Understanding the Limits of Computation

Determining that no algorithm exists for a problem P

1. Understand the properties common to all algorithms.
2. Show that the problem P requires an algorithm that contradicts these properties.

Determining that no 'efficient/feasible' algorithm exists for a problem P

1. Identify one such problem P' that does not have a feasible algorithm
2. "Reduce" problem P' to problem P.

Slide 6

2 Turing Machines

Turing Machines were conceived by Alan Turing in order to model what humans do in order to solve a problem when following *instructions* through *symbolic means*. He tried to extract the basic entities of this process (*e.g.,* a unit that keeps the current 'state of mind' of the human, another unit permitting input from, and output to, the outside world *etc.*) together with the basic operations (*e.g.,* reading symbols, writing symbols and moving to different parts of the input) that allow a human 'computer' to carry out the process of following a recipe. Nowadays, Turing Machines serve as a convincing basis for a theory of algorithms and are also used as a basis for efficiency analysis of algorithms.

2.1 Introductory Concepts

Turing Machines (TMs) are basically extended versions of pushdown automata. As with pushdown automata, a TM has a central 'processing' part (modelling the aforementioned human 'mind-state'), which can be in one of a *finite* number of states, and an *infinite* tape used for storage. However, there are a number of important differences between Turing machines and pushdown automata:

- Turing machines receive their *input* written on the same tape which they also use for *storage*;
- Turing machines *control the head* position to where reading and writing on the tape is performed.

Slide 8 gives one possible depiction of a Turing Machine. It consists of three components :

Tape: It consists of a a denumerable set of squares, each of which can hold exactly one symbol. The tape acts as the *memory* of our machine.

Control: It consists of an enhanced form of finite state machine, and thus encompasses the processing part of the machine.

Head: It acts as the mediator between the tape and the control units. The control component can interact with the tape by reading from the current position of the head and also writing to the current position. The Control can also make the head move to the neighbouring positions of the tape.

Components of a Turing Machine

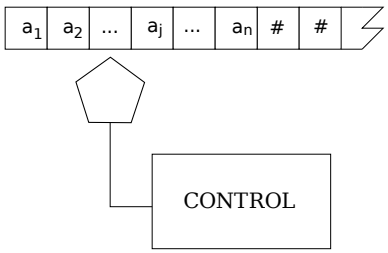
Tape a denumerable set of squares (locations) containing exactly 1 symbol.

Control an enhanced form of Finite State Machine.

Head the mediator between tape and control.

Slide 7

Depiction of a Turing Machine



Slide 8

How does it work? Depending on the *current state* the control is in, and depending on the *tape contents* at the location (square) where the head is, a Turing Machine then determines:

1. Its next state.
2. How to interact with the tape. In our version of Turing Machines, tape interactions can be either one the following:
   - write a symbol on the tape (and remain in the same position.)
   - move one location (square) to the right.
   - move one location (square) to the left.

When the Turing Machine decides to move the head left (or right), the tape contents remain unchanged. The tape is of infinite length on the right-hand side; it is however not infinite on the left-hand side. We can therefore visualise the locations on the tape as indexed, starting from 0 at the leftmost position and then increasing by 1 with every location to the right. There is one exceptional case that happens when a Turing Machine is currently at position 0 and tries to move the head one step to the left; when this happens, the Turing Machine stops executing and we say that the machine *hangs*.

A Turing Machine can potentially run forever when executing; we refer to this behaviour as *diverging*. We would ideally however have our Turing Machine stop. This typically happens when it reaches a state and a position for the head whereby it does not have any next state to go it. We identify three special cases for when our Turing Machine stops:

Crash A Turing Machine *crashed* when it attempts to move the head past the leftmost square.

Hang A Turing Machine *hangs* when it either crashes or when it reaches a state and a tape position for which is has no next step to take, and the state it reaches is a normal state *i.e.,* not a special state.

Halt A Turing Machine *halts* when it reaches a state and a tape position for which is has no next step to take, and the state it reaches is a special state, called a *final state*.

Termination modes of a Turing Machine

Halt reach a specially designated *end state* with no further transitions.

Crash attempt to move past leftmost endpoint of tape.

Hang crash or reach a "normal" state with no further transitions. (being stuck).

.

Slide 9

Slide 10 formalises these intuitions relating to Turing Machines. Note that:

- The set of tape symbols, Σ, is necessarily non-empty since it must at least have one element, $B \in \Sigma$.
- The set of states, Q, is necessarily non-empty because at least q_0 must be contained in it.
- The final state, q_H, is not part of the Turing Machine states and thus it is never part of the domain for the Turing Machine transition function δ. This means that computation must stop once the final state is reached, because we are guaranteed not to have any transitions from this state.

- The transition function, δ, is a *partial* function. There are state $\times$ symbol pairs for which there is no next step. These cases will constitute our hanging configurations of a Turing Machine.

Formalising Turing Machines

Definition 1 (Turing Machine)**.** $M \in$ TMach, is a sextuple

$M = \langle Q, \Sigma, \delta, q_0, q_H, B \rangle$ where

Q	*finite set of states*
$q_0 \in Q$	*start state*
$q_H \notin Q$	*final state*
Σ	*finite set of tape symbols*
$B \in \Sigma$	*blank symbol*
$\delta : (Q \times \Sigma) \rightharpoonup (Q \cup \{q_H\}) \times (\Sigma \cup \{\mathbf{L}, \mathbf{R}\})$	*transition function*

Slide 10

This formalisation may appear very cumbersome at first. However there are certain conventions we can use to simplify our descriptions of Turing Machines. These are outlined in Slide 11. This allows us to only specify 3 parameters from a Turing Machine sextuple.

Formalising Turing Machines... (cont.)

Input Symbols
For any $M = \langle Q, \Sigma, \delta, q_0, q_H, B \rangle$ the *input symbols* are $\Sigma \setminus B$.

Convention

q_0 will always be the start state.
h will always denote the end state, q_H.
$\#$ will always be the blank symbol B.

A Turing Machine M can thus be specified by supplying three parameters, *i.e.*, Q, Σ, and δ.

Slide 11

Example 2 (The Erase Turing Machine). Slide 12 gives a formal description of a Turing Machine that erases all the input on the tape, using the conventions for B, q_0 and q_H. It takes a string of (consecutive) a symbols starting from the leftmost position on the tape and returns a tape consisting of only blank symbols. Note that the machine assumes that the tape symbols can only come from the set $\{a, \#\}$. The machine starts from state q_0 and works by reading the tape symbol at the location where the head is:

- if the symbol is "a" then it erases it *i.e.*, the transition $(q_0, a) \mapsto (q_1, \#)$, and the moves to the head to the next location (and the start state q_0) *i.e.*, $(q_1, \#) \mapsto (q_0, \mathbf{R})$.
- If the symbol is $\#$(and we are in state q_0), then it means that we have reached the end of the string to, and thus we halt *i.e.*, $(q_0, \#) \mapsto (h, \#)$

Note that the transition $(q_1, a) \mapsto (q_1, a)$ is never used by this procedure; we introduced it to make the transition function *total*. It is not necessary to make our transition function total but some Turing Machine definitions require this (we won't in these notes).

Example 1

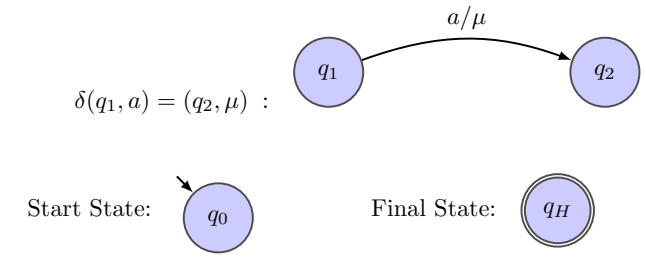
The Erase Turing Machine

$$M_{\text{erase}} = \left\langle \begin{array}{rl} Q & = \{q_0,\, q_1\} \\ \Sigma & = \{a,\, \#\} \\ \delta & = \left\{ \begin{array}{l} (q_0, a) \mapsto (q_1, \#),\ (q_0, \#) \mapsto (h, \#), \\ (q_1, a) \mapsto (q_1, a),\ (q_1, \#) \mapsto (q_0, \mathbf{R}), \end{array} \right\} \end{array} \right\rangle$$

Slide 12

Formal descriptions can sometimes be hard to understand (especially when the number of states (and transitions) are quite large. A more intuitive from of descriptions are Turing Machine diagrams, whereby we describe a transition $(q_1, a) \mapsto (q_2, \mu)$ where $\mu \in (\Sigma \cup \{\mathbf{L}, \mathbf{R}\})$ as shown on Slide 13. Start states and final states are also give a special representation.

Diagrammatic notation for Turing Machine Transitions

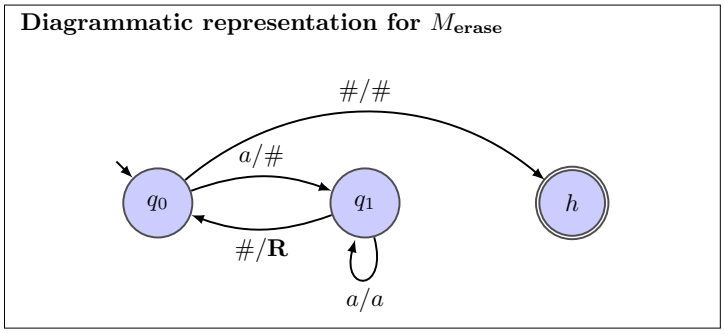


Slide 13

As an example, we consider a diagrammatic representation of M_{erase} from Slide 12. This is given on Slide 14.

2.1.1 Exercises

1. Construct a Turing Machine that takes a string of as as input and shifts the string by one position to the right.



Slide 14

2. Construct a Turing Machine that takes a string of as and bs as input and shifts the string by one position to the right.
3. Construct a Turing Machine that takes a string of as and bs and replaces every occurrence of a by c.
4. Construct a Turing Machine that takes a string of as and doubles it in length.
5. Construct a Turing Machine that takes a string of as and halves it in length.
6. One alternative definition of Turing Machines is to enforce the transition function Σ to be total. How does this affect the expressive power of our Turing Machines?

2.2 Configurations and Computations

Having formally described the structure of a Turing Machine, we now turn our attention to describing formally its behaviour. The observable state a Turing Machine is currently in is defined in terms of:

1. the contents of the tape
2. the position of the head on the tape
3. the current state the control is in.

For any Turing Machine, these three components uniquely determine the *next* step of the machine (recall that the transition function is deterministic). Moreover, such information completely determines the *entire sequence of steps* that the machine will take.

Such a description is also called the *instantaneous description* of the machine, because at each computational step, one or more components of the description will change. In these notes, we will refer to such a description as the *configuration* of the machine as is formally described in Slide 16.

In Def. 3, we split the string on the Turing Machine tape into three parts *i.e.*, x, a, and y. The string x describes the part of the string from the leftmost position of the tape up to the position just before the head. In the cases when the head happens to be at the leftmost position, the string x has the value ϵ, *i.e.*, the empty string. The symbol a denotes the tape symbol at the current position of the head. The string y describes the string on the tape to the right of the head. Note here that since the tape is *infinite* to the right, this string y is in reality infinite as well. However, we chose to describe only the string of tape symbols (ending with a non-blank symbol) beyond which there are only blank symbols to the right. Since we will always consider a finite number of steps, we can only write a finite number of symbols on the tape, which also means that a finite y would suffice.

Turing Machine Instantaneous Description

The status of a Turing Machine at any stage of computation can be described in terms of:

- the tape contents.
- the head position.
- the control state.

Slide 15

Formalising Configurations

Definition 3. For any Turing machine $M = \langle Q, \Sigma, \delta \rangle$, a configuration of M, ranged over by c, is a quadruple

$$\langle q,\ x,\ a,\ y \rangle \qquad \text{where}$$

$q \in Q \cup \{h\}$	*a state of M, possibly the halt state*
$x \in \Sigma^*$	*the symbols on the tape to the left of the head*
$a \in \Sigma$	*the symbol at the head position*
$y \in (\Sigma^*(\Sigma \setminus \{\#\})) \cup \{\epsilon\}$	*the symbols on the tape to the right of the head, up to the rightmost non-blank symbol*

Slide 16

Configuration Shorthand

We often express the configuration $\langle q,\ x,\ a,\ y\rangle$ as

$$\langle q,\ x\underline{a}y\rangle$$

.

Example Configurations of $M_{\mathbf{erase}}$

- $\langle q_0,\ \underline{a}aaa\rangle$
- $\langle q_1,\ \underline{\#}aaa\rangle$
- $\langle q_0,\ \#\underline{a}aa\rangle$
- $\langle q_1,\ \#\underline{a}aa\rangle$

Slide 17

We often employ a shorthand notation for configurations as shown in Slide 17, whereby a single string represents the string on the tape and a single symbol is checked to denote the current position of the head.

For any Turing Machine, not all configurations are *reachable* from the start state. For instance, let us consider the configurations on Slide 17 for M_{erase}:

- $\langle q_0,\ \underline{a}aaa\rangle$ is clearly reachable since it is the start state itself.
- $\langle q_1,\ \underline{\#}aaa\rangle$ is reachable after 1 step, using the transition $(q_0, a) \mapsto (q_1, \#)$.
- $\langle q_0,\ \#\underline{a}aa\rangle$ is reachable after 2 steps, using the transition $(q_1, \#) \mapsto (q_0, \mathbf{R})$.
- $\langle q_1,\ \#\underline{a}aa\rangle$ is *not* reachable.

An execution of a Turing Machine can be seen as a sequence of configurations, whereby every computation step involved going from one configuration to another using a transition from the Turing Machine. Slide 4 formally describes this as a relation between configurations of a Turing Machine M called *yield*, denoted as $\vdash_M$. This relation is then lifted to its transitive reflexive closure in Slide 5, formalising *reachability* between configurations (using an arbitrary number of computational steps.)

Reachability can be envisaged as the *evaluation* of a Turing Machine from one particular configuration to another, whenever the configuration reached is terminal: we have enough machinery to talk formally about the computation between one configuration and another, whereby we would be concerned with the intermediary configurations and the number of steps taken by the computation. Slide 21 describes this.

Using Def. 4, 5 and Def. 6, we can identify special terminal configurations of a Turing Machine, *i.e.*, computations that cannot *compute* any further:

- A configuration c is called *terminal* in M whenever there does not exists another configuration c' such that $c \vdash_M c'$. We sometimes write this as $c \nvdash_M c'$.
- A configuration c is called *diverging* in M if there exists a c' such that $c \vdash_M c'$, and c' is not terminal and diverging.
- A terminal configuration is called *halting* if it is of the form $\langle q_H,\ x\underline{a}y\rangle$, for arbitrary tape string $x\underline{a}y$.
- A terminal configuration is called *hanging* if it is of the form $\langle q,\ x\underline{a}y\rangle$, where $q \neq q_H$. We sometimes will refer to such configurations as *stuck*.

Formalising Yield amongst Configurations

Definition 4. For any $M = \langle Q, \Sigma, \delta \rangle$, the yield relation amongst two configurations of M is defined,

$$\langle q_1,\, x_1 \underline{a_1} y_1 \rangle \vdash_M \langle q_2,\, x_2 \underline{a_2} y_2 \rangle$$

when $\exists$ a move $\mu \in \Sigma \cup \{\mathbf{L}, \mathbf{R}\}$ such that $\delta(q_1, a_1) = (q_2, \mu)$ and

$\mu \in \Sigma$, where $x_1 = x_2$, $y_1 = y_2$ and $\mu = a_2$; or

$\mu = \mathbf{L}$, where $x_1 = x_2 a_2$ and either

$(a_1 = \# \wedge y_1 = \epsilon)$ and $y_2 = \epsilon$

$(a_1 \neq \# \vee y_1 \neq \epsilon)$ and $y_2 = a_1 y_1$.

$\mu = \mathbf{R}$, $x_2 = x_1 a_1$ and either

$y_1 = y_2 = \epsilon$ and $a_2 = \#$; or

$y_1 = a_2 y_2$

Slide 18

Formalising Reachability amongst Configurations

Definition 5. For any $M = \langle Q, \Sigma, \delta \rangle$, the reachability relation amongst two configurations of M is defined,

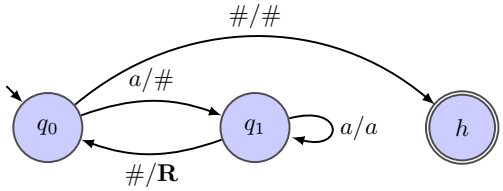
$$\langle q_1,\, x_1 \underline{a_1} y_1 \rangle \vdash^*_M \langle q_2,\, x_2 \underline{a_2} y_2 \rangle$$

as the transitive reflexive closure of the yields relation, $\vdash_M$.

Slide 19

Examples of Yield and Reachability

Recall M_{erase}:



- $\langle q_0, \underline{a}aaa\rangle \vdash_{M_{\text{erase}}} \langle q_1, \underline{\#}aaa\rangle$
- $\langle q_1, \underline{\#}aaa\rangle \vdash_{M_{\text{erase}}} \langle q_0, \#\underline{a}aa\rangle$
- $\langle q_0, \#\#\#\#\underline{\#}\rangle \vdash_{M_{\text{erase}}} \langle q_H, \#\#\#\#\underline{\#}\rangle$
- $\langle q_0, \underline{a}aaa\rangle \vdash^*_{M_{\text{erase}}} \langle q_0, \#\underline{a}aa\rangle$
- $\langle q_0, \underline{a}aaa\rangle \vdash^*_{M_{\text{erase}}} \langle q_H, \#\#\#\#\underline{\#}\rangle$

Slide 20

Formalising Turing Machine Computation

Definition 6. For any Turing Machine M, a computation is a sequence of configurations $c_1, c_2, \ldots, c_n$ such that

$$\underbrace{c_1 \vdash_M c_2 \vdash_M \ldots \vdash_M c_n}_{n}$$

If the sequence contains n configurations, we say that the computation is of length n.

Slide 21

A Turing Machine *evaluation* from a particular configuration is therefore a computation from that configuration to a terminal configuration (halting or hanging).

2.3 Turing Computability

We now consider how Turing Machine can be applied to perform different classes of computational tasks. The first (and perhaps most general) class of computations we shall consider is that of the computation of functions over arbitrary denumerable sets. By *denumerable* we effectively mean that the elements of the domain and the co-domain of the function to be computed can be encoded as strings over a suitable alphabet.

Computing with Turing Machines

The General Setup

Encode the problem to be computed on tape.

Execute the machine until it halts.

Decode the string on the tape as the result.

Caveats

- The machine may hang.
- The machine may diverge.

Slide 22

The setup we shall consider is the following: for an arbitrary function $f(x) = y$ Turing Machines will take encodings of the input from the domain x over the Turing Machines' alphabet, *i.e.*, Σ^*, and halt, return the encoding of the output from the function co-domain y, corresponding to the input x, again encoded over the alphabet Σ^*. Halting on an input x is defined in Def. 7 on Slide 23. Note that, since every Turing Machine computes deterministically, for every given input, irrespective of the number of times a machine is run, a unique string is left on the tape whenever the machine stops computing. A Turing Machine can also diverge (*i.e.*, loop forever) for a particular input encoding, and determinism ensures that if a Turing Machine diverges once on an input, it will diverge every time it is run with that input. We will therefore say that a Turing Machine computes a *partial function* rather than a total one. When it halts, the string left on the tape is the output for the function. When it hangs or diverges, we consider the output to be *undefined*. These points are summarised on Slide 23.

Slide 24 defines what it means for a function to be *Turing-computable*. The blank symbols at the beginning and end of the input and output strings are not essential. Here they act as delimiters of the input, but they will be used for other purposes in subsequent developments. Def. 8 is quite flexible. For instance, for functions of arity greater than one *i.e.*, $f : (\Sigma_1^*)^k \to \Sigma_2^*$ we simply generalise the form of input as strings separated by blanks. Thus, a Turing Machine M computes a function f if for any $x_1, x_2, \ldots, x_k \in \Sigma_1^*$ whenever $f(x_1, x_2, \ldots, x_k) = y$ the Turing Machine M evaluates to:

$$\langle q_0, \underline{\#}x_1\#x_2\#\ldots\#x_2\#\rangle \vdash_M^* \langle q_H, \#y\underline{\#}\rangle$$

Example 9 (Binary Inverse is Computable). As an example, we now consider the computability of the inverse function over binary numbers. By this we mean that, for any string inputted consisting of 0s and 1s, the

Turing Machines Computing Functions

Definition 7. A Turing Machine M *halts* on input x whenever the tape is initialised with the string $\underline{\#}x\#$, and the head is at the leftmost position on the tape, and starting from the initial state, the Turing Machine evaluates down to a halting configuration

$$\langle q_0, \underline{\#}x\# \rangle \vdash_M^* \langle q_H, y\underline{a}z \rangle$$

If M halts on every input, it computes a *total function*, otherwise, it computes a *partial function*.

Slide 23

Turing Computability

Definition 8. Let M be a Turing Machine over Σ, and let Σ_1 and Σ_2 be alphabets included in $\Sigma \setminus \{\#\}$. Then M computes a function $f : \Sigma_1^* \to \Sigma_2^*$ if for any $x \in \Sigma_1^*$ such that $f(x) = y$ we have

$$\langle q_0, \underline{\#}x\# \rangle \vdash_M^* \langle q_H, \#y\underline{\#} \rangle$$

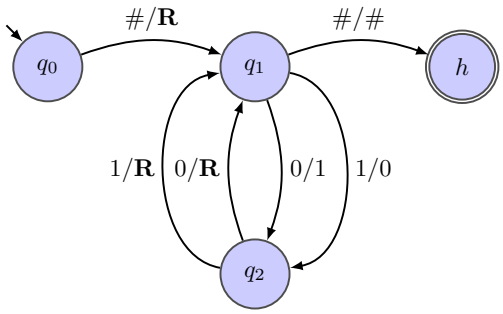
If such a Turing Machine M exists, then function f is *Turing-computable*.

Slide 24

function replaces all the 0s with 1s and vice-versa. The Turing Machine depiction of Slide 25 shows how this function can be computed by a Turing Machine with 4 states. This Turing Machine halts for any (finite) input, and is therefore said to compute a total function.

Examples of Computable Functions

Consider the inverse function for binary numbers, such that, for example $f(10100) = 01011$. This function is Turing-computable as shown below. The function computed is also total:



Slide 25

Using appropriate encodings, Turing Machines can compute functions over any countable domain. Slide 26 considers the naturals, NAT, as an example of a countable set. We can easily define a simple encoding of natural numbers defined over the simple alphabet $\Sigma = \{a\}$, whereby the number n would be encoded as n repetitions of a (*e.g.,* 5 is encoded as *aaaaa*). A Turing Computable k-ary function over the naturals can thus be expressed as a Turing Machine which takes k (encodings of) naturals as input and returns a natural as output. Thus addition over naturals is Turing Computable if we can define a Turing Machine that can take two encodings of natural numbers as input and returns the encoding of their addition as output. As an instance, such a Turing Machine M_{add} would take the encodings of 2 and 3 and return the encoding of 5 as a result:

$$\langle q_0, \underline{\#}aa\#aaa\#\rangle \vdash^*_{M_{\text{add}}} \langle q_H, \#aaaaa\underline{\#}\rangle$$

A similar notion can be applied to *predicates*. An n-ary predicate P over a countable domain D is Turing computable if there exists a Turing Machine M such that for some encoding from D to Σ, M on the encoding of input $x_1, \ldots, x_n$ halts with output `t` when $P(x_1, \ldots, x_n) = \mathit{true}$ and halts with with output `f` whenever $P(x_1, \ldots, x_n) = \mathit{false}$. The symbols `t` and `f` are assumed to be in the alphabet Σ of M but not necessarily in the encoding alphabet for domain D. Such a predicate is called *Turing Decidable* or just *decidable*. A predicate is said to be *Turing acceptable* (or just *acceptable*) if there exists a Turing Machine M such that for some encoding from D to Σ, M on the encoding of input $x_1, \ldots, x_n$ halts with output `t` when $P(x_1, \ldots, x_n) = \mathit{true}$ (but gives no guarantee of termination whenever $P(x_1, \ldots, x_n) = \mathit{false}$. This is summarised on Slides 27 and 29.

On Slide 28 we outline a Turing Machine that takes an input of the form $\#\overbrace{a \ldots a}^{n}\#$ and

- halts with output $\#\overbrace{a \ldots a}^{n}$t# whenever n is even;
- halts with output $\#\overbrace{a \ldots a}^{n}$f# whenever n is odd

Functions over Countable Domains

Simple encoding of naturals

- 0 as ϵ, 1 as a, 2 as aa, ...
- n as a^n.

Using this encoding we can define what it means for a k-ary function over naturals

$$f : \textsc{Nat}^k \to \textsc{Nat}$$

to be Turing Computable.

Examples of 2-ary functions over Naturals

- Addition, Subtraction, Multiplication, Division, ...

Slide 26

Turing Decidability

Consider the predicate

$$P : D^k \to \textsc{Bool}$$

Definition 10. P is *Turing decidable* there exists a Turing Machine M such that for some encoding from D to Σ, M on the encoding of input $x_1, \ldots, x_k$ halts with output #t# when $P(x_1, \ldots, x_n) = \mathit{true}$ and halts with with output #f# whenever $P(x_1, \ldots, x_n) = \mathit{false}$.

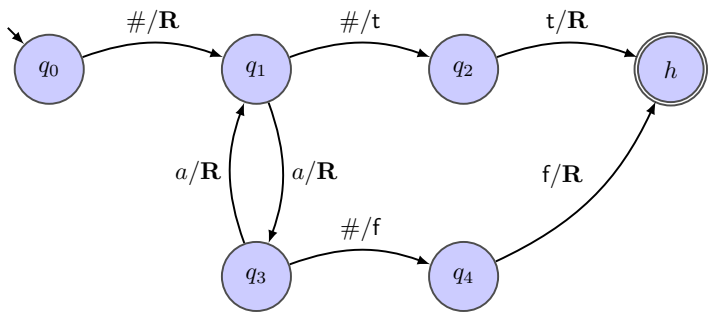
Example 11. The function

$$\text{isEven} : \textsc{Nat} \to \textsc{Bool}$$

Slide 27

Although this Turing Machine does not directly show that the predicate isEven : NAT → BOOL is decidable (for a respective encoding of the Naturals), it provides enough evidence that a Turing Machine can be constructed producing the output format required by Def. 8. In our cases, we simply need to extend the Turing Machine on Slide 28 so as to clean-up the as preceding the boolean value, something that can be done using a Turing Machine with mechanisms similar to M_{erase} of Slide 14.

Turing Machine quasi-accepting $\{a^n \mid n \bmod 2 = 0\}$



Slide 28

Turing Acceptability and Decidability

Consider the predicate

$$P : D^k \to \text{BOOL}$$

Definition 12. P is *Turing acceptable* there exists a Turing Machine M such that for some encoding from D to Σ, M on the encoding of input $x_1, \ldots, x_k$ halts with output #t# when $P(x_1, \ldots, x_n) = true$.

Example 13 (Language Membership in $\mathcal{L}(G)$). For arbitrary Phrase Structured Grammar G and string x, is there a derivation such that:

$$S_G \Rightarrow_G^* x$$

Slide 29

Note that since we can deal with predicates, we can also view Turing Machines as *language acceptors*. Recall that a machine (whether a DFSA, an NFSA, an NPDA or, in this case, a Turing Machine) is said to recognise a language L if it accepts any string x whenever $x \in L$. In the case of Turing Machines, *"accepting a string"* can be formulated as halting with output #t#. This suggests the following definition for the language accepted by a Turing Machine, M:

$$\mathcal{L}(M) \stackrel{\text{def}}{=} \left\{x \mid \langle q_0, \underline{\#}x\#\rangle \vdash_M^* \langle q_H, \#\mathtt{t}\underline{\#}\rangle\right\}$$

While this works in practice, we shall consider a slightly different (an perhaps more convenient) definition of language Turing acceptance in Sec. 2.4. The principle is however the same because:

1. language acceptors are merely predicates over languages
2. strings in a language, by definition, belong to a countable domain (*i.e.*, we can map any string to a unique natural number through some encoding).

2.3.1 Exercises

1. Construct a Turing Machine computing 2-ary addition function over naturals.
2. Construct a Turing Machine computing 2-ary subtraction function over naturals.
3. Construct a Turing Machine computing 2-ary multiplication function over naturals.
4. Construct a Turing Machine computing the unary functions **mod** 3 and **div** 3 over naturals.
5. Construct a Turing Machine to show that the predicate returning *true* when strings ranging over the alphabet $\{a\}$ are of length 1 (and false otherwise) is Turing decidable.
6. Construct a Turing Machine to show that the predicate returning *true* when strings ranging over the alphabet $\{a, b\}$ have an even number of as (and false otherwise) is Turing decidable.
7. Consider the function

$$f(n) \stackrel{\text{def}}{=} \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{otherwise} \end{cases}$$

   If $f^k(n)$ denotes k applications of the function f to the argument n, *i.e.*, $\overbrace{f(\ldots(f}^{k}(n))$, show that the following predicate is Turing acceptable.

$$P(n) \stackrel{\text{def}}{=} \begin{cases} \mathit{true} & \text{if } \exists k.f^k(n) = 1 \\ \mathit{false} & \text{otherwise} \end{cases}$$

8. One alternative definition of Turing Machines is to enforce the transition function δ to be total. This means that our Turing Machines can now only hang by crashing. Describe a way how one can take *any* Turing Machine description adhering to Definition 1 and convert it to adhere to the new definition with total transition functions.
9. One important aspect of formalising Turing Machine computations is that we can prove properties about them. Prove that every Turing Machine evaluation is deterministic. In other words, show that for any Turing Machine M:

$$c \vdash_M^* c' \text{ and } c \vdash_M^* c'' \text{ where } c' \text{ and } c'' \text{ are terminal implies that } c' = c''.$$

10. Prove that the Turing Machine computing the inverse function of Example 9 is total. In other words, formalise the Turing Machine on Slide 25 as M and show that for any $x \in \{0,1\}^*$:

$$\exists c \text{ such that } \langle q_0, \underline{\#}x\#\rangle \vdash_M^* c \text{ such that } c = \langle q_H, y\rangle \text{ for some } y.$$

2.4 Turing Machines as Language Acceptors

In Sec. 2.3 we briefly touched on the subject of using Turing Machines as language acceptors. In this section we discuss this problem directly and pose the question of whether there are languages whose membership predicates are not decidable by any Turing Machine. This question also leads to the formulation of a class of languages known as *recursively-enumerable* languages, which contain all languages that are recognised by some algorithm (or automaton). This class turns out to contain all regular languages and context-free languages, and a lot of other languages.

First, we *reformulate our definitions* of acceptance and decidability from those defined earlier in Sec. 2.3 for predicates, which required the alphabet of our Turing Machine to contain two special symbols $\mathtt{t}$ and $\mathtt{f}$ and the final tapes of terminating configurations to be of the form $\#\mathtt{t}\underline{\#}$ or $\#\mathtt{f}\underline{\#}$. This works, but in practice, this requires our language accepting/deciding Turing Machines to clean up the tape before terminating, which can sometimes lead to unnecessarily complicated Turing Machine definitions.

Luckily, since we only need to distinguish only between two modes of termination (*i.e.,* acceptance or rejection) we can use the two forms of termination, *i.e.,* halting and hanging (outlined earlier in Sec. 2.2) to recast our definitions as follows. First we identify a *hanging state*, q_N, to be a state with no transitions just like the final state q_H. The crucial aspect of such a definition is that, just as with q_H, when we reach q_N, we are guaranteed to terminate (there are are no further transitions and we are therefore stuck!). However instead of halting, we hang, since $q_N \neq q_H$. Equipped with this definition we recast acceptance and rejection as follows:

Accept: A Turing Machine M is said to accept a string x whenever it *halts* on the input $\underline{\#}x\#$.

Reject: A Turing Machine M is said to reject a string x (on q_N) whenever it *hangs* on the input $\underline{\#}x\#$.

Thus for acceptance, we now merely require the Turing Machine to end computation at the Turing Machine final state q_H, and for rejection we simply require the Turing Machine to end computation in q_N, which is *not* the final state. Crucially, no constraint is now placed on the contents of the tape of these terminal configurations. This reformulation is described on Slide 30.

Turing Machines as Language Acceptors

Definition 14 (Hanging State)**.** A state $q \in Q$ is *hanging* iff $\forall a \in \Sigma.\ \nexists \mu, q'$ such that $(q, a) \mapsto (q', \mu)$. We denote hanging states as q_N.

Definition 15. For any Turing Machine M and string $x \in \Sigma^*$

Accept: M accepts x iff for some $a \in \Sigma$, $y_1, y_2 \in \Sigma^*$

$$\langle q_0, \underline{\#}x\# \rangle \vdash_M^* \langle q_H, y_1 \underline{a} y_2 \rangle$$

Reject: M rejects x iff for some $a \in \Sigma$, $y_1, y_2 \in \Sigma^*$

$$\langle q_0, \underline{\#}x\# \rangle \vdash_M^* \langle q_N, y_1 \underline{a} y_2 \rangle$$

Slide 30

Slide 31 defines what it means for a language to be *Turing acceptable* which is in correspondence with the definition for predicate Turing Acceptance (*cf.* Def. 12.) An alternative way how to define this is by

first defining $\mathcal{L}(M)$, the language accepted by a Turing Machine M and then defining a language Turing acceptable whenever there exists a Turing Machine that accepts it. Turing acceptable languages are also known as *recursively-enumerable* languages (*cf.* Slide 32.)

Turing Acceptable Languages

Definition 16. A language L defined over Σ^* is *Turing acceptable* if there exists a Turing Machine M such that:

- $x \in L$ implies M accepts x.
- M accepts x implies $x \in L$.

Slide 31

Recursively Enumerable Languages

Definition 17. The language accepted by a Turing Machine M is defined as

$$\mathcal{L}(M) \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid M \text{ accepts } x\}$$

Definition 18. A language L is *recursively-enumerable* if and only if there exists a Turing Machine M that accepts L, *i.e.*, $L = \mathcal{L}(M)$.

Slide 32

An observation should be made about recursively-enumerable (r.e.) languages. If M is a Turing Machine that accepts a r.e. language L, then we do not know whether for any string x that falls outside of L, *i.e.*, $x \in \overline{L}$, M will reject it or not. Indeed, it may be the case that for such strings, the Turing Machine diverges. All we know is that if $x \in L$, then M will terminate and it will terminate in a success configuration, *i.e.*, it halts.

There is another class of languages for which the membership function is *decidable*. By this, we mean that for such a language L, there exists a Turing Machine M such that:

- if $x \in L$ then M accepts x
- if $x \notin L$ then M rejects x.

Such a Turing Machine gives us stronger guarantees than Turing Machines for r.e. languages because they terminate on every input. When such a Turing Machine exists, we say that the corresponding language recognised by it is a *recursive* language, as defined on Slide 33.

Example 21 (Recursively Enumerable Languages). Slide 34 describes a language that is recursively enumerable. The definition of L_1 is based on a non-monotonic[1] function f which can either divide the argument

[1]Without being too formal, the non-monotonic property means that when we apply the function to an argument, we are not guaranteed that the result will always be increasing (or decreasing) compared to the argument. The functions $\mathbf{suc}(x) = x + 1$ and $\mathbf{dec}(x) = x - 1$, by comparison, are monotonic: repeated applications of **suc** will lead to an ever increasing result whereas repeated applications of **dec** will yield an ever decreasing result.

Recursive Languages

Definition 19. A Turing Machine M *decides* a language L whenever, for any $x \in \Sigma^*$

- $x \in L$ iff M accepts x.
- $x \notin L$ iff M rejects x.

Definition 20. A language L is *recursive* if and only if there exists a Turing Machine M that decides L.

Slide 33

by 2 when it is even or else increment it by $3x + 1$ when it is odd. In spite of the fact that $f(n)$ is non-monotonic, there exist certain arguments for which repeated applications of $f(-)$ eventually leads to the result 1. The number 8 is one obvious example, which reduces to 1 after 3 applications of f *i.e.*, $f(8) = 4$, $f(4) = 2$, $f(2) = 1$. The number 3 is a less obvious example which can reduce to 1 after 7 applications of f, with the following intermediary values: $10, 5, 16, 8, 4, 2, 1$.

Example of a Recursively Enumerable Language

Consider the function

$$f(n) \stackrel{\text{def}}{=} \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{otherwise} \end{cases}$$

If $f^k(n)$ denotes k applications of the function f to the argument n, *i.e.*, $\underbrace{f(\ldots(f}_{k}(n))$, then L_1 below is recursively enumerable

$$L_1 \stackrel{\text{def}}{=} \left\{ a^n \mid \exists k. f^k(n) = 1 \right\}$$

Slide 34

Slide 34 describes a shorthand for writing k repeated applications of the function f as f^k, which allows us to state concisely that $f^3(8) = 1$ and $f^7(3) = 1$. Thus the language L_1 described in Slide 34 is a simple number encoding of these numbers that can be reduced to 1 after some number of applications of f, using the alphabet $\{a\}$. Thus, for example, $a^8 \in L_1$ and $a^3 \in L_1$. Determining whether $a^n \in L_1$ for some n amounts to finding a witness number k such that $f^k(n) = 1$. The witness number k can be found using the following algorithm:

1. if the argument is equal to 1 then transition to the final state q_H and accept.
2. if the argument is even then
   (a) divide by 2.

(b) goto step 1.

else

(a) multiply by 3 and add 1.

(b) goto step 1.

The argument for L_1 being recursively enumerable proceeds as follows:

- Testing whether a^n is of length 1 (step 1) and whether a^n is of even length (step 2) can be computed by a Turing Machine.
- Dividing a^n by half (step 2a), multiplying a^n by 3 and incrementing a^n by 1 (step 2c) are also functions that can be computed by a Turing Machine.
- Looping (steps 2b and 2d) can also be implemented using Turing Machines.

Thus we can construct a Turing Machine that implements the above algorithm and terminates when it finds a witness k for a^n such that $f^k(n) = 1$. Hence L_1 is r.e. Note however that we have no guarantee that the algorithm will terminate whenever $a^n \notin L_1$. Indeed it may well be the case that the Turing Machine implementing the above algorithm runs forever in these cases. As a result, we cannot claim that L_1 is recursive.

Example 22 (Recursive Languages). As an example of a recursive language, recall how we had discussed (in an earlier course) that the language $a^n b^n c^n$ was not a context-free language. Not only could the language *not* be described using a context-free grammar, but also, due to a result equating the languages recognised by context-free languages and NPDAs, it could *not* be recognised by any NPDA either (recall that NPDAs are non-deterministic automata augmented with a stack acting as a limited form of memory).

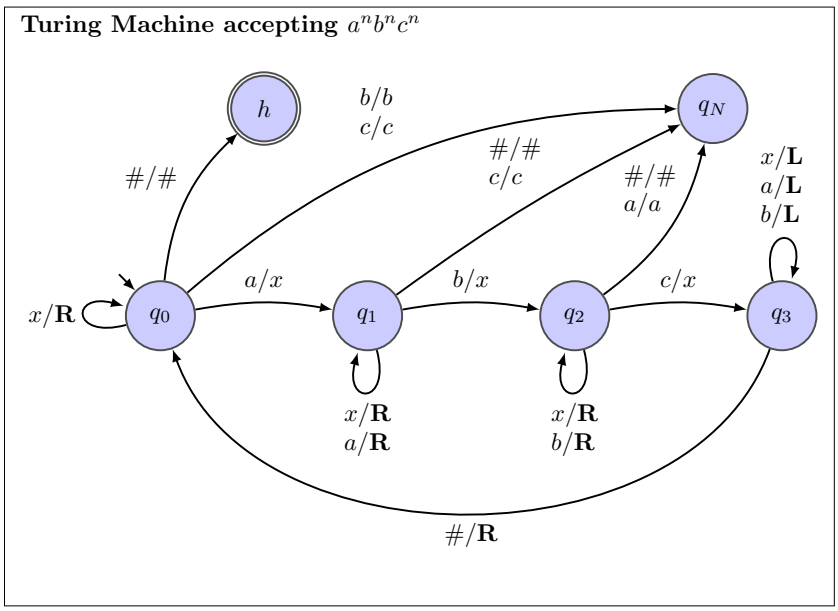
Here we get a sense as to the relative expressive power of Turing Machines when compared to NPDAs. Slide 35 describes a Turing Machine that can *decide* whether a string x is in the language $a^n b^n c^n$. This also effectively proves that the language $a^n b^n c^n$ is a recursive language. The algorithm implemented by the Turing Machine on Slide 35 works as follows. It employs an additional symbol x as part of its alphabet *i.e.,* $\Sigma = \{a, b, c, x\}$ (eliding #), and uses this special symbol to scratch out symbols of the string it has already processed. This procedure allows the Turing Machine to use the tape as both the input tape but also as the memory of the machine. The Turing Machine on Slide 35 also employs two terminal states, from which there are no outgoing transitions; the Turing Machine final state, q_H, denoting acceptance, and a 'stuck' state, q_N, denoting rejection.

The algorithm first checks whether there are any string symbols. If not, then it means that we have the empty string ϵ on our input tape, which is equal to $a^0 b^0 c^0$ *i.e.,* part of the language, it accepts. Otherwise, it first scratches the first a from the string, $(q_0, a) \mapsto (q_1, x)$, then moves on to scratch the first b, $(q_1, b) \mapsto (q_2, x)$, and the first c, $(q_1, c) \mapsto (q_3, x)$. If at any stage, this sequence is not found, the Turing Machine rejects. It then goes back to the beginning of the string and repeats the same procedure until it has scratched an equal number of as, bs and cs, at which point it accepts.

We conclude this section by stating some results relating to recursive and r.e. languages stated on Slide 36. Theorem 23 states that every recursive language is also a r.e. language.

Proof for Theorem 23. If L is recursive, then there exists a Turing Machine M that can *decide* on whether a string is in L or not. By Def. 19, Def. 18, Def. 17 and Def. 15, this same Turing Machine M can also accept every string belonging to L. The existence of such a Turing Machine makes L r.e. □

Theorem 24 talks about the complement of a language $\overline{L}$ (defined as $\Sigma^* \setminus L$) and states that if a language is recursive, its complement must also be recursive.



Slide 35

Proof for Theorem 24. If L is recursive, then there exists a Turing Machine M that can *decide* on whether a string is in L or not. By Def. 19 and 15 this M reaches its final state, q_H, whenever it is inputted a string $x \in L$ and reaches a hanging state, *i.e.*, some q_N, whenever it is inputted a string $x \notin L$. If we take this same Turing Machine M and replace all instances of q_N with the final state and $q_H$and vice-versa, then we would obtain a new Turing Machine M' that accepts $x \notin L$ and rejects $x \in L$. This Turing Machine precisely decides $\overline{L}$, making $\overline{L}$ recursive as well. □

As an example of the utility of Thm. 24 consider the case where we want to decide whether a string is in the language $L_2 = \{x \mid x \neq a^nb^nc^n \text{ for some } n \geq 0\}$. By Thm. 24 we can take the Turing Machine on Slide 35, which we know decides the language $a^nb^nc^n$, and make q_H a stuck state and q_N the final state, which would automatically give us a Turing Machine that can decide L_2.

Results about Recursive and Recursively Enumerable Languages

Theorem 23. *If L is recursive then L is recursively enumerable*

Theorem 24. *If L is recursive then its complement, $\overline{L}$, is recursive.*

Corollary 25. *If $\overline{L}$ is not r.e., then L is not recursive.*

Slide 36

Using Thm. 23 and Thm. 24 we can also derive useful corollaries. For instance, suppose that for some language L is it really easy to show that its complement $\overline{L}$ is not r.e. Then, from this information and Corollary 25 we can also deduce that L cannot be recursive.

Proof for Corollary 25. By taking the contrapositive of Thm. 23 and Thm. 24, then we know

1. If L is *not* r.e. then L is *not* recursive.
2. If $\overline{L}$ is *not* recursive, then L is *not* recursive either.

Now if $\overline{L}$ is not r.e., then by (1) we know $\overline{L}$ is not recursive. If $\overline{L}$ is not recursive, by (2) we know that L is not recursive. □

2.4.1 Exercises

1. Consider the following Alternative definition of rejection:
   Reject: M rejects x iff for some $a \in \Sigma$, $y_1, y_2 \in \Sigma^*$ where $q \neq q_H$ and $\langle q, y_1 \underline{a} y_2 \rangle$ is terminal.

   $$\langle q_0, \underline{\#} x \# \rangle \vdash_M^* \langle q, y_1 \underline{a} y_2 \rangle$$

   (a) How does this modified definition affect our proof for Thm. 24

   (b) Explain how, for any Turing Machine satisfying decidability with the modified definition of rejection, one can modify this Turing Machine so as to satisfy decidability with Def. 15

2. Consider the following Alternative definition of rejection:
   Reject: M rejects x iff it *crashes* on that input (*cf.* Sec. 2.1).
   Explain how, for any Turing Machine satisfying decidability with Def. 15 can be modified to satisfy the new decidability with the above definition for rejection.

3. Show that the language $\{a^n b^m \mid \quad n \geq m \geq 0\}$ is decidable.

4. Show that the language $\left\{a^n b a^{n+1} \mid \quad n \geq 0\right\}$ is decidable.