# 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 2025

*version 2025-05-05 13:27*

# Outline

# Outline

# 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:** Digital Music Collection, English Dictionary

  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 ≈ 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 $= \log_e(x)$, $\frac{d}{dx} \ln x = \frac{1}{x}$

**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 $\Theta$ 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

# Algorithms and problems: Review

Let us clarify a few more terms:

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

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 $\Pi$ if, for every instance $I$ of $\Pi$, $\mathcal{A}$ 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. Design many!

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

# 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.
- So we express the time or memory requirements as a mathematical function of the instances (e.g. $T(I)$)
- But then aggregate over all instances $\mathcal{I}_n$ of size $n$ (e.g. $T(n)$).
- Do we take max, min, avg? ($\rightarrow$ later)

# Measuring efficiency of algorithms

What do we count as running time/space usage of an algorithm?

First option: *experimental studies*

- Write a program implementing the algorithm.
- Run the program with inputs of varying size and composition and measure time and space.
- Plot/compare the results.

There are numerous shortcomings:

- Implementation may be complicated/costly.
- Outcomes are affected by many factors: *hardware* (processor, memory), *software environment* (OS, compiler, programming language), and *human factors* (programmer).
- We cannot test all instances; what are good *sample inputs*?
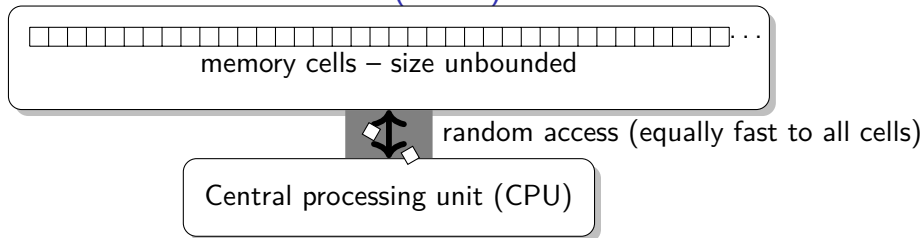
# Running time of algorithms

Better: theoretical analysis:

- Does not require implementing the algorithm (we work on *pseudo-code*).
- Is independent of the hardware/software environment (we work on an *idealized computer model*).
- Takes into account all input instances.

This is the approach taken in CS240.

We use experimental results only if theoretical analysis yields no useful results for deciding between multiple algorithms.

# Random access machine (RAM) model



memory cells – size unbounded

random access (equally fast to all cells)

Central processing unit (CPU)

- 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.
  (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:

- The **running time** is the number of memory accesses plus the number of primitive operations.
- The **space** is the maximum number of memory cells ever in use.
- **Size($I$)** of instance $I$ is the number of memory cells that $I$ occupies.

The real-life time and space is proportional to this.

We compare algorithms by considering the **growth rate**: What is the behaviour of algorithms as size $n$ gets large?

- **Example 1:** What is larger, $100n$ or $10n^2$?

# Running time and space

With this computer model, we can now formally define:

- The **running time** is the number of memory accesses plus the number of primitive operations.
- The **space** is the maximum number of memory cells ever in use.
- **Size($I$)** of instance $I$ is the number of memory cells that $I$ occupies.

The real-life time and space is proportional to this.

We compare algorithms by considering the **growth rate**: What is the behaviour of algorithms 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 $10^{67}n^{2.373}$?

# Running time and space

With this computer model, we can now formally define:

- The **running time** is the number of memory accesses plus the number of primitive operations.
- The **space** is the maximum number of memory cells ever in use.
- **Size($I$)** of instance $I$ is the number of memory cells that $I$ occupies.

The real-life time and space is proportional to this.

We compare algorithms by considering the **growth rate**: What is the behaviour of algorithms as size $n$ gets large?
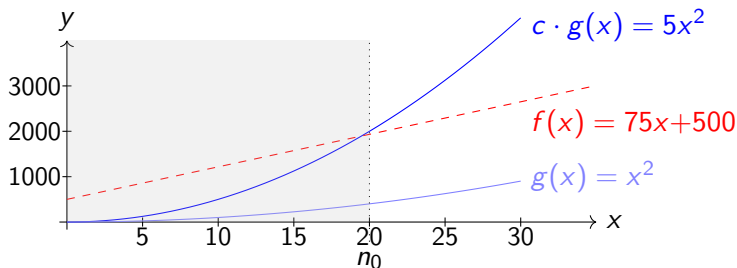
- **Example 1:** What is larger, $100n$ or $10n^2$?
- **Example 2 (Matrix multiplication, approximately):** What is larger: $4n^3$, $300n^{2.807}$, or $10^{67}n^{2.373}$?

To simplify comparisons, use **order notation** (big-$O$ and friends). Informally: ignore constants and lower order terms

# Order notation

Study relationships between *functions*.

**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:

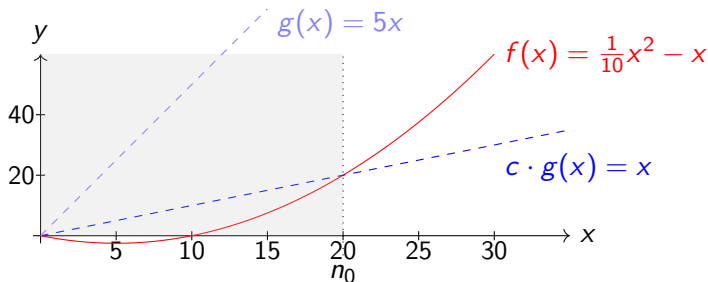$$2n^2 + 3n + 11 \leq c\, n^2 \text{ for all } n \geq n_0.$$

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.

$\Omega$-**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

$\Theta$-**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'?

# Growth rates and running time

It is interesting to see how the running time is affected when the size of the problem instance *doubles* (i.e., $n \to 2n$).

- constant complexity: $T(n) = c$
- logarithmic complexity: $T(n) = c \log n$
- linear complexity: $T(n) = cn$
- linearithmic $\Theta(n \log n)$: $T(n) = c\, n \log n$
- quadratic complexity: $T(n) = c\, n^2$
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c\, 2^n$

# Growth rates and running time

It is interesting to see how the running time is affected when the size of the problem instance *doubles* (i.e., $n \to 2n$).

- constant complexity: $T(n) = c$ $\rightsquigarrow T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$
- linear complexity: $T(n) = cn$
- linearithmic $\Theta(n \log n)$: $T(n) = c \, n \log n$
- quadratic complexity: $T(n) = c \, n^2$
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c \, 2^n$

# Growth rates and running time

It is interesting to see how the running time is affected when the size of the problem instance *doubles* (i.e., $n \to 2n$).

- constant complexity: $T(n) = c$ $\rightsquigarrow T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$ $\rightsquigarrow T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$
- linearithmic $\Theta(n \log n)$: $T(n) = c \, n \log n$
- quadratic complexity: $T(n) = c \, n^2$
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c \, 2^n$

# Growth rates and running time

It is interesting to see how the running time is affected when the size of the problem instance *doubles* (i.e., $n \to 2n$).

- constant complexity: $T(n) = c$      $\rightsquigarrow T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$      $\rightsquigarrow T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$      $\rightsquigarrow T(2n) = 2T(n)$.
- linearithmic $\Theta(n \log n)$: $T(n) = c\, n \log n$
- quadratic complexity: $T(n) = c\, n^2$
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c\, 2^n$

# Growth rates and running time

It is interesting to see how the running time is affected when the size of the problem instance *doubles* (i.e., $n \to 2n$).

- constant complexity: $T(n) = c \qquad \rightsquigarrow T(2n) = c.$
- logarithmic complexity: $T(n) = c \log n \qquad \rightsquigarrow T(2n) = T(n) + c.$
- linear complexity: $T(n) = cn \qquad \rightsquigarrow T(2n) = 2T(n).$
- linearithmic $\Theta(n \log n)$: $T(n) = c\, n \log n \quad \rightsquigarrow T(2n) = 2T(n) + 2cn.$
- quadratic complexity: $T(n) = c\, n^2$
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c\, 2^n$

# Growth rates and running time

It is interesting to see how the running time is affected when the size of the problem instance *doubles* (i.e., $n \to 2n$).

- constant complexity: $T(n) = c \quad\quad\quad \rightsquigarrow T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n \quad \rightsquigarrow T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn \quad\quad\quad \rightsquigarrow T(2n) = 2T(n)$.
- linearithmic $\Theta(n \log n)$: $T(n) = c\, n \log n \quad \rightsquigarrow T(2n) = 2T(n) + 2cn$.
- quadratic complexity: $T(n) = c\, n^2 \quad\quad\quad \rightsquigarrow T(2n) = 4T(n)$.
- cubic complexity: $T(n) = cn^3$
- exponential complexity: $T(n) = c\, 2^n$

# Growth rates and running time

It is interesting to see how the running time is affected when the size of the problem instance *doubles* (i.e., $n \to 2n$).

- constant complexity: $T(n) = c$ $\rightsquigarrow T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$ $\rightsquigarrow T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$ $\rightsquigarrow T(2n) = 2T(n)$.
- linearithmic $\Theta(n \log n)$: $T(n) = c\, n \log n$ $\rightsquigarrow T(2n) = 2T(n) + 2cn$.
- quadratic complexity: $T(n) = c\, n^2$ $\rightsquigarrow T(2n) = 4T(n)$.
- cubic complexity: $T(n) = cn^3$ $\rightsquigarrow T(2n) = 8T(n)$.
- exponential complexity: $T(n) = c\, 2^n$

# Growth rates and running time

It is interesting to see how the running time is affected when the size of the problem instance *doubles* (i.e., $n \rightarrow 2n$).

- constant complexity: $T(n) = c$ $\rightsquigarrow T(2n) = c$.
- logarithmic complexity: $T(n) = c \log n$ $\rightsquigarrow T(2n) = T(n) + c$.
- linear complexity: $T(n) = cn$ $\rightsquigarrow T(2n) = 2T(n)$.
- linearithmic $\Theta(n \log n)$: $T(n) = c\, n \log n$ $\rightsquigarrow T(2n) = 2T(n) + 2cn$.
- quadratic complexity: $T(n) = c\, n^2$ $\rightsquigarrow T(2n) = 4T(n)$.
- cubic complexity: $T(n) = cn^3$ $\rightsquigarrow T(2n) = 8T(n)$.
- exponential complexity: $T(n) = c\, 2^n$ $\rightsquigarrow T(2n) = (T(n))^2/c$.

# Strictly smaller asymptotic bounds

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



$g(x) = x^2$  $\frac{1}{2} \cdot g(x)$  $\frac{1}{4} \cdot g(x)$

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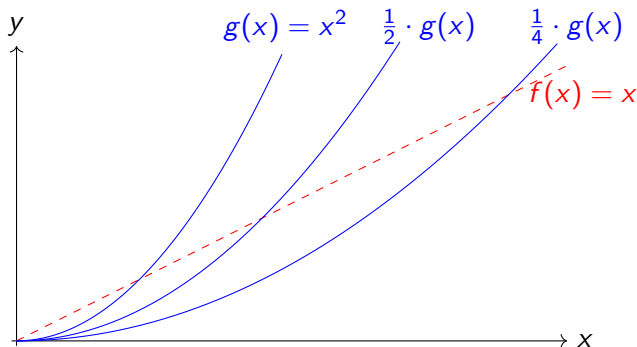
$o$-**notation:** $f(x) \in o(g(x))$ ($f$ is *asymptotically strictly smaller* than $g$) if for all constants $c > 0$, there exists a constant $n_0 \geq 0$ such that $|f(x)| \leq c\,|g(x)|$ for all $x \geq n_0$.

# 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).

$\omega$-**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 $x \geq n_0$.

$\Omega$-**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 $x \geq n_0$.

$\Theta$-**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 $x \geq n_0$.

$\omega$-**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

# 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

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

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

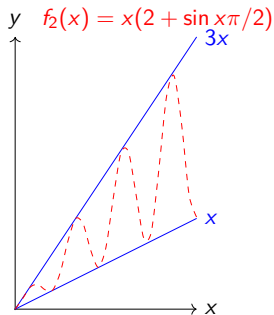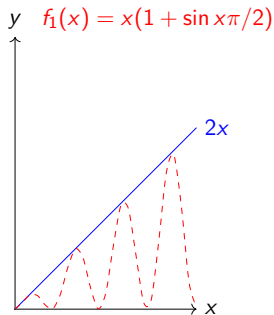Let $f(n)$ be a polynomial of degree $d \geq 0$:

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

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.

## 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 $f(n) \in \Omega(g(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \notin \Omega(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \notin O(g(n))$

**Example:** Fill the following table with TRUE or FALSE:

| | | Is $f(n) \in \ldots (g(n))$? | | | |
|---|---|---|---|---|---|
| $f(n)$ | $g(n)$ | $o$ | $O$ | $\Omega$ | $\omega$ |
| $\log n$ | $\sqrt{n}$ | | | | |

## 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.)

- 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.)

## 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.)

- 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 limited 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

# Techniques for run-time analysis

- Goal: Use asymptotic notation to simplify run-time analysis.
- Running time of an algorithm depends on the *input size n*.

  ---

  *print-pairs*$(A, n)$
  1. **for** $i \leftarrow 0$ **to** $n - 1$ **do**
  2.      **for** $j \leftarrow 0$ **to** $i - 1$ **do**
  3.          print 'the next pair is $\{A[i], A[j]\}$'

  ---

- Identify *primitive operations* that require $\Theta(1)$ time.
  (For doing arithmetic, 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*.

# Techniques for run-time analysis

- Goal: Use asymptotic notation to simplify run-time analysis.
- Running time of an algorithm depends on the *input size n*.

---

*print-pairs*$(A, n)$
1.  **for** $i \leftarrow 0$ **to** $n - 1$ **do**
2.      **for** $j \leftarrow 0$ **to** $i - 1$ **do**
3.          print 'the next pair is $\{A[i], A[j]\}$'

---

- Identify *primitive operations* that require $\Theta(1)$ time.
  (For doing arithmetic, 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*: The run-time is $\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} c$.

# Techniques for run-time analysis

Two general strategies are as follows.

**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 '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 $n$:

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

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

To prove a lower bound on the worst-case run-time: Pick one especially bad family of examples, and bound their run-time (using $\Omega$-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 $n$:

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

# O-notation and complexity of algorithms

Goal in cs240: For a problem, find an algorithm that solves it and whose tight bound on the worst-case running time has the smallest growth rate.

There are various pitfalls.

# O-notation and complexity of algorithms

Goal in cs240: For a problem, find an algorithm that solves it and whose tight bound on the worst-case running time has the smallest growth rate.

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?

# O-notation and complexity of algorithms

Goal in cs240: For a problem, find an algorithm that solves it and whose tight bound on the worst-case running time has the smallest growth rate.

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 $\Theta$-bounds.

# O-notation and complexity of algorithms

Goal in cs240: For a problem, find an algorithm that solves it and whose tight bound on the worst-case running time has the smallest growth rate.

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 $\Theta$-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?

# O-notation and complexity of algorithms

Goal in cs240: For a problem, find an algorithm that solves it and whose tight bound on the worst-case running time has the smallest growth rate.

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 $\Theta$-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.

# O-notation and complexity of algorithms

Goal in cs240: For a problem, find an algorithm that solves it and whose tight bound on the worst-case running time has the smallest growth rate.

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 $\Theta$-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 $n$.
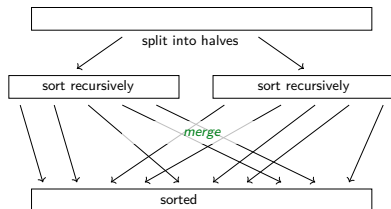
# Explaining the solution of a problem

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 ← NULL)
A is an array, A[ℓ..m] is sorted, A[m + 1..r] is sorted
1.  if S is NULL then initialize it with same size as A        // tmp-array
2.  copy A[ℓ..r] into S[ℓ..r]
3.  (i_L, i_R) ← (ℓ, m + 1);              // start-indices of subarrays
4.  for (k ← ℓ; k ≤ r; k++) do            // fill-index for result
5.      if (i_L > m) A[k] ← S[i_R++]
6.      else if (i_R > r) A[k] ← S[i_L++]
7.      else if (S[i_L] ≤ S[i_R]) A[k] ← S[i_L++]
8.      else A[k] ← 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 run-time then becomes a recursive function.

# The clean recurrences

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

**Observation:**

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

**Easy consequences:**

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

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

## Simplifying the recurrences

Consider the **simplified recurrence**

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

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

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

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:

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

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

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

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

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

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

# Order notation and arithmetic revisited

**Recall:** You should not intermix order notation and arithmetic.

- You should know what you're doing if you write $O(n) + O(n) = O(n)$.
- Example: What is wrong with the following proof?

# Order notation and arithmetic revisited

**Recall:** You should not intermix order notation and arithmetic.

- You should know what you're doing if you write $O(n) + O(n) = O(n)$.
- Example: What is wrong with the following proof?

**Claim (false!)**: If $T(n) = \begin{cases} 2\,T(\frac{n}{2}) + cn & \text{if } n > 1 \\ c & \text{if } n = 1. \end{cases}$ then $T(n) \in O(n)$.

"**Proof**": Use induction on $n$.

- In the base case ($n = 1$) we have $T(n) = c \in O(1) = O(n)$.
- Assume the claim holds for all $n'$ with $n' < n$.
- Step: We have

$$T(n) = 2\,T(\tfrac{n}{2}) + cn \stackrel{IH}{\in} 2O(\tfrac{n}{2}) + O(n) = O(n) + O(n) = O(n)$$

# 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
- If lhs is tight (= sign instead of $\leq$, $\Theta$), rhs is tight ($\Theta$)
- Many more recursions, and some methods to find the result, in CS341.

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