Theory of undecidability

For grading, we want to automatically test your code on sample inputs. But what if it doesn’t terminate? That will break the testing process.

(define halts? (lambda procdef input) …)

We don’t know of such a procedure that is guaranteed to give correct outputs. Fortunately, there is a reason for our ignorance. Turing and Church proved that there can be no such procedure.

Conveniently, a procedure definition in Lisp can be represented as a Lisp data object (a list), e.g.

(define foo
  (lambda (x)
    (if (prime? x) 1 (foo (* x x)))))

Want to know if procedure terminates when applied to particular inputs

(define halts? (lambda procdef input) …)

(halts? '(define foo (lambda …)) '(10 1))

We don’t know of such a procedure that is guaranteed to give correct outputs.

Fortunately, there is a reason for our ignorance. Turing and Church proved that there can be no such procedure.
Theory of undecidability

Assume for a contradiction that there was a correct implementation

\[
(\text{define } \text{halts? } (\lambda (\text{procdef} \ \text{input}) \ldots ))
\]

Let’s write another procedure…

\[
(\text{define } \text{trouble } \\
\quad (\lambda (\text{procdef}) \\
\quad \quad (\text{if } (\text{halts? } \text{procdef} \ \text{procdef}) \\
\quad \quad \quad (\text{mylength } \text{'}()) \\
\quad \quad \quad \#\text{false}))
\]

Procedure \text{trouble}:

- **input**: a procedure definition (represented as a list)
- **uses** \text{halt?} **to predict whether** that procedure would terminate if
  given as input the definition itself.
- **If** the program \textit{would} terminate then \text{trouble} **calls** \text{mylength},
  which never terminates.
- **If** the program \textit{would not} terminate, then \text{trouble} **just returns**
  \#false.
Now imagine we apply `trouble` to the list that gives the definition of `trouble`.

```
(define trouble
  (lambda (procdef)
    (if (halts? procdef procdef)
      (mylength '())
      #false))
```

**Procedure trouble:**
- **input**: a procedure definition (represented as a list)
- **uses halts? to predict whether that procedure would terminate if given as input the definition itself.**
- **If the program would terminate then trouble calls mylength, which never terminates.**
- **If the program would not terminate, then trouble just returns #false.**

Now imagine we apply `trouble` to the list that gives the definition of `trouble`.

```
(trouble '(define trouble (lambda (progdef) (if (halts? progdef progdef) (mylength '()) #false)))
```

Assume it halts. That means that the if-condition evaluated to #false.

The `halts?` procedure predicted that the `trouble` procedure does not halt when given itself as input. Assume it does not halt. That means the `halts?` procedure must have predicted that the `trouble` procedure halts when given itself as input.
Now imagine we apply `trouble` to the list that gives the definition of `trouble`.
```
(trouble '(define trouble (lambda (progdef) (if (halts? progdef progdef) (mylength '()) #false)))))
```
Assume it halts. That means that the if-condition evaluated to `#false`. The `halts?` procedure predicted that the `trouble` procedure does not halt when given itself as input. Assume it does not halt. That means the `halts?` procedure must have predicted that the `trouble` procedure halts when given itself as input.

This is a contradiction so our assumption of the existence of `halts?` must be false. The problem `halts?` purports to solve is an **undecidable** problem. Based on this, many other problems have been shown undecidable using **reductions**.
Halting Problem

Define DoesITHalt( program ):
{
    return true;
}

The big picture solution to the halting problem

https://xkcd.com/1266/
Example of hard computational problem: Subset Sum

**Subset sum:**

**input:** sequence of numbers \( w_1, w_2, \ldots, w_n \)

**output:** subset of input numbers summing to zero

How hard to solve?

We know:

- Enumeration: Algorithm that runs in time roughly \( O(2^n) \)
- Dynamic programming: Algorithm that runs in time roughly \( O(W) \), where \( W = w_1 + w_2 + \cdots + w_n \)

Say each number is written with \( b \) bits.
Then this algorithm takes time roughly \( O(2^b) \).

Can we escape exponential dependence on input size?
Is there an algorithm that runs in time \( O(n^3) \) or \( O(n^5) \) or even \( O(n^{10}) \)?

The theory of algorithms calls an algorithm *efficient* if the running time is \( O(n^d) \) for some constant \( d \) where \( n \) is the input size (measured, e.g., in bits).
That is, if the running time is bounded by a polynomial in \( n \).

We call it a *polynomial-time algorithm.*
Example of hard computational problem: Cryptography

Traditional (symmetric-key) cryptography:

Alice and Bob agree on a secret key $k$, say an $n$-bit sequence: 001011110101010…

Later, once Alice and Bob are separated, they can still communicate privately over an insecure channel:

Say Alice wants to send Bob a binary message $m$ (called the plaintext).

- Alice encrypts the message using the key $k$. This means applying an encryption procedure to the arguments $m, k$

  $(\text{define } \text{AES-encrypt}(m \ k) \ldots)$

Value $c$ returned is called the ciphertext.

- Alice sends the cipher text over the insecure channel.
- Bob receives the ciphertext and decrypts it using the key $k$. This means applying a decryption procedure to the arguments $c, k$

  $(\text{define } \text{AES-decrypt}(c \ k) \ldots)$

Value returned should be the original plaintext.
Some requirements for secure encryption:

**Usability:** decryption (with key) should take little time.

**Security against known-ciphertext attack:** Even if you know a plaintext and corresponding ciphertext, finding the key should take LOTS of time.

For practical cryptosystems, encryption takes linear time, and finding a key of size $n$ should take roughly $O(2^n)$.

AES (advanced encryption system) encrypts in linear time.

Can it withstand a known-ciphertext attack?

\[
\text{(define AES-encrypt(m k) ... )}
\]

**Known-ciphertext attack on AES:**

**Input:** plaintext $m$, cipher text $c$, key size $n$

**Output:** some key $k$ for which $m$ encrypts to $c$

Does the best known-ciphertext attack on AES take time $O(2^n)$?

Or is there an attack that takes time $O(n^3)$? $O(n^6)$?

\[
\text{(define AES-decrypt(c k) ... )}
\]
Hamiltonian Path

A Hamiltonian path in a graph is a path that visits every node exactly once.

Hamiltonian Path:
input: graph G
output: a Hamiltonian path in G

Best we know: finding a Hamiltonian path in an $n$-node graph takes roughly $O(2^n)$ time.

Is there an $O(n^3)$ algorithm? An $O(n^{40})$ algorithm?
Theory that addresses the hardness of these problems: *NP-completeness*
Your boss assigns you to come up with a fast algorithm for “determining whether or not any given set of specifications for a new bandersnatch component can be met and, if so, for constructing a design that meets them.”

“I can’t find an efficient algorithm, because no such algorithm is possible!”
Theory of NP-completeness: First turn problems into equivalent *decision problems*

A *decision problem* is a computational problem in which the output is Boolean.

**Subset sum (search):**

*input:* sequence of numbers $w_1, w_2, \ldots, w_n$

*output:* subset of input numbers summing to zero

**Subset sum (decision):**

*input:* sequence of numbers $w_1, w_2, \ldots, w_n$

*output:* Is there a subset of input numbers summing to zero?

The two versions are essentially equivalent: existence of an efficient algorithm for one implies existence of an efficient algorithm for the other.

**Hamiltonian Path (search):**

*input:* graph $G$

*output:* a Hamiltonian path in $G$

**Hamiltonian Path (search):**

*input:* graph $G$, start node $s$

*output:* Is there a Hamiltonian path in $G$ starting at $s$?
**Known-ciphertext attack on AES (search):**
**input:** plaintext $m$, cipher text $c$, key size $n$
**output:** some key $k$ for which $m$ encrypts to $c$

**Known-ciphertext attack on AES (decision):**
**input:** plaintext $m$, cipher text $c$, key size $n$, key prefix $w$
**output:** Is there a key $k$ with prefix $w$ for which $m$ encrypts to $c$?
Theory of NP-completeness: *Reductions*

A reduction from decision problem $A$ to decision problem $B$ is an efficient algorithm for the following:

**input**: an input for decision problem $A$
**output**: an input for decision problem $B$ such that correct outputs are the same (both *yes* or both *no*)

**Example**: a reduction from *chosen-ciphertext attack* to *Hamiltonian path* is an efficient algorithm for the following problem:

**input**: plaintext $m$, cipher text $c$, key size $n$, key prefix $w$
**output**: a graph $G$ and a starting node $s$

such that

there is an $n$-bit key whose prefix is $w$ that encrypts $m$ to $c$ if and only if

$G$ has a Hamiltonian path starting at $s$.

**Lemma**: There is such a reduction.

**Corollary**: If there is an efficient algorithm for Hamiltonian path then there is an efficient algorithm for chosen-ciphertext attach.
Theory of NP-completeness: Reductions

- chosen-ciphertext attack
- integer factorization
- graph isomorphism
- subgraph isomorphism
- subset sum
- partition of a graph into triangles
- Hamiltonian path
- 3-dimensional matching
- propositional satisfiability

AND HUNDREDS MORE!
Hundreds of problems have been defined that can all be reduced to each other. If we knew an efficient algorithm for any of them, we would know an efficient algorithm for every one of them.

What do they have in common?

• For each, there is an efficient algorithm for checking a solution, e.g.:

  **input:** graph \( G \), start node \( s \), candidate path \( P \)
  **output:** is \( P \) a Hamiltonian path for \( G \) that starts at \( s \)?

• The decision problem is of the form:
  Does there exist an \( x \) for which the check succeeds?

This characterizes a family of computational decision problems. It is called \( NP \).

An \( NP\)-complete problem is a decision problem in \( NP \) that every other problem in \( NP \) can be reduced to.
These are the hardest problems in \( NP \).
Many hundreds have been identified.
Is there a polynomial-time algorithm for an NP-complete problem? We don’t know. This is the question of whether P=NP.

Many people have tried very hard to prove or disprove P=NP. There is a million-dollar prize offered to the first person to resolve the question. Many crank solutions are submitted every year.

Our ignorance is somewhat embarrassing.