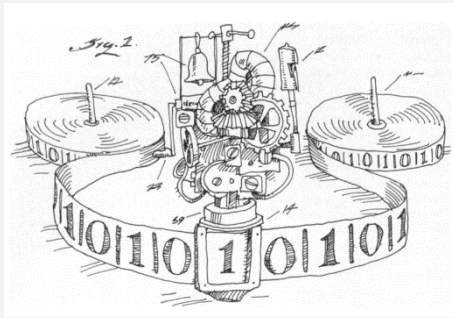
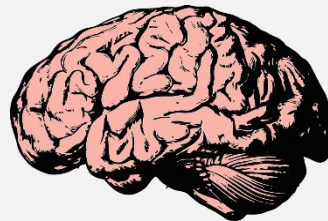
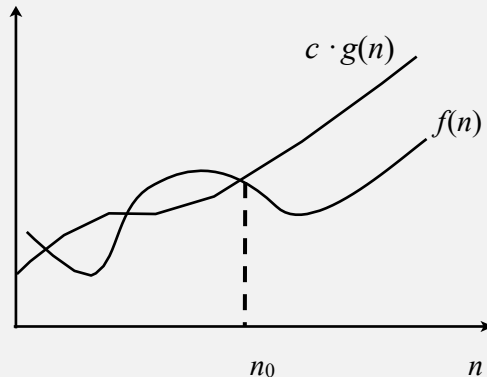


COMP 3602

Intro to Theory of Computation

Lecture 2

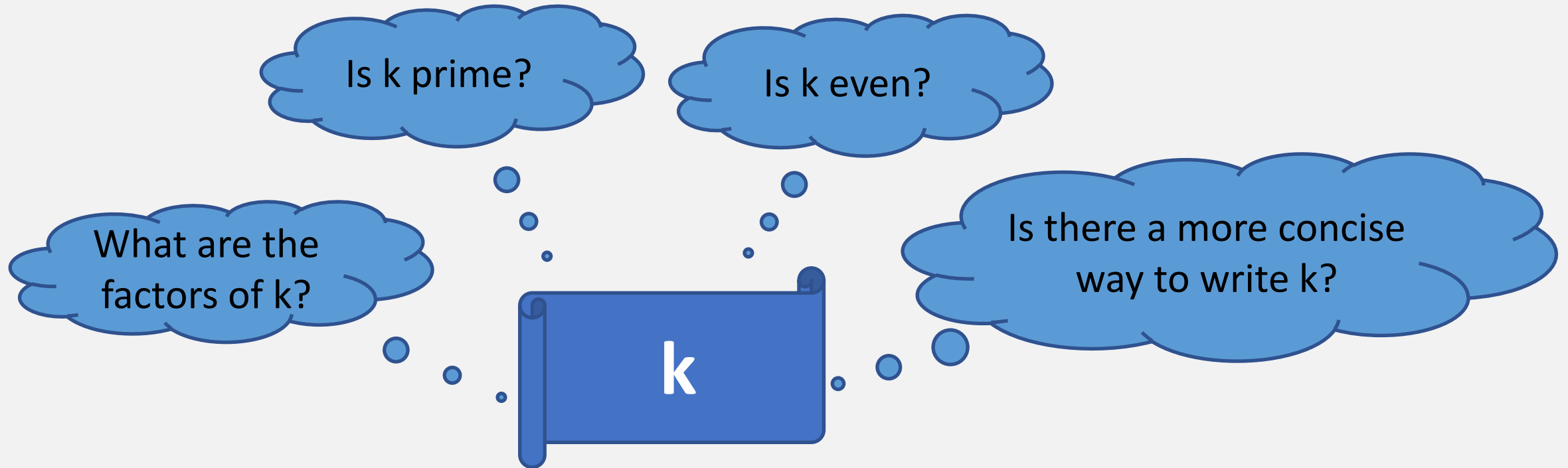


Logistics

- Office hours times: before lectures?
 - Reading group times? Your availability: MWF 10-11, MF 1-2
- I plan to be away for two workshops during the semester:
 - Jan 11-16 (next week), Apr 3-10 (last week of classes)
 - So from Jan 19th to Jan 30 (or Feb 6?) lectures will go till 5:15pm
 - We might be switching to EN-2007

Questions/comments?

How hard is it to solve each of these problems?

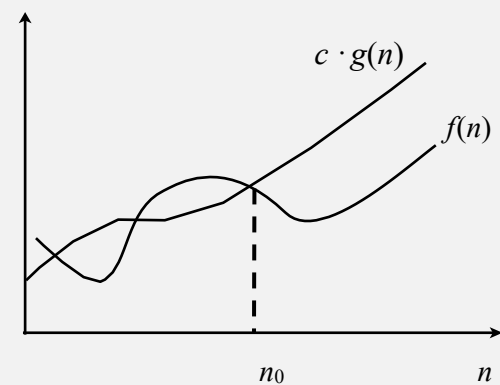


- Here, $k \in \mathbb{Z}$

Upper/lower/tight bounds notation

Upper bounds: big O.

$f(n)$ is in $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.



Lower bounds: big Omega, Ω

$f(n)$ is in $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \geq c \cdot g(n) \geq 0$ for all $n \geq n_0$.

choose $c = 50, n_0 = 1$

Tight bounds: big Theta, Θ

$f(n)$ is in $\Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.

Upper bounds: little o.

$f(n)$ is in $o(g(n))$ if $f(n)$ is in $O(g(n))$, but not in $\Theta(g(n))$

Lower bounds: little omega, ω

$f(n)$ is in $\omega(g(n))$ if $f(n)$ is in $\Omega(g(n))$, but not in $\Theta(g(n))$

Ex. $f(n) = 32n^2 + 17n + 1$.

$f(n) \in O(n^2)$ and $f(n) \in O(n^3)$, but $f(n) \notin O(n)$

$f(n) \in \Omega(n^2)$ and $f(n) \in \Omega(n)$, but $f(n) \notin \Omega(n^3)$

$f(n) \in \Theta(n^2)$, but $f(n) \notin \Theta(n^3)$ and $f(n) \notin \Theta(n)$

so $f(n) \in \omega(n)$ and $f(n) \in o(n^3)$

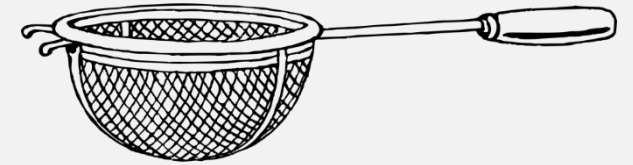
but not $f(n) \in \omega(n^2)$ or $f(n) \in o(n^2)$

Upper bounds (roughly O): $\tilde{O}$.

$f(n)$ is in $\tilde{O}(g(n))$ if $f(n)$ is in $O(g(n) \cdot h(n))$, where $h(n) \in O((\log g(n))^k)$

What are the factors of k ?

Problem 1: Integer factorization



- Already known in ancient Greece:

- Eratosthenes's sieve algorithm.
 - ~3rd century BC

- Put numbers from 2 to k in list L
- WHILE $i = \min(L) \leq \sqrt{k}$
 - Remove all multiples of i from L
 - If hit k , then i divides k

- Time: $O(k^2)$ (actually, $O(k \log \log k)$)

- Exponential-time when k is in binary!

- Lower bound? **Unknown!**

- Solving factoring efficiently would break RSA cryptosystem.

- There is a much faster $\tilde{O}((\log k)^2)$ quantum algorithm.



Is k prime?

Problem 2: Primality



- Can use Erathosthenes sieve again...
 - Is there a faster way?
- In 20th century: randomized efficient algorithm
- In 21st century, 2002: an IIT Kanpur professor Manindra Agrawal and his two undergrads, Neeraj Kayal, and Nitin Saxena, created a polynomial-time (efficient) algorithm!
 - Time complexity $\tilde{O}((\log k)^{12})$
 - It took more than 2000 years to get an efficient primality test.



Is k even?

Problem 3: Evenness



- Don't need Erathosthenes sieve: look at the last bit of the binary encoding of k , check if it is 0.
 - On a machine with random-access memory, for n in binary, constant time
 - For k in unary $O(k)$, or when no random access, $O(|k|)$ time
- Don't need to have a whole computer or a Turing machine: a finite automaton is enough
 - Both in unary and binary, though different automata.

Is there a more concise way to write k ?

Problem 4: Complexity of k



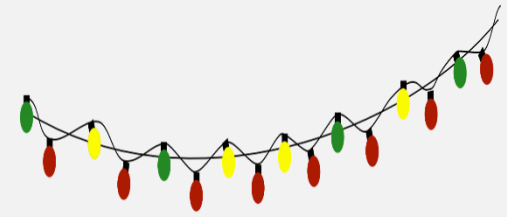
- Equivalently, how much information does k contain? Is k compressible or random-looking?
 - One way to define it is to give a shortest program printing k
 - A program to print $k=1000000000$ would be shorter than $k=1011101001$
 - We will formally define Kolmogorov complexity later.
- This is a much harder problem than the previous ones!
 - We can prove that there is no program which, given k , always outputs the length of the shortest description of k .
 - For that, we need to define formally what we mean by a program, and to what kinds of program this applies.

Alphabet

- A finite set of symbols is called an **alphabet**.
 - Often use the letter Σ (capital Sigma) to denote an alphabet
 - English alphabet contains 26 letters (ignoring case): {a, ..., z}
 - {0,1} is the *binary alphabet*. This is the most popular alphabet in computer science.
 - An alphabet containing only one letter, for example {a} or {1}, is called a *unary* alphabet.

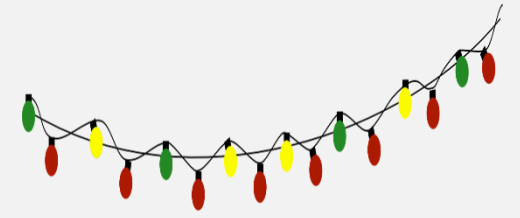


Strings



- A **string** (over an alphabet Σ) is a sequence (list) of symbols from Σ
Usually with repetition; order matters.
 - We will talk about 1st, 2nd, 3rd and so on symbol in a given string: indexing from 1.
- “mun” is a string over English alphabet (lowercase).
 - 2nd symbol in “mun” is “u”.
 - English words are strings over English alphabet.
- A natural number is a string over the alphabet of digits $\{0,1,2,3,4,5,6,7,8,9\}$
 - “1002” is a string over the alphabet of digits.
 - Or maybe of some other alphabet Σ such that $\{0,1,2\} \subseteq \Sigma$

Finite strings, binary strings



- A *finite string* of length n is a string containing n symbols
 - The length of a string s is denoted $|s|$
 - The same notation as cardinality of a set (number of elements in a set): $|A|$
 - $|\text{mun}|=3$. $|1002|=4$.
 - There is an empty (null) string with the length 0
 - Some books, including our textbook, denote the empty string by a Greek letter “lambda” λ , others by the Greek letter “epsilon” ϵ .
- A *binary string* is a finite string over the alphabet $\{0,1\}$
 - 0011011, 1111, λ , 0 are binary strings
 - with lengths 7,4,0,1 respectively.



Languages and problems

- A **language** (over an alphabet Σ) is a set of finite strings over Σ
 - The set of all finite strings over Σ is denoted Σ^* . Empty language is $\emptyset$.
 - For example, the set of all binary strings is $\{0,1\}^*$
- A **decision problem** is determining a membership in some set
 - For example, a set of binary strings encoding prime numbers
 - Equivalently, a decision problem is a function $f: \Sigma^* \rightarrow \{no, yes\}$
- In **search** and **optimization** problems the output is usually a string, rather than just yes/no.
 - So a search problem is a function $f: \Sigma^* \rightarrow \Sigma^*$



We want to pinpoint hardness of computational problems

And in some rare cases, we can determine it exactly.

C	O	M	P	U	T	E	R
---	---	---	---	---	---	---	---



C	E	M	O	P	R	T	U
---	---	---	---	---	---	---	---



Proving tight bounds: sorting

• Comparison-based sorting
→ Input: an array A containing n elements $a_1, \dots, a_n$
→ Here, the only operation allowed on $a_1, \dots, a_n$ is a comparison: $a_i \leq a_j$, which takes constant time.
→ Output: an array A' with $a_1, \dots, a_n$ in sorted order: $A'[1] \leq \dots \leq A'[n]$

• Theorem: Complexity of comparison-based sorting is $\Theta(n \log n)$
→ On a computer with random access memory and constant-time elementary operations
→ Comparisons, arithmetic, read/write, access $A[i]$ are constant time.

- Comparison-based sorting:
 - Input: an array A containing n elements $a_1 \dots a_n$
 - Here, the only operation allowed on $a_1 \dots a_n$ is a comparison: $a_i \leq? a_j$, which takes constant time.
 - Output: an array A' with $a_1 \dots a_n$ in sorted order, $A'[1] \leq \dots \leq A'[n]$

C	O	M	P	U	T	E	R
---	---	---	---	---	---	---	---



C	E	M	O	P	R	T	U
---	---	---	---	---	---	---	---

- **Theorem:** Complexity of comparison-based sorting is $\Theta(n \log n)$
 - On a computer with random access memory and constant-time elementary operations
 - Comparisons, arithmetic, read/write, access $A[i]$ are constant time.



Proving tight bounds: sorting

Theorem: Complexity of comparison-based sorting is $\Theta(n \log n)$

- *Upper bound:* Comparison-based sorting is in $O(n \log n)$ since MergeSort algorithm solves it in time $O(n \log n)$.
- *Lower bound:* Any comparison-based sorting algorithm takes $\Omega(n \log n)$ time (in the worst case).
 - We will show that every algorithm needs to make at least $\Omega(n \log n)$ comparisons to solve the problem correctly.
 - Therefore, its total running time is at least $\Omega(n \log n)$.
- Finally, since comparison-based sorting is both in $O(n \log n)$ and in $\Omega(n \log n)$ time, its time complexity is $\Theta(n \log n)$.



Proving tight bounds: sorting

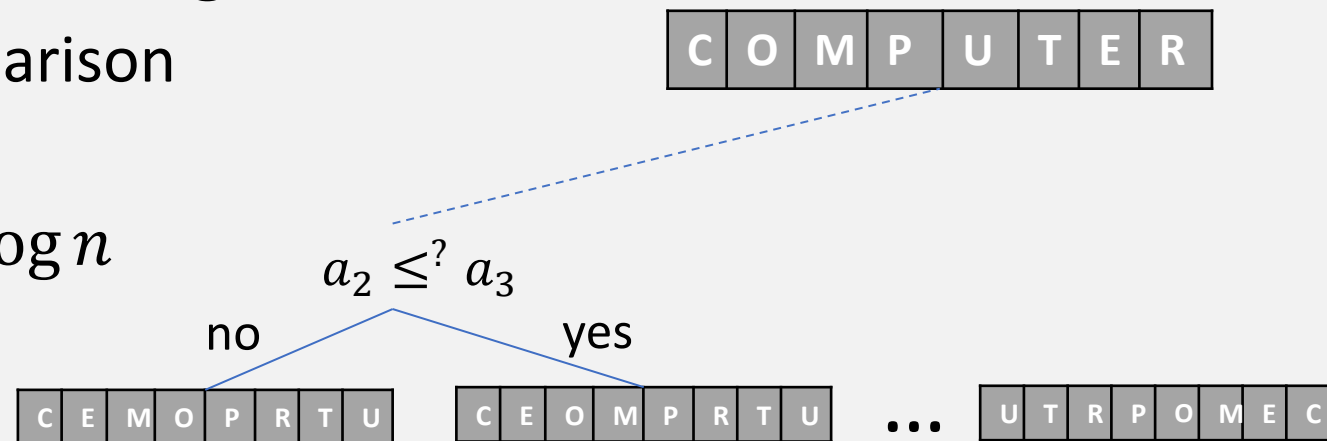
- *Lower bound*: Any comparison-based sorting algorithm takes $\Omega(n \log n)$ time (in the worst case).
- Proof idea: a *counting argument*.
 - There are lots ($n!$) possible solutions.
 - When all elements are distinct, exactly one is correct.
 - No matter what an algorithm is, after each comparison it eliminates, in the worst case, at most half of the remaining possible solutions.
 - Because at least one of “yes” or “no” answers to $a_i \leq? a_j$ leads to least a half of all remaining solutions.
 - So in the worst case it needs $\log_2 n! \in \Omega(n \log n)$ comparisons.





Proving tight bounds: sorting

- We will show that every algorithm needs to make at least $\Omega(n \log n)$ comparisons to solve the problem correctly.
 - There are $n!$ possible orderings of n elements.
 - So there are $n!$ possible solutions: one for each ordering of $a_1 \dots a_n$.
 - Take an arbitrary algorithm solving the sorting problem.
 - Consider a tree of decisions this algorithm makes
 - At each node, there is a comparison
 - There are $n!$ leaves
 - So height (# of steps) is $\geq n \log n$





Proving tight bounds: sorting

- Consider a tree of decisions this algorithm makes
 - There are $n!$ leaves
 - So height (# of steps) is $\geq n \log n$
 - Lemma: the height of a binary tree is $\geq \log$ of its number of leaves.

