Rings in CS

SIG Math Fall 2025

What are rings?

- A few weeks ago, we talked about semirings (in two presentations actually). Properties of semirings include:
  - Closed under addition and multiplication
  - Addition is commutative (and sometimes multiplication)
  - Addition and multiplication are associative
  - 1 and 0 exist and work as you expect (additive and multiplicative identities)
- Rings are very similar to semirings, but have one extra capability: **all values have an additive inverse** (except 0)
- Some examples of rings include: integers, rational numbers, real numbers, clock numbers, matrices, functions, polynomials.

A few weeks ago, we talked about semirings (in two presentations actually).

The power of abstraction

- Because ring theory is so general, if you can prove a theorem or that an algorithm works for a ring, that means it works *for any ring*.
- The Floyd-Warshall algorithm is a famous algorithm from computer science. By changing what semiring you use, it can solve many different problems:
  - Shortest path (min-plus tropical semiring)
  - Longest path (max-plus tropical semiring)
  - Highest probability (max-times tropical semiring)
  - Reachability (boolean semiring)
  - Number of paths
  - Maximum flow
  - Parsing/grammar problems

Operations on rings

- Rings are an abstract concept, and oftentimes mathematicians study even more complex objects such as ideals or quotients.
- Consider a clock. A swagtastic clock has the numbers 0 through 11, and once you go past 11 you go back around to 0.
- This is just the integers mod 12! Suppose it's 4am right now, and you want the time in 53 hours. It will be (4 + 53) % 12 = 9am.
- We can also perform multiplication. Say it's midnight and we do 6 5-hour long Silksong speedruns. Then the time is (5 * 6) % 12 = 6am.
- So the "clock numbers" form a ring! Specifically, it's the same as the integers modulo 12. We write this as Z/12Z.

Rings in physics

- In Hamiltonian mechanics (which applies to classical and quantum physics), observables are defined as functions over the phase space.
  - Functions can be added or multiplied in the usual sense, and there are infinitely many of them. Analyzing these functions requires deep knowledge of ring theory!
- Many physical systems are also classified by symmetry, such as rotations, Lorentz transformations, and gauge symmetries.
  - The tool for studying symmetries is group theory. However, groups are heavily connected to modules over rings. When you classify particles by spin or charge, you are studying modules over a ring!

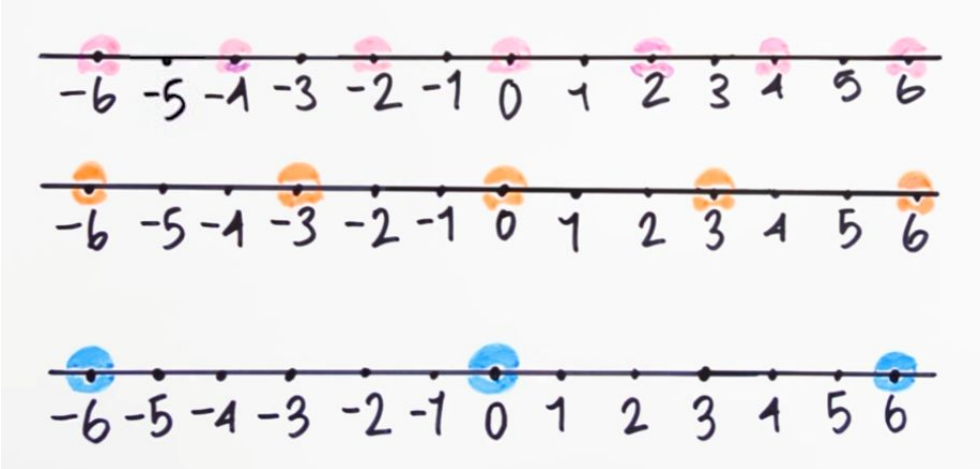
Topic: Algebraic Number Theory

Factoring in rings

- Not all rings work the same as the integers. One of the key properties of the integers is that **any integer can be separated into a unique prime factorization** (the Fundamental Theorem of Arithmetic)
- Now let's consider the complex integers a+bi (denoted Z[i]). Do they have unique prime factorizations?
- The answer is no: 5 can be factored 5 * 1, or (2-i)(2+i).
- This suggests that maybe factoring into prime numbers is not the right way to look at things from a ring-theoretic perspective.
- The solution is that instead of factoring into numbers, we **factor into ideals**. This is the starting point of **algebraic number theory**.

Ideals

- Take the equation 2 * 3 = 6. The idea is that instead of looking at just 2, 3, and 6, we look at **all multiples of those numbers**.
- All multiples of 2 are denoted in pink, multiples of 3 are in orange. If you take all their products, you get the numbers highlighted in blue, which are the multiples of 6!
- These subsets are known as **Ideals**. Instead of factoring into prime numbers, we will **factor into prime ideals**.

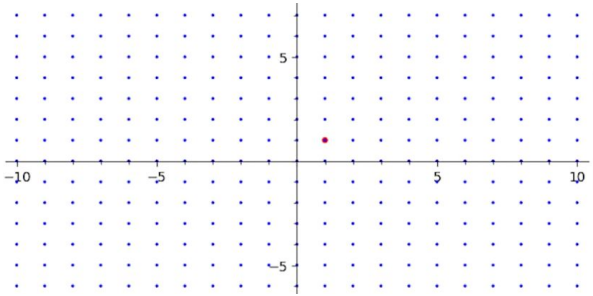


Properties of ideals

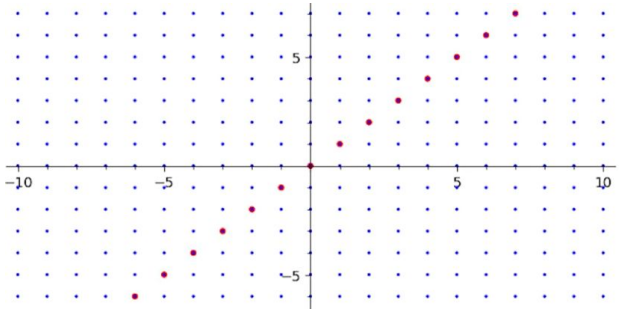
- The definition for ideals is somewhat complicated, but I wanted to write them down.
- Given a ring R, an ideal I is a subset of R such that:
  - I contains 0.
  - I is closed under addition and multiplication (adding or multiplying 2 things gives another thing)
  - If x is an element of I, then x times any element of R is in I.
- Suppose we generate an ideal from one element, such as all linear combinations of 2 (the pink numbers in the last slide). We write that ideal as (2).
- This can be hard to get used to, so let's see an example.

Complex ideals

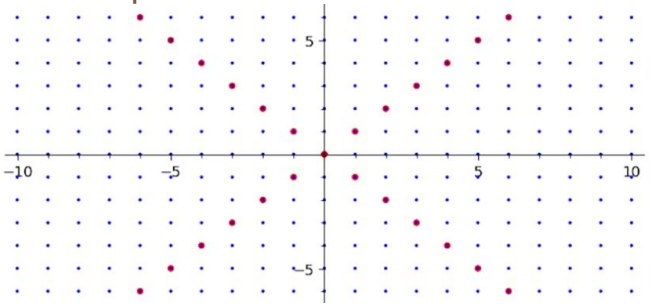
- Now we move to the complex plane. Take the ideal (1+i).



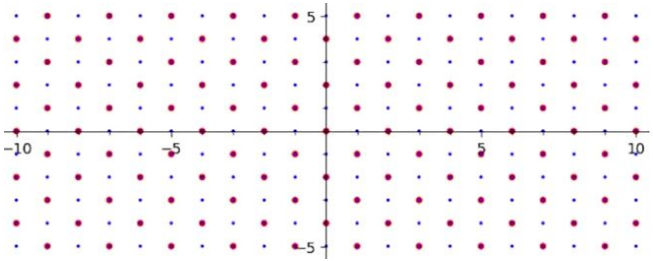
- We can add all integer multiples of 1+i:



- It should also contain all integer multiples of i times 1 + i:



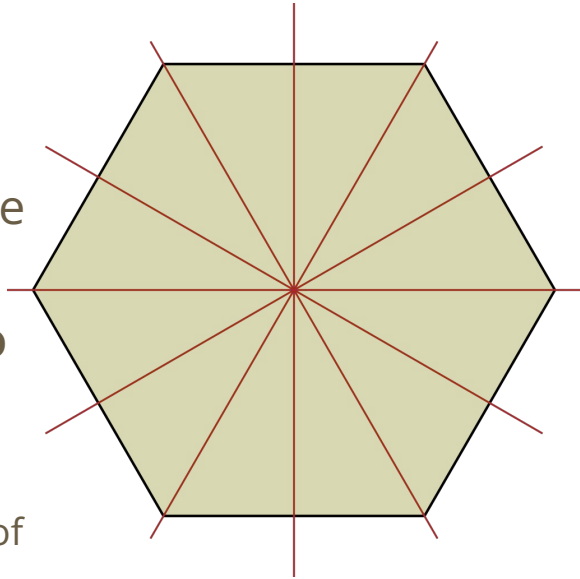
- And all possible sums of those numbers:



Topic: Representation Theory

Symmetries

- Symmetries show up all over in real life, including in rotations, functions, quantum physics, etc.
- It's easy to see the symmetries of the hexagon on the right. But how can we work with them algebraically?
- **Representation theory converts symmetries into matrices**, which we can study really well using ring theory and linear algebra.
    - The insolvability of the quintic was done by showing the lack of symmetries between roots of polynomials.
    - Lie tried to replicate this to find solutions of differential equations, but ultimately was unsuccessful.
    - Instead, he used representation theory to develop Lie algebras, which study symmetries on other manifolds.



Topic: Gröbner bases

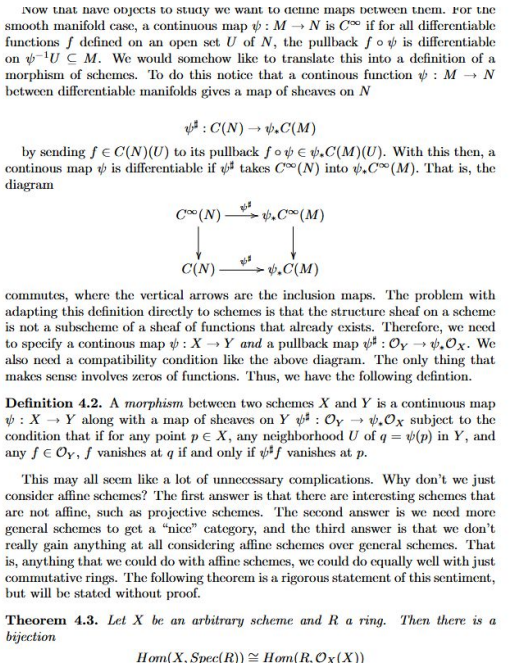
Working with polynomials

- One of the first things you learn in a linear algebra course is how to solve systems of linear equations using Gaussian elimination (row reduction).
- This doesn't directly apply to systems of polynomials. However, by constructing Gröbner bases from ideals of rings, we can essentially perform Gaussian elimination on polynomials.
- This has a number of key applications:
  - Solving polynomial systems
  - Deciding if one polynomial can be built from others (ideal membership)
  - Removing variables to get equations in fewer variables (elimination)
  - Symbolic computation (used for Maple, Mathematica, etc.)
  - Robotics & kinematics
  - Coding and cryptography

Topic: Algebraic Geometry

Algebraic geometry

- Algebraic geometry is a notoriously difficult topic in mathematics that also seems to show up everywhere.
  - I could give a whole presentation on algebraic geometry since the field is just so insanely deep and rich!
- From a very basic point of view, algebraic geometry is just “finding solutions to systems of polynomial equations”.
  - Linear algebra is finding solutions to systems of linear equations!
- But really, it’s about the deep connection between algebra and geometry.



Now that have objects to study we want to define maps between them. For the smooth manifold case, a continuous map $\psi : M \to N$ is C^∞ if for all differentiable functions f defined on an open set U of N, the pullback $f \circ \psi$ is differentiable on $\psi^{-1}U \subseteq M$. We would somehow like to translate this into a definition of a morphism of schemes. To do this notice that a continous function $\psi : M \to N$ between differentiable manifolds gives a map of sheaves on N

$$\psi^\sharp : C(N) \to \psi_* C(M)$$

by sending $f \in C(N)(U)$ to its pullback $f \circ \psi \in \psi_* C(M)(U)$. With this then, a continous map ψ is differentiable if $\psi^\sharp$ takes $C^\infty(N)$ into $\psi_* C^\infty(M)$. That is, the diagram

$$\begin{array}{ccc} C^\infty(N) & \xrightarrow{\psi^\sharp} & \psi_* C^\infty(M) \\ \downarrow & & \downarrow \\ C(N) & \xrightarrow{\psi^\sharp} & \psi_* C(M) \end{array}$$

commutes, where the vertical arrows are the inclusion maps. The problem with adapting this definition directly to schemes is that the structure sheaf on a scheme is not a subscheme of a sheaf of functions that already exists. Therefore, we need to specify a continous map $\psi : X \to Y$ *and* a pullback map $\psi^\sharp : \mathcal{O}_Y \to \psi_* \mathcal{O}_X$. We also need a compatibility condition like the above diagram. The only thing that makes sense involves zeros of functions. Thus, we have the following defintion.

Definition 4.2. A *morphism* between two schemes X and Y is a continuous map $\psi : X \to Y$ along with a map of sheaves on Y $\psi^\sharp : \mathcal{O}_Y \to \psi_* \mathcal{O}_X$ subject to the condition that if for any point $p \in X$, any neighborhood U of $q = \psi(p)$ in Y, and any $f \in \mathcal{O}_Y$, f vanishes at q if and only if $\psi^\sharp f$ vanishes at p.

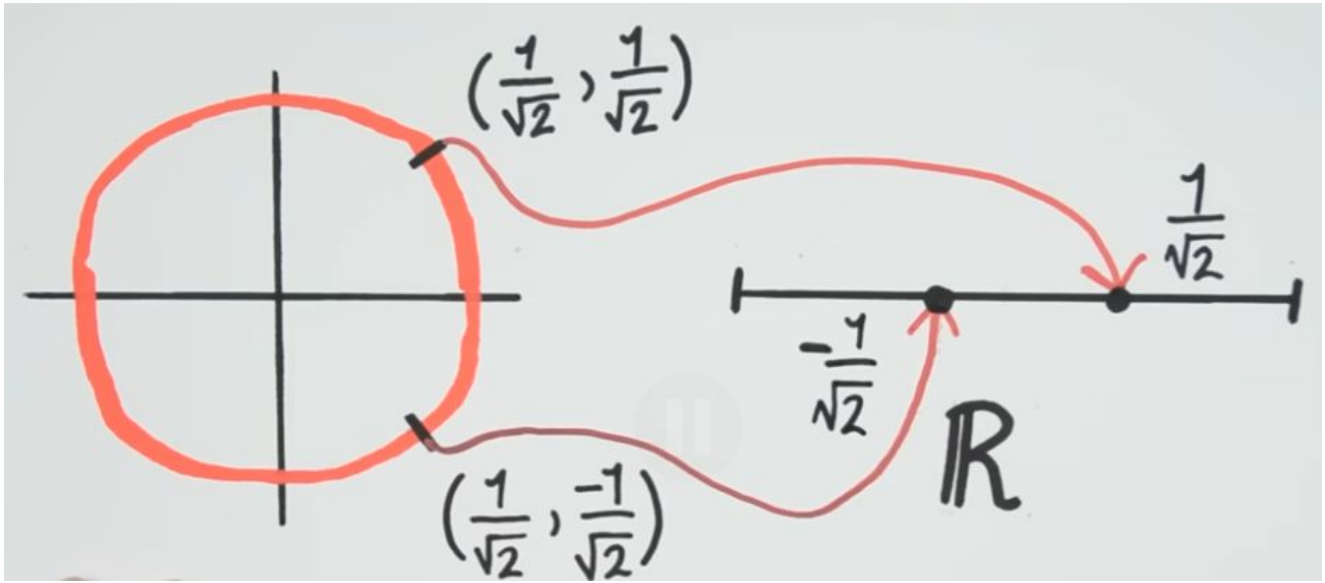
This may all seem like a lot of unnecessary complications. Why don’t we just consider affine schemes? The first answer is that there are interesting schemes that are not affine, such as projective schemes. The second answer is we need more general schemes to get a “nice” category, and the third answer is that we don’t really gain anything at all considering affine schemes over general schemes. That is, anything that we could do with affine schemes, we could do equally well with just commutative rings. The following theorem is a rigorous statement of this sentiment, but will be stated without proof.

Theorem 4.3. *Let X be an arbitrary scheme and R a ring. Then there is a bijection*

$$Hom(X, Spec(R)) \cong Hom(R, \mathcal{O}_X(X))$$

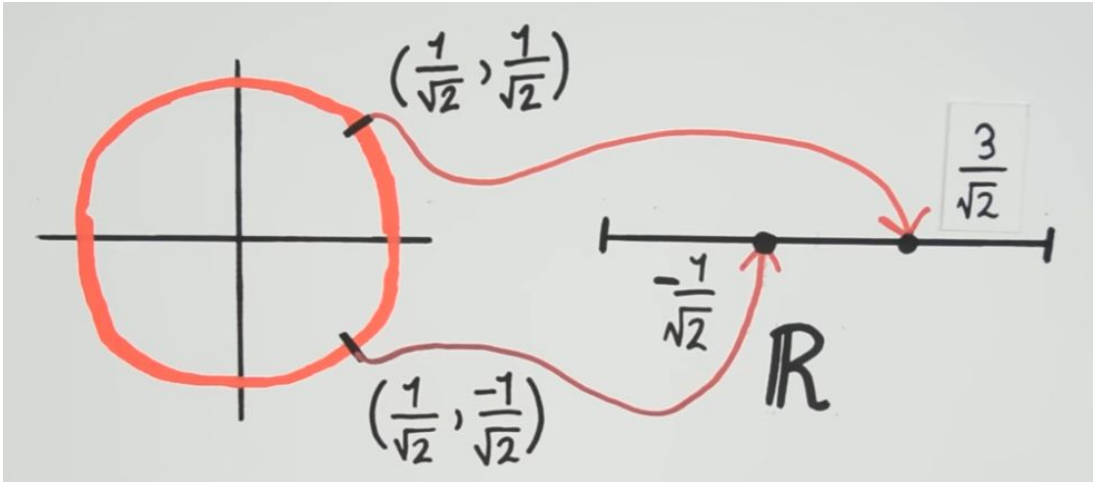
Coordinate rings

- Let’s consider the unit circle. One thing we can do is take functions from the unit circle to R, for example: **f(x, y) = y**.



Coordinate rings (cont.)

- We can also consider more complicated functions, such as **f(x, y) = x + 2y**
- In general, these are all polynomials. But polynomials form a ring! So we can add and multiply them. Let's call the ring of polynomials R.

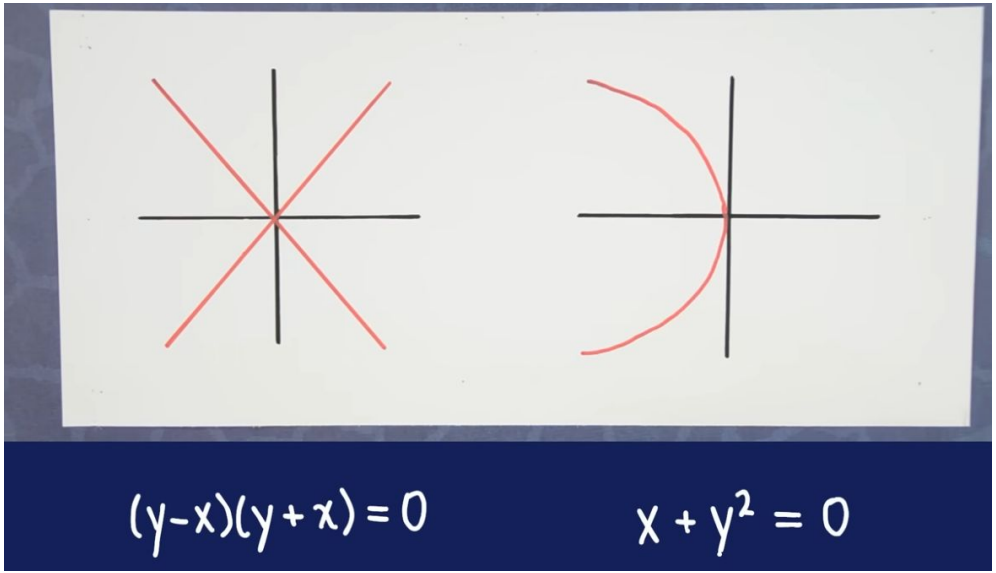


Coordinate rings (cont.)

- What are some other polynomials? I'll give two examples:
  $f(x, y) = x + 2y$
  $g(x, y) = x + 2y + x^2 + y^2 - 1$
- Note that for points on the unit circle, $f(x, y)$ and $g(x, y)$ are identical! So they're the same element in our ring R.
  - In fact, all functions of the form $g(x, y) = x + 2y + h(x, y)(x^2 + y^2 - 1)$ are equivalent!
- The geometric properties of the circle are encoded in the algebraic properties of our coordinate ring.
  - Mathematically, we are taking $R = R[x, y] / (x^2 + y^2 - 1)$

One last example

- What are the qualitative differences between these two curves?
- The first one is made of two pieces (reducible) and the second isn't (irreducible)



One last example (cont.)

- Let R be the coordinate ring of $(y-x)(y+x) = 0$, and S be the coordinate ring of $x + y^2 = 0$.
- Consider $f(x, y) = y + x$. It's clearly a nonzero function. Now consider $g(x, y) = y - x$. It's also clearly nonzero. However, their product is zero on the curve!
- This may feel weird. The product of two nonzero things is zero? This doesn't happen on S.
- By **finding this property that the product of two nonzero things is zero**, we have **detected that the curve is made of two pieces**!
- This is the power of algebraic geometry – we use algebra to study geometric things, often with lots of ring theory baked in.

Application: parallelization

Our problem

- This function finds the largest sum that can be made from a subsequence of a sequence.
- Given an array arr, it goes through each element and accumulates the sum into sofar. If sofar dips below 0, we reset it back to 0. The maximum value of sofar is saved into max.
- Note we start the function with a starting vector: [0, infinity]. 0 corresponds to us having summed over 0 elements, and -Infinity means we want the first sum To be the highest at that point.

```
function f1(arr) {
  let sofar = 0;
  let max = -Infinity;

  for (const b of arr) {
    sofar = sofar + b < 0 ? 0 : sofar + b;
    max = max < sofar ? sofar : max;
  }

  return [sofar, max];
}
```

Why is this hard?

- The problem in this code is the presence of a **serial dependency chain**. This is an insidious problem that many programmers make without even realizing!
- Essentially, each iteration of the loop depends on the previous iteration of the loop. It would be very hard to run this in parallel, because splitting the array into two function instances means we may split the longest subsequence.
- So how can we improve this?

```
function f1(arr) {
  let sofar = 0;
  let max = -Infinity;

  for (const b of arr) {
    sofar = sofar + b < 0 ? 0 : sofar + b;
    max = max < sofar ? sofar : max;
  }

  return [sofar, max];
}
```

First steps

- The first step is to write f1 with using Math.max instead of conditionals:
- Looks a bit nicer. It still hasn't addressed the serial dependency chain though, which means it will run at basically the same speed.
- Let's solve another problem instead. In f2 I'll replace Math.max with addition and addition with multiplication. 0 is the identity for addition so we should replace it with the multiplicative identity 1. Similarly, -Infinity is the multiplicative identity for max so we'll replace that with 0.

```
function f2(arr) {
  let sofar = 0;
  let max = -Infinity;

  for (const b of arr) {
    sofar = Math.max(0, sofar + b);
    max = Math.max(sofar, max);
  }

  return [sofar, max];
}
```

Trying things

- This function looks even cleaner but there are still two problems:
  - It doesn't look any more parallelizable – the serial dependency chain is still there.
  - It's solving the wrong problem.
- Let's ignore the second problem for now. What makes this function easy to work with is that it's almost a linear function acting on [sofar, max]. Linear functions have a very nice property: **if you have two linear functions f and g, you can make a new linear function f(g(x)).**
- We can express this by writing f and g as matrices!
- We have a +1 term in here which makes it not linear, so let's add an extra term:

```
function f3(arr) {
  let sofar = 1;
  let max = 0;

  for (const b of arr) {
    sofar = sofar * b + 1;
    max = sofar + max;
  }

  return [sofar, max];
}
```

Applying matrices

- Now our function is linear. I won't add the matrix code since it's long and implementation-dependent.
- Specifically, to compute n iterations, you find the matrix M, then compute M^n.
- But we know that matrices are rings! Specifically, matrix multiplication is associative. That means we can easily parallelize it!
  - Chop arr into a bunch of pieces
  - Give each piece to one processor (length p)
  - Each processor computes M^p
  - Resulting matrices are combined into one, which is applied to the starting vector.

```
function f4(arr) {
  let sofar = 1;
  let max = 0;
  let i = 1;

  for (const b of arr) {
    sofar = (sofar * b) + i;
    max = sofar + max;
    i = i;
  }

  return [sofar, max];
}
```

Solving the original problem

- This is all really cool, but we are still solving the wrong problem.
- Remember we replaced max with addition and addition with multiplication. All we have to do is undo that.
    - Literally, in your implementation of multiplyMatrices(a, b), replace all the additions with Math.max and all the multiplications with additions.
- Everything required to prove the parallelization holds is **valid over any semiring**, because we did not divide or subtract, and we only used basic properties of numbers such as associativity or distributivity.
    - Specifically, all we did was perform matrix multiplication over the tropical max-plus semiring.

Application: Error correction

Error correction

- When a piece of computer hardware communicates with another piece of computer hardware, it sends a series of 1’s and 0’s of arbitrary length, and of course there is the potential for error.
- The simplest method is to add up all the digits, take it mod n (where n is the base), and put that as the final digit.
    - Example: 14598 -> 145987. If you write it down as 145887, the check will fail and you get an error!
- This method works fine for basic cases, but it has some problems:
    - It can only tell you if there is an error, not how many there are
    - Changing the value of a digit causes an error, but swapping two digits doesn’t

Cyclic redundancy check (CRC)

- Computers are really good at performing bitwise operations quickly (AND, OR, NOT, etc.). Specifically, they can perform XOR very quickly!
- Knowing this, the idea behind CRC is as follows:
  - There is a fixed string called the generator (e.g. 1011) that is known by all hardware.
  - The hardware wants to send a string of arbitrary length (e.g. 11111001). The hardware then **performs XOR repeatedly with this string**, which is very fast.
  - 11111001 XOR 1011 = 01001001. So now we repeat this process with 1001001:
  - 1001001 XOR 1011 = 0010001, 10001 XOR 1011 = 111. We stop because 111 is shorter than 1011.
- In this case, the result of repeated XOR'ing the sent string and fixed string is our "check digit" that gets tacked onto the end of our message.

CRC math

- We can represent our strings as polynomials in F_2 with the coefficients being our string's values: e.g. 11111001 = $x^7 + x^6 + x^5 + x^4 + x^3 + 1$
- Also represent the generator string as a polynomial: 1011 -> $x^3 + x + 1$.
- The hardware is essentially finding that polynomial modulo the generator polynomial (the remainder).
    - Mathematically, this is finding the coset representative of this polynomial in the quotient ring $F_2[x]/(x^3 + x + 1)$, which in this case is $x^2 + x + 1$.
- This works best if the fixed string corresponds to an irreducible polynomial, so you can divide things. Otherwise, there are zero-divisors, which can hide errors.
    - Mathematically, the fixed string being irreducible means the quotient ring is actually a field.

Application: Algebird

Algebird

- Algebird is a library which provides abstractions for abstract algebra in the Scala programming language, specifically objects like rings, monoids, groups, and monads.
- The code is designed for large aggregation systems that may rely on giant datasets.
- By abstracting as much as possible, algorithms such as HyperLogLog, Bloom filter, CountMinSketch etc. can be **written once and used for a wide variety of things that may not even be numbers**!
- So yes, there are people that are being paid large sums of money to do abstract algebra!

Quote by Sam Ritchie

“One cool feature: When you visit a tweet, you want the reverse feed of things that have embedded the tweet. The MapReduce graph for this comes from: When you see an impression, find the key of the tweet and emit a tuple of the tweetId and Map[URL, Long]. Since Maps have a monoid, this can be run in parallel, and it will contain a list of who has viewed it and from where. The Map has a Long since popular tweets can be embedded in millions of websites and so they use a “CountMinSketch” which is an approximate data structure to deal with scale there. The Summingbird layer which the speaker [Sam Ritchie] shows on stage filters events, and generates key-value pairs and emits events.

Twitter advertising is also built on Summingbird. Various campaigns can be built by building a backend using a monoid that expresses the needs, and then the majority of the work is on the UI work in the frontend (where it should be — remember, solve systems problems once is part of the vision).”

Conclusion

- This was a broad overview of rings and all the different fields of math they are relevant in, as well as some programming related examples.
- The key takeaway is that **working with abstract entities allows the same words to be reused in a valid way in many different contexts**. This benefits both mathematicians and computer scientists.
- Specifically, knowledge about rings and semirings can have a lot of usefulness in programming related environments, especially problems relating to matrices.

Thanks for listening!

SIG Math Fall 2025