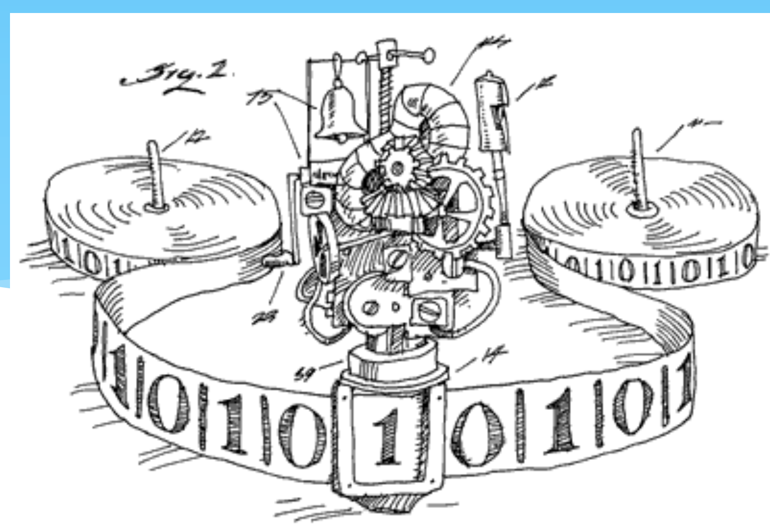
EECS 376: Foundations of Computer Science

Seth Pettie

Lecture 14

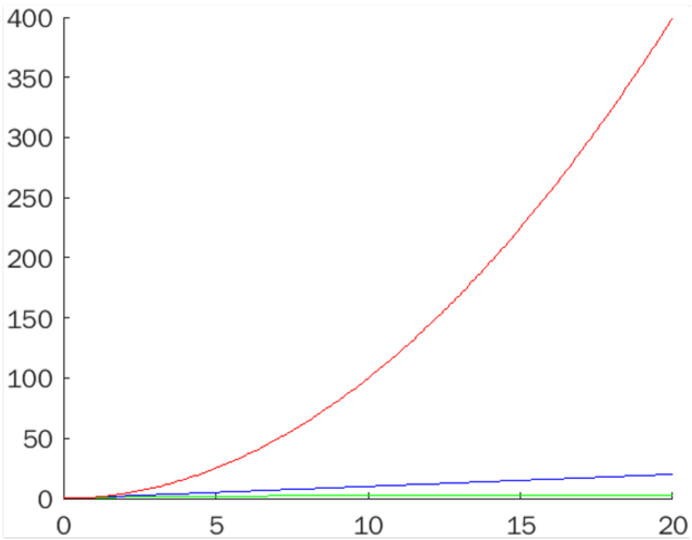


Complexity: Unit's Agenda

1) What resources are required to solve a problem on a computer?
2) Which problems can be solved **efficiently** on a computer?
3) *What does "**efficiently**" mean?*

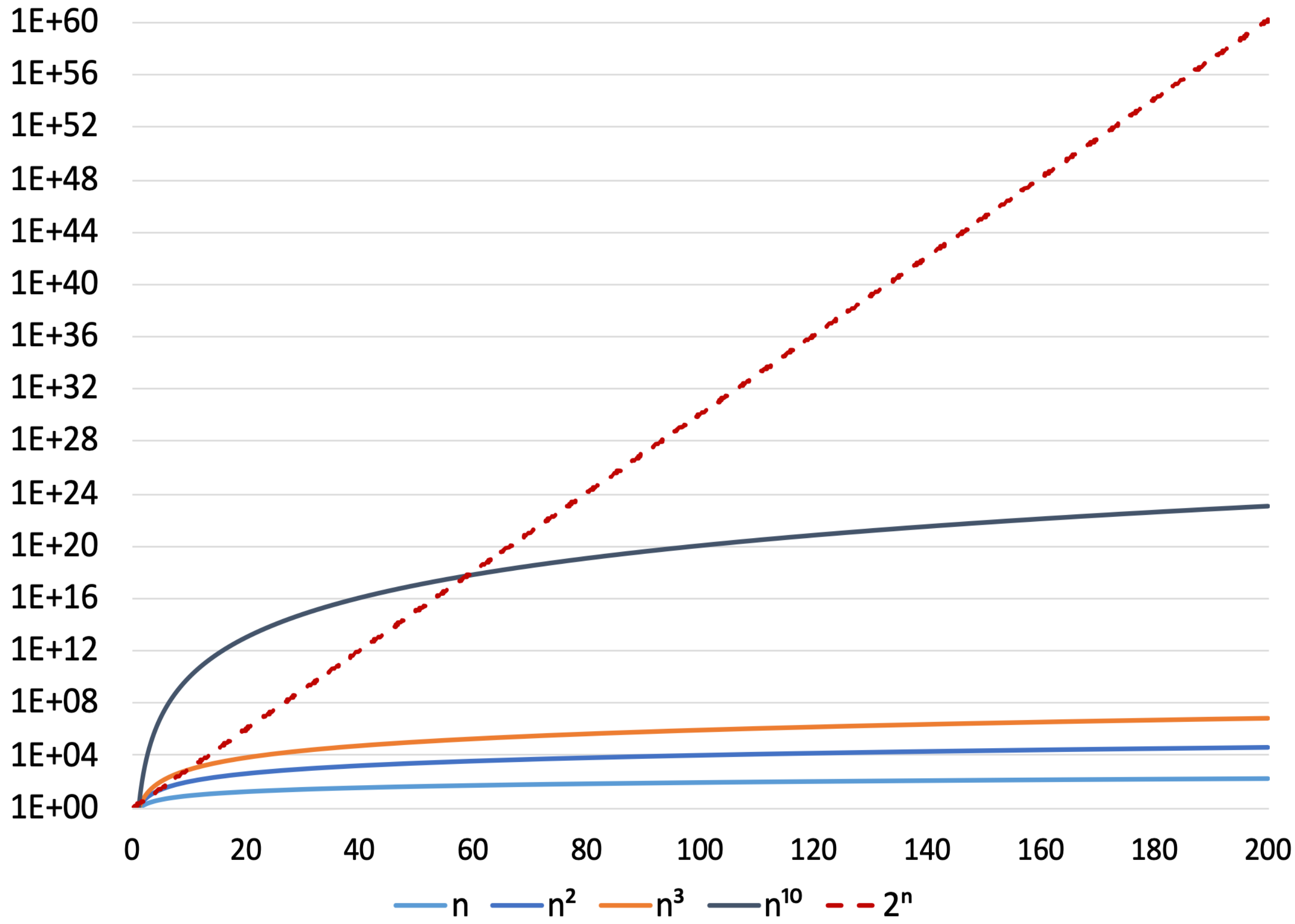
Review: Running Time

- We measure "efficiency" of an algorithm by how its (*worst-case*) ***runtime*** scales with input "size"
- We express this tradeoff in Big-O: e.g., $O(\log n), O(n), O(n^2)$, etc, where n is the size of the input.
- Common interpretations of "size":
  - size of array = O(# elements)
  - size of graph = O(# vert./edges)
  - size of integer = O(# digits)
  - **Rule of thumb:** size = O(# bits of memory to store on a computer)



Efficient ≡ "runtime is polynomial in size of input";
Very robust definition (will see)

Polynomial vs Exponential Growth



Polynomial time algorithms

We have seen various polynomial time (poly-time) algorithms.

GCD(x,y)	Euclid's alg. (time O(log $(x+y)$))
LIS(x)	Dynamic Prog. (time O(n^2))
LCS(x,y)	Dynamic Prog. (time O(n^2))
All pairs Shortest Path	Dynamic Prog. (time O(n^3)
MST(G)	Greedy O(n log n)

There are several others: Matching, Max-flow min-cut, ...
Deciding if x is prime [Agarwal, Kayal, Saxena 2002]: Time $(\log x)^6$ …

Based on lots of clever and ingenious ideas.
Several algorithms still to be discovered

A Polynomial time algorithm for every problem?

Longest Path: Given G and vertices s,t. find longest s-t path.

Hamiltonian Cycle: Given G, find a cycle that goes through each vertex exactly once.

Subset Sum: Given $a_1, \dots, a_n$ and target t. A subset that sums to t
(what about a DP solution?)

Independent Set: Given G and integer k, is there a subset S of size k so that no two vertices in S are adjacent
…

We believe that none of these problems has a poly-time algorithm
How can we say such things?

Theory of P vs. NP (a crown jewel of TCS)
Is P=NP ? (a profound question with a million dollar reward)

The Class P

* **Definition:** Class **P** (***efficiently decidable*** languages)
  * **P** = all languages that can be decided by a TM in polynomial time in the input size.
* In words, problems with a polynomial time decision algorithm
* **Formally:** $\mathbf{P} = \bigcup_{k \geq 1} DTIME(n^k)$
* **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.
  * *Proof idea*: $(n^k)^{k'} = n^{k \cdot k'}$ is also polynomial.
* **Definition:** Efficient = Polynomial Time in the *input size*

Search vs. Decision problems

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

In words, problems with a poly-time decision algorithm

Note: We are still talking about decision problems (with answers yes/no)
What about gcd(x,y), shortest path (s,t), LIS(x), …
(here the answer is not a yes/no.)

This is not a problem at all (will soon see).

Example

* **Problem:** Given integers x, y, z, is $\gcd(x, y) \geq z$?
  * $L_{GCD} = \{(x, y, z) : x, y, z \in \mathbb{N} \text{ and } z \leq \gcd(x, y)\}$
* **Claim:** L_{GCD} is efficiently decidable, i.e., $L_{GCD} \in \mathbf{P}$.

Run Euclid's algorithm and answer accordingly

* **Correctness analysis (sketch):** The Euclidean algorithm correctly computes the greatest common divisor of x, y.
* **Runtime analysis:** Given an input of length n
  * The size of the input is $n = |x| + |y| + |z| = O(\log(x) + \log(y) + \log(z))$. (# of digits of x is $O(\log(x))$)
  * (recall) Computing **Euclid**(x, y) takes $O(\log(x + y)) = O(n)$ time.
  * Comparing z and **Euclid**(x, y) takes $O(n)$ time (digit-by-digit comparison).
  * Therefore, total time taken is $O(n)$.

Another Example

* **Problem:** Given a weighted graph G and vertices s and t, is there a path of length at most 376 between s and t in G?
  * $L_{376PATH} = \{(G, s, t) : G$ is a weighted graph w/ a s-t path of length $\leq 376\}$.
* **Claim:** $L_{376PATH}$ is efficiently decidable.

bool **376path**(graph G, vertex s, vertex t)
1. $D \leftarrow$ **FloydWarshall**(G)
2. **return** $(D[s][t] \leq 376)$

* **Correctness analysis (sketch): FloydWarshall** algorithm returns matrix of all-pairs shortest path lengths.
* **Runtime analysis:** Given an input (G, s, t) of length n
  * (recall) **FloydWarshall**$(G = (V, E))$ takes $O(|V|^3) = O(|G|^3)$ time (V = vertices of G).
  * Since $n = |(G, s, t)| = O(|G|)$, time taken is $O(n^3)$.

Introducing: The Class NP

Common mistake: NP does *not* mean "Non-polynomial"

(NP Stands for **N**on-deterministic **P**olynomial time)

(We will not refer to "non-determinism" in this course — there is an equivalent definition of NP that does not mention non-determinism.)

A better name would have been VP (for ***verifiable problems***)

- Clyde Kruskal

Verifiable Computations

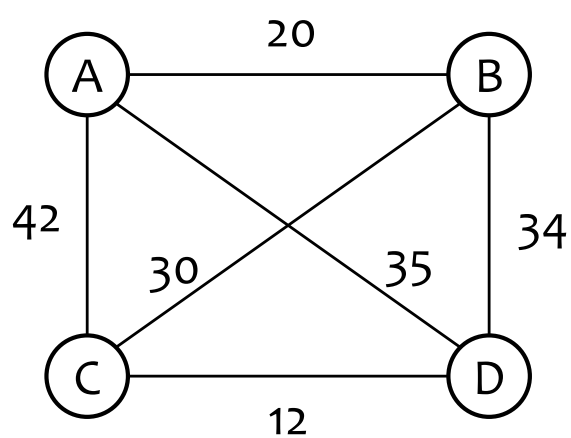
- **Example 1:** Given a sudoku puzzle, is there a solution?
- **Answer:** Yes.
- **Reply:** We are not convinced (i.e. you could be lying to us).
- **Reply:** Now we are convinced.

			2	6		7		1
6	8			7			9	
1	9				4	5		
8	2		1				4	
		4	6		2	9		
	5				3		2	8
		9	3				7	4
	4			5			3	6
7		3		1	8			

4	3	5	2	6	9	7	8	1
6	8	2	5	7	1	4	9	3
1	9	7	8	3	4	5	6	2
8	2	6	1	9	5	3	4	7
3	7	4	6	8	2	9	1	5
9	5	1	7	4	3	6	2	8
5	1	9	3	2	6	8	7	4
2	4	8	9	5	7	1	3	6
7	6	3	4	1	8	2	5	9

Verifiable Computations

* **Example 2:** Decision version of ***Traveling Salesperson Problem (TSP)*** Given 4 cities and pair-wise distances between them, is there a tour of length at most 100 that visits all the cities?
* **Remark:** Here we only care about feasibility and not the actual tour.
* **Answer:** $A \to B \to C \to D \to A$. Cost: 20+30+12+35 = 97.
* **Reply:** We are convinced.



Verifiable Computations

* **Example 3:** Subset Sum
* Given integers $a_1, \ldots, a_n$ and target t, is there a subset of numbers that sums to t?

* **Answer:** The subset of numbers.
* **Reply:** (adding them up) We are convinced.

Efficiently Verifiable Problems

* **Intuition:** A class of problems is ***efficiently verifiable*** when:
  * If the problem has a solution, then there is an "efficient" way to verify it given some *additional information* ("***certificate***").
  * If there is *no solution*, then no additional information (even maliciously produced) could convince us to say 'yes'.
* Let us look at the previous examples
* TSP: If there is a solution, we can be convinced
  If there is no solution, can a "malicious adversary" convince us?
* Subset-Sum: Convinced if solution.
  If there is no solution, can a "malicious adversary" convince us?

The Class **NP**

* **Definition:** A decision problem L is ***efficiently verifiable*** if there exists an algorithm $V(x, c)$ called a ***verifier*** such that:

1. $V(x, c)$ is efficient with respect to x (polynomial time in $|x|$).
2. If $x \in L$, then there is <u>*some*</u> ***certificate*** c such that $V(x, c)$ accepts.
3. If $x \notin L$, then $V(x, c)$ rejects <u>*all*</u> ***certificates*** c.

* **Definition:** The class **NP** = the class of efficiently <u>*verifiable*</u> languages

Example

* **TSP:** Given n cities and pair-wise distances, is there a route that starts/ends at the same city, visits every city *exactly once*, and has length at most k?
  * $\text{TSP} = \{(G, k) : G$ is a weighted, complete graph w/ **tour** of length $\leq k\}$

* **Claim:** TSP is efficiently verifiable, i.e., $\text{TSP} \in \mathbf{NP}$.

Consider the certificate in the form of a path c (which is supposed to correspond to the tour, if one exists)

bool **verifyTSP**(graph $G = (V, E)$, int k,
path $c = (v_1, \ldots, v_m)$):
1. if (c is not a permutation of V): **return** *false*
2. **return** length$(v_1, \ldots, v_m, v_1) \leq k$

Correctness Analysis of **verifyTSP**

bool **verifyTSP**(graph $G = (V, E)$, int k,
path $c = (v_1, \dots, v_m)$):
1. if (c is not a permutation of V): **return** *false*
2. **return** length$(v_1, \dots, v_m, v_1) \leq k$

- **Case 1:** If $(G, k) \in \text{TSP}$, then <u>*some*</u> certificate c makes **verifyTSP**$((G, k), c)$ return *true.*
  - Let $c = (v_1, \dots, v_m)$ be the permutation of the vertices of G where the vertices are ordered according to the tour.
    - Since c is a permutation of V, line 1 does not return *false*.
  - Then, length$(v_1, \dots, v_m, v_1) \leq k$ by assumption and **verifyTSP**(x, c) returns *true* on line 2, as desired.

Correctness Analysis of **verifyTSP**

```
bool verifyTSP(graph G = (V, E), int k,
               path c = (v_1, …, v_m)):
1. if (c is not a permutation of V): return false
2. return length(v_1, …, v_m, v_1) ≤ k
```

* **Case 2:** If $(G, k) \notin \text{TSP}$, <u>*every*</u> certificate c makes **verifyTSP**(x, c) return *false*.

  * If c does not represent a permutation of the vertices of G, then *false* is returned on line 1, as desired.

  * Now suppose $c = (v_1, \ldots, v_m)$ represents a permutation of the vertices of G.
  * By assumption, G has no tour of length $\leq k$.
    * Therefore, length$(v_1, \ldots, v_m, v_1) > k$.
    * Thus, *false* is returned on line 2, as desired.

Runtime Analysis of **verifyTSP**

bool **verifyTSP**(graph $G = (V, E)$, int k,
path $c = (v_1, \ldots, v_m)$):
1. if (c is not a permutation of V): **return** *false*
2. **return** length$(v_1, \ldots, v_m, v_1) \leq k$

* Suppose the input (G, k) has length n.
* There are at most n vertices in G.
* Line 1 takes $O(n)$ time – use a boolean array to check if each vertex appears exactly once in c.
* Line 2 takes $O(n)$ time – sum up the weights on the edges seen when following the tour.

* Therefore, runtime is polynomial in n, as desired.

Practice with certificates

L_{Comp} = {n: n is composite (not a prime)}

L_{HAM} = {G: G has a Hamiltonian cycle}

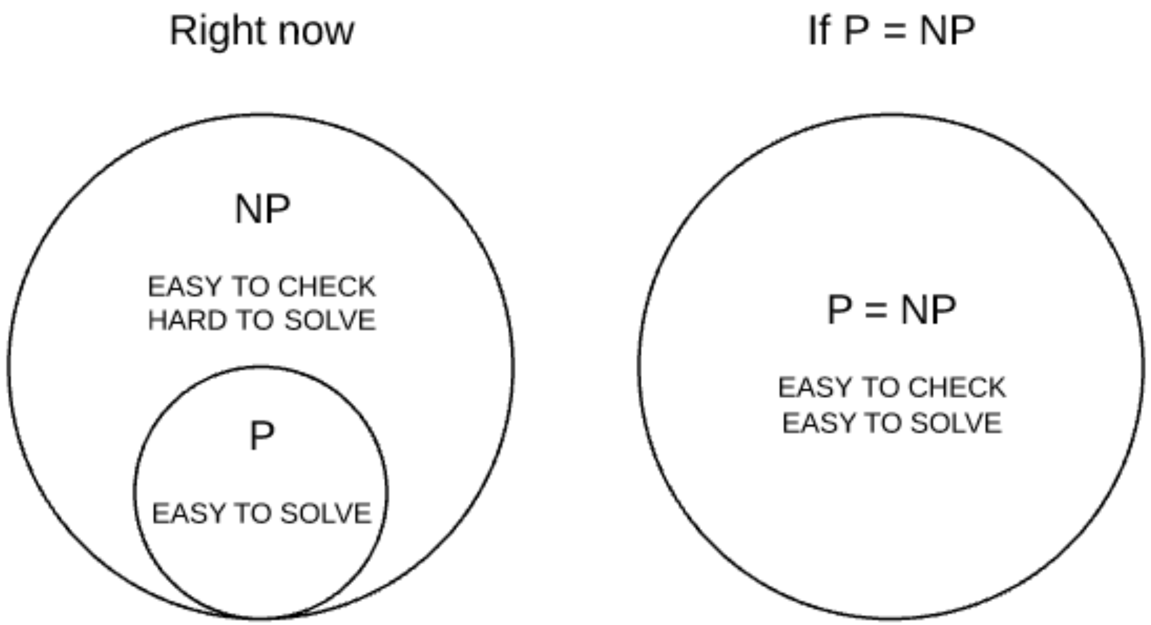
L_{Primes} = {n: n is a prime } (complement of L_{Comp})
Not obvious anymore, but there is a complicated one

$L_{non-HAM}$ = {G: G has no Hamiltonian-cycle}
We do not expect the problem to have an efficiently verifiable certificate (would have very unexpected consequences)

P vs NP

* **Formally:** Let L be a language. (decision problem)
* $L \in$ **P** if there exists a polynomial time in $|x|$ algorithm $M(x)$ such that:
  * $x \in L \Longrightarrow M(x)$ accepts
  * $x \notin L \Longrightarrow M(x)$ rejects
* $L \in$ **NP** if there exists a polynomial time in $|x|$ algorithm $V(x, c)$ such that:
  * $x \in L \Longrightarrow V(x, c)$ accepts for at least one c
  * $x \notin L \Longrightarrow V(x, c)$ rejects for every c
* **Note: P ⊆ NP** (V can ignore c and just run M)
* **$1,000,000 question:** Is **P = NP**?

P vs NP



"If P = NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in 'creative leaps', no fundamental gap between solving a problem and recognizing the solution once it's found."

- Scott Aaronson

EECS ECE CSE

The Major Open Problem of Computer Science