Theory of Computation (CS 422/MAS480B)

Lecturer:
Otfried Cheong

Lecture time: Wed, Fri 10:30–11:45

Course webpage:
http://otfried-cheong.appspot.com/courses/cs422

Warning: This is really a math course. It will cover concepts and proofs, and not so many facts.

We have a BBS for the course on Glassboard (www.glassboard.com).

You must regularly check the course board on Glassboard for announcements. You can ask questions there in English or Korean.

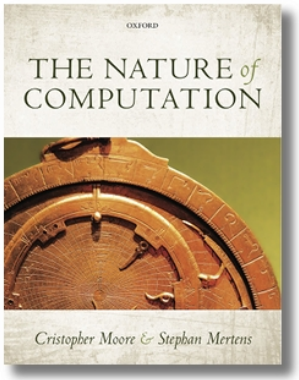
It is okay to register on Glassboard with a Nickname if it makes it easier for you to ask questions.

There is a Glassboard app for iOS and Android. It will make sure you get notifications for new posts on the BBS.

Invitation code: `ykpav`

Textbook

The Nature of Computation by Cristopher Moore and Stephan Mertens.
Expensive and has about 1000 pages.
The book is available on course reserve in the library.



We will follow the book closely.

I am not an expert on complexity theory and hope to learn a lot in this course, together with you!

Homework

Small homeworks on paper. You will have about one week for each homework. Start early on your homework, even if you just read the questions!

Grading Policy

Homework (20%), Quizzes (70%), Participation (10%).

Exams

I plan to have three quizzes during normal class hours.

Attendance

We will take attendance in nearly every class. You can miss four classes without penalty. This is meant so you can have doctor's appointments, interviews, award ceremonies, etc. No special excuses are given for such events.

Head-banging sessions

Once a week you meet with a small group of students and our TA to solve practice problems in a team of two or three.

You present your solution or ideas on the blackboard, others can comment and you can compare solutions.

Optional at the start of the semester. Come if you want.

We will have a survey of dates soon (check on Glassboard!).

Head-banging sessions will be held in Korean.

Theory of Computation

About problems solved by computers.

- Which problems can be solved? Computability
- How fast can a problem be solved? Complexity

It is impossible to give provable answers to these questions without a formal, mathematical definition of

problem and computer

- Can we discuss "computation" without talking about computers, electronics, and physics?
- Can we define (mathematically) a computer?
- Can a computer solve any problem?

A dinosaur course

The first computer science departments were created in the late 1960s, early 1970s.

Students learnt FORTRAN, assembler, the computer architecture of a PDP-11, shell sort, etc.

Some material of this course is older than 1970, and has survived practically unchanged (like turtles and sharks).

Two fundamental concepts

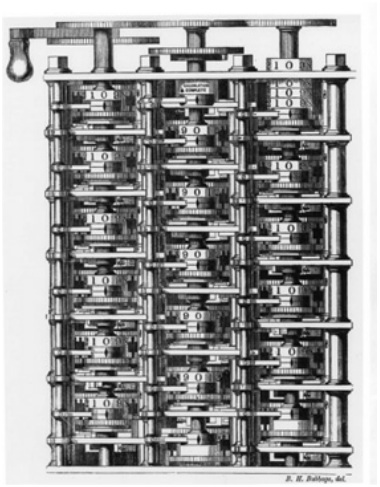
- Algorithms/Programs
  Algorithms were used since antiquity, but became an object of study only early in the 20th century.

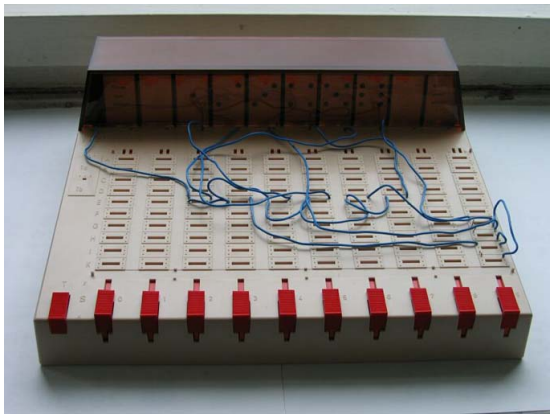
- Universal Computer
  In early "computers", the program was part of the hardware. You had to rewire the computer to run a different program.
  In contrast, a universal computer is a computer that can run any program.

1936 was the *annus mirabilis* of computation, the year where everything came to fit together.

Before universal computers

Charles Babbage's Differential Engine for computing tables of trigonometric functions, logarithms, etc.





Toy computer

Universal Computer

What shall we do to get rid of Mr. Babbage and his calculating machine? Surely if completed it would be worthless as far as science is concerned?

British Prime Minister Sir Robert Peel, 1842

Punched cards already existed in the early 19th century. Babbage's Analytical Engine used them to design a computer that can execute any program given to it.

A universal computer can execute an interpreter ("universal program"): It reads the source code of another program and executes it.

$$U(\Pi, x) = \Pi(x)$$

Universality implies non-halting

Consider the special case where $x = \Pi$

$$U(\Pi, \Pi) = \Pi(\Pi)$$

Assume that Π returns a Boolean value, and define a new program:

$$V(\Pi) = \overline{\Pi(\Pi)}$$

And now we can run V by giving it its own source code as input:

$$V(V) = \overline{V(V)}$$

$V(V)$ can never terminate, because either result would be a contradiction!

This is a form of Cantor's diagonalization argument.

Hilbert's problems

1900 David Hilbert gave a speech to the International Congress of Mathematicians presenting 23 open problems.

Hilbert's 10th Problem:

Specify a procedure which, in a finite number of operations, enables one to determine whether or not a given Diophantine equation with an arbitrary number of variables has an integer solution.

polynomial equation with integer coefficients

1928, Hilbert posed the Entscheidungsproblem:

The Entscheidungsproblem is solved if one knows a procedure that allows one to decide the validity of a given logical expression by a finite number of operations.

What is a "procedure"

Hilbert didn't say what he meant be a procedure, and computers were still decades away. He wanted it to be carried out by a mathematician following a clear sequence of operations.

The early 20th century saw mathematicians struggle with putting mathematics on a foundation of set theory and logic.

Russell's paradox: $R = \{S \mid S \notin S\}$.

1936: The Halting Problem is undecidable, and so the Entscheidungsproblem is unsolvable.

1970: Hilbert's 10th problem is undecidable.

Unprovable truth

A formal system has a set of axioms and rules of inference.

A theorem is a statement that can be proved, with some finite chain of reasoning, from the axioms.

A formal system is consistent if there is no statement T such that both T and $\overline{T}$ are theorems, and complete if, for all T, at least one of T or $\overline{T}$ is a theorem.

In 1931, Kurt Gödel proved that no sufficiently powerful formal system is both consistent and complete.

His proof constructs a self-referential statement that says:

This statement cannot be proved.

Kurt Gödel

1906–1978

Incompleteness theorem 1931: Any sufficiently powerful formal system contains true statements that cannot be proven inside the system.

Defining computation

There were several attempts to define computation. Mathematicians concentrated on computable functions.

- Primitive recursive functions are functions built from simpler functions (with zero and the successor function as the basis) and primitive recursion. It is equivalent to a straight-line program with for-loops.

The Ackermann-function is not primitive recursive.

- Partial recursive functions add μ-recursion (equivalent to while-loops).
- λ-calculus defines computation by functions that operate on strings:

$$(\lambda x : xax)bc \to bcabc$$

Alonzo Church

1903–1995

First language for programs:

- λ-calculus
- formal algebraic language for computable functions

The grand unification

Partial recursive functions and the λ-calculus turned out to be equivalent and are powerful. But do they cover everything that Hilbert would have called a "finite procedure"?

There were other definitions of recursive functions, and Gödel thought the λ-calculus to be "thoroughly unsatisfactory".

The Turing machine was more convincing to many people, and it is equivalent to partial recursive functions and to the λ-calculus.

Alan Turing

1912–1954

Won World War II.

Mathematically defined a computer (Turing machine): *On computable numbers, with an application to the Entscheidunsproblem* (1936).

Proved that uncomputable functions exist (*halting problem*).

Church-Turing Thesis

[A Turing machine] is able to imitate any automaton, even a much more complicated one. . . It has reached a certain minimum level of complexity. . . an automaton of this complexity can, when given suitable instrcutions, do anything that can be done by automata at all.

John von Neumann

Any system that can simulate a Turing machine can carry out any computation at all. It is computationally universal.