UCT

Complexity Theory

Klaus Sutner

Carnegie Mellon University

Spring 2022

Dramatis Personae

- Prof:
  Klaus Sutner ⟨sutner@cs.cmu.edu⟩

- TAs:
  Magdalen Dobson ⟨mrdobson@andrew.cmu.edu⟩
  Boyang Lyu ⟨boyangly@andrew.cmu.edu⟩

- Course secretary:
  Rosie Battenfelder ⟨rosemary@cs.cmu.edu⟩

Web and Communication

Course Page http://www.cs.cmu.edu/~15455

Piazza https://piazza.com

Make sure to read the syllabus posted on the course site. We will assume that you are familiar with all the rules set out in the document. If you have questions, ask (piazza seems like a good venue, you're probably not the only one).

Books, Computability

. . . are hopelessly outdated and obsolescent (only partially kidding). We have no textbook. If you still feel very attached to books, here are some plausible candidates:

- **H. Enderton**, *Computability Theory: Introduction to Recursion Theory*
- R. Soare, *Recursively Enumerable Sets and Degrees*
- H. Rogers, *Theory of Recursive Functions and Effective Computability*
- P. Odifreddi, *Classical Recursion Theory, Volumes I/II*
- S. Cooper, *Computability Theory*

Books, Complexity

- **M. Sipser**, *Introduction to the Theory of Computation*
- B. Barak, S. Arora, *Computational Complexity: A Modern Approach*
- C. Papadimitriou, *Computational Complexity*
- L. A. Hemaspaandra, M. Ogihara, *The Complexity Theory Companion*
- A. L. Selman, S. Homer, *Computability and Complexity Theory*
- O. Goldreich, *P, NP and NP-Completeness*
- M. Garey, D. Johnson, *Computers and Intractability*
- **C. Moore, S. Mertens**, *The Nature of Computation*

The Big Questions

- What is computability?
- What is feasible computation?

This may sound similar to an algorithms course, but it's actually the exact opposite: we are much more interested in non-computability and non-feasible computation.

Unfortunately, this direction tends to be more difficult: in computability, we need to rule out the existence of any algorithm whatsoever, rather than just exhibiting a single one. In complexity things get worse, since we need to rule out only algorithms obeying certain resource constraints, when there are obvious algorithms ignoring these constraints.

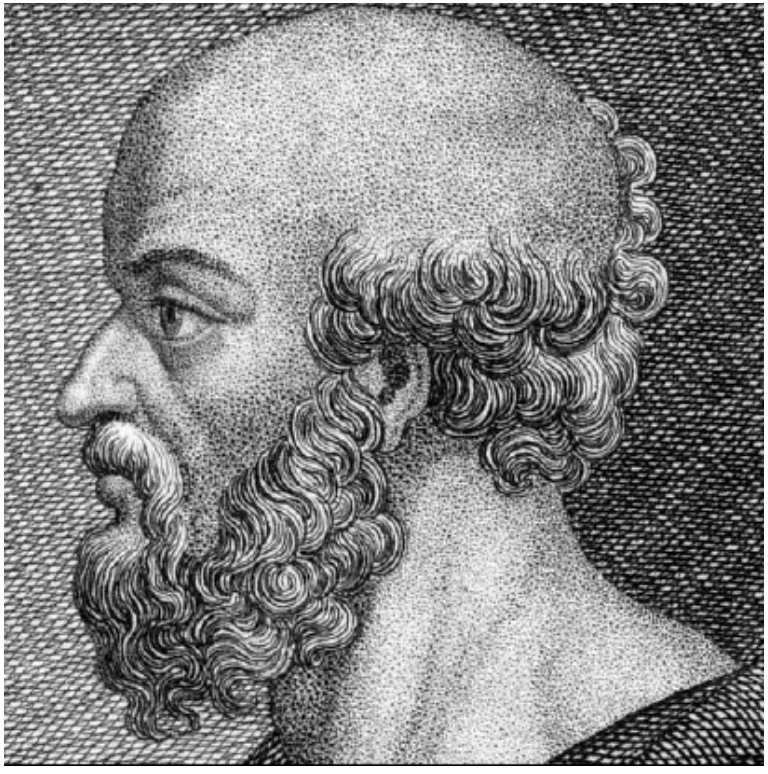
Why Bother With Computability?

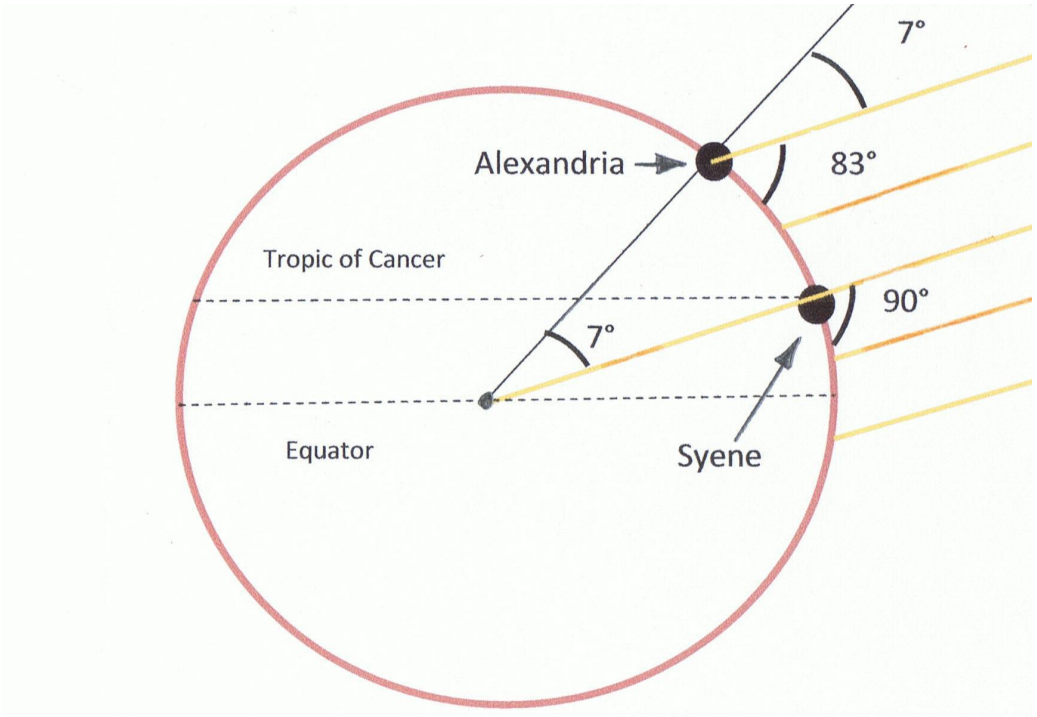
- Many of the fundamental ideas (complexity classes, hardness, reductions, separation, . . .) come directly from classical recursion theory (CRT).
- Some of the basic results are easier to understand in the context of general computability, rather than having to deal with resource bounds.

Here is the critical point: in many ways, old-fashioned computability is easier than modern complexity. This may sound counterintuitive, but bear with me. For example, there are straightforward separation theorems in CRT, whose counterparts in complexity turn out to be rather elusive and often remain open problems.

Early mathematics was very much focused on computation (Plimpton 322, about 1800 BCE). See Pythagorean triples for an explanation.

Eratosthenes





Calculated distance from earth to sun around 240 BCE, error may have been as low as 1%. Also calculated the diameter of the sun, not as accurate.

Around 1530, N. F. Tartaglio managed to solve equations of the form $x^3 + ax^2 + b = 0$.

Here is one of the solutions:

$$\frac{1}{3}\left(\sqrt[3]{-a^3 + \frac{3}{2}(-9b + \sqrt{3}\sqrt{b(4a^3 + 27b)})} + \frac{a^2}{\sqrt[3]{-a^3 + \frac{3}{2}(-9b + \sqrt{3}\sqrt{b(4a^3 + 27b)})}} - a\right)$$

Even just verifying that this is a solution is not so easy, never mind finding it in the first place.

Effective Computation

Aka effective calculation. Mathematics has a long tradition of computing: solving equations, measuring areas and volumes, constructing geometric objects, testing primality, and so on. A more recent example: high precision numerical integration. This stuff, while technically complicated, is often very intuitive and tangible.

But: The theory of computation (aka recursion theory) was developed to solve an entirely different problem.

To wit, the transition towards increased abstraction (and, in a sense, also increased sophistication) really got under way in the 17th and 18th century. Alas, more abstraction easily leads to places where natural intuition becomes unreliable, and very soon there is a manifest problem of rigor. The theory of computation was developed to overcome these problems.

Classical Period

Breakthrough

The invention of calculus in the late 17th century was a huge conceptual and practical breakthrough. The modern world is simply unthinkable without calculus.

And yet, there were also the beginnings of trouble: to wit, calculus was based on infinitesimals, vanishingly small quantities that somehow also were real numbers (though, at the time, no one could define the reals either). Calculus is so natural and intuitive that seasoned practitioners can get great results—even without weight-bearing foundations.

Leibniz realized there was a problem, but had nowhere near the tools to fix the issues, that would take till the 1960s with the discovery of non-standard analysis by A. Robinson.

Euler's Intuition

Leonhard Euler (1707–1773) had perfect intuition and could concoct arguments that were eminently plausible, and led to correct results, but were exceedingly difficult to justify in the modern sense. Here is an example:

Problem: Find a way to calculate e^x for real x.

Wlog $x > 0$. For x reasonably small we can write

$$e^x = 1 + x + error$$

with the error term being small. Alas, we have no idea what exactly the error is.

Infinitesimals

Euler considers an infinitesimal $\delta > 0$. This simplifies matters greatly:

$$e^{\delta} = 1 + \delta$$

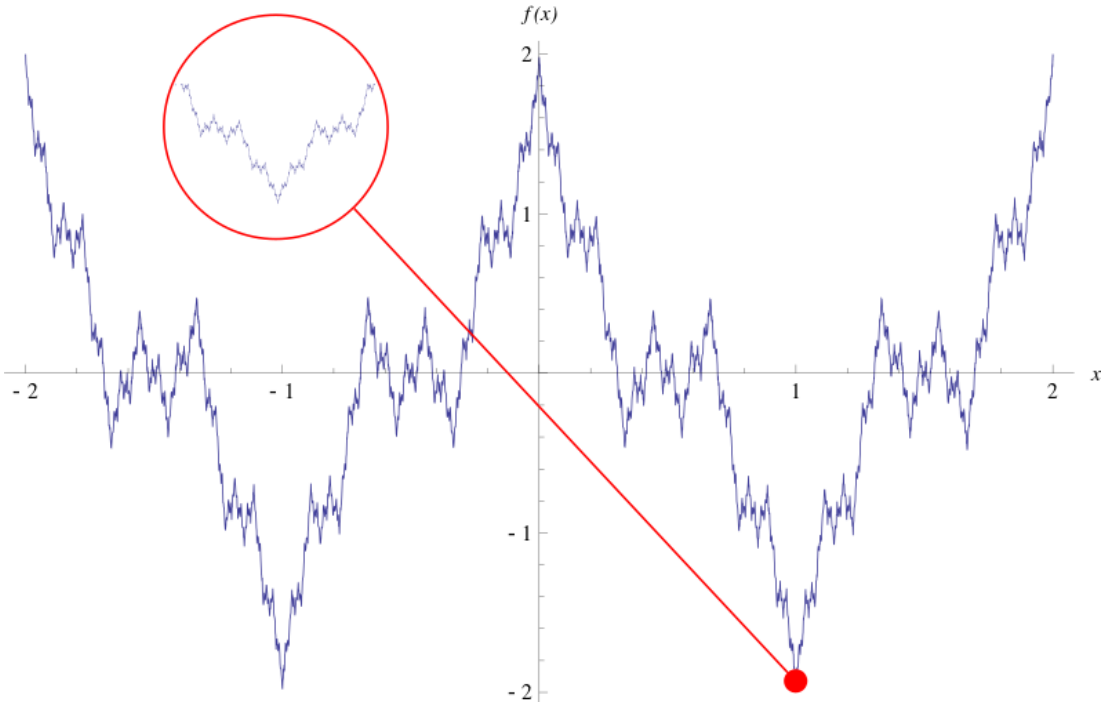
This identity is justified by Leibniz's lex homogeneorum transcendentalis, the transcendental law of homogeneity: proof of the law is by higher authority. Again, this is not Leibniz's fault, he was way ahead of his time.

Once we accept the last identity, we can calculate happily:

$$
\begin{aligned}
e^x &= (e^\delta)^{x/\delta} \\
&= (1+\delta)^{x/\delta} \\
&= 1 + \binom{x/\delta}{1}\delta + \binom{x/\delta}{2}\delta^2 + \dots \\
&= 1 + x + 1/2\, x(x-\delta) + \dots \\
&= 1 + x + 1/2\, x^2 + \dots \\
&= \sum_{i \geq 0} x^i/i!
\end{aligned}
$$

19th Century

Things got worse during the 19th century with the discovery of rather bizarre objects like a Weierstrass's continuous function that is nowhere differentiable.



Even worse, the function is just a Fourier series. Trigonometry creates monsters.

Rigor

Once it became clear that one needed to be very, very careful to avoid "results" that were plain wrong, a number of people started to work on the development of solid foundations for analysis.

- Cauchy and Dedekind gave precise definitions of the reals (Cauchy sequences, Dedekind cuts).
- Weierstrass used limits and ε/δ proofs.
- Cantor developed a fine-grained theory of Fourier analysis.

A combination of logic (G. Frege, G. Peano) and set theory (G. Cantor) seemed like the right mechanisms to build a completely rigorous foundation of math. One would just have to reconstruct math in these frameworks, and everything would be fine.

Alas, that was not meant to be, These tools are excellent, but used naively they lead to a whole number of other problems, even contradictions.

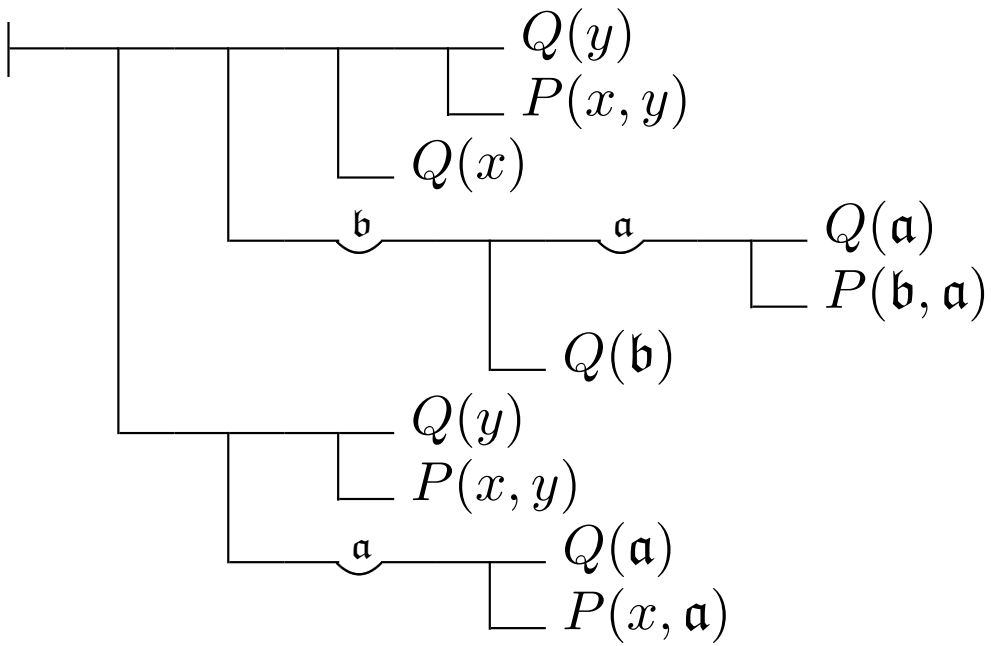
A typical paradox is produced by Russell's self-contradictory set

$$S = \{ x \mid x \notin x \}$$

This set S is rather bizarre and seems utterly useless, but that's beside the point: in Frege's system this set was perfectly legitimate, no less so than, say, the set of prime numbers.

Aside: Worst Notation, Ever

Russell's construction nearly destroyed Frege's lifework: it demonstrated that his logical system, the **Begriffsschrift** (1879) is inconsistent, a disaster even worse than Frege's notation:



A Silver Lining

What was terrible for Frege was actually good for math and CS, though. Russell's observation directly lead to two fundamental developments:

- Axiomatic Set Theory
  Zermelo with a critical contribution by Fraenkel produced the standard reference system for math: Zermelo-Fraenkel set theory, nowadays usually augmented by the Axiom of Choice. Chances are, all the math you ever learned was rooted in **ZFC**.

- Type Theory
  Russell himself developed type theory as a way to rule out problems like his paradoxical set. His system failed as a foundation of math, but types have become critical in TCS.

An Earthquake?

A naive person might think that Russell's discovery would have rocked a lot of boats; actually, it might have capsized them.

But, the establishment has a standard way to deal with crises: ignore them. Just about everybody kept on putzing around happily, as if nothing had ever happened.

A few people like David Hilbert took the problem seriously and decided to do something about it. On the other hand, some, like Henri Poincaré, actively resisted the idea of trying to fix a system they didn't think was broken. Notably, these two were the top mathematicians around 1900.

Poincaré and Hilbert

> Formerly, when one invented a new function, it was to further some practical purpose; today one invents them in order to make incorrect the reasoning of our fathers, and nothing more will ever be accomplished by these inventions.

Hilbert and Logic

- Axiomatization
  Updates Euclid, introduces the de facto modern standard.
- Hilbert's Program
  Construct a foundation of math by strictly finitary means.
- Entscheidungsproblem
  Find a mechanical, definite, finite procedure to determine the validity of any formal statement in an axiomatic system.

The last item directly translates into CS: the challenge is to find a decision algorithm, essentially for all of math.

The other two are also closely connected, but that is not as obvious. Note the constraint *strictly finitary*, this is the place where computation enters the picture.

Axiomatization has become so utterly and completely standard that it is almost no longer noticeable: any halfway serious development will have a clear axiomatic foundation, and proving anything means to derive it from the axioms (though emphatically not in a strictly formal manner).

There may be discussions about how to best axiomatize a particular domain, or even whether certain axioms are justified (axiom of choice in set theory), but there is no serious discussion about whether axiomatization is desirable and even necessary.

2: Hilbert's Program

In the 1920s, partially in response to paradoxes and intuitionistic lunacy, David Hilbert proposed a program to salvage all of mathematics. In a nutshell:

> Formalize mathematics and concoct a finite set of axioms that is is free from contradictions (consistency) and strong enough to prove all theorems of mathematics (completeness). Establish these properties by strictly finitary means.

Consistency means that the system is not self-contradictory, one cannot prove both an assertion and its negation.

Completeness means that all true statements can be proven, so the axioms are strong enough.

So, in a system that is both consistent and complete, we can derive exactly all true statements, the ideal scenario.

Quoi?

Note the hedge "by strictly finitary means" in Hilbert's approach. The details of a Hilbert style formal system can vary quite a bit, but the following characteristics always hold.

- We are only dealing with finite, discrete objects: formulae, axioms, rules of inference, whole proofs, ...
- There is a simple, purely mechanical method to check whether an alleged proof is in fact valid in our system.
- We can enumerate all valid proofs in a purely combinatorial manner, without any reference to insight or creativity.

So what? What does that have to do with computability, much less with complexity theory? Hold on, we're almost there.

To build a formal system, we fix a formal language of logic (usually first-order logic) and then use this language to give a precise definition of a proof: essentially a sequence of formulae that is constructed from axioms and certain logical rules of inference.

The important new idea is that

> A proof is just another mathematical object.
> We can investigate these proof objects just like any other mathematical object.

In fact, we can prove theorems about proofs. That's exactly what we need to do to establish consistency and completeness.

In fact, we can think of a formula and even a whole proof just as a string, a finite sequence of symbols over some suitable alphabet.

Checking whether a particular string really represents a valid proof is entirely mechanical and only requires a finite number of simple steps. With the slightest bit of exaggeration we could say it all comes down to a bit of word processing[†].

The important point here is that we totally ignore the semantics of a proof, we focus entirely on the syntax—and that is mostly straightforward and boring.

Note that we are not claiming that it is easy to find a proof, but checking a given string is not particularly difficult. If fact, that is exactly what makes $\mathbb{NP}$ interesting.

[†]Maybe not in Word, but certainly in emacs.

Computability to the Rescue

If we want to make precise what we mean by a purely mechanical procedure that yields a result after finitely many steps we wind up with the notion of computation.

In particular, to verify that a given string really represents a valid proof we want a proof checker, a decision algorithm that takes strings as inputs and returns yes or no depending on whether the given string really is a valid proof.

This has an important consequence: once we know how to check proofs, we automatically get a proof enumeration algorithm: just run through all possible strings (say, in length-lex order), and filter out the ones that constitute valid proofs.

Checkers versus Provers

For the record: to build a theorem prover is a much harder problem: we want an algorithm that takes as input a formula, and determines whether it has a proof in the system.

This works for some systems, and fails catastrophically for others, to the major dismay of Hilbert.

As we will see, the gap between checkers and provers is very similar to the $\mathbb{P}$ versus $\mathbb{NP}$ problem.

Promising Start

Initially good progress was made towards identifying logical systems that could provide the necessary deductive machinery, without worrying too much at this point about axiomatizing interesting areas such as arithmetic.

- completeness of propositional logic (Boolean logic, Ackermann 1928),
- completeness of predicate logic (aka first-order logic, Gödel 1930).

Propositional logic is far too weak to support interesting mathematics (it will play a major role in complexity, though), but first-order logic seemed very well suited for Hilbert's project. For example, ZFC is typically described as a first-order theory.

A Bombshell

Alas, in 1931, Kurt Gödel essentially wrecked Hilbert's program:

> Any Hilbert system that can express basic arithmetic
> is always incomplete or inconsistent.

So any consistent Hilbert system is always incomplete: there are some true statements that simply cannot be proven in the system.

Worse, the problem cannot be fixed by simply adding the unprovable statement: another true but unprovable statement will pop up in the new system. The problem is not that the designer of the system was careless.

> 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

In modern terminology: find a decision algorithm for statements of mathematics in any axiomatic formal system.

Note that no proof was required, just a one-bit answer.

The Nub of the Problem

Hilbert's Entscheidungsproblem comes down to the following.

> Given a sentence of first-order logic, determine whether the sentence is valid.

Validity here means true in all possible interpretations (all possible structures over which the sentence makes sense). By the completeness theorem, that is equivalent to provability in some suitable formal system.

But provability is "merely" a syntactical notion, it might well be the case that one can decide whether a proof exists or not (truth over all possible structures seems a lot more complicated).

Another Bombshell

Alas, that did not work out, either[†].

Theorem (Turing 1936)

The Halting problem for Turing machines is undecidable.
As a consequence, first-order logic is also undecidable.

In fact, a fairly small fragment of arithmetic known as Robinson arithmetic already suffices (just successor, addition and multiplication, no induction).

A finite set of fairly simple arithmetic axioms is enough to implement Turing machines.

[†] Church had another proof based on his λ-calculus at the same time.

Gödel vs Turing

Gödel has shown, in essence, that in any reasonable formalization of arithmetic there a assertions that can neither be proven nor refuted[†].

If the opposite were true, every assertion is either provable or refutable, then the Entscheidungsproblem would be solvable: just enumerate proofs until you either get to the assertion itself, or its negation.

Turing (and independently Church) showed that the Entscheidungsproblem is indeed unsolvable, even when restricted to a fairly weak subsystem of math.

[†]In the olden days this was expressed by saying "the assertion is undecidable in arithmetic." Hence the title of Gödel's seminal paper: *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme,* We will avoid this terminology like the plague.

> The precise and unquestionably adequate definition of the general concept of formal system [made possible by Turing's work allows the incompleteness theorems to be] proved rigorously for every consistent formal system containing a certain amount of finitary number theory.
>
> K. Gödel, 1963

Models of Computation

- K. Gödel: primitive recursive
- A. Church: λ-calculus
- J. Herbrand, K. Gödel: general recursiveness
- **A. Turing: Turing machines**
- S. C. Kleene: μ-recursive functions
- E. Post: production systems
- H. Wang: Wang machines
- A. A. Markov: Markov algorithms
- M. Minsky; J. C. Shepherdson, H. E. Sturgis: register machines

Listed roughly in historical order. Except for primitive recursive functions[†], these models are all equivalent in a strict technical sense.

[†]Arithmetic functions defined by recursion over a *single* variable. To get full computability one needs recursion over any number of variables as in Herbrand-Gödel

Before the 1940s, a "computer" invariably was a bunch of guys & gals with slide rules sitting in a room for hours and days on end. In this setting, asymptotic behavior does not mean much: all workable instances are pretty small, never mind the error rate. There is no need to worry about sorting a list of a billion numbers or checking whether a 1000-digit number is prime.

Once digital computers became fairly widely available, things changed radically. Computations that were orders of magnitude larger now became feasible. Needless to say, first applications were artillery tables and the construction of a hydrogen bomb.

Still, there were problems that seemed to be rather difficult to handle even in the presence of powerful machines: theorem proving being a prime example[†].

[†] *Logic Theory Machine* by Newell, Shaw and Simon, 1956; introduced linked lists.

Efficient Computation

The notion of effective computability had to be refined to efficient computability: as in the classical case, this is physically realized computation, but with digital computers rather than humans, at a qualitatively different scale. The theory of algorithms helps to push the boundaries of what can be computed efficiently.

By contrast, complexity theory explores the limitations of what algorithms can and cannot accomplish. Like with recursion theory, the focus here is on unsolvability, on the limits of what can be accomplished; on taxonomies of problems. There are new kinds of resources to consider, beyond time and space: nondeterminism, randomness, quantum physics.

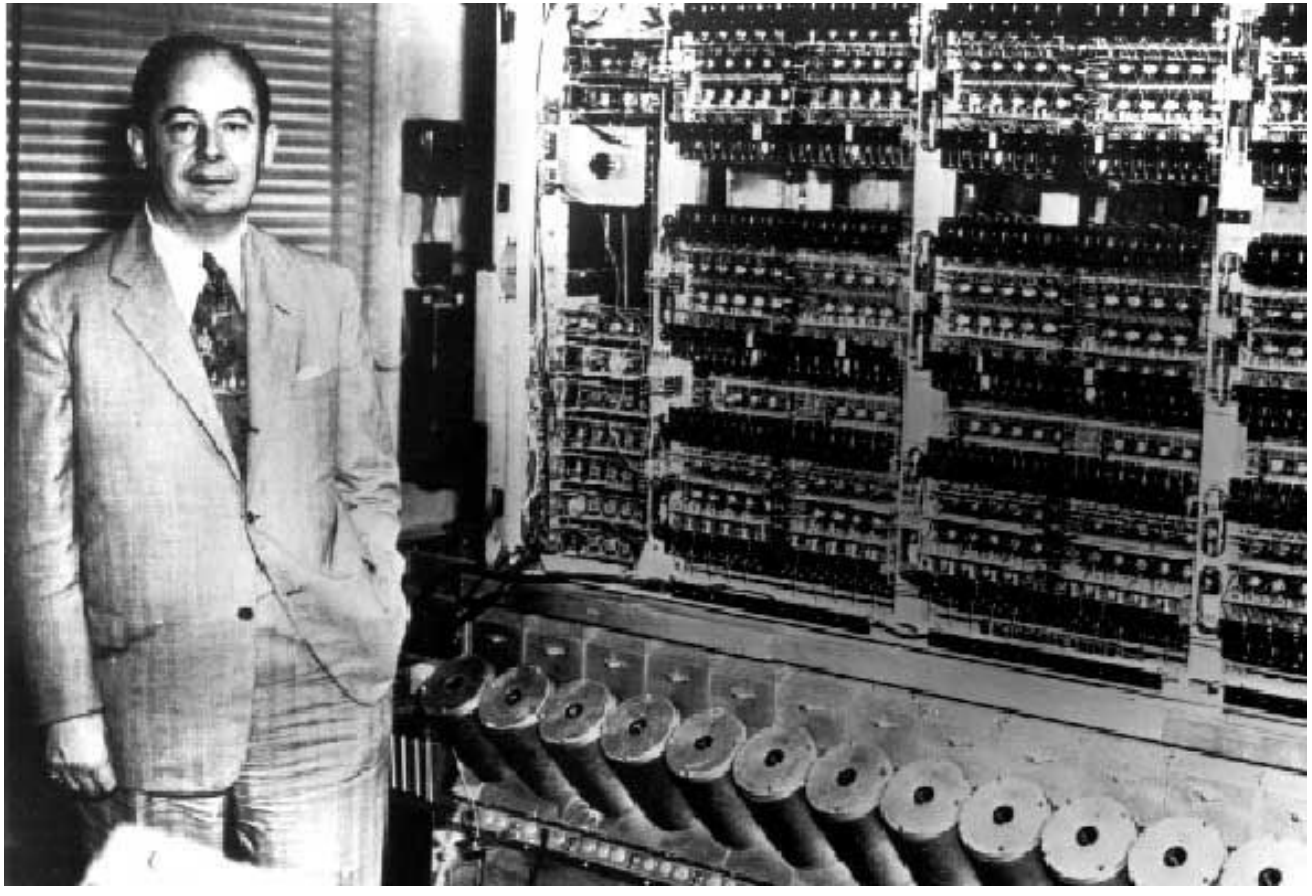
So What?

Clearly, CRT has little to do with many ideas that are central to present day *computer* science: clever data structures, typed programming languages, distributed computing, or even sophisticated algorithm analysis (say, based on complex function theory) play no role here, none whatsoever. It does not help with the construction of algorithms.

But: much of CRT is about non-computability, hardness, unsolvability. The limitations of "effective calculability" are just as important as the actual capabilities.

The same holds in connection with efficient calculability, that is, in complexity theory. In fact, it is not exactly surprising that very similar methods should apply in both scenarios, we are just adding some additional constraints to our models.

Full Disclosure: Some Work Ahead



John von Neumann with ENIAC at UPenn.

John von Neumann

Major contributions to: mathematical logic, functional analysis, ergodic theory, operator algebras, topology, numerical analysis, statistics, quantum mechanics, game theory, digital computing, linear programming, cellular automata.

Known for his apparently inhuman speed and memory.

> I have sometimes wondered whether a brain like von Neumann's does not indicate a species superior to that of man.
>
> Hans Bethe (Nobel Prize Physics, 1967)

Digression: It's a Trap

> Two bicycles, 10 km apart, approaching each other at 10 km/h each, a dog zigzags back and forth at 20 km/h till they meet.
>
> How far did the dog run?

Normal people solve this by realizing that it takes the bicycles $1/2$ hour to meet, or that the dog goes twice as far as the first bicycle. Alas, mathematicians often realize that each segment of the dog's path is a scaled version of the previous one, set up a geometric series, and sum it.

Someone challenged von Neumann. Who shot back with the right answer.

Disappointed questioner: "Mathematicians usually use an infinite series."

Von Neumann's response: "That's what I did."

Ignoring Running Time

> Throughout all modern logic, the only thing that is important is whether a result can be achieved in a finite number of elementary steps or not. The size of the number of steps which are required, on the other hand, is hardly ever a concern of formal logic. Any finite sequence of correct steps is, as a matter of principle, as good as any other. It is a matter of no consequence whether the number is small or large, or even so large that it couldn't possibly be carried out in a lifetime, or in the presumptive lifetime of the stellar universe as we know it.

This is von Neumann in 1951.

This sounds like the beginnings of algorithm analysis.

Getting Serious

> In dealing with automata, this statement must be significantly modified. In the case of an automaton the thing which matters is not only whether it can reach a certain result in a finite number of steps at all but also how many such steps are needed.

In other words, an algorithm alone is not enough, we need to understand its time complexity (among other things).

Logic is Hard

> Everybody who has worked in formal logic will confirm that it is one of the technically most refractory parts of mathematics. The reason for this is that it deals with rigid, all-or-none concepts, and has very little contact with the continuous concept of the real or of the complex number, that is, with mathematical analysis. Yet analysis is the technically most successful and best-elaborated part of mathematics. Thus formal logic is, by the nature of its approach, cut off from the best cultivated portions of mathematics, and forced onto the most difficult part of the mathematical terrain, into combinatorics.

Again von Neumann, this time in 1948. What he calls combinatorics would nowadays be referred to as discrete mathematics.

Complexity is Hard

> The theory of automata, the digital, all-or-none type as discussed up to now, is certainly a chapter in formal logic. It would, therefore, seem that it will have to share this unattractive property of formal logic. It will have to be, from the mathematical point of view, combinatorial rather than analytical.

Ouch, needless to say, complexity theory just like classical recursion theory is all about automata (in the sense von Neumann uses the term here).

It's not too bad, though. The field is fairly well developed at this point, and the basic parts are quite manageable.

Learning Style

Far and away the most important challenge is for you to develop your **intuition**.

Technical details are necessary and indispensable, but intuition comes first—by a long shot.

The same pattern of understanding applies to definitions, theorems and proofs.

Definitions/Theorems

intuition understand what the concept means, what it's purpose is

formal pin things down in a semi-formal way

examples some objects that the definition applies to

counterexpl some objects where it does not, but almost

results the basic theorems associated with the concept

intuition understand the objective precisely, develop a battleplan

formal refine the argument to a semi-formal level[†]

examples what does the proof say about some concrete objects

counterexpl what goes wrong if we change hypotheses/conclusions

results how does the proof help to clarify the given assertion

The last item is particularly important: ideally, a good proof provides additional insights into the claim; it shows not just that it is true, but why it is true.

[†]By a formal proof I will always mean an argument that can be verified by a proof checker. Such formal proofs are closely connected to our subject, but we will not construct them in our arguments.

We will start with a brief recap of general, old-fashioned computability.

Then we switch to complexity. As you will see, there is a lot of repetition and analogy—though things invariably get messier in the complexity world. So enjoy the clean, simple world of computability while it lasts.

What you should know already:

- At least one model of computation (Turing machines).
- Coding functions (computable bijections $\mathbb{N}^{<\omega} \to \mathbb{N}$).
- Existence of universal machines.
- Existence of semidecidable but undecidable sets (Halting).
- Reductions between Halting and other problems.

Key Ideas

- Robustness (equivalent models of computation, Church-Turing thesis)
- Universality (code is data)
- Undecidability (semidecidability)
- Reductions and degrees (oracle machines)
- Weirdness
  - Rapid growth
  - Turing degrees
  - Maximal sets
  - Recursion Theorem