Previously:
Introduce Turing machines

Today:
Finish discussion of Turing machines
Introduce decidability
Assignment 6 due

Enumerating Languages
So far, the Turing machines were recognizers

When a TM $E$ generates the words of a language, $E$ is an enumerator (cf. “recursively enumerable”)

A Turing machine $E$, enumerates the language $L$ if it prints an (infinite) list of strings on the tape such that all elements of $L$ will appear on the tape, and all strings on the tape are elements of $L$.

$E$ starts on an empty input tape. The strings can appear in any order; repetition is allowed.
Enumerating = Recognizing

A language $L$ is recursively enumerable if and only if some enumerator enumerates it

Proof (short version)
1. Show that if we have an enumerator $E$ that enumerates language $A$, a TM $M$ recognizes $A$
   - $M$: On input $w$
     - Run $E$ and check the strings it generates
     - If $w$ is produced, then accept
2. Now show if a TM $M$ recognizes $A$, we can construct enumerator $E$ for $A$
   - $E$: Ignore the input
     - Repeat the following for $i = 1, 2, 3, \ldots$
       - Let $s_1, s_2, \ldots$ be a listing of all strings $\in \Sigma^*$
       - Run $M$ on $s_1, \ldots, s_i$
       - If any computations accept, print out the corresponding $s$

A is recursively enumerable iff $A = L(E)$ for some enumerator $E$

Proof (need to prove both directions)
1. Convert $E$ to an ordinary recognizer $M$
   - Given $E$, we construct a TM $M$ with $E$ built inside it
   - Have $M$ leave the input string $w$ alone
   - $M$ moves to the blank portion of the tape and runs $E$
   - When $E$ decides to print something out, $M$ takes a look to see if the string is $w$
     - If not, then $M$ keeps simulating $E$
     - If the string is $w$, then $M$ accepts

If $M$ doesn’t find a match, it may go on forever. This is okay.

2. Convert $M$ to enumerator $E$
   - The idea is to feed all possible strings to $M$ in some reasonable order, for instance, lexicographic order $\varepsilon, 0, 1, 00, 01, 10, 11$
   - However, we have to be careful...
   - Suppose $M$ is running on 101
     - If $M$ accepts 101, then we print it out
     - If $M$ halts and rejects 101, then $E$ should move on to the next string
     - The only problem is when $M$ runs forever
     - What is $E$ supposed to do? $E$ doesn’t know $M$ is going forever!

- Can’t get hung up running $M$ on 101
  - We need to check the rest of the strings
- Solution
  - Run $M$ for a few steps on any given string, and if it hasn’t halted then move on, and come back to it later
  - Share time among all strings where computation hasn’t ended
  - Run more and more strings for longer and longer
  - More precisely
    - For $k = 1, 2, 3, \ldots$, $E$ runs $M$ on the first $k$ strings for $k$ steps
    - If $M$ ever accepts some string $s$, then print $s$
Other Computational Models

We can consider many other “reasonable” models of computation: DNA computing, neural networks, quantum computing...

Experience teaches us that every such model can be simulated by a Turing machine

Church–Turing Thesis

The intuitive notion of computing and algorithms is captured by the Turing machine model

A Brief History of Turing Machines

• Why are Turing machines so important, and why do we use them as a model for a general-purpose computer?
• The concept of a Turing machines dates back to the 1930s
  – One of a number of different models of computation that tried to capture *effective computability*, or algorithm, as we would now say
  – Other researchers came up with other models to capture computation, e.g., Alonzo Church developed lambda calculus
• *It wasn’t obvious that these different models are equivalent*, i.e., that they capture the same class of computations
  – However, they do!

Brief History, Continued

Nowadays we have programming languages
  – Can today’s more “advanced” programming languages (e.g., Python) do more than, say, a Fortran program?
  – They have a lot of new features compared to boring old do-loops
  – It’s conceivable that as we add more constructs to a programming language, it becomes more powerful, in the sense of computing functions and recognizing languages

Brief History, Continued

• **However, anything you can do with one language you can do with another**
  – It might be easier to program in one than another, or one might run faster
• How can we show we can do the same thing with Python as Fortran?
  – Convert Python programs into Fortran, or convert Fortran programs to Python
  – Simulate one language with the other
  – This “proves” that they have the same computational power
Brief History, Continued

- That’s what the researchers of computation theory did
- They gave ways to simulate Turing machines by \( \lambda \) calculus, \( \lambda \) calculus by Turing machines, as well as different variations of these models
- **They found that all these models were doing the same thing**
- Similarly, several variations of Turing machines all have the same computational power

Importance of the Church–Turing Thesis

The Church–Turing thesis marks the end of a long sequence of developments that concern the notions of “way-of-calculating”, “procedure”, “solving”, “algorithm”

Goes back to Euclid’s GCD algorithm (300 BC)

For a long time, this was an implicit notion that defied proper analysis

Aside: Euclidean algorithm

Find the GCD of two numbers by iteratively replacing the one of two numbers by the remainder from dividing it by the other.

*In Euclid’s original Python:*

```python
def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a % b)

print(gcd(3, 9)) # 3
print(gcd(9, 3)) # 3```

“Algorithm”

After Muhammad ibn Musa al-Khwarizmi (770–840)

His “Al-Khwarizmi on the Hindu Art of Reckoning” describes the decimal system (with zero), and gives methods for calculating square roots and other expressions

*Algebra* comes from the title of his book *Kitab al-jabr wa-l-muqabala (Book of Restoring and Balancing).* The term “al-jabr” is translated as “restoring”. Restoring, in this case, referred to the method of taking a subtracted quantity from one side and placing it to the other side of an equation
The notion of algorithm is a natural, robust notion.

This was a major step forward in our understanding of what computation is.

- It's almost saying something about the physical universe: there is nothing we can build in the physical world that is more powerful than a Turing machine.

Hilbert’s 10th Problem

In 1900, David Hilbert (1862–1943) proposed his Mathematical Problems; 23 of them.

Hilbert’s 10th problem is the determination of the solvability of a Diophantine equation.

- An indeterminate polynomial equation (i.e., one with more than one solution) where variables are integers only.
- Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients:
  - devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers.

Hilbert’s 10th Problem

- Suppose we want to solve a polynomial equation:
  \[ 3x^2 + 17x - 22 = 0 \]
  - Easily done.
  - But suppose we don’t want to know if a polynomial equation has a root, but whether it has a root where variables are integers.
  - Furthermore, we allow terms with several variables.
  - This makes things a lot harder: e.g., could have
    \[ 17xy^2 + 2x - 21z^2 + xy + 1 = 0 \]
- Is there an assignment of integers in \(x, y, z\) such that this equation is satisfied?

Hilbert’s 10th Problem

- Hilbert asked:
  - Is there a finite procedure which concludes after some finite number of steps, that tells us whether a given polynomial has an integer root?

- Put in the modern framework:
  - Let \(D = \{p \mid p\ \text{is a multivariable polynomial that has a solution (root) in integers}\}\).
  - Is \(D\) decidable?
Hilbert’s 10th Problem

- Without a precise notion of procedure, there was no hope of answering the question
  - Hilbert originally said, give a “finite procedure”
  - There was no notion that there might not be a procedure!
- It took 35 years before the problem could be addressed because we needed a formal notion of procedure to prove there is none
- Here, the Church–Turing Thesis played a fundamental role

(Un)solving Hilbert’s 10th

Hilbert’s “…a process according to which it can be determined by a finite number of operations…” needed to be defined in a proper way
This was done in 1936 by Church and Turing

The impossibility of such a process for exponential equations was shown by Davis, Putnam, and Robinson

Matiyasevich proved that Hilbert’s 10th problem is unsolvable in 1970

Terminology for TMs

- From now on we speak of TMs but real focus is on algorithms
  - TM is a precise model for the definition of algorithm
- Possible descriptions of algorithms/TMs:
  - Formal (as in previous slides): lowest level of detail
  - Implementation description: use prose to describe how the TM works (way it moves its head, stores data on tape), no details of states and transitions
  - High-level description: use prose to describe the algorithm, ignore implementation details

Format and Notation

- Input to TM is always a string
  - Must represent other objects as strings
    - Strings can represent anything!
  - TM may be programmed to decode the representation to so that it is interpreted as intended
  - Notation for the encoding of an object into its representation:
    - $<O>$
      - For several objects, $<O_1, O_2, \ldots, O_k>$
  - Example
    - Let $A = \{<G> | G$ is a connected undirected graph$\}$
    - Then $A$ is the language consisting of all strings representing undirected graphs
Procedures Versus Algorithms

There are two senses in which a TM accepts a language

1. The TM accepts the strings in the language (by final state), but does not halt on some of the strings not in the language
   - Thus, we can never be sure whether those strings are rejected, or eventually will be accepted
   - A language accepted in this way is called recursively enumerable (RE)
   - Such a language is called undecidable
   - The TM is sometimes referred to as a procedure

2. The TM accepts by final state, but halts on every string, whether or not it is accepted
   - A language accepted this way is called recursive
   - As a problem, the question is called decidable
   - The TM is called an algorithm

Intuitive Argument About an Undecidable Problem

- Problem: Given a Java program, does it print hello, world as the first 12 characters of output?
- Proof by contradiction:
  - We prove there is no program to (definitively) solve that problem by supposing that there is such a program $H$, the “hello world tester”
    - $H$ takes as input a Java program $P$ and an input file $I$ for that program, and tells whether $P$, with input $I$, prints hello, world (by which we mean it does so as the first 12 characters)
Modifying $H$

1. Modify $H$ to create a new program $H_1$ that acts like $H$, but when $H$ prints no, $H_1$ prints hello, world
   - we need to find where “no” is printed and change the print statement.
2. Modify $H_1$ to $H_2$. This program takes only one input, $P$, and acts like $H_1$ with both its program and data inputs equal to $P$ (the Java program).
   - i.e., $H_2(P) = H_1(P, P)$
   - $H_2$ must buffer its input so it can be used as both the $P$ and $I$ inputs to $H_1$

$H_2$ cannot exist!

If it did, what would $H_2(H_2)$ do?

- If $H_2(H_2) = \text{yes}$, then $H_2$ given $H_2$ as input evidently does not print hello, world
  - But $H_2(H_2) = H_1(H_2, H_2) = H(H_2, H_2)$, and...
  - $H_1$ prints yes if and only if its first input, given its second input as data, prints hello, world
  - So, $H_1(H_2, H_2) = \text{yes}$ implies $H_2(H_2) = \text{hello, world}$
  - But if $H_2(H_2) = \text{hello, world}$, then $H_1(H_2, H_2) = \text{hello, world}$ and...
  - $H(H_2, H_2) = \text{no}$.
  - Thus, $H_2(H_2) = \text{hello, world}$ implies $H(H_2, H_2) = \text{hello, world}$

But $H_2$ prints YES!
Hello, world

Does $P$ print “hello, world”?

Fundamental Questions

- **Universality.** What is a general purpose computer?
- **Computability.** Are there problems that no machine can solve?
- **Church–Turing thesis.** Are there limits on the power of machines that we can build?
- Pioneering work in the 1930s
  - Hilbert, Gödel, Turing, Church, von Neumann
  - Automata, languages, computability, universality, complexity, logic
Java: As Powerful as Turing Machine

- Turing machines are equivalent in power to Java
- Can use Java to solve any problem that can be solved with a TM
- Can use TM to solve any problem that can be solved with Java

Java simulator for Turing machines

```java
State state = start;
while (true) {
    char c = tape.readSymbol();
    tape.write(state.symbolToWrite(c));
    state = state.next(c);
    if (state.isLeft())
        tape.moveLeft();
    else if (state.isRight())
        tape.moveRight();
    else if (state.isHalt())
        break;
}
```

Universal Turing Machine

- Java program: solves one specific problem
- TM: solves one specific problem
- Java simulator in Java: Java program to simulate any Java program
- UTM: Turing machine that can simulate any Turing machine
- UTM is a “general purpose machine”.
  - Can be used to implement any algorithm
  - Anticipated development of first general purpose computer

- We will come back to this later

Church–Turing Thesis

*Church Turing thesis (1936). Turing machines can do anything that can be described by a physically harnessable process of the universe*

Implications:

- No need to seek more powerful machines
- If a yes–no problem can’t be solved by any Turing machine, then it can’t be solved on any physical computing device

Remarks

“Thesis” and not a mathematical theorem because it's a statement about the physical world and not subject to proof

Turing machine is a simple and universal model of computation
Implicit Physical Principles

- Turing machine: embodies physical constraints to which all concrete computational processes are subjected
- Axioms governing computational processes (partial list):
  - The speed of propagation of information is bounded
    - TM head only move to adjacent cells
  - The amount of information that can be encoded in the state of a finite system is bounded
    - TM stores finitely many symbols per tape cell
  - It is possible to construct … physical devices that perform in a recognizable and reliable way the logical functions AND, OR, and FAN-OUT
    - can fabricate a TM out of physical parts, and run it reliably