Computability

Klaus Sutner
Carnegie Mellon University
www.cs.cmu.edu/~sutner

Battleplan

- Das Entscheidungsproblem
- A Brief History of Computation
- Turing Machines
- Computability and Decidability
- Halting and Busy Beaver
- Universal Computation

19th Century

The main accomplishments in 19th century mathematics in a nutshell:

- complex variables
- abstract groups
- set theory

While such "top ten" lists are often contentious, this one is fairly uncontroversial and was generally accepted around 1900.

Note that set theory is a new-comer whose actual value and relevance was still somewhat in doubt at the time. The other areas were well-established and fairly mature.

Hilbert's List

After the International Congress of Mathematicians in Paris in 1900 David Hilbert published a list of the 23 main open problems in math.

Hilbert's manifesto was enormously influential throughout the 20th century.

Some of Hilbert's Problems

1. Prove the Continuum Hypothesis. Well-Order the reals.

2. Prove that the axioms of arithmetic are consistent.

   . . .

8. Prove the Riemann Hypothesis.

   . . .

10. Determination of the solvability of a Diophantine equation
    Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: to devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers.

    . . .

Entscheidungsproblem

The Entscheidungsproblem is solved when one knows a procedure by which one can decide in a finite number of operations whether a given logical expression is generally valid or is satisfiable. The solution of the Entscheidungsproblem is of fundamental importance for the theory of all fields, the theorems of which are at all capable of logical development from finitely many axioms.

D. Hilbert, W. Ackermann
Grundzüge der theoretischen Logik, 1928

Decision Procedures

In modern terminology the Entscheidungsproblem comes down to finding a *decision procedure* for statements of mathematics.

Note that Hilbert and Ackermann hedge a bit: the decision procedure needs to work only for areas where the fundamental assumptions can be specified in a finite list.

J. Herbrand pointed out that "In a sense, [the Entscheidungsproblem] is the most general problem of mathematics."

In a way, this can be traced back to Leibniz's *ars magna*:

- *ars inveniendi* generate all true scienctific statements.
- *ars iudicandi* decide whether a given scienctific statement is true.

20th Century

To produce a similar list of crucial accomplishments for the 20th century is rather hard.

Here is an extremely biased one which would surely draw a lot of flak from many:

- logic: formal systems, incompleteness, independence
- computability: (un)decidability, complexity, algorithms
- discrete mathematics: combinatorics, number theory, graph theory

Not coincidentally, all these areas are highly relevant to computer science.

Smale's List

No one wanted to step into Hilbert's shoes in 2000, but some attempts were made to come up with a similar list of crucial problems for the 21st century.

1. Prove the Riemann Hypothesis.
2. Prove the Poincaré Conjecture. 2006: Seems obsolete!
3. Is $\mathbb{P} = \mathbb{NP}$?
4. Bound the number of integer roots of a Diophantine polynomial.

. . .

18. What are the limits of intelligence, both artificial and human?

The interesting point here is that some of these problems have their origin in computer science rather than pure mathematics.

Invasion of the Mind Snatchers

Steve Smale is a classical mathematician (proved the Poincaré Conjecture for dimensions $n \geq 5$), but has become very interested in computational issues in the last decade.

It is truly remarkable that so many of his fundamental problems have a distinct computer science flavor.

▶ Computer science has effectively invaded all other sciences.

That's great from a CS perspective, but it means additional work: a CSist is supposed to be able to communicate effectively with scientists from other fields.

Problems and Algorithms

Informally, an *algorithm* is a mechanical procedure, which solves a *(computational) problem* in finitely many, well-determined steps.

Obviously, we cannot deal with problems such as

> "Does god exist?"

though a surprising number of people have rather strong attitudes about this issues. Also, we don't want to deal with

> "What is next week's weather?"

Note that the latter is far from meaningless, but it leads to a part of mathematics that we do not want to get involved with here.

Computability

There are several issues we need to tackle:

- Give a precise definition of an algorithm, or, more generally, of the notion of computability.
- Provide evidence that our definition adequately reflects the intuitive notion of computability.
- Study the basic properties of computability and prove the main results in the area.
- Develop tools to show that some problems are computationally solvable or fail to be so solvable.
- Develop a theory of tractable problems and practical computation.

Defining Computability

There are two main approaches to defining computability:

- Programs
- Machines

Programs are probably more familiar to the CS major, and therefore the more tempting approach. But, there is the problem of semantics: what exactly is the meaning of a program? This issue is rather difficult to tackle and would obscure the question we are trying to answer: what does it mean to be computable.

If you think otherwise, try to give a precise description of the semantics of a C++ program, complete with classes, multiple inheritance and templates.

A definition in terms of machines requires a bit of work but turns out to be easier and perhaps even more compelling in the end.

A Brief History of Computability

Since computers are now ubiquitous, the problem of defining computability is often considered trivial. However, things are not so simple, and in the 1930's even the giants in the field had to struggle to come to grips with this notion.

Historically, methods for computation go back to the very beginning of mathematics. E.g., Euclid's algorithm for the computation of the greatest common divisor is a perfect example for a number theoretic algorithm. Here are some events closer to our discussion.

- R. Dedekind, 1888
  Gives a recursive definition of addition, multiplication and exponentiation on the naturals.
- G. Peano, 1889
  As a companion to his famous axiom schema V introduces the principle of (primitive) recursive definition.
- K. Gödel, 1931
  Uses primitive recursive function to arithmetize logic (but calls the "recursive" at the time).

More History

- K. Gödel, 1934
  Following a suggestion by J. Herbrand introduces general recursive functions using systems of equations.
- A. Church, 1930
  Introduces λ-definable functions based on his λ-calculus (the theory of functional composition).
- Kleene, 1934
  Shows that many number-theoretic functions are λ-definable.
- A. Church, April 19, 1935
  Announces "Church's Thesis": effectivel calculable is the same as recursive. Published in 1936; claim that the Entscheidungsproblem is unsolvable.
- A. Turing, April 15, 1936
  Introduces his theory of computability based on Turing machines.
- Kleene, 1936
  μ-recursive functions based on unbounded search.

Gödel's Position

Gödel was dissatisfied with Church's early attempts to capture computability in terms of λ-definability.

After Church switched to recursive functions there was still not much positive reaction from Gödel; he was probably clearly aware of the persistent weakness in the attack on the Entscheidungsproblem.

But to Turing's work Gödel responded most enthusiastically.

> This concept, . . . is equivalent to the concept of a "computable function of integers" . . . The most satisfactory way, in my opinion, is that of reducing the concept of finite procedure to that of a machine with a finite number of parts, as has been done by the British mathematician Turing.

Gödel always gave full credit to Turing, never to Church or himself.

Alan Turing

Turing's Machines

Basic Idea: Observe the computor. We all agree that mathematicians compute (among other things such as drinking coffee or proving theorems).

So we could try to define an abstract machine that can perform any calculation whatsoever that could be performed in principle by a mathematician, and only those. To this end we need to formalize what a computor is doing.

Mathematicians can always write down their calculations on paper. This amounts simply to bookkeeping of all intermediate results.

We need to formalize what can be written down and the rules that control the process of writing things down.

Also note that Turing had terrible handwriting, he was always interested in typewriters. His machines are, in a sense, just glorified typewriters.

The Abstraction

- Mathematicians use two-dimensional notation, but could all be flattened out into one-dimensional notation.
- Only finitely many symbols are allowed.
- Can think of having a strip of paper subdivided into cells, each cell containing exactly one symbol (possibly the blank symbol).

```
12345 * 6789
    74070
     86415
      98760
      111105
    83810205
```

is flattened out into the linear string

```
12345 * 6789 74070 86415 98760 111105 = 83810205
```

Digression

Flattening out notation is standard when dealing with keyboard input.

Instead of

$$\int_0^1 \frac{1}{2+x^2}dx = \frac{\arctan(\frac{1}{\sqrt{2}})}{\sqrt{2}}$$

one has to deal with something like

```
Integrate[1/(2 + x^2), {x,0,1}] ==> ArcTan[1/Sqrt[2]]/Sqrt[2]
```

The Rules

- The input is written on an otherwise empty tape.
- The computor's mind has finitely many possible states and is always in one particular of these states.
- At each step, the computor focuses on one particular cell, the current cell.
- The computor inspects the current cell and takes action depending on his own state of mind and the symbol in the cell.
  - She may overwrite the symbol in the current cell.
  - She may shift attention to the cell on the left or right.
  - She may switch into a new state of mind.
- Initially, she is in state "start" and looks at the first empty cell to the left of the input.
- The computor keeps doing this until a state of mind "finished" is reached.
- What is written on the tape at this point constitutes the result of the computation.

Justification

These assumptions are quite robust.

- It would take the computor infinitely long to learn the meaning of infinitely many symbols.
- Likewise, the mind can assume only finitely many configurations (quantum physics, digital physics).
- A human mind cannot pay attention to infinitely many symbols. But finitely many symbols is the same as just one (blocking).
- Likewise a step cannot depend on infinitely many symbols. But using finitely many symbols is the same as just using one.
- All but finitely many cells will always contain a special blank symbol. Othewise someone would already have had to spend an infinite amount of time writing down the initial tape contents.

Finite Tapes

It should be noted that the last condition is a bit harsh.

It is certainly justified to insist on a very simple initial tape inscription, but all blanks on either side of the input pushes things a bit.

For example, an inscription of the form

$$\ldots 0000100001000001 x_1 x_2 \ldots x_{n-1} x_n 1000010000100001 \ldots$$

is surely harmless.

For Turing machines, this makes no difference whatsoever, but there are other modells of computation such as cellular automata where the more general starting configurations are of great interest.

The Crucial Constraints, Again

In a nutshell, here are the important constraints Turing imposes on his devices.

- The number of symbols is finite.
- The number of states is finite.
- The number of squares observed at any time is finite.
- Only one of the observed squares can be changed.
- Attention can only move a fixed, bounded distance away from an observed square.
- Every action depends only a the observed squares and the current state.
- These actions are uniquely determined.

We can now turn this into a formal definition in a relatively straightforward way.

Formalization

It is customary to call the strip of paper a *tape* and the placement of symbols in the cells a *tape inscription*. Thus, a tape inscription is a map $T : \mathbb{Z} \to \Sigma$. Think of a read/write head as being positioned at one tape cell.

- *alphabet* Σ: finite set of allowed symbols, special blank symbol $\underline{b} \in \Sigma$
- *state set* Q: finite set of possible mind configurations
- $\delta : Q \times \Sigma \to Q \times \Sigma \times \{-1, 0, 1\}$: *transition function*
- a special *initial state* $q_0 \in Q$
- a special *halting state* $q_H \in Q$.

Definition 1. *A* Turing machine *is a structure* $M = \langle\, Q, \Sigma, \delta, q_0, q_H \,\rangle$.

Exercise 1. *Explain how this definition captures Turing's constraints.*

Example: Successor Function

Tape alphabet $\Sigma = \{\underline{b}, 1\}$

States $Q = \{0, 1, 2, 3\}$

Initial state $q_0 = 0$

Final state $q_H = 3$.

The transition function δ is given be the following table:

p	σ	$\delta(p, \sigma)$		
0	$\underline{b}$	1	$\underline{b}$	$+1$
1	$\underline{b}$	2	1	0
1	1	1	1	1
2	$\underline{b}$	3	$\underline{b}$	0
2	1	2	1	-1

Don't need any other transitions.

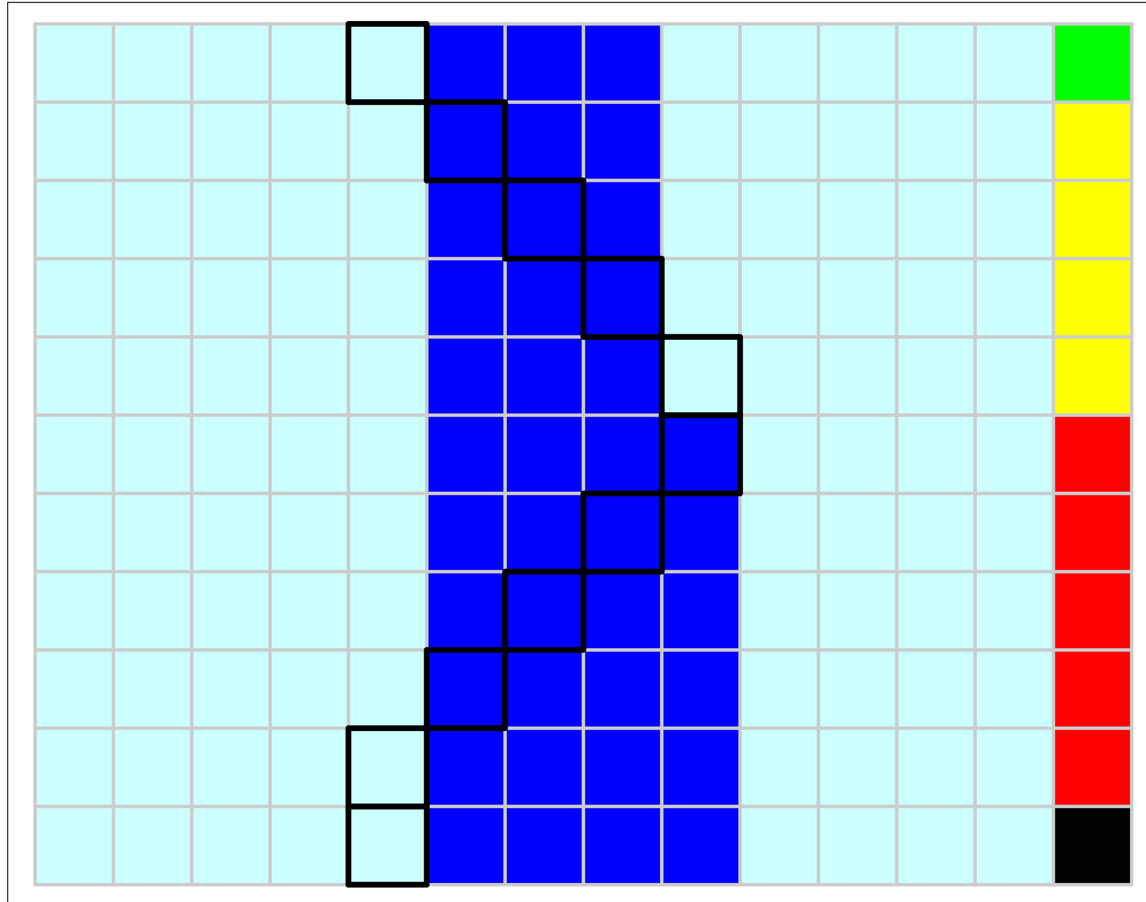
Simulation

Needless to say, it is not hard to write a program that simulates a Turing machine. Hence, we can perform some computational experiments on these machines. Alas, it's not alway easy to extract useful information from these simulations.

In the following, we use visualization to get some insight in the behavior of small Turing machines. In the pictures,

- the tape is represented by a row of colored squares.
- Time flows from top to bottom (usually; sometimes from left to right).
- The position of the tape head is indicated by a black frame around a square.
- The rightmost column indicates the state of the machine.

Sample Run



Addition

We are using unary notation, n is represented by

$$\underbrace{111\ldots11}_{n+1}$$

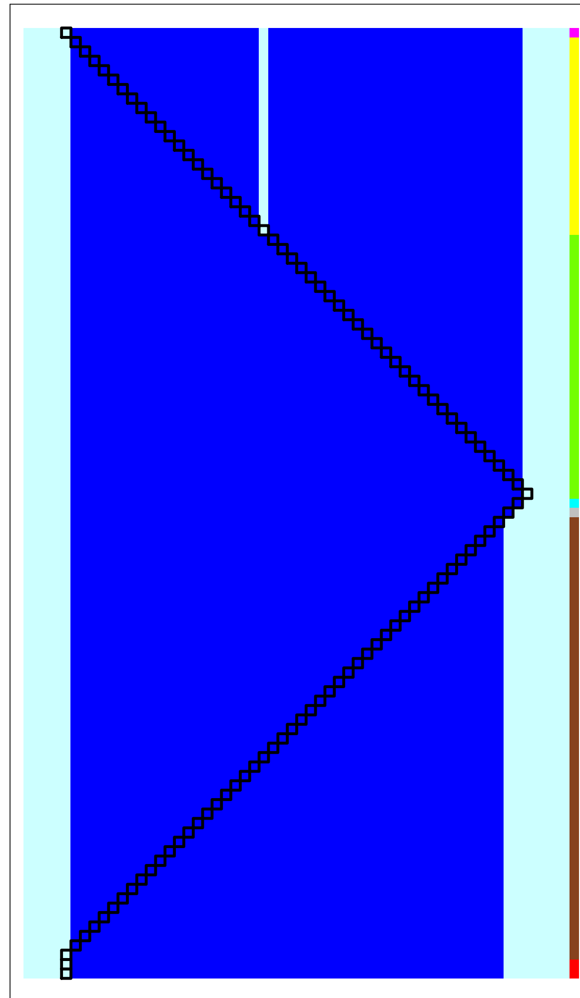
Hence we have to erase two 1's at the end:

$$\begin{array}{cc|ccc}
0 & \underline{b} & 1 & \underline{b} & 1 \\
1 & \underline{b} & 2 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
2 & \underline{b} & 3 & \underline{b} & -1 \\
2 & 1 & 2 & 1 & 1 \\
3 & 1 & 4 & \underline{b} & -1 \\
4 & 1 & 5 & \underline{b} & -1 \\
5 & \underline{b} & 6 & \underline{b} & 0 \\
5 & 1 & 5 & 1 & -1
\end{array}$$

0 is the initial state, 6 is the final state

Sample Run

A Longer Run



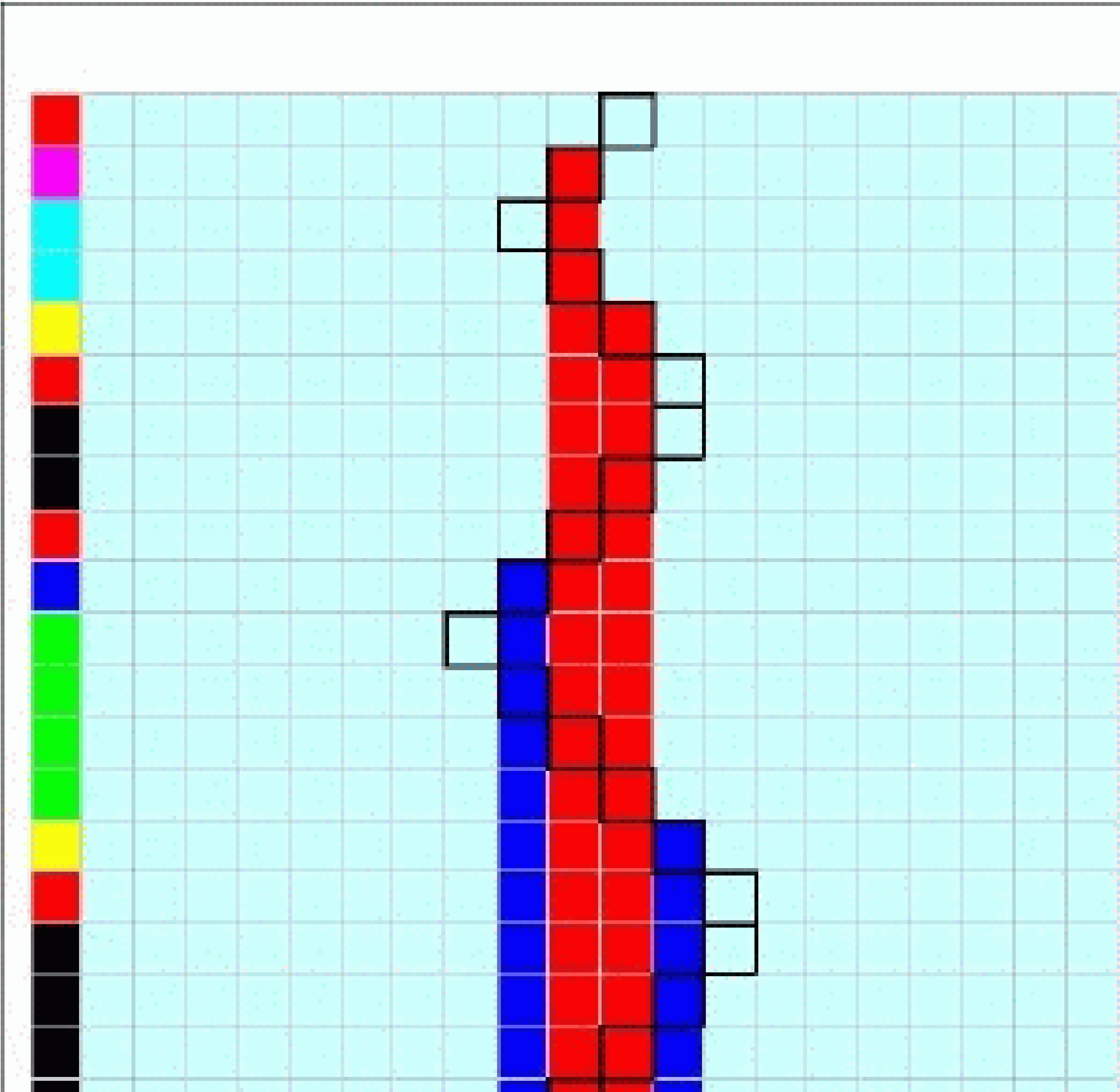
Correctness

Note that for very simple tasks such as addition in unary one can read off a correctness proof from these pictures.

Exercise 2. *What would a complete correctness proof for the Turing machine that performs unary addition look like? What is difficult about the proof?*

Exercise 3. *Construct a Turing machine that performs addition when the input is given in binary. What should be pictures look like in this case? How hard is a correctness proof?*

Palindromes



Time flows from left to right and the states are in the top row. The picture represents a successful computation.

Exercise 4. *Figure out how this machine works.*

Formalizing Turing Computation

Time to give a more precise definition of how a Turing machine can be used to “solve” a computational problem.

There are two issues that need to be clarified:

- What exactly do we mean by a computational problem?
- How does a Turing machine compute; in particular how does it solve a problem?

Let’s start with various kinds of computational problems.

Decision Problems

Definition 2. *A decision problem consists of a set of instances and a subset of Yes-instances.*

Problem: **Primality**
Instance: A natural number x.
Question: Is x a prime number.

Here the set of all instances is $\mathbb{N}$ and the set of Yes-instances is $\{ p \in \mathbb{N} \mid p \text{ prime} \}$.

The answer (solution) to any decision problem is just one bit (true or false).

Function Problems

Definition 3. *A function problem consists of a set of instances and, for each instance I, a solution* sol(I)*.*

Problem: **Next Prime**
Instance: A natural number x.
Solution: The least prime $p > x$.

Instances are again $\mathbb{N}$ and the solution for $x \in \mathbb{N}$ is sol$(x) = p$ where p is the appropriate prime (uniquely determined).

We insist that there always is a solution (otherwise sol would be a partial function).

Search Problems

Definition 4. *A search problem consists of a set of instances and, for each instance I, a set of solutions* sol(I)*.*

So here "the" solution is not required to by unique. And, we allow for sol(I) to be empty. Very convenient in practice.

Problem: **Factor**
Instance: A natural number x.
Solution: A natural number z, $1 < z < x$, dividing x.

Note that the set of solutions is empty if x is prime.

Instances

For abstract computability, it is irrelevant exactly how the input is given as long as it is given in explicit, unobfuscated form.

For example, natural numbers may be given in unary, binary or decimal. Or in Roman numerals.

What is not allowed are tricks like the following:

> Encode natural number n as $1x$, where x is the binary expansion of n, whenever n is prime; and as $0x$ when n is compound.

With this "input convention" primality testing would be trivial, but the coding procedure requires a lot of computational effort.

A similar trick could be used to trivialize any decision problem.

Also, descriptions along the lines of "consider x, the least number such that . . . " are not permissible as input.

Canonical Representations

For efficiency considerations (rather than just pure computability) we may need to specify precisely the data structure representing the input.

However, in all practical cases there is a canonical, natural choice. Essentially, all that is needed is:

- Natural numbers are given in binary.
- Lists of objects are defined by some natural coding function (more about this later).

There are several natural choices for coding lists, but they are be equivalent in the sense that we can convert from one format to the other without much computational effort.

Solving a Problem

To solve a *decision problem*, an algorithm has to accept each instance of the problem as input, and return "Yes" or "No" depending on whether the instance is a Yes-instance.

To solve a *function problem*, an algorithm has to accept each instance I of the problem as input, and return the unique sol(I).

To solve a *search problem*, an algorithm has to accept each instance I of the problem as input, and return either an element of sol(I) or "No" if sol(I) is empty (the instance has no solutions).

So decision problems are easier to deal with since there is a one bit answer as opposed to a potentially complicated data structure that is required to describe a solution in a function or search problem.

Note that an algorithm must terminate on all legitimate instances.

Problems and Turing Machines

Suppose we want to solve a decision problem using a Turing machine.

Intuitively, we do the following:

- Write the instance on the otherwise empty tape, put the machine into a special start state and place the head to the left of the input.
- Go through a sequence of transitions that change the state, head position and tape content according to the definition of the machine.
- In the end, erase the tape completely except for a single 0 ("corresponding to "false") or 1 (corresponding to "true") and halt.

For decision and search problems the answer is likewise left on the tape.

Input/Output Conventions

There are two compenents of this process:

- The input and output convention used to express an instance on the tape, and to decode the answer.
- The purely internal computation of the machine.

Often the input/output part is ignored since the coding is "natural". Still, it is worth remembering that there are usually several perfectly reasonable alternatives.

Also, input/output is really independent of the machine, it is just a mechanism to connect the machine to a computational problem.

Configurations

To describe a snapshot during a computation of a Turing machine we need

- the current tape inscription
- the current head position
- the current state

Definition 5. *A configuration or instantaneous description (ID) is a triple* $\langle T, i, p \rangle$ *where* $T : \mathbb{Z} \to \Sigma$ *is a tape inscription,* i *an integer and* p *a state.*

The machine computes by moving from one configuration to the next according to the transition function. This is best expressed by a relation on configurations.

One-Step Relation

More precisely, the machine moves from configuration $\langle\, T, i, p \,\rangle$ to configuration $\langle\, S, j, q \,\rangle$ in one step as follows:

Letting

$$\delta(p, T(i)) = (q, b, d)$$

we have

$$S(c) = \begin{cases} b & \text{if } c = i, \\ T(c) & \text{otherwise.} \end{cases}$$

$$j = i + d$$

That's it. Note that it is very easy to check if $\langle\, T, i, p \,\rangle$ leads to $\langle\, S, j, q \,\rangle$ in one step.

The Step Relation

It is now a straightforward discrete math exercise to define exactly how a Turing machine computes, i.e., moves from one configuration to the next.

- $\langle\, T, i, p \,\rangle \vdash^1_M \langle\, S, j, q \,\rangle$: one step
- $\langle\, T, i, p \,\rangle \vdash^t_M \langle\, S, j, q \,\rangle$: exactly t steps
- $\langle\, T, i, p \,\rangle \vdash^*_M \langle\, S, j, q \,\rangle$: any finite number of steps

For example, the t-step relation is defined like so:

$\langle\, T, i, p \,\rangle \vdash^0_M \langle\, S, j, q \,\rangle$ if $\langle\, T, i, p \,\rangle = \langle\, S, j, q \,\rangle$

$\langle\, T, i, p \,\rangle \vdash^{t+1}_M \langle\, S, j, q \,\rangle$ if
$\langle\, T, i, p \,\rangle \vdash^t_M \langle\, S', j', q' \,\rangle$ and $\langle\, S', j', q' \,\rangle \vdash^1_M \langle\, S, j, q \,\rangle$.

The turnstile is a remnant of Gottlob Frege's work, more later.

Input

As already mentioned, any input can be thought of as a finite sequence $x_1, x_2, \ldots, x_n$ of symbols in Σ.

The initial tape inscription is then

$$T_x = \begin{cases} x_i & \text{if } 1 \leq i \leq n, \\ \underline{b} & \text{otherwise.} \end{cases}$$

where $\underline{b}$ denotes the special blank symbol in Σ.

The initial configuration has the form

$$C_x = \langle\, T_x, 0, q_0 \,\rangle$$

Output

A *final configuration* is required to be of the form

$$C_y^H = \langle\, T_y, 0, q_H \,\rangle$$

Thus we require our machines to erase the tape (except for the output) and return the tape head to the initial position before halting; nothing important changes if we drop this condition.

But note that this convention makes it very easy to chain together computations.

If $C_x \vdash_M^* C_y^H$ then $y = y_1, y_2, \ldots, y_m$ is the output of the computation of machine M on input x.

Finitary Tape Inscriptions

Note that initially only finitely many tape cells are non-blank. Clearly the tape stays that way during the whole computation.

More precisely, $\langle T, i, p \rangle \vdash_M^t \langle S, j, q \rangle$ implies that

- $|i - j| \leq t$
- S differs from T in at most t places

So we don't really need all of T, just a finite piece containing all the non-blank entries.

Exercise 5. *Give an alternative definition of configuration and the next step relation where the tape inscription is a finite word over the tape alphabet.*

Prove that the two notions are equivalent.

Variants: Multiple Tapes

It is sometimes more convenient to think of a Turing machine as having multiple tapes. Popular are

- a read-only input tape
- a write-only output tape
- several read/write work tapes, each with a separate head

It is not hard to see that any multi-tape machine can be simulated (in some strict technical sense) by a single tape machine (albeit with slow-down).

Exercise 6. *Determine how long it takes to recognize palindromes on two tapes versus one tape.*

Exercise 7. *Determine the general slow-down caused by switching from two tapes to one.*

The Important Ideas

We can specify a model of computation by defining

- a collection of possible configurations (snapshots),
- how one configuration leads to the next,
- an input and output format.

The details vary, but it's always the same pattern.

In a sense, details such as input/output conventions don't matter, they are all interchangeable.

Exercise 8. *Show that it does not matter whether the tape head is required to return to the standard left position at the end of a computation.*

Turing Computability

We still need to explain how a Turing machine solves a particular computational problem. The easiest way is to tackle function problems first.

Definition 6. *A function $f : \Sigma^* \longrightarrow \Sigma^*$ is* Turing computable *if there is some Turing machine M such that for all x:*

$$C_x \vdash_M^* C_{f(x)}^H.$$

Correspondingly, the machine solves a function problem if it computes sol(I)*.*

Again, we really should explain the input/output conventions, but we simply assume a "natural encoding".

E.g., to deal with numerical functions we can assume $1 \in \Sigma$ and use unary encoding (where $n \mapsto 1^{n+1}$ so that 0 is represented by 1). Or we assume $0, 1 \in \Sigma$ and use binary encoding.

Fudging It

Usually we simply say that a function

$$f : \mathbb{N}^n \longrightarrow \mathbb{N}$$

is Turing computable. Likewise for functions defined on lists, tress, graphs, matrices, whatever.

The justification for this sloppiness is that the coding details do not affect computability.

Warning

When one is interested in the actual running time of algorithms there can, of course, be a huge difference depending on coding. E.g., unary input for multiplication makes little sense.

For the time being we ignore this complication.

The Real World

So how does this compare to "real" computability?

Claim 1. *Turing computable is exactly the same as intuitively computable.*

Proof. Clearly one could write a Turing machine simulator in any programming language.

For the opposite direction, think of a PC: all the bits in memory and CPU can be flattened out into a tape of $0/1$, plus some delimiters and markers. The operation of the CPU can be simulated (albeit very slowly) by a Turing machine which manipulates this tape inscription. This would be very tedious to do in practice, but it's not difficult conceptually.

$\square$

Exercise 9. *Convince yourself that the last sketch could be turned into a real proof; there are no hidden traps and pitfalls.*

Discrete Dynamical System

One way to look at Turing machines is in terms of DDSs:

- The configurations are the IDs of the TM.
- One step in the system corresponds to a single step in a computation.
- An orbit corresponds to a full computation.

Note that the DDS perspective opens a few cans of worms: in ordinary computability one is only interested in the special orbits starting at C_x, a rather sparse set from the point of view of the whole uncountable space.

But in a DDS one should study all orbits, even those that cannot be part of any "honest" computation.

Robustness

There are many ways we could modify our definitions.

- one-way infinite tapes
- multiple tapes
- multiple heads
- different input/output conventions

Claim 2. *Regardless of all these options, we always get exactly the same class of computable functions.*

Note that without robustness our model would be essentially useless.

Relations

We have taken care of function and search problems, but for decision problems we need one more simple idea: we translate sets (relations) to Boolean valued functions.

First, we can safely assume that the given relation is a subset $\Sigma^\star$ where Σ is some suitable alphabet. For example, the less-than relation on $\mathbb{N}$ can be expressed as $R = \{\, \mathsf{bin}(x)\#\mathsf{bin}(y) \mid x < y \,\} \subseteq \{0, 1, \#\}^\star$ where $\mathsf{bin}(x)$ denotes the binary expansion of x.

Definition 7. *The* characteristic function $\mathsf{char}_R : \Sigma^\star \to \{0, 1\}$ *of a relation* $R \subseteq \Sigma^\star$ *is defined by:*

$$\mathsf{char}_R(x) = \begin{cases} 1 & x \in R \\ 0 & \textit{otherwise.} \end{cases}$$

Bit-Vectors

Characteristic functions are just the mathematical counterpart of bit-vectors.

In the finite case, a characteristic function is nothing else but a bit-vector:
`bv[i] = 1` indicates that element i is in the set, and `bv[i] = 0` means that it's out.

Our definition also covers the infinite case. For example, the characteristic function $\mathbb{N} \longrightarrow \mathbf{2}$ of the set of primes looks like this

$$0011010100010100010100\ldots$$

Turing Decidability

Now we can tackle decision problems.

Definition 8. *A set $R \subseteq \Sigma^\star$ is* Turing decidable *if the characteristic function* char_R *is Turing computable.*

Likewise, one says that a decision problem is solvable if the set of Yes-instances is decidable.

This simply means that there is some Turing machine that, given an instance of the problem, computes for a finite amount of time, and then returns the correct answer Yes or No to the question: is the given instance a Yes-instance?

Exercise 10. *Argue that Turing decidable corresponds exactly to the intuitive notion of decidability.*

Closure Properties

Lemma 1. *The decidable sets are closed under intersection, union and complement.*

Proof.

Consider two decidable sets $A, B \subseteq \mathbb{N}$. We have two Turing machines M_A and M_B that decide membership.

Idea: Run both M_A and M_B on input x, returning output bits b_A and b_B.

For intersection return $b_A \wedge b_B$, for union return $b_A \vee b_B$. For the complement of A return $\neg b_A$.

$\square$

Of course, the hard part is to show that this can all be done on a Turing machine.

Entscheidungsproblem, Again

So how about Hilbert's Entscheidungsproblem?

Given an arbitrary decision problem $A \subseteq \mathbb{N}$, is A always (Turing) decidable?

Thus, given any set $A \subseteq \mathbb{N}$ we would like to be able to construct a Turing machine M_A that, on input any number $x \in \mathbb{N}$, halts with output 1 if $x \in A$ and halts on output 0 otherwise.

Unfortunately, the answer is no.

Here is a cardinality argument due to Cantor that shows that there are lots of undecidable problems.

Counting

Theorem 1. *There are undecidable decision problems.*

Proof.

There are uncountably many subsets of $\mathbb{N}$ ($2^{\aleph_0}$ to be precise).

But there are only countably many Turing machines.

Hence uncountably many problems $A \subseteq \mathbb{N}$ are not decidable.

$\square$

Note that this result does not exhibit any concrete undecidable problem. Here is a more constructive approach.

Halting

Any real algorithm is required to always halt after finitely many steps, no matter what the input looks like.

However, there is no guarantee that a TM M on input x will ever reach a halting state: the computation might continue forever without getting to the halting state.

One sometimes writes $M(x) \downarrow$ or $M(x) = y$ to indicate convergence and $M(x) \uparrow$ to indicate divergence.

This corresponds to a while loop that fails to terminate, or to an unbounded search that continues forever.

Halting is a very serious issue that causes major problems in many places.

Busy Beaver Problem

Here is a famous problem due to Tibor Rado (1962). Consider only TMs on tape alphabet $\Sigma = \{\underline{b}, 1\}$ and n states.

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

We write $\beta(n)$ for this number and refer to $\beta : \mathbb{N} \longrightarrow \mathbb{N}$ as the Busy Beaver function.

Halting is crucial, otherwise we could trivially write infinitely many 1's. We will not insist that the 1's are contiguous (that is a variant of the problem with much the same properties).

Note that it is standard to ignore the halting state in the count, so n means n ordinary states plus one halting state.

Variants

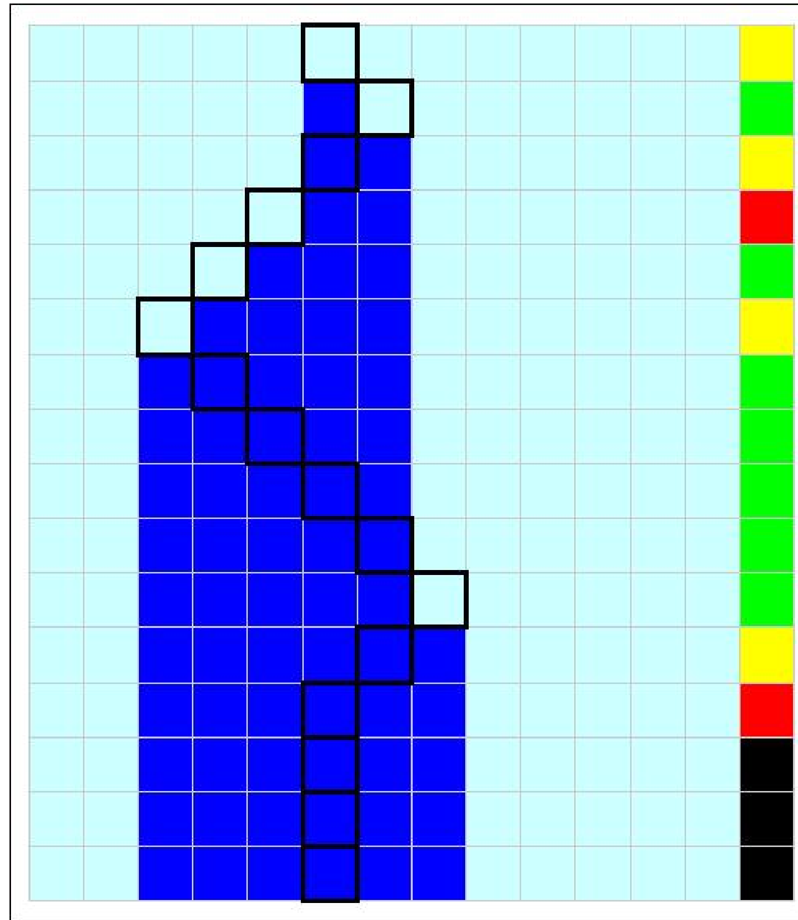
Note that there are several variants of the BBP in the literature:

- What is the largest number written in unary (one contiguous block of 1's) that can be written by a halting n-state machine?
- What is the largest number of moves a halting n-state machine can make (time complexity)?
- What is the largest number of tape cells a halting n-state machine can use (space complexity)?

These are all closely related, we will only discuss the "total number of 1's" version.

Exercise 11. *Establish a hierarchy of corresponding Busy Beaver functions.*

Busy Beaver $n = 3$



Busy Beaver Exercises

Exercise 12. *Derive the transition table of the 3-state Busy Busy machine from the last picture.*

Exercise 13. *Give an intuitive explanation of how this machine works.*

Exercise 14. *Prove that the last machine is indeed the champion: no other halting 3-state machine writes more than 6 ones.*

Exercise 15. *Hard*
Find the Busy Beaver champion for $n = 4$.

Exercise 16. *Extremely Hard*
Find the Busy Beaver champion for $n = 5$.

How bad can it be?

n	1	2	3	4	5	6
$\beta(n)$	1	4	6	13	≥ 4098	$\geq 10^{865}$

Already for $n = 5$ we only have a lower bound, not the exact value. The champion machine is a small miracle.

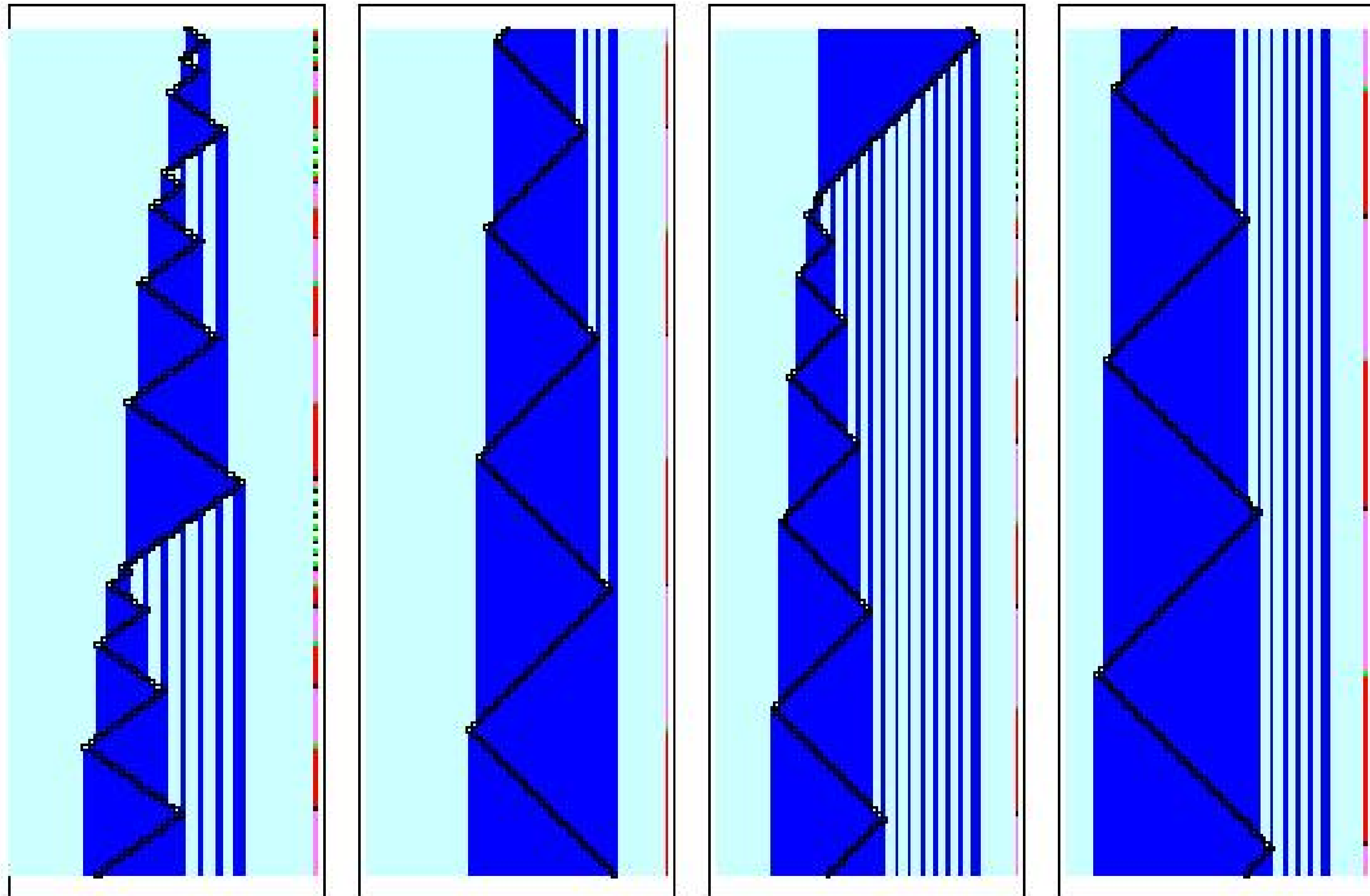
The Marxen-Buntrock Machine

Entry (q, b, X) in position (p, a) means: in state p, scanning an a, go into state q, write b and move X.

	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	(0,1,R)	(1,0,L)

This machine stops after an unbelievable 47176870 moves and leaves 4098 symbols 1 on the tape.

Marxen-Buntrock 800 Steps



Misleading Pictures

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 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 47176870 steps, 5 states are enough to produce such behavior.

Why is this Hard?

There are several fundamental obstructions to computing $\beta(n)$.

- No one has any good theory of the behavior of Turing machines (and for good reason as we will see).
- Hence, one has to use brute-force search, at least to some degree. But the number of Turing machines on n grows wildly exponentially.
- Most important is the last problem: Even if we could somehow consider all machines on, say, 10 states, there is the problem that we don't know if a machine will ever halt – it might just keep running forever.
  So if we run all 10-state TMs in parallel for a trillion years we still won't know $\beta(10)$.

The Key Issue

After a trillion years some of the machines will have halted and we can easily find the one with the longest legitimate output.

But some others are still running.

To get at $\beta(10)$ we would need to filter out those machines that will never halt (note that we could easily run all the others to completion).

The problem is that there is no algorithm to do this filtering (for arbitrarily large values of 10).

Exercise 17. *Figure out what is meant by the last remark.*

A General Simulator

Instead of building all 10 state machines and running them in parallel, we could also us a universal simulator: a machine U that can be set up to *simulate* any arbitrary machine M.

- M gets input x, and halts on y.
- U gets input $\#e\#x\#$ and halts on $\#e\#y\#$.

Here e is simply a flattened transition table for machine M (a program representing M).

For a CS person this is quite boring: U is essentially just an interpreter (or a compiler and run time system).

Of course, U is slower than M: for every step of M it has to look at the e part of the tape to see what action M would take and then perform this action on x.

The speed difference actually would not be terrible: some small polynomial factor.

Universal Turing Machines

Definition 9. *Such a machine is a universal Turing machine (UTM).*

Every UTM provides us with a simple enumeration $(M_e)_{e \geq 0}$ of all possible TMs. One usually fixes one specific UTM once and for all then simply refers to TM "number" e.

Definition 10. *The number e is called an index for Turing machine M_e (wrt. UTM U).*

Again, think of e simply as a program. Note, though, that many programs have the same input/output behavior so that different machines M_e and $M_{e'}$ may well compute the same function.

We can think of the input as being coded as natural numbers e and x, or as strings over some suitable alphabet, it does not matter.

Enumerating Machines

To repeat: here are two crucial facts about Turing machines that are often used tacitly in all kinds of arguments.

- One can effectively list all (say, one-tape) Turing machines, and the listing can be generated by a Turing machine.
- There exists a universal Turing machine that can simulate the machines in this listing.

This is a kind of closure property: a single Turing machine is as powerful as all Turing machines (though, of course, it may be much less efficient for certain computations).

The Halting Problem

Problem: **Halting**
Instance: Index e and input x.
Question: Does machine M_e halt on input x?

Note that this a perfectly well-defined decision problem; the instance here consists of two natural numbers. The effective enumeration (M_e) is fixed once and for all.

Halting is Undecidable

Theorem 2. *The Halting Problem is undecidable.*

Proof.

Otherwise we could build a TM P (for paradox) that, on input e,

- halts on output 0 if M_e on input e fails to halt, and
- halts on output $y1$ if M_e on input e halts on output y.

P is a TM, so it has some index p.

But running $P = M_p$ on input p produces a contradiction.

□

Note that we have used a universal Turing machine in this argument.

Diagonalization

This argument is a modification of Cantor's set-theory proof that the real numbers are uncountable.

It is rather surprising that the entirely non-constructive cardinality argument due to Cantor also turns out to be the main tool in establishing non-computability results.

Undecidability in Mathematics

The first undecidability results by Church and Turing concerned problems in logic, not really mathematics per se.

Church immediately suggested to search for undecidability in pure mathematics. Here are some famous results;

- Post, Markov 1947: Undecidability of the word problem for semigroups.
- Novikov 1955: Undecidability of the word problem for groups.
- Markov 1958: Undecidability of the homeomorphism problem for 4-dimensional manifolds.

Note that the word problem for groups is much harder to tackle than for semigroups; Novikov's seminal paper is 143 pages long.

Hilbert's 10th Problem

Perhaps the most famous example of an undecidability resutl in pure mathematics is Hilbert's 10th problem, the insolubility of Diophantine equations.

A Diophantine equation is a polynomial equation with integer coefficients.

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

Theorem 3. *Y. Matiyasevic, 1970*

It is undecidable whether a Diophantine equation has an integer solution.

It was known prior to Matiyasevic's result that exponential polynomials yield undecidability, but eliminating exponentiation is quite difficult.

Bounding Roots

One might think that integer roots of $P(x_1, \ldots, x_n)$ should be bounded by some simple function of n, the degree d of P, and the largest coefficient.

Perhaps something like

$$\bigl(n! \max(|c_i|)\bigr)^d$$

That would allow brute-force bounded search for the roots.

Note that we are not worried about efficiency yet; all that matters is that a bound would would reduce the problem to an exhaustive search over a finite (albeit huge) search space.

No Way

Finding bounds even in concrete cases turns out to be quite difficult.

Try to find a solution for

$$x^2 - 991y^2 - 1 = 0.$$

Of course, $x = 1$, $y = 0$ is a trivial solution. The smallest positive solution here is

$$\begin{aligned} x &= 379516400906811930638014896080 \\ y &= 12055735790331359447442538767 \end{aligned}$$

Exercise 18. *(in futility) Try to find a positive solution to* $313(x^3 + y^3) = z^3$.

Different Rings

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 (in fact the whole first order theory of the reals is decidable).

Exercise 19. *Why does undecidability over $\mathbb{Z}$ not simply imply undecidability over $\mathbb{Q}$? What is the obstruction?*

The Halting Set

Fix some effective enumeration (M_e) of all Turing machines. Let

$$K = \{ e \in \mathbb{N} \mid M_e \text{ halts on } e \}$$

For simplicity we assume here that everything is coded up as integers.

From the last proof, K is already undecidable.

So how complicated is the Halting Problem?

Are there more complicated problems, or is this as bad as it gets?

Semi-Decidable Sets

Recall that a set $A \subseteq \mathbb{N}$ is decidable iff there is a TM that, on input n, halts with output 1 if $n \in A$, and halts with output 0 otherwise.

Definition 11. *A set is* semi-decidable *if there is a TM that, on input x, halts if x belongs to the set, and fails to halt otherwise.*

Lemma 2. *The Halting set K is semi-decidable.*

Proof. Use a UTM to simulate the computation of M_e on input e. If M_e halts in the simulation, halt too. Otherwise just keep running forever. □

Decidable vs. Semi-Decidable

Lemma 3. *A set is decidable if, and only if, the set and its complement are both semi-decidable.*

Proof. Let A be decidable. The TM M that decides membership in A can easily be turned into machines that semi-decide membership in A and $\mathbb{N} - A$. E.g., for A run M; if it halts on 1 also halt, but if it halts on 1 go into an infinite loop.

For the opposite direction, suppose we have machines M_1 and M_2 that semi-decide A and $\mathbb{N} - A$. Given x, run both machines in parallel on x. One of them must halt, output 0 or correspondingly and halt. □

It's crucial for this argument that we can run two TMs in parallel (time-sharing).

Enumerations

Let us say that

Definition 12. *A TM M enumerates a set A if, possibly running forever, it writes the elements of A on a special write-only output tape, one after the other.*

$$\text{output tape:} \qquad \#a_0\#a_1\#a_2\#\ldots\#a_n\#\ldots$$

The rules are:

- Every element must appear at some point, and nothing else may appear.
- Elements may be repeated.
- The elements do not have to appear in any special order, though.

Of course, for infinite sets the computation never stops.

Computably Enumerable Sets

Definition 13. *A set is computably enumerable (c.e.) or recursively enumerable (r.e.) if it can be enumerated by a Turing machine.*

Lemma 4. *The c.e. sets are precisely the semi-decidable sets.*

Proof. Given a TM that enumerates A we can easily build a TM that semi-decides A: enumerate away and check if x appears. If so, halt.

If A is semi-decidable via TM M we can run M in parallel on multiple inputs. At level s we run M for s steps on $x = 0, 1, \ldots, s$. If M halts on any of these inputs we write them on the output tape. Then we continue with level $s + 1$. $\square$

Summary

- We are only interested in combinatorial problems.
- Computability may seem to be an obvious notion, but its historical development required quite a bit of effort.
- Turing machines seem to capture all the intuitive aspects of computability.
- Decidable and semi-decidable problems appear in many places in mathematics.
- The Halting Problem is the classical example of a semi-decidable problem that fails to be decidable.
- The notion of abstract computability does not translate directly into practical computability.
- But problems that are undecidable in general usually turn out to be very hard to deal with in practical special cases, too.