Class CS 6903, Lecture n. 2

- Welcome to Lecture 2!
- In Lecture 1 we studied:
  - Classical Cryptography, Encryption with Perfect Secrecy

Summary of Lecture 2

- More Background on Algorithms
- Some Background on Complexity Theory
- Modern Cryptography principles
- One-way functions
- Trapdoor functions
- Applications to constructing a public-key cryptosystem
- Implementation aspects

Algorithms: input size and running time

- Input size: number of bits needed to represent input value, using a conventional encoding scheme
  - Examples:
    - integers in [0,n-1] can be represented using ~ log n bits
    - k-degree polynomials with coefficients in [0,n-1] can be represented using ~ (k+1) log n bits
    - n-node graphs can be represented using n^2 bits
- Running time (of a given algorithm A on a given input x): number of A's primitive operations executed when run on input x, typically evaluated as a function of the length of x, denoted as |x|

Worst-case and average-case

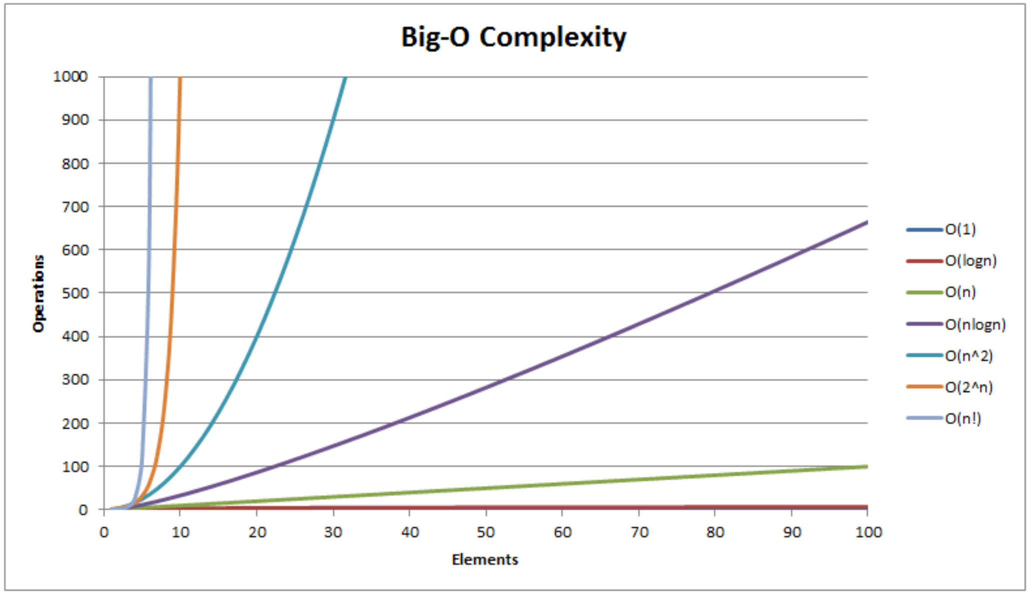
- Previous running time definition refers to the algorithm's performance on a single input.
- Worst-case running time: max (over all inputs) running time, expressed as a function of input size
  - Example:
    - Searching a value in an n-element array may take up to n comparison instructions $\forall c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n)$
- Average running time: average (over all inputs) running time, expressed as a function of input size
  - Example:
    - Searching a value in an n-element array will take, on average, n/2 comparison instructions
- Note:
  - Gap between average and worst-case can be much higher
  - Best case running time is rarely of interest

Asymptotic running time

- Order of growth: running time of algorithm, as input size grows
- Asymptotic notations: (most used is O)
  - f(n)= ω(g(n)) if $\forall c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq c \cdot g(n) < f(n)$
    - Order of growth of f(n) > order of growth of g(n)
    - limit for n→ ∞ of f(n)/g(n) is = ∞
  - f(n)= o(g(n)) if $\forall c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n)$
    - Order of growth of f(n) < order of growth of g(n)
    - limit for n→ ∞ of f(n)/g(n) is = 0
  - f(n)= Θ(g(n)) if $\exists c_1 > 0, \exists c_2 > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$
    - Order of growth of f(n) = order of growth of g(n)
    - limit for n→ ∞ of f(n)/g(n) is = constant
  - f(n)= Ω(g(n)) if $\exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)$
    - Order of growth of f(n) >= order of growth of g(n)
  - f(n)= O(g(n)) if $\exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)$
    - Order of growth of f(n) <= order of growth of g(n)

Examples of Algorithms - Searching

- Searching (one object among n):
  - Sequential: scan all n objects until desired one is found (if at all) → time $\Theta(n)$
  - Binary: if n objects are ordered, compare desired object with middle element, and continue search on left or right half only → time $\Theta(\log n)$
- Searching (one n-bit string satisfying a certain condition among all n-bit strings):
  - Sequential → time ~ $\Theta(2^n)$
  - Binary → time ~ $\Theta(n)$
- Searching (one n-bit string satisfying a certain condition among all in a subset M):
  - Sequential → time ~ $\Theta(|M|)$
  - Binary → time ~ $\Theta(\log |M|)$
- Note:
  - $\log n = o(n)$ (or, $n = \omega(\log n)$),
  - and $n = o(2^n)$ (or, $2^n = \omega(n)$)



Question set 3

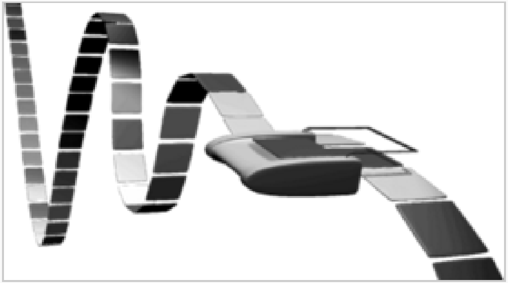
- For which X in {o, O, ω, Ω, Θ}, does f(n)=X(g(n)) hold, when:
  - $f(n) = n^2$, $g(n) = n$
  - $f(n) = 100n^{2.233}$, $g(n) = n^{2.234}$
  - $f(n) = (\log n)^{100}$, $g(n) = n^{1/100}$
  - $f(n) = n^{100}$, $g(n) = 2^n$
- Other questions on asymptotic notation:
  - Do additive or multiplicative constants to f(n) or g(n) matter when determining the asymptotic notation between them?
  - If f(n) = O(g(n)), for which X in {o, ω, Ω, Θ}, does g(n)=X(f(n)) hold?
  - If f(n) = o(g(n)), for which X in {O, ω, Ω, Θ}, does g(n)=X(f(n)) hold?
  - If f(n) = O(g(n)) and f(n) = Ω(g(n)) , for which X in {o, ω, Θ}, does f(n)=X(g(n)) hold?

Summary of Lecture 2

- More Background on Algorithms
- Some Background on Complexity Theory
- Modern Cryptography
- One-way functions
- Trapdoor functions
- Applications to constructing a public-key cryptosystem
- Implementation aspects

Complexity theory background: basics

- Complexity theory: branch of computer science and mathematics dealing with whether and how efficiently problems can be solved on a model of computation, using an algorithm.
- Model of computation: Turing machine
  - simple to formulate
  - can be used to prove results
  - it represents what many consider the most powerful possible "reasonable" model of computation

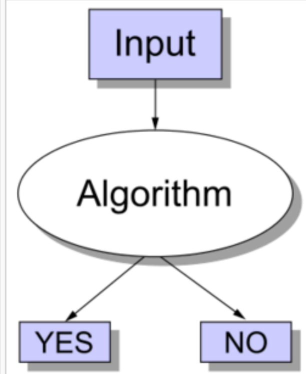


An artistic representation of a Turing machine

- Expressing problems in this model:
  - Which problems? → Decision, Optimization problems
  - Which language? → Language Theory

Complexity theory background: problems

- Language Theory: a language L is a set of binary strings; ex:
  - EVEN = { strings with an even number of 1's }
  - PRIMES = { strings denoting prime integers }
  - CONNECTED = { strings denoting a connected graph }
- Decision Problem: is a string x in language L?

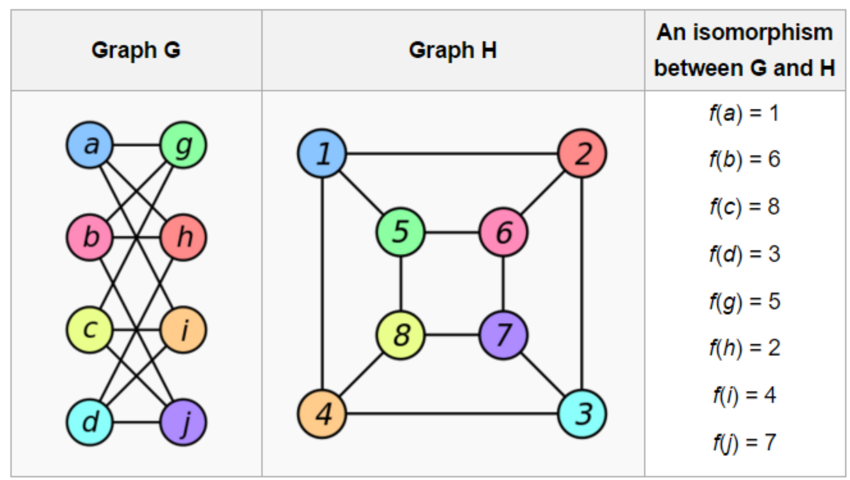


A decision problem has only two possible outputs, *yes* or *no* (or alternately 1 or 0) on any input.

- Optimization Problems can be solved by solving "related" decision problems
- Instance Encoding: choose a convention, and (say that you) use it
  - Integers <n can be represented using ~ log n bits

Complexity theory background: classes

- A t(n)-time algorithm is an algorithm with time complexity O(t(n))
- A polynomial-time algorithm is a t(n)-time algorithm where $t(n)=n^c$ for some constant c
- P is the class of decision problems that can be solved by a polynomial-time algorithm
  - Intuition: problems in P can be efficiently solved by today's computers
  - Examples: searching, sorting, primes, etc.
- NP is the class of decision problems that can be verified by a polynomial-time algorithm
  - Given the problem "is x in L?", |x|=n, the answer is yes if and only if there is a witness w of length polynomial in n, and a polynomial-time algorithm A that, given x,w, verifies that x is in L
    - A(x,w)=1 (for "yes") if x is in L
    - For any (poly-long) w', A(x,w')=0 (for "no") if x is not in L
  - The witness w is like a proof and A acts like a verifier
  - Examples: graph isomorphism, hamiltonian graphs, etc.
  - Intuition: problems in NP (most likely) cannot be efficiently solved by today's computer; however they can efficiently proved from one party (with w) to another



Complexity theory background: P vs NP

- Fundamental question in computer science: is P = NP?
  - Solving it will get you 1M$ (http://en.wikipedia.org/wiki/Millennium_Prize_Problems)
  - If you have ideas let me know!!! ☺

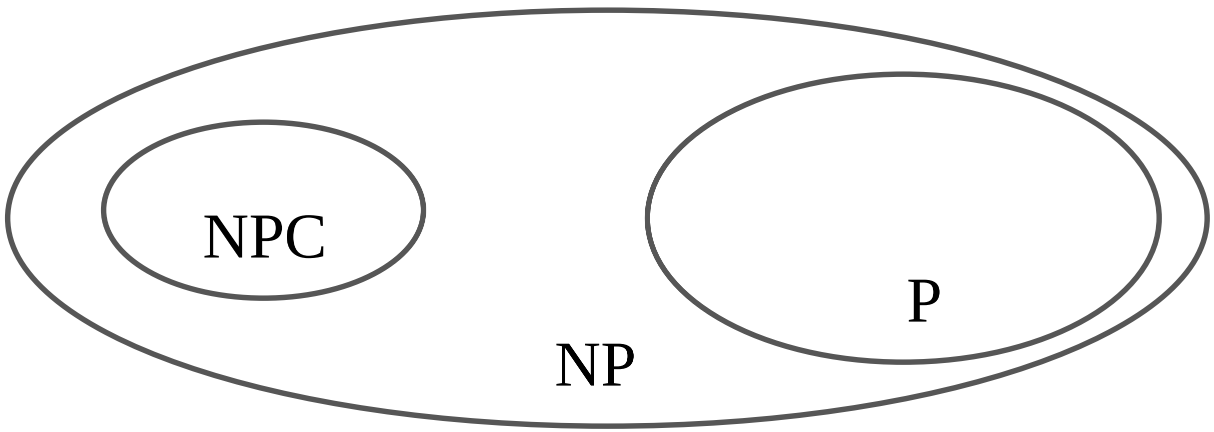
- Note: P is included in NP
  - Verifier is the polynomial-time algorithm itself
- But significant evidence points to the fact that there exist (even concrete) problems in NP that are not in P
  - Many researchers tried to exhibit polynomial-time algorithms for many problems in NP
  - Currently hottest question is actually: is BPP = NP?
    - Here, BPP is the probabilistic analogue of P

Complexity theory background: reductions

- Let L_1 and L_2 be two languages (decision problems)
- L_1 is reducible in polynomial-time to L_2, denoted as $L_1 \leq L_2$, if there is an algorithm that solves L_1 using, as a subroutine, an algorithm for solving L_2 and which runs in polynomial time if the algorithm for L_2 does
  - Informally, if $L_1 \leq L_2$, then L_2 is at least as difficult as L_1, or, equivalently, L_1 is no harder than L_2
- L1 and L2 are said to be computationally equivalent if $L_1 \leq L_2$. and $L_2 \leq L_1$
- Note:
  - If $L_1 \leq L_2$ and $L_2 \leq L_3$, then $L_1 \leq L_3$
  - If $L_1 \leq L_2$ and L_2 is in P, then L_1 is in P

Complexity theory: NP completeness

- Definition: A language L is NP-complete if
  - L is in NP, and
  - $L_1 \leq L$ for every L_1 in NP
- NP-complete languages (decision problems) are the hardest in NP, in that they are at least as difficult as every other language (decision problem) in NP
- There are thousands of problems from diverse fields such as combinatorics, number theory, and logic, that are known to be NP complete
  - Adds significant evidence to P being smaller than NP
  - Conjectured status is as below (NPC = set of NP complete languages):



Question set 4

- Which of the following languages are in P, NP, NPC ?
  - EVEN
  - CONNECTED
  - (PAIRS OF) ISOMORPHIC GRAPHS
  - HAMILTONIAN GRAPHS
- Questions on reducibility:
  - Without any further assumptions on L_1, L_2, can one say that $L_1 \leq L_2$ implies $L_2 \leq L_1$?
  - State an assumption on L_2 for which we can say that $L_1 \leq L_2$ suffices to imply $L_2 \leq L_1$.
  - Assume $L_1 \leq L_2$, $L_2 \leq L_3$, …, $L_{m-1} \leq L_m$. For which values of m (as a function of the instance length n) , can we say that L_m in P implies L_1 in P?

Summary of Lecture 2

- More Background on Algorithms
- Some Background on Complexity Theory
- Modern Cryptography principles
- One-way functions
- Trapdoor functions
- Applications to constructing a public-key cryptosystem
- Implementation aspects

Modern Cryptography

- Modern Cryptography (30+ years):
  - Transitions cryptography from an art to a science
  - Gives hope to break cycle of designing and breaking schemes
  - Significantly enlarged its scope
    - Numerous additional goals (other than private communication)
  - Scientific discipline based on mathematically rigorous:
    - security notions
    - design requirements
    - model and complexity assumptions
    - solution techniques
    - security proofs

Security notions and design requirements

- In modern cryptography formal definitions of security notions are essential prerequisites for the design, usage or study of any cryptographic primitive or protocol
  - Design: knowing our goal security notions helps improving our design capabilities
  - Usage: if solutions come together with associated security notions, in applications we can more easily choose which solutions should be used
  - Study: security notions give a new way to analyze or compare different solutions
- Design requirements for a given application can be formally defined as cryptographic schemes with appropriate groups of security notions
- A cryptographic scheme for some real-life application is secure if no adversary with a given set of resources can achieve a certain break
- How is this relevant (if at all) to real life? Approaches:
  - Intuition
  - Comparing security notions
  - Showing consistent real-life examples

Model and complexity assumptions

- In modern cryptography, we try to design solutions within rigorously specified interaction models and complexity assumptions
- Models:
  - E.g.: for encryption, we formally define an interaction model characterizing the communication between Alice and Bob
- Assumptions:
  - Given complexity theory, we can use problems in NP (and supposedly not in P) to design schemes in a way that the adversary's task of breaking the scheme is at least as hard as these problems (using reductions)
  - Note that for the one-time pad scheme we used no complexity theory assumptions
    - But complexity assumptions will improve our understanding and solutions

Solution techniques and security proofs

- In modern cryptography, we try to design formally and rigorously specified solutions with formal and rigorous security proofs within the mentioned interaction models and complexity assumptions
- Solutions:
  - E.g.: for encryption, we formally define the algorithms that Alice, Bob have to run and the (arbitrary) one that an adversary could run
- Proofs:
  - Given complexity theory, we can use problems in NP (and supposedly not in P) to prove the security of the schemes
  - E.g.: for encryption, we could formally prove that if an adversary recovers the plaintext from the ciphertext of a given scheme, then we can transform the adversary's algorithm into a related one that solves the problem in NP

Question set 5

- Consider the following types of cryptographic schemes:
  - One of the ciphers from Lecture 1 (e.g., shift cipher, etc.)
  - The one-time pad
  - A scheme constructed using principles in modern cryptography

  Consider if and how these features apply to these schemes:
  - Security notions
  - Design requirements
  - Model assumptions
  - Complexity assumptions
  - Rigorously specified solutions
  - Formal proofs

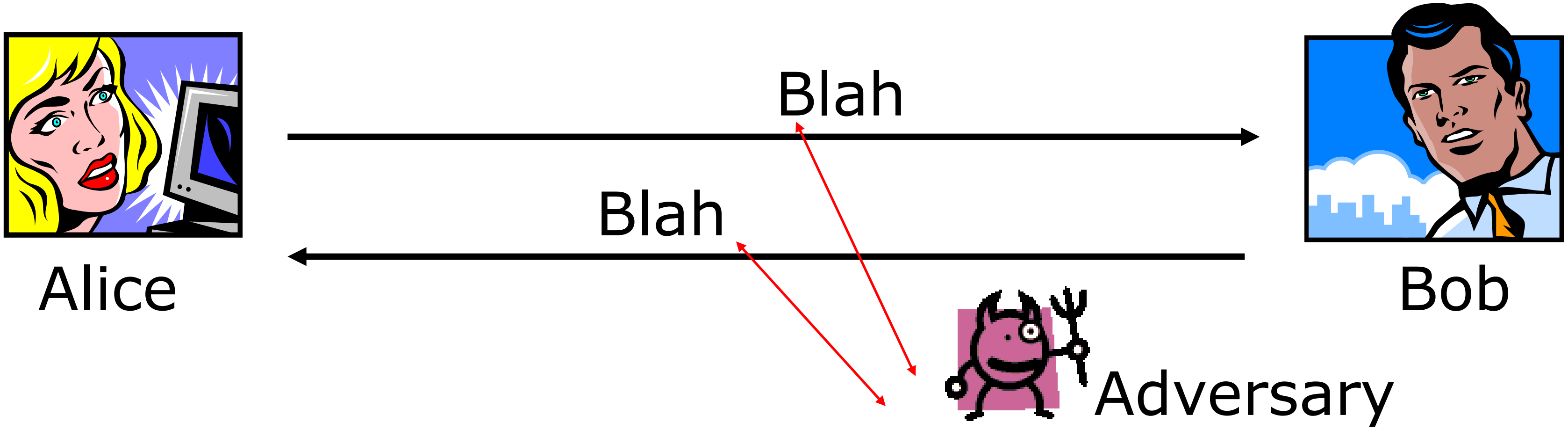
Summary of Lecture 2

- More Background on Algorithms
- Some Background on Complexity Theory
- Modern Cryptography principles
- One-way functions
- Trapdoor functions
- Applications to constructing a public-key cryptosystem
- Implementation aspects

Foundations of Modern Cryptography

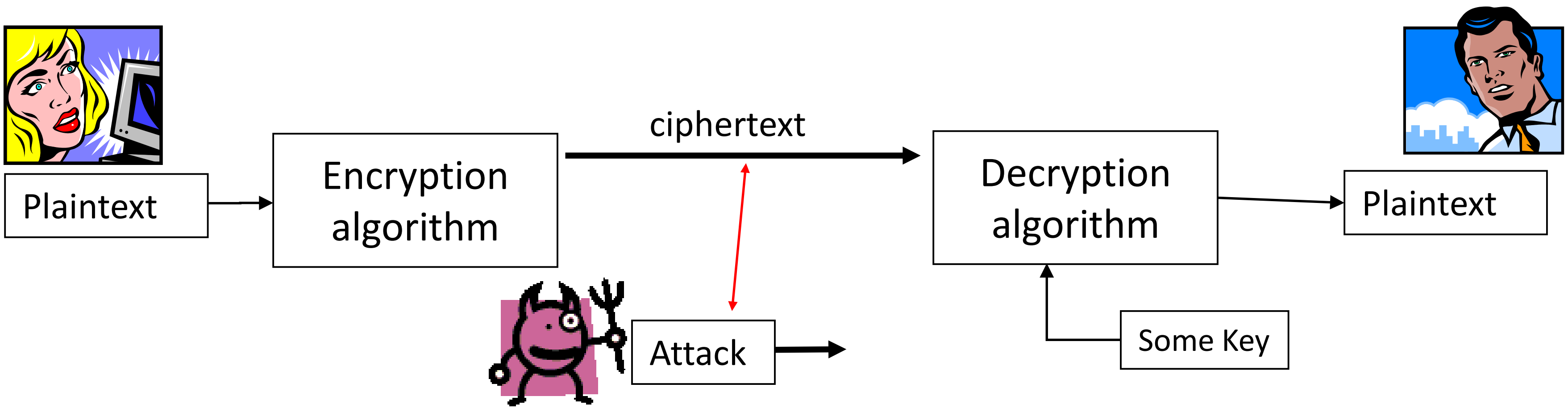
- Basic cryptographic primitives
  - Tools believed to be minimal / essential to achieve cryptographic goals
- Examples:
  - One-Way Functions
  - Trapdoor Functions
  - Pseudo-Random Generators
  - Pseudo-Random Functions
  - Zero-Knowledge Protocols

Modern Cryptography: main problem



- Main problem statement: Alice and Bob may not know each other but may still want to exchange messages privately from any eavesdropping adversary
  - Alice and Bob may not share a cryptographic key, as done in perfect secrecy encryption solutions
- But perfect secrecy required a shared key at least as long as plaintext! → we cannot hope to achieve perfect secrecy

Modern Cryptography: the new approach



- Approach:
  - Alice and Bob need to run efficient encryption and decryption algorithms
  - The adversary's task to, say, obtain the plaintext from the ciphertext, should be a hard problem and thus require an inefficient algorithm (in the sense of complexity theory)

OW Functions: Motivations

- Using a hardness assumption
  - Which one? P ≠ NP ? BPP ≠ NP ?
  - Worst case or Average case Hardness?
- Average Case Hardness seems to be required
  - Does BPP ≠ NP implies the existence of “hard on average” languages ?
  - Hard instances need to be generated by honest parties
- Method to efficiently generate instances that are hard for adversary → One way functions!

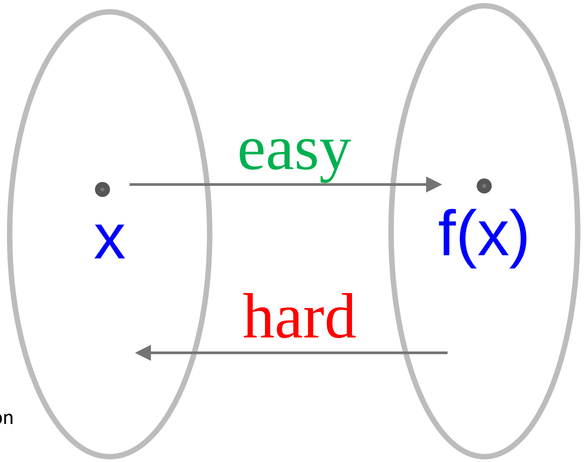
First, negligible functions

- Negligible functions
  - functions that tend to 0 "very quickly" as n increases
  - "very quickly" → smaller than any inverse polynomial
- We say that a function $e:N \rightarrow N$ is negligible if for all $c>0$ there exists $n_c>0$ such that for all $n>=n_c$ it holds that $e(n)<n^{-c}$
- Negligible probability events practically won't be observable
  - To polynomial time processes
  - If n is large enough

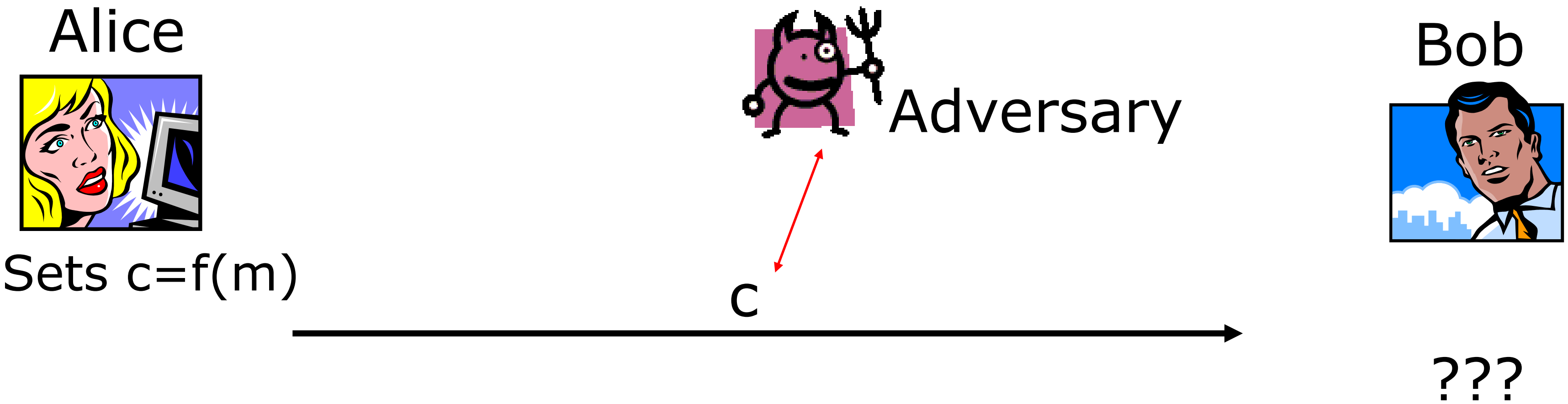
OW Functions: Definition

- OW Functions
  - Functions that are "easy to compute" but "hard to invert"
  - "easy to compute" → there exists a poly-time algorithm that can compute
  - "hard to invert" → no poly-time algorithm can invert f over uniformly distributed input
- We say that a function $f:\{0,1\}^n \rightarrow \{0,1\}^n$ is one-way if
  - There exists an efficient algorithm C such that C(x)=f(x) for all n and all x in $\{0,1\}^n$
  - For any efficient A, the following probability is negligible

$$\text{Prob}[x \leftarrow \{0,1\}^n;\ x' \leftarrow A(f(x)) : f(x')=f(x)]$$



Can we solve our problem now ?

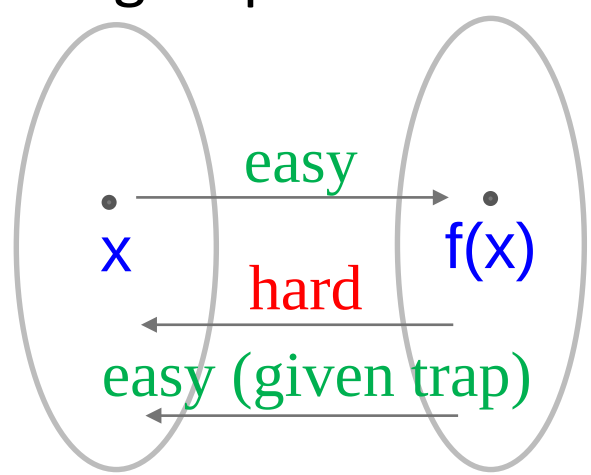


- Definition:
  - To encrypt a message m, Enc returns c=f(m), f being a one-way function
  - To decrypt ciphertext c, Dec does ???
- Properties:
  - It seems that the adversary cannot recover the plaintext from the ciphertext (due to one-wayness property)
  - But neither Bob can!

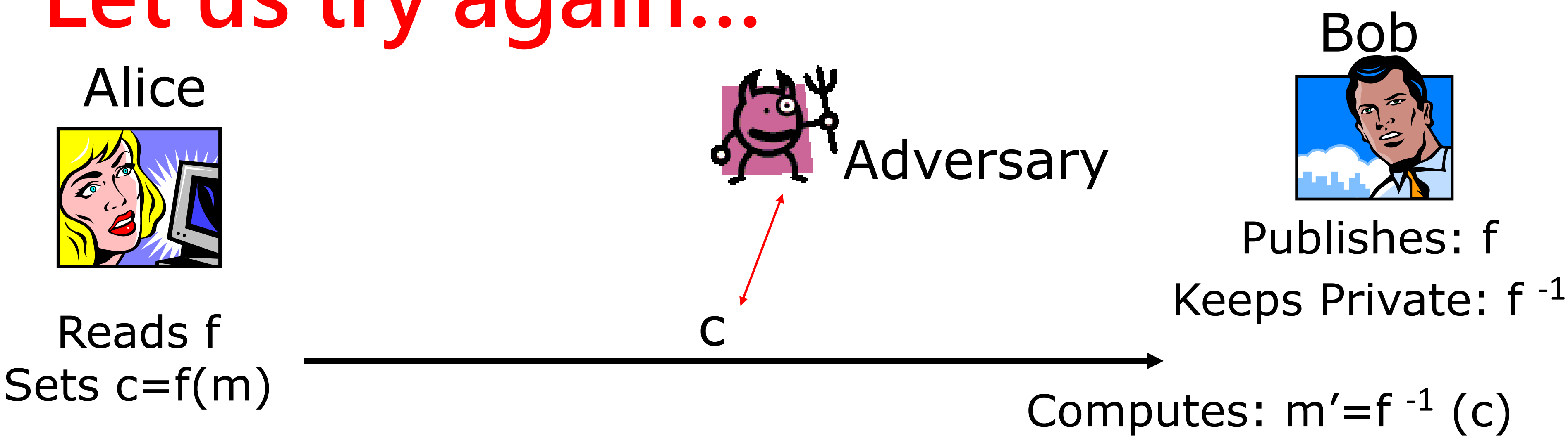
Trapdoor Functions: Definition

- Trapdoor functions are OW Functions such that
  - There exists a trapdoor string that allows its owner (and only him/her) to efficiently invert the function
- We say that a function $f_n:\{0,1\}^n \rightarrow \{0,1\}^n$ is a trapdoor function if
  - f is a one-way function, and
  - There exists an efficient algorithm E and a polynomial p such that for any n, there exists a string trap such that $|trap| < p(n)$ and for all x in $\{0,1\}^n$,

    $E(f_n(x),trap) = x'$ and $f_n(x) = f_n(x')$



Let us try again…



- Note: f is made public by Bob but f $^{-1}$ is kept private by Bob; (f, f $^{-1}$) is returned by the key generation algorithm
- Definition:
  - To encrypt a message m, Enc returns c=f(m), where f is a trapdoor function
  - To decrypt ciphertext c, Dec returns m’= f $^{-1}$(c)
- Properties:
  - Again, the adversary cannot recover the plaintext from the ciphertext
  - But Bob might still not be able to do that with some probability (as m’ may be different from m)

Trapdoor Permutation: Definition

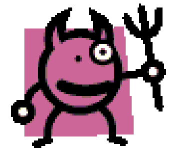
- Trapdoor permutations naturally extend trapdoor functions in that the underlying function is a permutation
- We say that a permutation $f_n:\{0,1\}^n \rightarrow \{0,1\}^n$ is a trapdoor permutation if
  - f is a one-way permutation, and
  - There exists an efficient algorithm E and a polynomial p such that for any n, there exists a string trap such that $|trap|<p(n)$ and for all x in $\{0,1\}^n$, $E(f_n(x),trap)=x$

Let us try one more time…

Alice

Reads f

Sets c=f(m)



Adversary

c

Bob

Publishes: f

Keeps Private: f^{-1}

Computes: $m'=f^{-1}(c)$

- Definition:
  - To encrypt a message m, Enc returns c=f(m), where f is a trapdoor permutation
  - To decrypt ciphertext c, Dec returns $m= f^{-1}(c)$
- Properties:
  - Bob can recover exactly m
  - Again, the adversary cannot recover the plaintext from the ciphertext
  - But the adversary might recover, say, half of the plaintext!

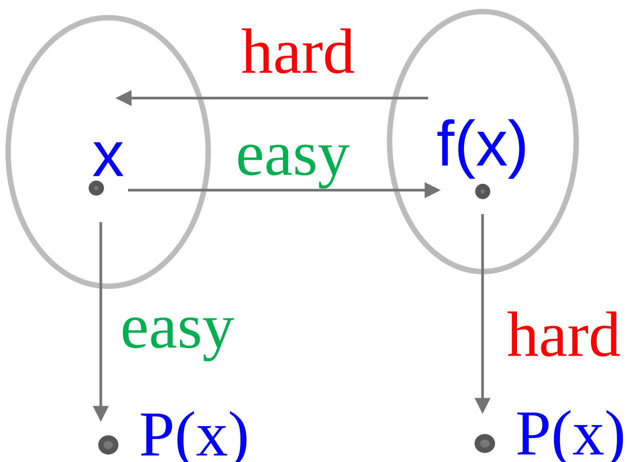
Hard Core Bit: Definition

- Hard Core Bits concentrate hardness of OW Functions in a single bit
- A function $P:\{0,1\}^n \rightarrow \{0,1\}^m$ is a boolean predicate if m=1
- Given function $f:\{0,1\}^n \rightarrow \{0,1\}^n$, predicate P is a hard-core bit for f if:
  - P(x) is "easy to compute" given x; formally: there exists an efficient algorithm E such that E(x)=P(x) for all x in $\{0,1\}^n$
  - P(x) is "hard to guess" given only f(x); formally: for any efficient A, the following quantity is negligible

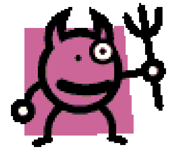
$$| \operatorname{Prob}[x \leftarrow \{0,1\}^n ; b \leftarrow A(f(x)) : b=P(x)] - 1/2 |$$

- Note: P(x) "looks" like a random bit (given only f(x))

  but is completely determined given x



Constructing a public-key cryptosystem: It works now…



Adversary

Publishes: f

Keeps Private: f^{-1}

Reads f

Sets c=(f(x),P(x) xor m)

c=(y,d)

Computes: x'=f^{-1}(y), m'=P(x') xor d

- Definition:
  - To encrypt a bit m, Enc chooses a random x and returns c=(f(x),P(x) xor m), where f is a trapdoor permutation, and xor is the boolean xor
  - To decrypt c=(y,d), Dec computes x'=f^{-1}(y) and returns m'=P(x') xor d
- Properties:
  - Bob can recover m'=m (exercise)
  - Adversary cannot recover any information about m from c if P is hard to guess (intuition: P hard to guess → P(x) looks like a random bit, independent from f(x) → P(x) xor m looks like a one-time pad encryption)
  - Note: ciphertext c has n+1 bits to encrypt a single plaintext bit m (!)

Question set 6

- Let f be a one-way function. Is it a one-way permutation?
- Let f be a one-way permutation. Is it a one-way function?
- Are the functions below one-way? Are they a permutation?:
  - f:Z→Z, where for each integer x, f(x) = 2x+5
  - f:$\{0,1\}^n$→ $\{\{0,1\}^n\}^{2^n}$, where for each n-bit string x, f(x) = list of all subsets of positions with a 1 in x
- Assume you have a hard-core predicate for a permutation. Is this permutation one-way?
- Fill a 3x3 table whose entries indicate which of the statements

  "if there exists A then there exists B"

  is true or unknown or false, where A and B are taken from set

  {one-way functions, trapdoor functions, hard-core predicate for some function}
- Prove that the public-key cryptosystem presented in this lecture satisfies decryption correctness

Summary of Lecture 2

- More Background on Algorithms
- Some Background on Complexity Theory
- Modern Cryptography principles
- One-way functions
- Trapdoor functions
- Applications to constructing a public-key cryptosystem
- Implementation aspects

Implementation Aspects

- From classical to modern cryptography
  - Perfectly-secret encryption required Alice and Bob to share a random string (the key) that is as long as the message
  - Public-key encryption does not require Alice and Bob to share a key and requires Bob to publish a short key
- Modern cryptography: saving on randomness and key management via computational hardness
  - One-way functions have to be (asymptotically) easy to compute and (asymptotically) hard to invert
  - Inherent tension:
    - To achieve hard to invert property, input length n has to be large
    - To achieve easy to compute property, input length n has to be small
  - In practice, to achieve hardness, one sacrifices efficiency (i.e., choose smallest input length for which f seems hard to invert)
    - E.g., length = 1024

Class CS 6903, End of Lecture n. 2

Topic ↓ Reference →	[KL]	[MOV]	[FSK]
More background on algorithms	A.2	1.3	
	http://en.wikipedia.org/wiki/Big_O_notation		
Some background on complexity theory		2.3.3	
	http://en.wikipedia.org/wiki/Computational_complexity_theory http://en.wikipedia.org/wiki/P_versus_NP_problem		
Modern cryptography principles	1.4, 3.1	1.13	2.1
One-way functions	7.1.1 (1st edition: 6.1.1)	1.3	
Trapdoor functions	13.1.1 (1st edition: 10.7.1)	1.3	
Hard-core bits	7.1.3 (1st edition: 6.1.3)		
Construction of a public-key cryptosystem	13.1.2 (1st edition: 10.7.2)	1.8.1	2.1, 2.3