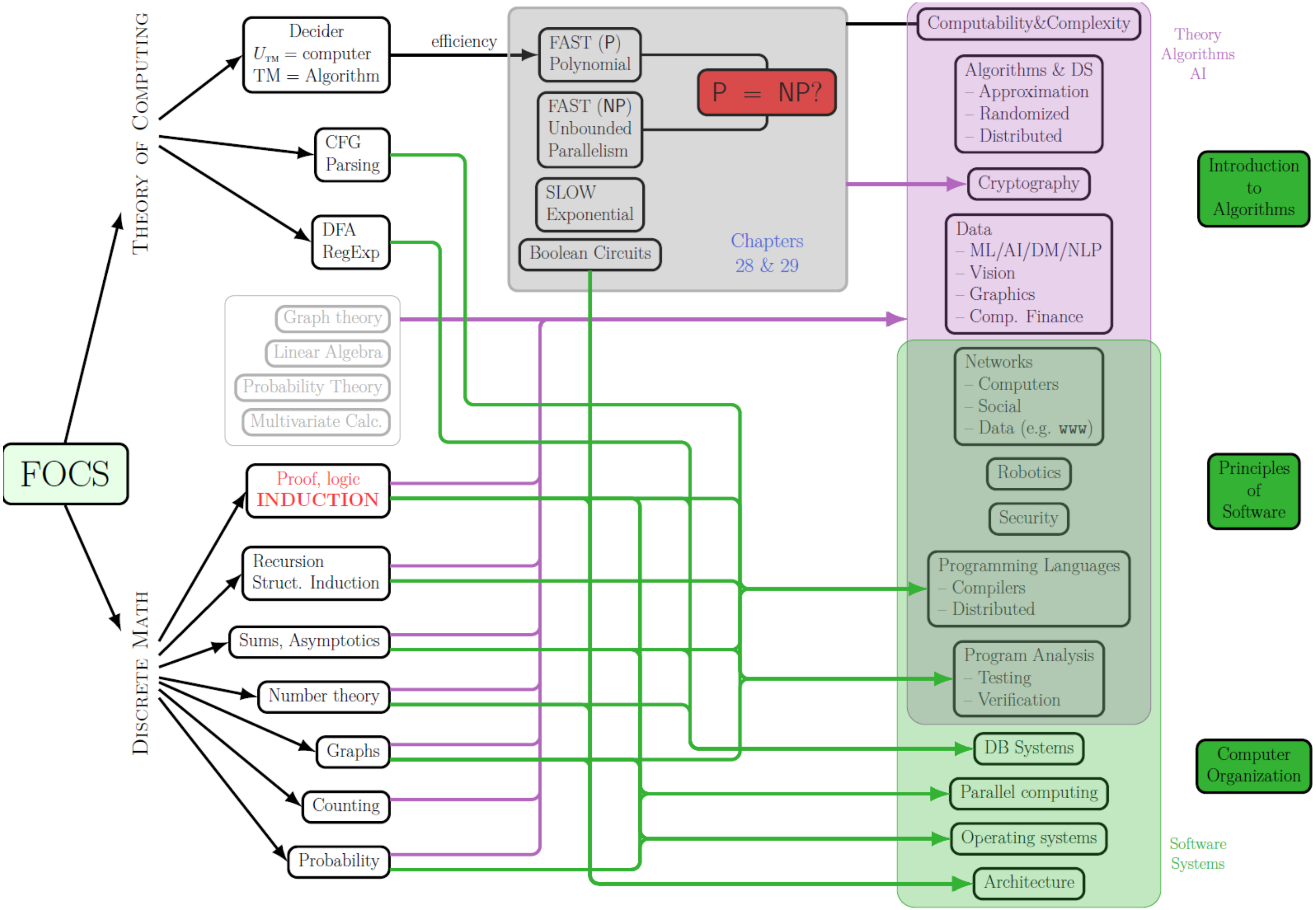
# Efficiency

# Reading

- Malik Magdon-Ismail. Discrete Mathematics and Computing.
  - Chapter 28

# The Path Forward: Focus on Decidable Problems

Rensselaer

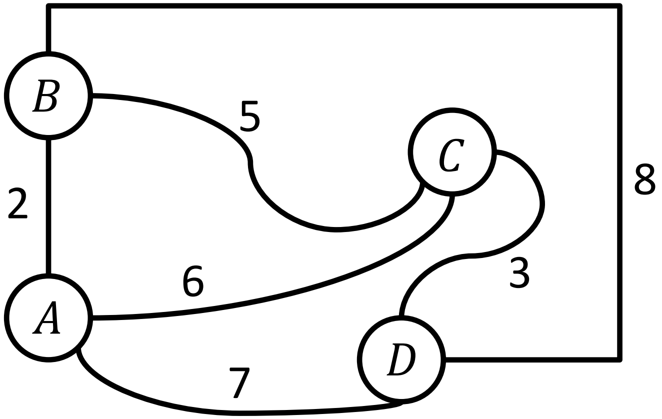


# Overview

- Time complexity: Asymptotic worst-case analysis.
- The class P: Efficiently solvable problems.
- Polynomial on one architecture means polynomial on pretty much any architecture.

# Running Time

- Suppose you are a traveling salesman
  - You want to visit all cities in your state and sell your stuff
  - Suppose the road network looks like this

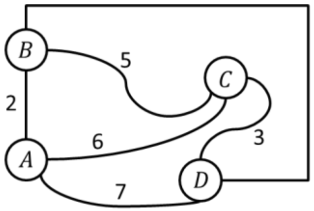



  - The numbers give the length of each road
  - What's the shortest path that starts at $A$, visits each city once, and returns to $A$?

$A, B, C, D, A$ (cost of 17)

  - How did you find it?
  - Enumerated all possible paths and took the fastest

# Running Time, cont'd




- What if we had 50 cities? How many paths would we have to check?
  - There are 49 options for the 1st city, then 48, etc.
  - In total, there are $49!$ paths
- How many paths are there in a general graph with $n$ nodes?
  - There are $n - 1$ options for the first node, then $n - 2$, etc.
  - In total there are $(n - 1)!$ paths
- Even for 50 cities, checking all paths on a 10GHz computer would take $10^{50}$ years!
  - The universe is $10^{12}$ years old…
- There's got to be a faster way to find the optimal path!

# Running Time, cont'd

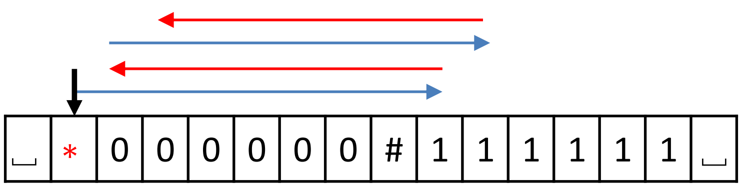
- Efficiently solving a problem, with low runtime, is as important as solving it!
  - I don't care about your algorithm if it will finish after there is no more Earth!
- How do we analyze algorithms?
- Let's look at our Turing Machine for balanced strings

$\underline{M = \text{Turing Machine that solves } \{0^{\bullet k}\#1^{\bullet k}\}}$

Input: Binary string $w$

1. Check that input has the correct format and return to $*$
2. Match each 0 left of # with a 1 right of #
3. If a match fails or there are more 1s, REJECT. Otherwise ACCEPT

- How fast is that algorithm?
  - Hard to tell from this high-level sketch
  - Machine is in reality doing a zig-zag



# Running Time, cont'd

- Let's be more specific with our step 2:

$M$ = Turing Machine that solves $\{0^{\bullet k}\#1^{\bullet k}\}$

Input: Binary string $w$

1. Check that input has the correct format and return to $*$
2. Match each 0 left of # with a 1 right of #
   Move right and mark the first unmarked 0 (if none, GOTO step 3)
   Move right and mark the first unmarked 1 (if none, REJECT)
   Move left until you come to a marked 0.
3. If a match fails or there are more 1s, REJECT. Otherwise ACCEPT

- Now we can at least count how many operations each step takes depending on string size

# Time Complexity

- To analyze the runtime of $M$, we must specify the input

$\underline{M = \text{Turing Machine that solves}}$ $\{0^{\bullet k}\#1^{\bullet k}\}$
Input: Binary string $w$
1. Check that input has the correct format and return to *
2. Match each 0 left of # with a 1 right of #
   Move right and mark the first unmarked 0 (if none, GOTO step 3)
   Move right and mark the first unmarked 1 (if none, REJECT)
   Move left until you come to a marked 0.
3. If a match fails or there are more 1s, REJECT. Otherwise ACCEPT

- Runtime increase with the input size. What size input shall we take?
- Runtime can vary within inputs of the same size. What do we do about this?

- **Worst case analysis**!
  - This is the norm in computer science
  - How does our algorithm perform over the worst possible input

# Worst Case Analysis

- The steps to get the worst case runtime are:

1. Fix the size of the input to $n$ and identify the worst input $w_*$ of size $n$
2. Determine the runtime for the input $w_*$. This worst case runtime will depend on $n$

- To determine the runtime, recall the definitions

| $T \in o(f)$ | $T \in O(f)$ | $T \in \Theta(f)$ | $T \in \Omega(f)$ | $T \in \omega(f)$ |
|---|---|---|---|---|
| "$T < f$" | "$T \leq f$" | "$T = f$" | "$T \geq f$" | "$T > f$" |

- In practice, inputs are very large, i.e., $n \to \infty$
  - We care about runtimes in the asymptotic regime
- Additive and multiplicative constants are minor
  - We care about the growth rate of the runtime with $n$

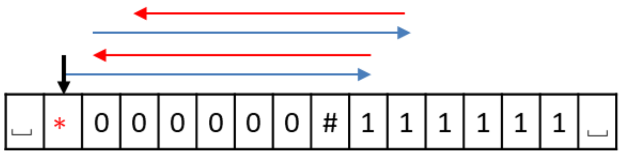
# Analyzing the decider for $\{0^{\bullet k}\#1^{\bullet k}\}$

- How many operations does this machine perform?

$\underline{M = \text{Turing Machine that solves }}\{0^{\bullet k}\#1^{\bullet k}\}$

Input: Binary string $w$

1. Check that input has the correct format and return to $*$
2. Match each 0 left of # with a 1 right of #
    - Move right and mark the first unmarked 0 (if none, GOTO step 3)
    - Move right and mark the first unmarked 1 (if none, REJECT)
    - Move left until you come to a marked 0.
3. If a match fails or there are more 1s, REJECT. Otherwise ACCEPT




- During step 1, in the worst case, go through the entire input twice
- During step 2, machine zig-zags for each 0
    - For a well-formatted string, there are at most $\frac{n}{2}$ zig-zags and each zig-zag is at most $2n$ steps
- Finally, step 3 takes one full scan
- So in total:

$$runtime \leq 2n + \frac{n}{2} \times 2n + n$$

- What is $O(2n + \frac{n}{2} \times 2n + n)$ simplified to?

# Can we do better than $O(n^2)$?

- What if we instead scan every other 0 and then every other 1?

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ␣ | * | ✓0 | 0 | ✓0 | 0 | ✓0 | 0 | # | ✓1 | 1 | ✓1 | 1 | ✓1 | 1 | ␣ |

- We cover roughly half the string in a single scan! Then the next scan is similar

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ␣ | * | ✓0 | ✓0 | ✓0 | 0 | ✓0 | ✓0 | # | ✓1 | ✓1 | ✓1 | 1 | ✓1 | ✓1 | ␣ |

- The final scan completes the last missing 0 and 1

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ␣ | * | ✓0 | ✓0 | ✓0 | ✓0 | ✓0 | ✓0 | # | ✓1 | ✓1 | ✓1 | ✓1 | ✓1 | ✓1 | ␣ |

- **Exercise**. Finish the details of this algorithm and prove it is correct
- How many passes does the above algorithm make?
  - Input is halved after each pass, so at most $\log_2 n$
- How long is each pass at most?
  - Can't go further than full input, so $n$ (rough analysis gets you far in algorithm analysis!)
- Total runtime is $O\left(nlog_2(n)\right)$

# Can we do better than $O(nlog_2(n))$? Two Tapes

- Suppose we had a Turing Machine with two tapes
  – Computer scientists like convoluted devices!

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| tape 1 | ␣ | * | 0 | 0 | 0 | 0 | 0 | 0 | # | 1 | 1 | 1 | 1 | 1 | 1 | ␣ |
| tape 2 | ␣ | * | ␣ | ␣ | ␣ | ␣ | ␣ | ␣ | ␣ | ␣ | ␣ | ␣ | ␣ | ␣ | ␣ | ␣ |

- How can we use the second tape?
  – Copy the 0s from tape 1 and then match 1s on tape 1 with 0s on tape 2

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| tape 1 | ␣ | * | 0 | 0 | 0 | 0 | 0 | 0 | # | 1 | 1 | 1 | 1 | 1 | 1 | ␣ |
| tape 2 | ␣ | * | 0 | 0 | 0 | 0 | 0 | 0 | ␣ | ␣ | ␣ | ␣ | ␣ | ␣ | ␣ | ␣ |

- Only need 2 passes now!
  – Runtime is $O(n)$, but we require better hardware

# Efficiently Solvable Problems: The Class P

- So far, we've seen three algorithms to decide $\{0^{\bullet k}\#1^{\bullet k}\}$
  - Brute-force, $O(n^2)$
  - Smart halving algorithm, $O(nlog_2n)$
  - Fast algorithm using a better architecture, $O(n)$
- Efficiency is all about faster algorithms!
- Contrast with computability
  - There EXISTS NO algorithm for the halting problem!
  - I don't care how many Turing Machines with how many tapes you throw at it
- So when we talk about efficiency, we're talking about decidable problems
- But then you might ask, "Fine, $O(n)$ is faster but is $O(n^2)$ really so bad?"
  - Obviously, it depends on what input sizes you're dealing with
  - If input sizes are in the 1000s, you won't notice a big difference
  - If input sizes are in the 1000000s, you might have to wait a while in one case!

# The Class P, cont'd

- Typically, we measure "fastness" with respect to a target function $f$ that we deem to be sufficiently fast
- What choices of $f$ are there?
  - Linear, quadratic, polynomial, exponential
  - Turns out polynomial is a good compromise

- A Turing Machine is **fast** if the worst case runtime is bounded by a function $f(n)$ which increases by at most a constant factor when you double the size of the input from $n$ to $2n$

$$\text{worst-case runtime} \leq f(n) \quad \text{AND} \quad f(2n) \leq \lambda f(n)$$

# The Class P, cont'd

- *Theorem [Fast means polynomial]*. A Turing Machine $M$ is fast if and only if its worst-case runtime on an input of size $n$ is in $O(n^k)$, for a constant $k$.
  - See book for proof.

- *Definition [The Class P]*. A problem $\mathcal{L}$ is in P if there exists a fast, polynomial-time, Turing machine that decides $\mathcal{L}$. The class P is a set of computing problems, i.e., languages.

- The class P is one of the most important classes in computer science
  - Generally, these are the problems that can be solved for very large inputs
    - Examples include sorting, shortest path, hashing, search…
  - Problems not in P are HARD!!
    - For example, factorization is not believed to be in P
    - The main encryption algorithms only work because we don't know how to quickly factorize very large numbers

# How About Turing Machines with Two Tapes?

- Already saw that a Turing Machine with two tapes gets us from $O(nlog_2n)$ to $O(n)$
- So you should be asking yourselves: what if we define P in terms of two-tape Turing Machines?
  - How about $2^{1000}$ tapes?
- Well, both $O(nlog_2n)$ and $O(n)$ are polynomial, so they're still in P
  - Adding more tapes didn't bring a "qualitative" change
- *Extended Church-Turing Thesis*. Any efficiently solvable problem can be decided by a fast Turing Machine with a single tape. The class P is independent of Turing Machine architecture.
  - (Within limits – if I can choose number of tapes depending on the input size, then the above doesn't hold)
  - Turns out a single-tape Turing Machine can simulate multi-tape machines in polynomial time (see book)
- The class P is robust. Also, Turing Machines are a very general computing framework

# A Decidable Non-Polynomial Problem

- We know that there exists no Turing Machine that can tells us whether a given other Turing Machine will halt
- How about whether another Turing Machine will terminate "fast"?
- Consider the language:

$$\mathcal{L}_{EXP} = \left\{ < M > \#w \middle| M \; accepts \; w \; within \; at \; most \; 2^{|w|} \; steps \right\}$$

  – What is this language?
    - All Turing Machines that run within at most exponential time
  – Is this language decidable?
    - Yes, use our simulator Turing Machine $U_{TM}$
      – Simulate $M$ on each input $w$ for exactly $2^{|w|}$ steps
      – If $M$ terminates, output YES; otherwise output NO
  – Is this language in P?
    - Ah, trickyyy
    - How can you tell if $M$ will terminate if you don't run it for all $2^{|w|}$ steps??

# A Decidable Non-Polynomial Problem, cont'd

- Consider the language:
$$\mathcal{L}_{EXP} = \{< M > \#w \mid M \text{ } accepts \text{ } w \text{ } within \text{ } at \text{ } most \text{ } 2^{|w|} \text{ } steps\}$$
- *Theorem*. $\mathcal{L}_{EXP}$ is not in P.
- *Proof*. By contradiction. Suppose there exists a decider $E_{TM}$ with polynomial worst-case runtime, i.e.,
$$E_{TM} = \begin{cases} ACCEPT & if \text{ } M \text{ } accepts \text{ } w \text{ } within \text{ } at \text{ } most \text{ } 2^{|w|} \text{ } steps \\ REJECT & otherwise \end{cases}$$
  - Let's build our diabolical diagonal Turing Machine $D$ again:

$D$: "Diagonal" Turing Machine derived from $E_{TM}$

**input**: $< M >$ where $M$ is a Turing Machine

1. Run $E_{TM}$ with input $< M > \# < M >$
2. If $E_{TM}$ accepts then REJECT; otherwise ($E_{TM}$ rejects) ACCEPT

# A Decidable Non-Polynomial Problem, cont'd

Rensselaer

- *Proof*. By contradiction. Suppose there exists a decider $E_{TM}$ with polynomial worst-case runtime, i.e.,

$$E_{TM} = \begin{cases} ACCEPT & if\ M\ accepts\ w\ within\ at\ most\ 2^{|w|}\ steps \\ REJECT & otherwise \end{cases}$$

- $E_{TM}$ implies $D$ exists, hence it will appear on the list of all Turing Machines:

$$< M_1 >, < M_2 >, < M_3 >, < M_4 >, < D >, \ldots$$

| $E_{TM}(< M_i > \# < M_j >)$ | $< M_1 >$ | $< M_2 >$ | $< M_3 >$ | $< M_4 >$ | $< D >$ |
|---|---|---|---|---|---|
| $< M_1 >$ | ACCEPT | ACCEPT | REJECT | ACCEPT | ACCEPT |
| $< M_2 >$ | ACCEPT | REJECT | REJECT | ACCEPT | REJECT |
| $< M_3 >$ | REJECT | ACCEPT | REJECT | REJECT | ACCEPT |
| $< M_4 >$ | REJECT | ACCEPT | REJECT | ACCEPT | REJECT |
| $< D >$ | | | | | |

# A Decidable Non-Polynomial Problem, cont'd

- *Proof*. By contradiction. Suppose there exists a decider $E_{TM}$ with polynomial worst-case runtime, i.e.,

$$E_{TM} = \begin{cases} ACCEPT & if\ M\ accepts\ w\ within\ at\ most\ 2^{|w|}\ steps \\ REJECT & otherwise \end{cases}$$

- If $D$ exists, then it will appear on the list of all Turing Machines:

$$< M_1 >, < M_2 >, < M_3 >, < M_4 >, < D >, \ldots$$

| $E_{TM}(< M_i > \# < M_j >)$ | $< M_1 >$ | $< M_2 >$ | $< M_3 >$ | $< M_4 >$ | $< D >$ |
|---|---|---|---|---|---|
| $< M_1 >$ | ACCEPT | ACCEPT | REJECT | ACCEPT | ACCEPT |
| $< M_2 >$ | ACCEPT | REJECT | REJECT | ACCEPT | REJECT |
| $< M_3 >$ | REJECT | ACCEPT | REJECT | REJECT | ACCEPT |
| $< M_4 >$ | REJECT | ACCEPT | REJECT | ACCEPT | REJECT |
| $< D >$ | REJECT | ACCEPT | ACCEPT | REJECT | ??? |

# A Decidable Non-Polynomial Problem, cont'd

- $D(< M_i >)$ does the opposite of $E_{TM}(< M_i > \# < M_j >)$
- Suppose $E_{TM}(< D > \# < D >)$ accepts
  - That means that $D$ accepts (fast) on input $< D >$
  - But $D$ should reject because $E_{TM}(< D > \# < D >)$ accepted
  - **FISHY**!
- Suppose $E_{TM}(< D > \# < D >)$ rejects
  - That means that $D$ either rejects or is slow to accept input $< D >$
    - May or may not accept (no contradiction so far)
  - We know $D$ should accept because $E_{TM}(< D > \# < D >)$ rejected
  - But $E_{TM}$ is fast
    - Note that $D$ runs by simulating $E_{TM}$ on $< D > \# < D >$
    - So the runtime of $D$ is bounded by the runtime of $E_{TM}$, plus the overhead of preparing the input to $E_{TM}$ (polynomial time to copy $< D >$, etc.)
      - The runtime of $E_{TM}$ is at most polynomial
  - **FISHY**!

## Boundary Between Efficient and Inefficient

- Turing Machines are the gold standard for defining solvable and efficiently solvable
  - We have a robust notion of an efficiently solvable problem, the class P
  - There are many interesting problems in P
  - There are problems that are not in P ($\mathcal{L}_{EXP}$)
  - There are problems that we believe are not in P
    - Traveling salesman, factorization, CLIQUE, etc.
    - Instant fame if you can prove this (P vs NP)!
- In practice, efficiency has many dimensions
  - When a problem has no fast solution but still needs to be solved, we use servers, clusters, etc. (salesmen need to travel!)
  - Mobile platforms optimize for battery consumption, at the expense of runtime
  - Distributed platforms spread data across and must solve problems with limited communication
  - Streaming platforms may pre-load data
  - Machine learning applications may need to preserve privacy, fairness, etc.

# . . . the high technology so celebrated today is essentially a mathematical technology.

Rensselaer

"To err is human, but to really foul things up you need a computer." – Paul Ehrlich

- **Mariner rocket explodes (1962).** Formula into code bug resulted in no smoothing of deviations.
- **WWWIII (1983)?** Soviet EWS detects 5 US-missiles (bug detected sunlight reflections).
  - **Luckily** *Stanislav "funny feeling in my gut" Petrov* thought: "surely they'd use more missiles?"
- **Therac 25 (1985).** Concurrent programming bug killed patients through massive 100 × radiation overdose.
- **AT&T Lines Go Dead (1990).** 75 million calls dropped (one line of buggy code in software upgrade).
- **Patriot missile defense fails (1991). 28 soldiers dead, 100 injured** (rounding error in scud-detection).
- **Pentium floating point long-division bug (1993).** Cost: $475 million – flawed division table.
- **Ariane rocket explosion (1996).** Cost: $500 million – overflow in 64-bit to 16-bit conversion.
- **Y2K (1999).** Cost: $500 billion spent because year was stored as 2 digits to save space.
- **Mars Climate Orbiter Crash (1998).** Cost: $125 million lost due to metric to imperial units bug.
- **Tesla Self-Driving Car (2016). 1 dead**. Auto-pilot didn't "see" tractor-trailer. (many more since then)
- **Financial Disasters:** London Stock Exchange down due to single server bug (**2009**; billions of pounds of trading); Knight Capital computer glitch trigers stock sale (**2012**; 500 million lost and Knight's value drops by 75%).
- **Airline Disasters:**
  - AirFrance 447 2009, **228 dead**: pitot-tube failure feeds inconsistent data to programs which then panic pilot.
  - Spanair 5022, 2008, **154 dead**: malware virus.
  - AdamAir 574, 2007, **102 dead**: navigation system errors (and pilot errors).
  - KoreanAir 801, 1997, **228 dead**: ground proximity warning system bug.
  - AeroPerú 603, 1996, **70 dead**: altimeter failures.
  - Scottish RAF Chinook, 1994, **29 dead**: faulty test program
  - AirFrance 296, 1988, **3 dead**: altimeter bug.
  - IranAir 655, 1988, **290 dead**: shot down by US Aegis combat system (misidentified as attacking military plane).
  - KoreanAir 007, 1983, **269 dead**: autopilot took plane into Soviet airspace where it got shot down.
  - Boeing 737 Max, 2018,2019, **346 dead**: attack sensor + algorithm errors.