How Hard Can It Be?

Margaret H. Wright
Computer Science Department
Courant Institute of Mathematical Sciences
New York University

"Math Matters"
Institute for Mathematics and Its Applications
University of Minnesota
November 2, 2006

About me: two hats (mathematician and computer scientist).

Tonight's topics fit in both categories.

Benefits: Exciting, endlessly fascinating things to work on.

One disadvantage: Lots of really bad jokes

An exception [math joke].

But the punch line is wrong. . . !

Mathematics is extraordinarily useful.

The title of this talk?

As of October 26, 2006, there were 331,000 Google hits for
"How hard can it be"

A typical example:

Thelma: I don't know how to fish.

Louise: Well, neither do I, but Darryl does—how hard can it be?

From the film *Thelma and Louise*, 1992.

How is mathematics related to "How hard can it be?"

One (pleasant and satisfying) way is to turn what seems to be a hard problem into an easy problem.

An example: adding 100 1-digit and 2-digit numbers by hand, i.e.,

$$x_1 + x_2 + x_3 + \cdots + x_{99} + x_{100} = ??$$

How hard can it be??

It looks as if 99 additions are needed.

(Apocryphal?) story about Carl Friedrich Gauss (1777–1855), one of the world's greatest mathematicians.

Karl Friedrich Gauss.

When Gauss was seven, his teacher asked students to add up the integers from 1 to 100, thinking that this would keep them occupied for a while.

Gauss answered almost immediately. How did he do it?

Gauss noticed that

$$
\begin{aligned}
1+2+3+4+\cdots+98+99+100 &= \\
(1+100)+(2+99)+(3+98)+\ldots &= \\
50 * 101 &= 5050.
\end{aligned}
$$

How hard was it for Gauss?

Easy: one addition, one multiplication!

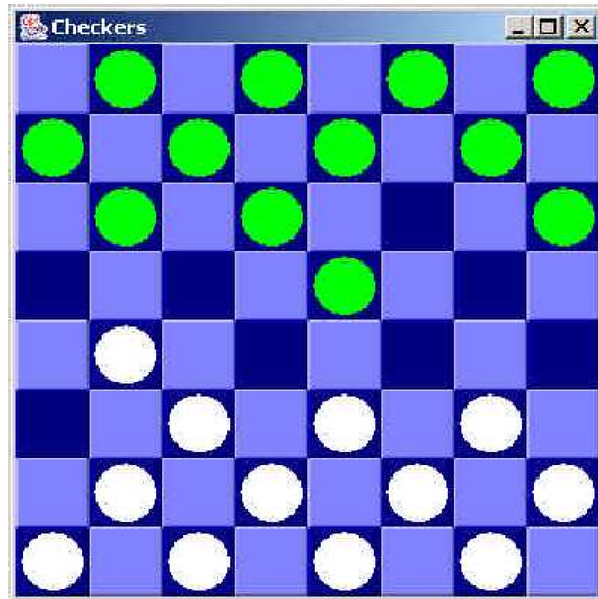
He noticed the special structure of the problem
and turned a hard problem into an easy one.

An equally useful role for mathematics is to tell us that some problems will definitely be hard!

Classic story of the king's favorite jester/checkers partner whom the king wished to reward.

The jester asked the king to place 1 piece of gold on the first square of a checkerboard, 2 on the second, 4 on the third, and so on.

Thinking of the first three squares, the king said to himself, "How hard can it be to reward my jester?"



Answer: Very hard and *very* expensive…

The king should have done the math!

The amount on square k is 2^{k-1}, so there will be $32,768$ pieces of gold on square 16, and $1,048,576$ pieces of gold on square 21, with 43 squares to go.

The king's solution (much easier): banishing the jester.

Time to get serious... we can't have a math talk without some definitions.

If x is a number, and a_0, a_1, $\ldots$, a_p are $p+1$ given constants,

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_p x^p$$

is a *polynomial* in x—for example, $2 + 10x + 6x^3$.

An *algorithm* is a systematic procedure for solving a problem.

We all use algorithms every day.

Exponential growth: speaking broadly, the larger a number gets, the faster an exponential growth function of that number grows.

A function of the form k^n, where k is a fixed number greater than one, displays exponential growth as n grows—like the growth with each square in the number of gold pieces paid to the jester.

If n is large enough, the ratio of a polynomial in n to an exponential function becomes *arbitrarily small*, even though the polynomial may be larger for smaller n.

n	n^5	2^n	$n^5/2^n$
20	3.05 seconds	1 second	3.05
40	1.7 minutes	12.1 days	9.3×10^{-5}
60	12.9 minutes	349 centuries	6.7×10^{-10}

A problem is said to be *in P* or *easy* (*in theory—but that's another talk...*) if there is a polynomial-time algorithm to solve it.

If the problem's "size" is n, the time needed by the algorithm to solve the problem is a polynomial in n.

P = Easy to solve

A problem is *in NP* if you can check the correctness of a proposed solution in polynomial time, but it *seems to be* very hard to solve the problem.

NP = Easy to check, hard to solve (as far as we know now).

[More on this later.]

Example of an NP-problem: completing a jigsaw puzzle with n pieces.

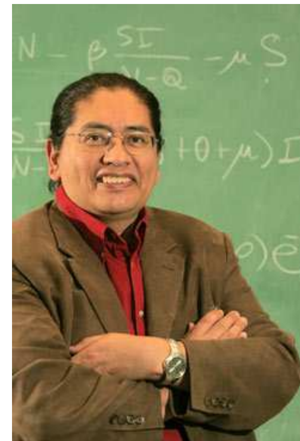
To check whether the puzzle is completed, just check the correctness of each piece in turn—does it match the neighboring pieces? This is approximately proportional to n, so it is polynomial-time and easy.

But if n is large, it is not feasible to solve the puzzle by a "brute force" technique of examining all possible combinations of the n pieces.

Mathematicians often contrast “brute force”...

with clever, insightful mathematicians,

like the following sample of people who will be here tomorrow for the Blackwell-Tapia conference:



My first (serious) example of the importance of “How hard can it be?”

Cryptography

An overly simplistic definition: Communicating secrets securely.

A problem for centuries, notably in military contexts.

Caesar cipher; named after Julius Caesar, famous Roman (100 B.C.–44 B.C.)

If Caesar had anything confidential to say, he wrote it in cipher ... by shifting the order of the letters of the alphabet... to decipher these, substitute the fourth letter of the alphabet, namely D, for A, and so on.

fubswrjudskb ⟶ **cryptography**

Later, more complicated variation: substitution of one letter for another rather than just shifting.

abcdefghijklmnopqrstuvwxyz

zebrascdfghijklmnopqtuvwxy

flee at once $\longrightarrow$ **siaa zq lbka**

To break this code: use frequency analysis (certain letters appear in English more often than others)

Some crypto terminology:

Encryption: transforming data (e.g., a message) into a form that you hope is impossible to understand without knowledge of a key.

Decryption: transforming an encrypted message into an intelligible form.

Key: information that allows encryption and decryption.

For thousands of years, everyone believed that it was impossible for two people to communicate securely without a *SECRET* key, known only to them and to other trusted parties.

But there's a big problem with a secret key: how can it be communicated securely?

(Consider sending a password by email—definitely not secure!)

A giant breakthrough, seemingly miraculous at the time, that revolutionized cryptography:

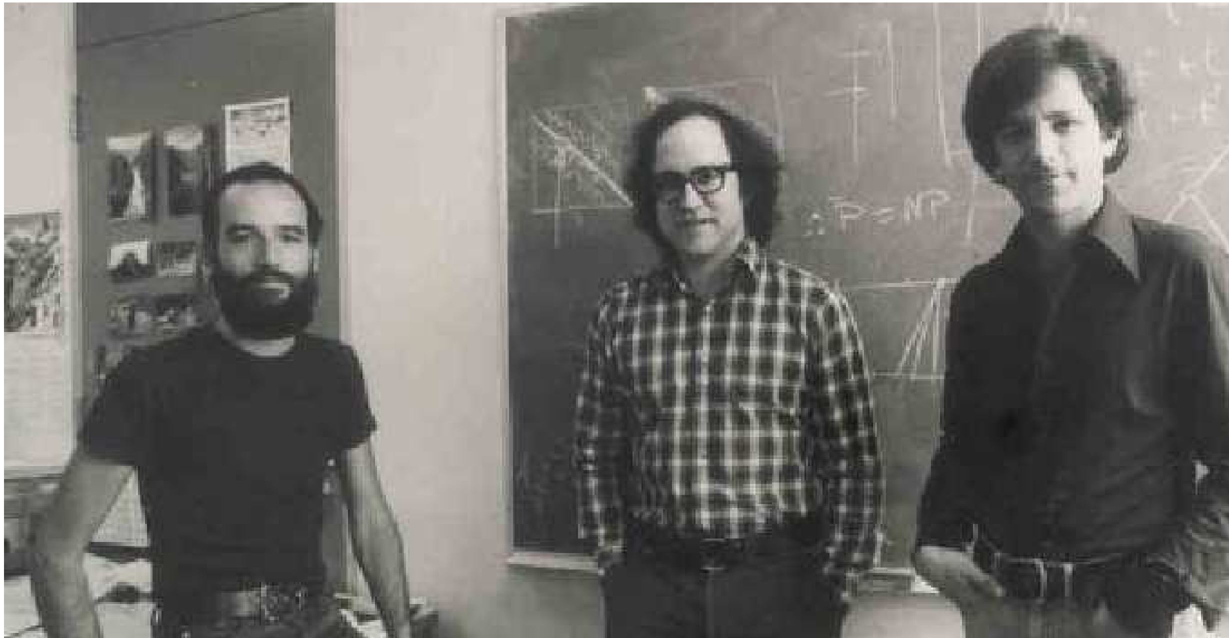
Use a **PUBLIC** *key*!

Diffie and Hellman (1976); related work by Merkle

How can this possibly work if the key is public???

Answer: Using mathematics, of course!

RSA public key algorithm, invented in 1977 by Ron Rivest, Adi Shamir, and Len Adleman



See `www.rsasecurity.com/rsalabs` for a wealth of information.

Idea: each person has a *pair* of keys, one public (published openly!), one private (known only to that person), and the private key is linked mathematically to the public key, as we shall see later.

Note: no need for two or more people to know a secret key.

An important side comment: you are about to meet the two most famous characters in cryptography, introduced in the original RSA paper and featured in discussions of cryptography in all languages.

Alice and Bob, cryptography's perennial couple, always trying to communicate.

To send a secret message to Bob, Alice looks up his public key and uses it to encrypt the message.

Bob is the only person who can decrypt it, using his secret key (which Alice does not know).

To send a secret reply to Alice, Bob does the same thing, using Alice's public key.

Eve (an evil eavesdropper) knows that Alice and Bob have been communicating, but cannot find out what they are saying.

One more useful definition:

Modulus arithmetic

a mod b, where a and b are integers, gives the *remainder* when a is divided by b

$21 \bmod 5 = 1$, since $21 = 5 * 4 + 1$

$42 \bmod 8 = 2$, since $42 = 8 * 5 + 2$

For Bob, here's how the RSA algorithm works:

1. Pick two primes, p and q, of approximately equal size.
2. Compute $N = p * q$, and $M = (p - 1) * (q - 1)$.
3. Select a number e so that e and M have no factors in common.
4. Compute d so that $e * d = 1 \bmod M$.
5. Bob's *public key* is (e, N) and his *private key* is d.

Anyone can find out Bob's public key, meaning e and N, so Alice can send a message m to Bob by encrypting it as $c = m^e \bmod N$.

To decrypt c, Bob computes $m = c^d \bmod N$ using d, his private key (known only to him).

Computing d requires M, which requires $p - 1$ and $q - 1$, which requires p and q, which are the factors of N.

Hence decryption without knowing d involves factoring N!

Simplest possible example of RSA:

$p = 11$, $q = 3$.

$N = p * q = 11 * 3 = 33$.

Note: a terrible choice, since the prime factors are obvious!

$M = (p - 1) * (q - 1) = 10 * 2 = 20$.

Choose $e = 3$ (relatively prime to M, i.e., no common factors with 20).

$d = 7$ (since $e * d = 21 = 20 + 1 = 1 \bmod 20 = 1 \bmod M$).

Public key $= (e, N) = (3, 33)$.

Private key $= d = 7$.

Alice would encrypt the message $m = 6$ with Bob's public key by computing

$$\begin{aligned} c = m^e \bmod N &= (6)^3 \bmod 33 \\ &= 216 \bmod 33 = (33 * 6 + 18) \bmod 33 \\ &= 18 \end{aligned}$$

and sending the encrypted message $c = 18$.

Bob would then decrypt $c = 18$ by using his private key d to compute

$$\begin{aligned} c^d \bmod N = 18^7 \bmod 33 &= 612,220,032 \bmod 33 \\ &= (18,552,122 * 33 + 6) \bmod 33 \\ &= 6, \end{aligned}$$

and discover that $m = 6$ was the message sent by Alice.

RSA's security depends on the difficulty of factoring large integers.

How can this be, since factoring is just the "opposite" of multiplying??

Multiplication	Factoring into prime factors
$37 * 59 = 2183$	$2183 = 37 * 59$

Multiplying two n-digit numbers using a grade-school algorithm takes n^2 steps and hence is in P, since it has a polynomial-time algorithm.

It's easy (by multiplying) to check whether alleged factors are correct, but the best known factoring algorithm on an m-digit number is *exponential* in $\sqrt{m}$. So factoring is in NP and it seems to be very hard.

Practical considerations: if N is not large enough, attackers can break the system using brute force by trying all possible combinations of prime factors.

This is possible because of enormous speedups in raw computing power.

In 1976, the fastest supercomputer could perform 10^6 (one million) operations per second.

Today's fastest computers available to the public can achieve hundreds of teraflops (10^{12} operations per second). This is a gain in speed by a factor of *one million*!

For perspective, keep in mind that a jet plane is only *40 times faster* than a fast human runner at his/her peak speed.

So a brute force attack with lots of fast computers could easily factor a too-small N.

A challenge for (some of? all of?) you:

the RSA factoring challenge.

RSA Security posts huge numbers on the Web and challenges interested people to factor them for cash rewards (and glory).

A recent result: the 193-digit number RSA-640

3107418240490043721350750035888567930037346022842727545720161948823206440518081504556346829671723286782437916272838033415471073108501919548529007337724822783525742386454014691736602477652346609

was factored in November 2005, taking about 5 months of calendar time, about 30 years of CPU time on multiple machines.

The factors of RSA-640 are

1634733645809253848443133883865090859841783670033092312181110852389333100104508151212118167511579

and

1900871281664822113126851573935413975471896789968515493666638539088027103802104498957191261465571

Still unfactored!

RSA-2048, with 617 digits, has a prize of $200,000:

2519590847565789349402718324004839857142928212620403202777 71
3783604366202070759555626401852588078440691829064124951508 21
8929855914917618450280848912007284499268739280728777673597 14
1834727026189637501497182469116507761337985909570009733045 97
4880842840179742910064245869181719511874612151517265463228 22
1686998754918242243363725908514186546204357679842338718477 44
4792073993423658482382428119816381501067481045166037730605 62
0161967625613384414360383390441495263443219011465754445417 84
2402092461651572335077870774981712577246796292638635637328 99
1215483143816789988504044536402352738195137863656439121201 03
9712282212072035 7

Again: the security of RSA depends on the hardness of factoring.

Experts are convinced that factoring is hard, and 30 years of experience support this view, but the result has not yet been rigorously proved.

Please don't lose too much sleep about this, since the National Security Agency and others of similar inclinations are thinking about it all the time.

Even so, it's a very good idea to be aware of the mathematics and computer science issues so that you can have an intelligent opinion about cryptography.

Second and last topic: linear programming (LP).

George B. Dantzig (1914–2005), affectionately known as “GBD”, the father of linear programming.

Photo credit: Edward Souza, Stanford News Service

Linear programming is a modern (60-year-old) field of mathematics and computer science, dating from near the end of World War II (approximately 1945).

"Programming" in those days (before modern computers) meant developing a plan, organization, or schedule.

What *is* linear programming?

Definition by example: a bad practice (like proof by intimidation), but please indulge me. . .

A linear function

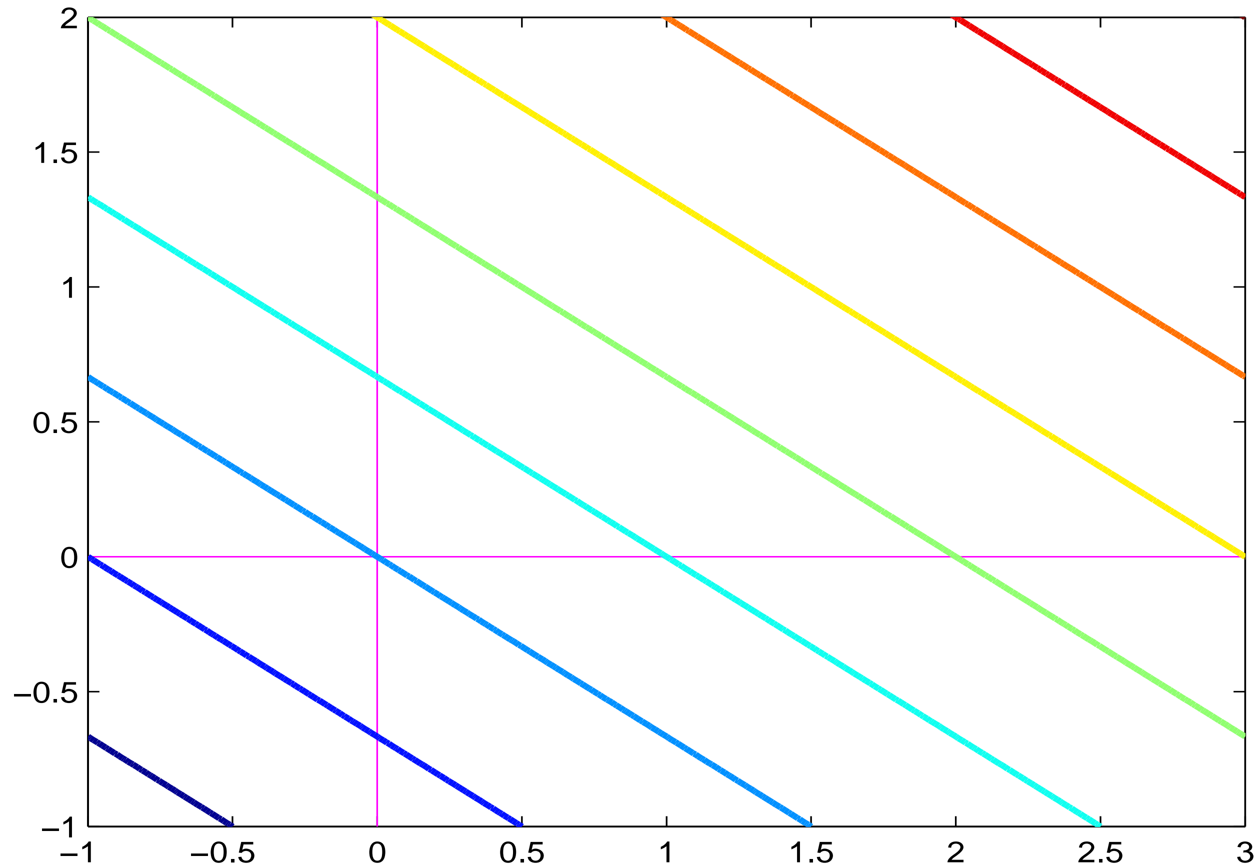
$2x$ is a linear function of the variable x;

x^2 is a nonlinear function of x.

$2x + 3y$ is a linear function of the pair (x, y);

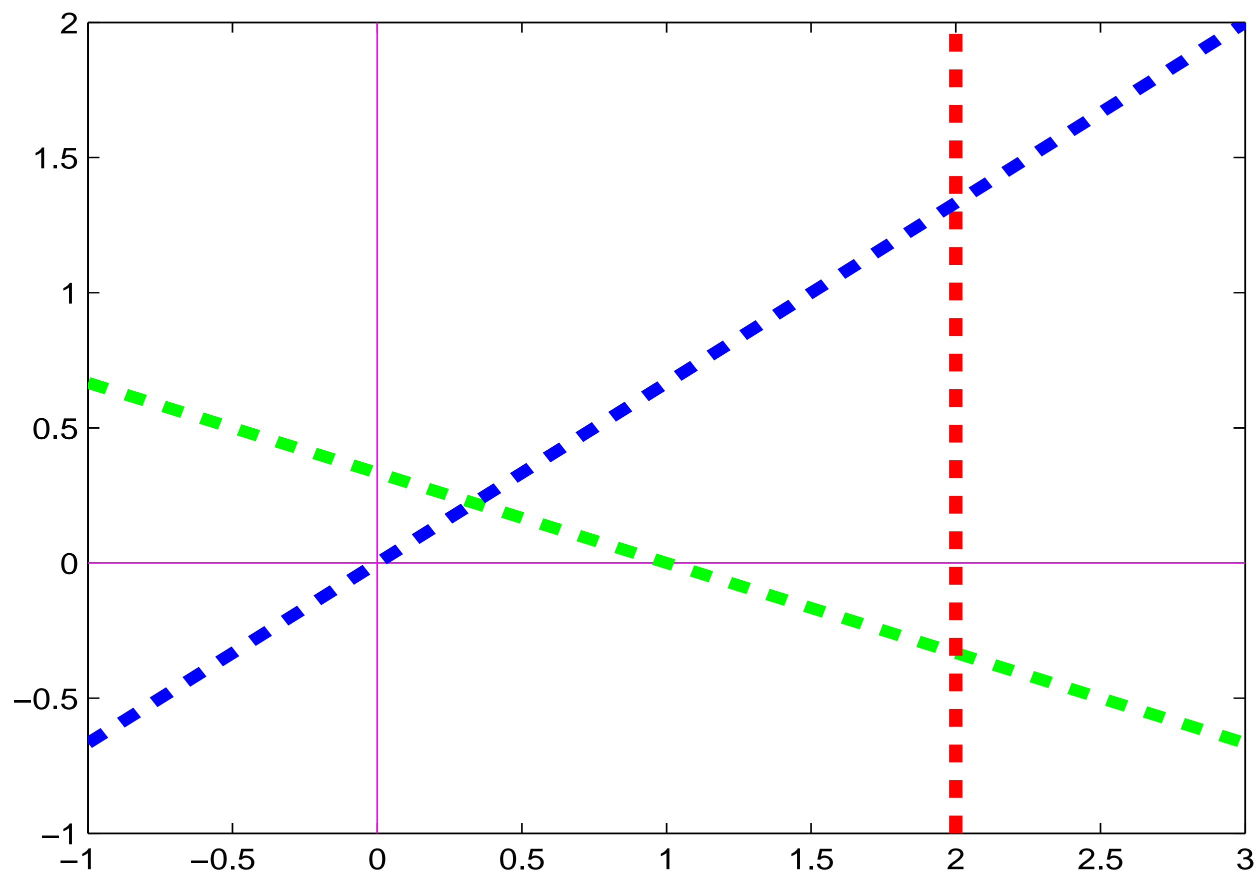
$2x + y^2$ is a nonlinear function of the pair (x, y).

In two dimensions, the contours of a linear function are straight lines.

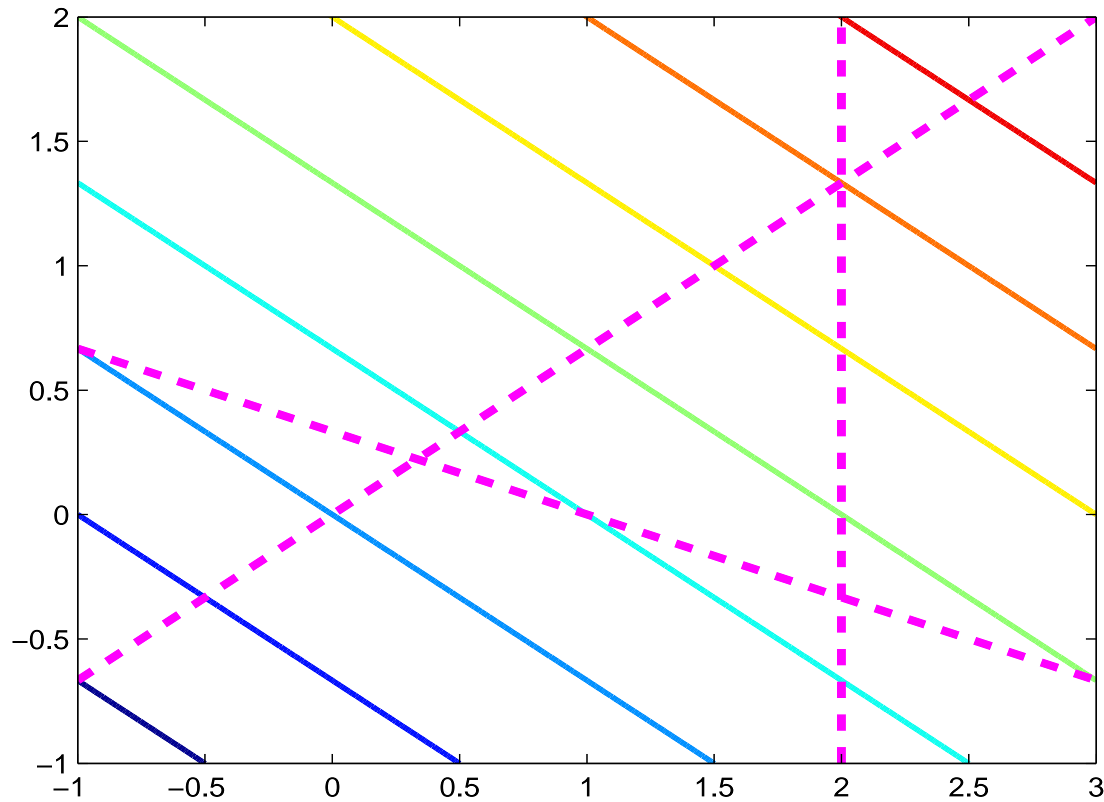


What about a “linear constraint”?

Another definition by example…



And "linear programming"? Optimizing (minimizing or maximizing) a linear function subject to linear constraints—our final definition by example. . .



How is eating a good breakfast related to linear programming?

The earliest-ever application of LP during World War II, and the basis for one of GBD's favorite anecdotes: feeding the troops a nutritionally adequate diet at minimal cost.

Problems like this are called *DIET PROBLEMS.*

Given a choice of foods, we want a minimum-cost daily diet that satisfies nutritional constraints.

Note that, without the constraints, the cheapest diet would contain no servings of anything!

An example constructed for this talk:

Eating not just a good breakfast, but an *optimal* breakfast.

Eight foods to choose from:

1. pancakes $= x_1$
2. milk $= x_2$
3. freshly squeezed orange juice $= x_3$
4. scrambled eggs $= x_4$
5. Weetabix, a high-fiber cereal $= x_5$
6. fresh blueberries $= x_6$
7. bacon $= x_7$
8. vitamin/mineral pills $= x_8$

Our goal is to find the cheapest breakfast satisfying

daily fiber ≥ 120, daily calcium ≥ 100, and all x's ≥ 0.

(Can't eat -5 units of a food.)

food	fiber	calcium	cost
pancakes	5	10	3
milk	0	40	1
orange juice	4	5	5
eggs	8	0	3
Weetabix	100	20	10
blueberries	50	20	5
bacon	0	0	10
pills	30	400	2

With this (apparently sensible) formulation, what's the optimal breakfast??

1.14 units of Weetabix and .19 units of vitamin/mineral pills.

Yuck. Not even milk for the Weetabix!

The original (true) story, as told by GBD...

LP was applied for the first time to the diet problem, which took several days to solve, with all computations done by humans—then called "computers"—using calculating machines.

To everyone's horror, the optimal daily diet came out as

20 beef bouillion cubes and 10 vitamin pills!

This was a terrible answer—but it was *not* the wrong answer.

It was *the right answer to the problem that had been given.*

What was wrong was the problem formulation!

More constraints were needed.

Mathematics can tell us a lot, but it may not be able to detect a wrongly posed problem.

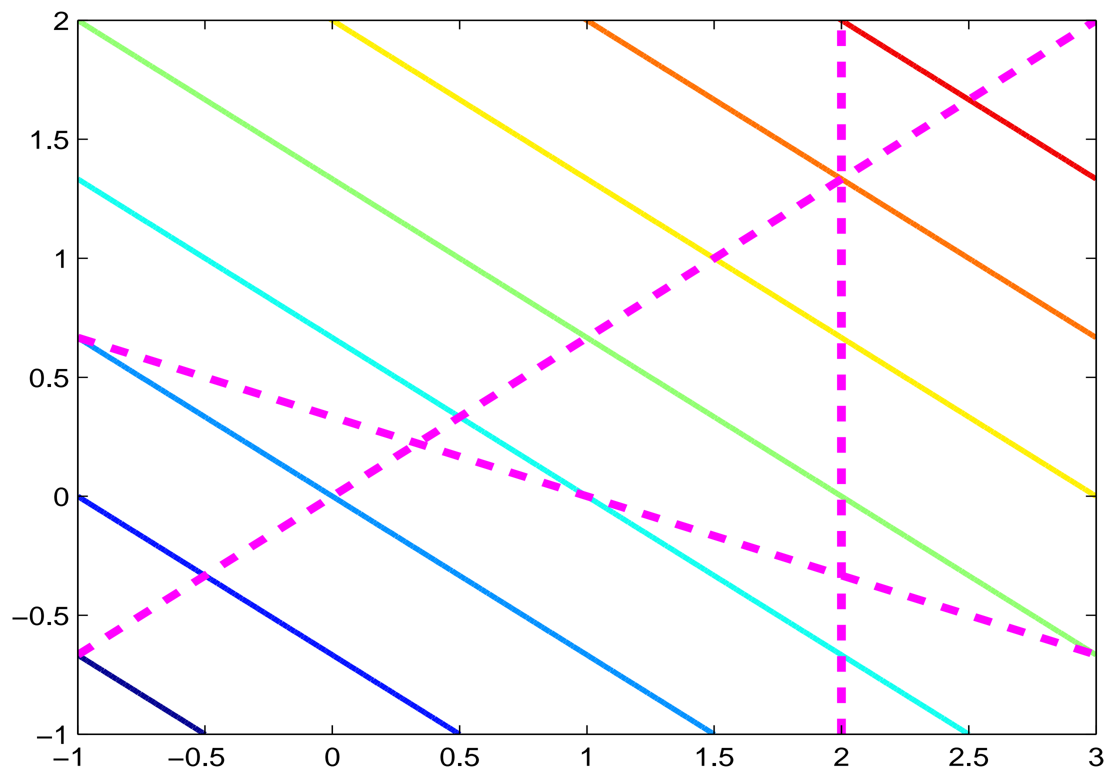
On the bright side, mathematical analysis can sometimes reveal that the solution will not be what you want.

In the breakfast problem as given, we would know in advance (from the mathematical nature of an LP solution) that the "optimal" breakfast would contain only two foods—a big clue that we should have more constraints.

What's the lesson about hardness?

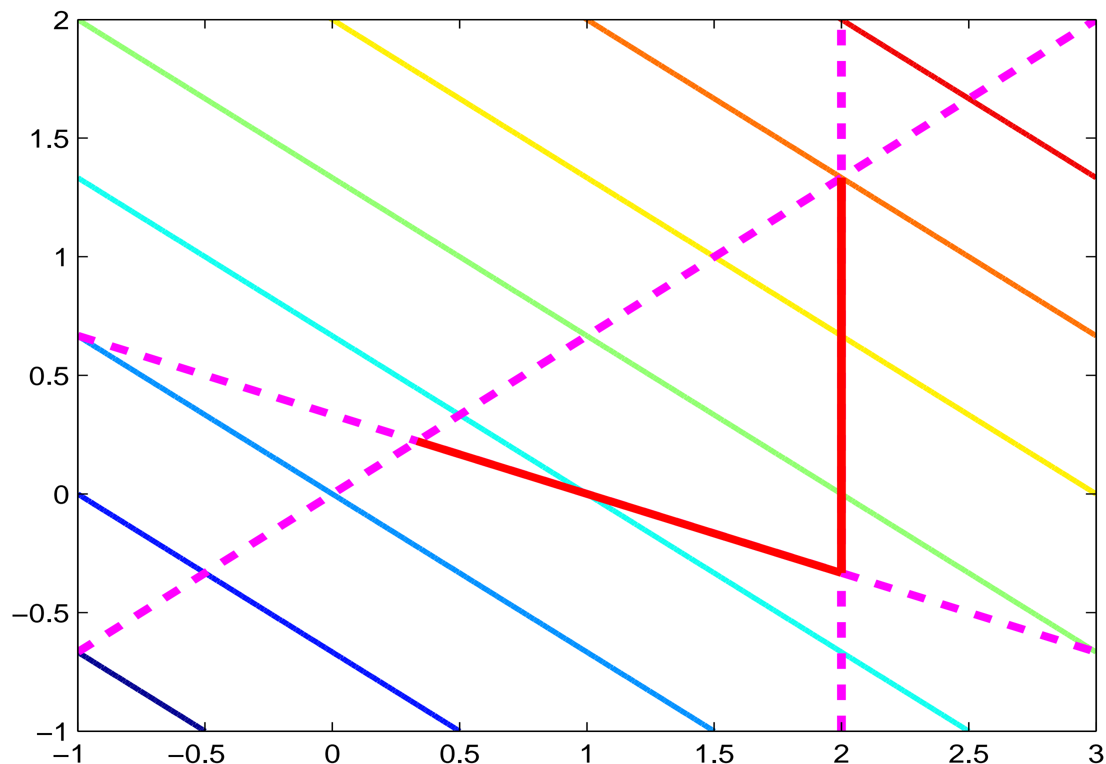
The hardest part can be getting the problem right.

How could one solve an LP??



Proof by picture: the optimal solution (essentially always) lies at a *vertex*, i.e., a “corner point”.

GBD's great invention was the *simplex method*, which solves linear programs by starting at a vertex and moving from vertex to vertex, reducing the *objective function* (the function we want to minimize) as it goes.



So how hard can it be to solve an LP with the simplex method??

Hmmmm. . . a worrying feature: the number of vertices in an LP can be *exponential* in the problem dimension.

But for the first 25 years after its invention, practitioners happily used the simplex method to solve increasingly large linear programs.

Why were they happy? The simplex method *almost always* took $2n$–$3n$ steps to solve the problem, where n is the number of variables, and each step took $O(n^2)$ operations.

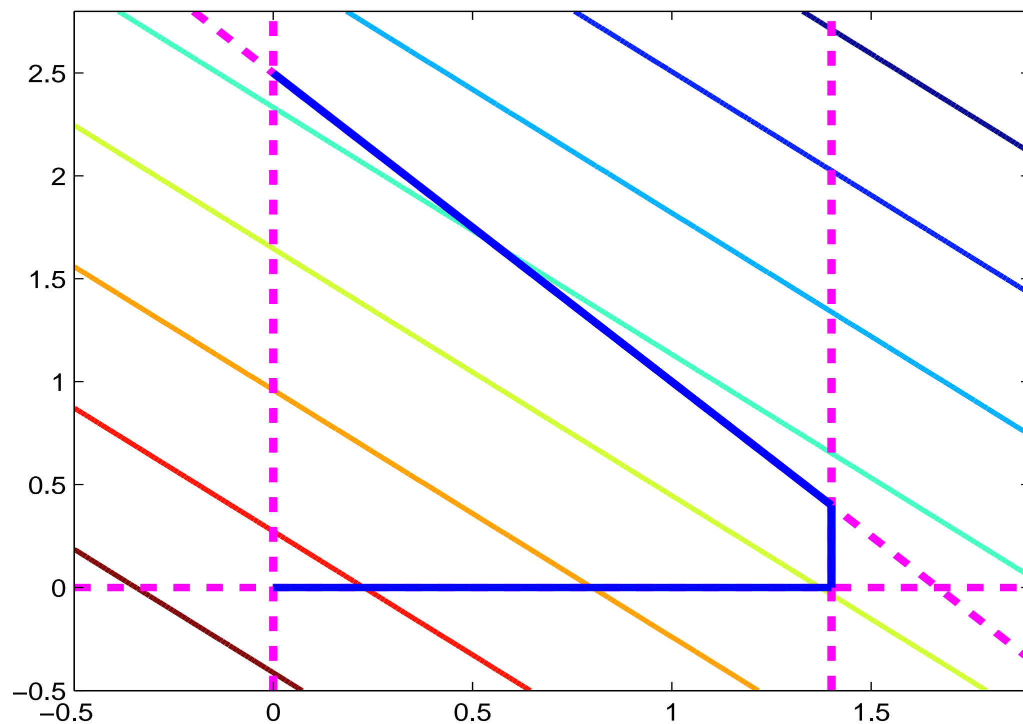
Thus, *in practice*, the simplex method was polynomial-time.

But *from a mathematical perspective*, success in practice did not prove anything!

A great (or terrible) thing: mathematicians and computer scientists are *really*, *really* good at finding nasty problems that cause algorithms to perform badly.

In 1972, Klee and Minty produced a linear program with n variables that has 2^n vertices, and for which the textbook simplex method, if started at the "wrong" vertex, visits every one of those 2^n vertices.

Klee-Minty in two dimensions



Just as with the king and jester, a two-dimensional example doesn't come close to conveying how bad this result is for the simplex method!

Klee-Minty showed that the simplex method is, without any doubt, an *exponential-time* method in the worst case.

However, since LPs seemed to be “easy” to solve in practice, this left an open question of whether there was a polynomial-time algorithm for linear programming.

Two breakthroughs about the hardness of linear programming:

1979: Leonid Khachiyan, a 28-year-old mathematician in the Soviet Union, defined a *polynomial-time* method for linear programming, so LP is in P.

Nonetheless, in practice Khachiyan's algorithm is MUCH, MUCH slower than the simplex method.

1984: Narendra Karmarkar, a 28-year-old mathematician at Bell Labs, defined a polynomial-time LP method claimed to be 50 times faster than the simplex method.

1985: Karmarkar's algorithm was shown to be formally equivalent to a class of methods for nonlinear optimization from the 1960s, and to be efficient in practice, leading to an explosion of interest in "interior-point methods" for optimization.

Still, an anomaly remains after 20 years of intense research.

Interior-point methods for LP, albeit polynomial-time, can be much slower than simplex, even on enormous problems (e.g., with tens or hundreds of millions of variables).

How and why can this be true?

Very recent theoretical work by Spielman and Teng shows that the simplex method is polynomial-time in the *average case*, but in practice the simplex method is almost always faster than their analysis shows.

How and why can this be true?

We do NOT have all the answers.

Summary...

Cryptography: There is every reason to believe that factoring is hard, and cryptographic systems based on this belief have flourished (as far as we know...). But the hardness of factoring (and other NP-problems) has *not* been proved.

Real-world cryptography is vastly more complicated than the simplified version in this talk.

Linear programming: Algorithmic improvements in both simplex and interior-point methods, plus faster computers, mean that huge linear programs can be solved very quickly.

What is it about some LPs that makes the simplex method more effective than a polynomial-time method? And why are both classes of methods so often more efficient than their formal complexity analyses suggest?

In both cryptography and linear programming, problems can be much easier to solve if they have special characteristics. (Remember Gauss.)

Mathematics leads to fascinating insights about how hard problems can be and how to solve them.

Today we are sooooo lucky to have an endless supply of new problems to work on while we figure out

How hard can it be?

Acknowledgements:

Above all, Google and Google Images

RSA Labs Web site

Wikipedia

Ian Stewart (jigsaw puzzle example)