Introduction to Algorithms and Data Structures

Lecture 29: Introduction to Computability

John Longley

School of Informatics
University of Edinburgh

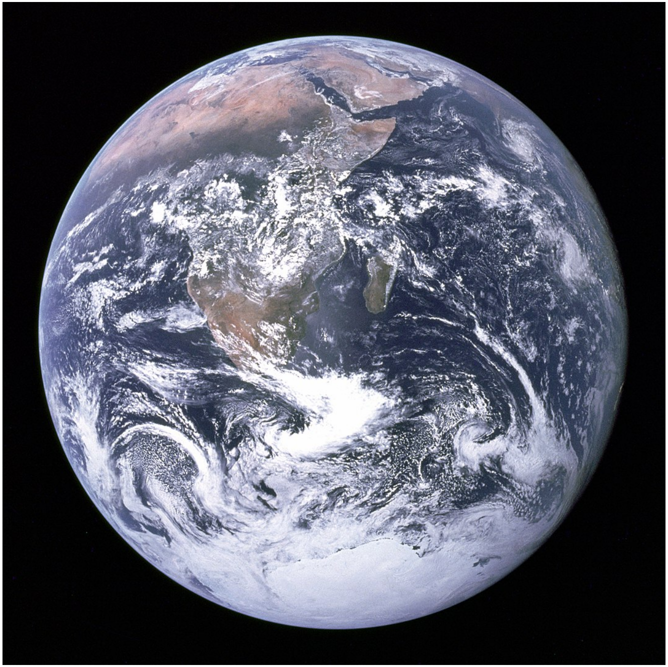
19 March 2024

Limitative results for algorithms

Are there theoretical limits to what algorithms can ever achieve — no matter how clever they are?

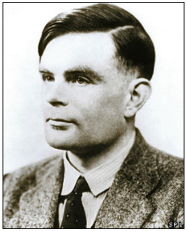
- CLRS 8.1: No general, comparison-based sorting algorithm for n-element lists can ever do better than $\Theta(n \log n)$.
- Lecture 25: It's conjectured that 3-SAT (or any other NP-hard problem) can't be solved in time $O(n^d)$ for any d. (P$\neq$NP.)

  [NB. We have a long way to go. Not even ruled out that 3-SAT is solvable in $O(n)$ time!]
- This+next lecture: Are there problems that can't be solved by any algorithm at all — no matter how much time and space we allow?



Church-Turing computability (c. 1936)

Alonzo Church


Alan Turing

There's a fundamental class of functions — the Church-Turing computable functions — which are generally accepted as coinciding with the 'algorithmically computable' functions (ignoring time and space limitations).

We'll focus on partial functions $f : \mathbb{N} \rightharpoonup \mathbb{N}$.
(After all, any reasonable 'data structure' can ultimately be represented by just 0's and 1's — i.e. as a long binary number!)

Idea extends easily to partial functions $f : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$
(which will sometimes crop up).

The plan . . .

This lecture:

- Precisely define the class of CT-computable functions $\mathbb{N} \rightharpoonup \mathbb{N}$.
- Review evidence that this includes all possible 'algorithmically computable' functions (the Church-Turing thesis).
- Sketch Turing's construction of a universal machine (origin of the general-purpose programmable computer!)
- * Glance at a crazy idea for a 'super-Turing' computer.

Next lecture:

- Show that the so-called halting problem is not CT-solvable.
- Mention other unsolvable problems in CS/maths.
- * Raise some philosophical questions.
- * Plug a book I'm working on.

* Not official course material.

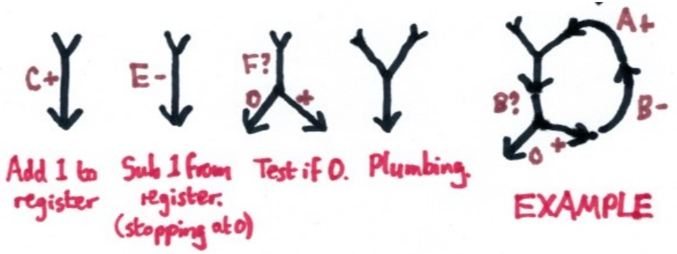
Register machines

Many roads lead to the same class of CT-computable functions. . .

- Church used λ-calculus (origins of functional programming!).
- Turing used FSMs with infinite memory tape ('Turing machines').
- Here we'll use register machines, due to Marvin Minsky.

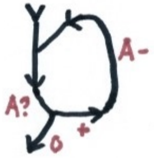
Machines have a fixed, finite set of registers (say A,B,. . . ,I), each capable of storing an arbitrary natural number (i.e. integer ≥ 0).

We build machines by plugging together trivial components:

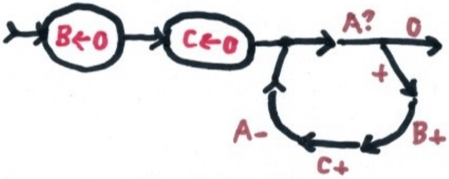


The machine here adds B to A, losing B in the process.

More register machines

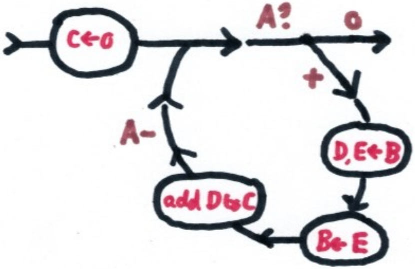


Set A to 0
(A←0)



Copy A to B and C
(losing A).

(B,C←A)



'C←A×B'

Functions computable by register machines

To use a register machine to compute e.g. a partial function $\mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$, we may supply the inputs in A and B registers, and read output from A. (Just a convention.)

So let's say a register machine **M** computes $f : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ if, for any $m, n \in \mathbb{N}$, the following works:

Suppose we set up the registers with $A = m, B = n, C = D = \cdots = 0$, then run **M**.

- The computation will terminate if and only if $f(m, n)$ is defined.
- If it does, the final value in A will be the value of $f(m, n)$.

We may say $f : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ is RM-computable if and only if there's some register machine that computes f.
E.g. $+$ and $*$ are RM-computable.

Same goes e.g. for 1-argument functions.

The Church-Turing thesis

A little goes a long way!
It turns out that the class of RM-computable (unary or binary) partial functions coincides with the class of:

- Functions definable in λ-calculus
- Functions computable by Turing machines
- Functions computable on arbitrary-size natural numbers in your favourite programming language. (Some work needed to define e.g. what a Python program would do if time/memory were unlimited.)

That's because each of these formalisms can simulate the others — e.g. we could write an 'interpreter' for RMs in Python.

From now on, we'll refer to this class (defined in any of these equivalent ways) as the class of Church-Turing computable functions.

The Church-Turing thesis

The Church-Turing thesis claims that, for functions $\mathbb{N}(\times\mathbb{N}) \rightharpoonup \mathbb{N}$,

- the precisely defined class of CT computable functions

. . . coincides with . . .

- the informally recognized class of functions computable by an algorithm.

We understand the latter as an *informal but seemingly clear* concept. E.g. think of what you could compute on paper by following some precise, intuitively 'mechanical' procedure (no choice or creativity) — given unlimited time, paper, patience etc.

Insofar as this is considered as an informal concept, the CT thesis isn't amenable to strict mathematical proof.
Nevertheless, no one seriously doubts it (in the sense that they think it's false).

Why accept the CT thesis?

Arguments sometimes given . . .

1. No one has ever come up with an obviously ‘mechanical’ algorithm that computes anything outside this class.

2. Very many attempts at defining a concept of ‘computable’ function converge on the same class.

3. Turing’s argument: Think about what a human calculator could *in principle* do with:
    - finitely many (distinguishable) mind states
    - unlimited paper, but finitely many (distinguishable) symbols
    - finitely many ‘fingers on the page’.

   This is in essence what Turing machines model.

Take your pick! In any case, can regard the Thesis as solidly established and safe to build on.

Towards Turing's universal machine

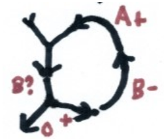
Two key observations re register machines:

- A complete set of register values can be coded up as a single natural number. E.g. the 9-tuple

A = 23	B = 65	C = 00	D = 00	E = 00
F = 00	G = 00	H = 00	I = 01	

  might be coded as 2600000035000001.

- With a bit more work, an entire RM flowchart can also be coded up as a natural number.

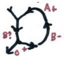
  E.g. our adding machine 

 might be coded as

  10220400401103003050320200204101 (details unimportant).

The universal machine

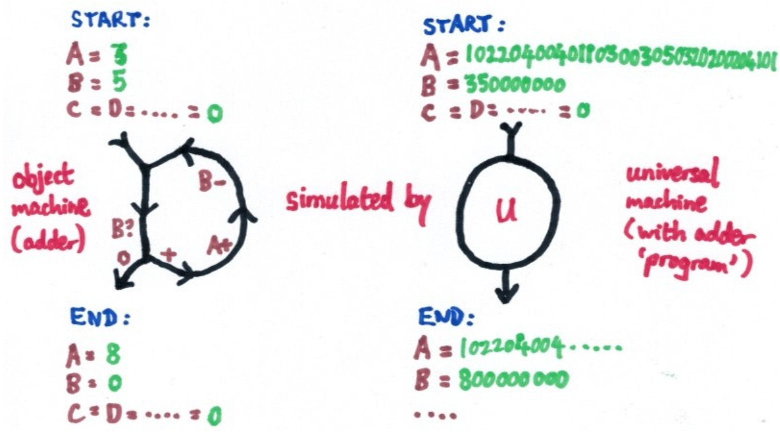
IDEA: If I gave you the numbers 102204004011030030503202002041 01 and 350000000 (and you know what the codings were), you could:

- Recover the flowchart  and register values A=3, B=5, ...
- Simulate the run of this machine with these initial register values.

What's more, this would itself be a purely algorithmic process.

So we can build a register machine to do it!
We'll call it the universal machine U.

The universal machine: illustration



Universal machines, general-purpose computers

With suitable ‘programming’ (in the A register),
our universal machine can simulate any other 9-register machine.
(Or even itself!)

Turing’s insight that ‘all machines could be simulated by just one single machine’ had vast repercussions.

Rather than building separate ‘hardware’ for each computing task, we can build just one piece of hardware which can run many different pieces of ‘software’.

This is really the origin of the modern general-purpose, programmable computer!

(Remaining three slides are NON-EXAMINABLE.)

Philosophical aside: A ‘physical’ Church-Turing thesis?

Both Church and Turing were thinking (initially) of algorithmic computation by humans (abstracted from time/space limitations).

But now that we have other ‘computing devices’, natural to ask whether . . .

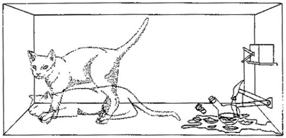
> ??? *Every function $\mathbb{N} \to \mathbb{N}$ computable by any physical device whatever is Church-Turing computable.* ???

Can view this as a fundamental question about the nature of the physical universe:

> *Are we in the sort of universe that allows us (under mild idealizations) to compute non-CT-computable functions?*

Nothing in today’s mainstream (digital) computers offers any hope of going beyond the ‘CT-computable’ functions. But what other kinds of computer might be possible?

Quantum computing



Loosely, QC exploits weird quantum superpositions of multiple 'computation paths' to achieve some kind of massive parallelism.

The race is on to make it work in practice!

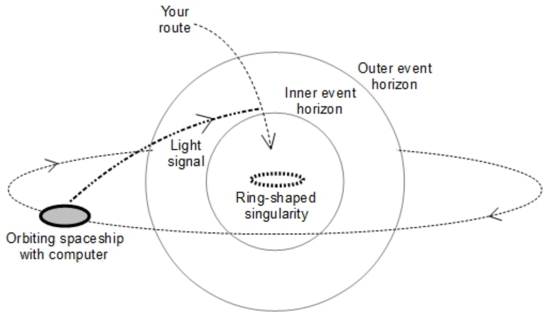
If it did, would it fundamentally change 'what we can compute'?

- In terms of complexity (and in practice!), yes. E.g. integer factorization would become polytime computable (best known classical algorithms are super-polynomial). Bad news for RSA!
- In terms of computability, no. Won't 'break Turing barrier'.

So we'd have to look elsewhere . . .

Thought experiment: Black hole computers

Seriously wild suggestion for ‘breaking the Turing barrier’ by Németi *et al*:



Some fun to be had here. But none of this undermines the importance of classical, Church-Turing computability.