Lecture 3

21 JAN 2014

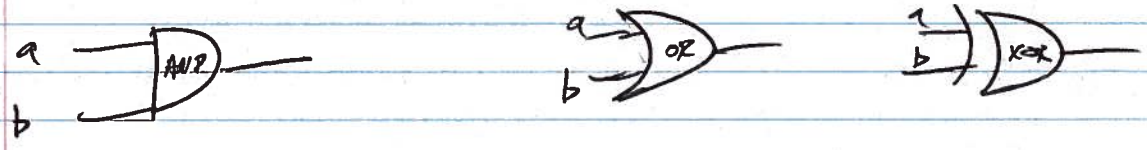
Circuit Model of Computation

- an alternative model that is equivalent to the Turing machine model but more convenient & realistic for many applications.

- a circuit is a directed, acyclic graph (no cycles!)
  to avoid instabilities...

consisting of wires & logic gates

Examples: NOT, AND, OR, XOR, NAND, & NOR



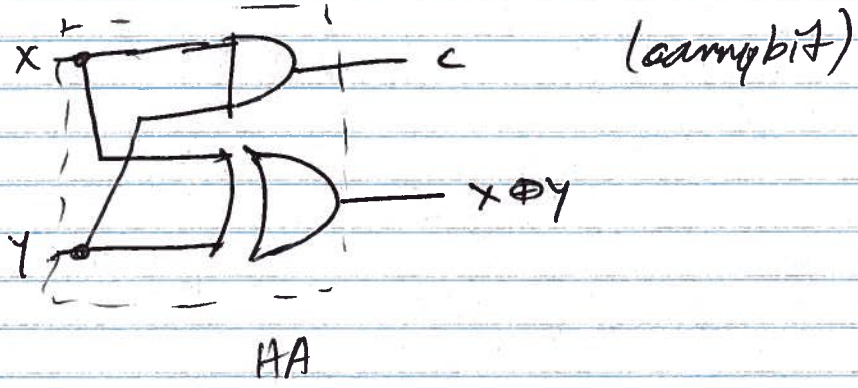
two other gates: FANOUT (copy) & SWAP

Also: ancilla preparation

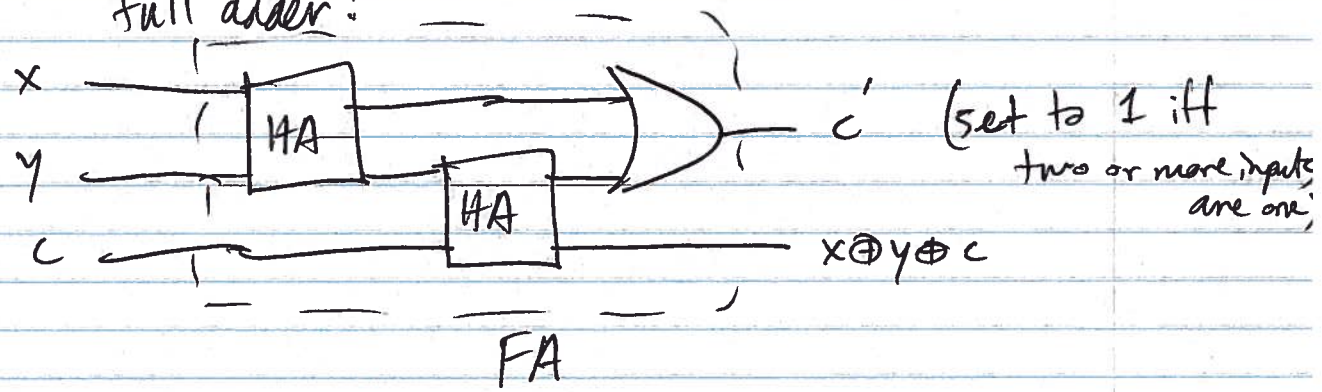
can use these elements to calculate any function whatsoever

Simple example of a circuit: adder

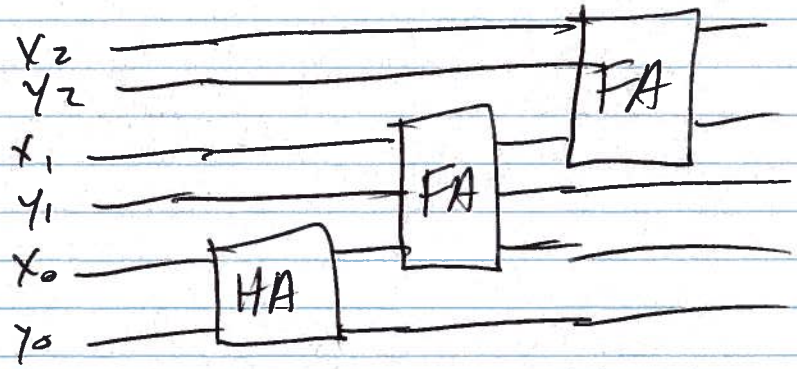
first need a half-adder



full adder:



can then add in binary



observe: to add two n-bit integers, there exists a circuit to do so which depends only on

These circuit elements are universal for classical computation. Putting together a few fixed gates, we can compute any function $f: \{0,1\}^n \to \{0,1\}^m$

suffices to show for $f: \{0,1\}^n \to \{0,1\}$

Prove by induction on n:

For $n=1$, only four functions identity, not (bit flip), constant 0 (AND w/ 0), constant 1 (OR w/ 1)

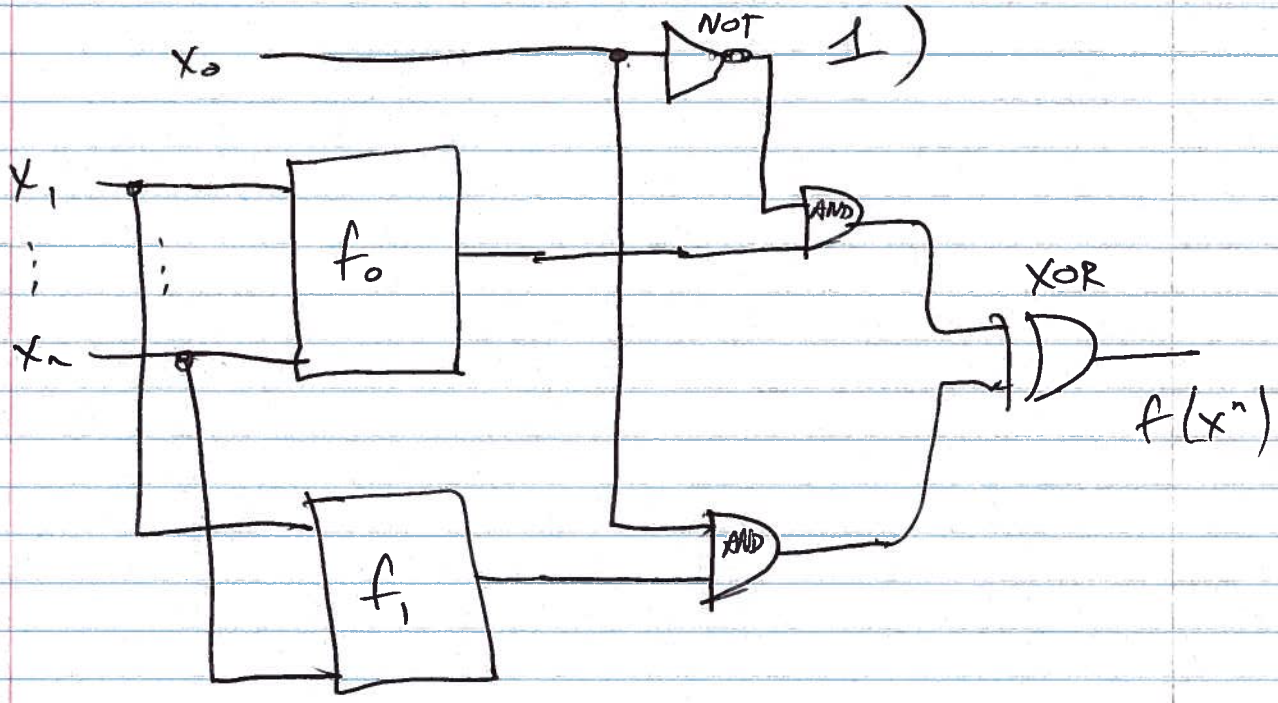
For inductive step, suppose that any function on n bits can be computed by a circuit, & let f be a function on $n+1$ bits. Define

$f_0(x_1, \ldots, x_n) \equiv f(0, x_1, \ldots, x_n)$ &

$f_1(x_1, \ldots, x_n) \equiv f(1, x_1, \ldots, x_n)$

By hypothesis $\exists$ circuits for these since they are on n bits.

Then here is a circuit to compute $f(x_0, x_1, \ldots, x_n)$ (just check whether the first bit is 0 or 1)



if $x_0 = 0$, don't use f_1

if $x_0 = 1$, don't use f_0

Five elements used:
1) wires
2) ancilla bits (used for n=1 case)
3) FANOUT or COPY
4) AND, XOR, & NOT gate
5) implicit SWAP or CROSSOVER

need to somehow extend these ideas to quantum case

Given that circuits can compute any function whatsoever, how are circuit model & TM model connected?

In order to build a circuit, we expect that there should be an algorithm for doing so. Indeed, since a circuit can compute any function, there is a circuit that can compute the halting function.
Can let $h_n(x)$ be the halting function from n bits to 1 bit. Then there exists a circuit C_n that can compute it. To prevent this unreasonable possibility, we have a notion of a uniform circuit family

Circuit family: collection $\{C_n\}$ of circuits

1) has n input bits & finite number of extra "work" bits & output bits
2) consistency: when $m < n$ & $|x| = n$ then $C_m(x) = C_n(x)$

~~polytime~~ uniform family if $\exists$ an algorithm running on a Turing machine which generates a description of circuit C_n given input n. (I.e., it outputs desc. of gates in circuit, how they are connected, ancilla bits, & which is the output.)

Idea is to think of this as the means by which an engineer (or computer) generates the circuit. (think of adding circuit)

can show that functions which are computable in TM model are the same as those which are computable in uniform circuit family model.

Want to divide decision problems into those that are easy to solve & those that are hard to solve. Would like a way to do so that is independent of a machine-specific implementation.

(Going from one-tape to two-tape Turing machine can reduce the time it takes to solve a problem.)

- Need some way of coarse graining to remove this dependence.

Use asymptotic notation to capture the essential behavior of a function as n gets large. want to know whether it scales like n or n^2 or 2^n, etc.

- say that $f(n) = O(g(n))$ if $\exists$ a positive constant c of positive integer n_0 such that $f(n) \leq cg(n) \quad \forall n \geq n_0$

Useful for studying worst-case behavior of algorithms.

- would also like to study the behavior of a class of algorithms (say for addition) in order to establish lower bounds on resource requirements.

Say that $f(n) \in \Omega(g(n))$ if $\exists$ c & n_0 such that

$cg(n) \leq f(n) \quad \forall n \geq n_0$

$f(n) = \Theta(g(n))$ if $f = O(g)$ &

Aside: we also say

$f(n) = o(g(n))$ if $\forall \varepsilon > 0$ $\exists$ n_0 such that $f(n) \leq \varepsilon \cdot g(n)$ $\forall n \geq n_0$.

$f(n) = \omega(g(n))$ if $\forall \varepsilon > 0$ $\exists n_0$.

" " $f(n) \geq \varepsilon g(n)$ $\forall n \geq n_0$.

Computational complexity is the study of time & space requirements needed to solve a given problem — one of its goals is to prove lower bounds on resources needed by best possible algorithm. (can view as a scaled down & more realistic version of computability theory.)

make distinction between polynomial & exponential resources.

1st objection is that this distinction may not be good when we're talking about

n^{10000} vs. $(1.000001)^n$

this never occurs in practice & first step in theory work is to find & apply...

good example of this is linear programming or semidefinite programming

(first algorithm not known to be in P + later improvements showed a poly-time alg.)

motivates Strong Church-Turing thesis:

Any model of computation can be simulated on a probabilistic Turing machine w/ at most a polynomial increase in the # of elementary operations required.

of course, q. computers cast doubt on this if they can be built

---

Complexity classes

Language is a division of $\{0,1\}^* = A_{yes} \cup A_{no}$ where $A_{yes} \cap A_{no} = \phi$

L = A_{yes}, L^c = A_{no}

Say that a TM decides a language L if it accepts when $x \in L$ & rejects when $x \notin L$

say that a problem is in $DTIME(f(n))$ if $\exists$ a deterministic Turing machine which decides whether $x \in L$ or $x \notin L$ in time $O(f(n))$ where $n = |x|$.

problem is ~~solvable~~ decidable in poly-time if it is in $DTIME(n^k)$ for some $k \geq 1$.

so $$P = \bigcup_{k \in \mathbb{N}} DTIME(n^k)$$

(supposed to capture class of problems for which finding a solution is easy)

We also say a TM is a poly time TM if its running time is ~~[illegible]~~ no more than $O(n^k)$ for some k where $n = |x|$.

can also define $$EXP = \bigcup_{k \in \mathbb{N}} DTIME(2^{n^k})$$

can prove that $P \neq EXP$ by a scaled down version of the halting problem.

NP is another class consisting of problems for which it is easy to check whether a solution is valid.

We say a language $L \in NP$ if $\exists$ a poly time deterministic TM such that

if input $x \in L$, then $\exists$ a witness string w such that $M(w,x) = 1$ (accept) with $|w| \leq p(|x|)$ (completeness condition)

If input $x \notin L$, then $\forall$ witness strings w, $M(w,x) = 0$ reject (soundness condition)

asymmetry in definition

"True statements have proofs. False statements don't."

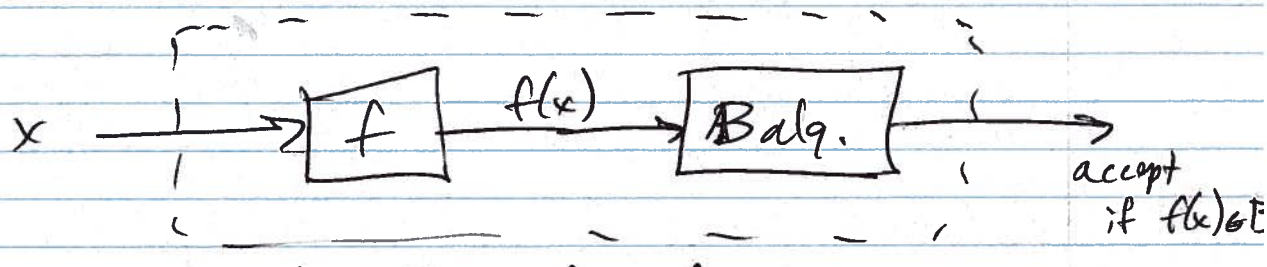
captures intuitive notion of what we do when checking a mathematical proof. Proof w is given & then we check in polynomial time whether it is a legitimate proof.

Simple problem in P: factoring — witness is the factorization of integer & verifier is the multiplying operation

problem is "hard" for a complexity class if any problem in the class can be reduced to it in polynomial time.

More formally, a language $A \subseteq \{0,1\}^*$ is poly-time Karp reducible to a language B if $\exists$ a poly-time computable function f such that $\forall\ x \in \{0,1\}^*$, $x \in A$ iff $f(x) \in B$.

Intuitively, you imagine an algorithm exists for deciding B. If so, then the following is an algorithm for deciding A (w/ only poly-overhead)



Algorithm for A

Idea inspired by computability theory of Turing. (show other problems are undecidable by reducing the halting problem to

problem A is complete for a class C if

$A \in C$ + A is C-hard.

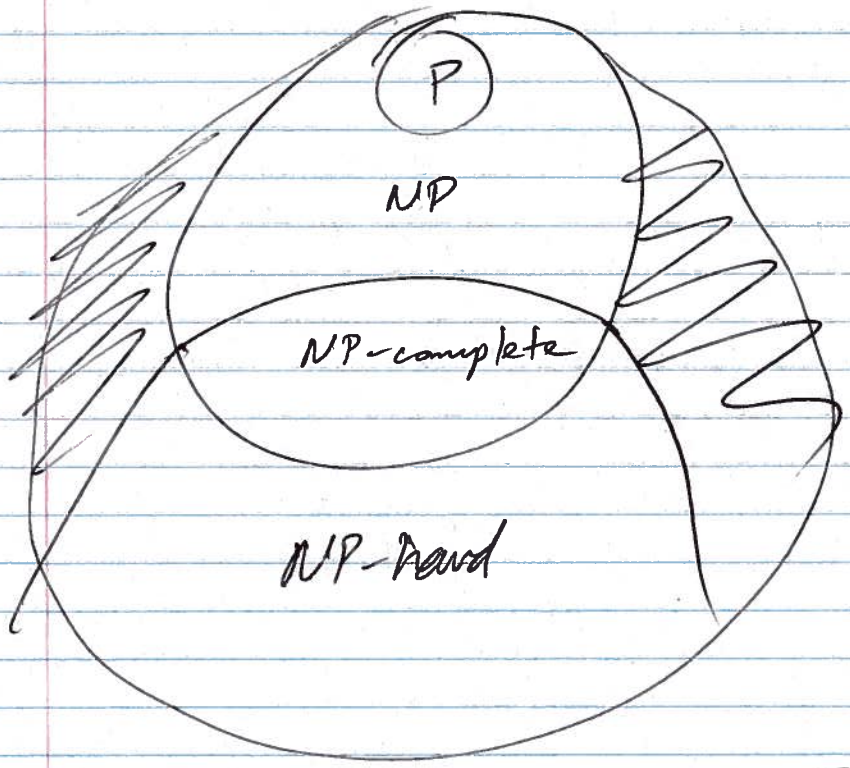
we can say

So there is a class of problems

NP-complete consisting of those problems which are in NP & as hard as any other problem in NP.

⇒ If any NP-complete is in P, then P=NP.

If P≠NP, then no NP-complete problem is in P



PSPACE class of problems decidable in poly-space

$P \subseteq NP \subseteq PSPACE \subseteq EXP$

↑ try all witnesses

only polynomial # of memory configurations → can simulate all of these

$P \neq EXP \Rightarrow$ one of the containments must be strict (believe all of them are)