https://users.soe.ucsc.edu/~sesh/cmps132-15/cmps132-15.html

CMPS132

Computability and Complexity Theory

Notes by: Andrew Edwards

Dr. Seshadhri Comandur • (Spring) 2015 • UC Santa Cruz

Last Revision: April 20, 2015

Table of Contents

Abstract

TODO

1 Introduction

The various disciplines of computer science can roughly be split into those focusing primarily on application or and those focusing on theory. We see the same division in most sciences: theoretical versus experimental physics, theories of molecular biology and laboratory experiments, pure and applied mathematics, etc. However, in computer science, the two sides are more closely intertwined, and the relationship between theory and application has a unique and far-reaching feature.

Theoretical computer science research puts physical limitations on what is acheivable by computation, and hence by computers. This top-down causality is peculiar and noteworthy. In physics, a theory must be supported by experimentation in order to be accepted, and more often than not, the theory is physically motivated from the outset. Biology is an even more extreme example of scientific law guided by physical experiment. As we will see, the physical ability to harness the power of computation is constrained by a purely mathematical result.

But what exactly do we mean by computation?

1.1 What is Computation?

For now, we appeal to intuition. Computation is the process of taking a set of **inputs** to an **output**, following some well-defined sequence of steps. The description of these steps is called an **algorithm**.

Computation is the mechanism of transforming information via some specific set of rules. Hitherto we have not defined *what form* this set of rules may take, nor have we even defined what form the input can take. For now, we simply assume that the input is **finite** and that the description for how the computation proceeds is also finite.

Every computer you have ever used operates on these basic principles. Program input is received through the keyboard and mouse, and potentially from complex commands to fetch input data from storage or from the Internet. Output can be displayed on a screen, written to a binary file, or "piped" into another program. Programs themselves come in a multitude of forms. A program is often written in a high level language like `Java`, `C`, or `Python`. Some other program then either interprets (in the case of `Java` and `Python`) or compiles (in the case of `C`) the program, so that the high level commands get converted into a low level, machine executable form.

The capabilities of modern computers are awesome. No human can claim superiority to a computer in a game of checkers or chess. Physics and graphics engines generate strikingly realistic scenes and simulations. What limits, if any, are there on what we can acheive via computation?

1.2 Hilbert's 23 Problems and Hilbert's Program

David Hilbert was a prominent mathematician of the late nineteenth and early twentieth century. His research areas included functional analysis, geometry, and mathematical logic. Hilbert also was one of the first mathematicians to distinguish between mathematics and metmathematics. [1]

In 1900, Hilbert presented a list of 23 unsolved problems at the International Congress of Mathematicians in Paris. The tenth problem of the list is,

> Find an algorithm to determine whether a given polynomial Diophantine equation with integer coefficients has an integer solution, or prove that no such algorithm exists.

In general, a Diophantine equation is one whose sought after solutions must be integers. For example, consider

$$x^n + y^n = z^n$$

If $n = 1$, then there are infinitely many solutions. The set

$$\{(x, y, z) = (p, q, p + q) : p, q \in \mathbb{Z}\}$$

gives a parametrization of all solutions. If $n = 2$, we reach the familiar Pythagorean equation. We can parametrize all solutions by the set [2]

$$\{(x, y, z) = (p^2 - q^2, 2pq, p^2 + q^2) : p, q \in \mathbb{Z}, p > q\}.$$

However, for all $n > 2$, this Diophantine equation has no solutions. This fact is known as Fermat's Last Theorem, and took mathematicians centuries to prove (it was finally proven by Andrew Wiles in 1994). Similarly, the (somewhat trivial) example $x^2 + 1 = 0$ has no integer solutions. So clearly some Diophantine equations have solutions and some do not. Hilbert's tenth problem asks for an algorithm that takes as input a Diophantine equation an returns (say) "TRUE" if a solution exists and "FALSE" otherwise.

It turns out that no such algorithm exists. Over the course of 21 years, mathematicians Yuri Matiyasevich, Martin Davis, Julia Robins, and Hilary Putnam worked on the problem, and eventually produced the MDRP theorem, which (essentially) proves that no such algorithm exists. Their proof draws an equivalence between the set of solutions to a given Diophantine equiation and computably enumerable sets (a concept we will study later). Their proof involves very sophisticated mathematics, so for now we will have to trust the mathematical community on the validity of this result.

Knowing that Hilbert's tenth problem cannot be solved by an algorithm, we have our first theorem:

Theorem 1.1. There exist problems unsolveable by computation.

[1] http://plato.stanford.edu/entries/hilbert-program/

[2] See Chapter 0 of Allen Hatcher's *Topology of Numbers* for a great exposition of this result.

We will not be able to prove this result just yet; we will come back to it in Chapter 2. However, given a specific model of computation, it's to not too hard to concoct an unsolveable problem.

1.3 Computing Functions

Consider the following set of functions

$$\mathcal{F} = \{f | f : \mathbb{N} \to \{0, 1\}\}$$

In English, $\mathcal{F}$ is the set of all functions mapping each natural number $n \in \mathbb{N} = \{0, 1, \ldots\}$ to either zero or one. Now, suppose we pick a specific function $f \in \mathcal{F}$. Can we compute f?

Clearly the answer depends on how we do the computing! As we saw in CMPS130, a Pushdown Automaton (PDA) is strictly more powerful than a Deterministic Finite Automaton (DFA), and a Turing Machine (TM) is strictly more powerful than a PDA. For now, let's not be too concerned with formality. Let's pick a familiar, powerful model of computation: `C` programs.

As an example, consider the function

$$f(n) = \begin{cases} 0 & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases}$$

The following `C` function, which we'll call `f.c`.

```
1  #include <stdio.h>
2  int main(int argc, char** argv) {
3      int n = argc - 1;
4      printf(``%d'', n % 2);
5  }
```

Here is the result of compiling and running this program on a Unix machine:

```
1  $ gcc f.c -o f
2  $ ./f
3    0
4  $ ./f 1
5    1
6  $ ./f 1 1
7    0
8  $ ./f 1 1 1
9    1
```

So our program `f.c` can compute $f(n)$ by passing in n space separated arguments to the compiled program. To keep it simple we can use a space separated unary encoding of n. We can alter this program in a virtually unlimited number of ways to compute other functions, such as $g(n) = 1$ if n is odd, $h(n) = 1$ if n is greater than 42, and $p(n) = 1$ if n is a prime number.

Does every function $f \in \mathcal{F}$ have a corresponding program `f.c`? We are about to see that the answer is no, which leads us to

Theorem 1.2. There exists a function $f \in \mathcal{F}$ such that no `C` program computes f.

Note the difference between Theorem 1.1 and Theorem 1.2. The first claims that there exist problems that **no** model of computation can solve. The second makes a weaker claim: given a specific model of computation (`C` programs), there exist problems that cannot be solved by the model.
The proof is not too hard, and it takes advantage of the following simple fact:

Every `C` *program is a finite string of characters.*

We now prove Theorem 1.2.

Proof. Imagine listing all finite character strings (say ASCII strings to keep things simple). Most of these won't correspond to a `C` program, but some of them do. We can imagine stepping through all the character strings of length 1, all of length 2, ad infinitum, and trying to compile the string with `gcc`. We keep all programs that compile and, when run via `./f` $\underbrace{\texttt{1 1} \ldots \texttt{1}}_{n}$, print out zero or one. We can just discard those that don't.[3]

We will eventually list all valid programs in our model. Let's label the program as `f_0.c`, `f_1.c`, ... For each program `f_i.c`, let $f_i(n)$ be the funciton this program computes. We can view all the functions and their outputs with a rectangular grid:

n	0	1	2	...
$f_0(n)$	0	0	0	...
$f_1(n)$	1	0	1	...
$f_2(n)$	1	1	0	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Now, consider the following function:

$$f(n) = 1 - f_n(n)$$

This is a well defined function. Given n, we can find the n^{th} `C` program `f_n.c` and compute $f_n(n)$. Then $1 - f_n(n)$ gives us $f(n)$.

The question is, does f appear in our list?

This is the crux of the proof: f *does not* appear in the list. It disagrees with every function in the list on where to map the input n. But this list contains all functions computable by `C` programs! Therefore, f is not computable.

This proves Theorem 1.2. □

[3] If you are worried about the program printing stuff other than zero or one, we can change our model to only allow `printf("0")` and `printf("1")` print statements. Or we could interpret any character that is not 0 to correspond to a 1. Our intention is not to be overly technical or pedantic at this point.

The argument above is called a **diagonal argument**, and its origins trace back to the German mathematician Georg Cantor. In 1891 he proved that there is no bijection between the set T of all infinite binary sequences and the set of natural numbers. Following this style of reasoning, one can show that there is no bijection between the set of real numbers and the set of natural numbers, and that there is no bijection between the natural numbers and the set of all subsets of natural numbers.

1.4 A Look Ahead

TODO

2 Computational Models

In this chapter we will explore three models of computation. The first we will call $\mathcal{C}$ programs. $\mathcal{C}$ programs are like Cprograms, but they have a much more limited syntax. Each $\mathcal{C}$ operation has an analogous Coperation, but not the other way around. In fact, there are only *three* basic operations $\mathcal{C}$, but as we will see, it is just as powerful as C. limited subset of the language, which is the first surprise of computability theory:

1. *Few rules give rise to all computation.*

The second model is the Turing Machine. We assume the reader has seen Turing Machines before, but we will review the notation and the semantics. The third model is called μ-Recursive functions. If you have ever programmed in a functional programming language, then this formalism will be familiar. Unlike C programs and Turing Machines, μ-Recursive functions operate exclusively on natural numbers.

On the surface, these three models seem to be very different. This brings us to the second surprise of computability theory:

2. *All models of computation are equivalent.*

2.1 $\mathcal{C}$ Programs

The first surprise of computability theory is that only a few rules are required to acheive the full power of computation. We demonstrate that in this chapter.

Writing $\mathcal{C}$ programs is very simple. All programs have the following form:

- Each take a fixed number of inputs $x_1, x_2, \ldots, x_k$, give a single output y, and have access to a fixed number of temporary variables $z_1, z_2, \ldots, z_n$. Temporary variables and output are initially zero.
- All input, ouput, and temporary variables are natural numbers.
- The following two operations are valid on any variables:

1. inc(v), which increments the variable v.
2. dec(v), which decrements the variable v, except at zero, whence $\text{dec}(v)_{v=0} = 0$.

- The description of a program is a sequence of instructions, which is either an operation on a variable, a return statement (which halts execution), or a conditional branch of the following form:

  If $v \neq 0$, then go to instruction n. Else, continue.

This turns out to be all we need.

Suppose we want to write a program that copies one variable to another. We would also like the input data to be unaltered at the end of the program. Now is a good time to pause and think about how you would do this.

Given that all way can do is increment and decrement, we will have to modify the input data as we copy it. The way we ensure it is unaltered at the end of the program is by also storing its value into a temporary variable, and then copying it back. Observe,

```
1  copy(x, y, z) {
2      if (x != 0)
3          goto 5
4      return
5      dec(x)
6      inc(y)
7      inc(z)
8      if (x != 0)
9          goto 5
10     dec(z)
11     inc(x)
12     if (z != 0)
13         goto 10
14     return
15 }
```

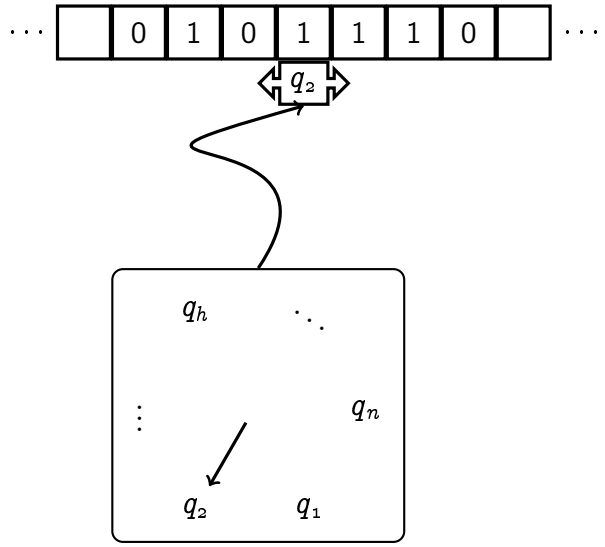
Visualizing the computation is a lot like debugging with `GDB`. We can view the execution of `copy(10, y, z)`, by listing each instruction with the state of memory on the right

```
1     copy(10, y, z)      ||    (x, y, z) = (10, 0, 0)
2     if (x != 0)         ||          10 != 0 is true
3         goto 5
4     dec(x)              ||    (x, y, z) =  (9, 0, 0)
5     inc(y)              ||    (x, y, z) =  (9, 1, 0)
6     inc(z)              ||    (x, y, z) =  (9, 1, 1)
7     if (x != 0)         ||           9 != 0 is true
8         goto 5          ||
9     ...
10    dec(x)              ||    (x, y, z) = (1, 9, 9)
11    inc(y)              ||    (x, y, z) = (0, 10, 9)
```

```
12    inc(z)              ||   (x, y, z) = (0, 10, 10)
13    if (x != 0)         ||          0 != 0 is false
14    dec(z)              ||   (x, y, z) = (0, 10, 9)
15    inc(x)              ||   (x, y, z) = (1, 10, 9)
16    if (z != 0)         ||          9 != 0 is true
17        goto 10         ||
18    ...
19    dec(z)              ||   (x, y, z) = (9, 10, 0)
20    inc(x)              ||   (x, y, z) = (10, 10, 0)
21    if (z != 0)         ||          0 != 0 is false
22    return
```

2.2 Turing Machines

We are ready to study our second model of computation: the Turing Machine (TM).



We assume the reader has encountered TM's before, and has studied its use as a language recognizer. We will use the following notation to define TM's:

Definition 2.1. A **Turing Machine** (TM) is defined by the five tuple $(Q, q_0, q_h, \Sigma, \Gamma, \delta)$, where Q is a finite set of states, $q_0 \in Q$ is the *initial state*, q_h is the *halt state*, Σ is the input alphabet, Γ is the tape alphabet, and $\delta : Q \times \Gamma \to Q \times \Gamma \times DIR$ is the transition function, where $\delta(q, \sigma) = (q', \sigma', D)$ means that reading the character σ while at state q causes the machine to replace σ with σ' and transition to state q', and move the reading head one step in the direction D.

DIR is the set $\{LEFT, RIGHT\}$.

The machine halts execution if it reaches state q_h.

Alan Turing, the inventor of the TM, writes in his seminal paper of 1936 ,

> We may compare a man in the process of computing ... to a machine which is only capable of a finite number of conditions, $q_1, ..., q_n$, which will be called m-configurations. The machine is supplied with a "tape" (the analogue of paper) [...] divided into sections (called "squares") each capable of bearing a "symbol".

Since their behavior at any point is completely determined by the current state and current symbol being read, Turing called them automatic machines.

2.3 μ-Recursive Functions

2.4 Church-Turing Thesis