees, AVL trees
5. skip lists
6. tries
7. hashing
8. quadtrees, kd-trees, range search
9. string matching
10. data compression
11. external memory

1 module $\approx$ 1 week per topic.

Required CS background

Topics covered in previous courses:

- arrays, linked lists
- strings
- stacks, queues
- abstract data types
- recursive algorithms
- binary trees
- basic sorting
- binary search
- binary search trees

Most are briefly reviewed in course notes, or consult any textbook (e.g. [Sedgewick,CLRS]).

Useful math facts

Logarithms:

- $y = \log_b(x)$ means $b^y = x$. e.g. $n = 2^{\log n}$.
- $\log(x)$ (in this course) means $\log_2(x)$
- $\log(x \cdot y) = \log(x) + \log(y)$, $\log(x^y) = y \log(x)$, $\log(x) \leq x$
- $\log_b(a) = \frac{\log_c a}{\log_c b} = \frac{1}{\log_a(b)}$, $a^{\log_b c} = c^{\log_b a}$
- $\ln(x) =$ natural log $= \log_e(x)$, $\frac{\mathrm{d}}{\mathrm{d}x} \ln x = \frac{1}{x}$

Factorial:

- $n! := n(n-1)(n-2)\cdots 2 \cdot 1 =$ # ways to permute n elements
- $\log(n!) = \log n + \log(n-1) + \cdots + \log 2 + \log 1 \in \Theta(n \log n)$

(We will define Θ soon.)

Probability:

- $E[X]$ is the expected value of X.
- $E[aX] = aE[X]$, $E[X + Y] = E[X] + E[Y]$ (linearity of expectation)

Useful sums

Arithmetic sequence:
$\sum_{i=0}^{n-1} i = ???$

Geometric sequence:
$\sum_{i=0}^{n-1} 2^i = ???$

Harmonic sequence:
$\sum_{i=1}^{n} \frac{1}{i} = ???$

A few more:

$\sum_{i=1}^{n} \frac{i}{2^i} = ???$

$\sum_{i=1}^{n} i^k = ???$

Useful sums

Arithmetic sequence:

$\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2}$ $\qquad$ $\sum_{i=0}^{n-1}(a+di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2) \quad \text{if } d \neq 0.$

Geometric sequence:

$\sum_{i=0}^{n-1} 2^i = 2^n - 1$ $\qquad$ $$\sum_{i=0}^{n-1} a\, r^i = \begin{cases} a\dfrac{r^n-1}{r-1} & \in \Theta(r^n) \quad \text{if } r > 1 \\ na & \in \Theta(n) \quad \text{if } r = 1 \\ a\dfrac{1-r^n}{1-r} & \in \Theta(1) \quad \text{if } 0 < r < 1. \end{cases}$$

Harmonic sequence:

$\sum_{i=1}^{n} \frac{1}{i} = ???$ $\qquad$ $H_n := \sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + o(1) \in \Theta(\log n)$

A few more:

$\sum_{i=1}^{n} \frac{i}{2^i} = ???$ $\qquad$ $\sum_{i=1}^{n} \frac{i}{2^i} \in \Theta(1)$

$\sum_{i=1}^{n} i^k = ???$ $\qquad$ $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1}) \quad \text{for } k \geq 0$

Outline

1. Introduction and Asymptotic Analysis
   - CS240 overview
   - Algorithm design
   - Analysis of algorithms I
   - Rules for asymptotic notation
   - Analysis of algorithms revisited

Algorithms and problems: Review

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

Instance: One possible input for the specified problem.

Algorithm: *Step-by-step process* (can be described in finite length) for carrying out a series of computations, given an arbitrary instance I.

Solving a problem: An Algorithm $\mathcal{A}$ *solves* a problem Π if, for every instance I of Π, $\mathcal{A}$ computes a valid output for the instance I in finite time.

Program: A program is an *implementation* of an algorithm using a specified computer language.

In this course, our emphasis is on algorithms (as opposed to programs or programming). We do not use any particular computer language to describe them.

Algorithms and programs

Pseudocode: communicate an algorithm to another person.

In contrast, a program communicates an algorithm to a computer.

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

- sometimes uses English descriptions, e.g. ‘swap’,
- omits obvious details, e.g. i is usually an integer
- has limited if any error detection, e.g. A is assumed initialized
- should be precise about exit-conditions, e.g. in loops
- should use good indentation and variable-names

Algorithms and programs

From problem Π to program that solves it:

1. Design an algorithm $\mathcal{A}$ that solves Π. → **Algorithm Design**
   A problem Π may have several algorithms.
2. Assess *correctness* and *efficiency* of each $\mathcal{A}$. → **Algorithm Analysis**
   Correctness → CS245 (here informal arguments are enough).
   Efficiency → later
3. If acceptable (correct and efficient), implement algorithm(s).
   For each algorithm, we can have several implementations.
4. If multiple acceptable algorithms/implementations, run experiments to determine best solution.

CS240 focuses on the first two steps.
The main point is to avoid implementing obviously-bad algorithms.

Outline

1. Introduction and Asymptotic Analysis
   - CS240 overview
   - Algorithm design
   - Analysis of algorithms I
   - Rules for asymptotic notation
   - Analysis of algorithms revisited

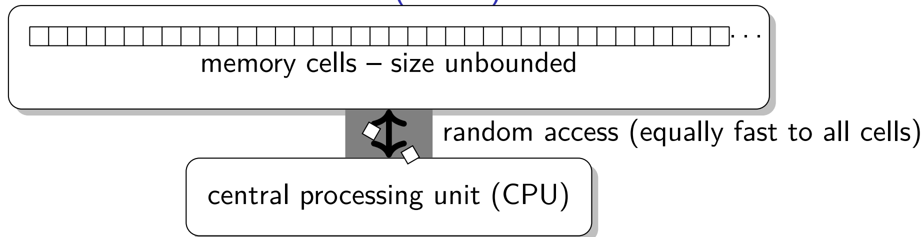
Efficiency of algorithms

What do we mean by 'efficiency'?

- In this course, we are primarily concerned with the *amount of time* a program takes to run. $\rightarrow$**Running Time**
- We also may be interested in the *amount of additional memory* the program requires. $\rightarrow$**Auxiliary space**
- The amount of time and (or memory) required by a program will usually **depend on the given problem instance**.

To properly define runtime and space, we need a **model** that formalizes real-life computers

Random access machine (RAM) model



- Each **memory cell** stores one (finite-length) datum, typically a number, character, or reference.
  Assumption: cells are big enough to hold the items that we store.
- Any **access to a memory location** takes constant time.
  (We will revisit this assumption late in the course.)
- Any **primitive operation** takes constant time.
  (read, write, add, subtract, multiply, divide, follow a reference, ...)
  Not primitive: $\sqrt{n}$, anything involving irrational numbers

These assumptions may not be valid for a “real” computer.

Running time and space

With this computer model, we can now formally define:

- **Size(I)** of instance I is the number of memory cells that I occupies.
- The **running time** $T(I)$ is the number of primitive operations we do on input I
- The **space** $S(I)$ is the maximum number of memory cells ever in use on input I

The real-life time and space are somewhat proportional to $T(I)$ and $S(I)$

Asymptotic analysis

Once we understand $T(I)$, we aggregate over all instances of size n, to get a function **T(n)**

Note: we can define $T(n)$ using the max, min or average

- we discuss this later
- let's assume max for the moment

We are interested in the **growth rate** of the function $T(n)$: what is its behaviour as size n gets large?

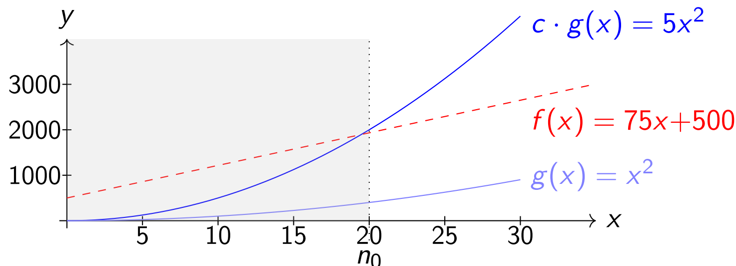
- **Example 1:** What is larger, $100n$ or $10n^2$?
- **Example 2 (Matrix multiplication, approximately):** What is larger: $4n^3$, $300n^{2.807}$, or $10^{67}n^{2.373}$?

To simplify comparisons, use **order notation** (big-O and friends).

Informally: ignore constants and lower order terms

Order notation

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



***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).

Order Notation: Example 1

In order to prove that $2n^2 + 3n + 11 \in O(n^2)$ **from first principles** (i.e., directly from the definition), we need to find c and n_0 such that the following condition is satisfied:

$$2n^2 + 3n + 11 \leq c\, n^2 \text{ for all } n \geq n_0.$$

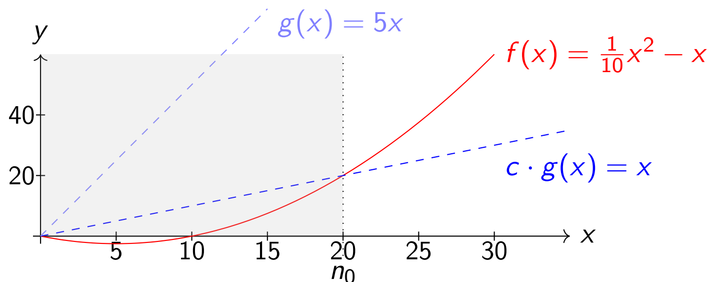
Many, but not all, choices of c and n_0 will work.

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: $f(x) = \frac{1}{10}x^2 - x$ and $g(x) = 5x$ (e.g. $c = \frac{1}{5}, n_0 = 20$)



Asymptotic lower bound

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

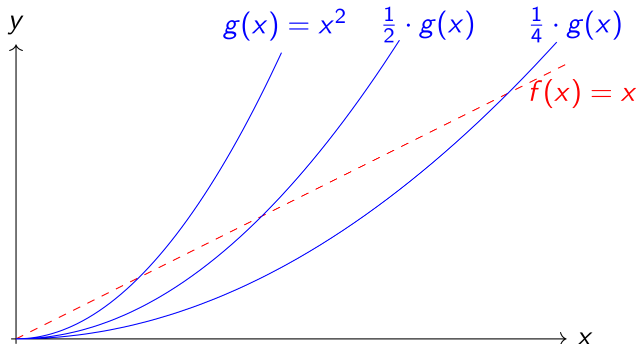
Commonly encountered growth rates in analysis of algorithms include the following:

- $\Theta(1)$ (*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.

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.

Order notation: Summary

.

O-notation: $f(x) \in O(g(x))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $|f(x)| \leq c\,|g(x)|$ for all $x \geq n_0$.

Ω-notation: $f(x) \in \Omega(g(x))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $c\,|g(x)| \leq |f(x)|$ for all $x \geq n_0$.

Θ-notation: $f(x) \in \Theta(g(x))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ such that $c_1\,|g(x)| \leq |f(x)| \leq c_2\,|g(x)|$ for all $x \geq n_0$.

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

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

Outline

1 Introduction and Asymptotic Analysis
- CS240 overview
- Algorithm design
- Analysis of algorithms I
- Rules for asymptotic notation
- Analysis of algorithms revisited

The limit rule

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

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

Then

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

The required limit can sometimes be computed using *l'Hôpital's rule*.

This gives *sufficient* (but not always necessary) conditions.

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 $d \geq 0$:

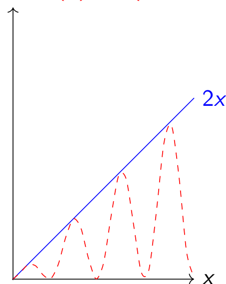
$$f(n) = c_d n^d + c_{d-1} n^{d-1} + \cdots + c_1 n + c_0$$

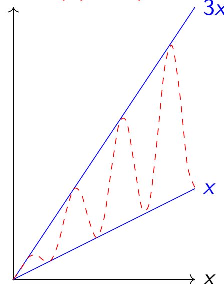
for some $c_d > 0$.

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

Example: Oscillating functions

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





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

Optional: algebra of order notations

Many rules are easily proved from first principle (exercise).

Identity rule: $f(n) \in \Theta(f(n))$

Transitivity:

- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ then $f(n) \in O(h(n))$.
- If $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n))$ then $f(n) \in \Omega(h(n))$.
- If $f(n) \in O(g(n))$ and $g(n) \in o(h(n))$ then $f(n) \in o(h(n))$.
- ...

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

- $f(n) + g(n) \in O(\max\{f(n), g(n)\})$
- $f(n) + g(n) \in \Omega(\max\{f(n), g(n)\})$

Key proof-ingredient: $\max\{f(n), g(n)\} \leq f(n)+g(n) \leq 2\max\{f(n), g(n)\}$

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

- $f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \notin \Omega(g(n))$ $\quad f(n), g(n) > 0$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \notin O(g(n))$ $\quad f(n), g(n) > 0$

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 may have allowed you to be sloppy here. CS240 does not.)
- Instead, when you do arithmetic, replace '$\Theta(f(n))$' by '$c \cdot f(n)$ for some constant $c > 0$'

  (That's still sloppy, but less dangerous.)
- There are some exceptions:
  - $f(n) = n^2 + \Theta(n)$ means "$f(n) = n^2 + g(n)$, with $g(n) \in \Theta(n)$"
    - nicer to read than "$n^2 + n + \log n$"
    - more precise about constants than "$\Theta(n^2)$"
  - We use this sparingly (e.g., last slide)

Outline

1. Introduction and Asymptotic Analysis
   - CS240 overview
   - Algorithm design
   - Analysis of algorithms I
   - Rules for asymptotic notation
   - Analysis of algorithms revisited

Techniques for run-time analysis

- Goal: Use asymptotic notation to simplify run-time analysis.

```
print-pairs(A, n)
1.  for i ← 0 to n − 1 do
2.      for j ← 0 to i − 1 do
3.          print 'the next pair is {A[i], A[j]}'
```

- Nice case: runtime of this algorithm depends only the *input size* n.
- Identify *primitive operations* that require $\Theta(1)$ time.
  Assume they require c time for some $c > 0$
- The complexity of a loop is expressed as the *sum* of the complexities of each iteration of the loop.
- Nested loops: start with the innermost loop and proceed outwards.
  This gives *nested summations*.

For *print-pairs*: runtime is $T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} c$.

Two strategies for runtime analysis

Strategy I: Use Θ-bounds *throughout the analysis* and obtain a Θ-bound for the complexity of the algorithm.
For *print-pairs*:

Strategy II: Prove a O-bound and a *matching* Ω-bound *separately*.
Use upper bounds (for O) and lower bounds (for Ω) early and frequently.
This may be easier because upper/lower bounds are easier to sum.
For *print-pairs*:

Complexity of algorithms

- 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 ← 1; i < n; i++) do
2.       for (j ← 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.

Could study this value for the **worst**-possible, **best**-possible and **average** instance I.

Complexity of algorithms

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

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

(that's the default choice)

Best-case: The *best-case runtime* of an algorithm $\mathcal{A}$ is the function $\mathbb{Z}^+ \to \mathbb{R}$ mapping n to the *shortest* runtime for any instance of size n:

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

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

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

O-notation and complexity of algorithms

Goal in cs240: for a problem, find an algorithm that solves it and whose Θ-bound on the *worst-case runtime* is the smallest we can.

There are various pitfalls.

- **Example:** Suppose algorithm $\mathcal{A}_1$ has worst-case run-time $O(n^3)$ and algorithm $\mathcal{A}_2$ has worst-case run-time $O(n^2)$, and both solve the same problem. Is $\mathcal{A}_2$ more efficient?

  No! O-notation is an upper bound. $\mathcal{A}_1$ may well have worst-case run-time $O(n)$. We should always give Θ-bounds.

- **Example:** Suppose the run-times are $\Theta(n^3)$ and $\Theta(n^2)$, respectively. We consider $\mathcal{A}_2$ to be better. But is it always more efficient?

  No! The worst-case run-time of $\mathcal{A}_1$ may only be achieved on some instances. Possibly $\mathcal{A}_1$ is better on most instances.

  Also, the hidden constants may be so large that $\mathcal{A}_1$ is better on all but unrealistically big n.

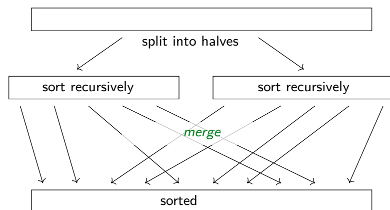
Explaining the solution of a problem

To give an algorithm that 'solves a problem', we usually do four steps. 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
 1.  if (n ≤ 1) then return
 2.  else
 3.       m = ⌊(n − 1)/2⌋
 4.       merge-sort(A[0..m], m + 1)
 5.       merge-sort(A[m + 1..n−1], n − m − 1)
 6.       merge(A, 0, m, n−1)
```

Two tricks to reduce constant in the run-time and auxiliary space:

- Do not pass array A by value, instead indicate the range of the array that needs to be sorted.
- *merge* uses an auxiliary array S. Allocate this only *once*.

Explaining the solution of a problem

Step 2: Give pseudo-code or detailed description.

```
merge-sort(A, n, ℓ ← 0, r ← n − 1, S ← NULL)
A: array of size n, 0 ≤ ℓ ≤ r ≤ n − 1
 1.  if S is NULL then initialize it as array S[0..n − 1]
 2.  if (r ≤ ℓ) then
 3.       return
 4.  else
 5.       m = ⌊(r + ℓ)/2⌋
 6.       merge-sort(A, n, ℓ, m, S)
 7.       merge-sort(A, n, m + 1, r, S)
 8.       merge(A, ℓ, m, r, S)
```

- This would be much better for an efficient implementation.
- But the idea is much harder to understand.
- CS240 pseudocode will often prefer clarity over improved constants.

Explaining the solution of a problem

Step 2: Give pseudo-code or detailed description.

Idea for merging: Always extract from each sub-array the value that is smaller and append it to the output.

merge(A, ℓ, m, r, S)
A is an array, $A[\ell..m]$ is sorted, $A[m+1..r]$ is sorted
1. copy $A[\ell..r]$ into $S[\ell..r]$
2. $(i_L, i_R) \leftarrow (\ell, m+1)$; `// start-indices of subarrays`
3. **for** $(k \leftarrow \ell; k \leq r; k{+}{+})$ **do** `// fill-index for result`
4. **if** $(i_L > m)$ $A[k] \leftarrow S[i_R{+}{+}]$
5. **else if** $(i_R > r)$ $A[k] \leftarrow S[i_L{+}{+}]$
6. **else if** $(S[i_L] \leq S[i_R])$ $A[k] \leftarrow S[i_L{+}{+}]$
7. **else** $A[k] \leftarrow S[i_R{+}{+}]$

Analysis of *merge-sort*

Step 3: Argue correctness.

- Can 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 runtime then becomes a recursive function.

The clean recurrences

Observation: for any array A,

- $T(A) = T(A_{\text{left}}) + T(A_{\text{right}}) + \text{time}(\textit{merge})$
- *merge* in size n takes between $c_1 n$ and $c_2 n$ operations, for some constants c_1, c_2.

Let $T^{\text{worst}}(n)$ and $T^{\text{best}}(n)$ be the worst-case and best-case runtime for *merge-sort* on an array of length n.

Easy inequalities:

- $T^{\text{worst}}(n) \leq T^{\text{worst}}(\lceil \frac{n}{2} \rceil) + T^{\text{worst}}(\lfloor \frac{n}{2} \rfloor) + c_2 n$
- $T^{\text{best}}(n) \geq T^{\text{best}}(\lceil \frac{n}{2} \rceil) + T^{\text{best}}(\lfloor \frac{n}{2} \rfloor) + c_1 n$

Proof: let A^{worst} be an array that gives the worst case in size n. Then ...

Simplifying the recurrences

Consider the **simplified recurrence**

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

and the **sloppy recurrence** (floors and ceilings removed):

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

When n is a power of 2, these recurrences are *identical* and can be solved by various methods. E.g., prove by induction that $T^{\mathrm{slop}}(n) = cn\log(2n)$ for these n.

Can prove: $T(n) \in \Theta(n \log n)$ by analyzing the simplified recurrence

Summary and conclusion

- Setup the exact recurrences:

$$T^{\text{worst}}(n) \le T^{\text{worst}}\left(\left\lceil \frac{n}{2} \right\rceil\right) + T^{\text{worst}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c_2 n$$

$$T^{\text{best}}(n) \ge T^{\text{best}}\left(\left\lceil \frac{n}{2} \right\rceil\right) + T^{\text{best}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c_1 n$$

- Solve the sloppy recurrence: $\Theta(n \log n)$ for n power of 2
- *Can prove:*

$$T^{\text{worst}}(n) \in O(n \log n), \quad T^{\text{best}}(n) \in \Omega(n \log n)$$

- but $T^{\text{best}}(n) \le T^{\text{worst}}(n)$ so both are $\Theta(n \log n)$.

This course: setup the sloppy recurrence and solve it (for nice n)

Some recurrence relations

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

- Once you know the result, it is (usually) easy to prove by induction
- If lhs is tight ($=$ sign instead of $\le$, Θ), rhs is tight (Θ)
- Many more recursions, and some methods to find the result, in CS341.

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