Computation

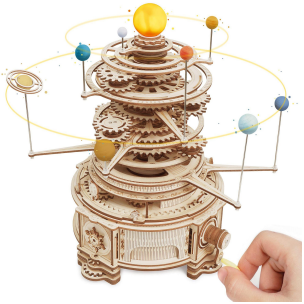
CS 220 — Applied Discrete Mathematics

April 16, 2025

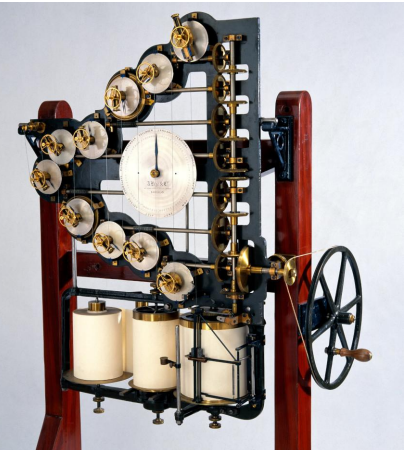
Analogy Machines

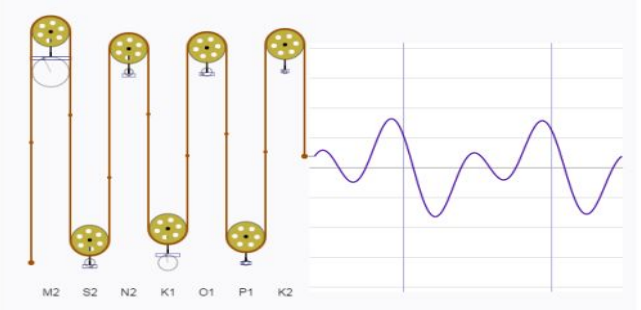
(Inspired partly by The Most Powerful Computers You've Never Heard Of.)

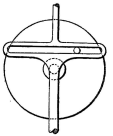
Orrery

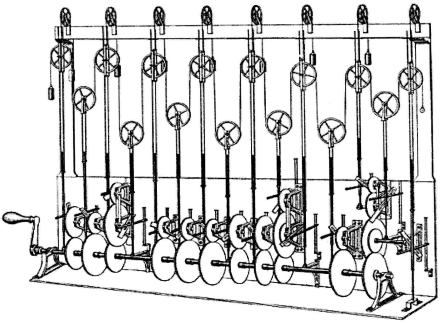


Tide predictor

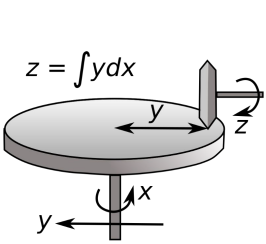


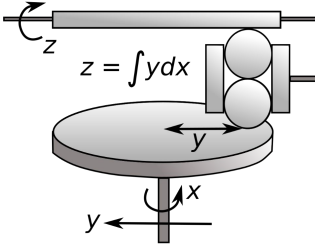


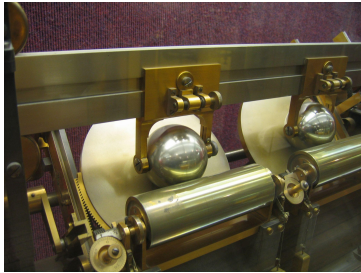


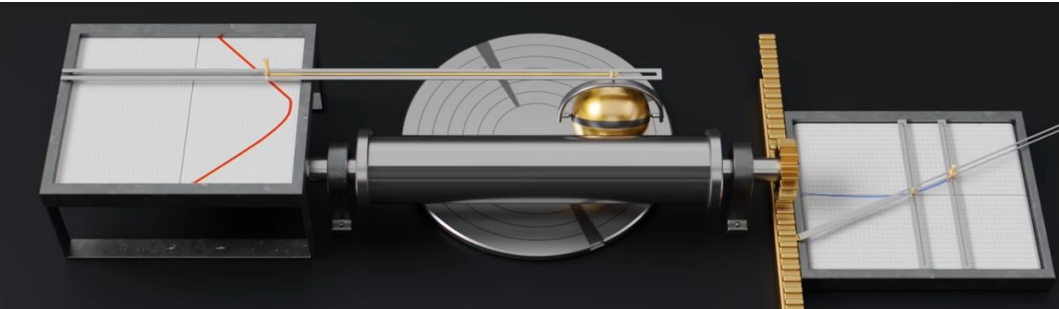


Integrator

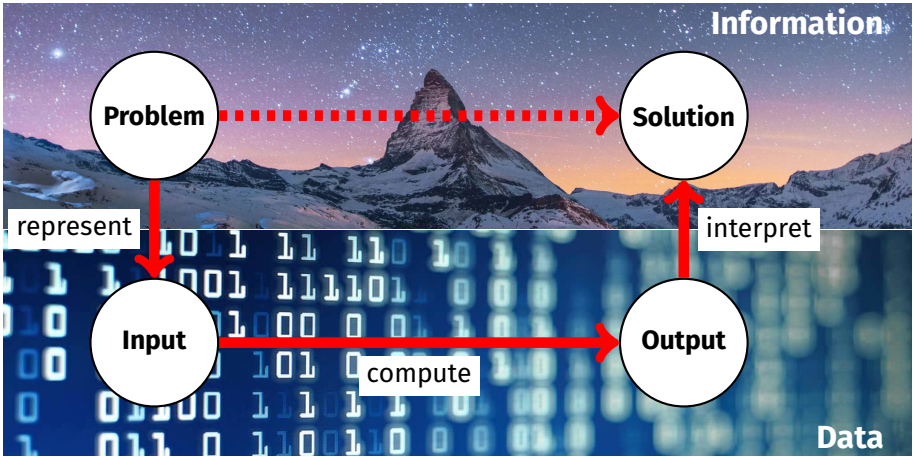








Analogy Engineering



Breaking Analogies

A Riddle:

There are three people in a room.

Five people leave the room.

How many people must enter the room for it to become empty?

Breaking Analogies

A Riddle:

There are three people in a room.

Five people leave the room.

How many people must enter the room
for it to become empty?

A Wrong Answer:

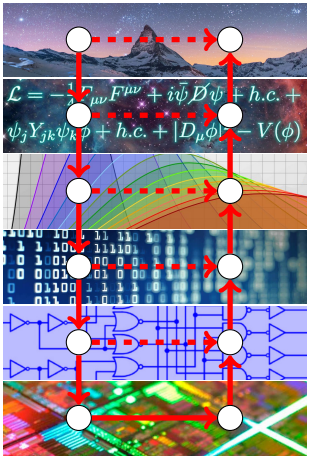
I know algebra! $3 - 5 + x = 0$, so $x = 2$.

Thus if two people enter the room it will be empty.
(Because the room currently has -2 people in it.)

Respect the analogy's limits and tolerances

Computing with misrepresented data is dangerous.
Interpreting meaningless data is dangerous.

Towers of Analogies



real-world physics

mathematical models

numerical solvers/algos

floating-point arithmetic

Boolean logic: T, F, $\wedge$, $\vee$, etc

electrical properties of silicon

Mechanical Reasoning

Mechanical Reasoning (or, Are Mathematicians Obsolete?)

In the last few modules, we learned about proof systems that let us **mechanically** (aka, **formally**) "reason"—that is, derive true propositions.

Is it possible that every mathematical question could automatically be "figured out" by some effective mechanical method?

This question, called the ***Entscheidungsproblem*** ("decision problem"), was famously posed by David Hilbert and Wilhelm Ackermann in 1928, after advances by George Boole (1847), Gottlob Frege (1879), and Giuseppe Peano (1888) distilled logic to a symbolic language governed by deterministic rules.

What is an Effective Mechanical Method?

What criteria do we need for an **effective mechanical method**?

- Each method is concerned only with a single kind of problem.
  The set of problem **inputs** and **outputs** should be well understood.
- It must consist of a **finite** arrangement of instructions.
- Each instruction must have a **definite** interpretation.
  That is, it must work without the aid of *luck* or *human ingenuity*.
- When applied to any problem in its domain, it must **terminate** in a finite number of steps with the **correct** answer.

What is an Effective Mechanical Method?

What criteria do we need for an **effective mechanical method**?

- Each method is concerned only with a single kind of problem.
  The set of problem **inputs** and **outputs** should be well understood.
- It must consist of a **finite** arrangement of instructions.
- Each instruction must have a **definite** interpretation.
  That is, it must work without the aid of *luck* or *human ingenuity*.
- When applied to any problem in its domain, it must **terminate** in a finite number of steps with the **correct** answer.

This is what we now call an **algorithm**.

A problem with a yes/no answer is called a **decision problem**,
and an algorithm that solves it is called a **decision procedure**.

Forms of Instructions

What does an "arrangement of instructions" look like?

- Kurt Gödel and Jacques Herbrand (1933): general recursive functions
- Alonzo Church (1936): λ-calculus terms (CS450)
- Alan Turing (1936): machines consisting of a finite state automaton paired with an unbounded read-write "tape" (CS420)

Each model provides an encoding of $\mathbb{N}$, and each model defines its own notion of whether a $\mathbb{N} \to \mathbb{N}$ function is **computable**.

Forms of Instructions

What does an "arrangement of instructions" look like?

- Kurt Gödel and Jacques Herbrand (1933): general recursive functions
- Alonzo Church (1936): λ-calculus terms (CS450)
- Alan Turing (1936): machines consisting of a finite state automaton paired with an unbounded read-write "tape" (CS420)

Each model provides an encoding of $\mathbb{N}$, and each model defines its own notion of whether a $\mathbb{N} \to \mathbb{N}$ function is **computable**.

By 1937, Church, Kleene, and Turing proved that all three models define the same set of computable functions.

The **Church-Turing Thesis**: That's what "effectively calculable" means, then.

The *Entscheidungsproblem*

Is there an algorithm that can decide every statement in mathematics?

The *Entscheidungsproblem*

Is there an algorithm that can decide every statement in mathematics?

Kurt Gödel: No. (**Gödel's Incompleteness Theorem**)

This is a big, awesome topic. I recommend *Gödel, Escher, Bach* by Douglas Hofstadter.

Algorithms

Algorithms

Definition (Algorithm)

An **algorithm** is a finite arrangement of precise instructions for performing a computation or for deciding a question.

An algorithm must have a definite interpretation, and it must terminate in a finite number of steps with a correct answer.

Analysis of Algorithms

Given a procedure:

- Is it an algorithm?
  - Does it always terminate?

    Show loop termination using a **termination measure**, usually a non-negative integer value that decreases each iteration.
  - Does it produce the correct answer?

    Often, loop correctness can be shown by a **loop invariant**, a proposition that is true on each iteration of the loop (and also when the loop exits).
- How efficient is it?
  - (Running time) How long does it take to run?
  - (Space) How much memory does it require?

Example: Find-Max

Algorithm 1 find-max$(A[1\,..\,n])$

Ensure: Returns $\max\{A[1], \ldots, A[n]\}$.

```
1: m ← A[1]
2: i ← 2
3: while i ≤ n do
4:     if m < A[i] then
5:         m ← A[i]
6:     end if
7:     i ← i + 1
8: end while
9: return m
```

Termination

- termination measure: $n - i + 1$

Correctness

- Loop Invariant:
  $m = \max\{A[1], \ldots, A[i-1]\}$
- Before loop starts (with $i = 2$):
  $m = \max\{A[1]\}$
- When the loop exits (with $i = n + 1$):
  $m = \max\{A[1], \ldots, A[n]\}$

Running time

- $n - 1$ loop iterations

Example: Linear Search

Algorithm 2 linear-search$(x, A[1 \ldots n])$

```
1:  i ← 1
2:  while (i ≤ n and x ≠ A[i]) do
3:      i ← i + 1
4:  end while
5:  if i ≤ n then
6:      location ← i
7:  else
8:      location ← 0
9:  end if
10: return location
```

Returns i such that $A[i] = x$,
or 0 if A does not contain x.

Termination

- termination measure: $n - i + 1$

Correctness

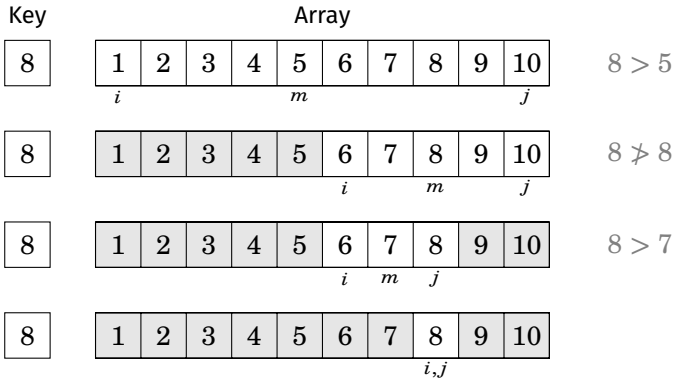
- Loop Invariant:
  x is not in $A[1 \,..\, (i-1)]$
- When the loop exits, either
  - $i > n$:
    Then by LI, x is not in $A[1 \,..\, n]$.
    So return 0.
  - $x = A[i]$.
    So return i.

Running time

- up to n loop iterations

Example: Binary Search

- If the terms in a sequence are ordered, a binary search algorithm is more efficient than linear search.
- The binary search algorithm iteratively restricts the relevant search interval until it closes in on the position of the element to be located.
- **Example:** Binary search for the number 8.



Example: Binary Search

Algorithm 3 binary-search$(x, A[1..n])$

Require: A is sorted in ascending order

```
 1: i ← 1, j ← n                          search A[1..n]
 2: while (i < j) do
 3:     m ← ⌊(i + j)/2⌋
 4:     if x > A[m] then
 5:         i ← m + 1                    search A[(m + 1)..j]
 6:     else
 7:         j ← m                        search A[i..m]
 8:     end if
 9: end while
10: if (x = A[i]) then
11:     location ← i
12: else
13:     location ← 0
14: end if
15: return location
```

Returns i such that $A[i] = x$,
or 0 if A does not contain x.

Termination

- termination measure: $j - i$

Correctness

- ~~$A[i] \leq x \leq A[j]$~~
- Loop Invariant: if x is in A, then x is in $A[i..j]$.
- When the loop exits, $i = j$, and either $A[i] = x$ or x is not in A.
- Relies on **precondition**: sorted.

Running time

- up to $\lceil \log_2(n) \rceil$ loop iterations

Measuring Computational Complexity

How much **time** will an algorithm take?

> *On what hardware? What language and compiler?*
> *Is the VM "hot"? Are there other threads running? …*

We want* to abstract these details away.

Furthermore, we disregard the time usage for small inputs.

Instead, we care how the time *grows* as a function of input size.

(We can also ask the same question about **space** usage.)

Computational Complexity

Input	Alg. A	Alg. B
n	$5000n$	$\lceil 1.1^n \rceil$
10	50,000	3
100	5×10^5	13,781
1000	5×10^6	2.5×10^{41}
10^6	5×10^9	4.8×10^{41392}

This means that algorithm B cannot be used for large inputs, while running algorithm A is still feasible.

The growth class of the time (or space) usage as a function of input size is

- a **mathematically stable** basis for comparing algorithms, and
- often a useful approximation to an algorithm's performance in practice.

Counterpoint: Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step

Big-O Notation

Definition (Big-O)

Suppose $t, f : \mathbb{R}^+ \to \mathbb{R}^+$.

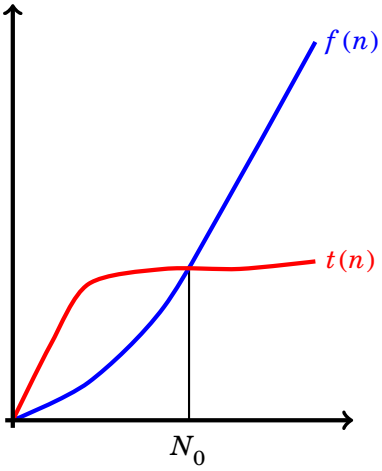
Then t is $O(f)$ if there are constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$:

$$t(n) \leq c \cdot f(n)$$

That is, there is some multiple of $f(n)$ that eventually bounds $t(n)$ from above.

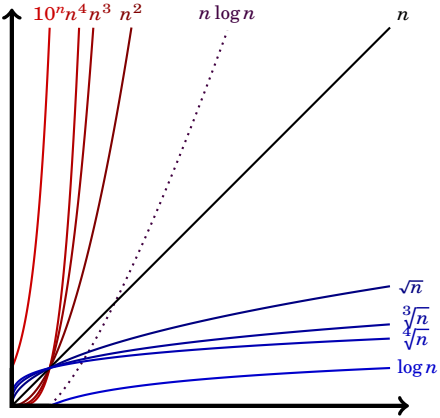
Example

- $2n + 4$ is $O(n)$
- $n^2 + 2n + 1$ is $O(n^2)$



Useful Nomenclature

Function	Name
c	Constant
$\log n$	Logarithmic
$\log^2 n$	Log-squared
n	Linear
$n \log n$	$n \log n$
n^2	Quadratic
n^3	Cubic
2^n	Exponential



Euclid's Algorithm

Greatest Common Divisor and Least Common Multiple

Recall:

Definition

The greatest common divisor (GCD) of a and b, where $a, b \in \mathbb{Z}^+$, written $\gcd(a, b)$, is the greatest $d \in \mathbb{Z}^+$ such that $d \mid a$ and $d \mid b$.

Update: We can also allow one of the arguments to be zero:

$$\gcd(a, 0) = \gcd(0, a) = a \qquad \text{if } a > 0$$

But $\gcd(0, 0)$ is undefined.

Euclid's Algorithm

There is a better algorithm for finding the GCD of two integers that dates back to Euclid's *Elements*, from around 300 BC.

Euclid's Algorithm

$$\gcd(a,b) = \begin{cases} \gcd(b,\ a \bmod b) & \text{if } b > 0 \\ a & \text{if } b = 0 \text{ and } a > 0 \end{cases}$$

Example

Suppose we want to find $\gcd(287, 91)$.

$$\begin{aligned} \gcd(287,91) &= \gcd(91,\mathbf{14}) && \text{because } 287 = 3 \cdot 91 + \mathbf{14} \\ &= \gcd(14,\mathbf{7}) && \text{because } 91 = 6 \cdot 14 + \mathbf{7} \\ &= \gcd(7,\mathbf{0}) && \text{because } 14 = 2 \cdot 7 + \mathbf{0} \\ &= 7 \end{aligned}$$

Euclid's Algorithm

Algorithm 4 euclid-$\gcd(a, b)$

Require: $a, b \in \mathbb{Z}^+$
Ensure: Returns $\gcd(a, b)$.

```
1: x ← a
2: y ← b
3: while y ≠ 0 do
4:     r ← x mod y
5:     x ← y
6:     y ← r
7: end while
8: return x
```

Termination

- **while** loop: y decreases

Correctness

- Loop Invariant:
  $\gcd(x, y) = \gcd(a, b)$
- After each loop iteration:
  see lemma
- When the loop exits (with $y = 0$):
  $\gcd(x, 0) = \gcd(a, b)$

Running time

- $O(\log(\max(a, b)))$ iterations

Correctness of Euclid's Algorithm

Lemma

Let $a, b \in \mathbb{N}$ with $b \neq 0$. Then $\gcd(a, b) = \gcd(b,\ a \bmod b)$.

Proof.

First we show that a, b have the same common divisors as $a,\ a \bmod b$.

Recall this property: for all $x, y, z \in \mathbb{Z}$, if $x \mid y$ and $x \mid z$, then $x \mid (y + z)$.

By division, there are q, r such that $a = qb + r$, where $r = a \bmod b$.

- By applying the divisibility property above to the equation for a, we see that any common divisor of b and $a \bmod b$ must also be a divisor of a.
- By rewriting the equation to $a \bmod b = a + (-q)b$ and applying the divisibility property again, we see that any common divisor of a and b is also a divisor of $a \bmod b$.

Since a, b have exactly the same common divisors as $b,\ a \bmod b$, they have the same *greatest* common divisor. That is, $\gcd(a, b) = \gcd(b, a \bmod b)$. □

The Extended Euclidean Algorithm

The GCD of a and b can be expressed as a linear combination of a and b. That is, $\gcd(a,b) = sa + tb$ for some $s, t \in \mathbb{Z}$.

Example

Previously: $\gcd(287, 91) = \gcd(91, 14) = \gcd(14, 7) = \gcd(7, 0) = 7$.

$$\begin{aligned} 287 &= 3 \cdot 91 + 14 & \rightarrow \quad 14 &= 287 - 3 \cdot 91 \\ 91 &= 6 \cdot 14 + 7 & \rightarrow \quad 7 &= 91 - 6 \cdot 14 \\ 14 &= 2 \cdot 7 + 0 \end{aligned}$$

If we substitute backwards using the equations on the right:

$$\begin{aligned} 7 &= 91 - 6 \cdot 14 && \text{by 2nd equation} \\ &= 91 - 6 \cdot (287 - 3 \cdot 91) && \text{by 1st equation} \\ &= -6 \cdot 287 + 19 \cdot 91 && \text{by algebra} \end{aligned}$$

(In fact, every intermediate remainder can be expressed as some $sa + tb$ ...)

The Extended Euclidean Algorithm

Algorithm 5 extended-euclid-gcd(a, b)

Require: $a, b \in \mathbb{Z}^+$

1: $x \leftarrow a$
2: $y \leftarrow b$
3: $s_x \leftarrow 1;\ t_x \leftarrow 0$
4: $s_y \leftarrow 0;\ t_y \leftarrow 1$
5: **while** $y \neq 0$ **do**
6: $\quad q \leftarrow \lfloor x/y \rfloor$
7: $\quad r \leftarrow x \bmod y$
8: $\quad x \leftarrow y$
9: $\quad y \leftarrow r$
10: $\quad s_x, s_y \leftarrow s_y, s_x - q s_y$
11: $\quad t_x, t_y \leftarrow t_y, t_x - q t_y$
12: **end while**
13: **return** x, s_x, t_x

Termination

- **while** loop: y decreases

Correctness

- Loop Invariants:
  - $\gcd(x, y) = \gcd(a, b)$
  - $x = s_x a + t_x b$
  - $y = s_y a + t_y b$
- After each loop iteration: see lemma; algebra
- When the loop exits (with $y = 0$):
  - $\gcd(x, 0) = \gcd(a, b)$
  - $x = s_x a + t_x b$

Running time

- $O(\log(\max(a, b)))$ iterations

The Multiplicative Inverse in Modular Arithmetic

Definition (Multiplicative Inverse)

Let $m \in \mathbb{Z}^+$, and let $x \in \mathbb{N}$. The **multiplicative inverse (mod m)** of x is a number $y \in \{0, \ldots, m-1\}$ such that $xy \equiv 1 \pmod{m}$.

The multiplicative inverse does not always exist.
It exists if and only if x and m are relatively prime.

Examples

- the multiplicative inverse (mod 10) of 7 is 3
  because $3 \cdot 7 = 21 \equiv 1 \pmod{10}$
- the multiplicative inverse (mod 8) of 7 is 7
  because $7 \cdot 7 = 49 \equiv 1 \pmod{8}$
- the multiplicative inverse (mod 6) of 4 does not exist
  because 4 times anything is even, and no even number is $\equiv 1 \pmod{6}$

Calculating the Multiplicative Inverse

We can use the extended Euclidean Algorithm to find the multiplicative inverse of $x \pmod{m}$:

- If $\gcd(x, m) \neq 1$ then the multiplicative inverse does not exist.
- Otherwise (if x and m are relatively prime), the algorithm computes s and t such that $sx + tm = 1$.

  Thus $sx - 1 = -tm$, and thus $sx \equiv 1 \pmod{m}$ (by definition).

Example

We could use the extended Euclidean algorithm to calculate

$$\gcd(31, 43) = 1 = -18 \cdot 31 + 13 \cdot 43$$

If we take -18 and normalize it to the proper range: $(-18) \bmod 43 = 25$.

So the multiplicative inverse (mod 43) of 31 is 25.

Check: $31 \cdot 25 = 775 = 18 \cdot 43 + 1 \equiv 1 \pmod{43}$.