Lecture 10

Time Complexity (P,NP)

Recap of Previous Lectures

1. Turing machines
   - Computational model for "real computers"
   - Results about "computability"
     - Decidable: Given time and memory . . . can be done
2. Reducibility
   - Proof technique for (un)decidable problems
   - Computational function, Turing machine exists

. . . So far we have ignored the time aspect!

Today: *Time complexity*

Running time

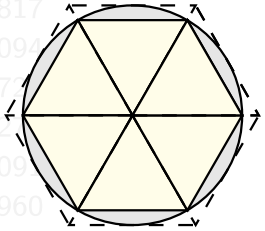
- Recap: TM is a model for computation
  - Input string w
  - Result:
    1. *Accept* or *Reject* state (decision problems)
    2. *String* on the tape
- Default TM:
  - Deterministic
  - Single-tape

$$\boxed{r(w) \triangleq \text{“the number of steps before halt on input } w\text{”}} \quad (3)$$

Example: Computing π

The beginning:
In 250BC, Archimedes developed an *algorithm* and computed

$$\frac{223}{71} < \pi < \frac{22}{7}$$

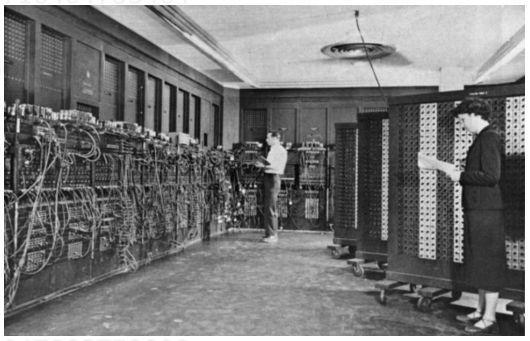


i.e., $3.1408 < \pi < 3.1429$, almost **3 digits!**

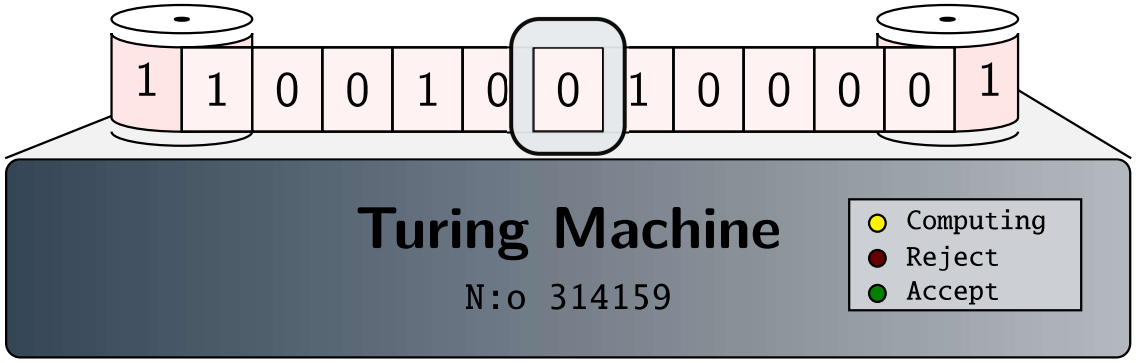
480 AD, Chinese mathematician Zu Chongzhi calculated 7 digits,
$3.1415926 < \pi < 3.1415927$

Computer era:
In 1949, John von Neumann et al. computed **2037 digits** in 70 hrs with ENIAC (the supercomputer of the era) using Machin's formula

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239} \quad \text{(Machin, 1706)}$$



Digits of π with a Turing machine



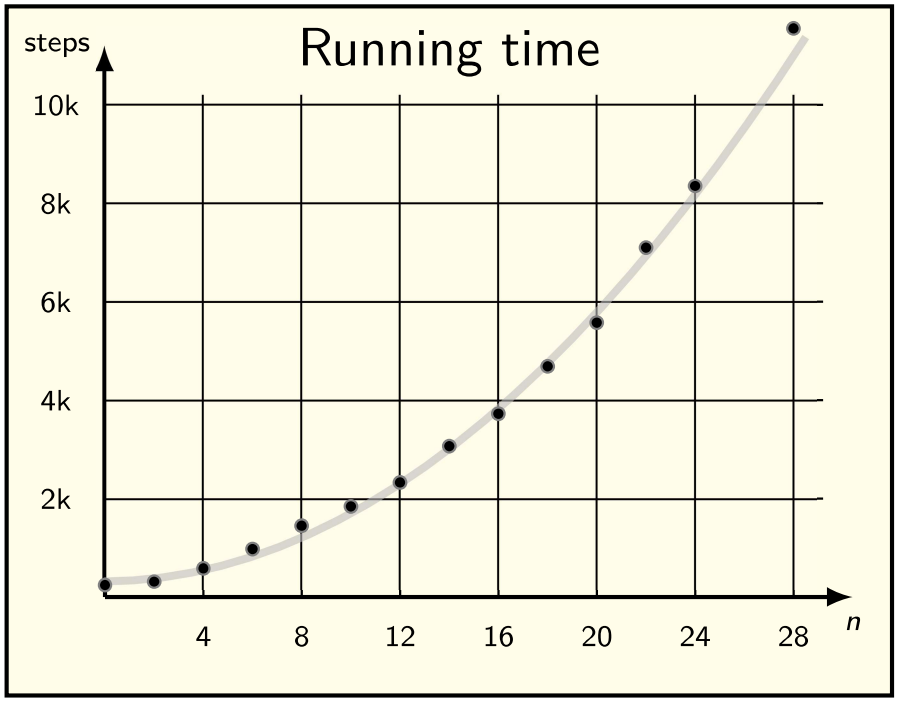
- The infinite tape $\Rightarrow$ any number of digits!
- Two-tape TM M_π implements Machin's formula

Results with a two-tape machine:

n	result		time
8	11.00100101	3.14	1446
12	11.001001000101	3.142	2324
16	11.0010010001000001	3.1416	3728
20	11.00100100001111110110	3.14159	5567
24	11.00100100001111101101011	3.1415927	8344

Running time with TM M_π computing digits of π

Experiments gives some insight to the running time:



The gray curve is a 2nd degree polynomial function of n.

Measuring Time Complexity

Good news: Running time $r(w)$ is well-defined:

- number of steps before TM halts

Bad news: $r(w)$ can be extremely complicated function...

- ...and w can be of arbitrary length

How to summarize if $r(w)$ was available?

Measuring Time Complexity

Idea 1: **Determine the maximum running time**

$$\max_w r(w)$$

- Input string can be arbitrarily long
- ... and the above often gives the infinity (cf. TM M_π)

Idea 2: **Determine the mean running time**

- Countably infinite number of input strings
- Need a probability distribution $p(w)$ for input strings
- Input W is a random variable, and

$$m = \mathbb{E}[r(W)] = \sum_w p(w) \cdot r(w)$$

- But we do not know the distribution of W!

Idea 3: **Condition on the length of the input string**

- Bigger problem instances *tend to be* computationally more demanding

Worst-case analysis

So we proceed as follows:

Definition 48 (Def. 7.1)

Let M be a deterministic Turing machine that halts on all inputs. The running time *or* time complexity *of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the* **maximum number of steps** *that M uses on any input of length n.*

In terms of $r(w)$

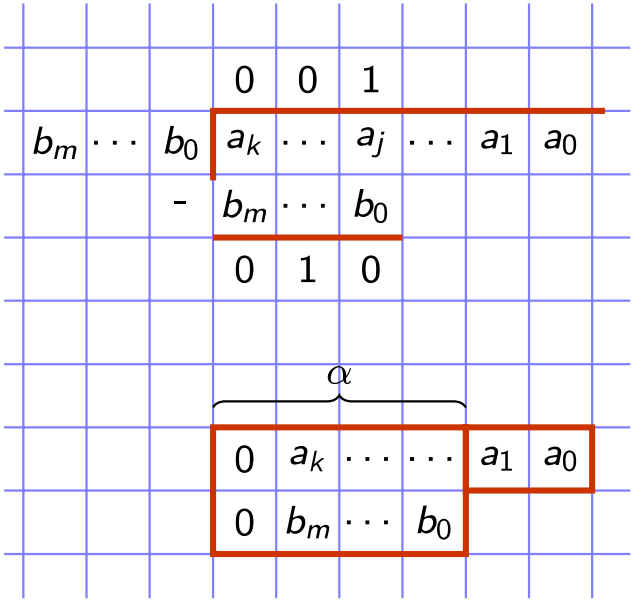
$$\boxed{f(n) = \max_{|w|=n} r(w)}$$

If $f(n)$ is the running time of M, we say that

1. *"M runs in time $f(n)$"*, and that
2. *"M is an $f(n)$ time Turing machine"*.

Note: No assumptions regarding the distribution

Example: Long division



Repeat while $|a| \geq |b|$:

1. Check if $\alpha \geq b$
2. If yes, subtract it (in place) and write 1 to the result
3. If not, write 0 to the result
4. Shift a one left

What is the running time?

Our two-tape Turing machine M_{div} carrying out the long division expects the input string to be of form $w = a_{k-1} \ldots a_0 / b_{m-1} \ldots b_0$, so that $n = |w| = k + m + 1$. We find that M_{div} runs in time

$$f(n) = n^2 + 6n + 7.$$

Average-time analysis

Similarly, the mean running time on condition $|w| = n$:

Definition 49

Let M be a deterministic Turing machine that halts on all inputs. The average-time complexity of M is the function $g : N \to N$, where $g(n)$ is the **mean number of steps** *that M uses on inputs of length n (uniform distribution).*

In terms of $r(w)$

$$g(n) = \frac{1}{|\Sigma|^n} \sum_{|w|=n} r(w)$$

Asymptotic Analysis

Issues with $r(w)$ and $f(n)$:

1. Exact running time $r(w)$ is often a complex expression
   - We were "lucky" with *TM* M_{div}
   - With M_{π}, the trend is clear but there is "inherent noise"
   - In practice, the situation is often much worse!
2. Short inputs often irrelevant as they are "fast" anyway

Instead:

Asymptotic behavior is interesting when n gets larger

- This is where the difficult instances tend to be

Recall that with polynomials, the highest degree term will eventually dominate:

$$a_0 + a_1 x + a_2 x^2 + \ldots + \underline{a_k x^k} \quad \Rightarrow \quad \underbrace{x^k}_{\text{Interesting part!}}$$

(In plots, consider the log-scale)

Big-O notation

Definition 50 (Def. 7.2)

Let f and g be functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that $f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$, $f(n) \leq cg(n)$. When $f(n) = O(g(n))$, we say that $g(n)$ is an upper bound for $f(n)$, or more precisely, that $g(n)$ is an asymptotic upper bound for $f(n)$, to emphasize that we are suppressing constant factors.

Big-O notation gives an upper bound (for large n):

Polynomial bounds:	n^c,	$c > 0$
Exponential bounds:	$2^{(n^\delta)}$,	$\delta > 0$

Big-O: "No more than" bound

Small-o notation

Definition 51 (Def. 7.5)

Let f and g be functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that $f(n) = o(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

That is, $f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists s.t. $f(n) < cg(n)$ for all $n \geq n_0$.

Small-o: $f(n)$ is asymptotically less than $g(n)$

Analysing Algorithms

Example 73 (TM deciding on lang. $A = \{0^n1^n \mid n \geq 0\}$)

$M_1 = M_1(w)$:

1. Scan across the tape
   Reject if a 0 is found to the right of a 1
2. Repeat if both 0s and 1s remain on the tape:
   Scan across the tape
   crossing off a single 0 and a single 1
3. If 0s or 1s still remain, *reject*, otherwise, *accept*

Stage 1:

- Scan verifies that the input is of the form 0^*1^*
- Scan uses n steps, where n is length of the input
- Repositioning the head at the start uses another n steps
- In total $2n$ steps $\Rightarrow$ Big-O, $O(n)$ steps

Stages 2:

- Repeated scans crossing off a 0 and 1 each time
- Each scan
  - Uses $O(n)$ steps
  - Crosses off two symbols $\Rightarrow$ at most $n/2$ rounds
- Total time: $(n/2)\,O(n) = O(n^2)$ steps

Stage 3:

- Single scan to decide whether to accept or reject
- The time taken is at most $O(n)$

Sum up:

- The total time of M_1 on an input of length n is

$$O(n) + O(n^2) + O(n) = \boxed{O(n^2)}$$

Summary so far

- We have Big-O and small-o notations
- We know how to analyze a given TM (that halts)
- Thus, we can argue how algorithm performs asymptotically
  - Ignoring constant factors

(In practice a constant factor improvement can be significant!)

Time complexity class

Definition 52 (Def. 7.7)

Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the **time complexity class**, *$TIME(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.*

Example 74

TM M_1 deciding on language $A = \{0^n 1^n \mid n \geq 0\}$ was $O(n^2)$, and thus $A \in \text{TIME}(n^2)$.

Faster algorithm for $A = \{0^n1^n \mid n \geq 1\}$

Example 75

$M_2 = M_2(w)$ for $A = \{0^n1^n \mid n \geq 1\}$:

1. Scan the input and *reject* if a 0 is found to the right of a 1
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*
4. Scan again, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1
5. If no 0s and no 1s remain, *accept*. Otherwise, *reject*

TODO

1. Verify the algorithm
2. Determine the time complexity

Time complexity of M_2

- Every stage takes $O(n)$ time
- How many times each is executed?
- Stages 1 and 5 are executed once:
  - Takes a total $O(n)$ time
- Stage 4 crosses off at least half the 0s and 1s each time it is executed, so at most $1 + \log_2 n$ iterations
- Total time of stages 2, 3, and 4 is $(1 + \log 2n)O(n) = O(n \log n)$
- The running time of M_2 is $O(n) + O(n \log n) = \boxed{O(n \log n)}$

Improvement, from $O(n^2)$ to $O(n \log n)$

Aftermath for M_1 vs. M_2

Summary of TMs deciding on CFL $A = \{0^n 1^n \mid n \geq 1\}$:

- First, $A \in \mathrm{TIME}(n^2)$
- Now, $A \in \mathrm{TIME}(n \log n)$
  - This cannot be further improved on single-tape TM

Theorem 76 (Kobayashi 1985)

Any language that can be decided in $o(n \log n)$ time on a single-tape Turing machine is regular.

Language $A = \{0^n1^n \mid n \geq 1\}$ with two-tape TM

Time-complexity can be reduced further with *"better hardware"*:

Example 77

$M_3 = M_3(w)$ for $A = \{0^n1^n \mid n \geq 1\}$:

1. Scan across tape 1 and reject if a 0 is found to the right of a 1
2. Scan across the 0s on tape 1 until the first 1
   - At the same time, copy the 0s onto tape 2
3. Scan across the 1s on tape 1 until the end of the input.
   - For each 1 read on tape 1, cross off a 0 on tape 2.
   - If all 0s are crossed off before all the 1s are read, *reject*.
4. If all the 0s crossed off, *accept*; otherwise *reject*.

M_3 *decides on* A *in* **linear time**, $O(n)$

"Time out"

Computability theory: The ChurchTuring thesis

"All reasonable models of computation are equivalent"

They all decide the same class of languages.

Complexity theory:
The choice of model affects the time complexity.

Example 78

Language $A = \{0^n 1^n \mid n \geq 1\}$

- Decidable in $O(n \log n)$ time with 1-tape TMs
- Decidable in linear time $O(n)$ with 2-tape TMs

Still we want to classify computational problems according to their time complexity!

But with which model?

Single-tape TM vs. Multitape TM

Theorem 79 (Thm. 7.8)

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

Proof:
(Idea): In *Theorem 3.13*, we showed how to convert any *multitape TM* into a single-tape TM that simulates it. One can analyze that simulation to determine how much additional time it requires.
It turns out that simulating each step of the multitape machine uses at most $O(t(n))$ steps on the single-tape machine. Hence the total time used is $O(t^2(n))$ steps. □

Nondeterministic TMs

Nondeterministic TMs:

- How to measure time with them?
- Recall the power sets!

Definition 53 (Def. 7.9)

Let N be a nondeterministic Turing machine that is a decider. The running time of N is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.

Nondeterministic TMs: running time

The running time with NTMs is illustrated below:

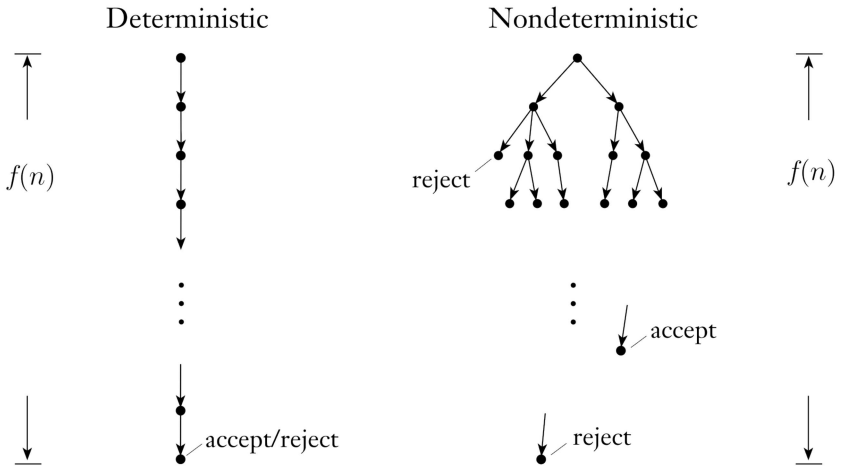


Figure: Deterministic vs. nondeterministic TMs (Fig. 7.10 / book).

Note:

- Def. of the running time of NTMs **is not** intended to correspond to any real-world computing device
- A useful mathematical definition that assists in characterizing the complexity of an important class of computational problems (see later)

NTM vs. Single-tape TM

Theorem 80 (Thm. 7.11)

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

Proof:
(Sketch) Proof of *Thm 3.16* constructed a deterministic TM D that simulated N by searching Ns nondeterministic computation tree. Results follows from analyzing that simulation. □

Summary

Theorem 7.8:

"At most a square or polynomial difference between the time complexity of problems measured on deterministic single-tape and multitape TMs"

Theorem 7.11:

"At most an exponential difference between the time complexity of problems on deterministic and nondeterministic TMs"

Polynomial differences are considered to be small,
whereas exponential differences are considered to be large

Summary (2)

- Exp. time is typical with **exhaustive search** a.k.a. **brute-force search**
- All reasonable deterministic computational models are **polynomially equivalent**
  - They can simulate each another with a polynomial increase in running time
- What is "reasonable"?
  - "models that closely approximate running times on actual computers"
  - E.g., *Theorem 7.8* shows that the deterministic single-tape and multitape Turing machine models are polynomially equivalent.

Fast Multiplication – Brief History

"How to multiply two n-bit integers?"

	The long multiplication was used for centuries	$O(n^2)$
1960	Anatoly Karatsuba proposes a better way	$O(n^{1.58})$
1965	FFT enters the game	$O(n \log(n) \cdot \log(\log(n)))$
1971	Schönhage and Strassen conjecture *"the true complexity is $O(n \log(n))$"*	
	(50 years passes . . .)	
2019	Harvey's and van der Hoeven's algorithm[8]	$O(n \log n)$

[8] David Harvey, Joris van der Hoeven. *Integer multiplication in time $O(n \log n)$*. March 2019. (hal-02070778).

Next steps

- Next we focus on aspects of time complexity theory that are unaffected by polynomial differences in running time
- This allows a theory that is independent of the selection of a computation model ("hardware")
- Goal: fundamental properties of computation

  (not properties of TMs or alike)

Class P

Definition 54 (Def. 7.12)

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine,

$$P = \bigcup_k TIME(n^k)$$

Note:

- k can be arbitrarily large, but still finite
- P is *invariant* for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine
- P roughly corresponds to the class of problems that are realistically solvable on a computer

Reasonable encoding

- Encoding method, e.g., $\langle w \rangle$, affects the result
- *Reasonable* methods allow for *polynomial time* encoding and decoding of objects into natural internal representations or into other reasonable encodings
- Familiar encoding methods for graphs, automata, and the like all are reasonable
- Reasonable encoding for graphs:
  1. List of nodes and edges
  2. Adjacency matrix (binary)
- Unary notation for encoding numbers isnt reasonable
  - It is exponentially larger than truly reasonable encodings, such as base k notation for any $k \geq 2$

Problem: Path from s to d

The PATH problem is to determine whether a directed path exists from s to d.

Definition 55 (PATH)

Let $G = (V, E)$ denote a directed graph, with $m = |V|$ nodes, and (s, d) a source-destination node pair. Then

$$PATH \triangleq \{\langle G, s, d\rangle \mid G \text{ is a directed graph w/ a path from } s \text{ to } d\}$$

Algorithm to find such a path?

Possible approaches:

1. Brute-force
2. Random walk (cf. Machine learning)
3. Dynamic programming

Path $s \to d$: Brute force

- Examine all potential paths in G
- How to enumerate them?
  - Loops obviously can be excluded
- **Idea 1:**
  - Path has at most m nodes
  - Consider m-tuples of nodes
  - Search space has m^m elements
  - Not very efficient but includes all potential paths
- **Idea 2:**
  - Each permutation π of V maps to a potential path
  - Search space has $m!$ elements
  - All potential paths are included

Both can be further reduced, e.g., by fixing the first node:

- Idea 1: $(m-1)^{(m-1)}$
- Idea 2: $(m-1)!$

Both are still exponential in the number of nodes ...

Path $s \to d$: Random walk

Probabilistic "random walk" algorithm:

$r_0 \leftarrow s$
repeat
  $e \leftarrow$ random edge from E with origin r
  $r \leftarrow$ destination of e
until $r = d$
return True

- This algorithm may eventually find the answer ...
- Maximum running time, however, is infinite[9]
- Hence, not very satisfactory for us
- But often used, e.g. in Machine learning

[9] Except in trivial cases where "wrong" turns cannot be made.

Path $s \to d$: in P

Theorem 81 (Thm. 7.14)

$PATH \in P$

Proof:
Construct a polynomial time algorithm:

$M = M(\langle G, s, d \rangle)$:

1. Mark node s
2. Repeat until no additional nodes are marked:
   - Scan all the edges (a, b) in G: If a is marked and b unmarked, mark b
3. If d is marked, *accept*, otherwise *reject*

TM M clearly "finds" a path if such exists, its running time is polynomial, and hence M is a polynimial time algorithm for PATH. □

Recap on Prime numbers

Definition 56

A positive integer number $n > 1$ is **a prime number** *if it is divisable only by itself and* 1.

- Hence $2, 3, 5, 7, 11, \ldots$ are the prime numbers
- Each $k > 1$ is either a prime or a *composite number*

Theorem 82 (Unique factorization theorem)

Each positive integer n has an unique factorization to primes

$$n = p_1^{n_1} \cdots p_k^{n_k} \qquad (n > 1)$$

where the p_i are k distinct primes and the $n_i \in \mathbb{N}$

Factoring of large integers is an interesting problem:

- Brute force: try 2 and every odd k less than $\sqrt{n}$
- If n is large, choose candidates randomly?
- Much better algorithms exist ... but still demanding!

RSA algorithm

Prime numbers are used in public key cryptography:

Example 83 (RSA)

Choose two prime numbers p and q, and compute

$$n = pq$$
$$\phi = (p-1)(q-1)$$

Choose the public exponent e such that $1<e<\phi$ and $\gcd(e, \phi) = 1$
Compute the secret key d such that $de \equiv 1 \mod \phi$

Encoding: compute $c = m^e \mod n$ (public key (n, e) is shared)
Decoding: compute $m = c^d \mod n$ (private key (n, d) is kept)

Relies on assumption that factoring large numbers is difficult!

Example

Example 84 (Xbox)

The original Xbox used the following public key in signing binaries:

20740119327258723760276023509063017138455993606274883526731955113241109007354362374128996096291046353572306742110305456946824862203867115042369878729703475765112280167498189046437794602966168812419423365196979669431929588951126804648743029387833666031765734337165949634731375592471670294246180877815104812674626967450097045005117546657068700545263064105024888769118032059917845867653040419404003684559882509195398630922824050405379620513589699993980205694266973236095772153476388267418476533663512746243310317853861946430053072890502949319703765023792161144942611323629444409600173894963797156859916567288947565058003

Can you factor the beast?! (and run your own binaries)

The original Xbox (from Wikipedia).

PRIMES

"Is x a prime?"

Determining if x is a prime is an important question:

- cf. RSA algorithm

Definition 57 (Language of prime numbers)

$PRIMES = \{\langle x \rangle \mid x \text{ is prime}\}$

So $10^{6400} - 10^{6352} - 1 \in$ PRIMES!

Theorem 85 (AgrawalKayalSaxena (2002))

$PRIMES \in P$.

Further reading:

`http://www.ams.org/notices/200305/fea-bornemann.pdf`

Relative primes

Definition 58

Two positive integer numbers $p, q > 1$ are **relative primes** *if $gcd(p, q) = 1$*

Let RELPRIME be the problem of testing whether two numbers are relatively prime

Definition 59

$RELPRIME = \{\langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime}\}$

Theorem 86 (Thm. 7.15)

$RELPRIME \in P$.

Proof (1)

Proof:

TM E computes the *gcd* using the Euclidean algorithm:

$E = E(\langle x, y \rangle)$, $x, y \in \mathbb{N}$ in binary (Euclidean Alg.):

1. Repeat until $y = 0$
2. Assign $x \leftarrow x \bmod y$
3. Exchange x and y
4. Output x $(=gcd(x, y))$

TM R solves RELPRIME using E as a subroutine:

$R = R(\langle x, y \rangle)$, $x, y \in \mathbb{N}$ in binary:

1. Run $E(x, y)$
2. If the result is 1, *accept*, otherwise *reject*

Note:

- R runs in polynomial time, if E does
- Correctness of E is well-known
- Remains to analyze the running time of E

Proof (2): Running time of E (Euclid's alg.)

- Stage 2:
  - After the first round, $x > y$
  - In consecutive rounds, x drops by at least half as[10]
    - If $x/2 \geq y$, then $x \bmod y < y \leq x/2$
    - If $x/2 < y$, then $x \bmod y = x - y < x/2$
- The values of x and y are exchanged every time stage 3 is executed, so each of the original values of x and y are reduced by at least half every other round.
- Stages 2 and 3 are run at most $2 \log_2 \min\{x, y\}$ times
- These logs are proportional to the lengths of the representations $\Rightarrow$ the number of stages run is $O(n)$.
- Each stage of E uses only polynomial time, so the total running time is polynomial.

[10] The modulo operation corresponding to remainder can be computed with an almost identical TM to M_{div} with a polynomial running time.

Context-free languages

Theorem 87 (Thm. 7.16)

Every context-free language is a member of P

Proof:
See the book. □

Example Languages from P

Example 88 (Perfect square)

The question whether $\sqrt{x}$ is an integer or not,

$$\text{PERFECT} = \{x \in \mathbb{N} \mid \exists\, n \in \mathbb{N} \text{ s.t. } x = n^2\}.$$

- Basic (inefficient) brute force solution:
  For $i = 0, 1, \ldots$ until $i^2 \geq x$, and report if $i^2 = x$
- Plenty of faster options exist!

Example 89 (Digits of Pi)

Does a given string w represents π (up to given accuracy):

$$\text{PI} = \{w \in \{0,1\}^* \mid \ w \text{ has } |w| \text{ digits of } \pi\}$$

Note: the binary point is implicit, e.g., $11001 \in \text{PI}$ as $11.001 \approx \pi$.

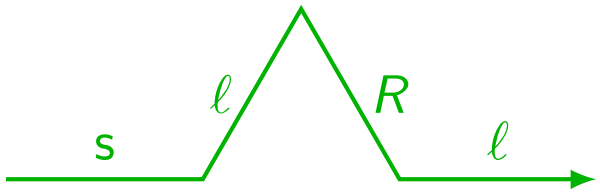
Example Languages from P (2)

Example 90 (Loops)

The question whether a path in the triangular grid is a loop

$$\mathrm{LOOP} = \{w \in \{L, \ell, s, r, R\}^* \mid \text{route } w \text{ is a loop}\}.$$

- w defines the changes in the direction:
  - L steep left
  - ℓ left
  - s straight
  - r right
  - R steep right
- State is the (position, heading)-pair



Class NP

- So we managed to avoid brute-force search in several problems
- . . . and obtained polynomial time solutions
- Attempts to avoid brute force in some other problems has been unsuccesful
  - No polynomial time algorithm is known
  - Neither is it proven that they do not exist
  - This class is known as **NP**

(NP does not stand for *"no problem"* in our context!)

Hamiltonian path

Hamiltonian path is a well-known combinatorial problem:

- Given a directed graph $G = (V, E)$
- Find a path that visits every node exactly once

Fixing two nodes gives HAMPATH problem:

Definition 60 (Hamiltonian Path problem)

$$HAMPATH \triangleq \{\langle G, s, d\rangle \mid \ G \text{ is a directed graph with h. p. } s \to d\}$$

Polynomial time verifiability:

Claim that $\langle G, s, d\rangle \in$ HAMPATH, i.e., a hamiltonian path from s to d exists, can be verified in polynomial time if such path is provided as *a certificate*.

Note: all exponential time algorithms devised for PATH (*Thm. 7.14*) can be modified for the HAMPATH problem

Composite numbers

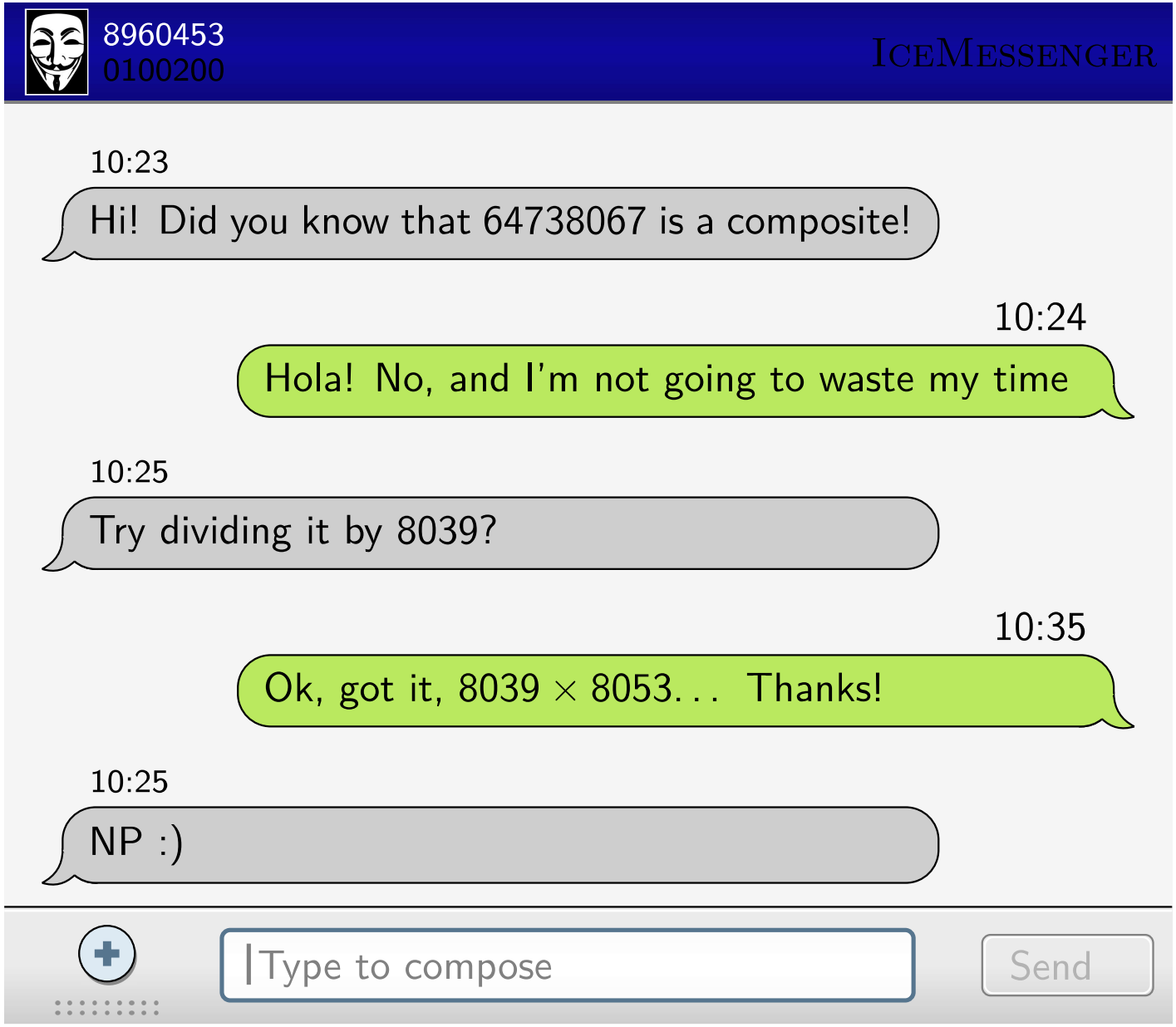
Definition 61

$COMPOSITES = \{x \mid x = pq \text{ for integers } p, q > 1\}$

Polynomial time verifiability:
Claim that $x \in$ COMPOSITES can be verified in polynomial time given a factor (p or q in above) is provided as a *certificate*.

In general, **a certificate** c can be any string that makes the verification process "easy".

> *"We can verify that $w \in A$ as we were given also c"*



Polynomially Verifiable

Definition 62 (Def. 7.18)

A verifier for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w \rangle \text{ for some string } c\}$$

We measure the time of a verifier only in terms of the length of w, so a polynomial time verifier runs in polynomial time in the length of w. A language A is polynomially verifiable if it has a polynomial time verifier.

- w describes a problem instance (e.g., a composite number x)
- c is **the certificate**, that makes the verification easy (e.g., a factorization $x = pq$)

Class NP

Definition 63 (Def. 7.19)

NP is the class of languages that have polynomial time verifiers.

- The term NP comes from **nondeterministic polynomial time** and is derived from an alternative characterization by using nondeterministic polynomial time Turing machines.
- Problems in NP are sometimes called NP-problems
- Problems in P are obviously also in NP

Example

Example 91 (NTM for *HAMPATH*)

$N_1 = N_1(\langle G, s, d \rangle)$:

1. Write a list of m numbers, $p_1, \ldots, p_m$, $m = |V|$
   Each number in the list is nondeterministically selected to be between 1 and m.
2. Check for repetitions in the list. If any are found, *reject*
3. Check whether $s = p_1$ and $d = p_m$. If either fail, *reject*
4. For each i between 1 and $m - 1$, check whether $(p_i, p_{i+1}) \in E$. If any are not, *reject*. Otherwise, *accept*

Each stage runs in polynomial time. Thus N_1 runs in nondeterministic polynomial time.

Class NP via nondeterministic TMs

Theorem 92 (Thm. 7.20)

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

Proof (sketch):
We show how to convert a polynomial time verifier to an equivalent polynomial time NTM and vice versa. The NTM simulates the verifier by guessing the certificate. The verifier simulates the NTM by using the accepting branch as the certificate. □

NTIME

The nondeterministic time complexity class NTIME($t(n)$) is defined similarly as TIME($t(n)$):

Definition 64 (Def. 7.21)

$NTIME(t(n)) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time NTM}\}$

Corollary 93 (Corollary 7.22)

$NP = \bigcup_k NTIME(n_k)$.

- The class NP is *insensitive* to the choice of reasonable nondeterministic computational model
- All such models are polynomially equivalent

Analysis on nondeterministic TMs

- Each stage of an NP algorithm must have an obvious implementation in nondeterministic polynomial time on a reasonable nondeterministic computational model
- Show that every branch uses at most polynomially many stages

Cliques in Graphs

Definition 65

A **clique** *in an undirected graph is a subgraph, wherein every two nodes are connected by an edge. Similarly, k-clique is a clique with k nodes.*

The k-**clique problem** is to determine whether a graph contains a clique of size k:

Definition 66

$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

A related problem is about finding the largest clique:

Definition 67 (max-clique problem)

Determine the largest clique in graph $G = (V, E)$.

Cliques (2)

Theorem 94 (Thm. 7.24)

CLIQUE is in NP.

Proof:
The following is a verifier V for CLIQUE.

$V = V(\langle G, k, c \rangle)$:

1. Test whether c is a subgraph with k nodes in G
2. Test whether G contains all edges connecting nodes in c
3. If both pass, *accept*; otherwise, *reject*

□

Knapsack and Subset-Sum Problems

Definition 68 (0-1 Knapsack Problem)

Given n items with weights w_i and values v_i, choose a subset of items that maximizes the total value without exceeding a given maximum weight W.

Definition 69 (Subset-Sum Problem)

Given n items with weights w_i, is there a subset of items with a total weight of W.

Alternatively, is there a non-empty subset whose sum is zero?

Time complexity of SUBSET-SUM

Definition 70 (SUBSET-SUM language)

$$\begin{aligned} \mathit{SUBSET\text{-}SUM} = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \\ \text{and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \\ \text{we have } \sum y_i = t\} \end{aligned}$$

Theorem 95 (Thm. 7.25)

SUBSET-SUM is in NP

Proof:
(sketch) The subset is itself the certificate. □

P and NP

NP class:

1. solvable in polynomial time on an NTM, or
2. membership can be verified in polynomial time (on a TM)

P class:

1. solvable in polynomial time on a TM

In other words:

- **NP class**: membership can be **verified** quickly
- **P class**: membership can be **decided** quickly

Is $P = NP$?

- For example, *HAMPATH* and *CLIQUE* are in NP
  - Are they also in P?
- In fact, we do not know if NP=P or NP≠P

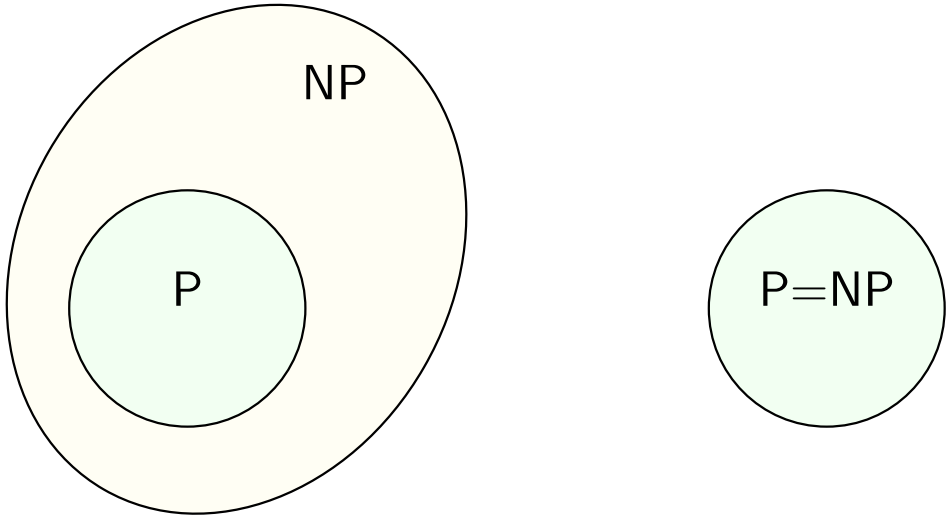


Figure: Two options, but only one can be true!

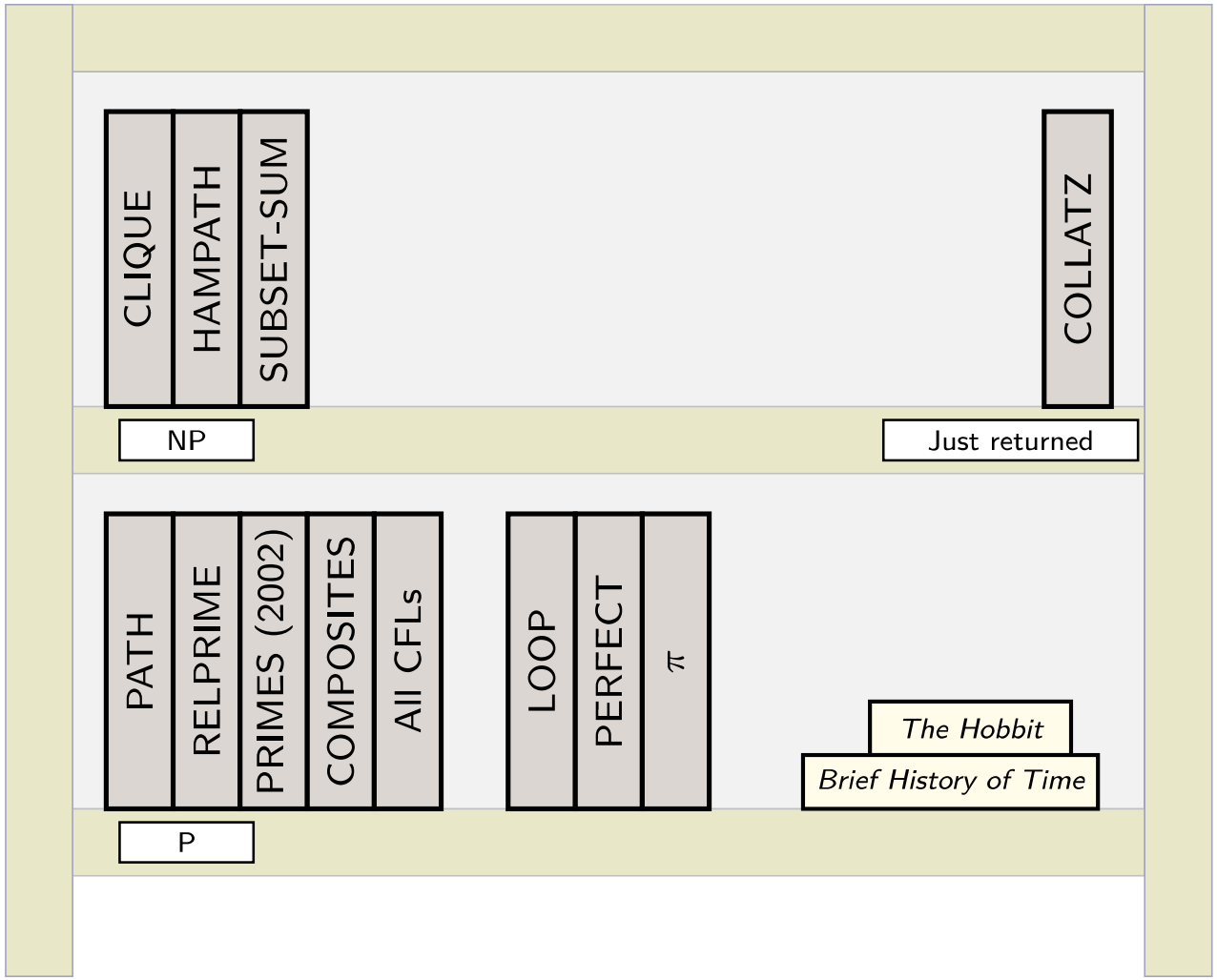
One of the greatest unsolved mysteries in CS!

Is $P = NP$? (2)

Common belief is that NP $\neq$ P

- Huge efforts invested to show the (in)equality
- . . . but no success (yet!?)

Great Library of Time Complexity



Books in order! *(if $N \neq NP$)*

Summary of Today

- Running time = *number of steps TM takes to decide*
- Asymptotic behavior, $n \to \infty$, is interesting
- Time Complexity Analysis of TM:
  - Condition on input length $|w| = n$
  - Derive an asymptotic upper bound for running time
  - Depends on computational model ...
- Class P problems: *decidable* in polynomial time
- Class NP problems: *verifiable* in polynomial time

Thanks!