AGENDA FOR NEXT TWO LECTURES:

The limits of efficient computation

- What is a computational problem?
- Decision problems
- What is efficient computation?
- What is a computer? Turing machines
- Uncomputable problems
- Efficient computation — the class P
- Reductions
- Solving problems vs verifying solutions
- The class NP
- NP-completeness — the problems right at the limit of what can be computed

LECTURER

JAKOB NORDSTRÖM
PROFESSOR IN THE ALGORITHMS & COMPLEXITY SECTION AT DIKU
http://www.jakobnordstrom.se

In order to do rigorous study of computation, need formal mathematical setting defining
- what is a computer?
- what does it mean to solve a problem efficiently?
- what is a computational problem?

COMPUTATIONAL PROBLEM

Represent objects as strings in some alphabet Σ

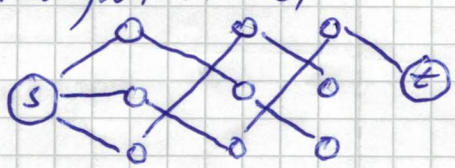
Think of Σ as
- 0 or 1
- a-z A-Z 0-9 _ - () [] <>
- ASCII or UTF-8

Choose Σ as convenient. Does not matter as long as size finite $|\Sigma| < \infty$

STRING $s \in \Sigma^*$ sequence of 0 or more characters from Σ

Some example problems:

(1) Given graph G, vertices s, t, find path in G from s to t (or report none exist)



(2) Given integer N, find prime factors $N = 25957$

(3) Given propositional logic formula, find satisfying assignment (or report none exist)

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3)$$

(4) Given integers $A_1, \ldots, A_n$ and target T, find subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} A_i = T$ (or report none exist) $\{2, 3, 5, 7\}$, target 11

Encoding issues

1) We can clearly encode all these problems as strings in some way
2) From now on, assume we've agreed on some reasonable encoding
   - Details of course matter in practice
   - Not really important for our discussion
   - Avoid silly encodings, e.g., unary (5 encoded as "11111")

SEARCH PROBLEM

Given $x \in \Sigma^*$ encoding problem instance, compute solution $y \in \Sigma^*$

Ex some path; some satisfying assignment

FUNCTION PROBLEM

If every problem has a unique answer, this defines function $f : \Sigma^* \to \Sigma^*$

Ex prime factors sorted in increasing order; lexicographically smallest path from s to t viewed as string in Σ^*

We will simplify even more

DECISION PROBLEM

Consider functions $f : \Sigma^* \to \{0, 1\}$

Think of $\boxed{\begin{array}{l} 0 = \text{"no"} \\ 1 = \text{"yes"} \end{array}}$

Examples

(1') Is there a path in G from s to t?

(2') Is there a prime factor of N of size at most U?

(3') Is the given formula satisfiable?

(4') Is there a subset of $\{A_1, \ldots, A_n\}$ summing to the target T?

Much cleaner to work with mathematically

Often does not matter - efficient algorithm for decision problem will give efficient algorithm for search problems

(Can you work this out for the problems above?)

DECISION PROBLEMS AND LANGUAGES

Decision problem
$f : \Sigma^* \to \{yes, no\}$

$\Leftrightarrow$

Language $L \subseteq \Sigma^*$
$L = \{x \in \Sigma^* \mid f(x) = yes\}$

$f(x) = 1 = yes$ x is "yes instance"
$f(x) = 0 = no$ x is "no instance"

Historical terminology
Rich literature
No way of avoiding it...

We say that an algorithm that computes f DECIDES L

What do we do with strings that are not valid encodings?

Ex If a graph is a list of the edges, is there a path from s to t in
$(s,a),(a,b),)c,b,a(,(b,t))$?

"Syntax error" - Define $f(s) = no$ for strings s that are not valid encodings

EFFICIENT COMPUTATION

What can be computed within reasonable limits on resources such as

- computation time
- computation memory
- computation energy
- network communication

Most important measure for us: **TIME**

Measure how the number of operations needed scale with the size of the input

- ignore constants depending on low-level details
- look at asymptotic behaviour as input size grows

ASYMPTOTIC NOTATION

$T(n)$ is $O(g(n))$ if exist positive constants c, N s.t. for $n \geq N$ it holds that $T(n) \leq c \cdot g(n)$

$T(n)$ is $\Omega(g(n))$ if exist positive constants c, N s.t. for $n \geq N$ it holds that $T(n) \geq c \cdot g(n)$

$T(n)$ is $\Theta(g(n))$ if $T(n)$ is $O(g(n))$ and $\Omega(g(n))$

$T(n)$ is $o(g(n))$ if for all $\varepsilon > 0$ exists N s.t. for $n \geq N$ $T(n) \leq \varepsilon \cdot g(n)$

$T(n)$ is $\omega(g(n))$ if for all $K > 0$ exists N s.t. for $n \geq N$ $T(n) \geq K \cdot g(n)$

But what is our computational model?

The TURING MACHINE

- Seems to be able to simulate all physically realizable computational methods with little overhead
- But very simple, so mathematically nice to work with
- But so simple and stupid that they are very annoying to deal with
  - CLRS completely skips details
  - We will follow Chapter 1 in Arora-Barak, but will be brief and informal

TURING MACHINE (TM)

- Fixed alphabet Σ (of finite size)
- Program Q (or "set of states" of TM)
- Tapes
  - input tape, contains input, read-only (after input, special EOF/blank symbol)
  - work/output tape (initialized to EOF/blank symbols)

  Tapes have a starting position but no end
- Read-write heads positioned on tapes (start in starting position)

At each time step the TM:

(a) reads symbols at current position on all tapes
(b) writes symbol to work tapes
(c) move tape head left or right one step (or stand still)
(d) jump to new state $q' \in Q$

(b) – (d) depend on
- current state q
- ~~positions~~ symbols read on tapes in (a)

Special state q_{halt} — TM stops
Running time # steps before reaching q_{halt}

To compute a function:
- write value on output tape
- then move to q_{halt}

Ex TM that decides whether # 1s in binary string odd

ALWAYS
~~q_{start}~~: Read symbol s on input tape, move input head right

q_{start}: If $s = 0$ go to q_{start}
If $s = 1$ go to q_{odd}
If $s = EOF$ write 0 on output and go to q_{halt}

q_{odd}: If $s = 0$ go to q_{odd}
If $s = 1$ go to q_{start}
If $s = EOF$ write 1 on output and go to q_{halt}

FACTS ABOUT TURING MACHINES

(1) Expressive: Can do "normal things" efficiently, so we can allow ourselves to write pseudocode (including subroutine calls)

(2) Robust to tweaks
- change of alphabet
- adding more work tapes

(3) Description of TM can be written as string and given as input to other TM

(4) There is a UNIVERSAL TURING MACHINE that can simulate any other Turing machine M given its string representation.

This is EFFICIENT — if original TM M runs in time T, then simulation runs in time $O(T \log T)$

All of this needs proving, of course, but we do not have time or patience for this now... so take it on faith.

From this it follows that there are UNCOMPUTABLE / UNDECIDABLE PROBLEMS!

Perfectly well-defined mathematical functions that cannot be correctly computed by any algorithm

HALTING PROBLEM

Fix alphabet Σ

Fix some way of encoding Turing machines

Consider the language

$$HALT = \{\langle M, x\rangle \mid \text{Turing machine } M \text{ halts on input } x\}$$

(As before, if M or x is not valid, then $\langle M, x\rangle$ is a no instance)

THEOREM The language HALT is not decidable by any Turing machine

Proof By contradiction.

Suppose that H is a TM that decides HALT. We can construct another TM H' that simulates H as a subroutine

Then we can feed H' to H with a suitable input

This is all legitimate, so if we reach a contradiction, then ~~H~~ cannot exist

TM H' with input M

```
if H(M, M) = yes then
    while true                  // infinite loop
    endwhile
else                            // H(M,M) = no
    halt
```

What does H' do when given input $M = H'$?

a) H' halts on $H' \Rightarrow$
   $H(H', H') = yes \Rightarrow$
   H' gets stuck in infinite loop

b) H' does not halt on $H' \Rightarrow$
   $H(H', H') = no \Rightarrow$
   H' halts

Contradiction! Hence H does not exist ∎

Another example:
Given set of polynomials with integer coefficients, do these equations have a common integral solution?

Undecidable!

Does not mean that no instance of these problems can ever be solved. But no algorithm can:
- always terminate
- always give correct answer

So some problems are undecidable
But even computable problems can be infeasible to solve in practice

$\boxed{P}$ = the set of languages that can be decided in POLYNOMIAL TIME, i.e. time $O(n^k)$ for some constant k (arbitrary but fixed)

Note
- P defined for decision problems
- Running time measured in size of input

CHURCH-TURING THESIS
Every physically realizable computational device can be simulated by a Turing machine

Not a theorem — it couldn't be — but consistent with what we currently know about nature

STRONG/EXTENDED CHURCH-TURING THESIS
Anything efficiently computable on any computational device can be efficiently simulated by a Turing machine (i.e., with at most polynomial overhead)

Might not be true if quantum computers can be built
But we think of P as capturing what is efficiently computable

Is P a reasonable model of efficiently solvable problems?

Pros

- Composes well: efficient programs can call efficient subroutines and stay efficient
- Exponents of polynomials in running times are often small
- Reasonable agreement between theoretical definition and what we see in practice

Cons

- Worst-case complexity – have to have polynomial running time for all problem instances – is too strict! What if difficulty due to some pathological instance never seen in practice?!
  - Not quite clear what "in practice" should mean mathematically
  - There have been attempts at average-case complexity
- Polynomial time is too slow! The small exponents we observe are due to that this is the kind of algorithms we can discover and understand. And for huge data sets, quadratic or sometimes even linear time is impractical
  - There is research into such scenarios
  - But P is still a relevant class
- Focusing on decision problems is too limited a framework!
  - Yes, sometimes. But surprisingly often not! Definitely not for the problems we are discussing here.

Cons, continued...

- What about computation in other physical models that might ~~be~~
  (a) be CONTINUOUS rather than discrete?
  - if so, we still need to measure, and to deal with noise
  (b) use randomness (obtained, say, from radioactive decay)
  - randomness can be useful in practice but does not seem to matter for our theoretical definition
  (c) use effects from quantum mechanics?
  - yes, that might make a difference — not for computability but for efficiency

---

We want to study different computational problems and understand how hard they are

In particular, is a given problem / language in P or not?

Turn out to be challenging to decide for many problems that we care about.

But we can use transformations, or REDUCTIONS, between problems to understand how the hardness of different problems are related.

REDUCTIONS

L_1 reduces to L_2, written $L_1 \leq L_2$ if exists efficient algorithm computing some function $g : \Sigma^* \to \Sigma^*$ such that

$$\begin{aligned} x \in L_1 &\implies g(x) \in L_2 \\ x \notin L_1 &\implies g(x) \notin L_2 \end{aligned}$$

Positive use case:

Have efficient algorithm A for L_2
Encounter a new problem L_1
If we can find a reduction from L_1 to L_2
then we can solve L_1 efficiently by
computing $A(g(x))$ for input x

"Solving L_1 is at least as easy as solving L_2"

Ex
- Reduce bipartite matching to max flow
- Encode problem as propositional logic formula and solve formula

Negative use case:

Believe (or know) that L_1 is a hard problem
Encounter a new problem L_2
If we can find a reduction g from L_1 to L_2,
then L_2 must be at least as hard as L_1

"Solving L_2 is at least as hard as solving L_1"

SOLVING A PROBLEM VS. VERIFYING SOLUTIONS

Doing an exam requires coming up with solutions — can be hard

Grading the exam just involves verifying correctness — much easier (hopefully)

Complexity class P

class of efficiently solvable (decision) problems (i.e., in polynomial time)

Complexity class NP

class of problems for which proposed solutions can be verified efficiently

Except we have decision problems, so what do we mean by a "solution"
Formally, some kind of auxiliary string (called "CERTIFICATE or WITNESS) that helps to verify that yes-instances are yes-instances.
For us, this will often be the solution to the ~~search~~ problem that the decision problem came from

Examples of certificates:

1. s-t-PATH: An ordered list of vertices forming the path.
2. FACTORING: The prime factorization of N
3. SATISFIABILITY: A satisfying assignment
4. SUBSET SUM: A subset summing to the target T

DEFINITION Language L is in **NP**, if
- exist polynomial p.
- exist polynomial-time Turing machine M (**verifier**) taking two arguments x, y such that

$$x \in L \iff \text{Exists } y \in \Sigma^* \text{ of length } |y| \leq p(|x|) \text{ such that } M(x, y) = 1$$

Again, y is called a **CERTIFICATE** or **WITNESS** for x

Why is **NP** an **interesting** problem class? Because for most practical problems that we want to solve by constructing some object, it is possible to check if a proposed solution meets the requirements

Is it possible to solve (all) problems in NP efficiently? We don't know.
One of the big open **MILLENNIUM PRIZE PROBLEMS** in modern mathematics

We will next take a closer look at NP and study the **hardest and most interesting** problems in this class — the **NP-complete problems**