15-451 Algorithms, Spring 2007

Lectures 19-26

Author: Avrim Blum
Instructor: Manuel Blum

March 27, 2007

* Linear programming

==========

In last couple of classes we looked at:

- Bipartite matching: Given a bipartite graph, find the largest set of edges with no endpoints in common.

- Network flow. (More general than bipartite matching)

- Min-Cost Max-flow (even more general than plain max flow).

Today, we'll look at something even more general that we can do algorithmically: LINEAR PROGRAMMING. (Except we won't necessarily be able to get integer solutions, even when specification of the problem is integral).

Linear Programming is important because it is so expressive: many many problems can be coded up as linear programs. Especially problems of allocating resources and a lot of business applications. In GSIA/Tepper there are entire courses devoted to linear programming. We're only going to have time for 1 lecture. So, will just have time to say what they are, and give examples of encoding problems as LPs. We will only say a tiny bit about algorithms for solving them.

Before defining, motivate with an example:

There are 168 hours in a week. Say we want to allocate our time between classes and studying (S), fun activities, relaxing, going to parties (P), and everything else (E) (eating, sleeping, taking showers, etc). To survive need to spend at least 56 hours on E (8 hrs/day). To maintain sanity need P+E >= 70. To pass courses, need S >= 60, but more if don't sleep enough or spend too much time partying: 2S+E-3P >= 150. (e.g., if don't go to parties at all then this isn't a problem, but if we spend more time on P then need to sleep more or study more).

Q1: can we do this? Formally, is there a *feasible* solution?

A: yes. For instance, one feasible solution: S=80, P=20, E = 68

Q2: suppose our notion of happiness is expressed by 2P+E. What is a feasible solution such that this is maximized? The formula "2P+E" is called an *objective function*.

This is called a *linear program*. What makes it linear is that all our constraints were linear in our variables. E.g., 2S+E-3P>=150. And our objective function is also linear. Not allowed things like requiring S*E >= 100, since this wouldn't be linear.

More formally, here is the definition of the linear programming problem LINEAR PROGRAMMING:

==========

* Given n variables $x_1, \ldots, x_n$.
* Given m linear inequalities in these variables (equalities OK too):

E.g., $3x_1 + 4x_2 <= 6$, $0 <= x_1 <= 3$, etc.

* May also have an objective function: $2x_1 + 3x_2 + x_3$.

* Goal: find values for the x_i's that satisfy the constraints and maximizes the objective.

"feasibility problem": no objective, just want to satisfy the constraints.

For instance, let's write out our time allocation problem this way.
Variables are S, P, E.
Objective is to maximize 2P+E subject to these constraints:

```
        S + P + E = 168
        E >= 56
        S >= 60
        2S+E-3P >= 150
        P+E >= 70
        P >= 0    (can't spend negative time partying)
```

MORE EXAMPLES OF MODELLING PROBLEMS AS LINEAR PROGRAMS:
==========

Typical Operations-Research kind of problem (from Mike Trick's course notes):
Have 4 production plants for making cars. Each works a little differently in terms of labor needed, materials, and pollution produced per car:

	labor	materials	pollution
plant 1	2	3	15
plant 2	3	4	10
plant 3	4	5	9
plant 4	5	6	7

We need to produce at least 400 cars at plant 3 according to labor agreement. Have 3300 hours of labor, 4000 units of material available. Allowed to produce 12000 units of pollution. Want to maximize number of cars produced. How to model?

First step: What are the variables?
x_1,x_2,x_3,x_4, where x_i = # cars at plant i.

Second step: What is our objective?
maximize $x_1+x_2+x_3+x_4$

Last step: what are the constraints?

```
x_i >= 0 for all i
x_3 >= 400
2x_1 + 3x_2 + 4x_3 + 5x_4 <= 3300
3x_1 + 4x_2 + 5x_3 + 6x_4 <= 4000
15x_1 + 10x_2 + 9x_3 + 7x_4 <= 12000
```

MAX FLOW
========
Can model this as LP too.

Variables: set up one variable x_e per edge e. Let's just represent

the positive flow.

Objective: maximize $\sum_{\text{edges } e \text{ into } T} x_e$

Constraints:

For all e, $x_e \le c_e$ and $x_e \ge 0$. [c_e is capacity of edge e]

For each node v except S and T,

$$\sum_{\text{edges } e \text{ into } v} x_e = \sum_{\text{edges } e' \text{ leaving } v} x_{e'}$$

Also, delete edges exiting T, or else add the constraint that no flow is allowed to be on them. Otherwise, the solution can fool the objective function by having flow looping through T....

Example:

S-->A, cap = 4. A-->C, cap=3. C-->T, cap = 2.
S-->B, cap = 2. B-->D, cap=3. D-->T, cap = 4.
C-->B, cap = 1. B-->C, cap=2.

LP is: maximize $x_{ct} + x_{dt}$ subject to

$0 \le x_{sa} \le 4$, $0 \le x_{ac} \le 3$, etc.
$x_{sa} = x_{ac}$,
$x_{sb} + x_{cb} = x_{bc} + x_{bd}$,
$x_{ac} + x_{bc} = x_{cb} + x_{ct}$,
$x_{bd} = x_{dt}$

How about min cost max flow? First solve for max flow f. Then add a constraint that flow must equal f, and minimize linear cost function $\sum_e \text{cost}(e) * x_e$. Or, you can do both together by adding an edge of infinite capacity and very negative cost from t to s, and then just minimizing cost (which will automatically maximize flow).

2-PLAYER ZERO-SUM GAMES:

Remember back to hwk1, where we looked at 2-player zero-sum games. Had Alice hiding a nickel or quarter and Bob guessing, and then based on whether the guess was right or wrong, some money changed hands. This is called a "zero-sum game" because no money is entering or leaving the system (it's all going between Alice and Bob).

A MINIMAX OPTIMAL strategy for a player is a (possibly randomized) strategy with the best guarantee on its expected gain --- i.e., one you would want to play if you imagine that your opponent knows you well.

Another game: shooting a penalty kick against a goalie who is a bit weaker on one side. Kicker can kick left or right. Goalie can dive left or right. Here is payoff matrix for kicker (chance of getting goal):

		goalie	
		left	right
	left	0	1

```
      kicker
             right     1     1/2
```

minimax optimal strategy for kicker: 1/3 chance kick left, 2/3 chance kick right. Expected gain = 2/3.

How about solving an nxn game?

```
            20    -10   5

            5     10    -10

            -5    0     10
```

How can we compute minimax optimal strategy (say, for row)?

Variables: the things we want to figure out are the probabilities we should put on our different choices. Those are our variables: p_1,...,p_n

Need to be legal prob dist: for all i, p_i >= 0. p_1 + ... + p_n = 1.

Want to maximize the worst case (minimum), over all columns opponent can play, of our expected gain. This is a little confusing because we are miximizing a minimum. So, add one new variable v (representing the minimum) and put in constraints that our expected gain has to be at least this, then maximize on v. So this looks like:

```
            maximize v such that:
```

p_i are a prob dist: (see constraints above).
for each column j, p_1*M[1][j] + p_2*M[2][j] + ... + p_n*M[n][j] >= v.

E.g., in example above we want:

```
get at least v if col1:   20*p_1 + 5*p_2 -5*p_3 >= v
.............. if col2:  -10*p_1 + 10*p_2 >= v
.............. if col3:    5*p_1 - 10p_2 + 10p_3 >= v
```

HOW TO SOLVE

How to solve linear programs? History: the standard algorithm for solving LPs the Simplex Algorithm, developed in the 40s. It's *not* guaranteed to run in polynomial time, and you can come up with bad examples for it, but in general it runs pretty fast. Only much later in 1980 was it shown that linear programming could be done in polynomial time by something called the Ellipsoid Algorithm (but it is pretty slow). Later on, a faster polynomial-time algorithm Karmarkar's Alg was developed, which is competitive with Simplex. There are a lot of commercial LP packages, for instance LINDO, CPLEX, Solver (in Excel).

Won't have time to describe any of these in detail. Instead, give some intuition by viewing LP as a geometrical problem.

Think of an n-dimensional space with one coordinate per variable. A

solution is a point in this space. An inequality, like x_1 + x_2 <= 6 is saying that we need solution to be on a specified side of a certain hyperplane. Feasible region is the convex region in space defined by these constraints. Then we want to find the feasible point that is farthest in the "objective" direction.

Let's go to first example with S, P, E. To make this easier to draw, can use our first constraint that S+P+E = 168 to replace S with 168-P-E. So, just draw in 2 dimensions: P and E.

Constraints are:

E >= 56
P+E >= 70
P >= 0
S >= 60 which means 168-P-E >= 60 or P+E <= 108.
2S-3P+E >= 150 which means 2(168-P-E)-3P+E >= 150 or 5P+E <= 186

For objective of max P, this happens at E=56, P = 26.
For objective of max 2P+E, this happens at P=19.5, E=88.5

Can use this view to motivate the algorithms.

SIMPLEX ALG: Earliest and most common algorithm called Simplex method. Idea: start at some "corner" of the feasible region (to make this easier, we can add in "slack variables" that will drop out when we do our optimization). Then we repeatedly do the following step: look at all neighboring corners and go to the best one if it is better. Stop when we get to a corner where no neighbor is better than we are. Neat fact is that since the objective is linear, the optimal solution will be at a corner (or maybe multiple corners). Furthermore, there are no local maxima: if you're *not* optimal then some neighbor of you is better than you are. That's because this is a convex region. So, Simplex method is guaranteed to halt at the best solution. The problem is that it is possible for there to be an exponential number of corners and it is possible for Simplex to take an exponential number of steps to converge. But, in practice this usually works well.

ELLIPSOID ALGORITHM: Invented by Khachiyan in 1980 in Russia.

Solve "feasibility problem" (Then can do binary search with our objective function). Start with big ellipse (called an ellipsoid in higher dimensions) that we can be sure contains the feasible region. Then try the center of the ellipse -- see if it violates any constraints. If not, you're done. If so, look at the constraint violated. So we know the solution (if any) is contained in the remaining at-most-half-ellipse. Now, find a new smaller ellipse that contains that portion of our initial ellipse. Then repeat. Can show that in each step, the new smaller ellipse has a volume that's significantly smaller -- each ellipse has volume at most (1 - 1/n) of the previous one. If ever get to too small volume can prove there can't be a solution.

One nice thing about ellipsoid algorithm is you just need to tell if the current solution violates any constraints or not, and if so, to produce one. Don't need to explicitly write them all down.

KARMARKAR'S ALGORITHM: Sort of has aspects of both. Works with feasible points but doesn't go from corner to corner. Instead it moves inside the interior of the feasible region. One of first of a whole class of so-called "interior-point methods".

Development of better and better algorithms is a big ongoing area of research. In particular, get a lot of mileage by using good data structures to speed up time in making each decision.

* NP-completeness and expressiveness
* informal definitions
* formal definitions
* Circuit-SAT and 3-SAT

==========

In the last few classes, we've looked at increasingly more expressive problems: network flow, min cost max flow, linear programming.

These problems have the property that you can code up a lot of different problems in their "language". So, by solving these well, we end up having some important hammers we can use to solve other problems.

In fact, to talk about this a little more precisely, let's make the following definitions:

* We'll say that an algorithm runs in Polynomial Time if for some constant c, its running time is O(n^c), where n is the size of the input. "Size of input" means "number of bits it takes to write the input down"

* Problem A is poly-time REDUCIBLE to problem B (A <=_p B) if given a poly-time alg for B, we can use it to produce a poly-time alg for A. Problem A is poly-time EQUIVALENT to problem B (A =_p B) if A <=_p B and B <=_p A.

For instance, we showed BIPARTITE MATCHING <=_p MAX FLOW
MIN-COST MAX-FLOW <=_p LINEAR PROGRAMMING
and more.

==========

Let's start today by thinking about what would be a *really* expressive problem, such that if we could solve it we could do all sorts of things. Here is a natural candidate:

The SHORT SOLUTION EXISTENCE PROBLEM: Given an algorithm V(I,X) [written in some standard programming language, and think of it as outputting either YES or NO], and given I and a bound B written in unary (B bits). Question: does there exist an input X such that V(I,X) halts in at most B steps and outputs YES?

Why am I calling this the "short solution existence problem"? Consider some problem we might want to solve like the TRAVELING-SALESMAN-PROBLEM: given a graph G and an integer k, is there a tour that visits all the nodes of G and has total length <= k? We don't know any fast ways of solving that problem, but we can easily write a program V that given I = (G,k) and given a proposed solution X verifies whether X indeed visits all the nodes of G and has total length <= k. Furthermore, this solution-verifier is linear time. So, if we could solve the "short solution existence problem", we could tell if there is a X that makes V output YES and thereby solve the TSP.

What if we actually wanted to find the tour? One way we could solve that would be to start deleting edges from G and then re-running the above procedure. If the answer is "NO" then we put the edge back in. If we do this for all edges, what's left in G will be just the tour itself.

Let's try another problem. Say we wanted to factor a big integer N. We don't know any polynomial-time algorithms for solving that problem. But, we can easily write a verifier that given N and a proposed factor X, tells us if X is a solution to the problem.

In fact, let's modify this slightly (you'll see why in a second) so the verifier takes in an additional integer k (so I = (N,k)) and outputs YES if X divides N *and* 1 < X < k.

So, if we can solve the short-solution-existence-problem, we can tell if N has a factor between 2 and k-1 by feeding V and I=(N,k) into our solver. Then if we want to actually *find* a factor, we can do binary search on k. (That's why we needed the k).

In fact, we could use an algorithm for the "short solution existence problem" to solve *any* problem for which we have a polynomial-time algorithm for simply *checking* if a proposed solution is correct: we just write down V and then make the length bound B big enough to handle the running time of V.

Interestingly, the short-solution-existence-problem also belongs to this same category. Namely if someone hands us a proposed solution X, we can check it by just running V.

WHAT WE NOW HAVE
================

This class of problems - problems for which we can efficiently verify a proposed solution - is called NP. A problem Q is said to be NP-complete if (a) Q is in NP and (2) you could use a polynomial-time algorithm for Q to solve *any* problem in NP in polynomial time. That is, for any Q' in NP, Q' <=_p Q.

And we have just proven our first NP-complete problem, namely the SSEP.

Now this problem seems pretty stylized. But we can now show that other simpler-looking problems have the property that if you could solve them in polynomial-time, then you could solve this problem in polynomial time, so they too are NP-complete.

So, the way to think of it is an NP-complete problem is super-expressive. It is so expressive, we believe there are no polynomial-time algorithms for solving them. In particular, if we *could* solve an NP-complete problem in polynomial-time, then it would mean that for any problem where we could *check* a proposed solution efficiently, we could also *find* such a solution efficiently.

==========
Now, onto formal definitions.

We will formally be considering decision problems: problems whose answer is YES or NO.

COMPLEXITY CLASSES: P and NP.

P: problems solvable in polynomial time. E.g., Given a network G and a flow value f, does there exist a flow >= f?

NP: problems that have polynomial-time verifiers. Specificially, problem Q is in NP if there is a polynomial-time algorithm V(I,X) such that:

  If "I" is a YES-instance, then there exists X such that V(I,X) = 1.
  If "I" is a NO-instance, then for all X, V(I,X) = 0.

Furthermore, X should have length polynomial in size of I (since we are really only giving V time polynomial in the size of the instance).

"X" is often called a "witness". E.g., consider 3-coloring. The witness that an answer is YES is the coloring. The verifier just checks that all edges are correct and that at most 3 colors are used. So, 3-coloring is in NP.

NP is like: "I might not know how to find it, but I'll know it when I see it" (BTW, Co-NP is the class of problems that work the other way around).

It's pretty clear that P is contained in NP. Huge open question in complexity theory is P=NP? It would be really weird if they are equal so most people believe P != NP. But, it's very hard to prove that a fast algorithm for something does NOT exist. So, it's still an open problem.

NP-Completeness
===============
Problem Q is NP-complete if:
 (1) Q is in NP, and
 (2) Any other problem Q' in NP is polynomial-time reducible to Q.

So if you could solve Q in polynomial time, you could solve *any* problem in NP in polynomial time. If Q just satisfies (2) then it's called NP-hard.

Here's another NP-complete problem:

  CIRCUIT-SAT: Given a circuit of NAND gates with a single output and no loops (some of the inputs may be hardwired). Question: is there a setting of the inputs that causes the circuit to output 1?

Proof Sketch:

First of all, it is clearly in NP, since you can just guess the input and try it. To show it is NP-complete, we just need to show that if we could solve this, then we could solve the short-solution-existence-problem. Here's how. Say we are given V, I, B, and want to tell if there exists X such that V(I,X) outputs YES in at most than B steps. What we can do (and this part is a little handwavy but in principle you did this in 251 when you built a computer out of NAND gates) is we can effectively compile V into a circuit of depth polynomial in B and the number of bits in I that mimics what V does in it first B time steps. We then hardwire the inputs corresponding to I and feed this into our circuit-SAT solver.

OK, fine so we now have one more NP-complete problem. But CIRCUIT-SAT looks really complicated. We weren't expecting to be able to solve it. But *now* we can show that a much simpler looking problem has the

property that if you could solve it efficiently, then you could solve CIRCUIT-SAT efficiently. ("efficiently" = "polynomial time").

This problem is 3-SAT.

3-SAT:
  Given: a CNF formula (AND of ORs) over n variables x1,...,xn, where each clause has at most 3 variables in it.
     (x1 OR x2 OR not(x3)) AND (not(x2) OR x3) AND (x1 OR x3) AND ...

  Goal: find an assignment to the variables that satisfies the formula if one exists.

Claim: we can reduce solving CIRCUIT-SAT to solving 3-SAT. I.e., if we can solve 3-SAT then we can solve CIRCUIT-SAT and so we can solve all of NP.

We'll do the reduction next time, but before we end, here is formally how we are going to do reductions:

Say we have some problem A that we know is NP-complete. We want to show problem B is NP-complete too. Well, first we show B is actually in NP but that is usually the easy part. The main thing we need to do is show that any polynomial-time algorithm for B would give a polynomial-time algorithm for A. We do this by "reducing A to B", and in particular what we want is this:

Reducing problem A to problem B
==========
To reduce problem A to problem B we want a function f that takes instances of A to instances of B such that:
(1) if x is a yes-instance of A then f(x) is a yes-instance of B
(2) if x is a no-instance of A then f(x) is a no-instance of B
(3) f can be computed in polynomial time.

So, if we had an algorithm for B, we could using it to solve A by running it on f(x).

NP-completeness cont
* Recap, Formal definitions.
* reducing CIRCUIT-SAT to 3-SAT [proving 3-SAT is NP-complete]
* reducing 3-SAT to CLIQUE [proving CLIQUE is NP-complete]
* Independent Set, Vertex Cover

==========

In the discussion below, a "problem" means something like 3-coloring or network flow, and an "instance" means a specific instance of that problem --- the graph to color or the network to find a flow on.

A DECISION PROBLEM is just a problem where each instance is either a YES-instance or a NO-instance and the goal is to decide which type your instance is. (e.g., for 3-coloring, G is a YES-instance if it has a 3-coloring and NO if not. For the perfect-matching problem, G is a YES-instance if it has one and a NO-instance if it does not.)

P: class of decision problems Q that have polynomial-time algorithms: A(I)=YES iff I is a YES-instance of Q.

NP: decision problems where at least the YES-instances have short proofs (that can be checked in polynomial-time) that the answer is YES. Q is in NP if there is a verifier V(I,X) such that:

If "I" is a YES-instance, then there exists X such that V(I,X) = YES.
If "I" is a NO-instance, then for all X, V(I,X) = NO.

and furthermore the length of X and the running time of V are poly in |I|.

co-NP: vice-versa: there are short proofs for NO-instances. (E.g., given two circuits, C_1, C_2, do they compute the same function?).

Problem Q is NP-complete if:
(1) Q is in NP, and
(2) Any other Q' in NP is "polynomial-time reducible" to Q. In particular, for any Q' in NP, there is some function f from instances of Q' to instances of Q, that maps YES-instances of Q' to YES-instances of Q, and maps NO-instances of Q' to NO-instances of Q.

If Q just satisfies (2) then it's called NP-hard.

Last time we showed that the following problem is NP-complete:

CIRCUIT-SAT: Given a circuit of NAND gates with a single output and no loops (some of the inputs may be hardwired). Question: is there a setting of the inputs that causes the circuit to output 1?

Proof sketch: It's clearly in NP, since V(I,X) just runs circuit I on proposed solution X and checks that the circuit outputs 1. Now, say Q' is some other problem in NP. Q' must have some polynomial-time verifier V'. Now, given some instance I of Q', what function f does is construct a circuit C_I such that C_I(X) = V'(I,X). So, f is like a compiler that compiles V' into a circuit of NAND gates. Claim is that we can do this by unrolling a polynomial number of loops of the kind of NAND-gate computer you built in 15-251. Finally, by design, C_I is a YES-instance of CIRCUIT-SAT iff I was a YES-instance of Q'. So, if we can solve CIRCUIT-SAT then we can solve Q'.

Aside: we could define the SEARCH-version of a problem in NP as: "..and furthermore, if I is a YES-instance, then *produce* X such that V(I,X)=YES". If we can solve any NP-complete decision problem in polynomial time then we can actually solve search-version of any problem in NP in polynomial-time too. Will talk about in recitation.

Now CIRCUIT-SAT is a little unweildy. What's REALLY INTERESTING about NP-completeness is not just that such problems exist, but that a lot of very innocuous-looking problems are NP-complete. To show results like that, we will first reduce CIRCUIT-SAT to a much simpler-looking problem called 3-SAT.

3-SAT:
  Given: a CNF formula (AND of ORs) over n variables x1,...,xn, where each clause has at most 3 variables in it.
     (x1 OR x2 OR not(x3)) AND (not(x2) OR x3) AND (x1 OR x3) AND ...

  Goal: find an assignment to the variables that satisfies the formula if one exists.

THEOREM: 3-SAT is NP-Complete

Proof: We need to define a function f that converts instances C of CIRCUIT-SAT to instances of 3-SAT such that the formula produced is satisfiable iff the circuit C had an input x such that C(x)=1.

First of all, let's assume our input is given as a list of gates, where for each gate we are told what its inputs are connected to. E.g., g1 = x1 NAND x3; g2 = g1 NAND x4; g3 = x1 NAND 1; g4 = g1 NAND g2; ... plus we are told which gate gm is the output of the circuit.

We will now compile this into an instance of 3-SAT as follows. We'll make one variable for each input xi of the circuit, and one for every gate gi. We now write each NAND as a conjunction of 4 clauses. In particular, we just replace the statement "y3 = y1 NAND y2" with:

```
     (y1 OR y2 OR y3)      // if y1=0 and y2=0 then we must have y3=1
 AND (y1 OR not(y2) OR y3) // if y1=0 and y2=1 then we must have y3=1
 AND (not(y1) OR y2 OR y3) // if y1=1 and y2=0 then we must have y3=1
 AND (not(y1) OR not(y2) OR not(y3)) //if y1=1,y2=1 we must have y3=0
```

Finally, we add the clause (gm). This forces the circuit to output 1. In other words, we are asking: is there an input and a setting of all the gates such that the output of the circuit is equal to 1, and each gate is doing what it's supposed to? So, the 3-CNF formula produced is satisfiable if and only if the circuit has a setting of inputs that causes it to output 1. The size of the formula is linear in the size of the circuit. The construction can be done in polynomial (actually, linear) time. So, if we had a polynomial-time algorithm to solve 3-SAT, then we could solve circuit-SAT in poly time too.

==========
IMPORTANT FACT: now that we have 3-SAT, in order to prove some other NP problem Q is NP-complete, we just need to show 3-SAT <= Q: if we could solve Q then we could solve 3-SAT. MAKE SURE YOU UNDERSTAND THIS - A LOT OF PEOPLE MAKE THE MISTAKE OF DOING IT THE OTHER WAY

AROUND.

MAX-CLIQUE: given a graph G, find the largest clique (set of nodes s.t. all pairs in the set are neighbors). Decision problem: "Given G and integer k, is there a clique of size >= k?".
(MAX-CLIQUE is clearly in NP.)

Theorem: Max-Clique is NP-Complete.

Proof: reduce 3-SAT to MAX-CLIQUE.
Given a 3-CNF formula F of m clauses over n variables, we construct a graph as follows. For each clause c of F we create one node for every assignment to variables in c that satisfies c. E.g., say

```
 F = (x1 OR x2 OR not(x4)) AND (not(x3) OR x4) AND (not(x2) OR not(x3)) AND ...
```

Then in this case we would create nodes like this:

```
      (x1=0,x2=0,x4=0) (x3=0,x4=0)  (x2=0,x3=0) ...
      (x1=0,x2=1,x4=0) (x3=0,x4=1)  (x2=0,x3=1)
      (x1=0,x2=1,x4=1) (x3=1,x4=1)  (x2=1,x3=0)
      ...
```

Then we put an edge between two nodes if the partial assignments are consistent. Note: max possible clique size is m. And, if the 3-SAT problem does have a satisfying assignment, then in fact there IS an m-clique. Claim is this is true in the other direction too. If the graph has an m-clique, then there is a satisfying assignment: namely, just read off the assignment given in the nodes of the clique. So, this graph has a clique of size m iff F was satisfiable. Also, our reduction is poly time since the total size of graph is at most quadratic in size of formula (O(m) nodes, O(m^2) edges). Therefore Max-Clique is NP-complete.

Independent Set
==========
An independent set in a graph is a set of nodes no two of which have an edge. E.g., in a 7-cycle, the largest independent set has size 3. (E.g, in the graph coloring problem, the set of nodes colored red is an independent set).

Theorem: Independent set (is there an Indep Set in G of size > k?) is NP-complete.

Proof: Reduce from clique. Given graph G for clique problem, just take complement of the graph . Ie. create graph H such that H has edge (u,v) iff G does NOT have edge (u,v). Then H has an indep set of size k iff G has a k-clique.

Vertex-Cover
==========
A vertex cover in a graph is a set of nodes such that every edge is incident to at least one. (e.g., look at cut-diamond graph). For instance, can think of as what rooms to put security guards in a museum. What we want is the smallest vertex cover.

Decision problem: does there exist a vertex cover of size < k?

Claim: if C is a vertex cover in a graph, then V-C is an independent set. Also if S is an independent set, then V-S is a vertex cover. So, to solve "is there an independent set of size > k?" just solve "is there a vertex cover of size < n-k?". So, Vertex cover is NP-Complete too.

* NP-completeness summary
* Approximation Algorithms

==========

NP-completeness summary:

- picture of P, NP, co-NP, PSPACE, turing-computable functions
  (PSPACE = problems solvable with polynomial amount of memory usage.
  Anything in NP is also in PSPACE since with polynomial space you can just have a counter that tries all possible proof strings, running each one into the verifier to see if any of them work.)

- NP-complete problems: in NP *and* capture essence of entire class in that a polynomial-time algorithm to solve one of them would let you solve anything in NP.

- Can talk about complete problems for other classes too like PSPACE.

- Prove 3-SAT NP-complete by reduction from Circuit-SAT. Show how to convert gate-by-gate. E.g., x3 = NAND(x1,x2). Write down truth-table.

- Show chain of reductions from factoring to 3-SAT. If you had a magic algorithm for 3-SAT, how could you use that to factor?

APPROXIMATION ALGORITHMS

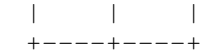
General Question: Maybe we can't hope for a fast algorithm that always gets the best solution, but can we at least guarantee to get a "pretty good" solution? E.g., can we guarantee to find a solution that's within 10% of optimal? Or how about within a factor of 2 of optimal? Or, anything non-trivial?

Interesting thing: even though all NP-complete problems are equivalent in terms of difficulty of finding optimal solution, the difficulty of getting a good approximation varies all over the map.

VERTEX COVER

- GIVEN: a graph G. GOAL: find the smallest set of vertices such that every edge is incident to (touches) at least one vertex in the set.

- Example:



- Can think of as: what is the fewest # of guards we need to place in museum to cover all the corridors.

- This problem is NP-hard. But it turns out that for any graph G we can get within a factor of 2.

Let's start with some strategies that *don't* work.

Strawman #1: Pick arbitrary vertex with at least one uncovered edge incident, put into cover, and repeat. What would be a bad example? [A: how about a star graph]

Strawman #2: how about picking the vertex that covers the *most* uncovered edges. Turns out this doesn't work either. [make bipartite graph where opt is size t, but this alg finds one of size t + floor(t/2) + floor(t/3) + floor(t/4) + ... + 1. This is Theta(t log t). Best examples are with t=6 or t=12.]

How to get factor of 2. Two algorithms:

Alg1: Pick arbitrary edge. We know any VC must have at least 1 endpt, so lets take both. Then throw out all edges covered and repeat. Keep going until no uncovered edges left. What we've found in the end is a matching (a set of edges no two of which share an endpoint) that's "maximal" (meaning that you can't add any more edges to it and keep it a matching). This means if we take all endpoints, we have a VC. So, if we picked k edges, our VC has size 2k. But, any VC must have size at least k since you need to have at least one endpoint of each edge (and, these edges don't touch, so these are k *different* vertices).

Here's another 2-approximation algorithm for Vertex Cover:

Alg2: Step1: Solve a *fractional* version of the problem. Have a variable x_i for each vertex. Constraint 0<= x_i <= 1. Think of x_i = 1 as picking the vertex, x_i = 0 as not picking it, and in-between as "partially picking it". Each edge should be covered in that for each edge (i,j) we want x_i+x_j >= 1. Then our goal is to minimize sum_i x_i. We can solve this using linear programming. This is called an "LP relaxation" because any true vertex cover is a feasible solution, but we've made the problem easier by allowing fractional solutions too. So, the value of the optimal solution now will be at least as good as the smallest vertex cover, maybe even better, but it just might not be legal any more.

E.g., triangle-graph. E.g., star-graph.

Step2: now pick each vertex i such that x_i >= 1/2.

Claim 1: this is a VC. Why? [get at least 1 endpt of each edge]

Claim 2: The size of this VC is at most twice the size of the optimal VC. Why? Let OPT_frac be the value of the optimal fractional solution, and OPT_VC be the size of the smallest vertex cover. First, as we noted above, OPT_frac <= OPT_VC. Second, our solution is at most 2*OPT_frac since it's no worse than doubling and rounding down. So, put together, our solution <= 2*OPT_VC.

Interesting fact: nobody knows any algorithm with approximation ratio 1.9. Best known is 2 - O(1/sqrt(log n)), which is 2 - o(1).

Current best hardness result: Hastad shows 7/6 is NP-hard. Improved to 1.361 by Dinur and Safra. Beating 2-epsilon has been related to some other open problems, but not known to be NP-hard.

SET-COVER

Set-cover:

Given a domain X of n points, and m subsets S_1, S_2, ..., S_m of these points. Goal: find the fewest number of these subsets needed to cover all the points.

Set-cover is NP-hard. However, there's a simple algorithm that gets a ratio of ln(n):

Greedy Algorithm: Pick the set that covers the most points. Throw out all the points covered. Repeat.

What's an example where this *doesn't* find the best solution?

Theorem: If the optimal solution uses k sets, the greedy algorithm finds a solution with at most k*ln(n) sets.

Proof: Since the optimal solution uses k sets, there must some set that covers at least a 1/k fraction of the points. Therefore, after the first iteration of the algorithm, there are at most n(1 - 1/k) points left. After the second, there are at most n(1 - 1/k)^2 points left, etc. After k rounds, there are at most n(1 - 1/k)^k < n*(1/e) points left. After k*ln(n) rounds, there are < n*(1/e)^{ln n} = 1 points left, which means we must be done.

In fact, it's been proven that unless everything in NP can be solved in time n^{O(loglog n)}, then you can't get better than (1-epsilon)*ln(n) [Feige].

MAX-SAT: Given a CNF formula (like in SAT), try to maximize the number of clauses satisfied.

To make things cleaner, let's assume we have reduced each clause [so, (x or x or y) would become just (x or y), and (x or not(x)) would be removed]

Claim: if every clause has size exactly 3 (this is sometimes called the MAX-exactly-3-SAT problem), then there is a simple randomized algorithm can satisfy at least a 7/8 fraction of clauses. So, this is for sure at least a 7/8-approximation.

Proof: Just try a random assignment to the variables. Each clause has a 7/8 chance of being satisfied. So if there are m clauses total, the expected number satisfied is (7/8)m. If the assignment satisfies less, just repeat. Since the number of clauses satisfied is bounded (it's an integer between 0 and m), with high probability it won't take too many tries before you do at least as well as the expectation.

How about a deterministic algorithm? Here's a nice way we can do that. First, let's generalize the above statement to talk about general CNF formulas.

Claim: Suppose we have a CNF formula of m clauses, with m_1 clauses of size 1, m_2 of size 2, etc. (m = m_1 + m_2 + ...). Then a random assignment satisfies sum_k m_k(1 - 1/2^k) clauses.

Proof: linearity of expectation.

Now, here is a deterministic algorithm : Look at x_1: for each of the two possible settings (0 or 1) we can calculate the expected number of clauses satisfied if we were to go with that setting, and then set the rest of the variables randomly. (It is just the number of clauses already satisfied plus sum_k m_k(1-1/2^k), where m_k is the number of clauses of size k in the ``formula to go''.) Fix x_1 to the setting that gives us a larger expectation. Now go on to x_2 and do the same thing, setting it to the value with the highest expectation-to-go, and then x_3 and so on. The point is that since we always pick the setting whose expectation-to-go is larger, this expectation-to-go never decreases (since our current expectation is the average of the ones we get by setting the next variable to 0 or 1).

This is called the ``conditional expectation'' method. The algorithm itself is completely deterministic --- in fact we could rewrite it to get rid of any hint of randomization by just viewing sum_k m_k(1-1/2^k) as a way of weighting the clauses to favor the small ones, but our motivation was based on the randomized method.

Interesting fact: getting a 7/8 + epsilon approximation for any constant epsilon (like .001) for MAX-exactly-3-SAT is NP-hard.

In general, the area of approximation algorithms and approximation hardness is a major area of algorithms research. Occupies a good fraction of major algorithms conferences.

15-451 Algorithms 4/10/07

* Online algorithms
  - rent-or-buy?
  - elevator problem
  - repeated play of matrix games Quiz Tues (up through last class)

==========

Today's topic: Online Algorithms

Last time: looked at algorithms for finding approximately-optimal solutions for NP-hard problems. Today: finding approximately-optimal solutions for problems where difficulty is that the algorithm doesn't have all the information up front.

Online algorithms: Algorithms for settings where inputs/data arriving over time. Need to make decisions on the fly, without knowing what will happen in the future. (As opposed to standard problems like sorting where you have all inputs at the start.) Data structures are one example of this. We'll talk about a few other examples today.

Rent-or-buy?

==========

Simple online problem that captures a common issue in online decision-making, called the rent-or-buy problem. Say you are just starting to go skiing. Can either rent skis for $50 or buy for $500. You don't know if you'll like it, so you decide to rent. Then you decide to go again, and again, and after a while you realize you have shelled out a lot of money renting and you wish you had bought right at the start. Optimal strategy is: if you know you're going to end up skiing more than 10 times, you should buy right at the beginning. If you know you're going to go < 10 times, you should just rent. But, what if you don't know?

To talk about quality of an online algorithm, we can look at what's called the "competitive ratio":

> Competitive ratio is worst case (maximum) over possible events of the ratio: (alg cost)/OPT, where OPT = optimal cost in hindsight.
>
> "cost" means total cost over all time.

E.g., what is CR of algorithm that says "buy right away"?
Worst case is: only go once. Ratio is 500/50 = 10.

What about algorithm that says "Rent forever"?
Worst case is: keep going skiing. Ratio is infinite.

Here's a nice strategy: rent until you realize you should have bought, then buy. (In our case: rent 9 times, then buy).

Case 1: If you went skiing 9 or less times, you're optimal.
Case 2: If you went 10+ times, you paid $450 + $500. Opt paid $500. Ratio = 2 - 1/10. In general, if purchase cost p is a multiple of rental cost r, the ratio is ((p-r)+p)/p = 2 - r/p.

Worst of these is case 2, so competitive ratio is 2- r/p.

Claim: above strategy has the best possible competitive ratio for

deterministic algorithms.

Proof: Consider the event that the day you purchase is the last day you go skiing. If you rent longer than the above strategy, then the numerator goes up but the denominator stays the same, so your ratio is worse. If you rent fewer times, then the numerator goes down by r but so does the denominator, so again the ratio is worse.

The elevator problem

You go up to the elevator and press the button. But who knows how long it's going to take to come, if ever? How long should you wait until you give up and take the stairs?

Say it takes time E to get to your floor by elevator (once it comes) and it takes time S by stairs. E.g, maybe E = 15 sec, S = 45 sec.

What strategy has the best competitive ratio?

- wait 30 sec, then take stairs. (in general, wait for S-E time) (I.e., take the stairs once you realize you should have taken them at the start)
- if elevator comes in < 30 sec, we're optimal.
- otherwise, OPT = 45. We took 30+45 sec, so ratio = (30+45)/45 = 5/3
- Or, in general, ratio = (S-E+S)/S = 2 - E/S.

This is really the same as rent-or-buy where stairs=buy, waiting for E time steps is like renting, and the elevator arriving is like the last time you ever ski.

Other problems like this: whether it's worth optimizing code, when your laptop should stop spinning the disk between accesses, and many others.

REPEATED PLAY OF MATRIX GAMES

We talked earlier in class about matrix games, and the notion of minimax optimal strategies, and how linear programming can be used to solve for them. But what if we are playing against an opponent who is not optimal? Can we find a way to take advantage? What we're going to look at now is an adaptive algorithm for repeatedly playing some matix game against an opponent, that is guaranteed to approach (or exceed) the performance of the best fixed strategy in hindsight given the series of plays of the opponent.

Given a series of rounds of us (row player) playing against some opponent (column player), define OPT to be performance of best fixed row in hindsight, given how the opponent played. Claim is we can get an algorithm whose performance will approach OPT. In game-theory terminology, we are doing nearly as well as the "best response to the opponent's empirical distribution".

What's pretty impressive about this is: suppose we didn't know the minimax theorem was true. So, we imagine there might be some game where if the opponent had to pick his randomized strategy first and tell us what it was, we could guarantee getting v, but if we had to

pick our randomized strategy first and tell him then we could only get some w = v - delta. Now, we run this adaptive algorithm in repeatedly playing against some opponent. OPT makes at least v per round on average, so we are approaching v. But, I've committed to my algorithm, so the opponent should be able to make sure I make at most w on average per round. This is a contradiction. So, the theorem we get actually gives a *proof* of the minimax theorem as a corollary.

A couple applications: say each week you go food shopping and at the end you have to pick a line to stand in at the checkout counter, and maybe you have n strategies for picking which line will go fastest (or each week you bet on a football game and have n strategies for picking the winner). Each week you pick one of the strategies, and then you find out how you did and compare it to how well you would have done with each of the other strategies that day. Want to do nearly as well as best of those strategies in hindsight.

To simplify the presentation, let's assume all entries in the matrix are 1 or 0 (win or lose). You can generalize but then the calculations get messier so let's not. We will think of our algorithm as the row-player. When the opponent plays some column, then we'll call rows with a 1 entry the "winning rows" and the others the "losing rows". E.g., some games to consider:

```
1 0
0 1
```

- or the n-by-n version (all zeros except for 1s on the diagonal).

To get a feel for this, let's consider some strategies that don't work very well.

Strategy 1: play 50/50. This is minimax optimal. But if the opponent always plays column 1, then when you look back in hindsight you see that row 1 would have won every time whereas you only won half the time. So, not horrible but you're off by a factor of 2. In the n-by-n version, the gap is worse (factor of n).

Strategy 2: always choose the row that's done best in the past (break ties by picking the lowest index). Q: That does well against the previous opponent, but how could you as the column player totally defeat this strategy? A: alternate columns and make it lose every time. So, in T rounds, best row won T/2 times, but this algorithm won zero times. So that's even worse!

Strategy 3: same as (2) but break ties at random. This does better in the bad example for strategy 2, but still not great. Every odd round it has a 1/2 chance of winning, and every even round it has a 0 chance of winning, so it's off by a factor of 2 (factor of n in n-by-n version).

We will give an algorithm that does much better: for any n-by-n game, given a value epsilon ("learning rate"), it will get in expectation at least:

OPT(1 - epsilon/2) - ln(n)/epsilon.

E.g., n=100 rows, epsilon = 0.1. When OPT=1000, you get at least 904.

Algorithm idea: just randomizing over the rows that did best in hindsight like #3 above is not enough randomness. Instead, want probability of choosing some row to degrade gracefully with distance from optimum.

EXPONENTIAL-WEIGHTS ALGORITHM
==========
Give each row a "weight" w_i starting at 1. Let W = w_1 + ... + w_n.

- Choose row with probability proportional to weight (i.e., w_i/W).
- After the opponent chooses his column, take all the winning rows and multiply their weight by 1 + epsilon to determine the probability distribution for the next round.

Analysis
========
Here's how the analysis is going to go. The total weights initially sum to n. At the end of the game, they sum to at *least* (1+epsilon)^OPT, where OPT is total score of the best row in hindsight, since that's the value of just the largest weight. What we'll show is that increase in the weights is tied to the probability we have of winning.

- Let W_t be total weight at time t. w_{i,t} = weight of row i at time t.
- Let p_t be our probability mass on the winning rows at time t.

All winning rows i had weights multiplied by 1+epsilon. So, the total weight went up by p_t*W_t*epsilon.

So, W_{t+1} = W_t + epsilon * p_t * W_t = W_t(1 + epsilon * p_t).

Notice: we've related the increase in weights to our probability of winning. The only way the total weight can go up by a large percentage is if this probability is large.

W_final = n*(1 + epsilon * p_1)(1 + epsilon * p_2)(1 + epsilon *p_3)...
>= (1+epsilon)^OPT.

Let's take ln (natural log) of both sides and do a little magic:

ln(n) + sum_t[ln(1 + epsilon*p_t)] >= OPT*ln(1+epsilon).

Now, use the fact that ln(1+x) <= x [draw curve or remember Taylor]

ln(n) + sum_t[epsilon*p_t] >= OPT*ln(1+epsilon)

And using the fact that our expected gain = sum_t p_t, we have:

Our expected gain >= [OPT * ln(1+epsilon) - ln(n)]/epsilon.

Using the fact that ln(1+epsilon) >= epsilon - epsilon^2/2. (from Taylor expansion), we can simplify this to:

Our expected gain >= [OPT(epsilon - epsilon^2/2) - ln(n)]/epsilon
== OPT(1 - epsilon/2) - ln(n)/epsilon. QED (ta-da)

* Number-theoretic algorithms
  - fast modular exponentiation
  - GCD
  - a^{-1} mod N

==========

In the next few classes we are going to talk about algorithms for number problems. Assume inputs given in binary, so if our input is some number N, then its "size" n is log(N).

Some basic things we can do in polynomial time: add two numbers, multiply two numbers, take A mod N (by this I mean the remainder when A is divided by N: the smallest non-negative integer of the form A - kN).

Some basic facts: if we want to compute, say, X*Y*Z mod N, we can mod out by N as we go, since (X*Y - kN)*Z = X*Y*Z - k'N for some k'. So, can keep the numbers at size O(n) = O(log N).

Let's look at our first nontrivial problem: modular exponentiation.

MODULAR EXPONENTIATION

Given A,B,N, all n-bit numbers. Goal: compute A^B mod N.

If B was small, like 3, we could do this easily by just multiplying A by itself B times. But what if B is a large n-bit number? E.g., you need to do this when decrypting under RSA. Note: our goal is at least plausible since the output is at most n bits long (if you didn't have the "mod N" then we couldn't even write down the output in polynomial time...). So, what's a faster way than multiplying A by itself B times?

Ans: Can use repeated squaring. Let X=1. Walk left-to-right down the bits of B. Each time we see a 1, do X = (X^2)*A mod N. Each time you see a 0, just do X = X^2 mod N. Notice that at each step we have A^B' mod N where B' is number corresponding to the part of B we've read so far.

GREATEST-COMMON-DIVISOR

GCD(A,B): GCD(A,B) is the largest integer d such that A and B are both multiples of d. gcd(A,0)=A.

Can we compute GCD(A,B) quickly? Notice that the number of bits in the input is log(A)+log(B) so we want to have a good dependence on that. Classic algorithm over 2000 years old called Euclid's alg. Based on observation that GCD(A,B) = GCD(B, A mod B). [Proof: if A and B are multiples of d, so A = A'*d and B = B'*d, then A-kB = A'd - kB'd is a multiple of d too. Similarly, if B is a multiple of d and "A mod B" is a multiple of d then A has to be a multiple of d.]

So, this is the algorithm:

```
GCD(A,B)  // assume A >= B (will be true after 1st iteration anyway)
if (B==0) return A
else return GCD (B, A mod B)
```

E.g., GCD(51, 9) = GCD(9,6) = GCD(6,3) = GCD(3,0) = 3.

Can anyone see quick argument that the number of iterations is linear

in the number of bits in the input? One way to see this is that "A mod B" is guaranteed to have at least one fewer bit than A. In particular, if B has fewer bits than A then this is easy, and if A and B have the same number of bits, then doing A-B gets rid of the leading 1, so again it is true. So, each iteration reduces the total number of bits in the inputs by at least 1.

- EXTENDED GCD: also compute integers x and y such that d = Ax + By.
For example, A=7, B=9. d=1. 1=4*7-3*9, so x=4, y=-3.

How to do it: can compute with same algorithm. Recursively, running on B and A-kB, we compute x', y',d such that d = Bx' + (A-kB)y'. This means that d = Ay' + B(x'-ky'). This seems like a curiosity but it turns out to be really useful.

More on Modular Arithmetic

Z_N = {0,1,2,...,N-1}
Z_N^* = {A in Z_N : gcd(A,N) = 1}. If N is prime, then Z_N^* = {1,2,...,N-1}

Z_N^* is a group under multiplication mod N: if you multiply two numbers relatively prime to N, you get another number relatively prime to N. (If N doesn't share any factors with either one, then it doesn't share any factors with their product). Z_N is a group under addition mod N.

[At this point we will use "(mod N)" to mean we are doing everything modulo N].

A^{-1} (mod N): the inverse of A modulo N is defined to be an integer B in Z_N^* such that AB = 1 (mod N). Each A in Z_N^* has an inverse modulo N.

Question: why do inverses exist, and how can we compute them quickly? E.g., what is 5^{-1} mod 17?

Here's how: compute extended GCD of N and A. Get x,y such that Nx+Ay=1. So, Ay = 1 mod N: y is the inverse of A.

E.g., EGCD(17,5) calls EGCD(5,2) where 2 = 17-3*5. This returns x'= 1, y' = -2. So, our answer is x = -2, y = 1 - 3*(-2) = 7. So, 5^{-1} = 7 mod 17.

Euler's Theorem, Fermat's little theorem

Fermat's little theorem: if N is prime and A is in Z_N^*, then A^{N-1} = 1 mod N.

So, if we pick some A (like A=2) and compute A^{N-1} mod N (which we now know how to do efficiently), and find the result is not equal to 1, this is a PROOF that N is not prime, even though it gives us no information about how to factor N.

Of course it could be that composites also have the same property, but it turns out that composites actually come in two types. A rare type, called Carmichael numbers have the same property as above, but they turn out to be easy to factor and in any case they are very sparse. The rest of the composites N have the property that at least half of the A in Z_N^* satisfy A^{N-1} != 1 mod N. So if your number is not Carmichael, and you pick 100 random A's and they all give you 1, you can be pretty confident that the number is

prime. This gives a a fast randomized primality test. Recently, a deterministic no-error polynomial-time primality test was devloped too (but it is slower).

In the next class we will look at all this in more detail.

* Number-theoretic algorithms II
  - an important property of groups
  - Fermat's little thm, Euler's thm
  - primality testing
  - complexity classes RP, co-RP, BPP

==========

We ended class last time with Fermat's little theorem: if N is prime and A is between 1 and N-1, then A^{N-1} = 1 mod N.

So, if we pick some A (like A=2) and compute A^{N-1} mod N (which we now know how to do efficiently), and find the result is not equal to 1, this is a PROOF that N is not prime, even though it gives us no information about how to factor N. Today we'll see a proof of FlT, and a generalization called Euler's theorem, and then we'll extend this further to get a randomized polynomial-time algorithm for testing if a number is prime or composite.

DEFN: Z_N^* = {A in 1..N such that GCD(A,N)=1}

E.g., Z_15^* = {1,2,4,7,8,11,13,14}
Recall, Z_N^* is a group under multiplication mod N. That means it's closed under the group operation (e.g., 7*8=11 mod 15), and also every A in Z_N^* has an inverse B in Z_N^* (AB=1 mod N). E.g., 2^{-1} = 8 mod 15.

Here is a REALLY IMPORTANT PROPERTY of groups: say G is a group and H is a subgroup of G (if A,B are in H then A*B is in H; if A is in H then A^{-1} is in H). Then THE SIZE OF H DIVIDES THE SIZE OF G.

> Proof: Say H is a subgroup of G and y is not in H. Then the coset yH = {yh : h in H} is a set of size |H| (if y*h1 = y*h2 then h1 = h2) and is disjoint from H (if y*h1 = h2 then y = h2*h1^{-1}, which is in H by H's group closure properties). Furthermore, all cosets are disjoint (if z*h1 = y*h2 then z = y*h3 for some h3 in H).
>
> E.g., G = Z_15^*, H = {1,2,4,8}.

DEFN: N is a Carmichael number if N is composite but A^{N-1}=1 for all A in Z_N^*.

THEOREM: if N is composite but not a Carmichael number, then A^{N-1}!=1 for *at least half* of the A in Z_N^*.

Proof: Let H = {A in Z_N^* : A^{N-1} = 1 mod N}. Then H is a subgroup of Z_N^* because it's closed under multiplication (if A^{N-1} = 1 and B^{N-1} = 1 then (AB)^{N-1}=1) and inverses (if AB = 1 then (AB)^{N-1} = 1, so if A^{N-1} = 1 then B^{N-1}=1). This means that if there is even a single element of G that is not in H (namely if N is not Carmichael) then |H| is at most |G|/2. So, right away we get that if there is even a single witness to N being composite, there must be a *lot* of witnesses. So, if you pick 100 random A's and you always get 1 you can be pretty confident that A is either prime or a Carmichael number.

OK, now let's go ahead and prove Fermat's little theorem.

DEFN: Euler phi function: phi(N) = |Z_N^*|.
E.g., if N is prime, then phi(N) = N-1.

DEFN: for A in Z_N^*, order(A) = smallest t such that A^t = 1 (mod N).

(e.g., in Z_15^*, order(2)=4, order(14)=2)

THEOREM: for all A in Z_N^*, order(A) divides phi(N).

PROOF: {1, A, A^2, ..., A^{t-1}} is a subgroup of Z_N^*:
it's closed under multiplication mod N and taking inverses. So we just use our Really Important Property.

COROLLARY 1: Euler's Theorem: for any A in Z_N^*, A^{phi(N)} = 1 (mod N).
Proof: if t is the order of A, then phi(N) = B*t for some B by our theorem, and A^{phi(N)} = (A^t)^B = 1 (mod N).

COROLLARY 2: Fermat's little theorem: If N is prime, then for any A in Z_N^*, A^{N-1} = 1 mod N.

PRIMALITY TESTING

We're now going to give a fast randomized algorithm for testing if a number is prime, with the following properties:

> if N is prime, then it outputs YES
> if N is composite, then it outputs NO with probability at least 1 - 1/2^100.

So, if it says YES, you don't have a 100% proof that the number is prime, but you can be pretty confident. [Note, it was only very recently that a *deterministic* poly-time algorithm for this problem was developed]

We actually already almost have the algorithm. If we ignore the Carmichael numbers, then the algorithm is just this:
- pick 100 random values A between 1 and N.
- If all have A^{N-1} = 1 mod N then output YES (Probably Prime)
- Else output NO (Definitely Composite)

[don't even need to test GCD since if GCD is not 1 then for sure A^{N-1} != 1 mod N]

The trick for Carmichael numbers is it turns out they are easy to factor. (They are also very rare. Smallest is 561.)
Combining these two gives us the Miller-Rabin primality test.

More on Carmichael numbers: We're going to be able to factor Carmichael numbers using the following idea. Suppose we have some number x that's not 1 or -1 mod N, such that x^2 = 1 mod N. E.g., 11^2 = 1 mod 15. This means that (x-1)*(x+1) is a multiple of N, even though neither x-1 nor x+1 is. So, GCD(x-1,N) gives us a factor of N

(as does GCD(x+1,N)), and GCD is something WE CAN COMPUTE EFFICIENTLY!

The way we will find such an x is via the following key lemma:

KEY LEMMA: Suppose N is odd, not a prime power or perfect square, and composite. Let t < N. If there exists x in Z_N^* such that x^t != 1 (mod N), then at least half of x in Z_N^* have x^t != {-1,+1} mod N. Furthermore, if t is ODD, then such an x exists.

Proof of KEY LEMMA: Don't have time to go through details but one step is you show the set of x such that x^t *is* in {-1,1} is a subgroup, and then you also use something called the Chinese Remainder Theorem.

Proof of factoring Carmichael given our key lemma: First, we can trivially handle Ns that are even, prime-powers, or perfect squares, so we can assume the lemma above holds. Now, take N-1 and pull out all powers of 2, so that we have N-1 = B * 2^k, where B is an odd number. Now, consider exponents t = B, 2B, 4B, 8B, ..., N-1. The lemma says that we can put them into two categories:

(1) all A in Z_N^* have A^t = 1 mod N
(2) at least half of A have A^t != 1 or -1 (mod N)

The lemma tells us that t=B is in category (2), and the fact that N is Carmichael tells us that N-1 is in category 1. So the world looks something like this:

t =	B	2B	4B	8B	...	N-1
category:	2	2	1	1	...	1

^
call this point t_critical

Now, pick random A and compute A^B, A^{2B},..., A^{N-1}. Define t_critical as largest exponent in category (2). By definition, there is at least 1/2 chance that A^{t_critical} != {1,-1}. Call this x. So, x is not 1 or -1, but x^2 = 1 mod N. So, we use this to factor!

A little more complexity theory

Turns out there are a number of interesting complexity classes that you can define in terms of a polynomial-time algorithm V that takes in two inputs: an instance I, and an auxiliary string w, where we assume that the length of w is polynomial in the size of the instance.

NP: A is in NP if there is a polynomial-time algorithm V(I,w) such that:
  If "I" is in YES_A, then there exists w such that V(I,w) = YES.
  If "I" is in NO_A, then for all w, V(I,w) = NO.

Co-NP: other way around from NP: swap YES and NO.

RP (randomized polynomial time): A is in RP if exists poly-time V s.t.:
  If "I" is in YES_A, then for at least half of w, V(I,w) = YES.
  If "I" is in NO_A, then for all w, V(I,w) = NO.

Co-RP: the other way around.

BPP: two-sided error
  If "I" is in YES_A, then for at least 2/3 of w, V(I,w) = YES.

If "I" is in NO_A, then for at least 2/3 of w, V(I,w) = NO.

Can boost up the 1/2, 2/3 by repetition. We showed primality in Co-RP.

RSA PUBLIC-KEY CRYPTOGRAPHY

RSA is an answer to the question of "how can two people communicate securely if they have never met to exchange secret keys before?". Answer is to somehow separate encryption and decryption. Each person has a public encryption key that's published in the phone book, and then their own private decryption key that's kept secret. To send a msg to A, you look them up in the phone book and use their public key. The point about public-key crypto is that just because you can encrypt a message to A doesn't mean you can decrypt anyone else's msgs sent to A.

RSA: Person A (Alice) subscribes by
(1) picking two large primes p and q (say 100 decimal digits long) and
computing N = p*q.
(2) Picking a small odd number e relatively prime to (p-1)*(q-1). E.g., e=3.
(3) Computing d = e^{-1} mod phi(N), where phi(N) = (p-1)*(q-1).
(4) Publishing the pair (e,N) as her PUBLIC KEY in a global phone
book, and keeping the pair (d,N) secret as her SECRET KEY.
The primes p and q can now be thrown away.

Person B(Bob) sends msg M to Alice by computing x = M^e mod N, and sending x.

(5) To decrypt, Alice computes x^d mod N, which is M.

[Ignoring various issues like: might want to prepend garbage onto M so that evesdropper can't guess-and-check]

Let's now look at details of how/why all this works:
STEP 1: a reasonable proportion of random numbers are prime. So can just pick random numbers and test, until we get two primes.

STEP 3: just requires computing inverse which we know how to do.

STEP (5): Why do we get back M?
Answer is that this comes from Euler's theorem.
x^d = M^{de} mod N. By definition, de = 1 + k*phi(N). So,
M^{1 + k*phi(N)} = M*M^{phi(N)^k} = M*1^k = M (mod N).
Also: use fast exponentiation here since d might be a large number.

Why might we expect RSA to be secure? Here is one fact: given N and e, finding the decryption key d is as hard as factoring. (Though this doesn't say there might not be some other way of decrypting a message that people haven't thought of).

* Multiplying polynomials
* Fast Fourier Transform (FFT)

==========

Today we are going to talk about the Fast Fourier Transform, a widely used algorithm in areas like signal processing, speech understanding, digital audio, radar.

We'll develop it in trying to solve the problem of "how to multiply quickly" we talked about on the first day of class. This is not how it was invented historically (and it obscures the connection to the usual kind of Fourier Transform), but I think it's most natural from the perspective of Algorithms. You don't need to know what a FT is for this lecture.

Warning: even though the algorithm in the end won't be that complicated, this will be one of the most difficult lectures of the class!

==========

At the start of class we talked about the problem of multiplying big integers. Let's look at a simpler version of the problem, which is the problem of multiplying polynomials. It sounds more complicated, but really it's just the problem of multiplying integers without the carries.

MULTIPLYING POLYNOMIALS

==========

E.g., multiply $(x^2 + 2x + 3)(2x^2 + 5)$ =

$$
\begin{array}{rrrrr}
 & & 2x^2 & + 0x & + 5 \\
 & & 1x^2 & + 2x & + 3 \\
 & & \multicolumn{3}{l}{\text{-------------}} \\
 & & 6 & 0 & 15 \\
 & 4 & 0 & 10 & \\
2 & 0 & 5 & & \\
\multicolumn{5}{l}{\text{----------------------------}} \\
\multicolumn{5}{l}{2x^4 + 4x^3 + 11x^2 + 10x + 15}
\end{array}
$$

MODEL: view each individual small multiplication as a unit cost operation.

More generally, given $A(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$,
$B(x) = b_0 + b_1 x + \ldots + b_{n-1}x^{n-1}$

Our goal is to compute the polynomial $C(x) = A(x)*B(x)$.
$c_i = a_0*b_i + a_1*b_{i-1} + \ldots + a_i*b_0$.

If we think of A and B as vectors, then the C-vector is called the "convolution" of A and B.

- Straightforward computation is $O(n^2)$ time. Karatuba is $n^{1.58..}$

- we'll use FFTs to do in $O(n \log n)$ time. This is then used in Schonhage-Strassen integer multiplication algorithm that multiplies two n-bit integers in $O(n \log n \log\log n)$ time. We're only going to do polynomial multiplication.

High Level Idea of Algorithm
============================
Let m = 2n-1. [so degree of C is less than m]
1. Pick m points x_0, x_1, ..., x_{m-1} according to a secret formula.
2. Evaluate A at each of the points: A(x_0),..., A(x_{m-1}).
3. Same for B.
4. Now compute C(x_0),..., C(x_{m-1}), where C is A(x)*B(x)
5. Interpolate to get the coefficients of C.

This approach is based on the fact that a polynomial of degree < m is uniquely specified by its value on m points. It seems patently crazy since it looks like steps 2 and 3 should take O(n^2) time just in themselves. However, the FFT will allow us to quickly move from "coefficient representation" of polynomial to the "value on m points" representation, and back, for our special set of m points. (Doesn't work for *arbitrary* m points. The special points will turn out to be roots of unity).

The reason we like this is that multiplying is easy in the "value on m points" representation. We just do: C(x_i) = A(x_i)*B(x_i). So, only O(m) time for step 4.

Let's focus on forward direction first. In that case, we've reduced our problem to the following:

GOAL: Given a polynomial A of degree < m, evaluate A at m points of our choosing in total time O(m log m). Assume m is a power of 2.

The FFT:
=======

Let's first develop it through an example. Say m=8 so we have a polynomial

$$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6x^6 + a_7x^7.$$
(as a vector, A = [a_0, a_1, ..., a_7])

And we want to evaluate at eight points of our choosing. Here is an idea. Split A into two pieces, but instead of left and right, have them be even and odd. So, as vectors,
A_even = [a_0, a_2, a_4, a_6]
A_odd = [a_1, a_3, a_5, a_7]

or, as polynomials:

$$A_{even}(x) = a_0 + a_2 x + a_4 x^2 + a_6 x^3$$
$$A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + a_7 x^3.$$

Each has degree < m/2. How can we write A(x) in terms of A_even and A_odd?

$$A(x) = A_{even}(x^2) + x\, A_{odd}(x^2).$$

What's nice is that the effort spent computing A(x) will give us A(-x) almost for free. So, let's say our special set of m points will have the property:

The 2nd half of points are the negative of the 1st half (*)

E.g., {1,2,3,4,-1,-2,-3,-4}.

Now, things look good: Let T(m) = time to evaluate a degree-m polynomial at our special set of m points. We're doing this by evaluating two degree-m/2 polynomials at m/2 points each (the squares), and then doing O(m) work to combine the results. This is great because the recurrence T(m) = 2T(m/2) + O(m) solves to O(m log m).

But, we're deluding ourselves by saying "just do it recursively". Why is that? The problem is that recursively, our special points (now {1, 4, 9, 16}) have to satisfy property (*). E.g., they should really look like {1, 4, -1, -4}. BUT THESE ARE SQUARES!! How to fix? Just use complex numbers! E.g., if these are the squares, what do the original points look like?

{1, 2, i, 2i, -1, -2, -i, -2i}

so then their squares are: 1, 4, -1, -4
and their squares are: 1, 16

But, at the next level we again need property (*). So, we want to have {1, -1} there. This means we want the level before that to be {1, i, -1, -i}, which is the same as {1, i, i^2, i^3}. So, for the original level, let w = sqrt(i) = 0.707 + 0.707i, and then our original set of points will be:

1, w, w^2, w^3, w^4 (= -1), w^5 (= -w), w^6 (= -w^2), w^7 (= -w^3)

so that the squares are: 1, i, i^2 (= -1), i^3 (= -i)
and *their* squares are: 1, -1
and *their* squares are: 1

The "w" we are using is called the "primitive eighth root of unity" (since w^8 = 1 and w^k != 1 for 0 < k < 8).

In general, the mth primitive root of unity is the vector
w = cos(2*pi/m) + i*sin(2*pi/m)

Alternatively, we can use MODULAR ARITHMETIC!
==========

E.g., 2 is a primitive 8th root of unity mod 17.
{2^0,2^1,2^2,...,2^7} = {1,2,4,8 16,15,13, 9}
= {1,2,4,8,-1,-2,-4,-8}.

Then when you square them, you get {1,4,-1,-4}, etc.
This is nice because we don't need to deal with messy floating-points.

THE FFT ALGORITHM
==========
Here is the general algorithm in pseudo-C:

Let A be array of length m, w be primitive mth root of unity.
Goal: produce DFT F(A): evaluation of A at 1, w, w^2,...,w^{m-1}.

```
FFT(A, m, w)
{
  if (m==1) return vector (a_0)
```

```
  else {
    A_even = (a_0, a_2, ..., a_{m-2})
    A_odd  = (a_1, a_3, ..., a_{m-1})
    F_even = FFT(A_even, m/2, w^2)    //w^2 is a primitive m/2-th root of unity
    F_odd = FFT(A_odd, m/2, w^2)
    F = new vector of length m
    x = 1
    for (j=0; j < m/2; ++j) {
      F[j] = F_even[j] + x*F_odd[j]
      F[j+m/2] = F_even[j] - x*F_odd[j]
      x = x * w
  }
  return F
}
```

THE INVERSE OF THE FFT
======================

Remember, we started all this by saying that we were going to multiply two polynomials A and B by evaluating each at a special set of m points (which we can now do in time O(m log m)), then multiply the values point-wise to get C evalauated at all these points (in O(m) time) but then we need to interpolate back to get the coefficients. In other words, we're doing F^{-1}(F(A) \cdot F(B)).

So, we need to compute F^{-1}. Here's how.

First, we can look at the forward computation (computing A(x) at 1, w, w^2, ..., w^{m-1}) as an implicit matrix-vector product:

```
+-------------------------------------+  +-----+    +-------+
|  1   1    1     1   ...    1       |  | a_0 |    | A(1)  |
|  1   w    w^2   w^3 ... w^{m-1}     |  | a_1 |    | A(w)  |
|  1   w^2  w^4   w^6 ... w^{2(m-1)}  |  | a_2 | == | A(w^2)|
|  1   w^3  w^6   w^9 ... w^{3(m-1)}  |  | a_3 |    | A(w^3)|
|          .....       .....          |  | ... |    | ...   |
|  1 w^{-1} w^{-2} w^{-3} ... w       |  |a_m-1|    | ...   |
+-------------------------------------+  +-----+    +-------+
```

(Note w^{m-1} = w^{-1} since w^m = 1)

(We're doing this "implicitly" in the sense that we don't even have time to write down the matrix.)

To do the inverse transform, what we want is to multiply by the inverse of the F matrix. As it turns out, this inverse looks very much like F itself. In particular, notice that w^{-1} is also a principal mth root of unity. Let's define \bar{F} to be the fourier transform matrix using w^{-1} instead of w. Then,

Claim: F^{-1} = (1/m) * \bar{F}. I.e., 1/m * \bar{F} * F = identity.

Proof: What is the i,j entry of \bar{F} * F? It is:

$$1 + w^{j-i} + w^{2j-2i} + w^{3j-3i} + \ldots + w^{(m-1)j - (m-1)i}$$

If i=j, then these are all = 1, so the sum is m. Then when we divide by m we get 1.

If i!=j, then the claim is these all cancel out and we get zero.
Maybe easier to see if we let z = w^{j-i}, so then the sum is:

$$1 + z + z^2 + z^3 + z^4 + \ldots + z^{m-1}.$$

Then can see these cancel by picture. For instance, try z = w, z = w^2.

Or can use the formula for summations: $(1 - z^m)/(1-z) = 0/(1-z) = 0$.

So, the final algorithm is:

```
Let F_A = FFT(A, m, w)                          // time O(n log n)
Let F_B = FFT(B, m, w)                          // time O(n log n)
For i=1 to m, let F_C[i] = F_A[i]*F_B[i]        // time O(n)
Output C = 1/m * FFT(F_C, m, w^{-1}).           // time O(n log n)
```

NOTE: If you're an EE or Physics person, what we're calling the "Fourier Transform" is what you would usually call the "inverse Fourier transform" and vice-versa.