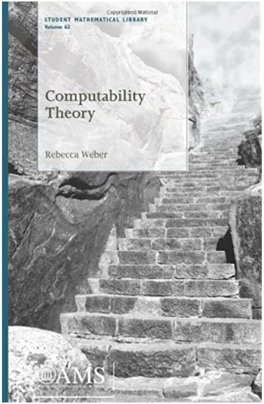
Computability Theory

Jose Emilio Alcantara Regio & Waseet Kazmi

University of Connecticut

December 9, 2020

Materials used



Computability Theory, Rebecca Weber

Overview

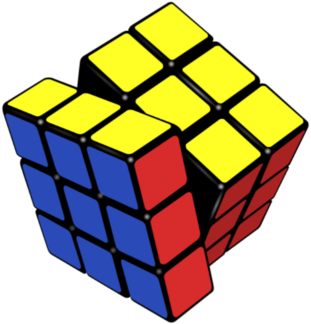
1. Introduction
2. Capturing Computability
3. Computable Functions
4. Computable and Computably Enumerable Sets
5. Turing Reductions
6. Turing Degrees

Introduction

Introduction

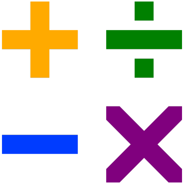
- What is Computability Theory?
- What does it mean to be *computable*?

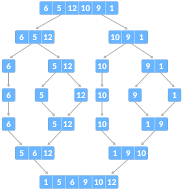
Examples of computability

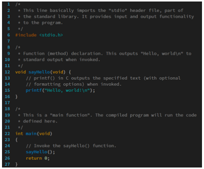


- There is an *algorithm*
- We can solve it in a finite amount of time using a finite number of *steps*

Examples of computability

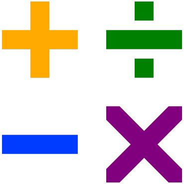


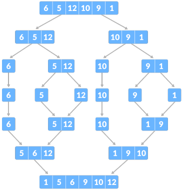


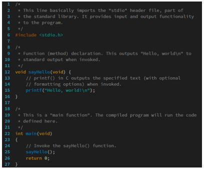


- Procedure
- Step-by-step

Examples of computability



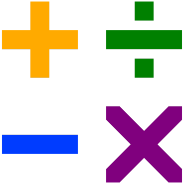


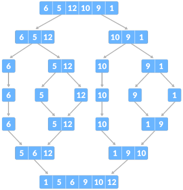


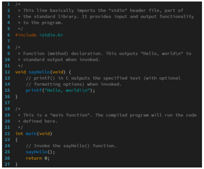
```c
/*
 * This line basically imports the "stdio" header file, part of
 * the standard library. It provides input and output functionality
 * to the program.
 */
#include <stdio.h>

/*
 * Function (method) declaration. This outputs "Hello, world\n" to
 * standard output when invoked.
 */
void sayHello(void) {
    // printf() in C outputs the specified text (with optional
    // formatting options) when invoked.
    printf("Hello, world!\n");
}

/*
 * This is a "main function". The compiled program will run the code
 * defined here.
 */
int main(void)
{
    // Invoke the sayHello() function.
    sayHello();
    return 0;
}
```

- “Procedure”?
- Step-by-step

Examples of computability







- “Procedure”?
- “Step-by-step”?

Defining Computability

- We need a **rigorous** definition for computability
- Must capture the intuitive understanding that we already have
- This was the goal of David Hilbert, Stephen Kleene, Alonzo Church, and Alan Turing
- Turing Machines were ultimately accepted as the satisfactory model for computation
- But why?

Capturing Computability

Preliminaries

Definition

A **partial function** is a function whose domain is a subset of $\mathbb{N} = \{0, 1, 2, \ldots\}$.

Ex: $f(x) = \frac{1}{x}$, $f(x) = \log(x)$

Definition

A **total function** is a function whose domain is the entirety of $\mathbb{N}$.

- Why do we need partiality for functions?

$\Rightarrow$ The function might not be defined on some inputs

$\Rightarrow$ Or, the computation of the function on an input might never stop

Preliminaries

Definition

If x is in the domain of f, then we say that the computation of f on x **halts** or **converges**, denoted by $f(x)\downarrow$.

Definition

If x is not in the domain of f, then we say that the computation of f on x **diverges**, denoted by $f(x)\uparrow$.

Preliminaries

Definition

The **characteristic function** of a set A is a *total* function defined as follows:

$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

Some attempts at defining computability

- Partial recursive functions
  - Stephen Kleene
  - Purely mathematical intuition
- Lambda calculus
  - Alonzo Church
  - Substitution
  - Used today in functional programming languages such as Haskell and Lisp
- Neither of these definitions were accepted as the satisfactory definition for computability

Turing Machine

- Alan Turing
  - Thought about what humans do when they solve problems
  - We read some symbols on a piece of paper, think, and then make a decision
- Turing Machine mimics this behavior
- Consists of a tape of infinite length and a tape head
  - Tape is divided into cells that contain a symbol
  - Tape head reads a symbol from a cell and then makes a decision to either write a new symbol onto the cell or move

Turing Machine

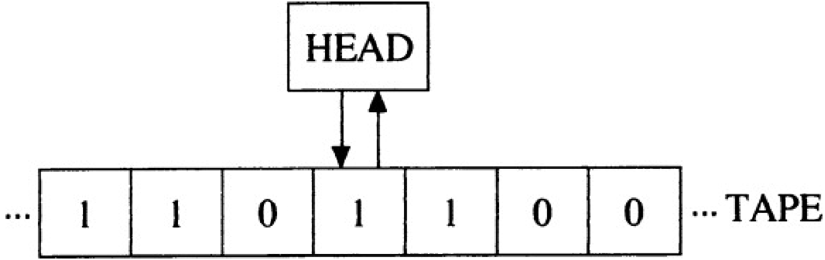


Figure: a visual representation of a Turing Machine.

Turing Machine

- Why were Turing Machines chosen as the model for computation?

$\Rightarrow$ Based on human behavior; intuitive to use

$\Rightarrow$ Mechanical aspect; visualize step-by-step process

Church-Turing Thesis

- Did we finally capture the full notion of computability?
  - We can never prove that we have done so
  - Requires an equivalence between a formal definition and an intuitive understanding
  - But, it turns out that partial recursive functions, Lambda functions, and Turing Machines are all equivalent!

Church-Turing Thesis

A function is computable iff it is Turing-computable, i.e., there is an equivalent Turing Machine.

Aside: Enumerating Turing Machines

- There is a computable bijection between the set of Turing Machines and $\mathbb{N}$
  - We can "translate" between Turing Machines and the natural numbers
  - Translation is done in a computable manner in both directions
  - The encoding of a Turing Machine is known as its **index**
- Notation: φ_e
  - Turing Machine with index e
  - Or, the eth Turing Machine

Computable Functions

Recursion Theorem

Recursion Theorem

Let f be a total computable function. Then there is an index n such that $\varphi_n = \varphi_{f(n)}$.

We will use the Recursion Theorem to prove Rice's Theorem.

Index Sets

Definition

Let $A \subseteq \mathbb{N}$. For any x and y, if we have that $x \in A$ and $\varphi_x = \varphi_y$ implies that $y \in A$, then A is an **index set**.

Index Sets

Examples:

- Fin $= \{e \mid \text{dom } \varphi_e < \infty\}$
  - Computable functions with finite domains
- Tot $= \{e \mid \text{dom } \varphi_e = \mathbb{N}\}$
  - Total computable functions

Basically "cherry-picking" computable functions based on what they do (semantic information)

Rice's Theorem shows us that we cannot do this "cherry-picking" in a computable manner

Rice's Theorem

Rice's Theorem

Suppose A is a nontrivial index set, i.e.,

$$\varnothing \subsetneq A \subsetneq \mathbb{N}.$$

Then χ_A is noncomputable.

Recall that for a set A, its characteristic function is defined as:

$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

Proving Rice's Theorem

- We will prove by contradiction.
- Suppose A is a nontrivial index set and that χ_A is computable.
- Since $\varnothing \subsetneq A \subsetneq \mathbb{N}$, then we can fix $a \in A$, $b \notin A$.
- Define f as follows:

$$f(x) = \begin{cases} a & \chi_A(x) = 0 \\ b & \chi_A(x) = 1 \end{cases}$$

Proving Rice's Theorem

- Since we assumed that χ_A is computable, then f is also computable.
- Additionally, since χ_A is a characteristic function, then it must be total. Thus, f is also total.
- Therefore, f is a total computable function.
- By the Recursion Theorem, there is some index n such that $\varphi_n = \varphi_{f(n)}$.

Proving Rice's Theorem

- We have two cases:
  1. If $n \in A$, then $f(n) = b \notin A$. But this contradicts our definition of an index set because if $n \in A$ and $\varphi_n = \varphi_b$, then we should have $b \in A$.
  2. If $n \notin A$, then $f(n) = a \in A$. Again, we have a contradiction of our definition of index sets because if $a \in A$ and $\varphi_a = \varphi_n$, then we should have $n \in A$.
- In both cases, we have a contradiction.
- Therefore, our assumption was incorrect and χ_A must be a noncomputable function. ■

Computable and Computably Enumerable Sets

Computable Sets

Definition

A set is **computable** if its characteristic function is computable.

Examples:

- $A = \{10, 15, 19, 5\}$
- $B = \{n \in \mathbb{N} \mid \exists k \in \mathbb{N} \text{ s.t. } n = 2k\} = \{0, 2, 4, 6, \ldots\}$

Computably Enumerable Sets

Definition

A set is **computably enumerable** if there is a computable procedure that outputs all the elements of the set, allowing repeats and does not have to respect an order.

Think of the procedure as an infinitely-printing printer, and the set as its receipt

Example: The Halting Set

- The set $K = \{e \mid \varphi_e(e) \downarrow\}$ is known as the *halting set*
  - The set of computable functions that halt on its index
- K is noncomputable
- However, K is computably enumerable
  - Step 1: Run one step of $\varphi_0(0)$
  - Step 2: Run another step of $\varphi_0(0)$, and then run two steps of $\varphi_1(1)$
  - Step 3: Run another step of $\varphi_0(0)$ and $\varphi_1(1)$, and then run three steps of $\varphi_2(2)$
  - Step i: Run i steps of $\varphi_0(0)$ to $\varphi_{i-1}(i-1)$
  - If any of the computations converge at any point, output the index

  $\Rightarrow$ Dovetailing

Computable vs Computably Enumerable Sets

- Difference is in the waiting time
- Computable Sets
  - We can know whether or not an element is in the set within a finite amount of time
- Computably Enumerable Sets
  - Keep waiting until the element is enumerated
  - If the element is in the set, then it is guaranteed that it will be enumerated at a certain point because the procedure enumerates all elements of the set
  - If the element is not in the set, then we are just waiting for something that will never come

Turing Reductions

Oracle Turing Machine

- Oracle Turing Machine
  - Turing Machine hooked up to a black box, known as the *oracle*
  - The oracle knows information about a particular set, say A
  - During computation, the Turing Machine can ask the oracle if a number is in A
- Notation: φ_e^A
  - eth Turing Machine with oracle A

Turing Reductions

Definition

Let $A, B \subseteq \mathbb{N}$. If there is an index e such that $\varphi_e^B = \chi_A$, then A is **Turing-reducible** to B, denoted by $A \leq_T B$

- We are using answers from χ_B to calculate the answer for χ_A
- In other words, if we know how to solve B, then we can solve A
- We are *reducing* the problem from A to B

Turing Degrees

Turing Equivalence

Definition

Let $A, B \subseteq \mathbb{N}$. If $A \leq_T B$ and $B \leq_T A$, then A and B are **Turing equivalent**, denoted by $A \equiv_T B$.

Turing Degrees

- Turing Equivalence is an equivalence relation
- Thus, you can take the quotient of $\mathcal{P}(\mathbb{N})$ by $\equiv_T$
    - i.e., partition sets of natural numbers by Turing equivalence
- Each equivalence class ("slice") is known as a **Turing degree**

Special thank-you to Waseet for his mentorship, and to Professor Katie for making this all happen!