CSI 4103 / 5138: Great Algorithms **Lecture 1**
University of Ottawa, Fall 2024

> Last year, Alice was three times as old as Bob and Charlie together. This year she is twice as old, and next year Charlie will be five times younger than Alice and Bob together. How old are they?

Solution: Writing x, y, and z for their ages, we are told that

$$\begin{aligned} x - 1 &= 3((y-1) + (z-1)) \quad \text{and} \\ x &= 2(y+z) \quad \text{and} \\ (x+1) + (y+1) &= 5(z+1). \end{aligned}$$

and all we have to do is solve for x, y, and z. To do this, first we group unknowns on one side and constants on the other:

$$\begin{aligned} x - 3y - 3z &= -5 \\ x - 2y - 2z &= 0 \\ x + y - 5z &= 3 \end{aligned} \tag{1}$$

Then we eliminate x by subtracting the last two equations from the first:

$$\begin{aligned} -y - z &= -5 \\ -4y + 2z &= -8 \end{aligned}$$

We have now reduced the problem to one with fewer unknowns. Eliminating y gives $-6z = -12$ so $z = 2$. Plugging in $-y - z = -5$ we get that $y = 3$, and plugging into say $x + y - 5z = 3$ gives $x = 10$, producing the solution

$$x = 10 \qquad y = 3 \qquad z = 2. \tag{2}$$

This algorithm is called Gaussian elimination. It can be applied to any system with the same number of equations and unknowns. Here is how it works in general:

Algorithm *GE*
Input: A system S of n linear equations with n unknowns.

```
If $n = 1$ and the equation is $ax = b$, output $b/a$.
Otherwise, pick a variable $x$ and an equation $e$ in $S$ in which $x$ has nonzero coefficient (the pivot).
Create a new system $S'$ of $n - 1$ equations and $n - 1$ unknowns:
For every equation $e' \in S$ other than $e$,
    Subtract enough copies of $e$ from $e'$ to eliminate $x$ and
    include the resulting equation in $S'$.
Apply $GE$ to $S'$ to obtain an assignment to all variables except for $x$.
Plug this partial assignment into $e$ to recover $x$.
Output the completed assignment.
```

That was a mouthful. Gaussian Elimination is complicated!

1 It is the algorithm (stupid)

Algorithms power the modern world. They route trains and planes and detect defects in nuclear reactors. They choose which news articles you read and which songs you listen to. They solve your homework, sometimes incorrectly. They decide whether your job application will end up in the digital dustbin or on a recruiter's desk.

We computer science students are taught a great deal about algorithms. We learn how to make them run (programming), how to craft them and compare them (design and analysis of algorithms), how to choose good representations (data structures), how to debug them (software engineering), and how to train them (statistics and machine learning). In these courses we are led to think of algorithms as ingenious products of (human) engineering. Should you want a shortest path, Dijkstra is there to help you. Need to build a spam filter? Train a naive Bayes classifier on your emails.

This type of knowledge is powerful for building problem-solving confidence. I find it less useful in understanding what algorithms can realistically accomplish, when we should trust them and when we shouldn't, and why minor changes in the input can sometimes wreak complete algorithmic havoc. To begin thinking about questions like these we need to look into the "soul" of the algorithm.

Every week will examine one of these great algorithms and try to answer questions like:

1. Why does it work? Why does Gaussian elimination correctly solve systems of linear equations? Does it ever fail? Why would it fail? Can its failures be patched or at least detected?

2. What makes it attractive? One reason is efficiency. Everyone likes fast algorithms. Another reason could be robustness. The actual input to the algorithm might be noisy. For example Gaussian elimination is supposed to solve linear equations. How would it react if the "input data" is not exactly linear?

3. Why is it useful? Where does solving linear equations come up? To tackle this question a "reductionist" perspective is often helpful. A problem might not look like a system of linear equations—for example it could be about finding components in a graph, or solving quadratic equations—but can be *reduced* to it by picking a good representation. This may lead us to revisit the first two questions: Why does the reduction work? How much complexity does it add to the algorithm? Is it robust?

4. Can we apply the algorithm, possibly after some tinkering, in a context different from our original motivation? Once we have a new "hammer" (the algorithm) we humans are quite adept at seeing "nails" (problems to attack) everywhere.

Why does Gaussian Elimination work? To get a sense of an algorithm it helps to test it on a few examples. We already tried it once on instance (1). We can confirm that it works on this instance because we plug in its output (2) into (1) the left and right sides are consistent. Now let's try it on another example:

> Last year, Alice was three times as old as Bob and Charlie together. This year she is twice as old, and in five years' time she will be exactly as old. How old are they?

The new equations are $(x-1) = 3((y-1)+(z-1)), x = 2(y+z), x+5 = (y+5)+(z+5)$, or

$$\begin{aligned} x - 3y - 3z &= -5 \\ x - 2y - 2z &= 0 \\ x - y - z &= 5 \end{aligned} \tag{3}$$

Let's now run Gaussian elimination. After pivoting on x in the first equation we reduce to the system

$$\begin{aligned} -y - z &= -5 \\ -2y - 2z &= -10. \end{aligned} \tag{4}$$

If we now pivot on y in the first equation we reduce to

$$0 = 0.$$

This type of input is not expected by the algorithm. The code doesn't tell us what to do if there are *no* variables. As described, Gaussian Elimination will fail on this example, even though $x = 10, y = 3, z = 2$ is a perfectly valid solution. So is $x = 10, y = 4, z = 1$. There are infinitely many solutions, all of the form

$$x = -5 + 3y + 3z = -10 + 6z, \qquad y = -5 + z \qquad z = \text{any value}. \tag{5}$$

If you know linear algebra you can identify the source of the problem: Equations (4) are *linearly dependent*. Linear dependencies cause some difficulties for Gaussian Elimination. As we will see, these difficulties are surmountable.

2 Problems and algorithms

What do we mean when we say "Gaussian elimination works"? The question presupposes a distinction between the problem (solving linear equations) and the algorithm (Gaussian Elimination). To clarify this distinction it helps to abstract what we mean by a problem and what we mean by an algorithm. This is a common source of confusion.

For our purposes a *computational problem* is a relation R on instance-solution pairs. For example, the problem "square an integer" *is* the relation

$$SQ = \{(x, x^2) : x \text{ is an integer}\} = \{(0, 0), (1, 1), (-1, 1), (2, 4), (-2, 4), \dots\}.$$

The problem "find an integer square root" is the transpose relation

$$SQRT = \{(x^2, x) : x \text{ is an integer}\} = \{(0, 0), (1, 1), (1, -1), (4, 2), (4, -2), \dots\}.$$

An algorithm is a procedure that takes an instance x and produces solution(s) y such that $(x, y) \in R$. For example an algorithm for SQ is supposed to output 1 on input -1 and 2 on input 4. In the case of $SQRT$ there are sometimes multiple solutions (both -2 and 2 are square roots of 4) and sometimes no solutions (3 has no square root). What should the algorithm do?

The answer to this determines the type of algorithm we want: A *search* algorithm is required to output any y of its choice as long as $(x, y) \in R$. A *decision* algorithm only needs to answer if such a y exists (a yes/no question). An *enumeration* algorithm should output all possible y (in a suitable representation). Later in the course we will also talk about *optimization* algorithms (output the best y according to some criterion) and possibly *sampling* algorithms (sample a random y conditioned on $(x, y) \in R$).

For the types of problems we will be looking at, once a solution has been found, it is fairly easy to verify that it is legitimate. For example if your algorithm for taking square roots takes input 15376 and produces output answer 124, you can easily verify that it is correct because $124^2 = 15376$. In contrast if it had output 125 you would have calculated $125^2 = 15625$ and concluded that the algorithm is incorrect. Similarly, if Gaussian Elimination returns a candidate solution to some system S of linear equation you can plug in the solution and verify that it is legitimate. For such problems, decision is no harder than search, and search is no harder than enumeration.

The problem "solving linear equations" consists of those instance-solution pairs (S, sol), where S is a system of linear equations like (1), and sol is an assignment like (2) that satisfies all equations simultaneously. The decision problem is to determine whether S has any solutions, the search problem is to find a solution if one exists, and the enumeration problem is to list all possible solutions. One problem is that the set of all possible solutions can be infinite as in example (3). We will need to think of a reasonable compact representation of this infinite set.

3 Correctness

Let us begin with a less ambitious objective. Algorithm GE might have its issues, but we do expect it to work on instances with a *unique* solution (i.e., the problem $\{(S, sol) : sol \text{ is the unique solution to } S\}$). Why is that?

The heart of the GE algorithm is in lines 5-6. This procedure of subtracting multiples of one row from another is called an *elementary row operation*. The key property of elementary row operations is that they are invertible. Therefore they preserve the solution space. For example, the systems

$$\begin{aligned} x - 3y - 3z &= -5 \\ x - 2y - 2z &= 0 \\ x + y - 5z &= 3 \end{aligned} \qquad \begin{aligned} x - 3y - 3z &= -5 \\ -y - z &= -5 \\ x + y - 5z &= 3 \end{aligned} \qquad \begin{aligned} x - 3y - 3z &= -5 \\ -y - z &= -5 \\ -4y + 2z &= -8 \end{aligned}$$

all have the same solution space consisting of the unique assignment $x = 10, y = 3, z = 2$.

Claim 1. *Let S be a system of linear equations and e be an equation in S. Let T be obtained from S by adding a multiple of e to some other equation e'. Then T has the same solutions as S.*

The reason is that being a solution is closed under taking linear combinations of the equations, for example:

$$-y - z = (x - 3y - 3z) - (x - 2y - 2z) = -5 - 0 = -5,$$

Therefore, any common solution to $x - 3y - 3z = -5$ and $x - 2y - 2z = 0$ must also solve $-y - z = -5$. Conversely, any solution to $-y - z = -5$ and $x - 3y - 3z = -5$ must also solve $x - 2y - 2z = 0$ because

$$x - 2y - 2z = (-y - z) + (x - 3y - 3z).$$

Proof. Take any solution a to S. Then a solves all the equations of T except possibly the one, let's call is $e^\star$, that was obtained by adding a multiple of e to e'. As a solves both e and e' and $e^\star$ is a linear combination of e, e', a solves $e^\star$.

Conversely, if a is a solution of T, it solves all equations of S except possibly e'. As $e^\star$ was obtained by adding a multiple of e to e', e' can be obtained from e by subtracting the same multiple of $e^\star$. Therefore $e^\star$ is a linear combination of e and e'. It must also be satisfied by a. □

Proposition 2. *Assuming S has a unique solution, algorithm GE applied to S finds it.*

Proof. We apply induction on n. When $n = 1$ the system has the form $ax = b$. If $a = b = 0$ the system would have infinitely many solutions (any value of x would solve it), and if $a = 0$ but $b \neq 0$ it would have no solutions. Therefore a must be nonzero and step 1 outputs the unique solution $x = b/a$.

Now assume GE solves all systems with $n - 1$ variables and equations that have a unique solution. Assume also the input S to GE has a unique solution. As $S' \cup e$ was obtained from S by elementary row operations, by Claim 1 they must have the same solutions. In particular, S' by itself must have at least one solution. If it had two (or more), these could be extended to two (or more) solutions to $S' \cup e$, and therefore S, by solving for x in e. So S' must have exactly one solution. By inductive hypothesis, the recursive call $GE(S')$ in line 7 finds this solution. Line 8 extends this to a valid solution of $S' \cup e$ and therefore also S. □

Armed with this success we can now tackle systems like (3) that might not have a unique solution. In that example $x = 10, y = -5 + z$ is a solution for any choice of z, and all the solutions have this form. To account for this type of example, we could ask of the algorithm to output some set $FREE$ of *free variables* and an assignment to the remaining variables in terms of the free variables. Under this convention taking $FREE = \{z\}$ and x, y as in (5) is a valid solution to (3). Another valid solution is $FREE = \{y\}, z = 5 + y, x = 10$.

Now that we have set expectations, let's understand why algorithm GE fails to meet them. When attempting GE on system (3), the algorithm derived the equation $0 = 0$ before "using up" all its variables (variable z in that example). As $0 = 0$ is a tautology, this suggests z should be allowed to take any value and the algorithm can set it free. It seems sensible to append the following line of code to GE:

If S has the form $0 = 0$ but there are still unused variables, set all of them free.

Where in the code of GE should this line be inserted? To answer this question we should "debug" GE on the problematic example (3) and see where it crashed. The point of failure is line 2: We are asked to pick an x, but there is no x to be picked in the system $0 = 0$! The modification should be made before line 2.

There is one more type of system we need to handle: A system with no solution like

$$\begin{aligned} x - 3y - 3z &= -5 \\ x - 2y - 2z &= 1 \\ x - y - z &= 5 \end{aligned} \tag{6}$$

Let's now run Gaussian elimination. Pivoting on x in the first equation gives

$$\begin{aligned} -y - z &= -6 \\ -2y - 2z &= -10. \end{aligned}$$

Now pivoting on y gives $0 = -2$, a contradiction! In this case GE should declare that there is no solution:

> If S has the form $0 = b$ for nonzero b, output `no solution`.

There is still one minor "bug" in line 1 of GE: even in the base case $n = 1$ we have to handle the possibility of the coefficient a being zero. As this was already addressed in the modifications, it is sensible to fold the base case in there as well. The resulting algorithm is attributed to Gauss and Jordan.

Algorithm GJE (Gauss-Jordan elimination)
Input: A system S of m linear equations with n unknowns.

```
1   If $S$ has the form $0 = 0$ or is empty, output the assignment in which all variables are free.
    If $S$ has the form $0 = b$ for $b \neq 0$, output no solution.
2   Otherwise, pick a variable $x$ and an equation $e$ in $S$ in which $x$ has nonzero coefficient.
3   Create a new system $S'$ of $m - 1$ equations and $n - 1$ unknowns:
4   For every equation $e' \in S$ other than $e$,
5       Subtract enough copies of $e$ from $e'$ to eliminate $x$ and
6       include the resulting equation in $S'$.
7   Apply $GE$ to $S'$ to obtain an assignment to all variables except for $x$.
8   Plug this partial assignment into $e$ to recover $x$ in terms of the free variables.
9   Output the completed assignment.
```

The number of equations and the number of variables no longer have to be the same.

Theorem 3. *GJE outputs a description of all the solutions of S if there are any, and* `no solution` *if there aren't.*

The theorem is proved similarly to Proposition 2 (using Claim 1).

4 Features and limitations

How efficient is Gaussian elimination on m equations with n unknowns? In the worst case it applies $m-1$ elementary row operations in step 4-5, then runs recursively on an instance with $m-1$ equations and $n-1$ unknowns, and completes the assignment in step 9. Step 5 consists of $O(n)$ additions or divisions, so each execution of the loop 4 entails $O(mn)$ operations. Step 8 involves at most $O(n^2)$ operations (there are at most $n-1$ free variables and a constant, and computing the coefficient of each takes $O(n)$ operations). The number of operations $C(m, n)$ then satisfies the recurrence

$$C(m, n) = C(m - 1, n - 1) + O(mn + n^2).$$

When $m = 0$ (no equations) or $n = 0$ (no variables), the complexity is $O(1)$, from where

$$C(m, n) = O(n(m + n) \min\{m, n\}).$$

When $m = n$, the algorithm takes cubic time in the number of variables. In general, it takes $mn + m$ numbers to describe a system with m equations and n unknowns so the worst-case complexity is at most the 1.5th power of the input size.

Some linear systems that arise in applications are sparse: The number of variables that participates in a typical equation is much smaller than n. If, say, every equation contains at most 5 variables then the system can be specified using the $5(m + 1) = O(m)$ coefficients (plus a bit of extra information that links each coefficient with an equation and a variable). For such sparse systems the worst-case running time of Gaussian Elimination is cubic in the size of the instance, which can be taxing on large instances (say $m = 10000$ which is quite small in modern applications!)

As this cubic upper bound is worst case, one may ask if it is attained in "typical" applications. Making precise mathematical sense of this question would lead us too far afield so let me try to argue informally that the answer is yes. The reason is that elementary row operations are not "friendly" to sparse equations. If e and e' have 5 variables each then adding copies of e to e' can at worst double the number of variables to 10. Typically we might expect the number of variables in e' to almost double (this can be made more precise in a probabilistic model of a sparse system). The system becomes a bit more dense after each elimination step. Quantifying this increase in density is tricky but I believe that after half the variables are eliminated the system becomes very dense: A typical equation will contain a constant fraction of the remaining variables, resulting in asymptotically cubic overall complexity. (The complexity of Gaussian Elimination for sparse linear systems could be an excellent course project.)

A nice feature of Gaussian Elimination is that it allows for an in-place implementation. The space used to describe the instance can be reused to describe all intermediate states and the solution. This is usually how the algorithm is described in linear algebra textbooks: A (augmented) matrix specifying the system is set up, and at the end the solution appears in place of the right-hand side.

An important property of Gaussian Elimination is that it can be implemented with perfect precision. If the input coefficients are provided as rational numbers and the additions and divisions in the elementary row operations are implemented without loss of accuracy, the algorithm will output an exact solution. (The numerators and denominators will typically grow during an execution, however, resulting in "bit complexity" that is higher than cubic.) Gaussian Elimination is in essence an *algebraic* algorithm that only relies on basic arithmetic $(+, -, \times, \div)$. It works in any "number system" with these operations. We will see shortly that the algebraic nature of Gaussian Elimination is a feature in some contexts but a limitation in others.

5 Certifying unsatisfiability

Suppose you implemented Gaussian Elimination but you are not quite sure that your code works correctly. You run it on system S and it outputs `no solution`. Can you trust this output? If it did output a solution, you could have plugged it in and verified that it works. Is there anything you can do to verify that S has no solution?

Let's look at example (6) again. What happens if I add up the first and third equation and subtract the second equation twice? The left-hand side is

$$(x-3y-3z)-2(x-2y-2z)+(x-y-z) = (1-2+1)x+(-3+4-1)y+(-3+4-1)z = 0x+0y+0z = 0$$

no matter what x, y, and z are set to. But the right-hand side is $-5 - 2 \cdot 1 + 5 = -2$. Thus 0 would equal to -2, which is clearly impossible. The only possible explanation is that x, y, and z solving this system cannot exist!

Let's abstract this reasoning. We argued that a system cannot have a solution by coming up with some linear combination of the equations that makes the left-hand side vanish but not the right-hand side.

Let's call such a linear combination of the equations *contradictory*. The existence of a contradictory linear combination is one reason that a system is unsolvable. Could there be others? Amazingly, no; this trick always works!

Theorem 4. *A system of linear equations is unsatisfiable if and only if there exists a contradictory linear combination of the equations.*

The reason this theorem holds is that elementary row operations preserve the existence of contradictory linear combinations. So if the theorem holds for the "base cases" (line 1 in *GJE*) then it must hold for all systems as they are reduced to these base cases via elementary row operations. We used similar logic to prove Proposition 2.

Theorem 4 posits the existence of contradictory linear combinations for unsatisfiable linear systems, but how do we go about finding one? One possibility is to augment algorithm *GJE* so that it "tracks" such combinations. This turns out not to be necessary.

How did I come up with the contradictory linear combination for example (6)? I did not know ahead of time that I needed to take 1, −2, and 1 copies of the first, second, and third equations, respectively. I had set up unknowns a, b, and c for these three numbers. What I wanted to happen is for the left-hand side to vanish for some choice of a, b, c, namely

$$a(x - 3y - 3z) + b(x - 2y - 2z) + c(x - y - z) = 0 \qquad \text{for all } x,\ y,\ z.$$

This is only possible if all coefficients in front of x, y, and z vanish simultaneously, namely

$$\begin{aligned} a + b + c &= 0 \quad (x\text{-coefficient}) \\ -3a - 2b - c &= 0 \quad (y\text{-coefficient}) \\ -3a - 2b - c &= 0 \quad (z\text{-coefficient}), \end{aligned} \tag{7}$$

but the right-hand side does not vanish

$$-5a + b + 5c \neq 0.$$

This is almost like a system of linear equations, except that there is one inequality. If this system had a solution, then I can obtain any number I want on the right-hand side of the inequality by scaling a, b, and c simultaneously. Therefore the inequality is "equivalent" to the equality

$$-5a + b + 5c = 1. \tag{8}$$

I now have a linear system (7-8) whose solutions are the contradictory linear combinations for system (6). Using Gaussian Elimination, I solve (7-8) to obtain $a = -\frac{1}{2}, b = 1, c = -\frac{1}{2}$. Indeed, linearly combining equations (6) with coefficients a, b, c produces the contradiction $0 = 1$.

In conclusion, the problem of finding a contradictory linear combination for a linear system reduces to the problem of solving linear equations and can therefore be solved by Gaussian Elimination. Thus Gaussian Elimination can *certify* that a system has no solution.

At this point it is useful to pass to matrix notation, in which (6) is represented as

$$\begin{bmatrix} 1 & -3 & -3 \\ 1 & -2 & -2 \\ 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -5 \\ 1 \\ 5 \end{bmatrix}.$$

To find a contradictory linear combination, we look for a row vector such that

$$\begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} 1 & -3 & -3 \\ 1 & -2 & -2 \\ 1 & -1 & -1 \end{bmatrix} = 0 \quad \text{and} \quad \begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} -5 \\ 1 \\ 5 \end{bmatrix} = 1.$$

In this notation, Theorem 4 says (with $\mathbf{x}, \mathbf{b}$ column vectors and $\mathbf{y}$ a row vector):

Theorem 4'. $A\mathbf{x} = \mathbf{b}$ *has no solution if and only* $\mathbf{y}A = 0, \mathbf{y}\mathbf{b} = 1$ *has a solution.*

6 Learning linear functions

The $f$inal grade formula for Great Algorithms is 30% $h$omework, 30% $m$idterm, and 40% $p$roject. Suppose you did not know this formula and would like to learn it. You have records of students who took the course:

	h	m	p	f
Alice	87.78	96.67	80.00	87.33
Bob	100.83	100.00	95.00	98.25
Charlie	84.44	76.67	80.00	80.33
Dave	48.89	86.67	72.50	69.67
Eve	84.72	96.67	85.00	88.42

This looks like a good setup for Gaussian elimination. All you need to do is solve the linear system

$$\begin{bmatrix} 87.78 & 96.67 & 80.00 \\ 100.83 & 100.00 & 95.00 \\ 84.44 & 76.67 & 80.00 \\ 48.89 & 86.67 & 72.50 \\ 84.72 & 96.67 & 85.00 \end{bmatrix} \begin{bmatrix} h \\ m \\ p \end{bmatrix} = \begin{bmatrix} 87.33 \\ 98.25 \\ 80.33 \\ 69.67 \\ 88.42 \end{bmatrix}.$$

Gaussian elimination says there is no solution! How is this possible?

The reason is that the recorded grades have been rounded to two digits of precision. Even though the rounding error is tiny it destroys the solution. If the records were perfect, the right-hand side would have been linearly dependent on the left. Rounding is a non-linear operation. It destroys not only the solution but the linear dependence itself, resulting in a system with no solutions (see Figure 1).

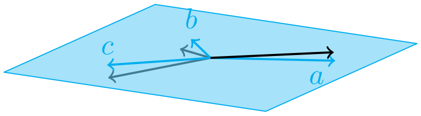

FIGURE 1: Vectors a, b, c (in blue) are linearly dependent. After rounding them to the black vectors the linear dependence is destroyed.

As there are only three unknowns that we want to solve for, we can hope that some three of the five records should be sufficient, as long as they happen to be linearly independent. Indeed, running Gaussian Elimination on the first three rows of the data matrix gives the solution $h \approx 0.294, m \approx 0.301, p \approx 0.405$. This is a pretty good approximation. Grading weights are usually assigned in multiples of 5%; if we round to the closest such multiple we recover the actual scheme.

The uOttawa administration asks that grades be rounded to the closest integer, so it may be more realistic to expect that you only have access to coarsely rounded records like

	h	m	p	f
Alice	88	97	80	87
Bob	101	100	95	98
Charlie	84	77	80	80
Dave	49	87	73	70
Eve	85	97	85	88

Apply Gaussian Elimination to the first three rows now gives $h \approx -.006, m \approx .351, p \approx .669$. This is very far from the ground truth. Gaussian Elimination is very brittle to rounding and other types of errors.

7 Modular Gaussian Elimination

A common setup in machine learning is that we have example input-output pairs $(x, f(x))$ for some unknown function f and we want to "learn" f. In the example we just saw x are the homework, midterm, and project grades, and $f(x)$ is the final grade.

To make sense of this question it is common to impose some restriction on what f should look like. In the grading example we assumed that f is a linear function and reduced the problem to learning its coefficients h, m, and p by solving a suitable linear system.

Let us now move to the other extreme and assume that f is as "nonlinear" as it can be. Here is a setup that may appear contrived but turns out to be quite enlightening. The inputs x are rows of n numbers x_1 up to x_n, each of which is equal to $+1$ or -1. The output $f(x)$ is the product of some unknown subset of these numbers, for example

x_1	x_2	x_3	x_4	$f(x)$
-1	$+1$	$+1$	$+1$	$+1$
$+1$	-1	-1	$+1$	-1
-1	-1	$+1$	-1	$+1$
-1	-1	$+1$	$+1$	-1
$+1$	-1	$+1$	$+1$	-1

Here $f(x)$ happens to be the product of x_2 and x_4. The problem of *learning hidden parities* is to figure out which subset of numbers multiplies to $f(x)$.

One candidate algorithm is to test out all possible subsets: The empty set (which always multiplies to $+1$), the singletons x_1, x_2, x_3, x_4, all pairs $x_1x_2, x_1x_3, \dots, x_3x_4$, and so on. This works well when n is small but becomes intractable rather quickly because the number of possible subsets is 2^n. When n equals 80 it is quite hopeless.

Gaussian Elimination happens to be the perfect algorithm for learning hidden parities. All we need to do is solve the a system of *modular* linear equations:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \pmod 2$$

obtained by replacing $+1$ and -1 from the data table with the values 0 and 1 modulo 2, respectively, representing the parity of the number of minus signs.

Gaussian elimination works perfectly well in arithmetic modulo 2. In this example it outputs the unique solution $b_1 = 0$, $b_2 = 1$, $b_3 = 0$, $b_4 = 1$, indicating that $f(x)$ depends on x_2 and x_4 but not on x_1 and x_3. In general, the hidden subset consists of those x_i for which b_i equals 1—in short, $f(x) = x_1^{b_1} \cdots x_n^{b_n}$.

Until 2012, Gaussian Elimination was the only known algorithm for solving linear equations modulo two. That year Prasad Raghavendra came up with an ingenious alternative. (Comparing Raghavendra's algorithm to modular Gaussian Elimination could be another project.)

A number system in which you can add, subtract, multiply, and divide (except by zero) is called a field. Examples you should be familiar with are rational numbers and arithmetic modulo a prime. Another important class of examples are finite extension fields which we might see later in the course. Gaussian Elimination works well over fields and all the theorems we proved in this lecture are true for them. In arithmetic over non-prime modulus, pivots that are not invertible (like 4 modulo 6) are problematic. As long as such pivots can be avoided, Gaussian Elimination is correct.