1 What is Computability

From an informal or intuitive perspective what might we mean by computability? One natural interpretation is that something is computable if it can be calculated by a systematic procedure; we might think of this as a process that can be described by one person and carried out by another.

We will now assert that any such process can be written as a computer program. This is known as the Church-Turing thesis because they were the first to give formal mathematical specifications of what constituted an algorithm in form of the lambda calculus and Turing Machines, respectively. Note that the claim is a thesis; it is inherently not something provable, for the notion of a systematic process is unavoidably imprecise.

We remark that an instruction such as “guess the correct answer” does not seem to be systematic. An instruction such as “try all possible answers” is less clear cut: it depends on whether the possible answers are finite or infinite in number. For specificity, henceforth we will imagine working with a specific programming language, and using this as the definition of reasonable instructions (you can think of this as Java, C, or whatever is your preferred programming language). Actually, in practice we will describe algorithms in pseudo-code and English, but with enough detail that their programmability should be clear.

We define computability in terms of language recognition.

**Definition 1** \( L \subseteq \Sigma^* \) is computable if there is a program \( P \) with two possible outputs, “Recognize” and “Reject” and on input \( x \in \Sigma^* \), \( P \) outputs “Recognize” if \( x \in L \) and “Reject” if \( x \notin L \). We will also say that \( P \) decides \( L \) (in the sense of decides set membership).

**Remark.** Often, one wants to compute some function \( f(x) \) of the input \( x \). By allowing 2-input programs we can define computable functions \( f \) as follows: \( f \) is computable if there is a 2-input program \( P \), such that on input \((x,y)\), \( P \) outputs “Recognize” if \( f(x) = y \) and “Reject” if \( f(x) \neq y \).
2 Encodings

Often, we will want to treat the text of one program as a possible input to another program. To this end, we take the conventional approach that the program symbols are encoded in ASCII or more simply in binary. We do the same for the input alphabet. Of course, if we are not trying to treat a program as a possible input to another program, we can have distinct alphabets for programs and inputs.

As it turns out, not all languages are computable (indeed, we have already seen that the halting problem is not computable). First, however, we are going to see some computable languages.

The first languages we are going to look capture properties of automata. To this end, we need an agreed method for writing the description of automata. This is similar to a standard input format for a graph, for example. To describe a DFA we specify it as a sequence of 4 sets: Σ, the input alphabet, Q, the set of vertices, \( \text{start} \in Q \), the start vertex, \( F \subseteq Q \), the set of recognizing vertices, and \( \delta \), a description of the edges and their labels. If \( \Sigma = \{a_1, a_2, \ldots, a_k\} \), \( Q = \{q_1, \ldots, q_r\} \) we could write this as a string.

\[
\left( \{a_1, a_2, \ldots, a_k\}, \{q_1, q_2, \ldots, q_r\}, \text{start}, \{q_{i_1}, q_{i_2}, \ldots, q_{i_j}\},
\begin{align*}
\text{start}, \{(q_1, a_1) &\rightarrow q_{j_1}, (q_1, a_2) \rightarrow q_{j_2}, \ldots, (q_1, a_k) \rightarrow q_{j_k}, \\
(q_2, a_2) &\rightarrow q_{j_{k+1}}, \ldots, (q_r, a_k) \rightarrow q_{j_{kr}}\}\right),
\]

where \( F = \{q_{i_1}, q_{i_2}, \ldots, q_{i_j}\} \) and \( \delta(q_h, a_i) = q_{j(h-1)k+i} \) for \( 1 \leq h \leq r, 1 \leq i \leq k \).

There is a further difficulty that we would like to use one alphabet to describe many different machines. One way of doing this is to write the vertices and the alphabet characters in binary. We then see that the above DFA description uses just 8 distinct characters: \{, }, (,), 0, 1, \rightarrow, . We could reduce this to two characters, 0 and 1 say, by encoding each of these characters using three 0s and 1s.

We will view an input string is a descriptor of a DFA only if it has the above format, and in addition:

- \( a_i \neq a_j \) for \( i \neq j \), i.e. the relevant binary strings are not equal.
- \( q_i \neq q_j \) for \( i \neq j \).
- \( \text{start} = q_i \) for some \( i, 1 \leq i \leq r \).
- For each \( h, 1 \leq h \leq j \), \( q_{i_h} = q_l \) for some \( l, 1 \leq l \leq r \).
- In each expression \( (q_h, a_i) \rightarrow q_{j(h-1)k+i}, q_{j(h-1)k+i} = q_l \) for some \( l, 1 \leq l \leq r \).

As this is rather tedious, henceforth we leave the description of correct encodings to the reader (to imagine). We will simply use angle brackets to indicate a suitable
encoding in a convenient alphabet (binary say). So \( \langle M \rangle \) denotes an encoding of machine \( M \), \( \langle w \rangle \) an encoding of string \( w \) (note that \( w \) is over some alphabet \( \Sigma \), while \( \langle w \rangle \) may be in binary). We also use \( \langle M, w \rangle \) to indicate the encoding of the pair \( M \) and \( w \). But even this can prove somewhat elaborate, so sometimes we will simply write \( M \) when we mean \( \langle M \rangle \). As a rule, when \( M \) is the input to another program, then what we intend is \( \langle M \rangle \), a proper encoding of the description of \( M \).

We are now ready to look at the decidability of some languages.

### 3 Decidability of Regular Language Properties

**Example 2** \( \text{Rec-DFA} = \{ \langle M, w \rangle \mid M \text{ is a DFA and } M \text{ recognizes input } w \} \).

**Claim 3** \( \text{Rec-DFA} \) is decidable.

**Proof:** To prove the claim we simply need to give an algorithm \( \mathcal{A} \) to determine whether \( M \) recognizes its input. Such an algorithm is easily seen at a high level: \( \mathcal{A} \) first checks whether the input is legitimate and if not it rejects. By legitimate we mean that \( \langle M, w \rangle \) is the encoding of a DFA, followed by the encoding of a string \( w \) over the input alphabet for \( M \). If the input is legitimate \( \mathcal{A} \) continues by simulating \( M \) on input \( w \): \( \mathcal{A} \) keeps track of the vertex \( M \) has reached as \( M \) reads its input \( w \). When (in \( \mathcal{A} \)'s simulation) \( M \) has read all of \( w \), \( \mathcal{A} \) checks whether \( M \) has reached a recognizing vertex and outputs accordingly: \( \mathcal{A} \) outputs “Recognize” if \( M \) has reached a recognizing vertex, and \( \mathcal{A} \) outputs “Reject” otherwise.

\[ \mathcal{A}(\langle M, x \rangle) = \begin{cases} \text{“Recognize”} & \text{if } M \text{ recognizes } x \\ \text{“Reject”} & \text{if } M \text{ does not recognize } x \end{cases} \]

The details are a bit more painstaking: given the current vertex reached by \( M \) and the next character in \( w \), \( \mathcal{A} \) looks up the vertex reached by scanning the edge descriptions (the triples \( (p, a) \rightarrow q \)). In yet more detail, \( \mathcal{A} \) stores the current vertex in a variable, the input \( w \) in a linked list, a pointer to the next character of \( w \) to be read, and the DFA in adjacency list format.

The details of how to implement algorithm \( \mathcal{A} \) should be clear at this point. As they are not illuminating we are not going to spell them out further.

**Note.** Henceforth, a description at the level of detail of the first paragraph of the above proof will suffice. Further, we will take it as given that there is an initial step in our algorithms to check that the inputs are legitimate.

Let \( P_{\text{RecDFA}} \) be the program implementing the just described algorithm \( \mathcal{A} \) deciding the language of Example 2. So \( P_{\text{RecDFA}} \) takes inputs \( (M, x) \) (strictly, \( \langle M, x \rangle \)) and outputs “Recognize” if \( M \) recognizes \( x \) and outputs “Reject” if \( M \) does not recognize \( x \). This is a notation we will use repeatedly. If \( \text{Prop} \) is a decidable property (i.e. the language \( L = \{ w \mid \text{Prop}(w) \text{ is true} \} \) is decidable) then \( P_{\text{Prop}} \) will be a program that decides \( L \); we will also say that \( P_{\text{Prop}} \) decides \( \text{Prop} \).
Example 4 \( \text{Rec-NFA} = \{ \langle M, w \rangle \mid M \text{ is a description of an NFA and } M \text{ decides } w \} \).

Claim 5 \text{Rec-NFA is decidable.}

\textbf{Proof:} The algorithm \( A_{\text{Rec-NFA}} \) to decide \text{Rec-NFA} simulates \( M \) on input \( w \) by keeping track of all vertices reachable on reading the first \( i \) characters of its input, for \( i = 0, 1, 2, \cdots \) in turn. \( M \) recognizes \( w \) exactly if a recognizing vertex is reachable on reading all of \( w \), and consequently \( A_{\text{Rec-NFA}} \) outputs “Recognize” if it finds \( M \) can reach a recognizing vertex on input \( w \) and outputs “Reject” otherwise. 

Example 6 \( \text{Rec-RegExp} = \{ \langle r, w \rangle \mid r \text{ is a regular expression that generates } w \} \).

Claim 7 \text{Rec-RegExp is decidable.}

\textbf{Proof:} The algorithm \( A_{\text{Rec-RegExp}} \) to decide \text{Rec-RegExp} begins by building an NFA \( M_r \) recognizing the language \( L(r) \) described by \( r \), using the procedure from Finite Automata, Part 4, Lemmas 1–4. \( A_{\text{Rec-RegExp}} \) then forms the encoding \( \langle M_r, w \rangle \) and simulates the program \( P_{\text{Rec-NFA}} \) from Example 4 on input \( \langle M_r, w \rangle \). \( A_{\text{Rec-RegExp}} \)’s output (“Recognize” or “Reject”) is the same as the one given by \( P_{\text{Rec-NFA}} \) on input \( \langle M_r, w \rangle \).

This is correct for \( M_r \) recognizes \( w \) if and only if \( w \) is one of the strings described by \( r \).

This procedure is taking advantage of an already constructed program and using it as a subroutine. This is a powerful tool which we are going to be using repeatedly.

Example 8 \( \text{Empty-DFA} = \{ \langle M \rangle \mid M \text{ is a DFA and } L(M) = \emptyset \} \).

Note that \( L(M) = \emptyset \) exactly if no recognizing vertex of \( M \) is reachable from its start vertex. It is easy to give a graph search algorithm to test this.

Claim 9 \text{Empty-DFA is decidable.}

\textbf{Proof:} The algorithm \( A_{\text{Empty-DFA}} \) to test this property, given input \( \langle M \rangle \), determines the collection of vertices reachable from \( M \)’s start vertex. If this collection includes a recognizing vertex then the algorithm outputs “Recognize” and otherwise it outputs “Reject”. □

Example 10 \( \text{Equal-DFA} = \{ \langle M_A, M_B \rangle \mid M_A \text{ and } M_B \text{ are DFAs and } L(M_A) = L(M_B) \} \).

Claim 11 \text{Equal-DFA is decidable.}
**Proof:** Let \( A = L(M_A) \) and \( B = L(M_B) \). We begin by observing that there is a DFA \( \tilde{M}_{AB} \) such that \( L(\tilde{M}_{AB}) = \emptyset \) exactly if \( A = B \). For let \( C = (A \cap \overline{B}) \cup (\overline{A} \cap B) \). Clearly, if \( A = B \), \( C = \emptyset \). While if \( C = \emptyset \), \( A \cap \overline{B} = \emptyset \), so \( A \subseteq \overline{B} = B \); similarly \( \overline{A} \cap B = \emptyset \), so \( B \subseteq A \); together, these imply \( A = B \).

But given DFAs \( M_A \) and \( M_B \), we can construct DFAs \( \overline{M}_A \) and \( \overline{M}_B \) to recognize \( \overline{A} \) and \( \overline{B} \) respectively. Then using \( M_A \) and \( \overline{M}_B \) we can construct DFA \( M_{AB} \) to recognize \( A \cap \overline{B} \) and \( M_{\overline{AB}} \) to recognize \( \overline{A} \cap B \). Given \( M_{AB}, M_{\overline{AB}} \) we can construct \( \tilde{M}_{AB} \) to recognize \( (A \cap \overline{B}) \cup (\overline{A} \cap B) \).

So the algorithm \( A_{Equal-DFA} \) to decide \( Equal-DFA \), given input \( \langle M_A, M_B \rangle \), constructs \( \tilde{M}_{AB} \) and forms the encoding \( \langle \tilde{M}_{AB} \rangle \). \( A_{Equal-DFA} \) then simulates program \( P_{Empty-DFA} \) from the preceding example on input \( \langle \tilde{M}_{AB} \rangle \). \( A_{Equal-DFA} \) outputs the result of the simulation of \( P_{Empty-DFA} \) on input \( \langle \tilde{M}_{AB} \rangle \).

This is correct for \( P_{Empty-DFA} \) outputs “Recognize” exactly if \( L(\tilde{M}_{AB}) = \emptyset \) which is the case exactly if \( A = B \).

We have now given two examples of a use of a subroutine in a very particular form. More specifically, program \( P \) has used program \( Q \) as a subroutine and then used the answer of \( Q \) to compute its own output. In these two examples, the calculation of \( P \)'s output has been the simplest possible: the output of \( Q \) has become the output of \( P \).

We call this form of algorithm design a *reduction*. If we have a decision procedure for a language \( A \) (e.g. \( Empty-DFA \)) using a program (algorithm) \( Q \), and we give a program to decide language \( B \) (e.g. \( Equal-DFA \)) using \( Q \) as a subroutine then we say we have *reduced* language \( B \) to language \( A \). What this means is that if we know how to decide language \( A \) we have now also demonstrated how to decide language \( B \).

**Example 12** \( Inf-DFA = \{ \langle M \rangle \mid M \text{ is a DFA and } L(M) \text{ is infinite} \} \).

**Claim 13** \( Inf-DFA \) is decidable.

**Proof:** Note that \( L(M) \) is infinite exactly if there is a path which has a cycle from \( M \)'s start vertex to some final vertex. This property is readily tested by the following algorithm \( A_{Inf-DFA} \).

**Step 1.** \( A_{Inf-DFA} \) identifies the non-trivial strong components of \( M \)'s graph, that is those that contain at least one edge (so any vertices with a self-loop will be in a non-trivial strong component).

**Step 2.** \( A_{Inf-DFA} \) forms the reduced graph, in which every strong component is replaced by a single vertex, and in addition it marks each non-trivial strong component (or rather the corresponding vertices).

**Step 3.** \( A_{Inf-DFA} \) checks whether any path from the start vertex to a final vertex includes a marked vertex (and thus can contain a cycle drawn from the corresponding strong component).

**Step 3.1.** By means of any graph traversal procedure (e.g. DFS, BFS), \( A_{Inf-DFA} \)
determines which marked vertices are reachable from the start vertex. It doubly marks these vertices.

**step 3.2.** By means of a second traversal, $A_{\text{Inf-DFA}}$ determines whether any final vertices can be reached from the doubly marked vertices. If so, there is a path with a cycle from the start vertex to a final vertex, and then $A_{\text{Inf-DFA}}$ outputs “Recognize”; otherwise $A_{\text{Inf-DFA}}$ outputs “Reject.”

**Example 14** $\text{All-DFA} = \{\langle M \rangle \mid L(M) = \Sigma^* \text{ where } \Sigma \text{ is } M \text{’s input alphabet}\}.

$L(M) = \Sigma^*$ exactly if $\overline{L(M)} = \emptyset$. So to test if $L(M) = \Sigma^*$, the decision algorithm $A_{\text{All-DFA}}$ simply constructs the encoding $\langle \overline{M} \rangle$ and then uses the program $P_{\text{Empty-DFA}}$ to test if $L(\overline{M}) = \emptyset$, where $\overline{M}$ is the DFA recognizing $\overline{L}$, and then outputs the same answer, namely:

$$A_{\text{All-DFA}}(\langle M \rangle) = \begin{cases} 
\text{“Reject”} & \text{if } P_{\text{Empty-DFA}}(\langle \overline{M} \rangle) = \text{“Reject”} \\
\text{“Recognize”} & \text{if } P_{\text{Empty-DFA}}(\langle \overline{M} \rangle) = \text{“Recognize”}
\end{cases}$$

4 Decidability of Context Free Language Properties

**Example 15** $\text{Rec-CFG} = \{\langle G, w \rangle \mid G \text{ is a CFG which can generate } w\}.

**Claim 16** $\text{Rec-CFG}$ is decidable.

**Proof:** The following algorithm $A_{\text{Rec-CFG}}$ decides $\text{Rec-CFG}$.

**Step 1.** $A_{\text{Rec-CFG}}$ converts $G$ to a CNF grammar $\tilde{G}$, with start symbol $S$.

**Step 2.** If $w = \lambda$, $A_{\text{Rec-CFG}}$ checks if $S \rightarrow \lambda$ is a rule of $\tilde{G}$ and if so outputs “Recognize” and otherwise outputs “Reject.”

**Step 3.** If $w \neq \lambda$, the derivation of $w$ in $\tilde{G}$, if there is one, would take $2|w| - 1$ steps. $A_{\text{Rec-CFG}}$ simply generates, one by one, all possible derivations in $\tilde{G}$ of length $2|w| - 1$. If any of them yield $w$ then it outputs “Recognize” and otherwise it outputs “Reject.” (This in not intended to be an efficient algorithm.)

**Example 17** $\text{Empty-CFG} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}.

**Claim 18** $\text{Empty-CFG}$ is decidable.

**Proof:** Note that $L(G) \neq \emptyset$ if and only if $G$’s start variable can generate a string in $T^*$, where $T$ is $G$’s terminal alphabet. We simply determine this property for each variable $A$ in $G$: can $A$ generate a string in $T^*$? This can be done by means of the following algorithm $A_{\text{Empty-CFG}}$, which marks each such variable.

**Step 1.** $A_{\text{Empty-CFG}}$ converts the grammar to CNF form (this just simplifies the rest of the description).
Step 2. $A_{\text{Empty-CFG}}$ marks each variable $A$ for which there is a rule $A \rightarrow a$ or $A \rightarrow \lambda$ (the latter could apply only to $S$, the start variable).

Step 3. Iteratively, $A_{\text{Empty-CFG}}$ marks each variable $A$ such that there is a rule $A \rightarrow BC$ and $B$ and $C$ are already marked. (We leave an efficient implementation to the reader). $A_{\text{Empty-CFG}}$ stops when no more variables can be marked.

Step 4. $A_{\text{Empty-CFG}}$ outputs “Reject” if $S$ is marked and “Recognize” otherwise. ■