UCT

Computability, Fundamentals

Klaus Sutner

Carnegie Mellon University

Spring 2022

Where Are We?

- Turing machines provide the reference model of computability.
- Every Turing machine can be associated with an index $e \in \mathbb{N}$.
- There is a universal Turing machine $\mathcal{U}$ that can simulate any Turing machine given its index.
- The Halting problem "does $\mathcal{U}(e, e) \downarrow$?" is the prime example of an undecidable problem.
- Halting is difficult to determine even for very small machines.
- Halting gives rise to the class of semidecidable problem.

Undecidability

We mentioned a few other undecidable problems, so all the following do not admit a decision algorithm:

- Halting
- First-order logic
- Entscheidungsproblem

OK, no way around this, but so far all the problems seem slight esoteric, not the kind a mathematician much less a programmer might be interested in.

Perhaps these results are somewhat disconcerting, but essentially just irrelevant?

No Such Luck

Alas, there are many problems in math that seem to have nothing to do with Halting or FOL, but turn out to be undecidable. In fact, undecidability seems to be a major feature of math.

Worse, as we will see, undecidability casts a direct shadow even in the world of feasible computation. Even when things are decidable in principle, computational hardness is a feature, not a bug. We have to figure out how to deal with it.

But before we go there, a few more basic results about abstract computability. Some of the methods here will carry over directly to complexity theory.

Halts, But Barely

Suppose we ignore partial functions for the moment. Maybe total recursive functions are much better behaved?

Alas, no. Below is an example of an arithmetic function $\alpha : \mathbb{N} \to \mathbb{N}$ that is trivial to compute in the sense that just a few lines of code in any programming language suffice. The logic behind the algorithm is utterly simple.

It is easy to determine $\alpha(1)$ and $\alpha(2)$ by hand. And then things blow up:

> It is impossible to compute $\alpha(3)$ in any reasonable sense of the word.

The reason is that the result is so absurdly large that our universe cannot even begin to accommodate this type of computation, not even a tiny fragment thereof.

Subsequences

A word u is a subsequence of v if we can erase some letters in v to get u (a scattered subword).

So abc is a subsequence of $cbabbabcac$ (in exactly 8 different ways), but ccb is not.

This defines a partial order on $\Sigma^\star$.

Exercise

Find a good algorithm to check whether u is a subsequence of v.
Then find a algorithm that counts the number of ways u appears as a subsequence of v.

Friedman's Self-Avoiding Words

Fix some alphabet Σ and use 1-indexing for words over Σ. For a word x write

$$x[i] = x_i x_{i+1} \dots x_{2i}$$

for the block of x from position i to position $2i$. Note this makes sense only for $i \leq |x|/2$.

Bizarre Definition: A word x is self-avoiding if for all $i < j \leq |x|/2$ the block $x[i]$ is not a subsequence of block $x[j]$.

Here is a consequence of a famous and difficult theorem by Higman[†]:

Theorem

Over any alphabet, there are only finitely many self-avoiding words.

[†]The subsequence order does not have infinite antichains.

How Long?

Clearly, the number of self-avoiding words depends only on the size of the alphabet, so write Σ_k for an alphabet of size k.

Now let $\alpha(k)$ be the length of the longest self-avoiding word over Σ_k:

$$\alpha(k) = \max\big(|x| \mid x \in \Sigma_k^*, x \text{ self-avoiding} \big)$$

Clearly, α is strictly increasing, but it is not so clear what the difference between $\alpha(k)$ and $\alpha(k+1)$ is.

At any rate, α is easily computable via a brute-force attack. see below.

The Algorithm

Here is the obvious brute-force algorithm.

- At round 0, define the list of words $L = \{\varepsilon\}$.
- In round $n > 0$, extend all words in L by all letters in Σ_k.
  Remove non-self-avoiding words from L.
- Stop when L becomes empty.
  Then $\alpha(k) = n - 1$.

Each round is easily primitive recursive, really just some wordprocessing.

Termination is guaranteed by the theorem: we are essentially growing a tree (actually: a trie). If the algorithm did not terminate it would produce infinitely many self-avoiding words[†].

[†] This requires Kőnig's lemma on finitely branching infinite trees.

Some Examples

Clearly, all words of length at most 3 are self-avoiding according to our definition. Here is the number of self-avoiding words of length up to 12, for $k \leq 4$.

1	2	3	4	5	6	7	8	9	10	11	12
1	1	1	0	0	0	0	0	0	0	0	0
2	4	8	8	16	12	24	4	8	2	4	0
3	9	27	60	180	348	1044	1518	4554	5334	16002	16674
4	16	64	216	864	2688	10752	29376	117504	285108	1140432	2569248

So $\alpha(1) = 3$ and $\alpha(2) = 11$, as witnessed by $abbbaaaaaaa$ and $abbbaaaaaab$.

And $k = 3$?

Alas, $\alpha(3)$ is a bit harder to describe. We will use the following, very rapidly growing arithmetic functions for this purpose.

$$B_1(x) = 2x$$

$$B_{k+1}(x) = B_k^x(1)$$

$B_k^x(1)$ means: iterate B_k x-times on 1. So B_1 is doubling, B_2 exponentiation, B_3 super-exponentiation (essentially a stack of exponentials of height x) and so on.

Beyond B_3, these functions spin out of control very quickly, they just grow exceedingly fast and are difficult to make sense of [†].

[†] Both Knuth and Conway have devised notation systems for fast growing functions, take a look at Wikipedia.

Drum Roll

$$\alpha(3) > B_{7198}(158386)$$

Smelling salts, anyone?

This is an incomprehensibly, mind-numbingly large number. One might even wonder whether it exists in the same sense as, say, the number 42 exists.

It is truly surprising that a function with as simple a definition as α should exhibit this kind of growth.

Quoi?

Just to be absolutely clear: a Turing machine for α does halt on all inputs, and, for inputs 1 and 2, it halts rather quickly.

But then for input 3 it runs for an absolutely incomprehensible number of steps. This is reminiscent of the 5-state busy beaver, but much, much worse. $\alpha(3)$ is so far removed from physically realizable computation that one might wonder whether our basic definitions are really right.

They are, insane growth in computable functions is a feature, not a bug.

Hilbert's 10th Problem

Perhaps the most famous example of an undecidability result in mathematics is Hilbert's 10th problem[†], concerning solutions of Diophantine equations. A Diophantine equation is a polynomial equation with integer coefficients:

$$P(x_1, x_2, \ldots, x_n) = 0$$

The problem is to determine whether such an equation has an integral solution.

Theorem (Y. Matiyasevic, 1970)

It is undecidable whether a Diophantine equation has a solution in the integers.

[†]Posed by Hilbert in 1900.

Examples

Pythagorean triples: it is not hard to classify all the solutions of

$$x^2 + y^2 - z^2 = 0$$

But tackling

$$x^k + y^k - z^k = 0$$

for $k > 2$ was one of the central problems of number theory and it took more than 350 years to show that solutions only exist for $k = 1, 2$.

Different Numbers

Note that the choice of $\mathbb{Z}$ as ground ring is important here. We can ask the same question for polynomial equations over other rings R (always assuming that the coefficients have simple descriptions).

- $\mathbb{Z}$: undecidable
- $\mathbb{Q}$: major open problem
- $\mathbb{R}$: decidable
- $\mathbb{C}$: decidable

Decidability of Diophantine equations over the reals is a famous result by A. Tarski from 1951, later improved by P. Cohen. $\mathbb{C}$ follows from $\mathbb{R}$, but can also be proven rather easily directly (using algebra and model theory).

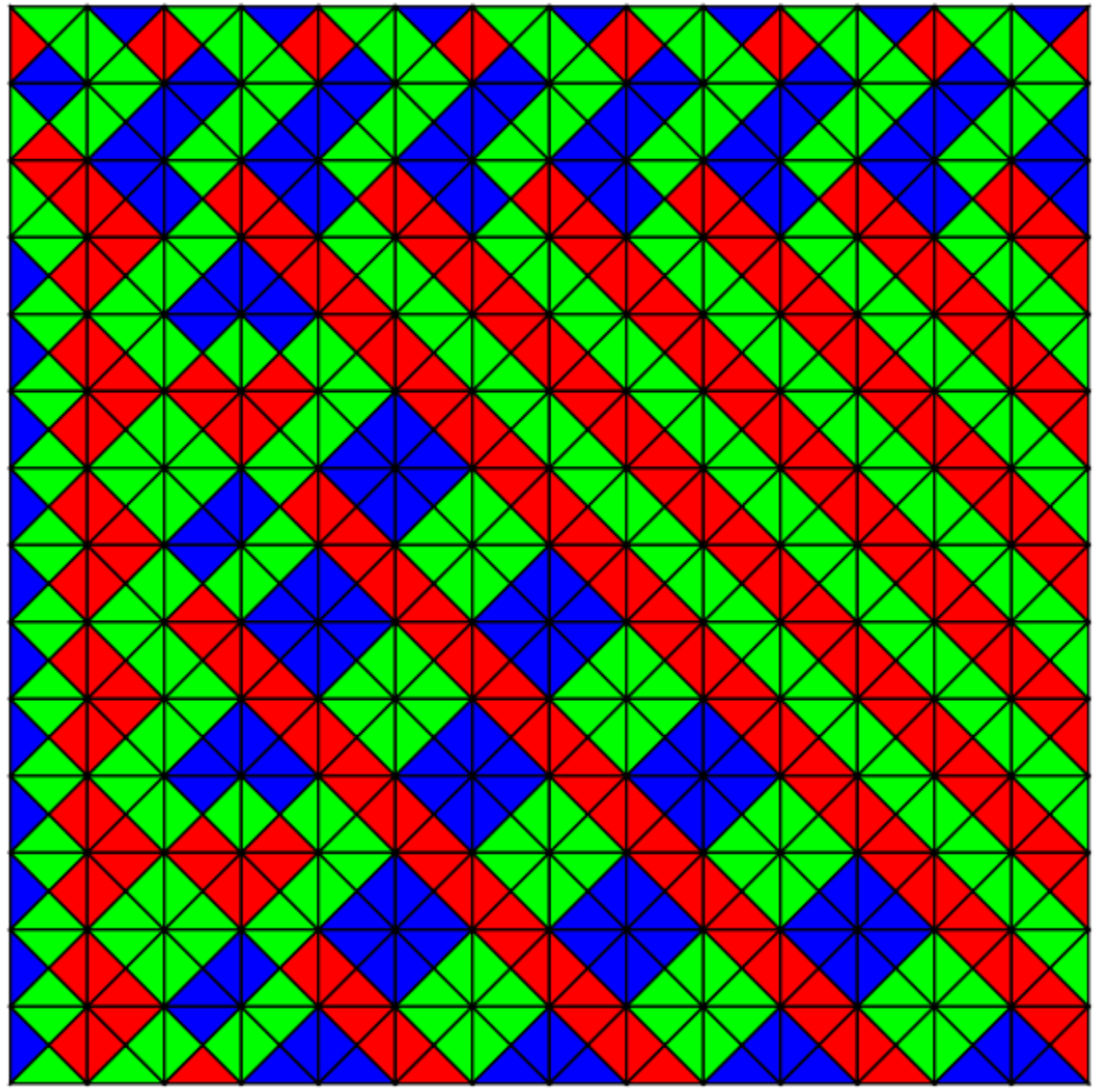
Wang Tiles

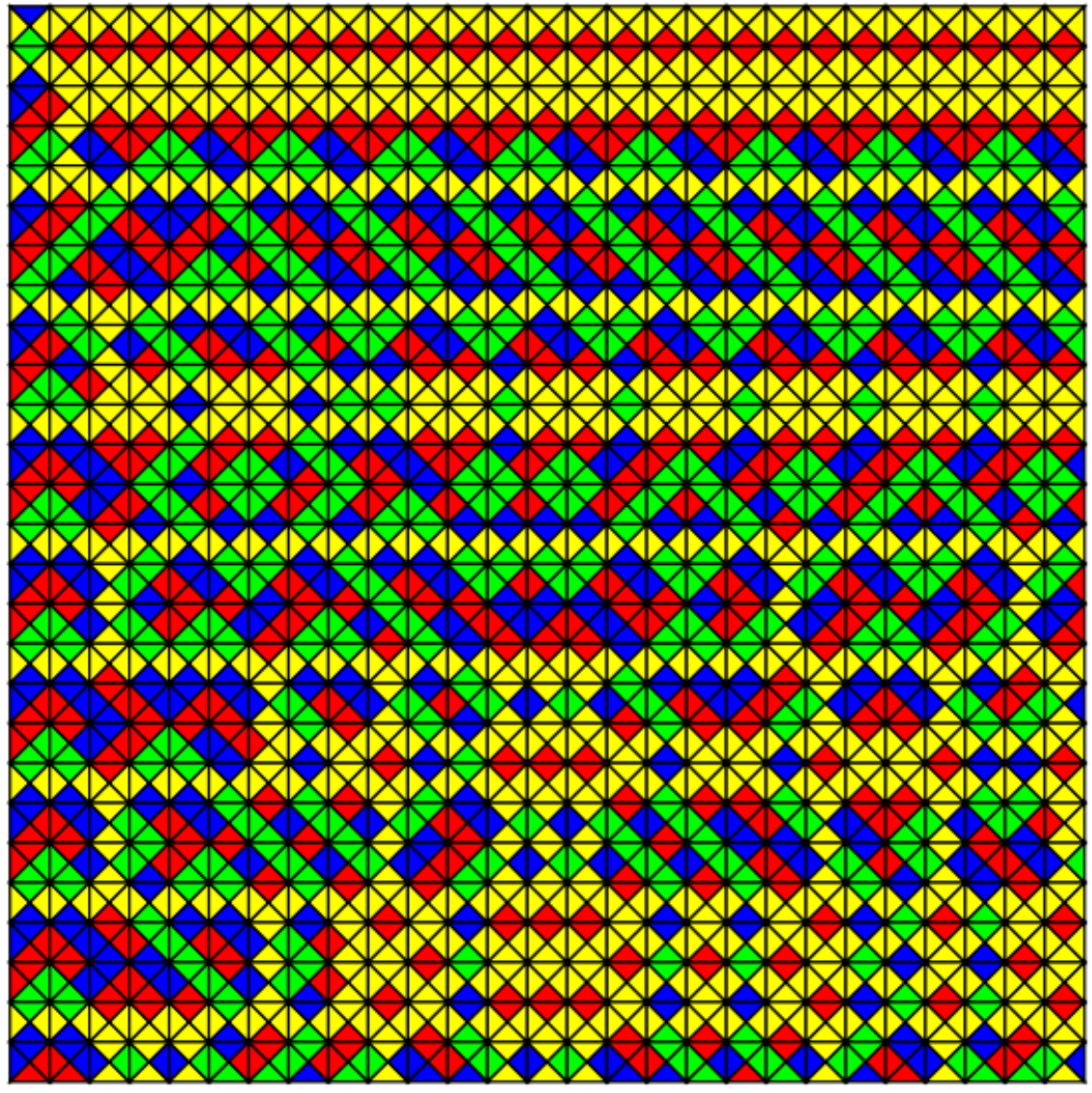
Hilbert's 10th problem is the classical example of an undecidable problem in number theory. How about geometry?

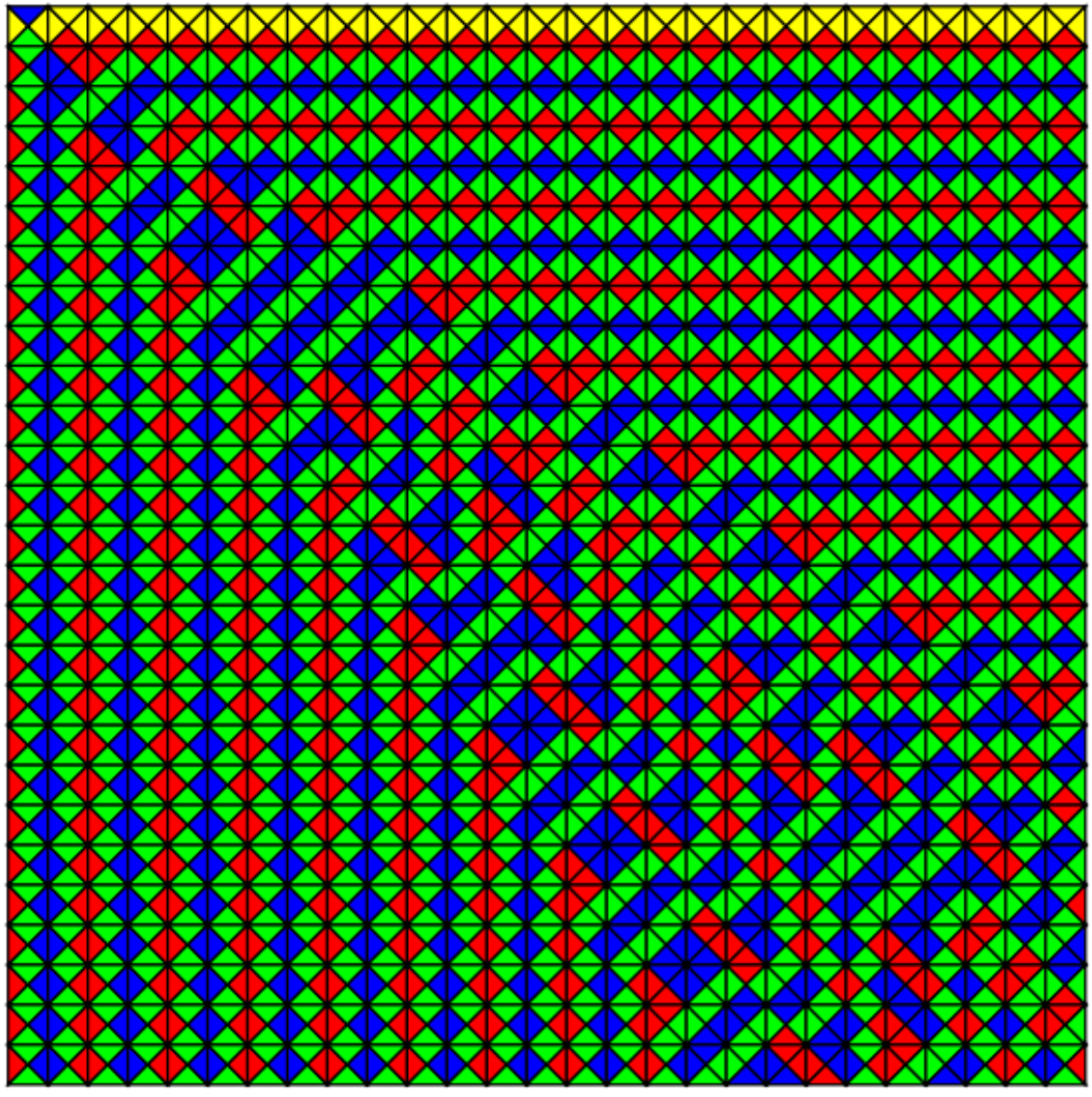
Here is a geometric problem: we are given a collection of tiles, unit squares that are colored as in this example:

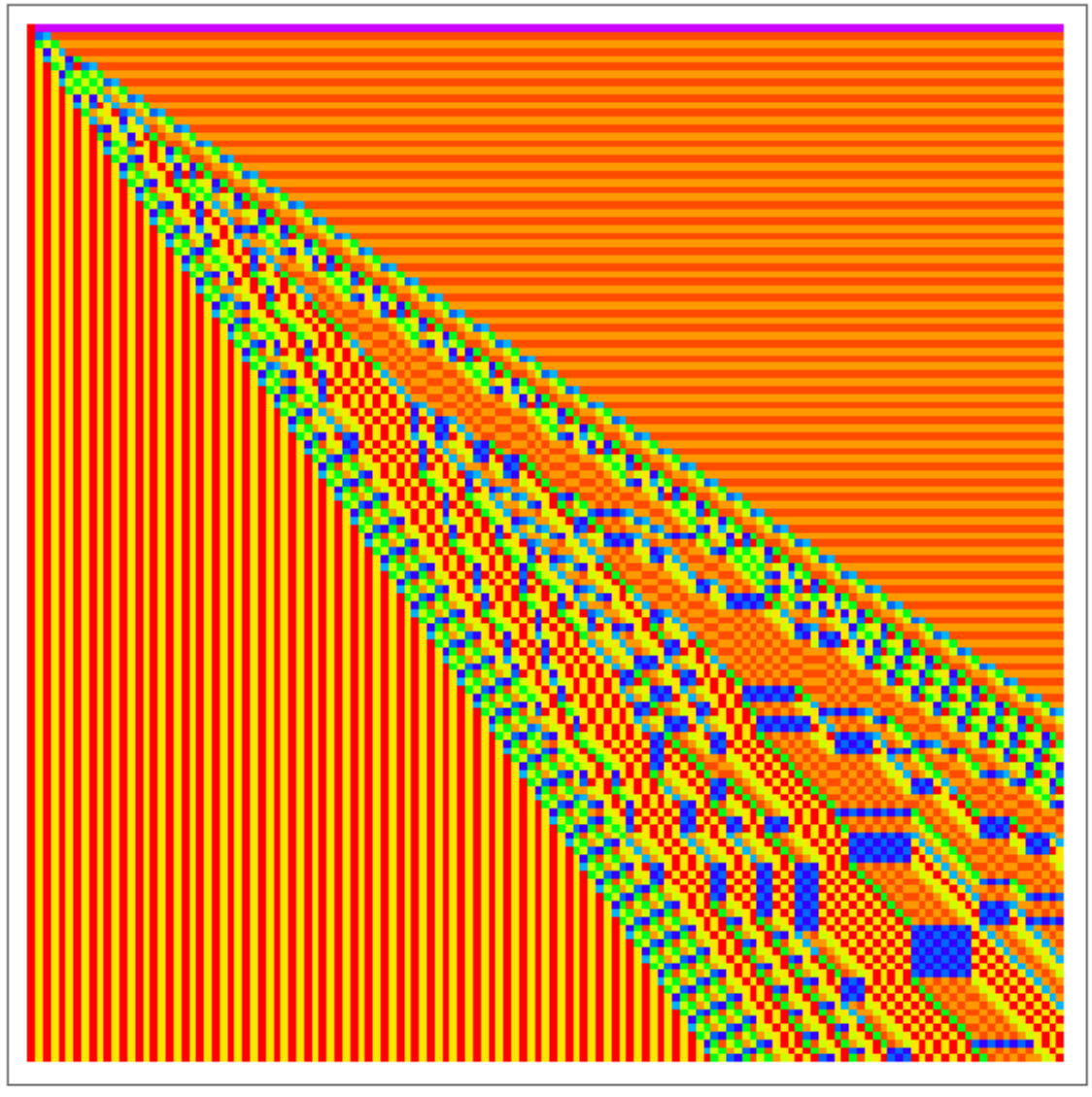
There is an infinite supply of each type of tile, rotations and reflections are not allowed.

The goal is to cover the whole plane with these tiles, like an infinite chess board, in a way that all adjacent colors match.









Aperiodic Tiling

Theorem (R. Berger 1966)

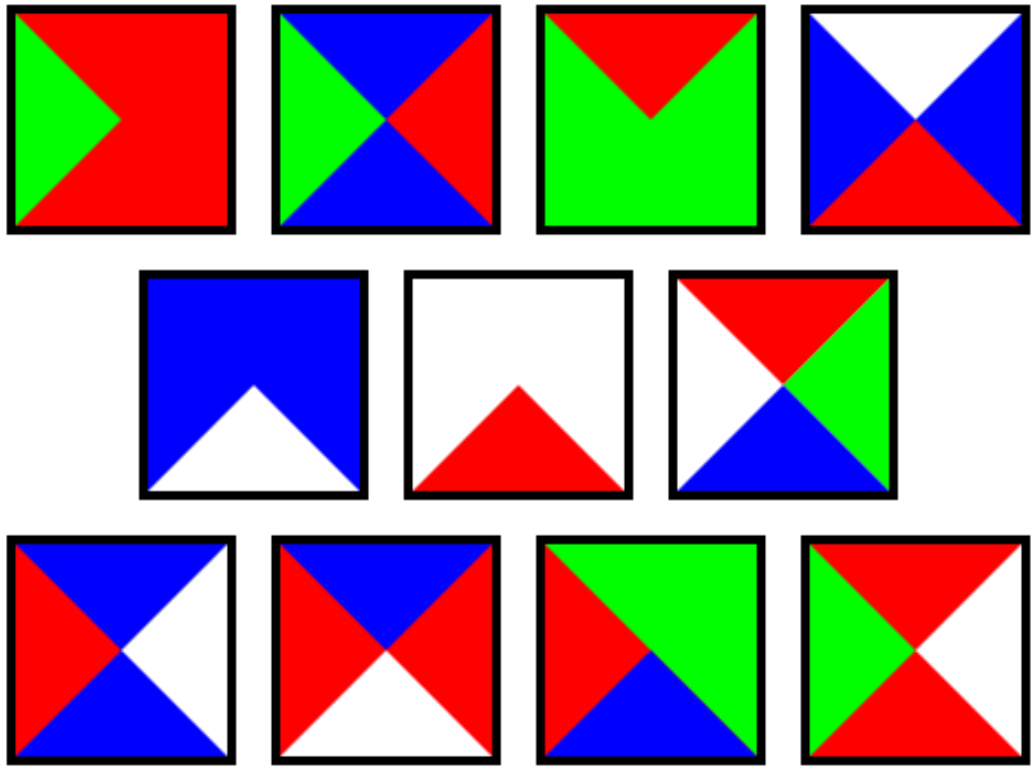
The Tiling Problem is undecidable.

This result is based on a surprising and somewhat counterintuitive fact: there are sets of Wang tiles that admit a tiling of the whole plane, but not a periodic one. Hao Wang originally conjectured in 1961 that such tiles to not exist.

Exercise

Why would Wang's conjecture imply that Tiling is decidable?

Aperiodic Wang Tiles



Interestingly, small sets of Wang tiles can be constructed using finite state machines and old-fashioned hyperbolic geometry.

Power of Halting

Somewhat surprisingly, even some major unsolved problems of math can be expressed nicely in terms of halting Turing machines.

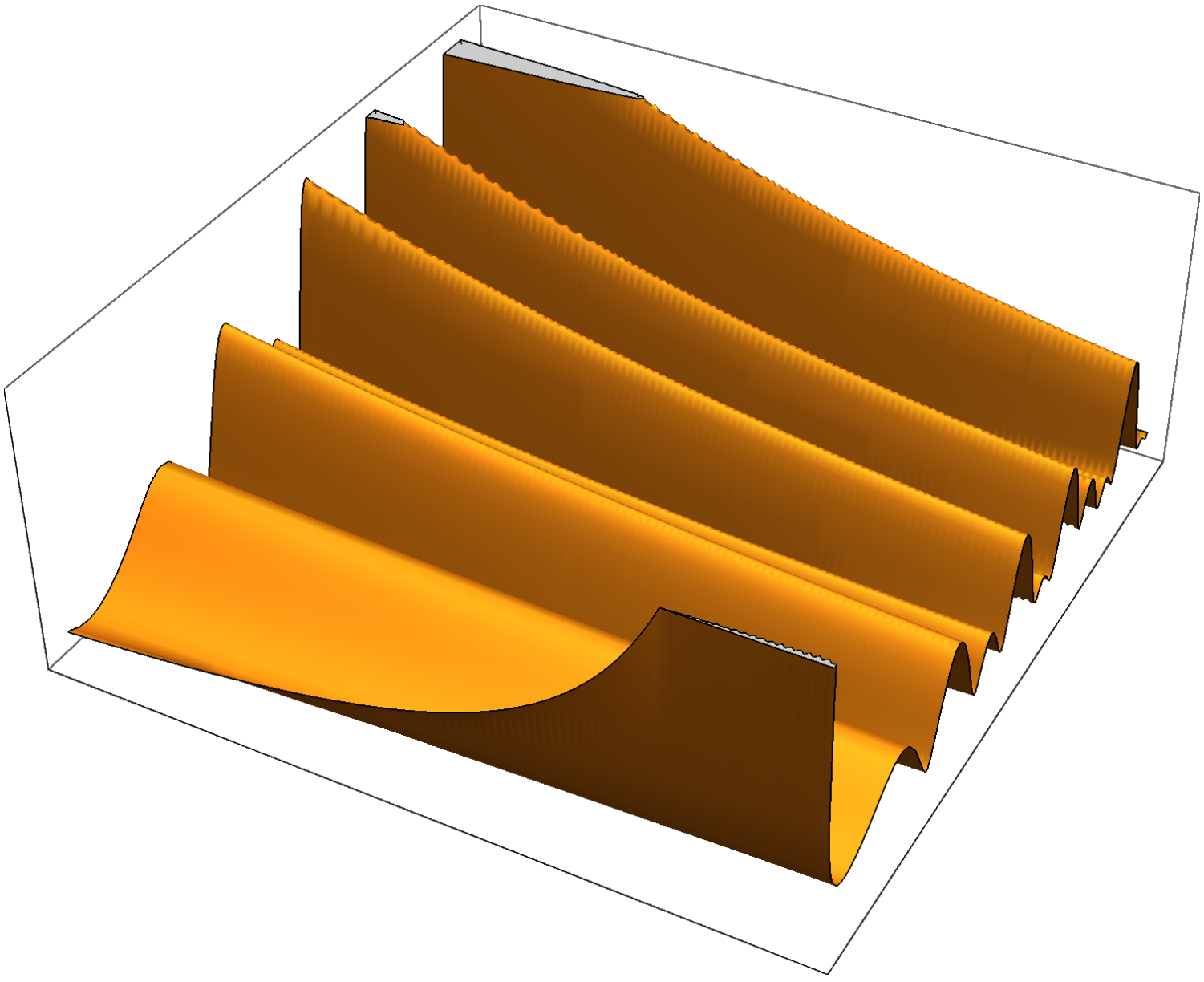
For example, the Riemann Hypothesis in its original formulation says the following. Let

$$\zeta(s) = \sum_{n \geq 1} n^{-s}$$

where $\mathrm{Re}(s) > 1$.

By analytic continuation, one extend this to a function defined on all of $\mathbb{C}$, so now $\zeta : \mathbb{C} \to \mathbb{C}$. Details don't matter for us, it's enough to note that this function is enormously complicated.

Zeta Function



In particular the location of the roots of ζ has major implications for number theory, even to the point that certain algorithms are fast provided the following is true:

Riemann Hypothesis
All non-real roots s of this function have $\mathrm{Re}(s) = 1/2$.

It's a Millennium Problem, worth a million dollars.

Simplifying Things

As written, this looks like a very complicated assertion. But, RH is equivalent to the following inequality, for all $n \geq 1$:

$$\sigma(n) \leq H_n + e^{H_n} \ln H_n$$

where

- σ is the divisor-sum function, $\sigma(n) = \sum_{d \mid n} d$
- $H_n = \sum_{i \leq n} 1/i$ is the nth harmonic number.

This condition is a bit messy to check by hand for given n (remember Euler's method to compute e^x?), but no problem with a computer algebra system.

```
In[423]:= nn = 40;
          DivisorSigma[1, nn]
          Hn = HarmonicNumber[nn]
          en = E^Hn
          ln = Log[Hn]
          N[ Hn + en ln ]
```

Out[424]= 90

Out[425]= $\frac{2\,078\,178\,381\,193\,813}{485\,721\,041\,551\,200}$

Out[426]= $e^{2\,078\,178\,381\,193\,813/485\,721\,041\,551\,200}$

Out[427]= $\text{Log}\left[\frac{2\,078\,178\,381\,193\,813}{485\,721\,041\,551\,200}\right]$

Out[428]= 109.135

Error Estimates

Just to be clear: the hard part here is to make sure that the approximation 109.135 is sufficiently accurate. If we had an error bound of 20 we would be in trouble.

No problem, that is exactly what numerical math is for.

One can construct a Turing machine $\mathcal{M}_0$ that verifies this inequality for a given n. "One can" here means that the person trying to do this will go stark raving mad[†].

Then we use this TM inside another main machine $\mathcal{M}$ that runs through all n, and halts only if $\mathcal{M}_0$ finds a counterexample.

Claim: $\mathcal{M}$ halts iff RH is false.

So a solution to Halting (even just for one particular machine) could eviscerate one of the hardest open problems in math.

[†]Not quite. There are estimates on how large such a machine would need to be. 8000 states seems like plenty.

Index Computation

Recall the enumeration $\{e\}$ of all partial computable functions. Having an index e that represents a function makes it possible to manipulate functions by manipulating their indices instead.

For example, there is an easily computable function f such that $f(e, e')$ is an index for a machine $\mathcal{M}$ that composes $\mathcal{M}_e$ and $\mathcal{M}_{e'}$: first run $\mathcal{M}_e$, if there is output, feed it to $\mathcal{M}_{e'}$ and return the output of that computation:

$$\mathcal{M}_{f(e,e')} \text{ computes } \{e\} \circ \{e'\}$$

In terms of functions:

$$\{f(e, e')\} \simeq \{e\} \circ \{e'\}$$

This is just software engineering, no problem.

Kleene's S-m-n theorem

Here is another index computation: we can fix a few arguments in a computable function and obtain another computable function. Yes, this is a no-brainer, but we are trying to be a bit formal here.

Theorem

For every $m, n \geq 1$ there is a primitive recursive function S_n^m such that

$$\{S_n^m(e, \boldsymbol{p})\}^{(n)}(\boldsymbol{x}) \simeq \{e\}^{(m+n)}(\boldsymbol{p}, \boldsymbol{x}).$$

Proof.

Klar (as per number theorist E. Landau).

You can also think of this as a form of currying.

□

TM Properties

We are often interested in collections of Turing machines (or their associated functions) that have certain properties. We can think of these collections as subset of $\mathbb{N}$ via indices. Typical example:

$$\mathsf{TOT} = \{\, e \mid \mathcal{M}_e \text{ halts on all } x \,\}$$

So TOT is the collection of all Turing machines that define total functions.

Note that for $\{e\} \simeq \{e'\}$ we have $e \in \mathsf{TOT} \Leftrightarrow e' \in \mathsf{TOT}$, we dealing with a structural property of the machines rather than some nonsense property of the indices (say, e is even).

Such structural properties are always undecidable unless they are trivial.

For this kind of argument, it is absolutely critical to first develop a solid intuition–the technical details are then typically not too difficult to figure out[†].

Key Intuition:
TOT seems even harder than Halting: we need to check whether $\mathcal{M}_e(x) \downarrow$ for all possible x, so in a sense we have to solve infinitely many Halting problems.

We will provide a very elegant proof in a while, here is a more pedestrian approach that is a template for many other arguments of this kind.

[†]This is a special feature of computability theory, similar claims do not hold in other areas (say, proof theory or real analysis).

Warning

Last time we took a look at the following problem.

Problem:	**Halting Somewhere**
Instance:	A partial computable function f.
Question:	Is there some input x such that f on x converges?

This also looks like "infinitely many Halting problems." But in this case dovetailing shows that it's no worse than Halting.

Alas, this time there will be no such luck.

Consider a TM $\mathcal{M} = \mathcal{M}_e$. We can build another TM $\mathcal{M}'$ that, on input z, will run $\mathcal{M}$ on input e. If this computation converges, $\mathcal{M}'$ returns 0; otherwise the new machine diverges. So $\mathcal{M}'$ ignores its input, it either always converges on 0 or always diverges.

More precisely, there is an easily computable function f (a program transformation) such that $e' = f(e)$:

$$\{f(e)\}(z) \simeq \begin{cases} 0 & \text{if } \{e\}(e)\downarrow, \\ \uparrow & \text{otherwise.} \end{cases}$$

So $\{f(e)\}$ ignores its input, but that is by design.

Inheriting Undecidability

Write $K \subseteq \mathbb{N}$ for the Halting problem. Then

$$e \in K \iff \{f(e)\} \text{ is constant } 0 \iff f(e) \in \text{TOT}$$

Since f is computable, and K is undecidable, TOT cannot be decidable.

This is just a lower bound, we know TOT is not decidable, but we cannot say anything more constructive about its level of unsolvability. We will see it is not even semidecidable nor co-semidecidable.

Defining Computable Functions

Intuitively, one can often reason about computable functions informally by appealing to an algorithm in the sense of CS, or perhaps some claim along the lines of "there clearly is a Turing machine . . . "

Formalizing this type of argument is often too tedious to consider, so it would be nice if we had other tools available to establish computability in an absolutely rigorous manner.

Batten down the hatches.

Second Recursion Theorem

The next result is utterly amazing: it shows that we can solve functional equations of computable functions in mind-numbing generality.

Theorem (Kleene, Second Recursion Theorem, 1938)

Let $F : \mathbb{N}^{n+1} \twoheadrightarrow \mathbb{N}$ be a computable function. Then there exists an index $e^\star$ such that for all $\boldsymbol{x} \in \mathbb{N}^n$:

$$\{e^\star\}(\boldsymbol{x}) \simeq F(e^\star, \boldsymbol{x}).$$

Moreover, $e^\star$ can be computed effectively from an index for F.

Think of $e^\star$ as some program, x as input and F as an interpreter running $e^\star$ on x: then the claim is obvious, even for arbitrary $e^\star$. However, the theorem holds for arbitrary computable F!

No Way

At first glance, this sounds utterly wrong.

What if $F(e, x) \simeq \{e\}(x) + 1$?

Then we need $\{e^\star\}(x) \simeq \{e^\star\}(x) + 1$.

Solution: Any totally undefined function $\{e^\star\}$.

This is why it's important to deal with $\simeq$ rather than $=$.

In general it requires extra effort to show that a function obtained from the theorem is, say, total. A priori, all we get is a computable function, no more. And, this computable function may be undefined in many places, including everywhere.

Weird Consequences

The theorem has some rather strange consequences:

- Let $F(e, x) \simeq e$. Then $e^\star \simeq \{e^\star\}(x)$.
- Let $F(e, x) \simeq \{x\}(e)$. Then $\{x\}(e^\star) \simeq \{e^\star\}(x)$.

These F are clearly computable, so there is no way around these conclusions–no matter how strange they look.

These example are not particularly useful, but there are applications where the recursion theorem is absolutely critical.

Quines

In a similar vein, programs with the property that $e \simeq \{e\}()$ are referred to as quines, programs that print themselves.

The real challenge here is that one needs to deal with the idiosyncrasies (more often: idiocies) of a particular programming language.

Exercise

Write a quine in your favorite programming language.

The Fixed Point Version

Here is slightly more straightforward version of the recursion theorem that often comes in handy.

Corollary (Rogers, 1967)

Let $f : \mathbb{N} \to \mathbb{N}$ be a recursive function. Then there exists an index $e^\star$ such that

$$\{e^\star\}(\boldsymbol{x}) \simeq \{f(e^\star)\}(\boldsymbol{x}).$$

Proof.

Define $F(e, \boldsymbol{x}) \simeq \{f(e)\}(\boldsymbol{x})$.

□

Intuition

Think of f as some program transformation. For example, $f(e)$ could be the program that does sequential composition of program e with itself.

Then, no matter how f is chosen, there will always be a program $e^\star$ for which application of f does not change the I/O behavior. So every program transformation has a fixed point, but note that the fixed point will be partial in general (and may well be totally undefined).

Example: the function that replaces a factor p^i in x by p. Any idempotent operation will do.

The Horror Proof

We will prove the second recursion theorem directly. Define

$$h(e, \boldsymbol{x}) \simeq F(S_n^1(e, e), \boldsymbol{x})$$

Let $\widehat{h}$ be an index for h and set

$$e^\star := S_n^1(\widehat{h}, \widehat{h})$$

Then

$$\begin{aligned}
\{e^\star\}(\boldsymbol{x}) &\simeq \{S_n^1(\widehat{h}, \widehat{h})\}(\boldsymbol{x}) \\
&\simeq \{\widehat{h}\}(\widehat{h}, \boldsymbol{x}) \\
&\simeq h(\widehat{h}, \boldsymbol{x}) \\
&\simeq F(S_n^1(\widehat{h}, \widehat{h}), \boldsymbol{x}) \\
&\simeq F(e^\star, \boldsymbol{x})
\end{aligned}$$

□

Quoi?

This is one of the most infuriating proofs known to mankind. Every step is trivial equational reasoning, but the whole argument makes no sense.

The recursion theorist J. C. Owings called it “barbarically short” and “nearly incapable of rational analysis.”

Owings also suggested a way to make sense out of it: think of it as a diagonal argument that fails. Cute, but does not really help either.

Did I mention that Kleene was a genius?

I Know Who I Am

Here is a good intuitive way to think about RT. The typical definition of a computable function Q looks like so:

Q:

input x
some computation
`return ...`

But with the RT we can use an index $q = \widehat{Q}$ for Q inside the definition:

Q:

input x
some computation using q
`return ...`

Again . . .

This may sound breathtakingly wrong: after all, we are in the process of defining Q, so where the hell is the index q supposed to come from?

Sorry, but that is precisely what the recursion theorem says we can do, with impunity.

Maybe it helps to think of Q as being able to read its own source code? Try.

Application: Halting

Assume for the sake of a contradiction that the halting set $K = \{\, e \mid \{e\}(e) \downarrow \}$ is decidable.

Define Q by

```
input x
if check q ∈ K          // K decidable
then ↑
else return 0
```

Then $Q(q) \downarrow$ implies $Q(q) \uparrow$, and $Q(q) \uparrow$ implies $Q(q) \downarrow$.

$\uparrow\downarrow\downarrow\uparrow\uparrow\downarrow\downarrow\uparrow\uparrow\downarrow\downarrow\uparrow\uparrow\downarrow\uparrow\downarrow\downarrow\uparrow\uparrow\uparrow\uparrow\uparrow\uparrow\downarrow\uparrow\downarrow\downarrow\uparrow\uparrow\downarrow\downarrow\uparrow\uparrow\downarrow\uparrow\downarrow\downarrow\downarrow\uparrow\uparrow\uparrow\downarrow\uparrow\downarrow$

Application: Ackermann

Recall the definition of the Ackermann function, a function defined by a double recursion. We write z^+ for $z+1$.

$$A(0, y) = y^+$$
$$A(x^+, 0) = A(x, 1)$$
$$A(x^+, y^+) = A(x, A(x^+, y))$$

A is a perfect example of a Herbrand-Gödel-computable function and it is known that these are the same as Turing-computable. The Ackermann function was introduced as an example of a function that is computable, but not primitive recursive (it grows faster than any primitive recursive function).

It turns out that A is total, but that requires a proof (by double induction or induction to ω^2).

Ackermann and RT

Define Q by

```
input x, y
if x = 0
then return y + 1
elseif y = 0
        then return {q}(x−1, 1)
        else return {q}(x−1, {q}(x, y−1))
```

Then $Q = \{q\}$ is none other than the Ackermann function and we have another proof of its computability. Of course, one still has to work to demonstrate totality.

Enumerating RE Sets

Define

$$W_e = \{ x \mid \{e\}(x) \downarrow \} \subseteq \mathbb{N}$$

to be the eth semidecidable set, so $(W_e)_{e \geq 0}$ is an effective enumeration of all semidecidable sets.

We can use this enumeration to ask questions about semidecidable sets. For example, we might be interested in all non-empty semidecidable sets:

$$\mathsf{NE} = \{ e \mid W_e \neq \emptyset \}$$

This is essentially the same as Halting Somewhere, so we already know that NE is semidecidable.

Note that membership in NE is determined by the properties of W_e, not e itself (like being prime, having an odd number of digits, . . .)

Index Sets

Any such set is called an index set.

More precisely, let's say that $P \subseteq \mathbb{N}$ is a non-trivial index set if

- $W_e = W_{e'}$ implies $e \in P \Leftrightarrow e' \in P$,
- $e_Y \in P$ and $e_N \notin P$ for some e_Y and e_N.

Since an index set is just a set of natural numbers we can ask whether it is decidable, semidecidable, co-semidecidable, ...

Application: Rice's Theorem

Typical examples are

- W_e is empty
- W_e is finite
- $W_e = \mathbb{N}$
- W_e is decidable

Claim

All of these index sets seem undecidable; in other words, all these properties of semidecidable sets are undecidable.

Rice's Theorem

One can slog one's way through separate proofs for all these properties; good exercise. For example $W_e = \mathbb{N}$ is just another way of saying $e \in$ TOT.

But there is a universal tool that kills them all off, in one fell swoop.

Theorem (Rice 1953)

Every non-trivial index set is undecidable. In other words, every non-trivial property of semidecidable sets is undecidable.

Proof of Rice

Let P be a non-trivial index set.

For the sake of a contradiction assume P is decidable.

Define Q by

```
input x
if q ∈ P                    // P decidable
then return {e_N}(x)
else return {e_Y}(x)
```

But then $q \in P$ implies $Q \simeq \{e_N\}$ and thus $q \notin P$.

On the other hand, $q \notin P$ implies $Q \simeq \{e_Y\}$ and thus $q \in P$.

□

Owings' Failed Diagonal Argument

Owings also suggested a way to make sense out of the proof of the recursion theorem: think of it as a diagonal argument that fails.

In a diagonal argument we have an infinite matrix S over some set A:

$$S : \mathbb{N} \times \mathbb{N} \to A$$

We can think of the rows as infinite sequences over A, so S is a table of infinite sequences.

We have some operation α on A such that the sequence obtained by applying α to the diagonal $\big(\alpha(S(i,i))\big)_{i \geq 0}$ is not in S.

Failure

Suppose f is computable and well-behaved on indices: $\{e\} \simeq \{e'\}$ implies $\{f(e)\} \simeq \{f(e')\}$; so f preserves semantics.

Define a matrix S of computable functions by

$$S = \big(\{\{i\}(j)\}\big)_{i,j}$$

Define $\alpha(\{e\}) = \{f(e)\}$.

In this case, the diagonal sequence $(S(i,i))_{i\geq 0}$ as well as its image under α is still a row in S.

The intersection of this row and the diagonal is what we are looking for.