1 Diagonalization and the Halting Problem (Draft)

Decision Problems. Suppose you are given some computational problem and want to write a C++ program for solving it. Is it always possible?

First, some clarifications. Examples of computational problems that we are interested in are the minimum spanning tree, the traveling salesman problem, the hamiltonian cycle problem, etc. In general, we can define a computational problem as the problem of computing some function \( f \). For example, in the minimum spanning tree problem, we want to compute a function \( f(G, w) = T \), that for a given graph \( G \) and weight function \( w \) produces a spanning tree \( T \) of \( G \) with minimum weight. Different functions may have different sets of arguments (graphs, numbers, strings, etc). However, we will usually assume that arguments are binary strings. This is not a real restriction, since any other types of arguments can be easily encoded as binary strings. As a matter of fact, we can even go one step further and assume that function arguments are nonnegative integers, because there is a 1 \leftrightarrow 1 \ correspondence between (finite) binary strings and non-negative integers.

We will mainly deal with boolean functions, whose values are 0 or 1 (true or false). The problems represented by such functions are called decision problems. The hamiltonian cycle problem is one example. Here, \( f(G) = 1 \) if \( G \) has a hamiltonian cycle and \( f(G) = 0 \) otherwise.

Next, we need to clarify what do we mean by “possible”. We are not interested (for now) in the cost, the running time, or the memory requirements. What we want to know is whether for any problem \( P \) there exists a C++ program that solves it.

Most people would answer “yes” without hesitation. It seems that if we only hire enough talented programmers and give them unlimited resources, there is no reason why eventually they would not produce correct code. However, it turns out that this is not true and the proof, as we shall see soon, is quite simple.

There are more (in fact, a lot more) functions than C++ programs. Therefore, there must exist a function that cannot be computed with a C++ program.

To prove this statement, we need to formalize what “more” means. There are infinitely many problems and infinitely many programs. So how can we say that one infinity is larger than the other?

Countable sets. Given a set \( X \), we say that \( X \) is countable if the elements of \( X \) can be ordered in a sequence \( X = \{x_1, x_2, x_3, \ldots \} \). Of course, every finite set is countable. The set of natural numbers is \( 1, 2, 3, \ldots \), so it is countable. The set of all integers is countable too, since we can order them \( 0, 1, -1, 2, -2, 3, -3, \ldots \). The set of pairs of natural numbers is countable, since we can order them \((1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3), (4, 1), (4, 2), \ldots \)

The set of all binary strings is countable too, because each binary string \( x \) can be mapped into the integer whose binary representation is \( 1x \). In this mapping, different strings are mapped to different integers. What about C++ programs? A C++ program is just a string of ASCII symbols, and by representing each ASCII character by its binary form, we can view a program as a binary string. So the set of all C++ programs is countable.

So are all sets countable? Nope. It turns out that some sets are “so large” that they cannot be listed in an order.

Cantor’s Theorem. Let \( X \) be the set of infinite binary strings. Given a sequence \( x \), by \( x^i \) denote the \( i \)th bit of \( x \). We want to prove that \( X \) is not countable.

Pick any infinite sequence \( x = (x_1, x_2, \ldots) \) of elements of \( X \). We now construct a string \( y \in X \) that is not listed in \( x \). For each \( i \), let \( y^i = x_i^i \) (negation of \( x_i^i \)). Intuitively, this can be explained as follows: Think of \( x \) as an infinite two-dimensional array, where the element in the row \( i \) and column \( j \) is \( x_{i,j}^i \). Then \( y \) is obtained by
selecting and flipping the elements on its diagonal.

\[
\begin{array}{cccccc}
  x_1 & : & 0 & 1 & 1 & 1 & 0 & \ldots & 1 \\
  x_2 & : & 1 & 1 & 0 & 1 & 0 & \ldots & 0 \\
  x_3 & : & 1 & 0 & 1 & 0 & 1 & \ldots & 0 \\
  x_4 & : & 1 & 0 & 0 & 0 & 0 & \ldots & 1 \\
  x_5 & : & 1 & 0 & 0 & 0 & 0 & \ldots & 1 \\
\end{array}
\]

⇒

Clearly, \( y \) cannot be in \( x \) since, for each \( i \), \( y \) differs from \( x_i \) in the \( i \)th position.

So let's summarize what we have accomplished. For each sequence \( x \) of elements in \( X \) we constructed \( y \in X \) that does not appear in \( x \). This means that there is no sequence that lists all elements of \( X \), so \( X \) is not countable. Ta da!

What we have just proved is called Cantor's Theorem. Actually, Cantor’s theorem says that there are uncountably many real numbers in the interval \([0, 1)\), but that’s essentially the same statement, because we can identify each real number number in \([0, 1)\) with its infinite binary representation (with a minor caveat – guess what is it?).

**Non-computable functions.** We are almost done now. It remains to show that the number of decision problems is not countable. Recall that we can identify each decision problem \( P \) with a function \( f \) from natural numbers into \([0, 1)\). But the difference between such functions and infinite binary strings is only in notation: \( f(i) \) instead of \( f_i \), so they are essentially the same thing. We conclude that there are non-countably many boolean functions on natural numbers, and therefore – since we showed earlier that the number of C++ programs is countable – there exists a function \( f \) that cannot be computed with a C++ program. Or, equivalently, there is a decision problem that cannot be solved using a C++ program. As a matter of fact, the argument above shows that most decision problems don’t have corresponding programs.

**More diagonalization.** So we know that most functions are not computable. But can we actually show a specific and natural function that is not computable? Consider the following problem.

**Halting Problem:** Given a C++ program \( P \) and a string \( x \), does \( P \) halt when we enter \( x \) as input?

Towards contradiction, suppose that there is a program \( H \) that solves this problem. Using \( H \) as a procedure, we could write another program \( R \) that works like this. An input to \( R \) is a program, say \( P \). After reading \( P \), \( R \) uses \( H \) to decide whether \( P \) halts on \( P \) (again, we treat \( P \) both as a program and a binary string). If so, \( R \) loops forever. If not, \( R \) halts. So \( R \) has the following property: \( R \) halts on \( P \) iff \( P \) loops on \( P \).

We are now almost done. What happens if we feed \( R \) (as a string) as input into program \( R \)? The above equivalence then becomes: \( R \) halts on \( R \) iff \( R \) loops on \( R \), so we get a contradiction. Therefore our assumption about the existence of \( H \) must have been incorrect, and we conclude that there is no program that solves the Halting Problem.