Automata Theory and Formal Languages

Summer Term 2026

19th Lecture

The Busy Beaver Problem

Prof. Dr. Matthias Mnich *Institute for Algorithms and Complexity*

The Busy Beaver Problem

- *M(n)*: set of Turing machines with *n* states w/ $\Sigma = \{0,1\}$

Based on work of Bram van Heuveln

The Busy Beaver Problem

- *M(n)*: set of Turing machines with *n* states w/ $\Sigma = \{0,1\}$
- *[k]*: Configuration of having a block of *k* consecutive *1*'s on an otherwise all-*0* tape, and with head at leftmost *1*.

The Busy Beaver Problem

- *M(n)*: set of Turing machines with *n* states w/ $\Sigma = \{0,1\}$
- *[k]*: Configuration of having a block of *k* consecutive *1*'s on an otherwise all-*0* tape, and with head at leftmost *1*.
- *e*: empty tape (all 0's)

The Busy Beaver Problem

- *M(n)*: set of Turing machines with *n* states w/ $\Sigma = \{0,1\}$
- *[k]*: Configuration of having a block of *k* consecutive *1*'s on an otherwise all-*0* tape, and with head at leftmost *1*.
- *e*: empty tape (all 0's)
- *Σ(M)*: *Σ(M) = k* if machine *M,* when started on *e*, halts with *[k]*. Otherwise, *Σ(M) = 0*.

The Busy Beaver Problem

- *M(n)*: set of Turing machines with *n* states w/ $\Sigma = \{0,1\}$
- *[k]*: Configuration of having a block of *k* consecutive *1*'s on an otherwise all-*0* tape, and with head at leftmost *1*.
- *e*: empty tape (all 0's)
- *Σ(M)*: *Σ(M) = k* if machine *M,* when started on *e*, halts with *[k]*. Otherwise, *Σ(M) = 0*.

Busy Beaver Problem:

Find $\Sigma(n) = \max\{\Sigma(M) | M \in M(n)\}$

Variations

One can define a variety of Busy Beaver problems:

Variations

One can define a variety of Busy Beaver problems:

- Do we use the quadruple or quintuple formalization?

Variations

One can define a variety of Busy Beaver problems:

- Do we use the quadruple or quintuple formalization?
- Do we use a binary alphabet or more than 2 symbols?

Variations

One can define a variety of Busy Beaver problems:

- Do we use the quadruple or quintuple formalization?
- Do we use a binary alphabet or more than 2 symbols?
- How does the machine come to a halt?
  - *Explicit halting state*: machine halts by a transition to an explicit halting (in which case halting state does not get counted towards n)
  - *Implicit halting state*: machine halts by the lack of a transition for current state and symbol

Variations

One can define a variety of Busy Beaver problems:

- Do we use the quadruple or quintuple formalization?
- Do we use a binary alphabet or more than 2 symbols?
- How does the machine come to a halt?
  - *Explicit halting state*: machine halts by a transition to an explicit halting (in which case halting state does not get counted towards n)
  - *Implicit halting state*: machine halts by the lack of a transition for current state and symbol
- Are there any restrictions on the output configuration?
  - *Standard configuration*: head positioned at leftmost 1 (or other non- blank symbol) of consecutive string of 1's on otherwise empty tape
  - *Anything goes* (head does not need to be at leftmost 1, and 1's may be scattered all over tape)

Variations

One can define a variety of Busy Beaver problems:

- Do we use the quadruple or quintuple formalization?
- Do we use a binary alphabet or more than 2 symbols?
- How does the machine come to a halt?
  - *Explicit halting state*: machine halts by a transition to an explicit halting (in which case halting state does not get counted towards n)
  - *Implicit halting state*: machine halts by the lack of a transition for current state and symbol
- Are there any restrictions on the output configuration?
  - *Standard configuration*: head positioned at leftmost 1 (or other non- blank symbol) of consecutive string of 1's on otherwise empty tape
  - *Anything goes* (head does not need to be at leftmost 1, and 1's may be scattered all over tape)

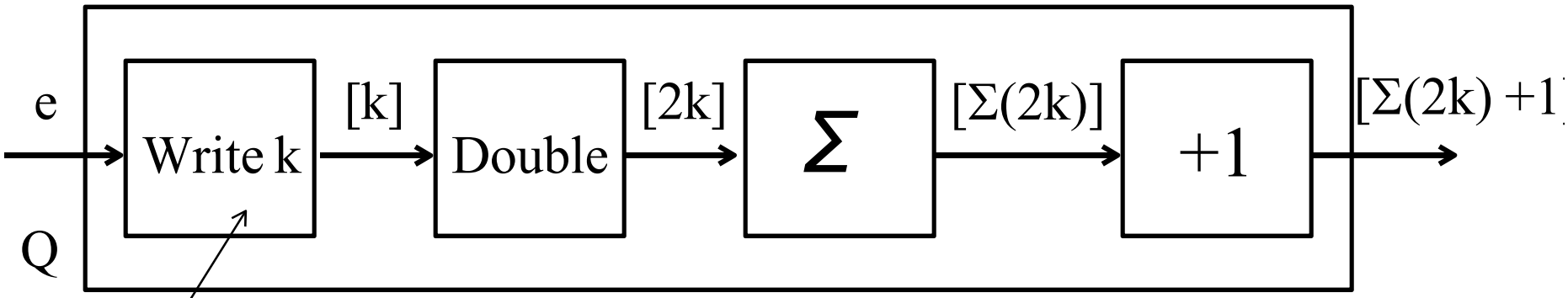
Let's stick to binary alphabet and a standard configuration.
But all proofs can be modified to accommodate all other types.

$\Sigma(n)$ is Turing-uncomputable

Proof by Contradiction: Suppose there is some Turing-Machine *Σ* that computes *Σ(n).*
Then consider the following Turing-machine Q, where k is the number of states of the last 3 components:

$\Sigma(n)$ is Turing-uncomputable

Proof by Contradiction: Suppose there is some Turing-Machine *Σ* that computes *Σ(n).*
Then consider the following Turing-machine Q, where k is the number of states of the last 3 components:



Can be implemented using k states. So, Q has $2k$ states … and outputs $\Sigma(2k) + 1$ when started on an empty tape – contradiction.

The Shifting Problem

S(M): *S(M) = m* if machine *M,* when started on *e*, takes *m* steps before it halts with *[k]* for some *k*. Otherwise, *S(M) = 0*.

The Shifting Problem

S(M): *S(M) = m* if machine *M,* when started on *e*, takes *m* steps before it halts with *[k]* for some *k*. Otherwise, *S(M) = 0*.

Shifting Problem:
Find $S(n) = \max\{S(M) | M \in M(n)\}$

$S(n)$ is Turing-Uncomputable

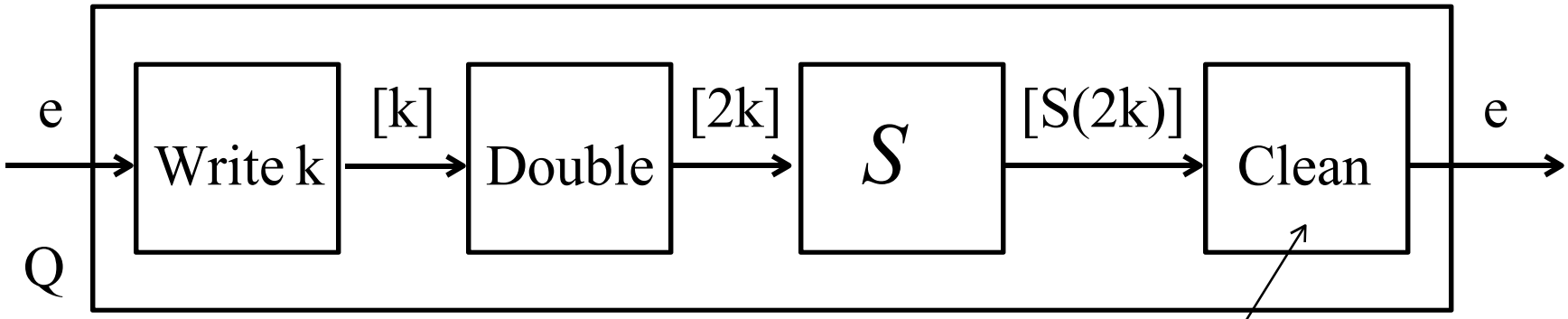
Proof by contradiction: Suppose there is some Turing machine *S* that computes *S(n).*

Then consider the following Turing machine Q, where k is the number of states of the last 3 components:

$S(n)$ is Turing-Uncomputable

Proof by contradiction: Suppose there is some Turing machine *S* that computes *S(n).*

Then consider the following Turing machine Q, where k is the number of states of the last 3 components:



Cleaning the tape will take at least $S(2k)$ steps. So Q has $2k$ states but takes more than $S(2k)$ steps before halting – contradiction.

$S(n)$ is Uncomputable: Alternative Proof

$S(n)$ is Uncomputable: Alternative Proof

The uncomputability of *S(n)* can also be derived from the uncomputability of *Σ(n)*:

$S(n)$ is Uncomputable: Alternative Proof

The uncomputability of *S(n)* can also be derived from the uncomputability of *Σ(n)*:

Suppose *S(n)* is computable. Then *Σ(n)* can be determined simply by running all of the finitely many Turing machines M with n states, starting on e.

$S(n)$ is Uncomputable: Alternative Proof

The uncomputability of *S(n)* can also be derived from the uncomputability of *Σ(n)*:

Suppose *S(n)* is computable. Then *Σ(n)* can be determined simply by running all of the finitely many Turing machines M with n states, starting on e.

If a Turing machine is still running after *S(n)* steps, you know it is a non-halter. For all the others *Σ(M)* can be determined, and now simply take the max.

Computability, uncomputability, and the Church-Turing Thesis

Notice that the proof on the previous slide establishes that *S(n)* is uncomputable, rather than Turing-uncomputable.

Computability, uncomputability, and the Church-Turing Thesis

Notice that the proof on the previous slide establishes that *S(n)* is uncomputable, rather than Turing-uncomputable.

- In particular, the proof assumed that *Σ(n)* is uncomputable, rather than just Turing-uncomputable.

Computability, uncomputability, and the Church-Turing Thesis

Notice that the proof on the previous slide establishes that *S(n)* is uncomputable, rather than Turing-uncomputable.

- In particular, the proof assumed that *Σ(n)* is uncomputable, rather than just Turing-uncomputable.
- So, the proof appeals to the Church-Turing Thesis, which states that anything that is computable is Turing-computable (so: since we know that *Σ(n)* is Turing-uncomputable, *Σ(n)* is not computable).

Computability, uncomputability, and the Church-Turing Thesis

Notice that the proof on the previous slide establishes that *S(n)* is uncomputable, rather than Turing-uncomputable.

- In particular, the proof assumed that *Σ(n)* is uncomputable, rather than just Turing-uncomputable.
- So, the proof appeals to the Church-Turing Thesis, which states that anything that is computable is Turing-computable (so: since we know that *Σ(n)* is Turing-uncomputable, *Σ(n)* is not computable).

In many of the other proofs we make similar use of the Church-Turing Thesis, i.e. from now on by computability we mean Turing-computability.

Problem Reductions

The proof from 2 slides ago is a good example of reducing one problem into another.

Problem Reductions

The proof from 2 slides ago is a good example of reducing one problem into another.

- Problem A reduces to problem B if being able to solve problem B allows you to solve problem A.

Problem Reductions

The proof from 2 slides ago is a good example of reducing one problem into another.

- Problem A reduces to problem B if being able to solve problem B allows you to solve problem A.
- So, if problem A reduces to problem B, then if we know that problem A cannot be solved, then we know that B cannot be solved either.

Problem Reductions

The proof from 2 slides ago is a good example of reducing one problem into another.

- Problem A reduces to problem B if being able to solve problem B allows you to solve problem A.
- So, if problem A reduces to problem B, then if we know that problem A cannot be solved, then we know that B cannot be solved either.
- In the proof, we reduced the computability of $\Sigma(n)$

Problem Reductions

The proof from 2 slides ago is a good example of reducing one problem into another.

- Problem A reduces to problem B if being able to solve problem B allows you to solve problem A.
- So, if problem A reduces to problem B, then if we know that problem A cannot be solved, then we know that B cannot be solved either.
- In the proof, we reduced the computability of $\Sigma(n)$ to the computability of $S(n)$.

The General Busy Beaver Problem

In general, the busy beaver problem is to find the 'most productive' Turing machine with n states and m symbols.

The General Busy Beaver Problem

In general, the busy beaver problem is to find the 'most productive' Turing machine with n states and m symbols.

- The 'productivity' of a Turing machine can be defined in many ways:
  - The number of steps taken ('time')
  - The number of symbols written ('$f(n)$')
  - The number of cells moved away from the starting cell ('space')
  - …

The General Busy Beaver Problem

In general, the busy beaver problem is to find the 'most productive' Turing machine with n states and m symbols.

- The 'productivity' of a Turing machine can be defined in many ways:
  - The number of steps taken ('time')
  - The number of symbols written ('$f(n)$')
  - The number of cells moved away from the starting cell ('space')
  - …

Any of these kinds of functions can be found to be uncomputable.

The General Busy Beaver Problem

In general, the busy beaver problem is to find the 'most productive' Turing machine with n states and m symbols.

- The 'productivity' of a Turing machine can be defined in many ways:
  - The number of steps taken ('time')
  - The number of symbols written ('$f(n)$')
  - The number of cells moved away from the starting cell ('space')
  - …

Any of these kinds of functions can be found to be uncomputable.

- For any particular problem you can show this either by a direct proof, or by reducing it into another problem that you already established to be uncomputable.

Upper Bounds

Another interesting thing to note about the proof (from 4 slides ago) is that it uses the technique of using upper bounds: if we know an upper bound to the number of steps a machine can take before halting (such as $S(n)!$), then we can determine any machine with n states to be a halter or non-halter simply by running it: any machine that is still running after $S(n)$ steps is a non-halter.

So, once you have discarded all non-halters, you can simply run all halters to completion to figure out any kind of Busy Beaver function you want.

Connecting the Halting Problem to Busy Beaver

There seems to be an intimate connection between the Busy Beaver Problem and the Halting Problem:

Connecting the Halting Problem to Busy Beaver

There seems to be an intimate connection between the Busy Beaver Problem and the Halting Problem:

- Indeed, one might be inclined to say that the Halting Problem is immediately be solvable if $S(n)$ is computable

Connecting the Halting Problem to Busy Beaver

There seems to be an intimate connection between the Busy Beaver Problem and the Halting Problem:

- Indeed, one might be inclined to say that the Halting Problem is immediately be solvable if $S(n)$ is computable
  … but that would be a mistake!
  - Remember that $S(n)$ gives an upper-bound to the number of steps taken by all machines with n states … when started on an empty tape!
  - So, it does not give an upper-bound taken by all machines with n states given any kind of input tape, and the Halting function considers input tapes of any kind.

Connecting the Halting Problem to Busy Beaver

There seems to be an intimate connection between the Busy Beaver Problem and the Halting Problem:

- Indeed, one might be inclined to say that the Halting Problem is immediately be solvable if $S(n)$ is computable
  … but that would be a mistake!
  - Remember that $S(n)$ gives an upper-bound to the number of steps taken by all machines with n states … when started on an empty tape!
  - So, it does not give an upper-bound taken by all machines with n states given any kind of input tape, and the Halting function considers input tapes of any kind.

Still, the two problems are intimately related, but more complex.

If Halting Problem is solvable, then Busy Beaver Problem is.

One connection is obvious: If the Halting Problem is solvable, then (any) Busy Beaver Problem is solvable.

If Halting Problem is solvable, then Busy Beaver Problem is.

One connection is obvious: If the Halting Problem is solvable, then (any) Busy Beaver Problem is solvable.

- That is, $\Sigma(n)$ (or $S(n)$, or what have you) can be computed if we can solve the halting problem: Simply go through all the finitely many machines with n states, use the halting solution to discard any non-halters, and simply run all others to completion to get the desired answer.

If Halting Problem is solvable, then Busy Beaver Problem is.

One connection is obvious: If the Halting Problem is solvable, then (any) Busy Beaver Problem is solvable.

- That is, $\Sigma(n)$ (or $S(n)$, or what have you) can be computed if we can solve the halting problem: Simply go through all the finitely many machines with n states, use the halting solution to discard any non-halters, and simply run all others to completion to get the desired answer.

The other way is more difficult.

Empty Tape Halting Problem

The Empty Tape Halting Problem (ETHP) is to determine for any machine M whether or not it will halt when started on an empty tape.

Empty Tape Halting Problem

The Empty Tape Halting Problem (ETHP) is to determine for any machine M whether or not it will halt when started on an empty tape.

Obvious: if the general Halting Problem would be solvable, then ETHP would be solvable as well.

Empty Tape Halting Problem

The Empty Tape Halting Problem (ETHP) is to determine for any machine M whether or not it will halt when started on an empty tape.

Obvious: if the general Halting Problem would be solvable, then ETHP would be solvable as well.

Question: does the unsolvability of the Halting problem imply the unsolvability of ETHP?

Uncomputability of the Empty Tape Halting Problem

Yes! First we devise the Create-M_T routine that, given any M and T, constructs a machine M_T that, when given an empty tape, first puts T on the tape, and then runs M on that tape.

Uncomputability of the Empty Tape Halting Problem

Yes! First we devise the Create-M_T routine that, given any M and T, constructs a machine M_T that, when given an empty tape, first puts T on the tape, and then runs M on that tape.

Uncomputability of the Empty Tape Halting Problem

Yes! First we devise the Create-M_T routine that, given any M and T, constructs a machine M_T that, when given an empty tape, first puts T on the tape, and then runs M on that tape.

- Note: This routine is far from easy to write as a Turing-machine routine, but it is intuitively obvious that we should be able to do this, i.e. that such a routine does exist.

Uncomputability of the Empty Tape Halting Problem

Yes! First we devise the Create-M_T routine that, given any M and T, constructs a machine M_T that, when given an empty tape, first puts T on the tape, and then runs M on that tape.

- Note: This routine is far from easy to write as a Turing-machine routine, but it is intuitively obvious that we should be able to do this, i.e. that such a routine does exist.

Now assume the Empty Tape Halting Problem is solvable. Then the Halting Problem is solvable as well:

Empty Tape Halting Problem and $S(n)$

Theorem: The Empty Tape Halting Problem is solvable if and only if the $S(n)$ is computable.

Empty Tape Halting Problem and $S(n)$

Theorem: The Empty Tape Halting Problem is solvable if and only if the $S(n)$ is computable.

Proof:

Empty Tape Halting Problem and $S(n)$

Theorem: The Empty Tape Halting Problem is solvable if and only if the $S(n)$ is computable.

Proof:

- If ETHP is solvable, then we can figure out $S(n)$ by discarding all non-halters, running all others to completion, and determine max.
- If $S(n)$ is computable, then ETHP is solvable: simply start running any machine on an empty tape, and if it is still running after $S(n)$ steps, then it is a non-halter, otherwise it is a halter.

$S(n)$ and $\Sigma(n)$

We saw that $\Sigma(n)$ is computable if $S(n)$ is computable, as $S(n)$ is an upper-bound to the possible number of steps taken.

$S(n)$ and $\Sigma(n)$

We saw that $\Sigma(n)$ is computable if $S(n)$ is computable, as $S(n)$ is an upper-bound to the possible number of steps taken.
Does the other way around also hold?

$S(n)$ and $\Sigma(n)$

We saw that $\Sigma(n)$ is computable if $S(n)$ is computable, as $S(n)$ is an upper-bound to the possible number of steps taken.
Does the other way around also hold?

- Yes. For any machine M_1 that takes n steps before halting with $[k]$ when started on an empty tape, you can construct a machine M_2 that simulates M_1, but also prints out a 1 for every step that M_1 makes, and where the number of states of M_2 is a linear (and thus computable!) function of the number of states of M_1

$S(n)$ and $\Sigma(n)$

We saw that $\Sigma(n)$ is computable if $S(n)$ is computable, as $S(n)$ is an upper-bound to the possible number of steps taken.
Does the other way around also hold?

- Yes. For any machine M_1 that takes n steps before halting with $[k]$ when started on an empty tape, you can construct a machine M_2 that simulates M_1, but also prints out a 1 for every step that M_1 makes, and where the number of states of M_2 is a linear (and thus computable!) function of the number of states of M_1
  - E.g. for the quintuple formalization, you can show that if M_1 has n states, then such a M_2 can be constructed with 20n states.

$S(n)$ and $\Sigma(n)$

Suppose $S(M_1) = S(n)$ (i.e. M_1 is a machine with n states that makes the most steps for any machine with n states) we thus have $S(n) \leq \Sigma(f(n))$ for some computable $f(n)$.

$S(n)$ and $\Sigma(n)$

Suppose $S(M_1) = S(n)$ (i.e. M_1 is a machine with n states that makes the most steps for any machine with n states) we thus have $S(n) \leq \Sigma(f(n))$ for some computable $f(n)$.

So, if $\Sigma(n)$ is computable, then $S(n)$ becomes computable too: simply start any machine with n states on an empty tape, and any machine that still runs after having taken $\Sigma(f(n))$ steps must be a non-halter. For all halters, simply determine the maximum number of steps taken. In short, $\Sigma(f(n)$ provides an upper-bound for $S(n)$.

Summing it all up: Busy Beaver and Halting Problem

We have shown that:

- The Halting Problem is solvable if and only if the Empty Tape Haling Problem is solvable.
- The Empty Tape Halting problem is solvable if and only if $S(n)$ is computable.
- $S(n)$ is computable iff □(n) is computable.

Summing it all up: Busy Beaver and Halting Problem

We have shown that:

- The Halting Problem is solvable if and only if the Empty Tape Haling Problem is solvable.
- The Empty Tape Halting problem is solvable if and only if $S(n)$ is computable.
- $S(n)$ is computable iff □(n) is computable.

- So, these are all equivalent statements.

Summing it all up: Busy Beaver and Halting Problem

We have shown that:

- The Halting Problem is solvable if and only if the Empty Tape Haling Problem is solvable.
- The Empty Tape Halting problem is solvable if and only if $S(n)$ is computable.
- $S(n)$ is computable iff □(n) is computable.

- So, these are all equivalent statements.
- In particular, all these problems (functions) are unsolvable (uncomputable).

More on Upper Bounds

More on Upper Bounds

Suppose there is some computable function $f(n)$ that gives an upper bound to the maximum number of steps that a machine with n states can take, starting on an empty tape. I.e., suppose that for all n: $S(n) \leq f(n)$.

More on Upper Bounds

Suppose there is some computable function $f(n)$ that gives an upper bound to the maximum number of steps that a machine with n states can take, starting on an empty tape. I.e., suppose that for all n: $S(n) \leq f(n)$.

Then $S(n)$ would be computable too: all machines that still run after f(n) are non-halters, so $S(n)$ can be determined by examining all halters.

More on Upper Bounds

Suppose there is some computable function $f(n)$ that gives an upper bound to the maximum number of steps that a machine with n states can take, starting on an empty tape. I.e., suppose that for all n: $S(n) \leq f(n)$.

Then $S(n)$ would be computable too: all machines that still run after f(n) are non-halters, so $S(n)$ can be determined by examining all halters.

As $S(n)$ is not computable, we know that no such function exists: there is no computable upper bound for $S(n)$.

So what?

So this means that $S(n)$ is a function that 'grows faster' than any computable function.

So what?

So this means that $S(n)$ is a function that 'grows faster' than any computable function.

- And, you can easily come up with some computable functions that grow crazy fast.

So what?

So this means that $S(n)$ is a function that 'grows faster' than any computable function.

- And, you can easily come up with some computable functions that grow crazy fast.
- But $S(n)$ will grow even faster than that!

So what?

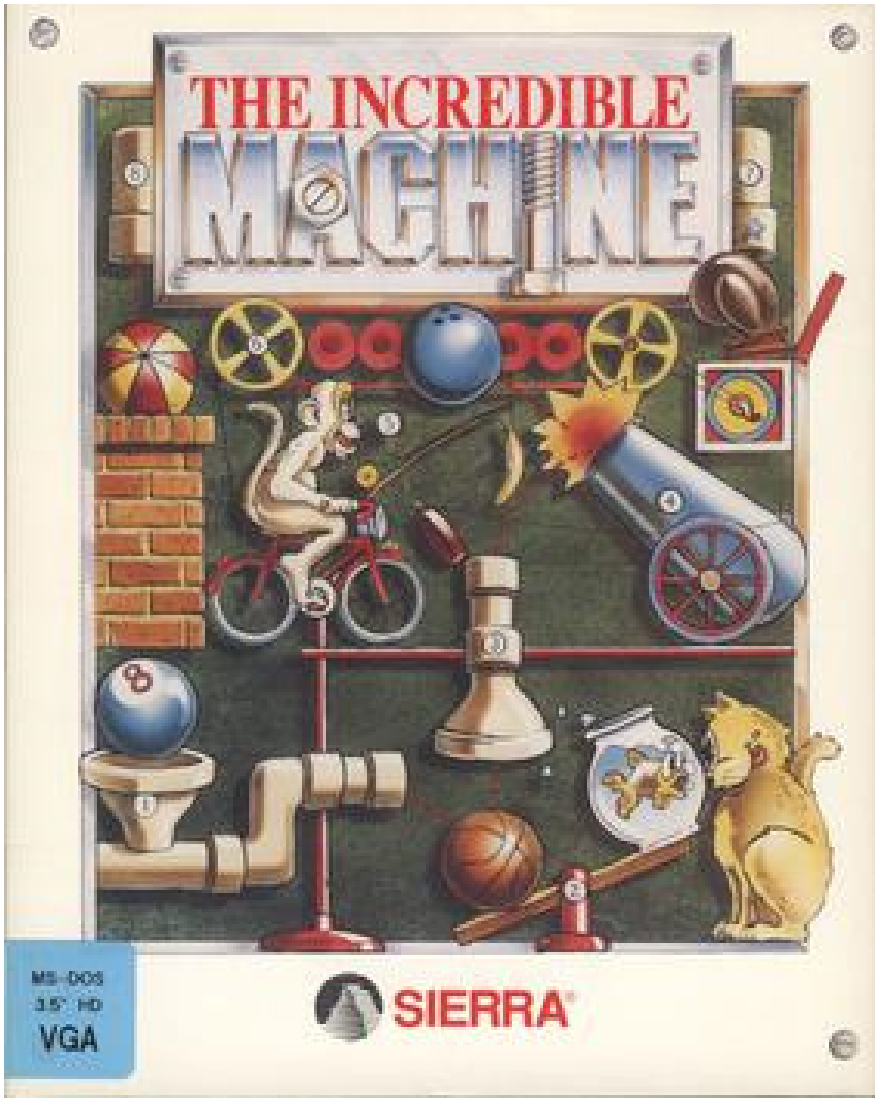
So this means that $S(n)$ is a function that 'grows faster' than any computable function.

- And, you can easily come up with some computable functions that grow crazy fast.
- But $S(n)$ will grow even faster than that!
- Moreover, since we found that $S(n) \leq \Sigma(f(n))$ for some computable $f(n)$, we know that $\Sigma(n)$ also grows faster than any computable function. (otherwise, we would once again have a computable upper bound for $S(n)$).

Established values for $\Sigma(n)$ (for quintuple TM's)

n	$\Sigma(n)$
1	1 (trivial)
2	4 (easy)
3	6 (Lin and Rado)
4	13 (Brady)
5	>= 4098 (Marxen et al.)
6	> $3.514 * 10^{18276}$ (Marxen et al.)
7	Huge
8	Insane

Why bother finding $\Sigma(n)$?



Why bother finding $\Sigma(n)$?

Many open math problems could be solved in a systematic way given the value of $S(n)$ for sufficiently large n.

Why bother finding $\Sigma(n)$?

Many open math problems could be solved in a systematic way given the value of $S(n)$ for sufficiently large n.

Goldbach's Conjecture: Write a program A that sequentially tests every even number ≥ 4 sequentially if it is the sum of two prime numbers or not. Simulate A on an n-state Turing machine. If it finds a counterexample, it halts and indicates that. But if the conjecture is true, then A will never halt.

Why bother finding $\Sigma(n)$?

Many open math problems could be solved in a systematic way given the value of $S(n)$ for sufficiently large n.

Goldbach's Conjecture: Write a program A that sequentially tests every even number ≥ 4 sequentially if it is the sum of two prime numbers or not. Simulate A on an n-state Turing machine. If it finds a counterexample, it halts and indicates that. But if the conjecture is true, then A will never halt.

So if we know $S(n)$ we can decide (in finite time) if it will ever halt by simply running the machine that many steps. And if, after $S(n)$ steps, the machine does not halt, we know that it never will and so there are no counterexamples to Goldbach.

Problems in Determining $\Sigma(n)$

First, search space is fairly big:

- Explicit halting state: $|M(n)| = (4n + 4)^{2n}$
- Implicit halting state: $|M(n)| = (4n + 1)^{2n}$

Problems in Determining $\Sigma(n)$

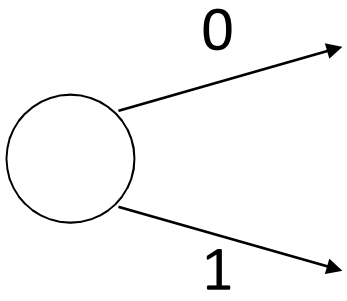
First, search space is fairly big:

- Explicit halting state: $|M(n)| = (4n + 4)^{2n}$
- Implicit halting state: $|M(n)| = (4n + 1)^{2n}$

But more importantly, the behavior of even small machines is strange and hard to grasp, and it is hard to tell whether they halt or not.

Size of Search Space

For every state:



For every transition:

Quintuple:

0 or 1 and L or R

⇒ 2*2 = 4 possibilities

Quadruple:

0 or 1 or L or R

⇒ 4 possibilities

In all cases: 2n state-symbol pairs

Explicit: n + 1 possible new states with full transition

Implicit: n possible new states with full transition + empty transition to halting state

n	$(4n + 4)^{2n}$	$(4n + 1)^{2n}$
1	64	25
2	20,736	6,561
3	16,777,216	4,826,809
4	$\sim 2.56 * 10^{10}$	$> 10^{9}$
5	$> 10^{13}$	$> 10^{13}$
6	$> 10^{17}$	$> 10^{16}$

Strange Behavior

For $n = 5$ (quintuple), some machines only come to a halt after over 47 million steps (this is the suspected shifting champion … but we haven't been able to prove that yet)

- For $n = 6$, some machines halt after more than 10^{36534} steps.
- In 1983, Brady conjectured that it's impossible to classify all machines with n = 5
- In 2024, the case n = 5 was settled by graduate students!
- Still, it's unlikely we'll ever settle the n = 6 case.
- And it's a virtual certainty we'll never settle the n = 7 case!

Attacking the Busy Beaver Function

Still, people are trying to determine Busy Beaver values.

Attacking the Busy Beaver Function

Still, people are trying to determine Busy Beaver values.

This is done by:

– Reducing the search space

– Writing custom-made computer programs to simulate the behavior of Turing-machines and detecting knowable forms of non-halting behavior.

Reducing the Search Space

The search space can be reduced in a number of ways:

- Perform some initial analysis to rule out certain corners of the search space
- Only consider one machine of a set of machines that can be shown to have equal productivity
- Use Tree normalization method to create only 'relevant' machines.

Halters, Non-halters, and Holdouts

As the halting problem cannot be solved in general, any practical algorithm that tries to figure out halting behavior is such that for any machine M:

- A eventually declares that M halts (M is a halter)
- A eventually declares that M does not halt (M is a non- halter)
- A eventually declares that it doesn't know whether M halts or not (M is a *holdout*)

Taking care of the holdouts

To take care of the holdouts, do the following:

- Simulate a holdout for some number of steps
- Observe the behavior, and see if there is some pattern present which one can use to prove that the machine will halt or not
- Try and generalize the pattern of behavior, and incorporate it into the original algorithm to obtain a new and improved algorithm
- If there are still holdouts left with the new algorithm, repeat this routine

History of $\Sigma(3)$

- Rado was at first very pessimistic:
  - "… there is no evidence that any known approach will yield the answer, even if we avail ourselves of high-speed computers and elaborate programs." (Rado, 1963)

History of $\Sigma(3)$

- Rado was at first very pessimistic:
  - "… there is no evidence that any known approach will yield the answer, even if we avail ourselves of high-speed computers and elaborate programs." (Rado, 1963)
- Indeed, it turns out he was too pessimistic:
  - "The solution of this quite special problem was attempted by several competent mathematicians and programmers, by means of increasingly elaborate computer programs." (Lin and Rado, 1965)

History of $\Sigma(3)$

- Rado was at first very pessimistic:
  - "… there is no evidence that any known approach will yield the answer, even if we avail ourselves of high-speed computers and elaborate programs." (Rado, 1963)
- Indeed, it turns out he was too pessimistic:
  - "The solution of this quite special problem was attempted by several competent mathematicians and programmers, by means of increasingly elaborate computer programs." (Lin and Rado, 1965)
- Lin and Rado reduced the 56871 holdouts from 1963 to 40 holdouts in 1965, and analyzed those last 40 by hand to determine $\Sigma(3)$.

History of $\Sigma(4)$

"As regards $\Sigma(4)$, … the situation seems to be entirely hopeless at present."
(Rado, 1963)

History of $\Sigma(4)$

"As regards $\Sigma(4)$, … the situation seems to be entirely hopeless at present."
(Rado, 1963)

> "More than 18 other programs were written for various housekeeping purposes, simulating and displaying machine behavior, exploring other reduction and filtering possibilities, etc. In all, at least 53 files were created and maintained for the project. Keeping track of what resembled a large scientific experiment became a major task in itself."
> (Brady, 1983)

History of $\Sigma(4)$

"As regards $\Sigma(4)$, … the situation seems to be entirely hopeless at present."
(Rado, 1963)

> "More than 18 other programs were written for various housekeeping purposes, simulating and displaying machine behavior, exploring other reduction and filtering possibilities, etc. In all, at least 53 files were created and maintained for the project. Keeping track of what resembled a large scientific experiment became a major task in itself."
> (Brady, 1983)

- Brady obtained 0 holdouts for n = 3, and 218 holdouts for n = 4. These "were examined by means of voluminous printouts of their histories along with some program extracted features. It was determined to the author's satisfaction that none of these machines will ever stop."