CDM

Wild Computation

Klaus Sutner

Carnegie Mellon University

Fall 2023

1 General Recursion

2 Evaluation

3 The Busy Beaver Problem

4 Insane Growth

Ackermann's Function (1928)

Primitive recursive functions uses only a single variable. One might suspect that recursion over multiple variables could potentially produce more complicated functions, but one needs to be careful: who knows, maybe there is some clever way to express a multiple recursion in terms of a single one.

Here is a classical example: the Ackermann function $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined by double recursion. We write x^+ instead of $x + 1$.

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

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

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

On the surface, this looks more complicated than primitive recursion. We need to make sure that there really is no trick to rewrite this as a single recursion.

Family Perspective–Currying

It is useful to think of Ackermann's function as a family of unary functions $(A_x)_{x \geq 0}$ where $A_x(y) = A(x, y)$ ("level x of the Ackermann hierarchy").

The definition then looks like so:

$$A_0(y) = y^+$$

$$A_{x^+}(0) = A_x(1)$$

$$A_{x^+}(y^+) = A_x(A_{x^+}(y))$$

From this it follows easily by induction that

Lemma

Each of the functions A_x is primitive recursive (and hence total).

The Bottom Hierarchy

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

$$A(1, y) = y^{++}$$

$$A(2, y) = 2y + 3$$

$$A(3, y) = 2^{y+3} - 3$$

$$A(4, y) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} - 3$$

The first 4 levels of the Ackermann hierarchy are easy to understand, though A_4 starts causing problems: the stack of 2's in the exponentiation has height $y + 3$.

Tetration

A_4 is usually called super-exponentiation or tetration and often written ${}^n a$ or $a{\uparrow\uparrow}n$.

$$a{\uparrow\uparrow}n = \begin{cases} 1 & \text{if } n = 0, \\ a^{a{\uparrow\uparrow}(n-1)} & \text{otherwise.} \end{cases}$$

For example,

$$A(4,3) = 2{\uparrow\uparrow}6 - 3 = 2^{2^{65536}} - 3$$

an uncomfortably large number (we'll see much worse in a moment).

Alas, if we continue just a few more levels, darkness befalls.

$$A(5,y) \approx \text{super-super-exponentiation}$$

$$A(6,y) \approx \text{an unspeakable horror}$$

$$A(7,y) \approx \text{speechlessness}$$

For level 5, one can get some vague understanding of iterated super-exponentiation, $A(5,y) = (\lambda z.z \uparrow\uparrow y+3)^{y+3}(1) - 3$ but things start to get murky.

At level 6, we iterate over the already nebulous level 5 function, and things really start to fall apart.

At level 7, Wittgenstein comes to mind: "Wovon man nicht sprechen kann, darüber muss man schweigen."*

*Whereof one cannot speak, thereof one must be silent. *Tractatus Logico-Philosophicus*

Theorem

The Ackermann function dominates every primitive recursive function f in the sense that there is a k such that

$$f(\boldsymbol{x}) < A(k, \max \boldsymbol{x}).$$

Hence A is not primitive recursive.

Sketch of proof.

Since we are dealing with a rectype, we can argue by induction on the buildup of f.

The atomic functions are easy to deal with.

The interesting part is to show that the property is preserved during an application of composition and of primitive recursion. Alas, the details are rather tedious.

Ackermann and Union/Find

One might think that the only purpose of the Ackermann function is to refute the claim that computable is the same as p.r. Surprisingly, the function pops up in the analysis of the Union/Find algorithm (with ranking and path compression).

The running time of Union/Find differs from linear only by a minuscule amount, which is something like the inverse of the Ackermann function.

But in general anything beyond level 3.5 of the Ackermann hierarchy is irrelevant for practical computation.

Exercise

Read an algorithms text that analyzes the run time of the Union/Find method.

But Is It Computable?

Here is an entirely heuristic argument: we can write a tiny bit of C code that implements the Ackermann function (assuming that we have infinite precision integers).

```
int acker(int x, int y)
{
  return( x ? (acker(x-1, y ? acker(x, y-1) : 1)) : y+1 );
}
```

All the work of organizing the nested recursion is easily handled by the compiler and the execution stack. So this provides overwhelming evidence that the Ackermann function is intuitively computable.

Proofs by Hashing

We could memoize the values that are computed during a call to $A(a, b)$: build a hash table H such that $H[x, y] = z$ whenever an intermediate result $A(x, y) = z$ is discovered during the computation.

In practice, this helps in computing a few small values of A, but does not go very far.

More interesting is the following: suppose we call $A(a, b)$ and obtain result c, producing a hash table H as a side effect.

Claim: H provides a proof that $A(a, b) = c$.

Not a proof in the classical sense, but an object that makes it possible to perform a simple coherence check and conclude that the value c is indeed correct.

Checking

We have to check the following properties everywhere in H:

$$
\begin{aligned}
H[0,y] = z &\quad \text{implies} \quad z = y+1 \\
H[x^+,0] = z &\quad \text{implies} \quad H[x,1] = z \\
H[x^+,y^+] = z &\quad \text{implies} \quad H[x,z'] = z \text{ where } z' = H[x^+,y]
\end{aligned}
$$

The whole check comes down to performing $O(N)$ table lookups where N is the number of entries in H.

Once the table is verified, we check $H[a,b] = c$. Done.

Obvious Question: how much do we have to add to primitive recursion to capture the Ackermann function?

As it turns out, we need just one modification: we have to allow unbounded search: a type of search where the property we are looking for is still primitive recursive, but we don't know ahead of time how far we have to go.

Proposition

There is a primitive recursive relation R such that

$$A(a,b) = \mathsf{fst}\big(\min\big(\, z \mid R(a,b,z)\,\big)\big)$$

Here $\mathsf{fst}(s)$ is a decoding function that extracts the first element of a coded sequence, expressed as a code number s.

Sketch of proof. Think of z as a pair $\langle c, h\rangle$ where h encodes the hash table H from above, and $c = H[a,b]$.

R performs the coherence test described above and is clearly primitive recursive. □

Here is a more direct, computational description.

The computation of, say, $A(2,1)$ can be handled in a very systematic fashion: always unfold the rightmost subexpression.

$$A(2,1) = A(1, A(2,0)) = A(1, A(1,1)) = A(1, A(0, A(1,0))) = \ldots$$

Note that the A's and parens are just syntactic sugar, a better description would be

$$2,1 \rightsquigarrow 1,2,0 \rightsquigarrow 1,1,1 \rightsquigarrow 1,0,1,0 \rightsquigarrow 1,0,0,1 \rightsquigarrow 1,0,2 \rightsquigarrow 1,3 \rightsquigarrow 0,1,2$$
$$\rightsquigarrow 0,0,1,1 \rightsquigarrow 0,0,0,1,0 \rightsquigarrow 0,0,0,0,1 \rightsquigarrow 0,0,0,2 \rightsquigarrow 0,0,3 \rightsquigarrow 0,4 \rightsquigarrow 5$$

We can model these steps by a list function Δ defined on sequences of naturals (or, we could use a stack).

$$\varDelta(\ldots, 0, y) = (\ldots, y^+)$$

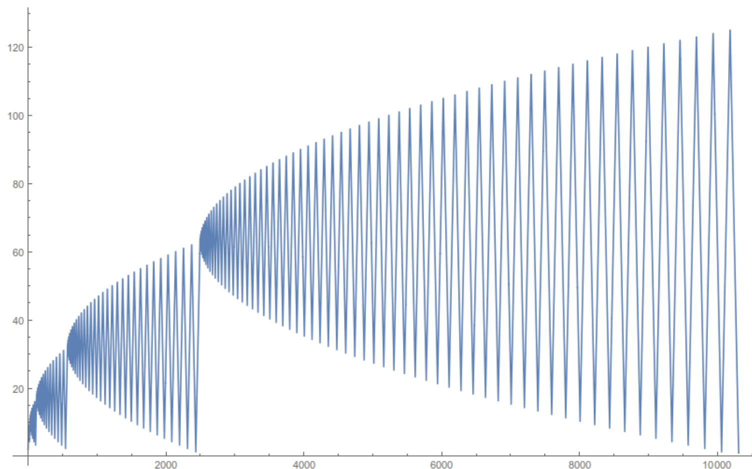
$$\varDelta(\ldots, x^+, 0) = (\ldots, x, 1)$$

$$\varDelta(\ldots, x^+, y^+) = (\ldots, x, x^+, y)$$

Since A is total, there is some time t for any a and b such that

$$\varDelta^t(a, b) = (c)$$

Clearly this condition is primitive recursive in (a, b, c, t).



The computation takes 10307 steps, the plot shows the lengths of the list.

1 General Recursion

2 **Evaluation**

3 The Busy Beaver Problem

4 Insane Growth

Evaluation

Very rapidly growing functions such as the Ackermann function are one reason primitive recursion is not strong enough to capture computability. Here is another obstruction: we really need to deal with partial functions.

Recall the evaluation operator for our PR terms:

$$\mathsf{eval}(\tau, \boldsymbol{x}) = \text{ value of } [\![\tau]\!] \text{ on input } \boldsymbol{x}$$

It is clear that eval is intuitively computable (take a compilers course). In fact, it is not hard to implement in eval in any modern programming language.

Question: Could eval be primitive recursive?

A useless answer would be to say **no**, the types don't match.

The first argument of eval is a term τ in our PR language, so our first step will be to replace τ by an index $\widehat{\tau} \in \mathbb{N}$.

The index $\widehat{\tau}$ will be constructed in a way that makes sure that all the operations we need on indices are clearly primitive recursive.

The argument vector $\boldsymbol{x} \in \mathbb{N}^n$ will also be replaced by its sequence number $\langle x_1, \ldots, x_n \rangle$. Hence we will be able to interpret eval as a function of type

$$\mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

and this function might potentially be primitive recursive.

Coding PR

Here is one natural way of coding primitive recursive terms as naturals:

term	code
0	$\langle 0, 0 \rangle$
P^n_i	$\langle 1, n, i \rangle$
S	$\langle 2, 1 \rangle$
$\mathsf{Prec}[h, g]$	$\langle 3, n, \widehat{h}, \widehat{g} \rangle$
$\mathsf{Comp}[h, g_1, \ldots, g_n]$	$\langle 4, m, \widehat{h}, \widehat{g_1}, \ldots, \widehat{g_n} \rangle$

Thus for any index e, the first component $\mathsf{fst}(e)$ indicates the type of function, and $\mathsf{snd}(e)$ indicates the arity.

There is nothing sacred about this particular way of coding PR terms, there are many other, equally natural ways.

Diagonalization

Now suppose eval is p.r., and define the following function

$$f(x) := \mathsf{eval}(x, x) + 1$$

This may look weird, but certainly f is also p.r. and must have an index e. But then

$$f(e) = \mathsf{eval}(e, e) + 1 = f(e) + 1$$

and we have a contradiction.

So eval is another example of an intuitively computable function that fails to be primitive recursive.

This example may be less sexy than the Ackermann function, but it appears in similar form in other contexts.

Partial Functions

How do we avoid the problem with eval?

The only plausible solution appears to be to admit partial functions, functions that, like eval, are computable but may fail to be defined on some points in their domain. In this case, $\text{eval}(e, e)$ is undefined.

For a CS person, this is a fairly uncontroversial idea: everyone who has ever written a sufficiently sophisticated program will have encountered divergence: on some inputs, the program simply fails to terminate.

What may first seem like a mere programming error, is actually a fundamental feature of computable functions.

We presented the last argument in the context of primitive recursive functions, but note that the same reasoning also works for any clone of computable functions—as long as

- successor and eval both belong to the clone, and
- each function in the clone is represented by an index.

But then eval must already be partial, no matter what the details of our clone are.

General computability requires partial functions, basta.

Fiat Halting

Since any general model of computation must deal with partial functions, it is entirely natural to ask whether a given function f is defined on some particular input x.

Another reasonable question would be to ask whether f is total; or even whether f is nowhere defined.

So we automatically run into the Halting Problem, the first example of a perfectly well-defined question that turns out to be undecidable.

We write

$$f : A \nrightarrow B$$

for a partial function from A to B.

Terminology:

domain	$\text{dom}\, f = A$
codomain	$\text{cod}\, f = B$
support	$\text{spt}\, f = \{\, a \in A \mid \exists\, b\, f(a) = b \,\}$

Warning: Some misguided authors use "domain of definition" instead of support, and then forget the "of definition" part.

Suppose we have a partial function $f : \mathbb{N} \nrightarrow \mathbb{N}$. We could try to turn f into a total function $F : \mathbb{N} \nrightarrow \mathbb{N}$ by setting

$$F(x) = \begin{cases} f(x) + 1 & \text{if } x \in \text{spt}\, f \\ 0 & \text{otherwise.} \end{cases}$$

F clearly is total, and we can easily recover f from it.

In set theory la-la land there is no problem at all. But this construction is not very useful for us: there are computable f such that F fails to be computable.

Kleene's Notation

Given a clone of computable functions, such as the primitive recursive ones, we write

$$\{e\}$$

for the eth function in the collection, $e \geq 0$. Here the index e is a sequence number, but it is helpful to think of it as a program (in some suitable language).

Since these functions are partial in general we have to be a bit careful and write

$$\{e\}(x) \simeq y$$

to indicate that $\{e\}$ with input x returns output y. This notation is a bit sloppy, arguably we should also indicate the arity of the function–but for us that's overkill.

More Convergence

To express convergence we also write

$$\{e\}(x)\downarrow$$

if $\{e\}$ on input x terminates and produces some output, and

$$\{e\}(x)\uparrow$$

when the computation fails to terminate.

For example, Kleene equality $\{e\}(x) \simeq \{e'\}(x)$ should be interpreted as:

- either $\{e\}(x)\downarrow$ and $\{e'\}(x)\downarrow$ and the output is the same; or
- $\{e\}(x)\uparrow$ and $\{e'\}(x)\uparrow$.

1 General Recursion

2 Evaluation

3 **The Busy Beaver Problem**

4 Insane Growth

In 1962, Tibor Rado described a now famous problem in computability. Consider Turing machines on tape alphabet $\Sigma = \{0, 1\}$ (where 0 is the blank symbol) and n states.

> **Question:** What is the largest number of 1's any such machine can write on an initially blank tape, and then halt?

Halting is crucial, otherwise we could trivially write infinitely many 1's.

Other Variants

Rado's original question is actually slightly arbitrary, here are two versions more firmly rooted in computability theory.

Time Complexity What is the largest number of moves a halting n-state machine can make?

Space Complexity What is the largest number of tape cells a halting n-state machine can use?

Incidentally, it is standard practice to ignore the halting state in the count, so n means "n ordinary states plus one halting state."

We write $\mathrm{BB_T}(n)$ for largest number of steps of any halting n-state machine and refer to $\mathrm{BB_T}$ as the Busy Beaver function.

We will also consider the original version of the problem and write $\mathrm{BB_W}(n)$ for the largest number of 1's written by any halting n-state machine.

Clearly, $\mathrm{BB_T}(n) \geq \mathrm{BB_W}(n)$, but the former has the advantage of relating more directly to the Halting Problem, which one would suspect to be the central issue with busy beaver functions.

Busy Beaver $n = 1$

$\text{BB}_{\mathrm{T}}(1) = 1$

To see this, note that any attempt to make a second move would already lead to an infinite loop.

Similarly, $\text{BB}_{\mathrm{W}}(1) = 1$.

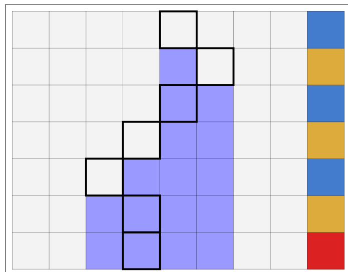
Busy Beaver $n = 2$

Amazingly, the answer is no longer obvious: $\mathrm{BB_W}(2) = 4$ and $\mathrm{BB_T}(2) = 6$ with the same champion.

	0	1
p	(q,1,R)	(q,1,L)
q	(p,1,L)	halt

$$p0 \vdash 1q0 \vdash p11 \vdash q011 \vdash p0111 \vdash 1q111$$

Orbit

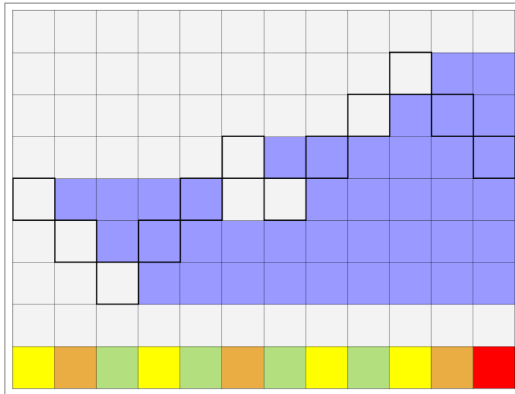


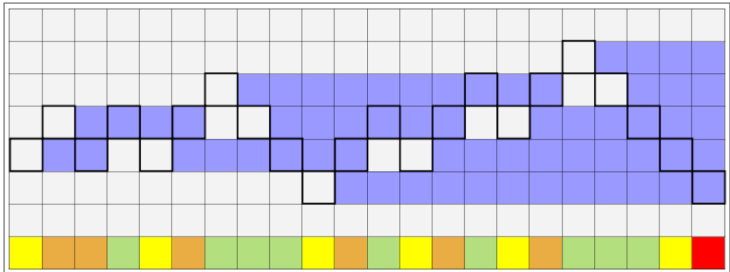
Here things start to get messy: there are $4\,826\,809$ Turing machines to consider.

Exploiting isomorphisms, filtering out machines where all 4 states are reachable (in the diagram, not necessarily the computation on empty tape), and checking for halting we get down to $405\,072$

From the last group we can pick out the champions.

Write-Champion





How bad can it be?

The number of machines quickly becomes very difficult to manage:

n	#machines
4	6 975 757 441
5	16 679 880 978 201

As usual, the problem is not isomorph-rejection (which requires constructing all machines first), but to onlt build non-isomophic ones to begin with. And, given these numbers, it won't make much of dent no matter what.

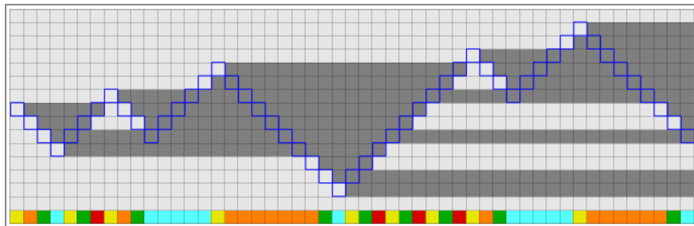
The Marxen-Buntrock Machine

The current champion machine was found by Marxen and Buntrock, and its discovery is a small miracle. Here is the table of the machine. Clearly all 5 states plus the halt state are reachable in the diagram.

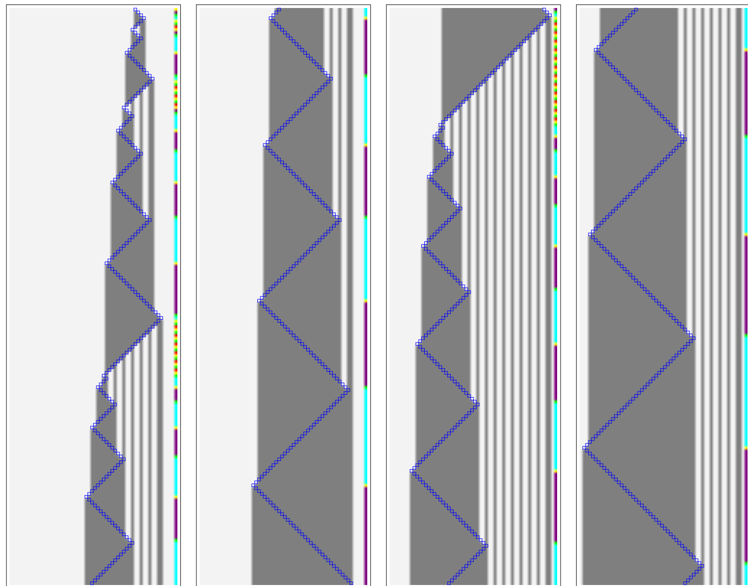
	0	1
1	(2,1,R)	(3,1,L)
2	(3,1,R)	(2,1,R)
3	(4,1,R)	(5,0,L)
4	(1,1,L)	(4,1,L)
5	halt	(1,0,L)

Of course, that's nowhere near enough: they need to appear in the computation on empty tape.

50 Steps



400 Steps



Looking at a run of the Marxen-Buntrock machine for a few hundred or even a few thousand steps one invariably becomes convinced that the machine never halts: the machine zig-zags back and forth, sometimes building solid blocks of 1's, sometimes a striped pattern.

Whatever the details, the machine seems to be in a "loop" (not a an easy concept to clarify for Turing machines). Bear in mind: there are only 5 states, there is no obvious method to code an instruction such as "do some zig-zag move 1 million times, then stop".

Still, this machine stops after 47 176 870 steps on output $10(100)^{4097}$.

Why is this Hard?

There are several fundamental obstructions to computing busy beaver numbers, in increasing levels of depth.

- Brute-force search quickly becomes infeasible, even for single-digit values of n.
- The Halting conundrum: Even if we could somehow deal with combinatorial explosion, there is the problem that we don't know if a machine will ever halt – it might just keep running forever.
- Reasoning about the behavior of Turing machines in a formal system like Peano arithmetic or Zermelo-Fraenkel set theory is necessarily of limited use.

State of the Art

n	$\mathsf{BB}_{\mathrm{T}}(n)$	$\mathsf{BB}_{\mathrm{W}}(n)$
1	1	1
2	6	4
3	21	6
4	107	13
5	$\geq 47\,176\,870$	≥ 4098
6	$> 7.4 \times 10^{36\,534}$	$> 3.5 \times 10^{18\,267}$

Concrete values are available for $n \leq 4$; beyond that, we only have bounds. And these bounds soon get ridiculous:

$$\mathsf{BB}_{\mathrm{T}}(7) > 10^{2 \cdot 10^{10^{10^{18\,705\,353}}}}$$

Alas, these results are not as robust as one would like them to be, see Harland 16 for a critique.

6-State Busy Beaver

For $n = 6$ all hell breaks loose.

The raw search space here has size $59\,604\,644\,775\,390\,625$, but this can be improved a bit exploiting symmetries and reachability.

Halting gets very messy here, though: there is no good heuristic as to when the run should be truncated (leading to the conlusion that the machine is not halting).

A new machine by Pavel Kropitz, takes about $10 \uparrow\uparrow 15$ steps to halt. A more precise bound is

$$10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{4.023873729}}}}}}}}}}}}}}}$$

You're welcome.

Small Machines can be Complicated

Define a (n, k)-Turing machine to be a TM that has n states and a tape alphabet of size k.

Clearly, there is a Busy Beaver problem for (n, k) TMs, the standard problem is just the special case $(n + 1, 2)$. Very little is known about the general case.

In a similar spirit, one can ask for small values of n and k if there is a universal (n, k) machine. One would expect a trade-off between n and k. Some values where universal machines are known to exist are

$$(24, 2), (10, 3), (7, 4), (5, 5), (4, 6), (3, 10), (2, 18), (2, 5)$$

Exercise

Derive the transition table of the 3-state Busy Beaver machine. Give an intuitive explanation of how this machine works.

Exercise

Prove that the last machine is indeed the champion: no other halting 3-state machine writes more than 6 ones.

Exercise (Hard)

Find the Busy Beaver champion for $n = 4$.

Exercise (Extremely Hard)

Organize a search for the Busy Beaver champion for $n = 5$.

1 General Recursion

2 Evaluation

3 The Busy Beaver Problem

4 **Insane Growth**

Subsequence Order

Recall the subsequence ordering on words where $u = u_1 \ldots u_n$ precedes $v = v_1 v_2 \ldots v_m$ if there exists a strictly increasing sequence $1 \leq i_1 < i_2 < \ldots i_n \leq m$ of positions such that $u = v_{i_1} v_{i_2} \ldots v_{i_n}$.

In symbols: $u \sqsubseteq v$.

In other words, we can erase some letters in v to get u. Note that it is easy to check for subsequences in linear time.

Subsequence order is never total unless the alphabet has size 1.

One nice features of subsequence order is that is entirely independent of any underlying order of the alphabet (unlike, say, lexicographic or length-lex order).

Warmup: Antichains

An antichain in a partial order is a sequence x_0, x_1, ..., x_n, ... of elements such that x_i and x_j are incomparable for $i < j$.

Example

Consider the powerset of $[n] = \{1, 2, \ldots, n\}$ with the standard subset ordering. How does one construct a long antichain?

For example, $x_0 = \{1\}$ is a bad idea, and $x_0 = [n]$ is even worse.

What is the right way to get a long antichain?

Theorem (Higman 1952)

Every antichain in the subsequence order is finite.

Proof. Here is the Nash-Williams proof (1963): assume there is an infinite antichain. Then there is a non-increasing sequence $\boldsymbol{x} = (x_n)$ in the sense that $i < j$ implies that $x_i \not\sqsubseteq x_j$.

Choose the minimal such sequence in the sense that x_n is the length-lex minimal word such that $x_0, x_1, \ldots, x_n$ starts a non-increasing sequence.

There must be a letter a such that the subsequence $x_{n_j} = a\, y_j$, $j \geq 0$, of words starting with a is infinite. Let $k = n_0$ and define a new sequence

$$\boldsymbol{z} = x_0, x_1, \ldots, x_{k-1}, y_0, y_1, \ldots$$

One can check that the new sequence $\boldsymbol{z}$ is again non-increasing.

But $\boldsymbol{z}$ violates the minimality constraint on $\boldsymbol{x}$ at position k, contradiction.

Note that this proof is highly non-constructive. We are essentially performing surgery on a branch in an infinite tree that exists by assumption. A lot of work has gone into developing more constructive versions of the theorem, but things get a bit complicated.

See Seisenberger.

We are using 1-indexing. For a finite or infinite word x define the ith block of x to be the factor (of length $i+1$) of x:

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

Note this makes sense only for $i \leq |x|/2$ when x is finite. We will always tacitly assume that this bound holds.

Bizarre Definition: A word is self-avoiding if, for all $i < j$, the block $x[i]$ fails to be a subsequence of block $x[j]$.

For example,

$abbbaaaa$	is self-avoiding
$abbbaaab$	is not self-avoiding

The following is an easy consequence of Higman's theorem.

Theorem

Every self-avoiding word is finite.

If there were an infinite self-avoiding word $x \in \Sigma^\omega$, the collection $\{ x[i] \mid i \geq 1 \}$ of all its blocks would form an infinite antichain.

How Long?

Write Σ_k for an alphabet of size k.

By the last theorem and König's lemma, the set S_k of all finite self-avoiding words over Σ_k must itself be finite.

But then we can define the following max-length function:

$$\alpha(k) = \max\big(|x| \mid x \in S_k \big)$$

So $\alpha(k)$ is the length of the longest self-avoiding word over Σ_k.

Clearly, α is total and it is strictly increasing.

Moreover, α is easily computable, there is a very straightforward algorithm to determine the value of $\alpha(k)$.

Note that any prefix of a self-avoiding word must also be self-avoiding. This produces a simple, brute-force algorithm to compute α.

- At round 0, define $S = \{\varepsilon\}$.
- In each round, extend all words in $x \in S$ by all letters $a \in \Sigma_k$. If xa is still self-avoiding, keep it; otherwise toss it.
- When S becomes empty at round $n + 1$, return $\alpha(k) = n$.

Each step 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, the tree would be infinite and thus have an infinite branch, corresponding to an infinite self-avoiding word; contradiction.

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 $\alpha(1) = 3$: the first time a word contains 2 blocks, it is not longer self-avoiding.

A little fumbling (or writing a program) shows that $\alpha(2) = 11$, as witnessed by $abbbaaaaaa$ and $abbbaaaaab$ and their duals.

Challenge for Hackers

Write a program in your favorite fast language that extends the table, ideally by a column or a row.

I suspect the former is feasible, the latter may be tricky.

And $k = 3$?

Alas, $\alpha(3)$ is a bit harder to describe. We will use a slight variant of the Ackermann function for this purpose.

$$B_1(x) = 2x$$

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

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

Just like the Ackermann function, B_5 essentially makes no sense to mere mortals, its growth rate is stupendous.

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

This is an incomprehensibly, mind-numbingly large number.

Never mind the 158386, it's the 7198 that kills any chance of understanding, at least roughly, what this means.

Smelling salts, anyone?

It is truly surprising that a function like α with a really simple algorithm should exhibit this kind of growth.

And, of course, there is $\alpha(\alpha(3))$, $\alpha(\alpha(\alpha(3)))$, and so on. Or how about

$$\alpha^{\alpha(3)}(3)$$

At this point one might wonder whether our whole approach to computability is perhaps a bit off—we certainly did not intend to deal with monsters like α.

Alas, as it turns out, this is a feature, not a bug: all reasonable definitions of computability admit things like α, and worse. Far worse.

It is a fundamental property of computable functions that some of them have absurd growth rates.