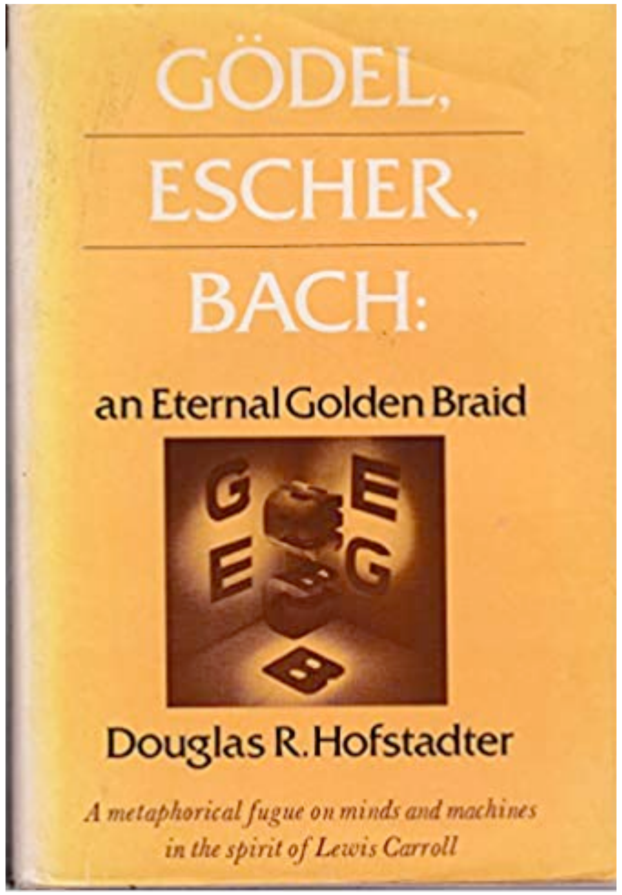
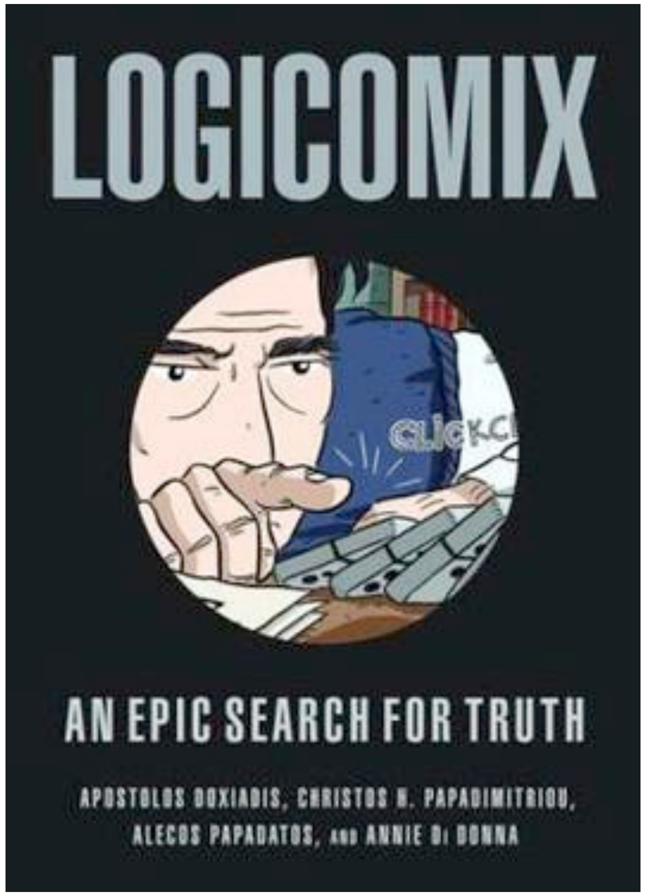
CSC 226
Algorithms & Data Structures II

Nishant Mehta

Lecture 1





The biggest difference between time and space is that you can't reuse time.

—Merrick Furst

Definition of Algorithm

- An **Algorithm** is a sequence of unambiguous instructions for solving a problem for obtaining the desired output for any legitimate input in a finite amount of time.

  (Levitin, *Introduction to the Design & Analysis of Algorithms*)

Definition of Algorithm

- An **Algorithm** is a sequence of unambiguous instructions for solving a problem for obtaining the desired output for any legitimate input in a finite amount of time.

  (Levitin, *Introduction to the Design & Analysis of Algorithms*)

- It really does have to be unambiguous
- Care has to be taken in specifying the range of inputs
- There can be different ways to implement an algorithm
- The same problem might be solvable by very different algorithms, and these algorithms can have very different efficiencies.

Example: Matrix-chain multiplication

- Suppose you are given a chain of matrices $A_1, A_2, \ldots, A_n$ and want to compute the product $A_1 A_2 \cdots A_n$

- Is $A_1 A_2 \cdots A_n$ an algorithm?

Example: Matrix-chain multiplication

- Suppose you are given a chain of matrices $A_1, A_2, \ldots, A_n$ and want to compute the product $A_1 A_2 \cdots A_n$

- Is $A_1 A_2 \cdots A_n$ an algorithm?

- Consider $n = 3$ with the matrices having dimensions:

  3×500, 500×2, and 2×2000

Example: Matrix-chain multiplication

- Suppose you are given a chain of matrices $A_1, A_2, \ldots, A_n$ and want to compute the product $A_1 A_2 \cdots A_n$
- Is $A_1 A_2 \cdots A_n$ an algorithm?
- Consider $n = 3$ with the matrices having dimensions:

  3×500, 500×2, and 2×2000
- Order of multiplication matters!

Complexity

- Time Complexity: How fast does the algorithm run?
- Space Complexity: How much (extra) space does the algorithm require?
  - Extra space means space in excess of the input
  - Time complexity typically is lower bounded by space complexity. Why?

Two Types of Analysis

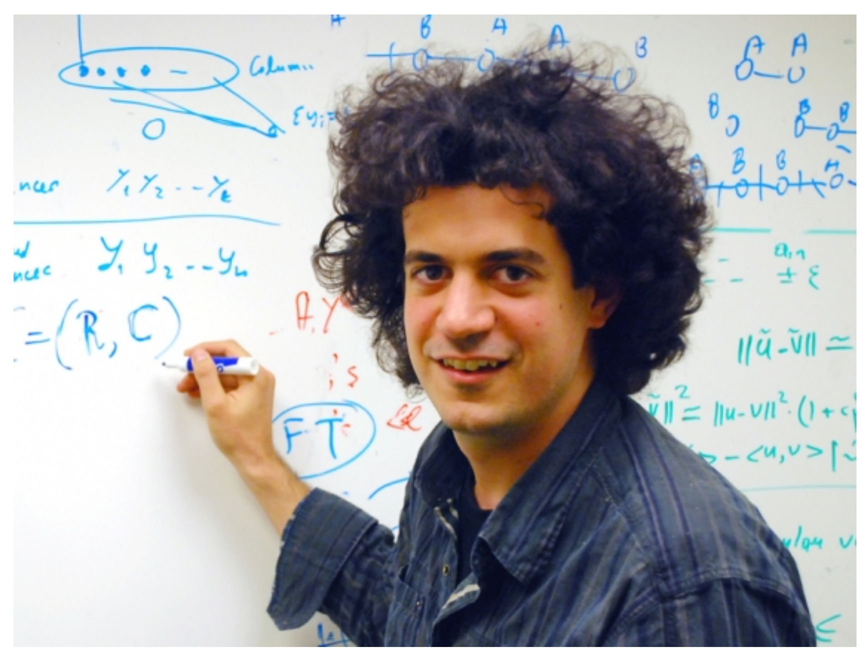
(1) The Empirical Method: “just run it and see what happens”

- Complexity measure: number of clock cycles
- Method: Instrumentation and Profiling
    - Closer to software engineering; covered in SENG 265

Two Types of Analysis

(2) The Theoretical Method: “hypothetically, how many primitive operations would this perform ***if*** I ran it?”

- Complexity measure: number of primitive operations
- Method: Math and Theoretical Computer Science
    - Derive upper and lower bounds on complexity



Two Types of Analysis

	Empirical Method	Theoretical Method
Good	More precise comparison for typical inputs and particular machine	Consider all possible inputs Compares algorithms in an architecture-agnostic way No implementation required
Bad		
Ugly		

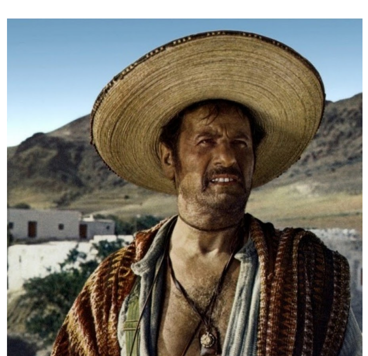
Two Types of Analysis

	Empirical Method	Theoretical Method
Good	More precise comparison for typical inputs and particular machine	Consider all possible inputs Compares algorithms in an architecture-agnostic way No implementation required
Bad	Limited by the set of inputs used Hard to identify good set of inputs Can only compare algorithms on the same machine Requires implementation	May be too pessimistic if one considers worst-case inputs Might be hard to analyze algorithms
Ugly		

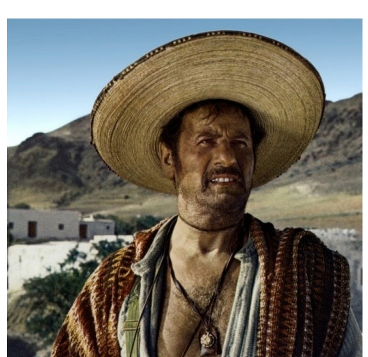
Two Types of Analysis

	Empirical Method	Theoretical Method
Good	More precise comparison for typical inputs and particular machine	Consider all possible inputs Compares algorithms in an architecture-agnostic way No implementation required
Bad	Limited by the set of inputs used Hard to identify good set of inputs Can only compare algorithms on the same machine Requires implementation	May be too pessimistic if one considers worst-case inputs Might be hard to analyze algorithms
Ugly		

Two Types of Analysis

	Empirical Method	Theoretical Method
Good	More precise comparison for typical inputs and particular machine	Consider all possible inputs Compares algorithms in an architecture-agnostic way No implementation required
Bad	Limited by the set of inputs used Hard to identify good set of inputs Can only compare algorithms on the same machine Requires implementation	May be too pessimistic if one considers worst-case inputs average-case analysis! Might be hard to analyze algorithms this course can help!
Ugly		

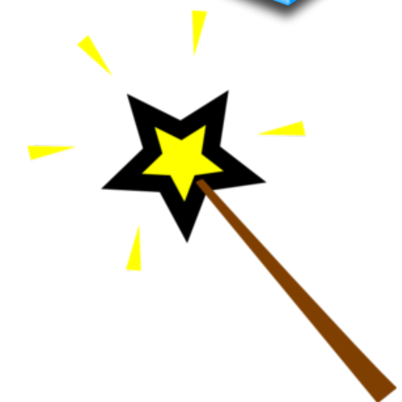
Two Types of Analysis

	Empirical Method	Theoretical Method
Good	More precise comparison for typical inputs and particular machine	Consider all possible inputs Compares algorithms in an architecture-agnostic way No implementation required
Bad	Limited by the set of inputs used Hard to identify good set of inputs Can only compare algorithms on the same machine Requires implementation	May be too pessimistic if one considers worst-case inputs **average-case analysis!** Might be hard to analyze algorithms **this course can help!**
Ugly		

Time complexity analysis

- Complexity as a function of input size
- Measured in terms of number of *primitive operations*
- Three main kinds: worst-case, best-case, average case
- Abstracting to asymptotic behavior/order of growth
- For recursive analysis, use the master theorem (sometimes)

Two wands problem



- **Input:** n boxes, where boxes $1, \ldots, i$ contain pearls, and boxes $i + 1, \ldots, n$ are empty, for some i
- **Output:** i, where i is the index of the rightmost box containing a pearl
- **Model of Computation:** At a cost of 1, a wand taps a box and reveals if it is empty or not. If empty, the wand disappears.

Can this problem be solved using two wands with $o(n)$ worst-case cost?

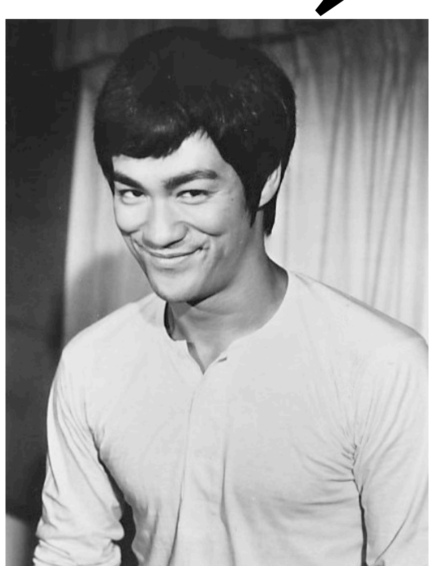
Two wand problem

- What does a solution look like?
- Need to give an algorithm, along with:
  - Proof of correctness: does it correctly identify i ?
  - Cost analysis. Is the number of boxes tapped $o(n)$?

Two wand problem

- What does a solution look like?
- Need to give an algorithm, along with:
  - Proof of correctness: does it correctly identify i ?
  - Cost analysis. Is the number of boxes tapped $o(n)$?

But what does $o(n)$ mean?

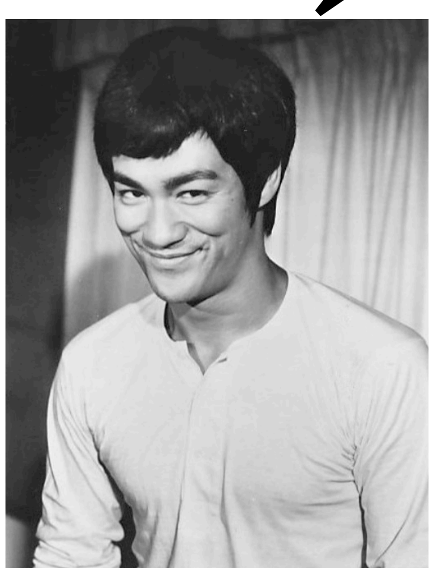


Two wand problem

- What does a solution look like?
- Need to give an algorithm, along with:
    - Proof of correctness: does it correctly identify i ?
    - Cost analysis. Is the number of boxes tapped $o(n)$?

But what does $o(n)$ mean?

Patience, Bruce.
We must review big-O notation…



Asymptotic notation

- Big-O $O(g(n))$
- Big-Omega $\Omega(g(n))$
- Big-Theta $\Theta(g(n))$
- Less commonly used (but still important!)
    - Little-o $o(g(n))$
    - Little-omega $\omega(g(n))$

Big-O notation

- Let $f: \mathbb{N} \to \mathbb{R}$, $g: \mathbb{N} \to \mathbb{R}$

- We say that f is $O(g(n))$ if, for some $c > 0$ and $n_0 > 0$, for all $n \geq n_0$, it holds that:

$$f(n) \leq cg(n)$$

- "For all n 'big enough' and for some c 'big enough', $f(n)$ is at most a constant c times $g(n)$"

Examples of Big-O

$$f(n) = n^4 + 7n^2 + 3$$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$

$f(n) = O(n^4)$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$ $\qquad$ $f(n) = O(n^4)$

$f(n) = 2 \log n$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$ $\quad$ $f(n) = O(n^4)$

$f(n) = 2\log n$ $\quad$ $f(n) = O(\log n)$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$ $f(n) = O(n^4)$

$f(n) = 2 \log n$ $f(n) = O(\log n)$

$f(n) = \log(n^4)$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$ $\quad$ $f(n) = O(n^4)$

$f(n) = 2\log n$ $\quad$ $f(n) = O(\log n)$

$f(n) = \log(n^4)$ $\quad$ $f(n) = O(\log n)$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$ $\quad$ $f(n) = O(n^4)$

$f(n) = 2\log n$ $\quad$ $f(n) = O(\log n)$

$f(n) = \log(n^4)$ $\quad$ $f(n) = O(\log n)$

$f(n) = 3000$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$ $\quad$ $f(n) = O(n^4)$

$f(n) = 2\log n$ $\quad$ $f(n) = O(\log n)$

$f(n) = \log(n^4)$ $\quad$ $f(n) = O(\log n)$

$f(n) = 3000$ $\quad$ $f(n) = O(1)$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$ $\quad$ $f(n) = O(n^4)$

$f(n) = 2\log n$ $\quad$ $f(n) = O(\log n)$

$f(n) = \log(n^4)$ $\quad$ $f(n) = O(\log n)$

$f(n) = 3000$ $\quad$ $f(n) = O(1)$

$f(n) = 4/n$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$	$f(n) = O(n^4)$
$f(n) = 2 \log n$	$f(n) = O(\log n)$
$f(n) = \log(n^4)$	$f(n) = O(\log n)$
$f(n) = 3000$	$f(n) = O(1)$
$f(n) = 4/n$	$f(n) = O(1/n)$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$ $\quad$ $f(n) = O(n^4)$

$f(n) = 2\log n$ $\quad$ $f(n) = O(\log n)$

$f(n) = \log(n^4)$ $\quad$ $f(n) = O(\log n)$

$f(n) = 3000$ $\quad$ $f(n) = O(1)$

$f(n) = 4/n$ $\quad$ $f(n) = O(1/n)$

$f(n) = \log n + \log\log n$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$ $\quad$ $f(n) = O(n^4)$

$f(n) = 2 \log n$ $\quad$ $f(n) = O(\log n)$

$f(n) = \log(n^4)$ $\quad$ $f(n) = O(\log n)$

$f(n) = 3000$ $\quad$ $f(n) = O(1)$

$f(n) = 4/n$ $\quad$ $f(n) = O(1/n)$

$f(n) = \log n + \log \log n$ $\quad$ $f(n) = O(\log n)$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$ $\qquad$ $f(n) = O(n^4)$

$f(n) = 2 \log n$ $\qquad$ $f(n) = O(\log n)$

$f(n) = \log(n^4)$ $\qquad$ $f(n) = O(\log n)$

$f(n) = 3000$ $\qquad$ $f(n) = O(1)$

$f(n) = 4/n$ $\qquad$ $f(n) = O(1/n)$

$f(n) = \log n + \log \log n$ $\qquad$ $f(n) = O(\log n)$

$f(n) = n(n \log n + 3 \log n)$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$ $\quad$ $f(n) = O(n^4)$

$f(n) = 2\log n$ $\quad$ $f(n) = O(\log n)$

$f(n) = \log(n^4)$ $\quad$ $f(n) = O(\log n)$

$f(n) = 3000$ $\quad$ $f(n) = O(1)$

$f(n) = 4/n$ $\quad$ $f(n) = O(1/n)$

$f(n) = \log n + \log\log n$ $\quad$ $f(n) = O(\log n)$

$f(n) = n(n\log n + 3\log n)$ $\quad$ $f(n) = O(n^2\log n)$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$ $\quad$ $f(n) = O(n^4)$

$f(n) = 2\log n$ $\quad$ $f(n) = O(\log n)$

$f(n) = \log(n^4)$ $\quad$ $f(n) = O(\log n)$

$f(n) = 3000$ $\quad$ $f(n) = O(1)$

$f(n) = 4/n$ $\quad$ $f(n) = O(1/n)$

$f(n) = \log n + \log\log n$ $\quad$ $f(n) = O(\log n)$

$f(n) = n(n\log n + 3\log n)$ $\quad$ $f(n) = O(n^2\log n)$

$f(n) = 2^{\log_2 n}$

Examples of Big-O

$f(n) = n^4 + 7n^2 + 3$	$f(n) = O(n^4)$
$f(n) = 2\log n$	$f(n) = O(\log n)$
$f(n) = \log(n^4)$	$f(n) = O(\log n)$
$f(n) = 3000$	$f(n) = O(1)$
$f(n) = 4/n$	$f(n) = O(1/n)$
$f(n) = \log n + \log\log n$	$f(n) = O(\log n)$
$f(n) = n(n\log n + 3\log n)$	$f(n) = O(n^2\log n)$
$f(n) = 2^{\log_2 n}$	$f(n) = O(n)$

Properties of Big-O

- Sum

  Suppose that $f(n) = O(a(n))$ and $g(n) = O(b(n))$

  Then $f(n) + g(n) = O(a(n) + b(n))$

- Product

  Suppose that $f(n) = O(a(n))$ and $g(n) = O(b(n))$

  Then $f(n) \cdot g(n) = O(a(n) \cdot b(n))$

- Multiplication by a constant

  Suppose that $f(n) = O(a(n))$

  Then, for any $c > 0$**,** $c \cdot f(n) = O(a(n))$

- Transitivity

  Suppose that $f(n) = O(g(n))$ **and** $g(n) = O(h(n))$

  Then $f(n) = O(h(n))$

Properties of Big-O

- Max degree

  Suppose that $f(n) = a_0 + a_1 n + \ldots + a_d n^d$

  Then $f(n) = O(n^d)$

- Polynomial is subexponential

  Let $d > 0$ **be arbitrary.**

  Then $n^d = O(a^n)$ **for all** $a > 1$

- Polylogarithmic is subpolynomial

  Let $d > 0$ **be arbitrary.**

  Then $(\log n)^d = O(n^r)$ **for all** $r > 0$

Proof that polylogarithmic is subpolynomial

To be shown: Is there some $c > 0$ such that for all large enough n, we have:

$$(\log n)^d \overset{??}{\leq} cn^r$$

$$\Updownarrow$$

$$\log n \overset{??}{\leq} c^{1/d} n^{r/d}$$

$$\Updownarrow$$

$$\log n \overset{??}{\leq} bn^k \qquad \text{for } b = c^{1/d} \text{ and } k = r/d$$

And we are done! By choosing c large enough, we can make b large enough such that the last inequality holds (since $\log(n)$ is $O(g(n))$ for any polynomial $g(n)$, including $g(n) = n^k$)

Common Examples of Big-O

	$O(1/n)$
Accessing min in a min-heap	$O(1)$
	$O(\log \log n)$
Search in a balanced binary tree	$O(\log n)$
	$O(\sqrt{n})$
(i) Median. (ii) Range-limited Radix sort	$O(n)$
Merge sort	$O(n \log n)$
Insertion sort	$O(n^2)$
	$O(2^n)$
Brute force sorting	$O(n!)$ or $O(n^n)$
	$O(2^{2^n})$

increasing complexity

Big-Omega notation

- Let $f : \mathbb{N} \to \mathbb{R}, g : \mathbb{N} \to \mathbb{R}$
- We say that f is $\Omega(g(n))$ if, for some $c > 0$ and $n_0 > 0$, for all $n \geq n_0$, it holds that:

$$f(n) \geq cg(n)$$

- "For all n 'big enough' and for some c 'small enough', $f(n)$ is at least a constant c times $g(n)$"
- Equivalently, f is $\Omega(g(n))$ if and only if g is $O(f(n))$

Big-Theta notation

- Let $f : \mathbb{N} \to \mathbb{R}$, $g : \mathbb{N} \to \mathbb{R}$

- We say that f is $\Theta(g(n))$ if $f = O(g(n))$ and $f = \Omega(g(n))$

- "For all n 'big enough', f and g grow at the same rate, i.e., there are constants $c_1, c_2 > 0$ such that:

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Little-o and little-omega

- Asymptotic dominance

- Less common in undergrad-level computer science, but they *do* come up in statistics, optimization, machine learning

- We say that f is $o(g(n))$ if, for all $\varepsilon > 0$, there is some $n_0 > 0$ such that, for all $n \geq n_0$, it holds that:

$$f(n) \leq \varepsilon g(n)$$

- f is $\omega(g(n))$ if and only if g is $o(f(n))$

Little-o and little-omega

- If g is non-zero for large enough n, then we can use shorter, calculus-based definitions:

$$f(n) \text{ is } o(g(n)) \text{ if } \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) \text{ is } \omega(g(n)) \text{ if } \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

- little-o: "the growth of f is nothing compared to the growth of g"

- little-omega: "the growth of f strictly dominates the growth of g"

Typical model of computation: RAM model

- Primitive operations (can be done in 1 time step):
    - Addition, Subtraction, Multiplication, Division, Exponentiation*, Boolean operations, Assignment, Array indexing, Function calls *when each operand fits in one word of storage*

- When using this model, we will implicitly assume that a word contains $\Theta(\log n)$ bits, for input size n. Why?

Typical model of computation: RAM model

- Primitive operations (can be done in 1 time step):
  - Addition, Subtraction, Multiplication, Division, Exponentiation*, Boolean operations, Assignment, Array indexing, Function calls *when each operand fits in one word of storage*

- When using this model, we will implicitly assume that a word contains $\Theta(\log n)$ bits, for input size n. Why?

- Does the code below run in ***polynomial time*** with respect to input n?

$x \leftarrow 2$

for $i = 1$ **to** n

$\quad x \leftarrow x^2$

Example

A: Assignment
C: Comparison
S: Subtraction
D: Division
I: array Indexing

Mean(x, n):

$\text{sum} \leftarrow 0$	1 A
For $j = 0$ **to** $n - 1$	$(n + 1) \cdot A + (n + 1) \cdot C + n \cdot S$
$\quad \text{sum} \leftarrow \text{sum} + x[j]$	$n \cdot (I + S + A)$
$\text{mean} \leftarrow \text{sum} / n$	$1 \cdot (A + D)$
return mean	

Example

A: Assignment
C: Comparison
S: Subtraction
D: Division
I: array Indexing

Mean(x, n):

sum $\leftarrow 0$	1 A
For $j = 0$ **to** $n - 1$	(n + 1) · A + (n + 1) · C + n · S
sum $\leftarrow$ sum $+ x[j]$	n · (I + S + A)
mean $\leftarrow$ sum / n	1 · (A + D)
return mean	

Complexity: (2A + 2S + C + I) · n + (3A + C + D) · 1

= O(n)

Example

A: Assignment
C: Comparison
S: Subtraction
D: Division
I: array Indexing

Mean(x, n):

sum $\leftarrow 0$	1 A
For $j = 0$ **to** $n - 1$	(n + 1) · A + (n + 1) · C + n · S
sum $\leftarrow$ sum $+ x[j]$	n · (I + S + A)
mean $\leftarrow$ sum $/ n$	1 · (A + D)
return mean	

Ignore! O(1)

Complexity: (2A + 2S + C + I) · n + (3A + C + D) · 1

= O(n)

Back to the two wands problem

- **Input:** n boxes, where boxes $1, \ldots, i$ contain pearls, and boxes $i+1, \ldots, n$ are empty, for some i
- **Output:** i, where i is the index of the rightmost box containing a pearl
- **Model of Computation:** At a cost of 1, a wand taps a box and reveals if it is empty or not. If empty, the wand disappears.

Can this problem be solved using two wands with $o(n)$ worst-case cost?

Some friends to remember From CSC 225

- Pseudocode, counting number of operations
- Recursion
- Proof by induction: review this ASAP if you need to
- Big-O analysis: review this ASAP if you need to
- Merge sort, Quicksort, Priority queues (heaps)
- Lower bounds for sorting
- Trees, Binary Search Trees, Balanced Binary Search Trees (e.g. red-black trees, 2-3 trees, AVL trees)
- Graph theory topics from CSC 225
- BFS, DFS, strong connectivity

Course Outline

Graph Algorithms & Graph Theory	Minimum Spanning Trees
	Introductory Graph Theory
	Shortest Path Algorithms
	Network Flow
Randomized Algorithms	Randomized Quickselect and Quicksort
	Hashing
More Algorithms	String Search Algorithms
	Greedy Algorithms
	Data Compression
	Dynamic Programming

Administrivia

Instructor: Nishant Mehta

Email: nmehta@uvic.ca

Office: ECS 608

Office hours (tentative):

Mondays 11:30am-12:30pm, Wednesdays 4pm–5pm

TAs: Ali Mortazavi, Chuan Zhang

Course webpage: http://web.uvic.ca/~nmehta/csc226_fall2025

Administrivia

Lectures, ECS 125

Mondays and Thursdays, 10am - 11:20am

Labs, ECS 258, Instructed by Ali and Chuan

Tuesdays	12:30pm - 1:20pm	(B01)
Wednesdays	2:30pm - 3:20pm	(B02)
Thursdays	12:30pm - 1:20pm	(B03)
	1:30pm - 2:20pm	(B04)

First lab will be Sep 16th–18th (in two weeks)

Please register for labs as soon as possible

Course webpage: http://web.uvic.ca/~nmehta/csc226_fall2025

Administrivia

- When emailing: always start your subject line with [CSC226]
- Any student who has registered in CSC 226 and does not have the required prerequisites and no waiver must drop the class. Otherwise: the student will be dropped and a prerequisite drop will be recorded on the student's record.
- Taking the course more than twice:
  - According to university rules, you must request (in writing) permission from the Chair of the Department and the Dean of the Faculty to be allowed to stay registered in the course. The letter should be submitted to Irene Statham, the CSC Undergraduate Advisor

Evaluation

- Points breakdown:
  - 5 Problem Sets - 6% each (total 30%)
  - Midterm - 25%
  - Final - 40%
  - Participation (via attending labs) - 5%
- **Even though the final only counts for 40%, you must pass the final to pass the course!!**
- The midterm exam will be in-class and is scheduled to take place on October 9th. The final exam will be 3 hours and scheduled by the registrar. For both exams, you cannot use any devices or material (no books or notes)

Problem Sets

- There will be 5 problem sets, each with about 3 problems
- **Late submissions won't be accepted:** With a valid excuse, the weight of the other problem sets will be increased
- **Collaborating:**
  - You may discuss problem sets at a high level (*you can discuss major ideas but not detailed solutions*), but your solutions must be written individually, from scratch, and all programming must be done individually (*you can't share written code*)
- **Cheating**
  - First time offense: zero on the entire problem set or exam
  - Second time offense: you fail the course

Policy on using LLMs / Generative AI

- It’s strictly prohibited to use generative AI (such as ChatGPT, Claude, Gemini, etc.) for any part of graded work (problem sets, labs, exams).
- To be clear, it’s prohibited to use AI-based tools for translation, formatting, typesetting, or as a brainstorming tool for any graded component of the course.
- It’s prohibited to provide problem set materials (such as questions from the problem sets) in any form to a generative AI tool.
- What about as a study aid? We strongly discourage the use of generative AI tools as a study tool for this course. Generative AI can often produce meaningless or contradictory information. As a result, when learning new information, you may be unable to verify the correctness of material generated by generative AI.

Textbooks

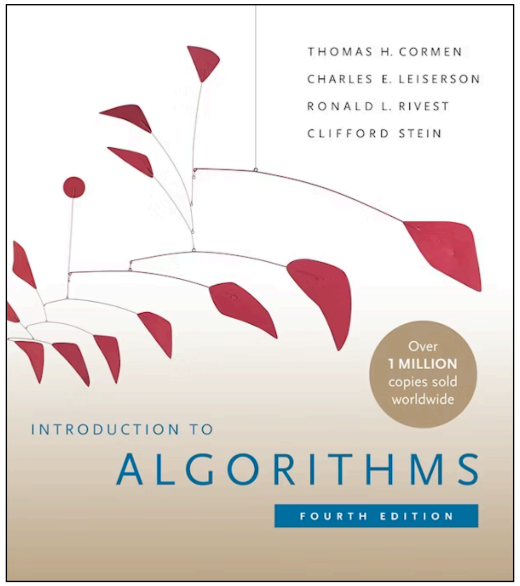
(1) Introduction to Algorithms, 4th edition

(Cormen, Leiserson, Rivest, Stein)

Required

It's OK to use the 3rd edition instead

(library online version of 3rd edition)



(2) Algorithm Design, 1st edition

(Kleinberg and Tardos)