Introduction to Computability

What is computability?

Up until this point, the course has focused on tractability: whether problems can be solved by a computer algorithm in a reasonable amount of time with respect to the size of the problem. Here, we consider the broader question of computability, or what can be achieved by a computer at all.

Computability is a property of a computational problem. More specifically, a problem is computable if there exists an algorithm that solves the problem that can be performed by a computer with unlimited memory in finite time. We do not further specify what the upper bound of “finite time” is; it could require exponentially many steps or arbitrarily large amounts of memory to run an algorithm that solves this problem, but if it is computable, we know that the algorithm will conclude in a finite amount of time. Even with this unrestricted a bound on what is required to be computable, however, there still exist problems that are uncomputable, i.e. for which we can prove that no such algorithm exists to solve them.

Java as a model of computation

In order to prove things about the computability — or, more often, the uncomputability — of problems, we need some model of what a computer can do. You may in the past have encountered simple models for processing input data such as deterministic finite automata (DFAs) as seen in CS 2800. These are useful for solving specific types of problems, but they lack memory: a finite state machine only knows its current state, and in looking at the next character in an input string, it cannot reference any prior information about the string or its previous states. A full model of computation should be able to use unlimited memory, and should be capable of making decisions about what steps of computation it takes based upon any region of memory.

The most famous model used for proofs like this is called a Turing machine, a simplistic model created in 1936 by Alan Turing. These machines are effectively finite state machines with memory: they can only read and write one character at a time, but they have infinite space to read and write characters. They also have a finite set of states, much like finite state machines, but they can transition between these states based on reading characters written in memory.
We will discuss Turing machines later in the course.

In this course, we have used Java to write code for homework assignments, so perhaps the most natural definition of computability for the course is anything that can be computed using Java. We will assume that our Java program has access to an infinite memory. To keep things simple, we will be using a simplified form of Java as our model of computation. We limit ourselves to a small portion of what you might typically use in your programs. The parts of Java that we need for computation are sequences of statements, conditionals, and loops, all of which are supported with no additional libraries, though we can assume that any code from a library may simply be added to that program. We therefore only consider a few pieces of the Java specification:

- primitive data types `int`, `char`, and `boolean`, and basic operations for them;
- control flow in the form of `if...else` statements and `for` and `while` loops;
- the `String` class;
- primitive Java arrays, e.g. `String[]`; and
- the ability to create named methods callable from other methods.

Traditionally, Java methods can input and output any type of primitive or `Object`, while Java `main` functions must take in an array of `Strings` and have `void`, or nothing, as their return value. This is not to say that Java programs do not have any kind of output at all, but the output it produces is based on writing text to a buffer representing a file on a computer or text on the screen. We would like to be able to use Java programs to solve decision problems, such as the problems we discussed so far in the course. We can always think of the input as a string, but we need to find some way of retaining the true/false result of an entire program instead of a single internal method. To do this, we will assume that each program we write contains a `buffer` member variable that we can access to see what output was written by the simulated program. The primary functionality of a program will be stored in the function `execute`, which will take a list of `String` arguments, while the `main` function will be used to ensure that `execute` is used to write to the buffer. For decision problems, we can assume the only possible String values after a program has run are `true` and `false`. Before a program terminates, `buffer` will be equal to the empty string, ε or “”. We make this variable public so that a program can use the output of this to execute another program, and then inspect the buffer afterwards.
Useful properties of computational models. For the proofs we will write, there are a few key properties of Java that are useful to observe.

1. Source code can be passed as an argument in Java code. Because a Java program can be represented by a string of the source code, we are able to consider programs themselves as objects in our code. This means we can write a method in a Java class that takes as input a String intended to represent another Java program. Note that, as a String, there is no inherent power of the code to perform computation on its own; it has to be passed to some program that can interpret that code as a set of instructions and run those instructions.

2. Java can simulate itself. Not only is it possible to pass around Java programs inside Java, it is also possible to write a Java program that can execute an arbitrary Java program so long as the code is valid Java. This is a property called universality: because Java can simulate any possible computation, and the execution of a Java program is a possible computation, Java can simulate itself.

While a programming language simulating itself might seem a little confusingly recursive, it is common practice in code interpreters and compilers. The first compiler for Java could not be written in Java, as nothing would exist to compile it; however, once a Java compiler exists, it is completely possible to write and execute a new Java compiler in Java, using the old Java compiler to execute it. This may also be convenient for developers working on such a compiler, as they need only be proficient in Java to continue developing for it.

Though it is completely possible to write a Java interpreter in Java, doing so requires a significant amount of code and effort that is beyond the scope of this course. For our proofs, we will assume this work has already been performed. We will posit the existence a class `UniversalJavaProgram` whose `execute` method takes as input a string containing the Java program to be executed and a list of strings containing that program’s arguments, returning a string output. We show the source code of this program in Figure 1.

What about errors? One question we have not resolved is we validate inputs to our programs: how do we ensure that the Strings passed to some program can be used as inputs by the execute function? In general, we assume that we can pass any String or array of Strings to the main function of this program: it is the responsibility of the program to validate the input; if it fails to validate, it should simply output `false` to its buffer. We assume the same thing occurs if an error is thrown: if, for instance, the `UniversalJavaProgram` is given uncompileable code, it will find this and simply write `false` to

---

1 In fact, this has already been done for Java; see Espresso at [http://types.bu.edu/Espresso/report/Espresso.html](http://types.bu.edu/Espresso/report/Espresso.html). More popular examples of writing language compilers and interpreters in their own language include Clang for C++ ([https://clang.llvm.org](https://clang.llvm.org)) and PyPy for Python ([https://pypy.org](https://pypy.org)).
public class UniversalJavaProgram
{
    public String buffer;

    public UniversalJavaProgram()
    {
        buffer = ""; // Where we write output
    }

    public String execute(
            String programCode,
            String[] argumentArray)
    {
        // Simulate the execution of the Java program
        // 'programCode' when given the arguments
        // 'argumentArray' as input, and return the output
        // (a String).
    }

    public static void main(String[] args)
    {
        // Validate the number of arguments
        if (args.length !== 2) {
            buffer = "false";
            return;
        }

        // Execute the intended program
        String programCode = args[0];
        String[] argumentArray = new String[args.length - 1];
        for (int i = 1; i < args.length; i++) {
            argumentArray[i - 1] = args[i];
        }
        UniversalJavaProgram ujp = new UniversalJavaProgram();
        ujp.buffer = ujp.execute(programCode, argumentArray);
    }
}
the buffer. In this sense, we assume that UniversalJavaProgram is a perfect interpreter of Java code, and has no bugs: if the code itself inputted is buggy or incorrect, it will handle the error and return false.

3 Formalizing uncomputability

Now that we have a model of computation, we need to formalize what it means for something to be computable under a model of computation. To simplify how we consider computation, we are going to describe all of our problems as decision problems, or problems with a true or false answer. While this might seem like a restriction, this does not affect what we can or cannot compute, as we can use decision problems to reconstruct more complex solutions bit by bit.

Suppose we execute some valid3 Java program M for a decision problem with input x. There are three possible outcomes:

1. accept: the program terminates and returns a true or “yes” answer,
2. reject: the program terminates and returns a false or “no” answer, or
3. the program never terminates; i.e., it reaches an infinite loop.

If the program M reaches outcome 1 or 2 for input x, where the program terminates, we say that M halts on input x. We define the set \( \mathcal{L}_M \) as the collection of all x accepted by M, i.e., such that \( M(x) \) terminates and returns true.4 We can think of \( \mathcal{L}_M \) as specifying a decision problem solved by M: inputs where M returns true are inputs where the answer to the decision problem is “true” or “yes,” which belong in \( \mathcal{L}_M \), and inputs where M returns false or doesn’t return at all are inputs where the answer is “false” or “no,” which do not belong in \( \mathcal{L}_M \).

There are two definitions of interest to us, then, with respect to these outcomes:

- **recognizability**: If the program M accepts all and only inputs from \( \mathcal{L}_M \)—meaning outcome 1 is reached by M for input x if and only if \( x \in \mathcal{L}_M \)—we say that M recognizes \( \mathcal{L}_M \), or that \( \mathcal{L}_M \) is a problem recognized by M. We can also consider a language of strings \( \mathcal{L} \) independent of a specific program, such as “the set of all binary strings with an even number of 0s” or “the set of all descriptions of a graph for which there exists a Hamiltonian cycle.” We describe a language \( \mathcal{L} \) as being recognizable if there exists any program M for which \( M(x) \) returns true if and only if \( x \) is in \( \mathcal{L} \). If \( x \) is not in \( \mathcal{L} \) for some decision program M, M may either return false (outcome 2) or never terminate (outcome 3).

3 If M itself does not compile, we treat it as rejecting all input.

4 The set \( \mathcal{L}_M \) of all input strings accepted by M is often referred to as the language of M.
• **decidability**: We say that \( M \) **decides** \( L_M \) if \( M \) not only returns `true` for all \( x \) in \( L_M \), but also returns `false` in finite time for all \( x \) not in \( L_M \). In other words, \( M \) must make the correct decision about whether \( x \) is in \( L_M \) in finite time; it can never achieve outcome 3 above. Just as a language is **recognizable** if there exists a program that recognizes it, a language is **decidable** if there exists a program that decides it. Note that decidability is a strictly stronger requirement than recognizability; any program \( M \) that decides a language also recognizes it, and any language \( L \) that is decidable must also be recognizable.

We have encountered a wide variety of decidable problems so far in this course. For instance, SAT (the Boolean satisfiability problem) is NP-complete, but it is still decidable. We can show this by construction: we can write a program that, given a formula, methodically iterates through every single possible assignment of variables to `true` or `false` to see if any assignment satisfies the given formula. If the program finds a satisfying assignment, it outputs `true`; otherwise, after iterating through all possible assignments, it outputs `false`. In practice, we would never want to run this program for large \( n \), as for \( n \) different variables, it would take exponential time, \( O(2^n) \), before it would output `false` for an unsatisfiable formula. However, exponential time is still finite, so this program still decides SAT.

When we prove a problem to be uncomputable, we specify one of the definitions above which the problem does not satisfy. For example, we might want to prove that a problem described by some language \( L \) is **undecidable**, i.e., that there exists no program \( M \) that can for any input \( x \) determine whether \( x \) is in \( L \) in finite time. Much like in NP-completeness, for most undecidable problems, we prove that they are undecidable using reductions: if we can reduce a known undecidable problem to an unknown problem, then we can show that unknown problem is also undecidable. However, just like in NP-completeness, to use reductions, we need to bootstrap our set of undecidable problems by proving a single problem is undecidable without a reduction. For NP-completeness, that problem is SAT; for uncomputability, that problem is the halting problem, which we will get to in the next section.

## 4 Diagonalization and the halting problem

When first learning recursion in a programming class, you may have accidentally written code that infinitely recursed, producing a notorious “stack overflow” error. As your computer wound to a halt, you may have wished that there were some way to have known
This would happen before the code ran. Wouldn’t it be great if there were a program that, given the code for any program, \( M \), and some input to that program, \( x \), would tell you whether or not \( M \) would finish running on \( x \) in finite time?

This problem is called the halting problem, and we can write it out formally as a language \( L_{\text{HALT}} \):

\[
L_{\text{HALT}} = \{ \langle M, x \rangle | M \text{ is a Java program, and computing } M(x) \text{ halts.} \}
\]

We can first establish that this problem is recognizable: to prove it, we can simply imagine a program \( H(M, x) \) that simulates \( M \)’s execution on input \( x \) and, once the program halts, returns \text{true}. We can prove that this program \( H \) recognizes \( L_{\text{HALT}} \) by showing that it returns \text{true} if and only if \( M \) halts on input \( x \):

- if \( M \)’s execution halts on \( x \), then \( H \)’s simulation of \( M \)’s execution on \( x \) will also halt, and \( H \) will subsequently return \text{true};
- if \( H \) returned \text{true}, then \( H \) must have finished simulating \( M \) running on input \( x \) in finite time, implying that it halted.

Of course, recognizability does not really solve the problem we are interested in: the case that a student learning recursion would be interested in is not learning when a program will halt, but learning when it won’t. Unfortunately for them and for us, however, \( L_{\text{HALT}} \) is not decidable. To prove this, we will use a tool called diagonalization.

A quick tutorial on diagonalization

Diagonalization is a proof strategy that was first introduced by Georg Cantor in 1873 for describing relative magnitudes of infinite sets. The observation Cantor used was that, in order to show two different sets were of the same size, one could use a matching argument: if every single element of a set \( A \) is matched to exactly one element from set \( B \) and vice-versa, then the sets must have the same number of elements. This one-to-one matching, or bijection, can be extended to sets of infinite sizes: for infinite \( A \) and \( B \), if there is a bijection from \( A \) to \( B \), then \( A \) and \( B \) have the same cardinality of infinity.

Cantor used this to show that there were multiple cardinalities of infinity. For instance, one cardinality is represented by countability, where a set \( S \) is countable if there is a bijection from \( S \) to the natural numbers \( \mathbb{N} = 0, 1, 2, 3, \ldots \). Integers and all possible strings over a fixed set of characters are countable: you can write a function that enumerates every possible string such that, given a string, you could compute the finite index corresponding to that number.

However, real numbers are not countable. We construct a proof by contradiction of this by iterating through an imagined enumeration...
of real numbers and constructing an element that we show is not in the enumeration of those reals. Suppose we had some enumeration function real(n), where n is a positive natural number input and real(n) outputs the nth real number in the enumeration starting from n = 1. We will also write a helper function, digit(s, i), which returns the ith digit after the decimal place of a real number s. Construct a real number r as follows:

- r has no nonzero digits before the decimal point.
- r’s first digit after the decimal point is a digit in the range (1, 8) that is not the first digit after the decimal point for real(1); that is, digit(r, 1) ≠ digit(real(1), 1).
- r’s second digit is another digit in the range (1, 8) that is not equal to the second digit of the second real number, or digit(real(2), 2).
- We continue to generate digits infinitely, with the ith digit of r being some digit between 1 and 8 inclusive such that digit(r, i) ≠ digit(real(i), i).

Consider the real number r produced this way. We know from the way it is constructed that it cannot equal any enumerated real number produced by real, as for any output real(n), r will differ from it by at least the nth digit. We also know it is a real number, as it can be represented using a decimal representation, even if that representation is infinite. Thus, for any possible enumerative function real, we can construct a real number that it will never produce, implying that no such valid enumeration function could exist. This disproves the possibility of a bijection existing between the natural numbers and real numbers, and thus shows that the real numbers are not countable.

This argument relies on constructing r to differ explicitly from every single thing in our infinite list. The specific different element we use comes from the “diagonal” of the list: we differ from number i at the ith digit, and so on. We use this intuition to describe a style of proof argument called diagonalization, in which we argue that we cannot make a comprehensive list of some set of things (e.g.,

---

### Table 1: How we choose digits for assembling r based on a function real enumerating real numbers. Notice how we specifically differentiate each digit of r from a digit on the diagonal produced by aligning the real numbers—hence, diagonalization.

<table>
<thead>
<tr>
<th>number</th>
<th>digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>real(1)</td>
<td>0.212567...</td>
</tr>
<tr>
<td>real(2)</td>
<td>12.63367...</td>
</tr>
<tr>
<td>real(3)</td>
<td>1.50000...</td>
</tr>
<tr>
<td>real(4)</td>
<td>-5.981448...</td>
</tr>
<tr>
<td>r</td>
<td>0.3415...</td>
</tr>
</tbody>
</table>
the real numbers) by constructing an element that should belong in such a list, but would explicitly differ from every other element in that comprehensive list of things (e.g., \( r \)). We can use this same kind of construction in the case of the halting problem to construct an adversarial program, Diagonalizer, that makes it impossible for a hypothetical program that decides the halting problem, HaltChecker, to work. Next, we show the proof by contradiction that does this.

Proving the halting problem undecidable

Now we prove the primary result of this section: that the halting problem is undecidable.

**Theorem 1** (Undecidability of the Halting Problem). Consider the following decision problem that takes as input program \( M \) and input to the program \( x \):

\[
L_{\text{HALT}} = \{ \langle M, x \rangle | M \text{ is a Java program, and computing } M(x) \text{ halts.} \}
\]

This problem, called the halting problem, is undecidable.

**Proof.** We know if a language is decidable, then there exists some program that actually decides it. So, let’s assume we have access to that program, HaltChecker:

- **Arguments:** Our program takes in two arguments: program, the source code for a Java program, and input, the input that should be fed to the Java program.

- **Output:** For any possible program and input, HaltChecker will output in finite time whether the execution of program(input) will complete in finite time (either true or false).

Now, we will construct an additional program that uses HaltChecker as a subroutine. This program is going to force HaltChecker to contradict itself. Let’s call this program Diagonalizer. We show the outline of its source code in Figure 3.

The core method of Diagonalizer, execute, takes in only one string as an argument: the code of a program program. The method uses a HaltChecker to check if program halts when fed its own source code as input. If execute(program, input) returns true, meaning the program would halt, then we will make the execute method go into an infinite while loop. However, if execute(program, input) returns false, meaning the program would not halt, then we will tell our execute method to immediately return true and halt.

Just like how we constructed a real number different from any other possible real number in our list, Diagonalizer explicitly be-

\[\text{This might seem like a nonsense input for most programs, but if the code either fails to validate the input or throws some kind of error, we can still treat that as giving a false return value in finite time.}\]
public class HaltChecker
{
    public String execute(String program, String input) {
        // returns "true" if program would halt on input,
        // otherwise returns "false".
        // Note this cannot do this by calling execute on
        // program and input, as this might never terminate.
    }

    public static void main(String[] args)
    {
        String program = args[0];
        String input = args[1];
        HaltChecker hc = new HaltChecker();
        hc.buffer = hc.execute(program, input);
    }
}

Figure 2: Skeleton code for a HaltChecker.

haves differently from each other program $M$ on at least one input: the source code corresponding to that $M$. If $M$ halts on itself, the Diagonalizer will not halt; if $M$ does not halt on itself, Diagonalizer will halt. Now comes the weird part. What happens if we feed our halt-checking program two copies of the source code of Diagonalizer? Or, in code: if outsideHaltChecker is a HaltChecker and String diagonalizerCode is the code for Diagonalizer, we could define the boolean out as:

boolean out = outsideHaltChecker.execute(diagonalizerCode, diagonalizerCode);

After this line executes, what does the variable out equal?

• Suppose out is true. This implies that the Diagonalizer halts when fed its own source code. However, this only happens if haltChecker.execute returned false inside the Diagonalizer. This indicates that the two HaltCheckers, haltChecker and outsideHaltChecker, disagreed on the same inputs, two copies of diagonalizerCode: haltChecker thought that Diagonalizer would not halt, but outsideHaltChecker did. This is a contradiction, so outsideHaltChecker could not have returned true.

• Suppose out instead is false. This implies the opposite of before, that the Diagonalizer would not halt when fed its own source code. However, this infinite loop only happens if haltChecker.execute returned true. This again implies that the two HaltCheckers,
public class Diagonalizer
{
    public HaltChecker haltChecker;

    public Diagonalizer(String) {
        haltChecker = new HaltChecker();
        this.input = input;
    }

    public String execute(String program) {
        String wouldTerminate = haltChecker.execute(program, program);
        if (wouldTerminate == "true") {
            while (true) {
                continue;
            }
        } else {
            return "true";
        }
    }

    public static void main(String[] args)
    {
        String program = args[0];
        Diagonalizer diag = new Diagonalizer();
        diag.buffer = diag.execute(program);
    }
}

Figure 3: Skeleton code for our Diagonalizer.
haltChecker and outsideHaltChecker, disagreed on the same inputs: haltChecker thought that Diagonalizer would halt, but outsideHaltChecker did not. Because of this contradiction, outsideHaltChecker could not have returned false, either.

In effect, we have shown that there is no possible way for a hypothetical HaltChecker to give a correct answer about whether Diagonalizer would halt if fed its own source code. However, we said HaltChecker decided this problem: for it to do so, it must be able to return a correct answer for any program in finite time, even our adversarial Diagonalizer! So, no such HaltChecker can exist. And, if no program can exist to decide the halting problem $L_{\text{HALT}}$, then $L_{\text{HALT}}$ is undecidable. □

References


