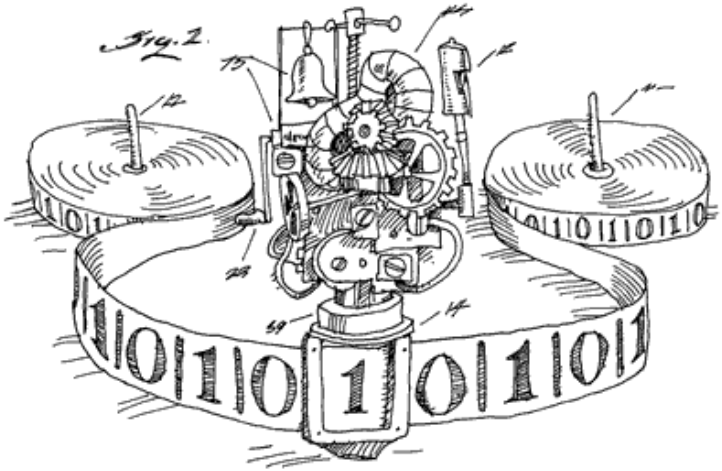
EECS 376: Foundations of Computer Science

Chris Peikert

17 April 2023



EECS M ECE CSE

Big Questions

* **Q:** What is computation, fundamentally?
* **Q:** Which problems are (un)solvable by computers? What are useful techniques for showing this?
* **Q:** Which problems are solvable "efficiently"?
* **Q:** How to relate different problems to each other?
* **Q:** Can we make use of "hard" problems?
* **Q:** How is this useful to me?
  * "Fundamental" knowledge, **rigorous thinking**

EECS ECE CSE

Design & Analysis of Algorithms

* **Algorithm Design:** A set of methods to create algorithms for certain types of problems
* **Examples:** Divide and Conquer, Dynamic Programming, Greedy, …
* **Algorithm Analysis:** A set of techniques to prove correctness of algorithms and determine the amount of resources (e.g., time, memory) they use
* **Examples:** Master Theorem, Potential Method, …

Divide-and-Conquer Algorithms

Main Idea:

1. Divide the input into smaller subinstances
2. Solve each subinstance recursively
3. Combine the solutions in a "meaningful" way

Runtime Analysis:

* Tools to solve recurrence relations
* The "Master Theorem"

Divide-and-Conquer Algorithms

Examples:

* Sorting:

  MergeSort $O(n \log n)$ vs Naïve Sorting $O(n^2)$

* Integer Multiplication:

  Karatsuba $O(n^{\log_2 3})$ vs Naïve Multiplication $O(n^2)$

* Closest Pair:

  D&C $O(n \log n)$ vs Naïve $O(n^2)$

* …

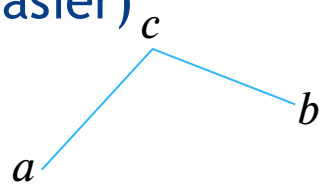
Quote of The Day

"If you can solve it, it is an exercise; otherwise it is a research problem"

– Richard E. Bellman,
Inventor of Dynamic Programming

Dynamic Programming

High-level Idea: Break a problem instance into smaller (easier) subinstances subject to:



1. Optimal substructure:
   any "piece" of an optimal solution is itself optimal.

Example: Any subpath of any shortest path is itself a shortest path.

2. Overlapping sub-problems: smaller subinstances occur more than once in the breakdown

Example: When computing the Fibonacci sequence using $F_n = F_{n-1} + F_{n-2}$, the same values will re-occur many times.

Dynamic Programming

* **Examples:**
  * Longest Common Subsequence (LCS)
  * Longest Increasing Subsequence (LIS)
  * Floyd–Warshall algorithm (All-Pairs Shortest Paths)
  * …
* **Implementations:**
  * Recursive: “TOP-DOWN” – often exponential time
  * Table: “BOTTOM-UP” – often time efficient
  * Memoization: “BOTTOM-UP” in “TOP-DOWN” clothes

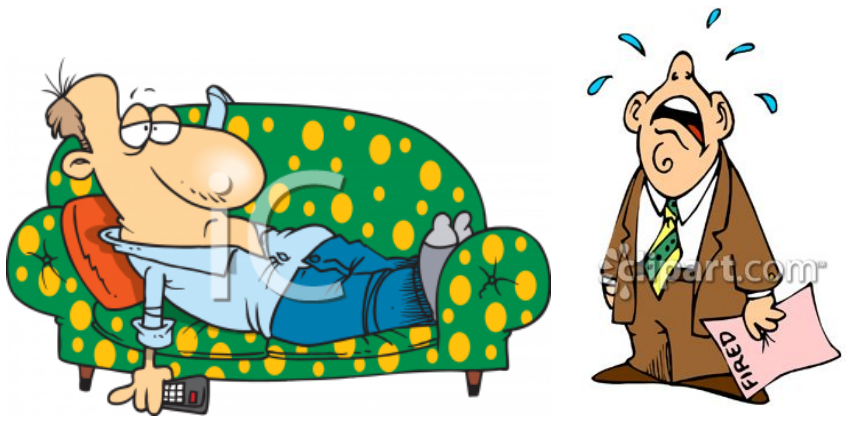
Quote of The Day

"I believe that the question:
'Can machines think?'
is too meaningless to deserve discussion."

– Alan Turing

Computability

* **Question:** Are there problems computers can't solve?
* **Example:** Can we write a compiler that tests whether two given programs (in C/C++/Java) have the same behavior?
* **Application:** For lazy EECS 280 graders…
* **Answer:** NO!



Computability: Review

* **Question:** Which problems are solvable by a computer?
* **Answer:** Depends on what *solvable* and *computer* means.
* **Definition:** A program M ***decides*** a language A if given x as input:
  * If $x \in A$, M *accepts* x ("return 1'')
  * If $x \notin A$, M *rejects* x ("return 0'')
* **Remark:** M is called a ***decider***, halts on any input; A is ***decidable***.
* **Definition:** The ***language*** of M, $L(M) = \{x : M \text{ accepts } x\}$
* **Definition:** M ***recognizes*** a language A if $A = L(M)$. In other words:
  * If $x \in A$, M *accepts* x
  * If $x \notin A$, M *either rejects* x *or* loops on x
* **Remark:** M is called a ***recognizer***, *might not* halt on every input.

Language Reducibility

* **Question:** How do we show that there are undecidable languages?
* **Fact 1:** There are countably many Turing Machines.
* **Fact 2:** There are uncountably many Languages.
* **Conclusion:** There are "more" languages than Turing machines. Therefore, there exist undecidable (and unrecognizable) languages.
* **Definition:** Language A is ***Turing reducible*** to language B, written $A \leq_T B$, if there exists a program M that decides A using a membership oracle ("black box") for B.
* **Theorem:** If $A \leq_T B$ and A is undecidable, then so is B.
* **Explicit undecidable languages:** L_{ACC}, L_{HALT}, L_{EQ}

Applications

* ***Full Employment Theorem for Compiler Writers:*** There does not exist a perfect size-optimizing compiler.
* **Alternate Version:** There is no perfect program analyzer.
* **More undecidable problems:**
  * $L_{\text{FORMAT}} = \{\langle M \rangle : M \text{ will erase your hard drive}\}$
  * $L_{\text{NOMONEY}} = \{\langle M \rangle : M \text{ will donate all your money to charity}\}$
  * $L_{\text{SUBMIT}} = \{\langle M \rangle : M \text{ will submit my HW with a perfect grade}\}$
* **Conclusion:** Many fundamental problems in CS are undecidable!
* **Industry Implications:** Intel, Google, IBM have entire divisions dedicated to tackling these problems!

Final Review: Complexity

What does "efficiently" mean?

* **Question:** Which tasks can computers do **efficiently**?
* **Example:** Can a computer produce a good schedule for the next academic year, in a timely manner?
* **Application:** Scheduling (university, trains, etc.)

EECS M ECE CSE

The Class P

* **Definition:** **P** = the set of all languages that can be decided by a TM in time polynomial in the input size.
* **Formally:**

$$\mathbf{P} = \bigcup_{k \geq 1} DTIME(n^k)$$

* The class **P** = the class of ***efficiently decidable*** languages
* **Properties:**
  * Model independent: can replace TM with any "realistic" (deterministic) model.
  * Composition: if an efficient program M calls an efficient subroutine M' then the whole procedure is efficient.
* **Definition:** Efficient = Polynomial Time in the *input size*

Efficiently Decidable vs Efficiently Verifiable

* **Formally:** Let L be a language.
* $L \in \mathbf{P}$ if there exists a poly-time algorithm $M(x)$ such that:
  * $x \in L \iff M(x)$ accepts
* $L \in \mathbf{NP}$ if there exists a poly-time (in $|x|$) algorithm $V(x, c)$ such that:
  * $x \in L \iff V(x, c)$ accepts for some c
* **Conclusion:** $\mathbf{P} \subseteq \mathbf{NP}$ (Proof: $V(x, c)$ ignores c, runs $M(x)$.)
* **$1,000,000 question:** Is $\mathbf{P} = \mathbf{NP}$? *We do not know.*

Examples of NP Languages

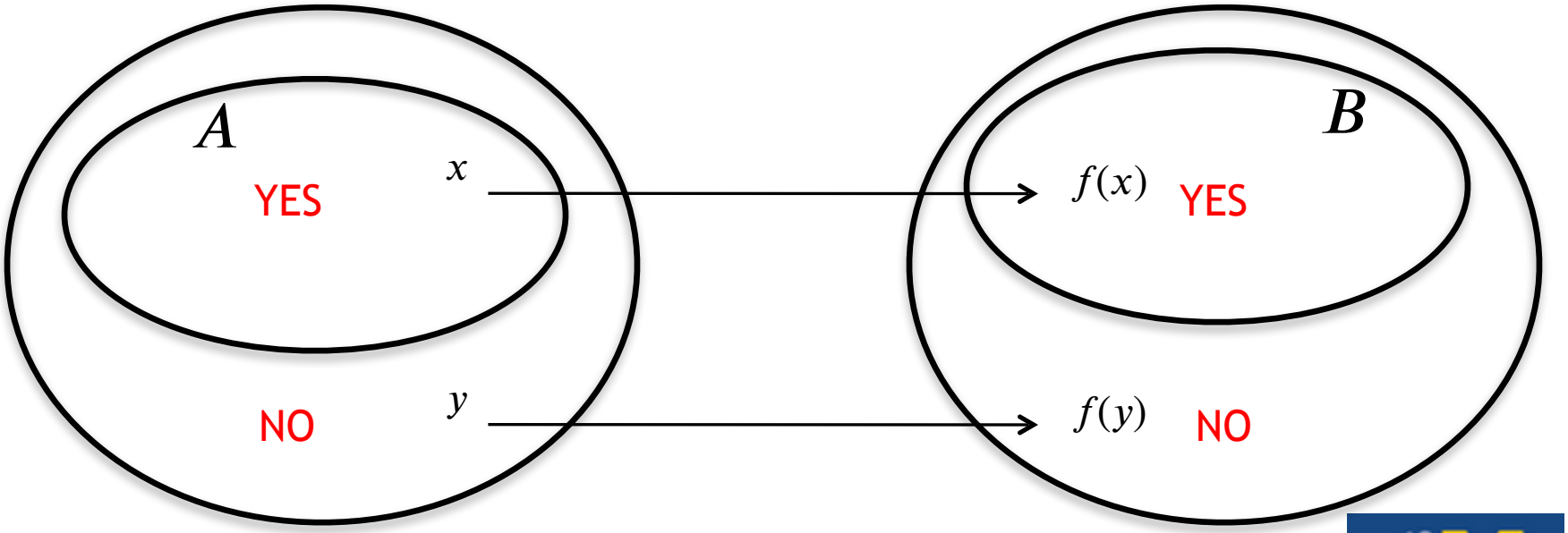
* $\text{MAZE} = \{(G, s, t) : G \text{ is a graph. There is a path } s \to t \text{ in } G\}$
* $\text{TSP} = \{(G, k) : G \text{ is weighted graph. } G \text{ has a tour of weight at most } k\}$
  * **Certificate:** Sequence of vertices; verifier checks conditions
* $\text{CLIQUE} = \{(G, k) : G \text{ has a clique of size } k\}$
* $\text{VERTEXCOVER} = \{(G, k) : G \text{ has a vertex cover of size } k\}$
  * **Certificate:** A set of vertices; verifier checks conditions
* $\text{SAT} = \{\phi : \phi \text{ is a satisfiable formula}\}$
  * **Certificate:** An assignment; verifier evaluates ϕ on it

Poly-Time Mapping Reductions

* **Theorem [Cook-Levin]:** For every language $L \in \mathbf{NP}$, there exists a polynomial-time algorithm f such that:
  * $x \in L \iff f(x) \in \mathrm{SAT}$
* **Definition:** Language A is ***polynomial-time mapping reducible*** to language B, written $A \leq_p B$, if there is a polynomial-time algorithm f such that:
  * $x \in A \iff f(x) \in B$
* **Recall:** If $A \leq_T B$ and B is decidable, then so is A.
* **Theorem:** If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.

$A \leq_p B$

Σ^* f is poly-time computable Σ^*



* **Remark:** f need not be onto B, nor 1-to-1!

NP-Hardness/-Completeness

* **Intuition:** Problems "at least as hard as" any in **NP**
* **Formal Definition:** A language A is **NP-*Hard*** if:
  * For every language $L \in \mathbf{NP}$, $L \leq_p A$.
* **Formal Definition:** A language A is **NP-*Complete*** if:
  1. $A \in \mathbf{NP}$
  2. A is **NP**-Hard
* **Conclusion:** A poly-time algorithm for *any one* **NP**-Hard language implies that $\mathbf{P} = \mathbf{NP}$.
* **Theorem [Cook-Levin]:** SAT is **NP**-Complete.

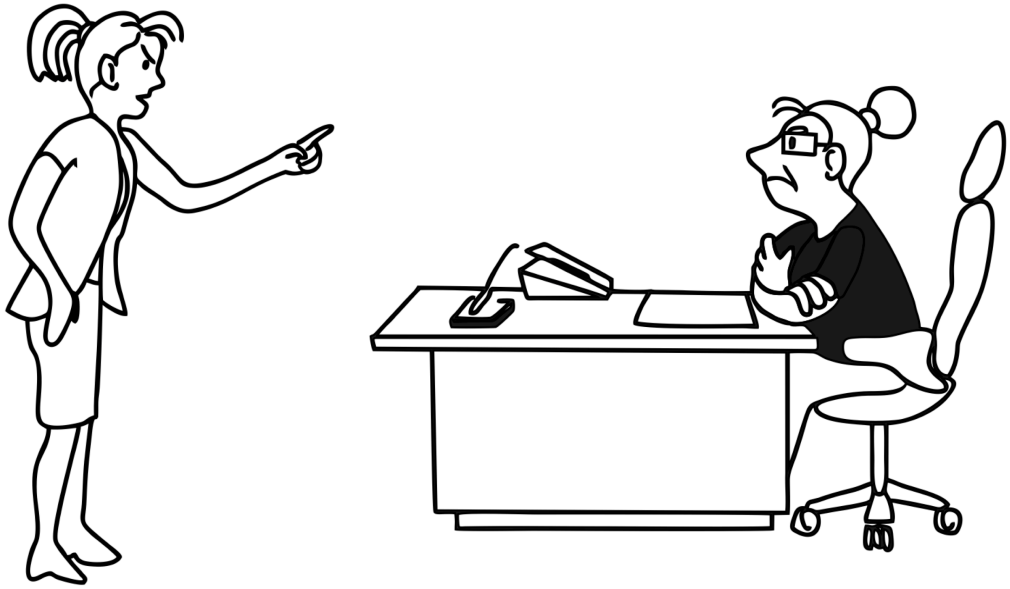
NP-Completeness is Everywhere

* **Constraint Satisfaction:** CSAT, SAT, 3SAT
* **Covering Problems:** Vertex Cover, Set Cover
* **Coloring Problem:** 3-colorability of a graph
* **Scheduling Problems:** best schedule for classes
* **Model Checking:** context-bounded reachability
* **Social Networks:** Clique, Max-Cut
* **Routing:** Long-Path, HAMPATH, TSP
* **Games:** Sudoku, Battleship, Tetris, Mario, Pokémon
* … P vs NP: One ALGORITHM to rule them all?

NP-Completeness

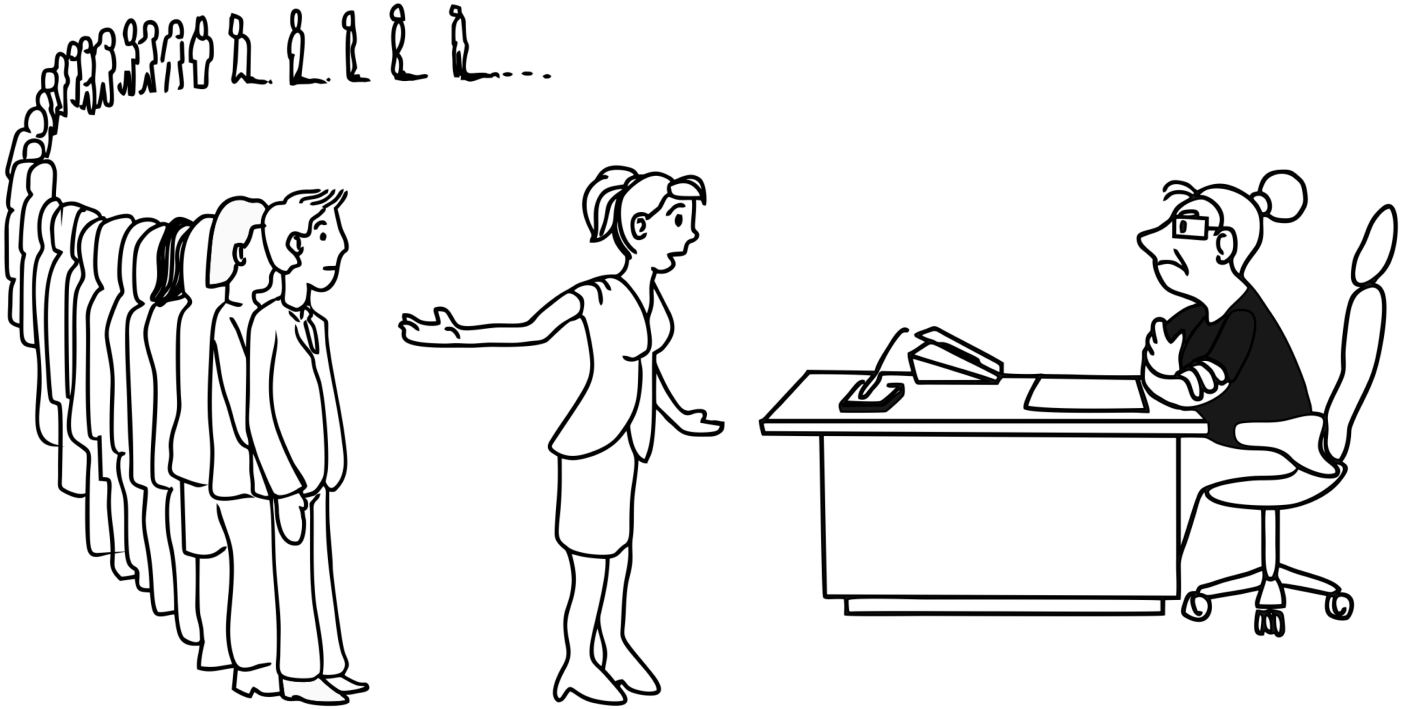
“I can’t find an efficient algorithm, I guess I’m just too dumb.”

NP-Completeness



“I can’t find an efficient algorithm, because no such algorithm is possible!”

NP-Completeness



"I can't find an efficient algorithm, but neither can all these famous people."

Beyond Decision: Search Problems

* **Types of Search Problems:**
  * **Maximization:** Maximum Clique, Knapsack
  * **Minimization:** Minimum Vertex Cover, TSP
  * **Exact:** SAT, HAMPATH
* **Decision/Limited-Budget version:**
  * Does G have a clique of size (at least) k?
  * Does G have a vertex cover of size (at most) k?
  * Is ϕ satisfiable?
* **Informal Theorem:** An **NP**-Hard search problem has an efficient algorithm if and only if its decision version does.
* **Proof Idea:**
  * Step 1: Find the size k of an (optimal) solution.
  * Step 2: Find a solution of size k by modifying the input.

Approximation to Search Problems

* An algorithm is an ***α-approximation*** ($\alpha > 1$) for a *minimization* problem if it outputs an answer that is *at most* α times optimal.
  (e.g., $|S| \leq \alpha |S^*|$, where S^* is an arbitrary vertex cover)
* An algorithm is an ***α-approximation*** ($\alpha < 1$) for a *maximization* problem if it outputs an answer that is *at least* α times optimal.
  (e.g., $|S| \geq \alpha |S^*|$, where S^* is a largest max-cut)
* α is the ***approximation ratio***
  * The closer α is to 1, the better.

EECS M ECE CSE

Types of Approximation Algorithms

* **Greedy algorithms:** Smart Greedy, Dumb Greedy,..
* **Smart Greedy:** ½-approx for Knapsack
* **Ad-hoc algorithms:** 2-approx for Min-VC
* **Local-search algorithms:** ½-approx for Max-Cut
* **Randomized algorithms:** ½-Max-Cut, 7/8-Max-E3SAT
* **Remark:** The approximation guarantee is with respect to an *optimal* solution (which often can't be computed efficiently)
* Typically, this is shown via an appropriate proxy.

Quote of The Day

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin"

– John von Neumann

Randomized Approximations

* **Definition:** Max-E3SAT problem: given an ***exact*** 3CNF formula $\phi(x_1, \ldots, x_n)$ as an input, satisfy as many clauses as possible.
* **Theorem:** There exists an efficient randomized algorithm that given an exact 3CNF formula $\phi(x_1, \ldots, x_n)$ as input, satisfies 7/8 of the clauses ***in expectation***.
* **Proof:** Suppose ϕ has m clauses. Assign each x_i at random.
* Define $Z =$ number of clauses satisfied by the algorithm.
* For each clause $1 \leq i \leq m$ define an indicator random variable:
* $$Z_i = \begin{cases} 1, & \text{if the } i\text{'th clause is satisfied by the assignment} \\ 0 & \text{otherwise} \end{cases}$$
* $\mathbb{E}[Z] = \mathbb{E}[Z_1] + \ldots + \mathbb{E}[Z_m] = 7/8 \bullet m$

Randomized Algorithms/DS

* Randomness can be exploited to design algorithms and data structures that are simple to implement but give good performance in expectation.
  * **Examples:** Quick Sort, Skip Lists
* **Common Analysis:**
  * Define a random variable X that counts something (e.g. number of comparisons, number of levels).
  * Define indicators X_i and $X = X_1 + \ldots + X_n$.
  * Compute $\mathbb{E}[X_i]$, apply linearity of expectation to obtain $\mathbb{E}[X]$.

Chernoff-Hoeffding Bounds: Deviation From Expectation

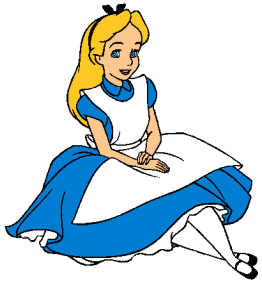
* Let $Y = Y_1 + \ldots + Y_n$ be the sum of independent indicator random variables where $\mathbb{E}[Y_i] = \Pr[Y_i = 1] = p$.
  * Then $\mathbb{E}\left[\frac{1}{n}Y\right] = p$
* **Theorem:** Let $\varepsilon > 0$ then:
* **Upper tail:** $\Pr\left[\frac{1}{n}Y \geq p + \varepsilon\right] \leq e^{-2\varepsilon^2 n}$
* **Lower tail:** $\Pr\left[\frac{1}{n}Y \leq p - \varepsilon\right] \leq e^{-2\varepsilon^2 n}$
* **Combined:** $\Pr\left[\left|\frac{1}{n}Y - p\right| \geq \varepsilon\right] \leq 2e^{-2\varepsilon^2 n}$
* **Observation:** The probability of deviation decays *exponentially* in the number of samples.

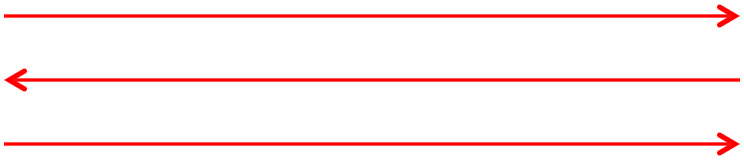
EECS M ECE CSE

Final Review: Cryptography

* **Question:** If a problem is notoriously hard, can we make some good of it?

Establishing a Secret Key

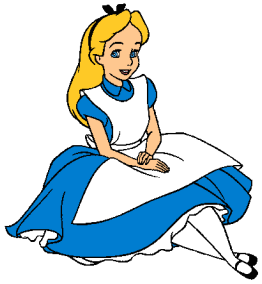




Problem Setup: Alice and Bob want to communicate, but there is an eavesdropper Eve who listens in on their communication.

Question: Many encryption schemes require a pre-shared secret key. How can those who never met establish one?

Diffie-Hellman: Protocol



$x = (g^a \bmod p)$

$y = (g^b \bmod p)$

System parameters: a huge prime p, a generator g of $\mathbb{Z}_p^*$

Alice picks $a \in \mathbb{Z}_p^*$ *at random* and sends $x = (g^a \bmod p)$ to Bob.

Bob picks $b \in \mathbb{Z}_p^*$ *at random* and sends $y = (g^b \bmod p)$ to Alice.

Their secret shared key is $k = (g^{ab} \bmod p)$.

Alice *locally* computes: $y^a \equiv (g^b)^a \equiv g^{ba} \pmod p$.

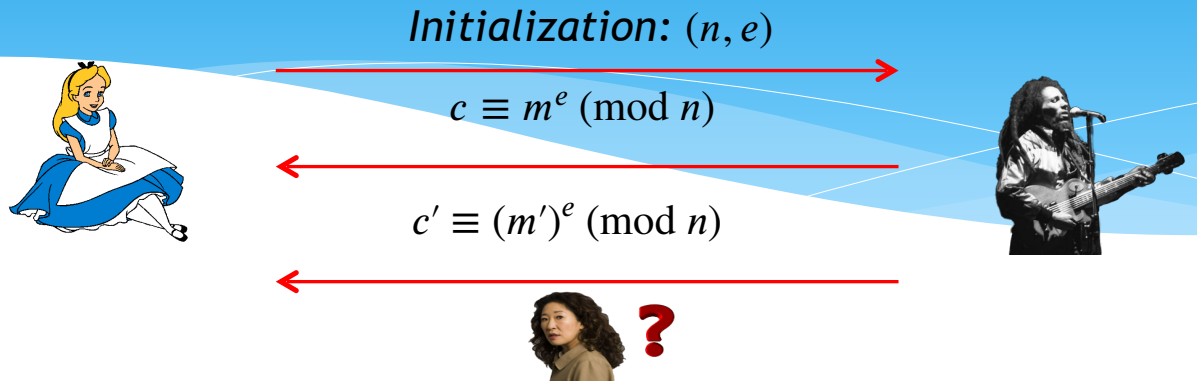
Bob *locally* computes: $x^b \equiv (g^a)^b \equiv g^{ab} \pmod p$.

Key: These are equal.

Public-Key Encryption: A Mathematical Lock

* **Main idea:** Every user has two keys: a ***public*** key and a ***private*** key.
* **Public key:** ("the lock") – public knowledge, used to encrypt messages
* **Private key:** ("the key") – known only to the owner, used to decrypt messages encrypted under the public key

RSA: Protocol



* To initialize the protocol, **Alice:**
  * picks two *large*, *secret* primes p, q and sets $n = pq$
  * generates ***matching public/private exponents*** (e, d)
    * $e \cdot d = 1 + k(p-1)(q-1)$ for some integer $k \geq 1$
  * sends Bob (n, e) (public modulus and exponent)

Alice initializes the "lock" and "key" and gives the lock to Bob; only need to do this *once*

* To send m to Alice, Bob sends the ciphertext:

$$c \equiv m^e \pmod{n}$$

* After receiving c, Alice computes:

$$c^d = m^{e \cdot d} = m^{1+k(p-1)(q-1)} \equiv m \pmod{n}$$

RSA Identity

Factoring is Hard

1024 bits, 309 digits

* **RSA $100,000 challenge:** factor the following modulus n into two large primes:
* n=135066410865995223349603216278805969938881475605667027524485143851526510604859533833940287150571909441798207282164471551373680419703964191743046496589274256239341020864383202110372958725762358509643110564073501508187510676594629205563685529475213500852879416377328533906109750544334999811150056977236890927563

All / Some / None

* **Question:** All/Some/No languages in **NP** are decidable.
* **Answer:** All.
* **Question:** All/Some/No **NP**-Hard languages are decidable.
* **Answer:** Some.
* **Question:** All/Some/No grades on the final exam will be at least average, if they are not all the same.
* **Answer:** Some.
* **Question:** All/Some/No **NP**-Complete problems are in **NP**.
* **Answer:** All.
* **Question:** All/Some/No RSA moduli are prime.
* **Answer:** No.

True / False / Unknown

* **Question:** There exists an algorithm for integer factorization.
* **Answer:** True.
* **Question:** There exists an efficient (classical) algorithm for integer factorization.
* **Answer:** Unknown.
* **Question:** The person in the photo is Alan Turing.

* **Answer:** False.
* **Why:** The person in the photo is Benedict Cumberbatch.

EECS ECE CSE

Your Journey Continues…

* **Design & Analysis of Algorithms** –
  EECS 477, EECS 498-XX, EECS 586, MATH 416
* **Complexity and Approximation** –
  EECS 574, EECS 598-XX
* **Randomness and Computation** – EECS 572
* **Cryptography** – EECS 475, EECS 575

Thanks!

* GSIs/IAs: Aditya, Ajay, Ankith, Ben, Chad, Claire, Cooper, Daniel, Daphne, Frankie, Julianne, Junghwan, Lily, Liz, Nikki, Peter, Sam, Samantha, Samin, Siddharth, Teoh
* Graders too!
* And you! Have a great summer.