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

CMPS132

Computability and Complexity Theory

Seshadhri Comandur • (Spring) 2015 • UC Santa Cruz

Last Revision: April 16, 2015

Table of Contents

Abstract

TODO

1 Introduction

1.1 What is Computation?

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 wait!" you may be thinking. "What is computation, anyway?"

Ian Horswill, a computer scientist at Northwestern, taught an introductory programming course and wrote a small set of notes to introduce students to computation. The unpublished notes are titled *What is Computation?*, and in them he gives the following first approximation toward defining computation:

> "While we still don't know what computation is in the abstract, we can at least say something about what we mean by a computational problem. It's a group of related questions, each of which has a corresponding desired answer. We'll call the information in the specific question the **input** value(s) and the desired answer the **output** value. For any input, there is a corresponding desired output. The notions of input and output are the most basic concepts of computation: computation is - for the moment - the process of deriving the desired output from a given input(s)."

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.

It turns out that the requirement that computer's description be finite immediately leads to the following theorem:

Theorem 1.1. There exist problems unsolvable by computation.

Before proving this result, it is worth looking at the origin of the question, "What are the limits of 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.

However, it is not hard to show that there exist unprovable problems.

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

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

1.3 Computing Functions

Consider the following set of functions

$$\mathcal{F} = \{f : \mathbb{N} \to \mathbb{N}\}$$

In English, $\mathcal{F}$ is the set of all functions between natural numbers $\mathbb{N} = \{0, 1, \dots\}$. 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.

For example, consider the function $f(n) = n^2$, and consider 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 * n);
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    4
8  $ ./f 1 1 1
9    9
```

So our program `f.c` can compute $f(n) = n^2$ 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 $g(n) = \lfloor n/2 \rfloor$, $h(n) = \min(100 - n, 0)$, and $p(n) =$ the n^{th} 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 exist 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 a number. 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	1	2	...
$f_1(n)$	1	4	9	...
$f_2(n)$	111	112	113	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Now, consider the following function:

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

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

2 Computational Models

2.1 "C" Programs

2.2 Turing Machines

2.3 μ-Recursive Functions

2.4 Church-Turing Thesis

[3] If you are worried about printing out non-numbers, we can change our model (or our `C` compiler) to only allow `printf("%d", n)` print statements.