Theory of Computation

- A natural consequence of "being in the business" of determining, designing, and implementing algorithms is a desire to find out how far we can go:
  - Are there limits to what algorithms can do?
  - Are there limits to what our machines can do?
  - What relationship exists between the machines that we build and the algorithms that they perform?
- The *theory of computation* asks and investigates these questions — and already has answers to some of them

"Reasoning About" Computing

- So where do we start? Algorithms aren't "natural phenomena," as organisms are for biology, or matter and energy for chemistry and physics — we can't just go "out in the field" to study and observe them
- What we need is a formal set of definitions and rules that we can use to "think about" algorithms in an organized and systematic way — in the same way that can reason about numbers (number theory), shapes (geometry), and data (relational model)
- This is where Alan Turing comes in

Who's Your Daddy?

- Alan Turing (1912-1954) is widely considered to be the father of modern computer science, and here's why:
  - In 1936, he proposed the *Turing machine*, which has turned out to be the prime "formalism" that allows us to study algorithms in a systematic way
  - With the Turing machine, truths about computing can be *proven* unequivocally — and they have been
- Turing also devised the *Turing test* (1950), an early standard for determining whether a machine exhibited "sentience" or intelligence

Fundamentals

- We start with a core idea of what an algorithm does: we can think of it as computing a *function*
- In this context, a *function* is more than just a numeric calculation; it is a "conversion" from some object (input) into another object (output)
  - The "plus one" function takes a number, and converts it into that number plus one
  - The "sort" function takes a list, and converts it into another list
  - The "query" function takes some data and constraints, and converts it into another set of data (that satisfies the given constraints)
- If this function "conversion" can be expressed as an algorithm, then that function is said to be *computable*

The Turing Machine

- Given that functions map elements of one set into elements of another set, the *Turing machine* performs the computation from one set to another:
  - It has an *alphabet* of symbols, a set of *states*, an infinite *tape* consisting of cells that each hold one symbol from the alphabet, and a set of *instructions* which list, for a given state and *input symbol*, a resulting *output symbol*, a *direction* to move, and a *new state* to enter
  - The Turing machine starts in some *initial* or *start state*, with the tape holding an initial set of symbols and its "read/write head" positioned at some location on the tape
  - The machine operates by "reading" the symbol on the tape, finding the instruction for that symbol and the current state, then writing/moving/changing to a new state
- Recall that this was 1936 — before *any* modern digital computing device had been successfully built

- The Turing machine repeats this cycle of reading symbols and following instructions until it reaches one of a predefined set of *halt* or *accept states* — when it reaches such a state, the content of the tape holds the "answer" to the function
- If a function can be computed by a Turing machine, then this function is said to be *Turing computable*
- Some functions just answer "yes" or "no" to some input — if a Turing machine that provides this answer can be built, then that function is *Turing decidable*
- The *Church-Turing thesis* (it isn't a theorem because it's widely accepted but can't be proven) states that the sets of computable and Turing computable functions are one and the same

The Limits of Computation

- The ultimate formulation of the Turing machine is the *universal Turing machine* — or *U* for short
  - *U*'s tape is "loaded" with the *complete instruction set* — the *program* — for some other Turing machine *M*
  - *U*'s own sets of states and instructions allow *U* to *read* this instruction set then *perform* the computation that is done by *M* — thus *U* can be *any* Turing machine in existence!
- With the preceding concepts, it is now possible to prove whether or not *all* functions are computable
- First, we can see that a computable function can be represented as a finite sequence of symbols from a finite alphabet (since it can be placed on *U*'s tape)

- The set of all finite sequences of symbols from a finite alphabet is *countably infinite* — that is, it has the same number of members as the set of natural numbers
- Now, how about the set of all possible *functions*? Let's reduce this to just one kind of function: given a natural number, state whether that number is "in" or "out"
  - For example, the function that computes whether or not a number is even belongs to this category: for every input {1, 2, 3, 4, 5, …} we get output {out, in, out, in, out, …}
  - So does a function that computes whether or not a number is 3 (useless as that may seem): input {1, 2, 3, 4, 5, 6, 7, …} has output {out, out, in, out, out, out, …}
- Thus, the list of *all* functions of this type is the list of all countably infinite combinations of "in" and "out"
- But there are *more* of those combinations than there are natural numbers — and therefore there are more functions overall than there are programs

"Show Me the Function!"

- One such uncomputable function — specifically, it is undecidable since the expected answer is simply a "yes" or "no" — is the *halting problem*
- Simply stated, the halting problem poses this question: given a program and some input to that program, is it possible to compute whether or not that program will stop (output "yes" if it will stop, "no" if it won't)?
- Remember that this is *one* program (let's call it *halt(p)*, where *p* is the program being tested) which, for *any* program *p*, will correctly say "yes" if *p* stops

- The proof is somewhat mindbending, but it works — first, suppose *halt(p)* does exist
  - Now we create a new program, called *uh-oh(p)*:
    - Run *halt(p)*, and put the answer in a variable called *answer*
    - If *answer* is "yes" then run forever, else (i.e., *answer* is "no") stop
  - Now, let's run *uh-oh(p)* with *itself* as the input — in other words, let's run *uh-oh(uh-oh*'s program*)*
  - If *uh-oh* stops, then the *answer* variable contains "yes" — in which case *uh-oh* will run forever!
  - If *uh-oh* doesn't stop, then the *answer* variable contains "no" — in which case *uh-oh* will stop!
- We've just specified an impossible program, and have therefore proven that *halt(p)* does not exist

Problem Complexity

- While one avenue of theory explores whether some problems are *solvable*, another avenue is interested in problems that *are* solvable but whose solutions may not be *practical* enough for real-world use
- This area of theory is concerned with *complexity* — and you've already seen a hint of it in our discussion of an algorithm's *big theta*, or the rate at which a problem grows based on the size of its input
- A study of various solvable problems reveals that there are different *classes* of problems based on their Θ

- Complexity can refer to either *time* or *space* — that is, a problem's "growth" can be studied either in terms of how long it takes to find a solution or in terms of how much *memory* is needed to find that solution
- In any case, an important class of problems, known simply as *P*, consists of problems with *polynomial* complexity — that is, they grow in a manner that is bounded by a polynomial expression
- Generally speaking, problems that do not have polynomial complexity — for example, problems of complexity 2^n — are viewed to be impractical or *intractable*: that is, they grow so fast that solving even moderately-sized versions of the problem would simply take way too long (or too much memory)

An Unanswered Question

- Thus far we have seen *unsolvable* problems (such as the halting problem), *intractable* problems (problems beyond polynomial complexity), and *P* (problems of polynomial complexity)
- Computer scientists have identified one more class of problems, which has been called *NP*, short for *nondeterministic polynomial*
- What is *nondeterministic*? Informally, nondeterminism is guesswork, whether by tossing a coin, rolling a die, or just going on a hunch

- A classic *NP* problem is the *traveling salesman*: given a salesman, a number of stops in a city, and a maximum allowable "mileage," find a travel plan that visits every stop without exceeding the maximum mileage
  - The straightforward solution of testing all possible plans until you find one that doesn't exceed the maximum mileage is *not* of polynomial complexity
  - However, if you allow yourself to just "guess" a solution, then computing and verifying that guess *is* doable in polynomial time
- That is the essence of *NP*: problems that don't have known polynomial solutions unless you allow guesswork (or non-determinism)
- The relationship of *P* and *NP* is one of the great unanswered questions of computer science: are they really one and the same set of problems? Or is *NP* genuinely distinct? This is *not yet known!*

Real-World Impact

- While the usefulness of tractable solutions should be fairly apparent (after all, without them computers wouldn't be of any use!), *intractable* problems also have a place in real-world systems
- In particular, current *encryption* and *decryption* algorithms — the stuff that makes sure that no one can eavesdrop on us over the 'net — *depend* on the intractability of certain problems to be effective
- Put simply, cryptography works because it takes an *impractically long time* to crack