Lecture 1 : Intro

Note Title 3/28/2013

Classify problems according to the computational resources required to solve them. In particular, we will consider:

- running time
- memory/space
- parallelism
- randomness
- rounds of interaction (2 parties communicating).

⇒ What can we compute with a limited set of resources?

⇒ Decidability
What is computable (ignoring restrictions on resources)?

For example, here are some of the big open questions in the field (& the conjectured answer to them).

Conjecture

NO * $P \stackrel{?}{=} NP$: is finding a solution more difficult than recognizing/checking one

NO * $P \stackrel{?}{=} NC$ is every efficient algorithm parallelizable?

NO * $P \stackrel{?}{=} L$ Can every efficient algorithm be turned into one that uses a very small amount of memory?

NO * $EXP \stackrel{?}{\subseteq} P/poly$ Are there small (poly-size) boolean circuits for all problems that can be computed in polynomial time?

YES * $P \stackrel{?}{=} BPP$ Can every efficient randomized algorithm be converted to a deterministic one?

If any of these conjectures is wrong, it will have a big impact on computation.

We will start with the following groundwork:

Problems + Languages
Complexity Classes
Turing Machines
Reductions
Completeness

We need a formal definition of a "computational problem"

Informally we say:

- Given a graph G + vertices s and t, find the shortest path from s to t in G.
- Given matrices A + B, compute AB
- Given an integer N, find its prime factors.
- Given a Boolean formula, find a satisfying assignment.

How to encode the inputs to these problems?

Σ: finite alphabet $\{0,1\}$ or $\{0,1,2,\dots,9\}$

Inputs are encoded in strings over the alphabet. This is done routinely in computer science. We will usually not worry too much about the details of the encoding but there needs to be an agreed upon interpretation of strings.

Its not necessary for every string to represent a valid problem instance. The specific encoding will effect finer grained analysis (e.g. adjacency matrix vs. adjacency list). We will mostly be concerned with higher level analysis. Avoid unary encodings.

Given an encoding a problem can be expressed formally as a function from strings to strings: $f: \Sigma^* \longrightarrow \Sigma^*$

given x, compute $f(x)$.

For a decision problem, we have $f: \Sigma^* \longrightarrow \{yes, no\}$.
given x, accept (yes) or reject (no).

We will work mostly with decision problems.
This simplification usually doesn't give up too much.

Given an algorithm to solve a decision problem, it can often be used to solve a related optimality/search problem without too much additional overhead.

Given n & k : is there a factor of $n < k$? (Decision).
Can use binary search to solve:
Given n, find its prime factors.

Given a Boolean Formula $\phi(x_1, \ldots, x_n)$ is it satisfiable (Decision)
Given $\Phi(x_1 \cdots x_n)$ find a statisfying assignment.
First test if $\Phi(x_1, \ldots, x_n)$ is satisfiable: If NO $\rightarrow$ STOP.
Is $\Phi(T, x_2, \ldots, x_n)$ satisfiable?
Yes: fix $x_1 \leftarrow T$
No: fix $x_1 \leftarrow F$
continue on to $x_2, x_3, \ldots, x_n$.

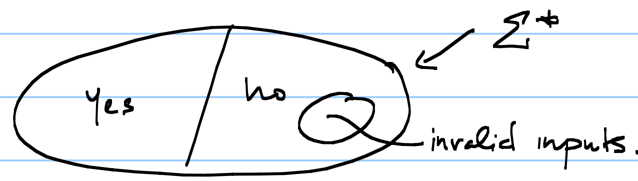
We will view decision problems as a language. (i.e. set).
This requires a fixed encoding of problem instances in Σ^*.

Language L is subset of Σ^* corresponding to "yes" instances.

e.x. · (n,k) s.t. n has a factor $< k$.
· all satisfiable boolean formulas.

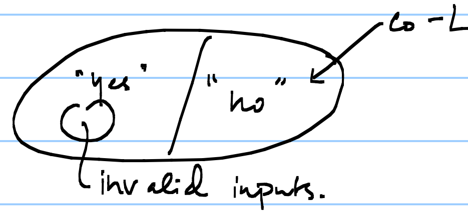
Decision problem: Given x, is x in L?

Strings which are not valid encodings are treated as no instances.



Complement of L co-L

Σ^* − {invalid inputs}.



So far languages are just a set-theoretic definition.
no reference to computation yet.

Complexity class: class of languages.
typically, these are defined by a computational constraint.

P = set of all languages decidable in polynomial time.
NP = set of L where

$$L = \{x \mid \exists y \ |y| \le |x|^k, \ (x,y) \in R\}$$

R is a language in P.

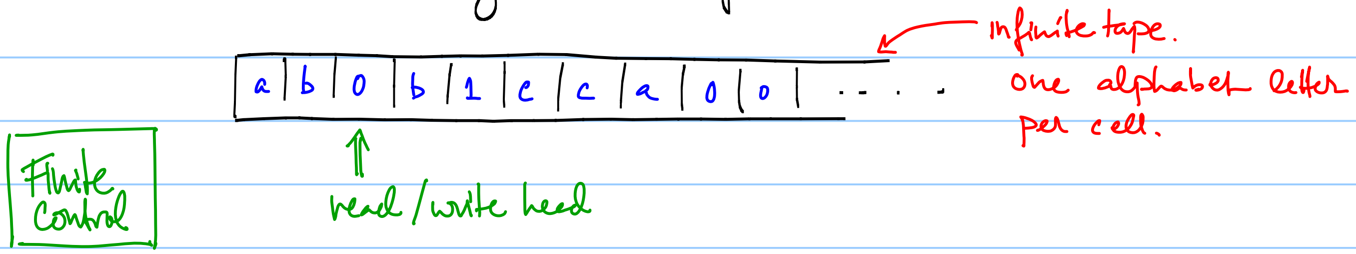
Meaningful complexity classes will have certain properties:

* Capture genuine computational phenomenon (e.g. parallelism).
* Contain natural & relevant problems

* Characterized by natural problems (completeness).
* robust under various models of computation.
* possibly closed under operations s.t. $\wedge, \vee, \neg$

To define a complexity class, we need a model of computation. We want this model to yield complexity classes that capture important aspects of computation.

We will use a Turing Machine for this:



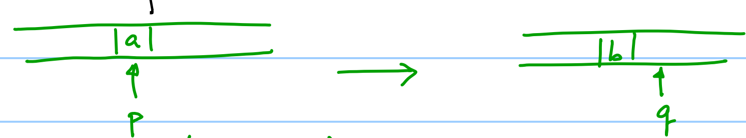
Turing Machine

Q: finite set of states.

Σ: finite alphabet including blank character $\sqcup$

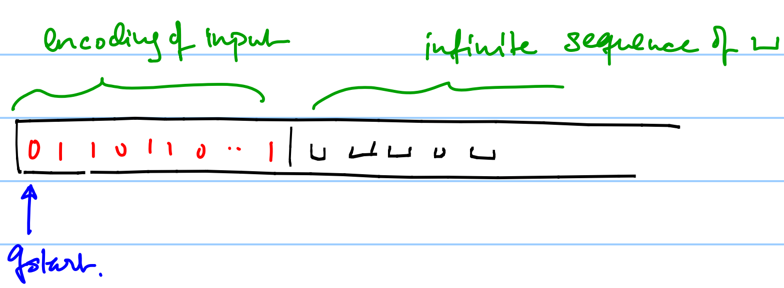
q_{start} q_{acc} q_{rej} special states in Q.

Transition function $\delta: Q \times \Sigma \to Q \times \Sigma \times \{L, R, -\}$



$\delta(p, a) = (q, b, R)$ (– means stay in same place).

Start configuration:



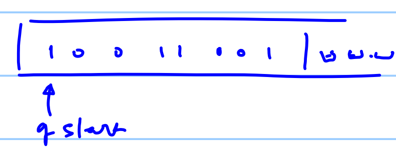
Sequence of steps specified by δ.
if q_{acc} or q_{rej} are reached, then halt.
(no outgoing transitions from these states)

3 ways for a Turing Machine to Compute
in all, input x written on tape.

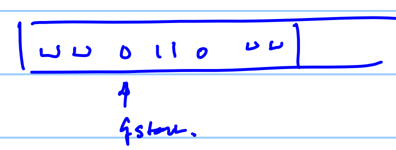
1. function computation: output $f(x)$ is left on tape when TM halts.
2. language decision: TM halts in state q_{acc} if $x \in L$. TM halts in state q_{rej} if $x \notin L$
3. language recognition: TM halts in state q_{acc} if $x \in L$; may loop forever otherwise.

TM: example. Is x a palindrome? $(x = x^R)$.

$Q = \{q_{start}\ q_1\ q_0\ q_2\ c_0\ c_1\ q_{acc}\ q_{rej}.\}$
$\Sigma = \{0, 1, \sqcup\}$



$q_{start}, 0$ $\sqcup\ q_0\ R$
$q_{start}, 1$ $\sqcup\ q_1\ R$
$q_{start}, \sqcup$ q_{acc}.



$q_0\ 0$ $0\ q_0\ R$
$q_0\ 1$ $1\ q_0\ R$
$q_0\ \sqcup$ $\sqcup\ c_0\ L$

Same for q_1
except $q_1\ \sqcup \to \sqcup\ c_1\ L$

$c_0\ 1$ q_{rej}.
$c_1\ 0$ q_{rej}.

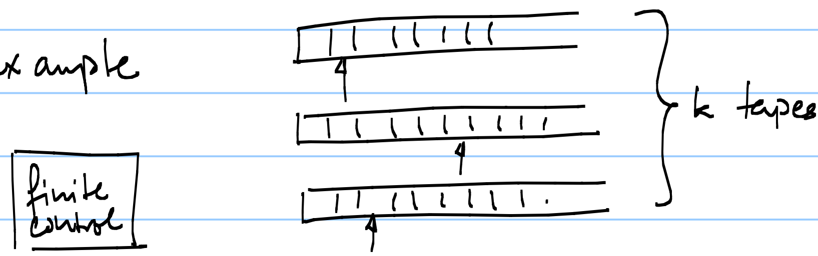
$c_0\ 0$ $\sqcup\ q_2\ L$
$c_1\ 1$ $\sqcup\ q_2\ L$

q_L 0	0 q_L L		c_0 ⊔	q_{acc}
q_L 1	1 q_L L		c_1 ⊔	q_{rej}.
q_L ⊔	⊔ q_{start} ⊔			

TM is very robust to variation:

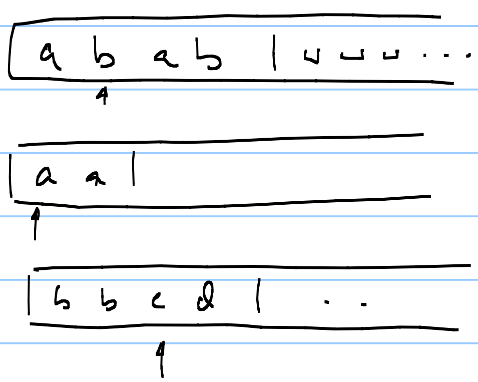
$\delta : Q \times \Sigma^k \to Q \times \Sigma^k \times \{L, R, -\}^k$

For example



Usually one read-only to hold input
one write-only to write output
$k-2$ read/write work tapes.

Can simulate a k-tape TM by a single tape TM



a $\boxed{b}$ a b # $\boxed{a}$ a # b b $\boxed{c}$ d # . . .

$\boxed{a}$ $\boxed{b}$ $\boxed{c}$ . . .

new alphabet characters denoting head location.

M k-tape TM $\longrightarrow$ M' 1-tape TM

$\Sigma \longrightarrow \Sigma \cup \Sigma^{\square} \leftarrow a \in \Sigma \leftrightarrow \boxed{a} \in \Sigma^{\square}$

One step of M $\longrightarrow$ head scans the entire length of the tape remembering each symbol at each head. Then scans back to implement the step on each tape.

M takes T steps on input x.
M' takes $O(T^2)$ steps on input x.

If a # is hit then the contents of the tape are moved over to make room.

When M halts: erase everything except the output string.

Turing Machines are clumsy at computation but they are a simple abstract model that allows us to formalize our intuitive notion of what it means to compute efficiently.

A multi-tape TM M computes a language L in time $t(n)$ if $\forall x$ on input x M "halts" (i.e. reaches q_{acc} or q_{rej}) in at most $t(|x|)$ steps.

$$x \in L \iff M \text{ accepts } x.$$

The "Extended" Church-Turing Thesis

Everything that can be computed in time $t(n)$ on a physical computing device, can be computed in time $[t(n)]^{O(1)}$ on a Turing Machine.

polynomial slow-down.

Quantum computers have been the only real challenge to this belief.

RAM, multi-tape TM, etc.

All physically realizable models of computation can be simulated by a TM w/ polynomial overhead.

Our first concern is what can be computed by a TM w/o any restrictions on running time.

L is decided by TM M if
$\forall x$ M halts on input x after a finite number of steps.
└ no other restriction.

$x \in L \iff M$ accepts x.

There are natural problems (languges) that can not be decided by any TM. $\Rightarrow$ undecidable.

$\text{Halt} = \{ \langle M, x \rangle : M \text{ halts on input } x \}$
└ requires an encoding of every Turing Machine M
for example, use ASCII char.

Theorem: HALT is Undecidable.

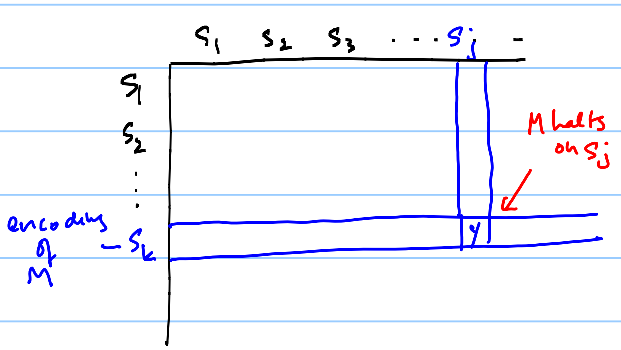
$\langle M \rangle$ will refer to an encoding of M. Similarly $\langle x \rangle$ refers to an encoding of x.

$\nexists$ a TM (program) that will always finish in a finite amount of time and determines whether $\langle M, x \rangle \in \text{HALT}$.

Proof by contradiction:
Fill in an infinite chart.
Rows + columns indexed by all strings in lexicograph order:

$s_1 \quad s_2 \quad s_3 \quad s_4 \quad \cdots$

0 1 00 01 10 11 000 ....



If s_k is a valid encoding of a TM M Fill in the row as follows:
If M halts on (s_j) $\langle M, s_j \rangle \leftarrow$ Yes
o.w. $\langle M, s_j \rangle \leftarrow$ NO

If s_k doesn't make sense as an encoding of a TM, then fill in the row with all "N".

Every TM M is encoded by some string s_k

Now suppose there is some TM H that decides HALT.

On input (s_k, s_j)

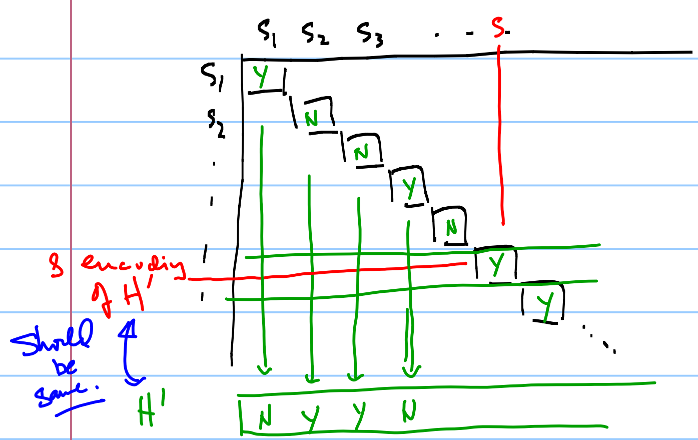
M outputs answer in square (s_k, s_j) in a finite number of steps.

New H': on input s_j run according to H on input (s_j, s_j)

if about to enter state q_{acc} → go to an inf loop.

if about to enter state q_{rej} → go to q_{acc} instead.

Consider diagonal of chart.



on input s_j H' does the opposite of what it says in square (s_j, s_j).

But H' is encoded by some string s.

If H' halts on s. $(s,s) = Y$ but then H' loops forever on s.

If H' loops forever on s $(s,s) = N$ H' accepts s.

This proof is an example of "diagonalization".

⇒ Profound implications for program testing & verification.

Back to complexity classes:

$TIME(f(n))$ for $f: \mathbb{N} \to \mathbb{N}$.

$\doteq$ the set of all languages decided by a multi-tape TM in at most $f(n)$ steps
(n is the # characters in the input).

$SPACE(f(n))$ set of all languages decided by a multi-tape TM and touches at most $f(n)$ tape squares.

$$P = \bigcup_{k \geq 1} TIME(n^k)$$

if $L \in P \Rightarrow L \in TIME(n^k)$ for some fixed k independent of the input.

Goal: ① Find an algorithm that decides L
② Prove that no algorithm does better (use of resources).

Good @ ①   Hopeless @ ②

Instead: relate difficulty of problems to each other by reductions. show that certain problems are the "hardest" in a complexity class. (complete)

Powerful + useful surrogate for ②

Reductions: relating problems to each other.

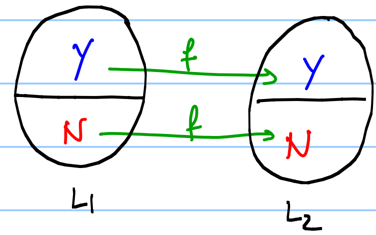
Two languages L_1 + L_2 $\quad$ L_1 "reduces to" L_2

$L_1 \propto L_2$

if $\exists$ an efficient (for now polynomial time) algorithm that computes f s.t.

$x \in L_1 \implies f(x) \in L_2$

$x \notin L_1 \implies f(x) \notin L_2$



If $L_1 \propto L_2$ and $L_2 \in P \implies L_1 \in P$.

L_1 is as easy as L_2.

If $L_1 \propto L_2$ and $L_1 \notin P \implies L_2 \notin P$.

L_2 is as hard as L_1.

Example $\quad$ 3SAT $\propto$ Independent Set.

3-SAT = { φ : φ is a 3-CNF formula that has a satisfying assignment }

3-CNF = 3 Conjunctive Normal Form.

AND of m clauses: each clause OR of ≤ 3 literals (x or $\neg x$)

ex: $\underbrace{(x_1 \vee x_7 \vee \neg x_9)}_{\text{clause}} \wedge (\neg x_2 \vee x_{17}) \wedge (x_5) \wedge \dots$

Independent Set

Input: Graph G, integer k

Is there an independent set of size k?

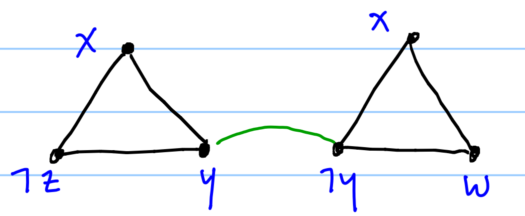
(set of vertices, no two of which are connected.)

Reduction Sketch:

Input: $\phi = (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee w) \ldots$

Output: G_ϕ, $k = m = \#$ clauses in ϕ.

G_ϕ: triangle for each clause in ϕ (or edge, vertex if <3 literals)



. . . . add edge between each y, $\neg y$ pair.

Indep set of size m $\Rightarrow$ Sat assignment.

one variable per triangle.

Let each literal chosen be true.

No contradictions because of green edges.

Sat assignment $\Rightarrow$ set of true literals.

corresponds to subset of nodes – at least one per triangle.

green edges not violated.

pick one vertex per triangle – black edges not violated.

Completeness Complexity class $\mathcal{C}$.

Language L is $\mathcal{C}$-complete if

a) $L \in \mathcal{C}$

b) $\forall L' \in \mathcal{C}$ $L' \propto L$.

L is the hardest problem in complexity class $\mathcal{C}$.

L is $\mathcal{C}$-hard if only b) holds

L is at least as hard as everything in $\mathcal{C}$.

This allows us to reason about an entire class by thinking only about a single concrete problem.

How to reduce every language in C to L?

- Show $L' \propto L$ for some L' that is C-complete.
- Hard to find the first C-complete problem.

For example

NP = set of languages L s.t. $\exists$ k, $R \in P$
$L = \{x \mid \exists y \ \ |y| \le |x|^k, \ (x,y) \in R\}$

Cook in 1971 showed that SAT (boolean satisfiability) is NP-complete.

Karp in 1972 many other important problems in CS & OR are also NP-complete.

Recap: Here are the basic ideas we have so far:

→ formal definition of problems
  - functions, decision
  - language = set of strings.

→ Complexity class = set of languages

→ efficient computation – efficient computation on a Turing Machine.
  - single-tape, multi-tape.
  - diagonalization technique
    → HALT is undecidable

→ Time and Space classes.

→ Reductions.

→ C-completeness, C-hardness.