



# Lecture 3: Machines and Models

COMP526: Efficient Algorithms

Updated: October 10, 2024

Will Rosenbaum  
University of Liverpool

# Announcements

---

## 1. First quiz live now, due Friday

- Administered through Canvas
- One question, multiple choice
- 20 minutes
- Covers basic logic (Tuesday's lecture, this week's tutorial, posted notes)
- **Don't start until you're ready to take the quiz.**

## 2. Programming Assignment 1 released next week

- Due 13 November

## 3. Attendance Code:

# Meeting Goals

---

- Finish discussion of mathematical induction
- Analyze algorithm correctness with loop invariants
- Formalize the RAM model of computation
- Introduce asymptotic notation

# Mathematical Induction

# Mathematical Induction

---

## The Principle of Mathematical Induction

Let  $P$  be a predicate over the natural numbers  $\mathbf{N} = \{0, 1, 2, \dots\}$ . Suppose  $P$  satisfies

- **Base case:**  $P(0)$  is true.
- **Inductive step:** For every  $i \in \mathbf{N}$ ,  $P(i) \implies P(i+1)$ .

Then for every  $n \in \mathbf{N}$ ,  $P(n)$  is true. In strictly symbolic notation:

$$(P(0)) \wedge (\forall i[P(i) \implies P(i+1)]) \implies \forall nP(n).$$

# Loop Invariants

## Loop Invariants

Given an algorithm  $A$  containing a loop, a **loop invariant** is a predicate  $P$  on the iterations of the loop such that for each iteration  $i$ ,  $P(i)$  is satisfied at the end of the  $i$ -th iteration of the loop.

## An Uninteresting Example

Consider the following procedure

```
1: procedure COUNT( $n$ )
2:    $t \leftarrow 0$ 
3:   for  $i = 1, \dots, n$  do
4:      $t \leftarrow t + 1$ 
5:   end for
6:   return  $t$ 
7: end procedure
```

## Loop Invariant

After iteration  $i$ ,  $t$  stores the value  $i$ .  
Induct on  $t$ .

- Base case:  $t$  initialized to 0. (Line 2)
- Inductive step:
  - Suppose after iteration  $i$ ,  $t$  stores the value  $i$  (inductive hypothesis)
  - Then in iteration  $i + 1$  after line 4,  $t$  stores the value  $i + 1$ .

# A More Interesting Example

1: **procedure** MININDEX( $(a, i, k)$ ) ▷  
Find the index of the minimum value stored in array  $a$  between indices  $i$  and  $k$ .

2:      $m \leftarrow i$

3:     **for**  $j = i, i + 1, \dots, k$  **do**

4:         **if**  $a[j] > a[m]$  **then**

5:              $m \leftarrow j$

6:         **end if**

7:     **end for**

8:     **return**  $m$

9: **end procedure**

## PollEverywhere Question

What loop invariant does the loop in MININDEX satisfy that will help us analyze its behavior?



[pollev.com/comp526](https://pollev.com/comp526)

# A More Interesting Example

1: **procedure** MININDEX( $(a, i, k)$ )   ▷  
Find the index of the minimum value stored in array  $a$  between indices  $i$  and  $k$ .

2:      $m \leftarrow i$

3:     **for**  $j = i, i+1, \dots, k$  **do**

4:         **if**  $a[j] < a[m]$  **then**

5:              $m \leftarrow j$

6:         **end if**

7:     **end for**

8:     **return**  $m$

9: **end procedure**

## Loop Invariant

After iteration  $j$ ,  $m$  stores the index of the minimum value of  $a$  between indices  $i$  and  $j$ .

## Proof.

Induct on  $j$

- Base case:  $j = i$ .
- Inductive step:  
 $j \implies j+1$





# Further Application

---

Consider the following algorithm that uses `MININDEX` as a subroutine:

```
1: procedure SELECTIONSORT( $a, n$ )           ▷ Sort the array  $a$  of size  $n$ 
2:   for  $i = 1, 2, \dots, n$  do
3:      $j \leftarrow \text{MININDEX}(a, i, n)$ 
4:     SWAP( $a, i, j$ )
5:   end for
6: end procedure
```

## Exercise (Tutorials)

Show that `SELECTIONSORT` correctly sorts any array  $a$  of length  $n$ .  
Specifically:

- Find a suitable loop invariant satisfied by `SELECTIONSORT`
- Prove your loop invariant holds (by induction)
- Argue that your loop invariant implies the final array is sorted

# Induction and Recursion

Induction is essential in reasoning about *recursively defined* methods.

## A Recursive Method

```
1: procedure MYSTERY( $n$ )  
2:   if  $n = 1$  then  
3:     return 1  
4:   end if  
5:   return  $2n - 1 + \text{MYSTERY}(n - 1)$   
6: end procedure
```

## PollEverywhereQuestion

What is the output of MYSTERY(5)?



[pollev.com/comp526](http://pollev.com/comp526)

# Analysis of a Mystery

```
1: procedure MYSTERY( $n$ )
2:   if  $n = 1$  then
3:     return 1
4:   end if
5:   return
    $2n - 1 + \text{MYSTERY}(n - 1)$ 
6: end procedure
```

## Claim

For all  $n$ , MYSTERY( $n$ ) returns the value  $n^2$ .

## Proof.

Induction on  $n$ . Base Case:  $n = 1$ .  
Inductive step: Suppose  
MYSTERY( $n$ ) =  $n^2$ . Then

$$\begin{aligned} \text{MYSTERY}(n+1) &= 2n+1 \\ &\quad + \text{MYSTERY}(n) \\ &= 2n+1 + n^2 \\ &= (n+1)^2. \end{aligned}$$



# Modelling Computation

# What is an Algorithm?

---

**Informally**, an algorithm is a **sequence of instructions** carried out to perform a **prescribed task**.

**More precisely**, an algorithm...

1. is mechanically executable
  - uses only *elementary* operations
  - each operation is determined by the algorithm description and the results of previous operations
    - executing the algorithm requires *no thought*
2. has a finite description
3. solves a *problem* (i.e., a set of instances), not just a single instance

input → processing → output

## Example

The SELECTIONSORT algorithm

- sorts *any* array of size  $n$
- elementary operations/logic

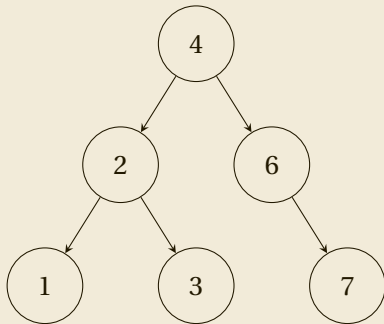
# What is a Data Structure?

---

A **data structure** is

1. a rule for *encoding data* in computer memory, and
2. algorithms for accessing and manipulating the data according to the specified encoding.
  - See next week for much more detail!

**Typical example:** binary search trees



# What Makes an Algorithm “Good?”

---

**Overarching Goal:** find the “best” algorithm and data structures for a task

1. **Correctness:** the algorithm performs the **specified task**...
  - always exactly?
  - always approximately?
  - typically?
2. **Efficiency:** the algorithm doesn't over-use resources
  - fast running *time*
  - small memory *space*
  - small *energy consumption*
  - ...

**Algorithm analysis** gives us a way to

- compare different algorithms
- predict their performance (efficiency) in applications

# Limitations of Empirical Analysis

---

**Question.** Why not just implement your algorithm and test it on a real computer with real data?

- This is *empirical* algorithm analysis

Limitations of **empirical analysis**:

- examines a single (or few) machines
- examines tested inputs
- tests particular implementation
- value of results are highly context dependent

Our Focus is **formal analysis** of algorithms on an abstract computer

- *prove* results for our computational model
- results apply to any computer that satisfy our assumptions

**Note.** Neither formal nor empirical analysis is *better*—both are important to computer science!

- this module just focuses on formal analysis



# Data Models

---

To perform formal algorithm analysis, we must specify

1. the computational model
2. the relevant performance parameter  $\implies$  notion of **cost**

Types of performance:

- **worst-case performance**

Over all possible inputs, what is the *worst* cost of our algorithm's execution?

- **best-case performance**

Over all possible inputs, what is the *best* cost of our algorithm's execution?

- **average-case performance**

What is the *average* or *expected* cost for a of a *random* input

Typically, we analyze performance as a function of **input size**,  $n$

$\implies$  measure performance as a function of  $n$ : how does performance *scale* with input size

# Computational Models

---

A **computational model** defines

1. *syntax*: what operations can be performed
2. *semantics*: what are the effects of those operations
3. the computational cost of the operations

These features determine what problems can be solved and with what efficiency

Choosing the “right” computational model is a balance of

- computational power
- simplicity
- realism

Successful & general computational models defined in the 1930's



Credit: Princeton University

Alonzo Church  
(Lambda Calculus)



Credit: Unknown

Alan Turing  
(Turing Machines)

# Random Access Machines

---

The RAM Model consists of:

- Unlimited **memory**, access by address
  - stores program, input, intermediate data
  - each address stores fixed sized “word”
- Fixed number of **registers**
  - stores “working” data
- Elementary **instructions**
  - load & store: move data between registers/memory
  - arithmetic operations: bit-wise ops, addition/subtraction (fixed precision),...
  - conditional/unconditional jump
- **cost** = number of instructions executed



Credit: LANL

Jon von Neumann

**RAM** is a standard model for *sequential* computation, similar to assembly

# Pseudocode

---

The RAM model captures many aspects of real computers...  
...but it is not intuitive for **high level** algorithm description.

Simplifying abstractions:

- Higher level abstract **pseudocode**:
  - named variables, assignment
  - control flow: **if**, **for**, **while**, etc.
  - assumed memory management
- Cost: *dominant operations* (e.g., memory access) instead of all RAM instructions

Pseudocode can (in principle) be implemented in RAM model, just as C++ can be compiled to assembly

```
procedure EUCLID( $a, b$ )  
  while  $b \neq 0$  do  
     $t \leftarrow b$   
     $b \leftarrow a \bmod b$   
     $a \leftarrow t$   
  end while  
  return  $a$   
end procedure
```

# (Important) Things We Ignore

---

## System & Hardware Level Details

- Memory allocation
  - required to implement arrays, dynamic memory usage, etc
- Pointers
  - correspondence between variable names, values, and memory addresses
- Support for procedures/methods/functions
  - call stack, call frame, etc.

This are all fundamental problems in computer science, just not within the purview of COMP526

- See modules on operating systems, compilers, programming languages, etc.

# Asymptotic Notation

# Measuring (Time) Efficiency

---

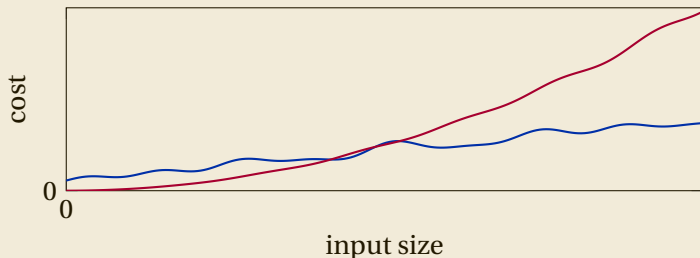
**Recall.** We measure *(time) efficiency* in terms of number of elementary operations performed

- assume all operations are unit cost

We want a **robust** measure of efficiency that is *independent* of particular “real world” costs of operations

- focus on how the number of operations **scales** with input size

This motivates the study of **asymptotic analysis**.



# Big-O Notation

---

**Goal** of “Big- $O$ ” or asymptotic notation: a way of describing the *growth* of functions that is:

- **coarse** enough to be simple enough to analyze
  - independent of hardware or implementation constants
- **precise** enough to be informative

## Definition

Suppose  $f$  and  $g$  are functions from  $\mathbf{N}$  to  $\mathbf{R}^+$ . Then we say that  $f = O(g)$  (read:  $f$  is *big O* of  $g$ ) if there exist constants  $N_0 \in \mathbf{N}$  and  $C \in \mathbf{R}$  such that for all  $n \in \mathbf{N}$

$$n \geq N_0 \implies f(n) \leq Cg(n).$$

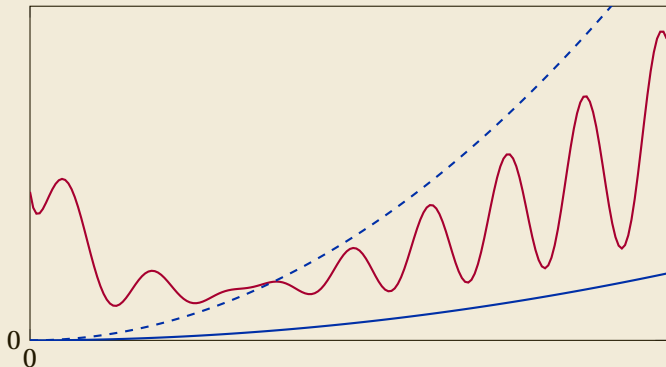
Equivalently,  $f = O(g) \iff \limsup \frac{f(n)}{g(n)} < \infty$



# Big-O in Pictures

## Definition

Suppose  $f$  and  $g$  are functions from  $\mathbf{N}$  to  $\mathbf{R}^+$ . Then we say that  $f = O(g)$  (read:  $f$  is *big O* of  $g$ ) if there exist constants  $N_0 \in \mathbf{N}$  and  $C \in \mathbf{R}$  such that for all  $n \in \mathbf{N}$ ,  $n \geq N_0 \implies f(n) \leq Cg(n)$ .



# Properties of $O$

## Proposition

Suppose  $f, f_1, f_2, g, g_1, g_2, h$  are functions and  $a$  is any constant. Then:

1.  $(\forall n f(n) \leq a) \implies f = O(1)$
2.  $(\forall n f(n) \leq g(n)) \implies f = O(g)$
3.  $f = O(g) \implies a \cdot f = O(g)$
4.  $f = O(g)$  and  $g = O(h) \implies f = O(h)$
5.  $f = O(h)$  and  $g = O(h) \implies f + g = O(h)$
6.  $f_1 = O(g_1)$  and  $f_2 = O(g_2) \implies f_1 \cdot f_2 = O(g_1 \cdot g_2)$

Consequence:

- If  $a \leq b$  then  $n^a = O(n^b)$

**Exercise.** Show that if  $a > b$ , then  $n^a \neq O(n^b)$ .

# Example with Functions

---

Analyze the asymptotic growth of

$$f(n) = 17n^2 + 42n + 38\sqrt{n} + 972$$

1.  $(\forall n f(n) \leq a) \implies f = O(1)$
2.  $(\forall n f(n) \leq g(n)) \implies f = O(g)$
3.  $f = O(g) \implies a \cdot f = O(g)$
4.  $f = O(g)$  and  
 $g = O(h) \implies f = O(h)$
5.  $f = O(h)$  and  
 $g = O(h) \implies f + g = O(h)$
6.  $f_1 = O(g_1)$  and  
 $f_2 = O(g_2) \implies f_1 \cdot f_2 = O(g_1 \cdot g_2)$

# Example with Pseudocode

---

**Example.** Analyze the *worst-case* running time of the INSERTIONSORT procedure defined below.

- *assume* elementary operations have running time  $O(1)$

```
1: procedure INSERTIONSORT( $a, n$ )
2:   for  $i = 2, 3, \dots, n$  do
3:      $j \leftarrow i$ 
4:     while  $j > 1$  and  $a[j-1] < a[j]$  do
5:       SWAP( $a, j-1, j$ )
6:        $j \leftarrow j-1$ 
7:     end while
8:   end for
9: end procedure
```

# Best Case Running Time

```
1: procedure INSERTIONSORT( $a, n$ )
2:   for  $i = 2, 3, \dots, n$  do
3:      $j \leftarrow i$ 
4:     while  $j > 1$  and  $a[j-1] < a[j]$  do
5:       SWAP( $a, j-1, j$ )
6:        $j \leftarrow j-1$ 
7:     end while
8:   end for
9: end procedure
```

## PollEverywhere Question

What is the **best case** running time for INSERTIONSORT? What arrays has this running time?



[pollev.com/comp526](https://pollev.com/comp526)

# Variations of O

---

- $f = \Theta(g)$  if  $f = O(g)$  and  $g = O(f)$ 
  - Example:  $4n^2 + 3n + 7 = \Theta(n^2)$
- $f = \Omega(g)$  if  $g = O(f)$ 
  - Example:  $0.01n^2 - 7n = \Omega(n^2)$
- $f = o(g)$  if for every  $\varepsilon > 0$ , there exists  $N_0$  such that  $n \geq N_0 \implies \frac{f(n)}{g(n)} < \varepsilon$ .
  - Equivalently:  $f = o(g) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
  - Example:  $n^{1.999} = o(n^2)$
- $f = \omega(g)$  if  $g = o(f)$ 
  - Example:  $0.01n^{2.01} = \omega(n^2)$

**Mnemonic** for Variations

Big-O	(in)equality
$\omega$	$>$
$\Omega$	$\geq$
$\Theta$	$\approx$
$O$	$\leq$
$o$	$<$

# Interpretation

---

Suppose:

- two algorithms  $A$  and  $B$  for solving the same problem
- running time of  $A$  is  $f$ , running time of  $B$  is  $g$
- $f = o(g)$

Consider running  $A$  on a **slow** machine  $M_1$  and  $B$  on a **fast** machine  $M_2$ . Then: regardless of how much slower  $M_1$  is than  $M_2$ , for *sufficiently large* inputs,  $A$  will complete faster than  $B$ .

**The Moral.** Efficient *algorithms* are better than faster hardware.

- little- $o$  notation gives the “right” abstraction to formalize this relationship

# Common Orders of Growth

---

**Named** orders of growth:

name	asymptotic growth
constant	$O(1)$
logarithmic	$O(\log n)$
polylogarithmic	$O(\log^c n)$
linear	$O(n)$
almost linear	$O(n \log^c n)$
quadratic	$O(n^2)$
polynomial	$O(n^c)$
exponential	$O(c^n)$

## Relationships

---

Between classes:

For all  $a, b > 0$

- $a = o(\log^b n)$
- $\log^a n = o(n^b)$
- $n^a = o(b^n)$

---

Within classes:

For all  $a, b, a < b$

- $\log^a n = o(\log^b n)$
- $n^a = o(n^b)$
- $a^n = o(b^n)$



# Next Time

---

- Abstract Data Types
- Fundamental Data Structures

# Scratch Notes

---