Topic list for revision

General techniques

1. Mathematical induction: ordinary, course of values, nested.
2. The countable/uncountable distinction is not required.

Languages and automata

1. Regular expression
2. Deterministic finite automaton, error state
3. Nondeterministic finite automaton, nondeterministic finite automaton with ε-moves
4. You should know how to algorithmically convert a regex to an DFA. You should know that a DFA can be converted to a regex, but don't need to know an algorithm for this.
5. Equivalence of DFAs
6. Minimization of total DFAs
7. DFAs for complements and intersections.
8. You should be able to prove that a language isn't regular. (The Pumping Lemma wasn't taught and isn't needed.)
9. Context free languages (not pushdown automata)
10. Chomsky normal form.
11. Ambiguity of context free grammars.

Complexity

1. Complexity of algorithms vs complexity of problems
2. Lower and upper bounds
3. O, Ω, θ notation

4. Polynomial and exponential complexity
5. $\mathcal{NP}$ problems, the two definitions, the question of whether $\mathcal{P} = \mathcal{NP}$.
6. $\mathcal{NP}$-completeness and SAT.

Turing machines

1. Execute Turing machines
2. Design Turing machines for simple tasks
3. Extended alphabet (2 tape, etc.) Turing machines look more powerful than a Turing machine, but it isn't because a Turing machine can simulate it
4. Macros and expansion
5. Church's thesis: any function on words that can be computed algorithmically can be computed by a Turing machine
6. Nondeterministic Turing machines.

Decidability

1. Problem, decision problem (a problem whose answer is yes/no)
2. Decidable and semidecidable problems
3. Primitive recursive functions: you need to be able to show that a function is primitive recursive by implementing it in Primitive Java. (You would be given the definition of Primitive Java.)
4. The Halting Theorem: whether a given program halts on given inputs is undecidable. (Knowing the proof is not required.)
5. Rice's Theorem: any semantic property of code that sometimes hold and sometimes fails to hold is undecidable. (Knowing the proof is not required, but may help you to understand other problems.)
6. If you're asked to show a problem is decidable, write a program to decide it. Or at least sketch how you would write a program.
7. If you're asked to show a problem is undecidable, you might reduce the halting problem to it, or you might appeal to Rice's theorem.

Note that at each point you should know practical aspects, e.g. uses of regular and context free languages, consequences of non-computability, etc.

Introducing Regular Languages and Automata

1 Regular expressions

1.1 Prologue: matching and finding

Look at these two problems.

1. A string is a "valid password" when it contains at least 8 characters and at least 2 digits. Given a string, say whether it is a valid password.

2. Given a string (e.g. a file), list all occurrences of email addresses within it. Each occurrence should be represented as a pair of numbers (i, m), where i is the start position (e.g. 0 for an occurrence at the start of the file) and m is the length.

The first of these is an example of a *matching problem*. The second is a *finding problem*. Such problems—and variations—often arise in computing, so people have made tools to solve them efficiently, To use such a tool, you have to specify when a word is a valid password or an email address or whatever. Often, we do this by means of a *regular expression*.

1.2 Definitions

The alphabet (set of characters) is called Σ. To keep things simple, let's suppose that it's $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. In a practical situation, it might instead be the ASCII alphabet, which has 128 characters. Or it might be the Unicode alphabet, which has $137,439$ characters. In any case, we'll assume that Σ is a finite set and contains at least two characters.

We write Σ^* for the set of all words. A *language* is a set of words, i.e. a subset of Σ^*. For example: the set of valid passwords is a language, and so is the set of email addresses. A regular expression (regexp), such as $\mathtt{c}(\mathtt{bb}|\mathtt{ca})^*$, represents a language, just as an arithmetic expression, such as $2 + (5 \times 3)$, represents a number.

Now let me explain how regexps work.

- The regexp $\mathtt{a}$ matches only the word $\mathtt{a}$.
- The regexp $\mathtt{b}$ matches only the word $\mathtt{b}$.
- The regexp $\mathtt{c}$ matches only the word $\mathtt{c}$.
- The regexp ε matches only the empty word ε.
- If E and F are regexps, then the regexp EF matches any word that's a concatenation of a word matched by E and a word matched by F.
- If E and F are regexps, then the regexp $E|F$ matches any word that either E or F matches.
- If E is a regexp, then the regexp E^* matches any word that's a concatenation of several (i.e. zero or more) words matched by E.
- The (rarely used) regexp $\emptyset$ doesn't match any word.

1.3 Precedence

For arithmetic expressions, $\times$ has higher precedence than $+$. Knowing this enables us to parse the expression $3+4\times2$ as $3 + (4 \times 2)$, for example. For regexps, the precedence laws are as follows: juxtaposition (which means "putting things next to each other") has higher precedence than $|$ and lower precedence than $*$. Knowing this enables us to parse $\mathtt{c}(\mathtt{bb}|\mathtt{ca})^*$ as $\mathtt{c}(((\mathtt{bb})|(\mathtt{ca}))^*)$, for example.

More operators that you can use:

- E^+ is short for EE^*. It matches any word that is a concatenation of one or more words matched by E.
- $E?$ is short for $\varepsilon|E$.

These have the same precedence as *.

Some tools provide additional operators, which make it possible to express fancier languages. Expressions using these additional operators may be called "regular expressions" in the tool documentation, but technically they are not regular.

Example 1

1. *Does the regexp* $\texttt{c(bb|ca)}^*$ *match* `ccacabb`*? YES/NO.*
2. *Does the regexp* $\texttt{c(bb|ca)}^*$ *match* `cbbcacac`*? YES/NO.*
3. *Does the regexp* $\texttt{(c(bb|ca)}^*\texttt{)}^*$ *match* `cccacacbbcbbca`*? YES/NO.*
4. *Do* $\texttt{(a|b)c}^*$ *and* $\texttt{ac}^*\texttt{|bc}^*$ *represent the same language? YES/NO.*
5. *Do* $\texttt{(a|b)c}^*$ *and* $\texttt{ac}^*$ *represent the same language? YES/NO.*

2 Some questions about regular expressions

Before we look into regexps in more detail, let's consider some questions. For some of these, the answers are far from obvious, but will emerge over the coming lectures.

2.1 Regular and Irregular Languages

Recall that a regexp represents a language. Any language $L \subseteq \Sigma^*$ that can be represented in this way is said to be *regular*. Questions:

1. Are there any languages that are not regular? Answer: Yes, and we'll see some examples.
2. Is the complement of a regular language always regular? (For example, is there a regexp for those words that are *not* matched by $\texttt{c(bb|ca)}^*$?) Answer: Yes.
3. Is the intersection of two regular languages always regular? (For example, is there a regexp for those words that are matched by *both* $\texttt{cc(bb|ca)}^*$ and $\texttt{c(bbbb|cca)}^*$?) Answer: Yes.

2.2 Decidability questions

A *decision problem* is a problem that, for any given argument, has a Yes/No answer. For example, the finding problem above is not a decision problem, because the answer (for a given file) is a set of pairs of numbers. A decision problem is said to be *decidable* when there is some program that, given an argument, says whether the answer is Yes or No.

We can ask the following questions about regexps:

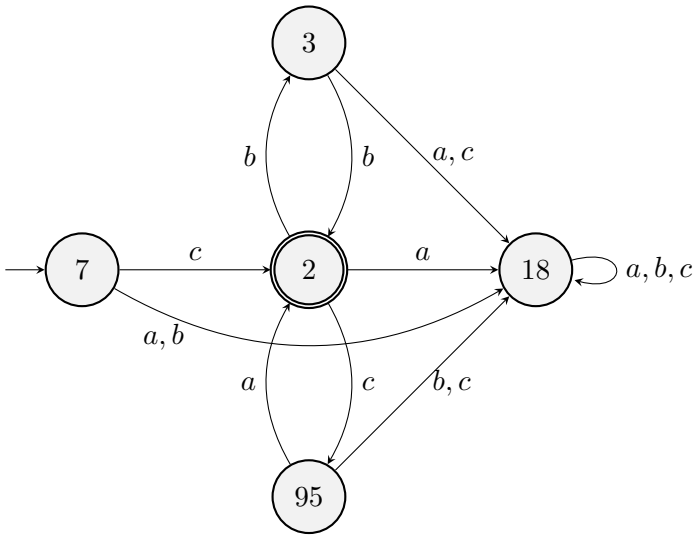
1. Is the matching problem for the regexp $\texttt{c(bb|ca)}^*$ *decidable*? In other words, is there some program that, when given a word w over our alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$, returns True if w matches $\texttt{c(bb|ca)}^*$, and False if it doesn't? (If w isn't a word over our alphabet, then it doesn't matter what happens.) Answer: Yes.
2. Is the matching problem for the regexp $\texttt{(c(bb|ca)}^*\texttt{)}^*$ decidable? Answer: Yes.
3. Is it the case that, for *every* regexp E, the matching problem for E is decidable? Answer: Yes.
4. Is the matching problem for regexps decidable? In other words, is there some program that, when given a regexp E and word w, returns True if w matches E, and False if it doesn't? Answer: Yes.
5. Is language equality for regexps decidable? In other words, is there some program that, when given regexps E and F, returns True if they represent the same language and False otherwise? Answer: Yes.

2.3 Efficiency questions

In the previous section, we asked whether certain problems can be solved at all. Another question is: can they be solved efficiently? After all, your customers aren't willing to wait a long time for an answer from your program. This question isn't very precise, but it's important. We'll see that for some of these problems, we can give a reasonably efficient solution.

3 Introducing automata

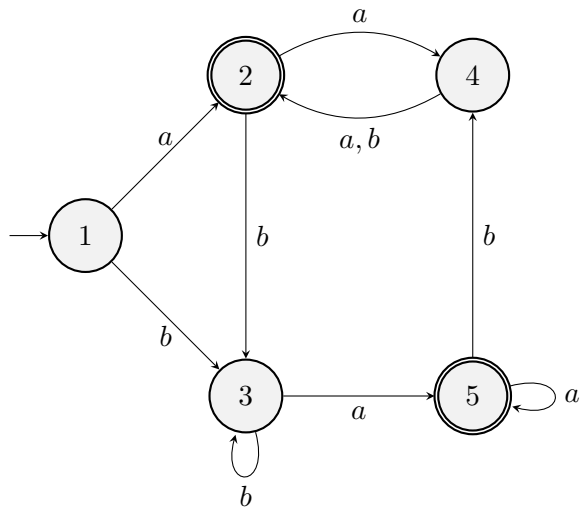
3.1 Deterministic automata



Recall that we wanted a program to solve the matching problem for $\texttt{c(bb|ca)}^*$. This can be achieved by the automaton shown. There are five *states*, represented as circles. The automaton processes a word by starting at the *initial state* (indicated by $\rightarrow$) and performing a *transition* as it inputs each letter. When the whole word has been input, the automaton returns Yes if the current state is *accepting*, indicated by a double ring. It returns No if the current state is *rejecting*, indicated by a single ring.

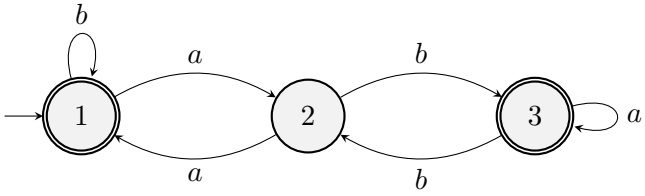
This is a *deterministic finite automaton* (DFA). "Deterministic" because the initial state and the result of each transition are specified. "Finite" because the set of states is finite.

Example 2 *Here is a DFA over the alphabet* $\{\texttt{a}, \texttt{b}\}$. *Are these words accepted?*



`abab`	*Y/N*
`ababba`	*Y/N*
`ababbaa`	*Y/N*
`bab`	*Y/N*
`baa`	*Y/N*
ε	*Y/N*

Example 3 *What about this DFA?*



`abb`	*Y/N*
`abbaba`	*Y/N*
`bba`	*Y/N*
ε	*Y/N*

Definition 1 *A deterministic finite automaton consists of the following data.*

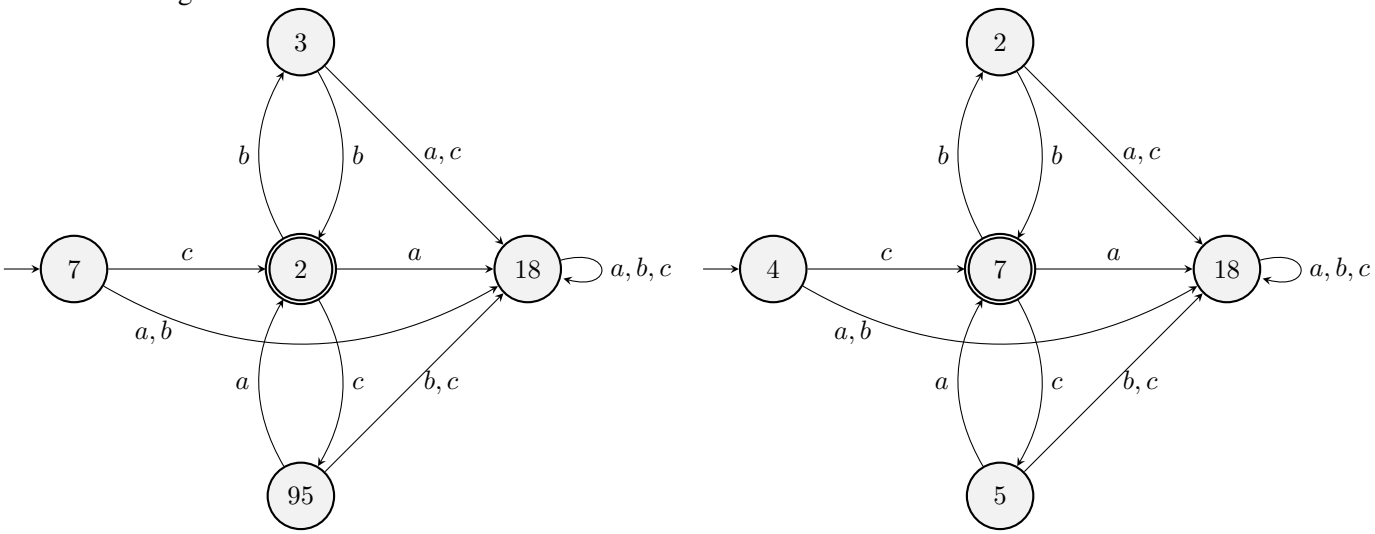
- *A finite set X of states.*
- *An initial state $p \in X$.*
- *A transition function $\delta\colon X \times \Sigma \to X$.*
- *A set of accepting states $\mathsf{Acc} \subseteq X$.*

In the below example,

$$\begin{aligned} X &= \{7, 2, 3, 95, 18\} \\ p &= 7 \\ \delta &= \{(7, \mathtt{a}) \mapsto 18, (7, \mathtt{b}) \mapsto 18, (7, \mathtt{c}) \mapsto 2, \\ &\quad (2, \mathtt{a}) \mapsto 18, (2, \mathtt{b}) \mapsto 3, (2, \mathtt{c}) \mapsto 95, \\ &\quad (3, \mathtt{a}) \mapsto 18, (3, \mathtt{b}) \mapsto 2, (3, \mathtt{c}) \mapsto 18, \\ &\quad (95, \mathtt{a}) \mapsto 2, (95, \mathtt{b}) \mapsto 18, (95, \mathtt{c}) \mapsto 18, \\ &\quad (18, \mathtt{a}) \mapsto 18, (18, \mathtt{b}) \mapsto 18, (18, \mathtt{c}) \mapsto 18\} \\ \mathsf{Acc} &= \{2\} \end{aligned}$$

3.2 Isomorphisms

Look at the following two DFAs.



Each of them solves the matching problem for `c(bb|ca)*`. They are almost the same, but not quite. We see the following correspondence:

State of the left DFA	State of the right DFA
3	2
7	4
2	7
18	18
95	5

This is called an *isomorphism*. It is a bijection (one to one correspondence) between the sets of states of the left DFA and the set of states of the right DFA with the following properties.

- The initial state in the left DFA corresponds to the initial state in the right DFA.
- For each state x in the left DFA corresponding to x' in the right DFA, and for each character c, the result of starting at x and reading c in the left DFA corresponds to the result of starting at x' and reading c in the right DFA.
- For each state x in the left DFA corresponding to x' in the right DFA, they're either both accepting or both rejecting.

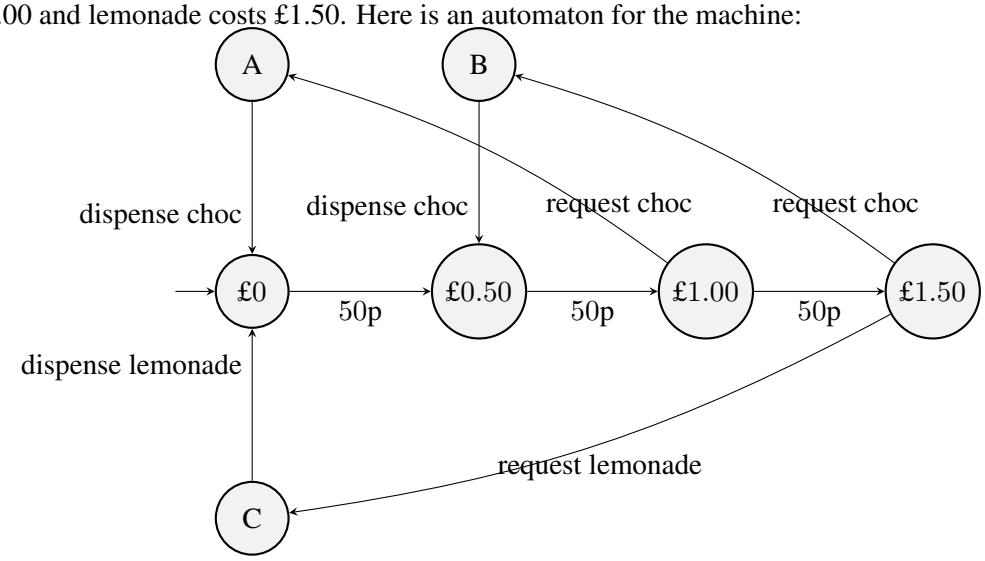
To make the isomorphism obvious, I drew the two diagrams the same way.

Because isomorphic automata have the same language (i.e. they accept the same words), we can leave the circles blank when drawing an automaton. You might like to imagine that each circle is filled with its coordinates on the page. However, if we want to refer to specific circles, it is helpful to number them in some way.

3.3 Vending machines

The idea of a DFA was invented for a specific purpose: solving a language's matching problem. All a DFA can do is input letters and say whether the word that's been read is accepted or not. But for other purposes, there are other kinds of automaton.

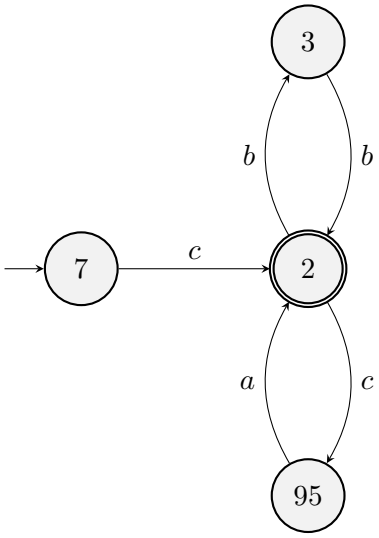
In the lobby there's a vending machine that can receive 50p coins, up to a maximum of £1.50. Chocolate costs £1.00 and lemonade costs £1.50. Here is an automaton for the machine:



Note that this automaton can *input* money and requests, and also *output* chocolate and lemonade. There are no accepting states, since recognizing words is not the purpose of this machine.

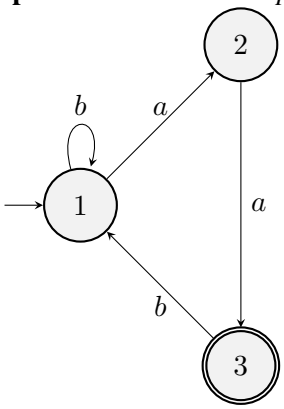
4 Partial deterministic automata

Look at the following automaton.



It's more efficient than the one at the start of Section 3.1. It is a *partial DFA*, meaning that δ is merely a partial function, i.e. it can sometimes be undefined.[1] As soon as a character cannot be input, the word is rejected. For example, the word `cbccabbabcababccabcc` is rejected after just three characters. (A partial DFA can also have no initial state, but then every word is rejected straight away, so this isn't very useful.)

Example 4 *What about this partial DFA?*



`aaab` *Y/N*
`aabaa` *Y/N*
`abaab` *Y/N*
`aa` *Y/N*

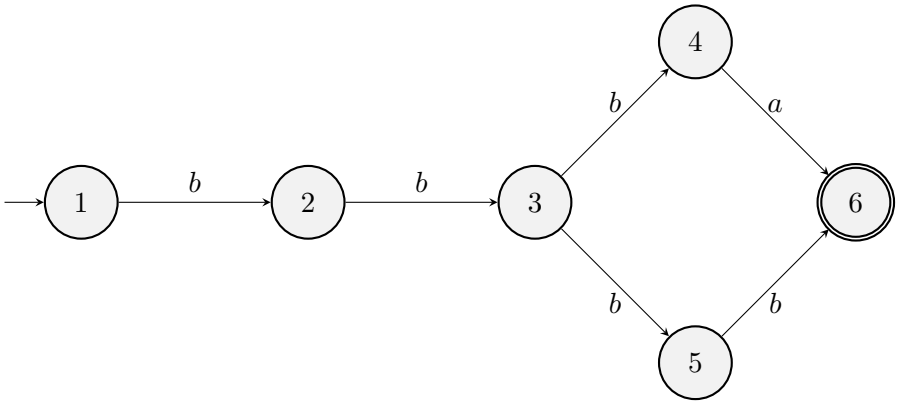
We can easily turn a partial DFA into a total DFA; just add an extra non-accepting state, called the *error state* (18 in the example). Transitions that are undefined in the partial DFA go to the error state. And every transition from the error state goes to the error state. (If the partial DFA has no initial state, the error state will be initial.)

5 Nondeterministic automata

5.1 The concept

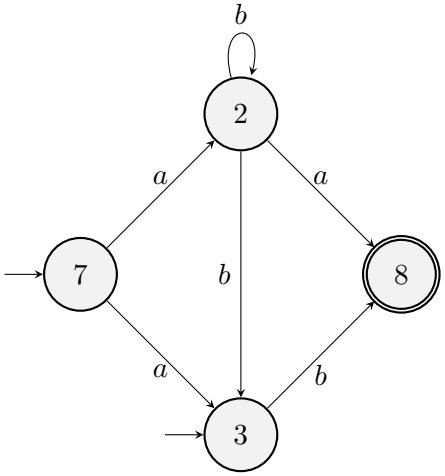
Sometimes it is difficult to obtain a DFA for a regexp, but we can more easily obtain a *nondeterministic* finite automaton (NFA). Here's an example, for the language `bb(ba|bb)`.

[1] Every function from A to B is a partial function from A to B, but not every partial function from A to B is a function from A to B.



A nondeterministic finite automaton (NFA) differs from a DFA in two respects. Firstly an NFA can have several initial states. Secondly, from a given state, when `a` (or any other character) is input, there can be several possible next states. Thus δ is a *relation* but not a function. The automaton chooses its initial state, and chooses what state to move to as it inputs a character. A word w is *acceptable* when there is **some** path from an initial state to an accepting state that goes through the characters of w.

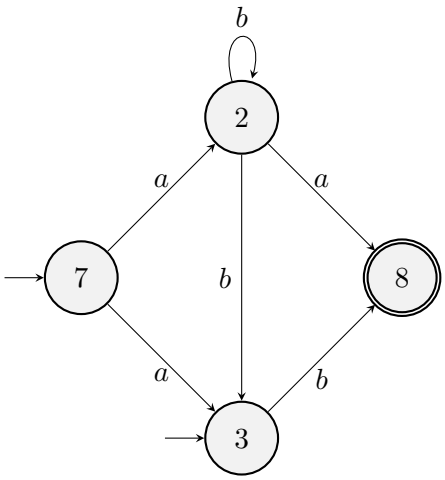
Example 5 *Here's a NFA for the alphabet* $\{\mathtt{a}, \mathtt{b}\}$. *Are these words acceptable?*



`abbb` *Y/N*
`abb` *Y/N*
ε *Y/N*
`abba` *Y/N*

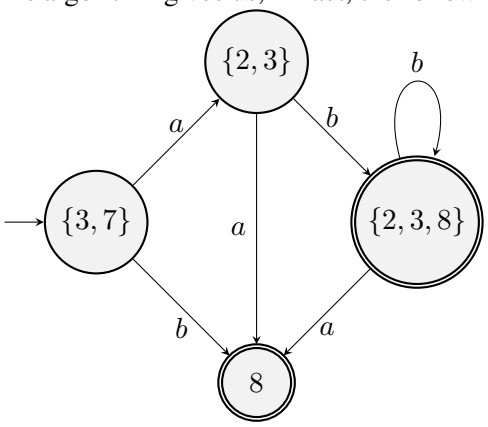
5.2 Determinizing an NFA: transforming an NFA into a DFA

An NFA is useless in practice: we want a program that always says Yes to a good word and No to a bad word, and an NFA doesn't do that. But we can *determinize* it, i.e. turn it into a DFA that recognizes the same language. To see how this works, look at the following example.



Let's think how to find out whether the word `abb` is acceptable. We can do it by trial and eror, but there's an algorithmic way: *keep track of the current set of possible states*. Initially the set of possible states is $\{3, 7\}$. After inputting `a`, the set of possible states is $\{2, 3\}$. After inputting `b`, the set of possible states is $\{2, 3, 8\}$. After inputting `b` again, the set of possible states is $\{2, 3, 8\}$. We have reached the end of the word, and we note that one of the currently possible states, viz. 8, is accepting. Therefore the word `abb` is acceptable.

This algorithm gives us, in fact, the following DFA:



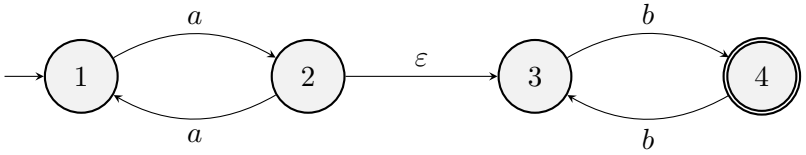
The "states" of the DFA are *sets* of states of the NFA. The initial "state" is the set of all the initial states of the NFA. A "state" is accepting when it contains an accepting state of the NFA. From a given "state", when we input a character, we collect all the possible next states. This process is called *determinization*.

We see that a word w is accepted by the DFA iff it's acceptable to the NFA. Therefore they represent the same language.

6 ε-transitions

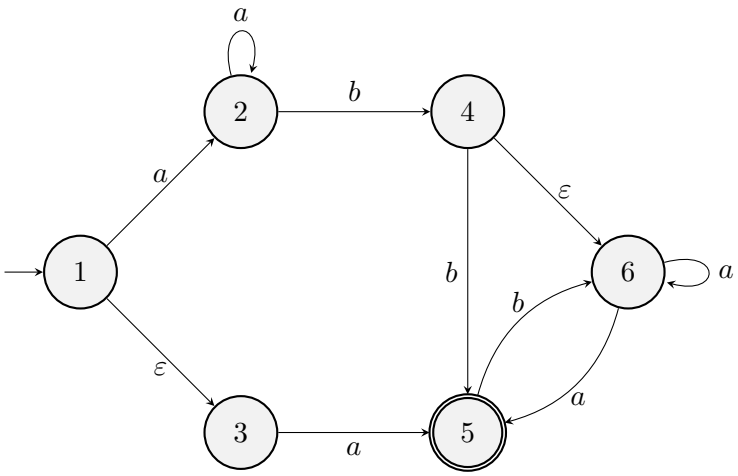
6.1 The concept

Sometimes even an NFA is difficult to obtain, but we can obtain an automaton that spends some time thinking. As it thinks, it moves from one state to another **without inputting any character**. Here's an example, for the regexp $\texttt{a}(\texttt{aa})^*\texttt{b}(\texttt{bb})^*$.



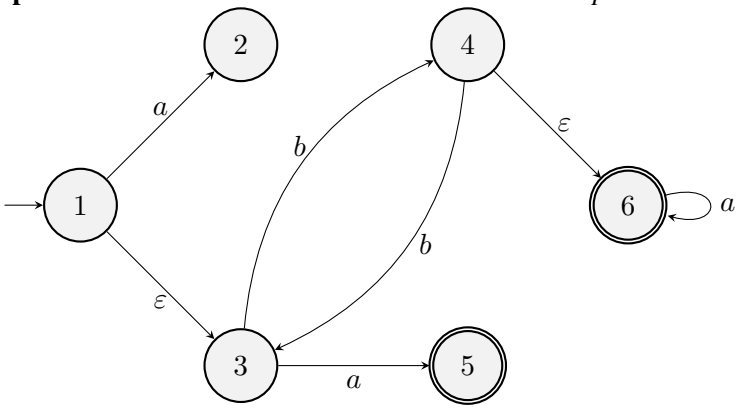
We call this a *nondeterministic automaton with ε-transitions* or εNFA for short. A word w is acceptable when there is some path from an initial state to an accepting state that goes through the characters of w, padded with ε transitions.

Example 6 *Here is a εNFA. Are these words acceptable?*



`aba` *Y/N*
`ab` *Y/N*
`aaabb` *Y/N*
`a` *Y/N*

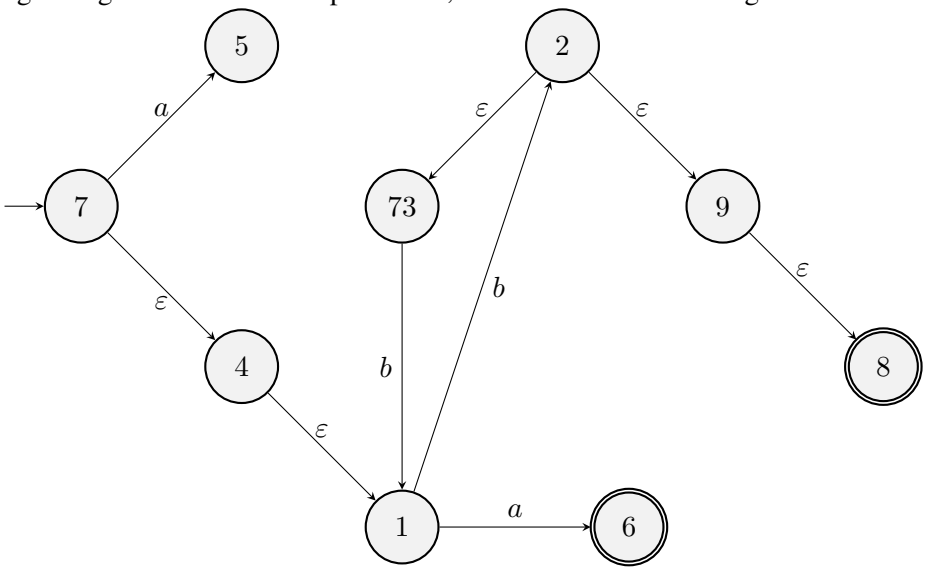
Example 7 *Here's another εNFA. Are these words acceptable?*



`a` *Y/N*
`aba` *Y/N*
`bbb` *Y/N*
`bbba` *Y/N*
`ab` *Y/N*

6.2 Removing ε-transitions

Happily, it's possible to remove the ε-transitions from a εNFA, i.e. convert it to an NFA that recognizes the same language. To get the idea for this procedure, let's look at the following εNFA.



The word `bbb` is acceptable, because of the following path:

$$7 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 1 \xrightarrow{b} 2 \xrightarrow{\varepsilon} 73 \xrightarrow{b} 1 \xrightarrow{b} 2 \xrightarrow{\varepsilon} 9 \xrightarrow{\varepsilon} 8$$

This path consists of the following pieces:

$7 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 1 \xrightarrow{b} 2$ is a *slow* b-transition from 7 to 2;

$2 \xrightarrow{\varepsilon} 73 \xrightarrow{b} 1$ is a *slow* b-transition from 2 to 1;

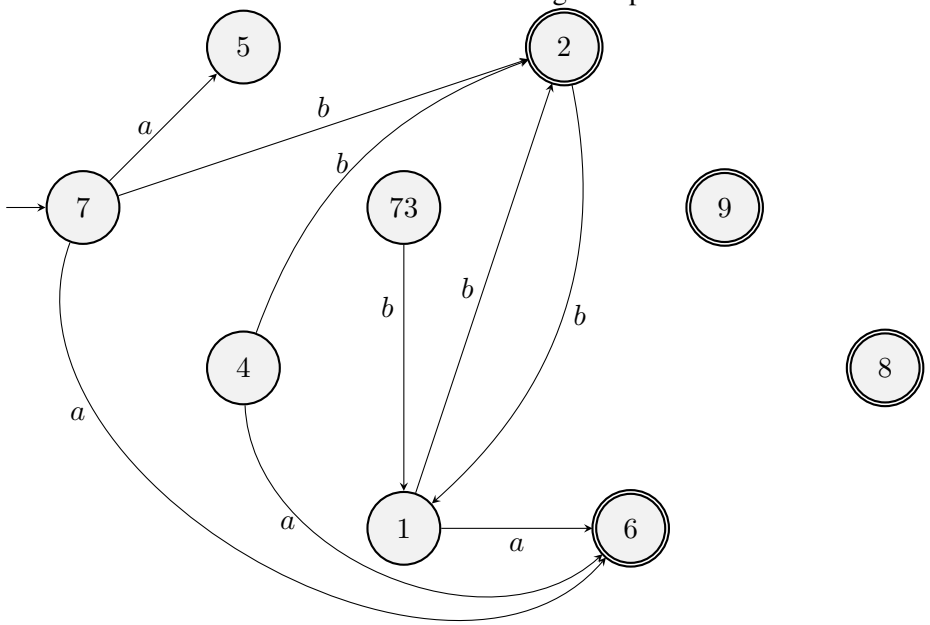
$1 \xrightarrow{b} 2$ is a *slow* b-transition from 1 to 2;

$2 \xrightarrow{\varepsilon} 9 \xrightarrow{\varepsilon} 8$ is *slowly accepting*.

You can see that this path consists of three *slow* `b`*-transitions* followed by *slow acceptance*. A slow `b`-transition consists of several (zero or more) ε-transitions, culminating in a `b`-transition. Slow acceptance consists of several

ε-transitions, culminating in an accepting state.

Now let's see how to remove the ε-transitions to give a plain NFA.



The automaton looks similar to before: the states are the same and the initial state is the same. The difference is that the transitions you see here are the slow transitions, and the accepting states you see here are the slowly accepting states.

Clearly we can remove the unreachable states 8, 9 and 4. It's always acceptable to remove unreachable states, because this doesn't change the language of the automaton (i.e. the set of acceptable words).

7 Coming soon: Kleene's Theorem

Up to this point, we have seen examples of

- using a regexp to describe a language
- solving a matching problem on a DFA.

It turns out that these two things are closely connected.

Theorem 1 *(Kleene's Theorem) For a language $L \subseteq \Sigma^*$, the following are equivalent.*

1. *L is regular, i.e. it can be described by a regexp.*
2. *The matching problem for L can be solved by a DFA.*

This means that every regexp can be converted into a DFA, and vice versa. The first direction is more important, because it gives us a practical way to solve a matching problem for a regexp. We shall see how to do this next week.

Induction

1 Introduction

Induction is a powerful proof technique that is widely used in computer science and mathematics. It has many variations, and we shall look at some of them.

- Ordinary induction over $\mathbb{N}$.
- Course-of-values induction over $\mathbb{N}$.

2 Induction over $\mathbb{N}$

Imagine an infinite sequence of dominoes standing on an infinite table, with Domino $n+1$ standing just behind Domino n, and someone pushes Domino 0. Then Domino 0 falls, causing Domino 1 to fall, causing Domino 2 to fall, causing Domino 3 to fall... What about Domino $10^{10^{100}}$? It will eventually fall. Indeed it is obvious that *each domino will fall*.

This is the idea behind induction over $\mathbb{N}$. Let P be a property of natural numbers. Suppose that $P(0)$—this is called the *base case*. Suppose also that, for any natural number $\mathbb{N}$, the statement $P(n)$ implies $P(n+1)$—this is called the *inductive step*, and the hypothesis $P(n)$ is called the *inductive hypothesis*. From these two facts, we may conclude that *every* natural number satisfies P, even big ones like $10^{10^{100}}$.

Example. Let's prove $0 + \ldots + (n-1) = \frac{1}{2}(n-1)n$ by induction on $n \in \mathbb{N}$. Clearly this is true for $n = 0$, since the sum of no numbers is defined to be 0. Assuming it's true for n, let's show that it's true for $n+1$.

$$\begin{aligned}
0 + \cdots + ((n+1)-1) &= 0 + \cdots + (n-1) + n \\
&= \frac{1}{2}(n-1)n + n \quad \text{(by the inductive hypothesis)} \\
&= \frac{1}{2}n^2 - \frac{1}{2}n + n \\
&= \frac{1}{2}n^2 + \frac{1}{2}n \\
&= \frac{1}{2}n(n+1) \\
&= \frac{1}{2}((n+1)-1)(n+1) \quad \text{as required.}
\end{aligned}$$

3 Variations

Here are some variations. Let P be a property of natural numbers.

- Suppose we have proved that P holds for 0, 1 and 2, and also that, if it holds for n, $n+1$ and $n+2$, then it also holds for $n+3$. We now know that P holds for all natural numbers.
- Suppose we have proved that P holds for 1 and 3, and also that, if it holds for n and $n+2$, then it also holds for $n+4$. We now know that P holds for all odd natural numbers.
- Suppose we have proved that P holds for 1, and also that, if it holds for n, then it also holds for $2n$. We now know that P holds for every power of 2.

4 Course-of-values induction

When we give a proof by ordinary induction, the inductive step proves that $P(1)$ follows from $P(0)$, that $P(2)$ follows from $P(1)$, that $P(3)$ follows from $P(2)$, and so on. But surely, when proving $P(3)$, it should be acceptable to assume not just $P(2)$ but also $P(1)$ and $P(0)$. This thinking leads to *course-of-values induction* (also called "strong induction").

The principle is as follows. Let P be a property of natural numbers. Suppose that, for any natural number n, the statement $P(n)$ holds if P holds for all natural numbers less than n. (The latter assumption is called the *inductive hypothesis*.) This means that

- $P(0)$
- if $P(0)$, then $P(1)$
- if $P(0)$ and $P(1)$, then $P(2)$
- if $P(0)$ and $P(1)$ and $P(2)$, then $P(3)$
- etc.

From this fact, we may conclude that *every* natural number satisfies P, even big ones like $10^{10^{100}}$.

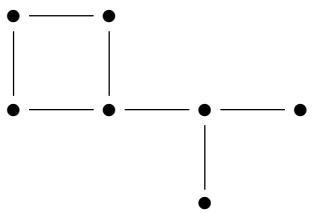
Example. The *merge sort* algorithm is the following recursively defined algorithm for sorting a list p.[1]

- If the length of p is 0 or 1, return p.
- If the length of p is $2k$ where $k > 0$, then sort the left part of length k, and sort the right part of length k, and merge the results.
- If the length of p is $2k+1$ where $k > 0$, then sort the left part of length k, and sort the right part of length $k+1$, and merge the results.

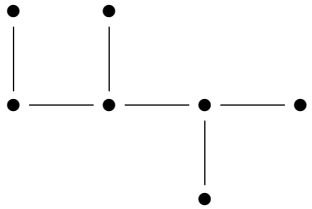
How do we know that this algorithm terminates, and returns a list that is a sorted version of p? By course-of-values induction on the length of the list. In each of the three cases, it is easy to see that the algorithm yields a sorted version of p, assuming that it works correctly on shorter lists.

[1]The version given here is intended to return the sorted version of the list. The version for sorting an *array* with in-place update is slightly different, but the idea is the same.

Example. An undirected graph G is *connected* when it has at least one vertex and there is a path between any two vertices.[2] Show that, if G is connected, then $V(G) \leqslant E(G) + 1$, where $V(G)$ is the number of vertices and $E(G)$ the number of edges. For example, if G is



then $V(G) = E(G) = 7$, whereas if G is



then $V(G) = 7$ and $E(G) = 6$.

Our proof proceeds by induction on $E(G)$. If $E(G) = 0$, then there can only be one vertex, so the property holds. So suppose that $E(G) > 0$. Pick an edge e from x to y. Let H be G with the same vertices, but e removed.

- Suppose there's a path p from x to y in H. Then H is connected. (For any nodes z and w, take a path in G from z to w, then replace e in this path by p.) Since $E(H) = E(G) - 1$, we apply the inductive hypothesis to H giving $V(H) \leqslant E(H) + 1$. So

$$\begin{aligned} V(G) &= V(H) \\ &\leqslant E(H) + 1 \\ &= E(G) \\ &< E(G) + 1 \end{aligned}$$

- On the other hand, suppose there's no path from x to y in H. Then H consists of two connected components, the part H_x that's connected to x and the part H_y that's connected to y. (For any vertex z, there's a path in G from z to x with no cycles. It either lies entirely in H, in which case $z \in H_x$, or consists of a path in H from z to y followed by the edge e, in which case $z \in H_y$. We can't have both $z \in H_x$ and $z \in H_y$, as this would give a path from x to y.) Since H_x and H_y are connected and have fewer edges than G, we can apply the inductive hypothesis to them, giving $V(H_x) \leqslant E(H_x) + 1$, and $V(H_y) \leqslant E(H_y) + 1$. Thus

$$\begin{aligned} V(G) &= V(H_x) + V(H_y) \\ &\leqslant E(H_x) + E(H_y) + 2 \\ &= E(H) + 2 \\ &= E(G) + 1 \end{aligned}$$

  as required.

[2] Some authors do not require the first condition. It makes no difference to this question.

Notice that, in this example, we don't quote the inductive hypothesis at the start of the inductive step. Instead, we put some work into obtaining two smaller graphs, and only then do we apply the inductive hypothesis to each of them.

Proving Kleene's theorem

1 Kleene's theorem

Previously we saw:

Theorem 1 *(Kleene's Theorem) For a language $L \subseteq \Sigma^*$, the following are equivalent.*

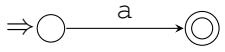
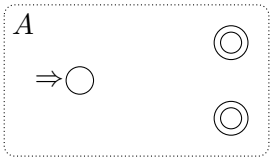
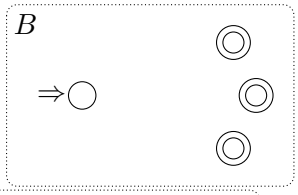
1. *L can be described by a regex.*
2. *The matching problem for L can be solved by a DFA.*

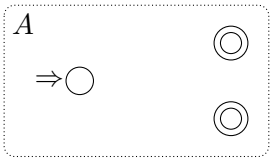
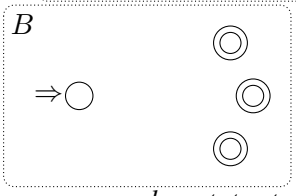
We are going to prove this theorem.

2 From regex to DFA

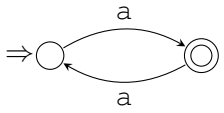
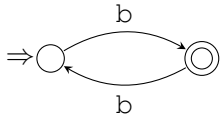
We shall see the forward direction: how to convert a regex into a DFA that recognizes the same language. We shall only study the proof of (1) $\Rightarrow$ (2), i.e. we'll see how to turn a regex into a DFA that recognizes the same language. As we saw, it suffices to construct an εNFA, because then we remove the ε-transitions to obtain an NFA, and lastly we determinize to obtain a DFA.

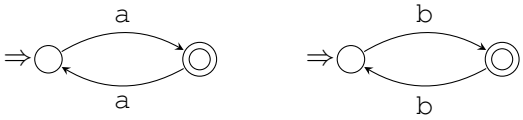
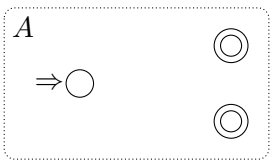
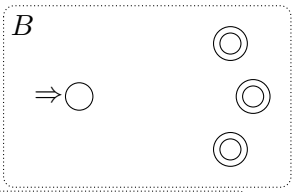
So we want to convert a regex into an εNFA that recognizes the same language.

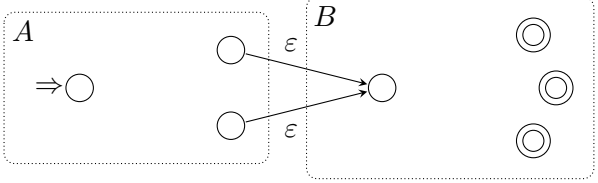
- The regex `a` is recognized by  and likewise if E is `b` or `c`.
- The regex ε is recognized by 
- If E_0 is recognized by  and E_1 by 

  then $E_0|E_1$ is recognized by  

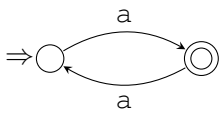
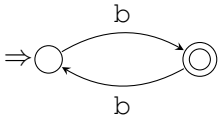
  (It's essential for the sets of states to be disjoint. If necessary, renumber states to achieve this.)

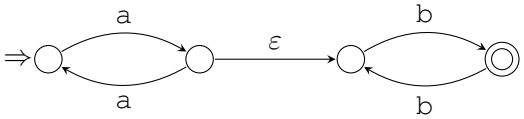
  For example, knowing that $\texttt{a}(\texttt{aa})^*$ is recognized by  and $\texttt{b}(\texttt{bb})^*$ by 

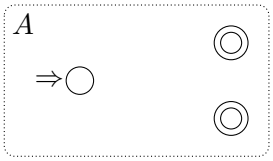
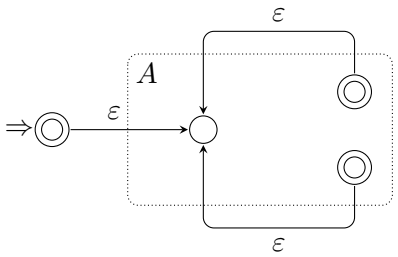
  we deduce that $\texttt{a}(\texttt{aa})^*|\texttt{b}(\texttt{bb})^*$ is recognized by 
- If E_0 is recognized by  and E_1 by 

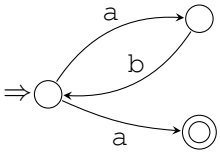
  then E_0E_1 is recognized by 

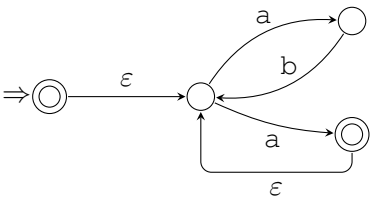
(Again the sets of states need to be disjoint.)

For example, knowing that $\texttt{a}(\texttt{aa})^*$ is recognized by  and $\texttt{b}(\texttt{bb})^*$ by 

we deduce that $\texttt{a}(\texttt{aa})^*\texttt{b}(\texttt{bb})^*$ is recognized by 

- If E is recognized by  then E^* is recognized by 

For example, $(\texttt{ab})^*\texttt{a}$ is recognized by 

so $((\texttt{ab})^*\texttt{a})^*$ is recognized by 

- Finally, $\emptyset$ is recognized by the empty automaton (the partial DFA with no states).

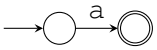
We thus see that for every regex E there is a DFA that recognizes the same language. This is a course-of-values induction on the *length* of E. In other words, we prove that the statement is true for E assuming that it is true for all *shorter* expressions.

In fact, all we need to assume is that the property holds for *subexpressions*. For example, to prove the property for E_0E_1, we need only assume that it's true for the subexpressions E_0 and E_1. This kind of argument often appears in computer science, and is called *structural* induction.

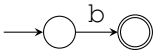
Negative viewpoint This argument can also be seen negatively. If there's a regex E that doesn't have an equivalent DFA, then there's a smaller regex E' that also doesn't, and therefore an even smaller one E'', and so on. But these are finite expressions, so this can't continue forever. Contradiction!

Let's follow the procedure described to convert the expression $((ab|ac)^*(abb)^*)^*$ into an automaton. To save on work, we'll obtain εNFAs all the way through, and then, right at the end, we'll remove the εs and determinize. We will not explicitly write name of the states for most of this process (we may consider the name of a state to coincide with its coordinates on the page).

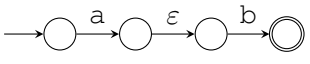
- a gives automaton



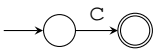
- b gives automaton



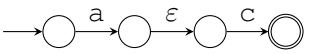
- ab gives automaton



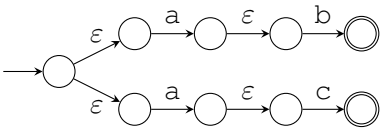
- c gives automaton



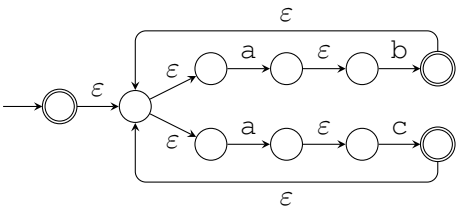
- ac gives automaton



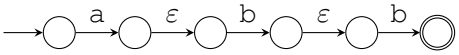
- $ab|ac$ gives automaton (with a minor simplification)



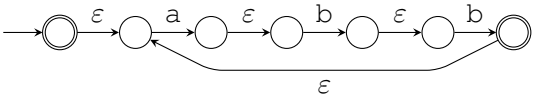
- $(ab|ac)^*$ gives automaton



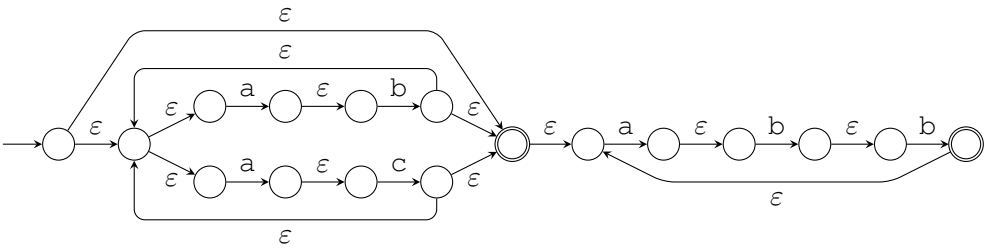
- abb gives automaton



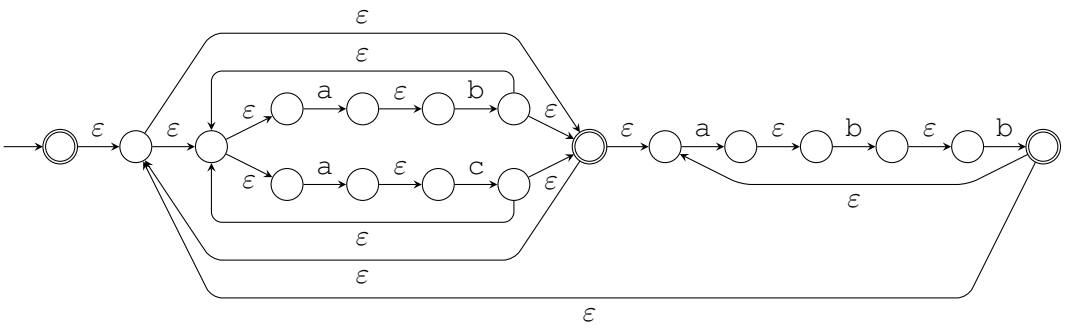
- $(abb)^*$ gives automaton



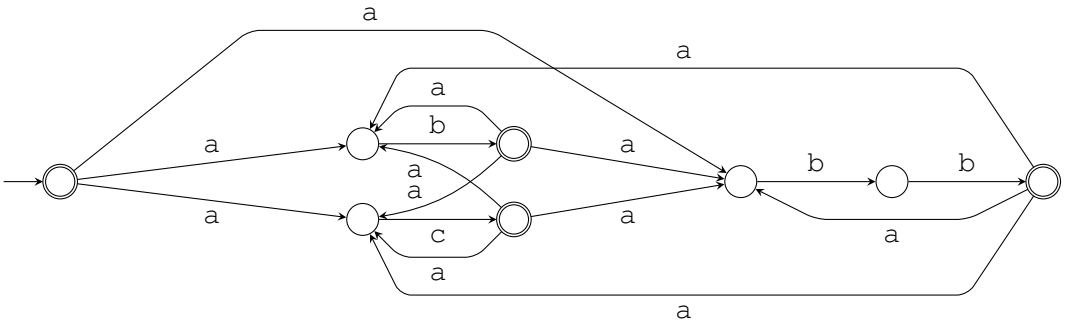
- $(ab|ac)^*(abb)^*$ gives automaton



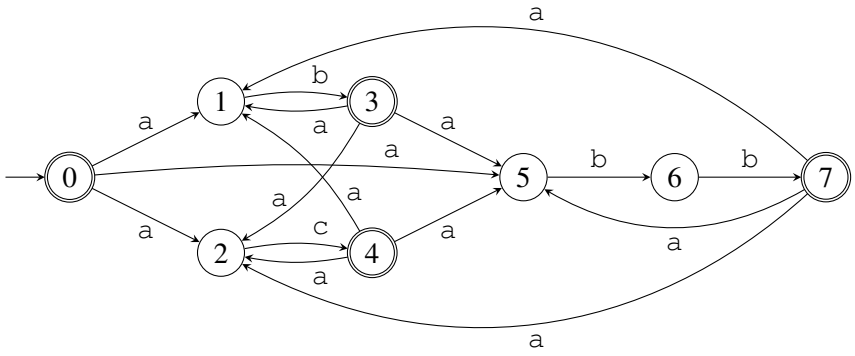
- $((ab|ac)^*(abb)^*)^*$ gives automaton



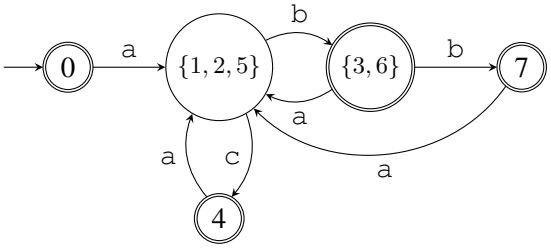
- Removing εs and all states that become unreachable gives the NFA



Before we determinize, let us tidy the automaton, and name the states



- Determinization gives the DFA

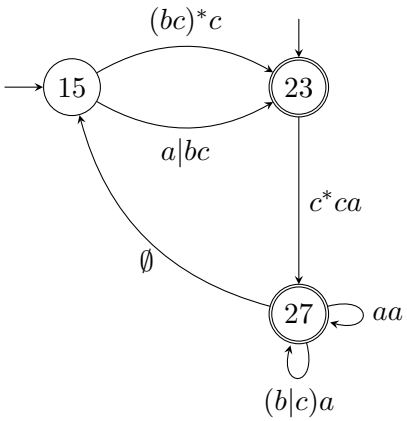


3 Generalized NFAs (not examinable)

Before explaining how to convert an automaton into a regex, let me first introduce the notion of a *generalized NFA*. This is a version of NFA where each arrow is labelled with a regex. There are finitely many states and arrows.

Given a word w, we start at an initial state and move from step to step. At each stage, we read in several characters at a time, forming a word x, and follow an arrow E that matches x. If we end on an accepting state, the word is accepted.

Example 1 *Here's a generalized NFA.*



Are these words acceptable?

`cccaaa` *Y/N*
`aacca` *Y/N*
`acaca` *Y/N*

Note that any εNFA is also a generalized NFA. Also note that any $\emptyset$-labelled arrow can be removed without changing the language.

4 From Generalized NFA to regex (not examinable)

We convert a generalized NFA to a regex in several stages. Note that these operations do not change the language of the automaton, i.e. which words are acceptable.

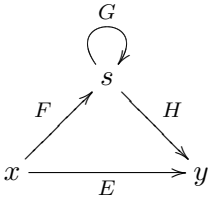
1. Combine any two distinct arrows $s \overset{E}{\underset{F}{\rightrightarrows}} t$ into an arrow $s \xrightarrow{E|F} t$. Continue until there is at most one arrow between any two states. In particular, there is at most one loop on each state.

2. For the sake of simplicity, we would like an automaton with exactly one arrow between any two states. To achieve this, wherever there is no arrow $x \to y$, we insert an $\emptyset$-labelled arrow. In particular, when there is no loop on x, we insert an $\emptyset$-labelled loop.

3. Add in two new states: a Start state, which becomes the sole initial state, and an End state, which becomes the sole accepting state. Connect Start to each old state s by an arrow labelled with ε if s was initial, and with $\emptyset$ otherwise. Connect each old state s to End by an arrow labelled by ε if s was accepting, and with $\emptyset$ otherwise. Connect Start to End by an $\emptyset$-labelled arrow.

   Note that there is now exactly one arrow between every pair of states, *except* that no arrow goes into Start, and no arrow comes out of End.

4. Next we remove the old states one by one. (The order of removal doesn't matter.) When we remove a state s, we remove all the arrows to it and from it, and also adjust the labels on all the other arrows. Specifically, if we had

   where neither x nor y is equal to s (but possibly $x = y$), then, after the removal of s, the new label on $x \to y$ will be $E|FG^*H$. It's worth noting that $\emptyset^*$ is equivalent to ε.

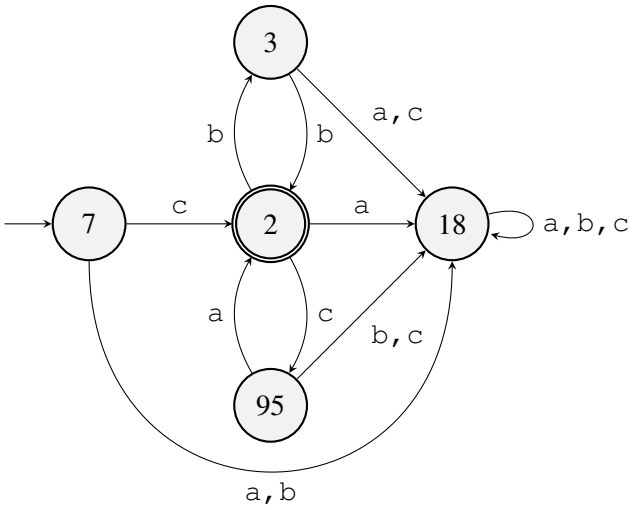
5. When there are no old states left, read off the label on Start $\to$ End. This is a regex that's equivalent to the automaton we started with.

Applications of Kleene's Theorem

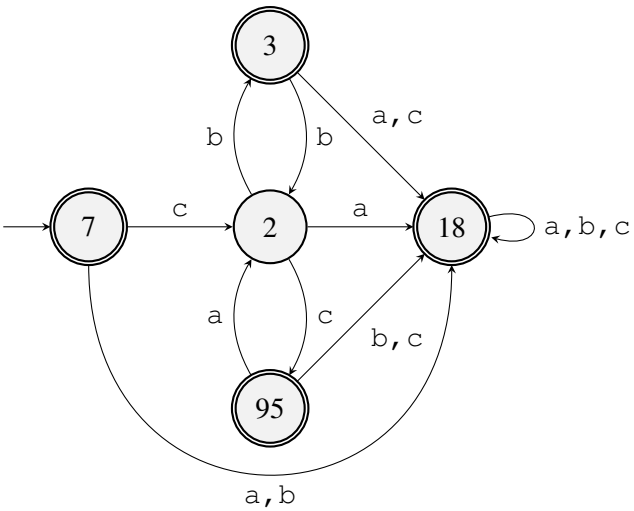
Recall that a *language* L is a set of words, i.e., a subset of Σ^*. We say that it's *regular* when it's the language of a regex. By Kleene's theorem, this is equivalent to being the language of a DFA. In particular, we have shown how to algorithmically transform a regular expression first to an ϵNFA, then to an NFA (by removing ϵ-transitions), and then to a DFA (by determizing).

1 Complementation

The complement of L, written $\overline{L}$ is the set of all words that are not in L. How can we show that the complement of a regular language is regular? This isn't obvious if we think about regexes. But using (total) DFAs, it is clear: just replace accepting states by non-accepting ones and vice versa.[1] For example, for the alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, we know that $\mathtt{c(bb|ca)}^*$ is recognized by



so its complement is recognized by



[1] Note that this does not work for partial DFAs, so adding an error state is essential!

We know that complementation satisfies some laws:

$$\overline{L \cap M} = \overline{L} \cup \overline{M}$$
$$\overline{L \cup M} = \overline{L} \cap \overline{M}$$
$$\overline{\overline{L}} = L$$

The first two laws are called *de Morgan's laws*, and the last one says that complementation is *involutive*. It follows that we can express intersection in terms of complementation and union:

$$L \cap M = \overline{\overline{L \cap M}} = \overline{\overline{L} \cup \overline{M}}$$

Therefore, if L and M are regular, so is $L \cap M$ (since the union of regular languages is regular).

For example, think of the password question on the first exercise sheet. You can make a DFA that determines whether a word has at least 3 characters, and another that determines whether it has a letter, and another that determines whether it has a digit. Then we obtain a DFA for the intersection of these languages. Kleene's theorem tells us that there's a corresponding regex.

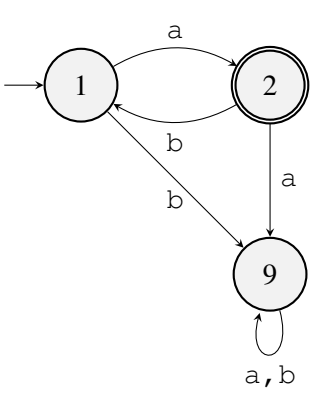
We have learned that the complement of a regular language is regular, and the intersection of two regular languages is regular.

2 Language equivalence

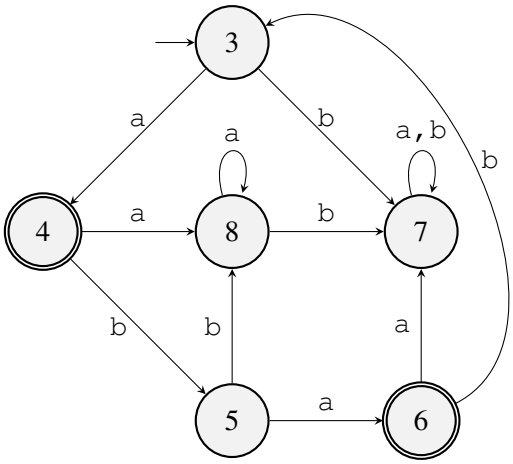
Recall that two regexes are called *language equivalent*, if they define the same language, e.g., `a(b|c)` and `ab|ac` are language equivalent. It's not obvious how to test whether two regexes are language equivalent, i.e., whether they define the same language, but it is easier to test whether two DFAs are language equivalent. Note that the method, we will present here, doesn't work for partial DFAs, so you should add an error state if needed.

Here's a suggestion: Let's build an automaton-like diagram consisting of pairs (x, y) where x is a state of the first automaton and y one of the second.

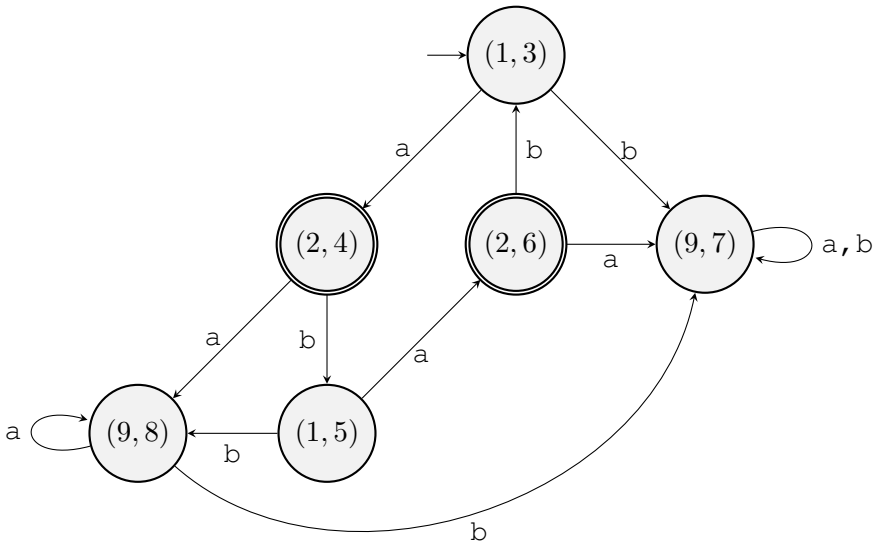
Start at the initial state of each automaton. If one is accepting and the other rejects, then the automata are not language equivalent (since one accepts ε and the other doesn't). If they both accept or both reject then see what pair of states we transition to by inputting `a`, and what pair by inputting `b`. And we carry on forever. For example, comparing the automata



and

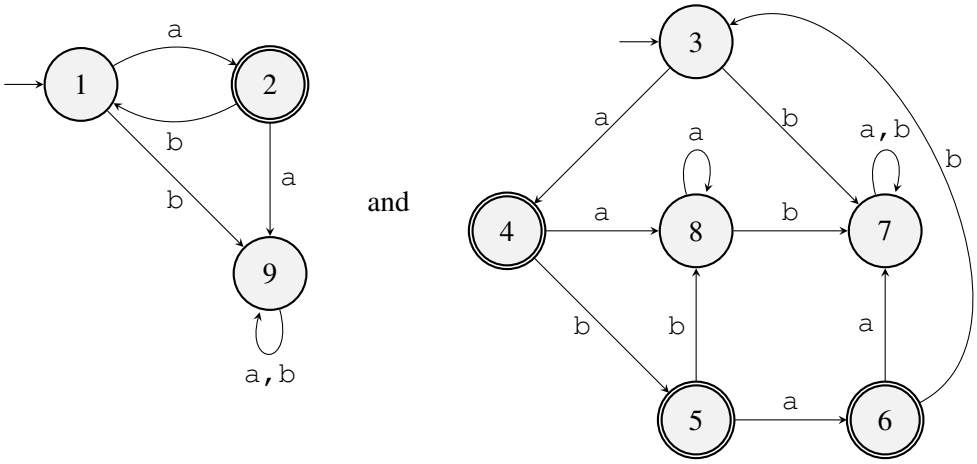


leads to the following:



In this example, we see that each pair of states consists of either two accepting states or two rejecting states. So the two automata are language equivalent.

Now let's compare



Note that the first automaton is the same as above, and the second differs only in which states are accepting, so the general shape of the resulting diagram will be the same as above. But, unlike above, in this case, when we hit the state $(1, 5)$, we see that state 1 is rejecting in the left automaton, but state 5 is accepting in the right automaton. So these DFAs are inequivalent. More precisely, we may observe that this state is reached after reading the word `ab`, hence the word `ab` is accepted by the left automaton, and rejected by the right automaton.

Either way, the procedure will always stop, since there are only finitely many pairs of states.

Exercise 1. Returning to unions and intersection of regular languages. We have seen how to see whether two DFAs are language equivalent on this diagram with pairs of states. Assume that it is not the case. How can you assign which pairs of states are accepting so that the diagram results with an automaton that (A) recognises the intersection of the two languages, (B) recognises the union of the two languages?

3 Minimal automata

As we know, there are several automata that recognise the same language. Are some of them better than others? One measure is size: we may try to make the automaton 'as small as possible'.

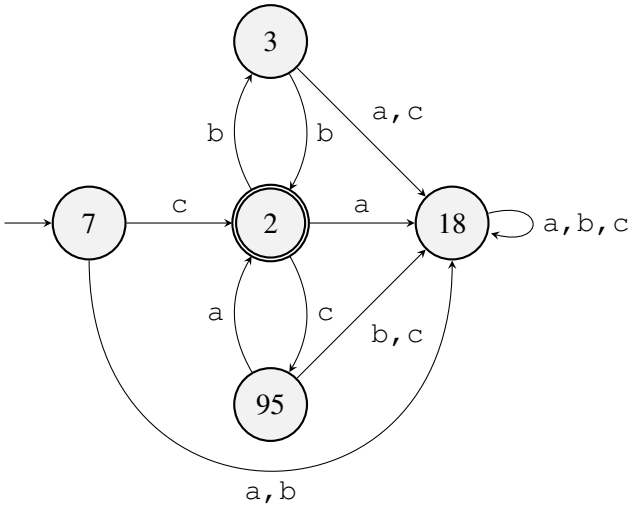
An automaton is minimal if it cannot be made smaller either by removing states, or collapsing equivalent states.

Definition 1. A DFA is said to be *minimal* when it has the following two properties.

1. Each of its states x is *reachable*. This means that there's a path from the initial state to x.
2. Any two distinct states x, y are *inequivalent*. This means that there's a word that x accepts (i.e., that leads from x to an accepting state) but y rejects, or vice versa.

The second item is equivalent to saying that the two automata obtained from the original one by changing the initial state to x and y, respectively, are not language equivalent.

For example, look at this DFA.



We show that it's minimal, as follows.

- 7 is reachable via ε, and 2 via `c`, and 3 via `cb`, and 95 via `cc`, and 18 via `ca`.
- 2 is inequivalent to the other states, since 2 accepts ε and the other states reject it.
- 3 is inequivalent to the other states, since 3 accepts `b` and the other states reject it.
- 95 is inequivalent to the other states, since 95 accepts `a` and the other states reject it.
- 7 is inequivalent to the other states, since 7 accepts `c` and the other states reject it.

Any automaton which is not minimal can be reduced to a smaller automaton either by removing unreachable states, or collapsing equivalent states. We may collapse two (or more) equivalent states by the following procedure: First choose one of the states to keep, let us call it x. Then one-by-one pick another equivalent state y, redirect all incoming transition to x and remove the state and all outgoing transitions. To obtain a minimal automaton, repeat the above while they are any unreachable states or pairs of equivalent states.

It should be noted that there are many algorithmic methods of minimising automata, and checking pairs of state for equivalence is by far not the most efficient one.

Fact. The minimal automaton accepting a given regular language is unique up to isomorphism. In other words, two minimal automata are language equivalent if and only if they are isomorphic.

Non-Regular Languages

1 Intermezzo: Countability

Finite sets may have different sizes (formally, size of a set is called *cardinality*). What might be less intuitive is that infinite sets come in many different sizes as well — there are smaller and bigger 'infinite cardinalities'. For now, we will distinguish the smallest infinite sets, which we will call *countably infinite*, from any larger set which we will call *uncountable*.

Definition 1. A set X is *countable* if there you can number its elements with natural numbers, so that no element is missed, i.e.,

$$X = \{x_0, x_1, x_2, x_3, \dots\}$$

where for each $x \in X$, there is $n \in \mathbb{N}$ with $x_n = x$.

If the set X is infinite, we may achieve that by listing each element of x exactly once, i.e., in such a way that $x_n \neq x_m$ for any $n \neq m$. (Countable sets include finite sets, so we usually explicitly say that a set is countable *and* infinite, countably infinite.)

Example 1. The sets $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$, the set of all primes, are all countably infinite. Any subset of $\mathbb{N}$ is countable, although not all of them are infinite. The sets $\mathbb{R}$, the set of all subsets of $\mathbb{N}$, the set of all subsets of $\mathbb{R}$ are all uncountable.

The set $\{0,1\}^*$ is countably infinite. This is since each word over $0, 1$ can be assigned a natural number by prepending 1 and interpretting the result in binary, e.g., the word 0110 is assigned the number 10110 which is the binary representation of 22. In order words, we may order all $0, 1$ words in a sequence as follows:

$$\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, \dots$$

(First by length, then lexicographically.) The same is true for any finite alphabet Σ: The set Σ^* of all words over a finite alphaber Σ is countable.

- The set of all regular expressions over a fixed alphabet Σ is countably infinite since each regular expression is a word over

  $$\Sigma \cup \{\epsilon, \emptyset, ^*, |, (,)\}.$$

  Hence, there are *countably many* regular languages!
- The set of all languages over Σ is *uncountable*! It has the same cardinality as the set of all subsets of $\mathbb{N}$.

2 Non-regular languages

We know that some languages are not regular, because there are only countably many regexps and uncountably many languages. But are there any *useful* non-regular languages? The answer is Yes. Here is an example.

Suppose a word is built up from open brackets, written `a`, and closed brackets, written `b`, and we want to know whether it's well-bracketed. This is a commonly arising problem; for example, when you write a Java program on Eclipse, Eclipse checks whether your brackets match correctly. This can be done using a stack. When you read `a`, you push a pebble onto the stack, and when you read `b` you pop the pebble off the stack. If you reach the end of the word, and the stack is empty, then you know the word is well-bracketed. But if it's not empty, or you read `b` when the stack is empty, the word is not well-bracketed.

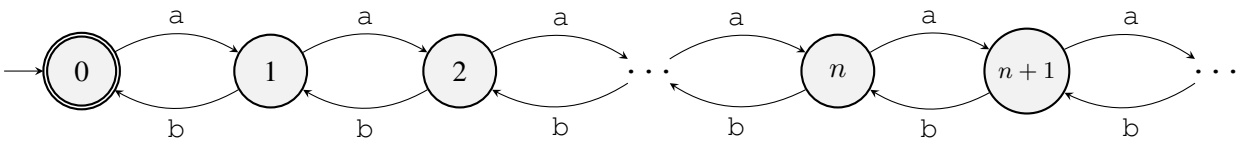
Let's think for a moment about your computer. It has a finite memory, and therefore only finitely many states. It can try to run the above program, allocating part of its memory to represent the stack. But if the word begins with a very large number of `a`'s, the stack will overflow.

Can your computer run a program that works for *all* words? The program should read in a word, letter by letter, and then announce whether or not the word is well-bracketed.

One solution is to use an external stack. That way, even though your computer has only finite memory, there's no limit on the amount of memory that the stack can occupy. But what if you don't have access to external memory? Can you install such a program on your computer? The answer is No. We shall now prove this.

To put this more technically, we shall prove that the set L of well-bracketed words is not regular.

The above procedure can be described by the *infinite* automaton. Note each state corresponds to a possible state of the pebble stack: the state n corresponds to the stack of n pebbles. So, well-bracketed expression can be checked with an automaton with infinite states.

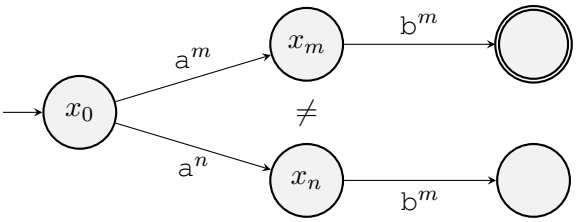


We will show that finitely many states are not enough. Suppose L is regular; then there's a DFA $(X, p, \delta, \mathsf{Acc})$ that recognizes it. Let's say that

- x_0 is the initial state
- x_1 is the state that we reach after reading `a`
- x_2 is the state that we reach after reading `aa`
- x_3 is the state that we reach after reading `aaa`
- etc.

(In the above infinte automaton, we have $x_n = n$ for all $n = 0, 1, \ldots$.) In summary, x_n is the state that we reach when we start at p and read $\mathtt{a}^n$. Now we're going to show that these states are all distinct, which implies that there are infinitely many states, a contradiction.

Suppose m and n are natural numbers with $m < n$. We want to show that $x_m \neq x_n$; we will show that the states x_m and x_n are not *equivalent*. If we start at x_m and read $\mathtt{b}^m$ we reach an accepting state, because $\mathtt{a}^m\mathtt{b}^m \in L$, but if we start at x_n and read $\mathtt{b}^m$ we reach a non-accepting state, because $\mathtt{a}^n\mathtt{b}^m \notin L$. Consequently, x_m and x_n can't be the same.



This is a general method to prove that a language is not regular (although it is not the only method of proving a language is irregular). For a different language, you'll need to adjust the definition of x_n, and adjust the way you show $x_m \neq x_n$ for $m < n$.

Exercise 1. Let the alphabet be $\{\mathtt{a}, \mathtt{b}\}$. Prove that the set of words in which `a` occurs more times than `b` is not regular.

Let's repeat the main point: a computer with finitely many states, and no access to external memory, cannot solve the matching problem for any non-regular language. For example, it can't determine whether a word (read in letter by letter) is well-bracketed.

Beyond Regular Languages

1 Context-free Languages

In the previous weeks, we have seen two different yet equivalent methods of describing regular languages i.e. automata and regular expressions. We have also seen other examples such as matching brackets, and understood that these are not regular. In this document, we present *context-free grammars*, a more powerful method of describing languages. The set of languages that can be generated using context-free grammars are known as *context-free languages*.

[Note: parts of this handout are adapted from Sipser's book: "Introduction to the Theory of Computation".]

1.1 Context-free grammars

We begin with an example. The alphabet is

$$\Sigma \;=\; \{+, \times, (,), 3, 5, \text{if}, \text{then}, \text{else}, \text{and}, >\}$$

(Perhaps we should call the elements of Σ "tokens" rather than "characters".) Our language L is going to be the set of all integer expressions. For example

$$\text{if } 3 > (3 + 5) \text{ then } 5 \text{ else } 3$$

is in L, but $3 > (3 + 5)$ is not. Here is a *context-free grammar* describing L.

$$\begin{array}{rcl} A & ::= & 3 \\ A & ::= & 5 \\ A & ::= & A + A \\ A & ::= & A \times A \\ A & ::= & (A) \\ A & ::= & \text{if } B \text{ then } A \text{ else } A \\ B & ::= & A > A \\ B & ::= & B \text{ and } B \\ B & ::= & (B) \\ \text{Start:} & & A \end{array}$$

We can write this in a more concise form:

$$\begin{array}{rcl} \Rightarrow A & ::= & 3 \mid 5 \mid A + A \mid A \times A \mid (A) \mid \text{if } B \text{ then } A \text{ else } A \\ B & ::= & A > A \mid B \text{ and } B \mid (B) \end{array}$$

which is called BNF (Backus-Naur Form). A and B are called *nonterminals* while the characters in Σ are called *terminals*. Each line with $::=$ is called a *production*, and it tells us how to replace a nonterminal with a string of terminals and nonterminals.

Formally, a context-free grammar is a 4-tuple (V, Σ, R, S), where:

1. V is a finite set called the **variables** or **non-terminals**,
2. Σ is a finite set, disjoint from V, called the **terminals**,
3. R is a finite set of **rules** or **productions**, with each rule consisting of a variable and a string of variables and terminals, and
4. S $\in$ V is the start variable.

Here is a derivation of the above term.

$$\begin{aligned}
\dot{A} &\rightsquigarrow \text{if } B \text{ then } \dot{A} \text{ else } A \\
&\rightsquigarrow \text{if } B \text{ then } 3 \text{ else } \dot{A} \\
&\rightsquigarrow \text{if } \dot{B} \text{ then } 3 \text{ else } 5 \\
&\rightsquigarrow \text{if } A{>}\dot{A} \text{ then } 3 \text{ else } 5 \\
&\rightsquigarrow \text{if } \dot{A} > (A) \text{ then } 3 \text{ else } 5 \\
&\rightsquigarrow \text{if } 3 > (\dot{A}) \text{ then } 3 \text{ else } 5 \\
&\rightsquigarrow \text{if } 3 > (A + \dot{A}) \text{ then } 3 \text{ else } 5 \\
&\rightsquigarrow \text{if } 3 > (\dot{A} + 5) \text{ then } 3 \text{ else } 5 \\
&\rightsquigarrow \text{if } 3 > (3 + 5) \text{ then } 3 \text{ else } 5
\end{aligned}$$

Note the principles:

- We begin with the Start nonterminal.
- At each step we replace a nonterminal (indicated with a dot) by a string of terminals and nonterminals according to one of the productions in the grammar.
- At the end, we have the desired word in Σ^*.

The grammar is called "context free" because you can apply a production to any nonterminal regardless of the other symbols in the string. The set of strings that can be produced or generated from a grammar is known as the **language of this grammar**, and can be written as L(G). A language produced by a context-free grammar is known as **Context-free Language**(CFL).

1.2 Leftmost and Rightmost Derivations

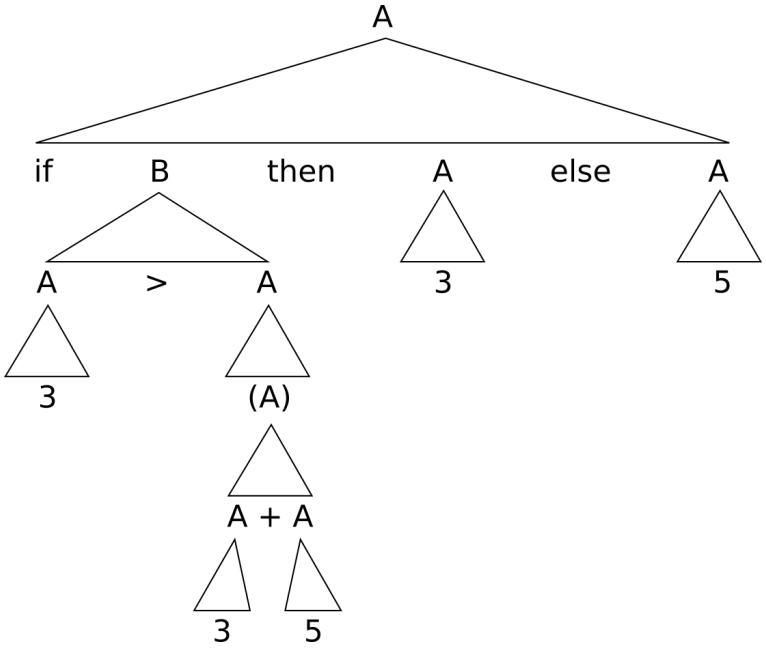
The above derivation jumps all over the word. The following performs the same replacements, but it is a *leftmost* derivation, meaning that at each step the nonterminal replaced is the leftmost one.

$$\begin{aligned}
\dot{A} &\rightsquigarrow \text{if } \dot{B} \text{ then } A \text{ else } A \\
&\rightsquigarrow \text{if } \dot{A} > A \text{ then } A \text{ else } A \\
&\rightsquigarrow \text{if } 3 > \dot{A} \text{ then } A \text{ else } A \\
&\rightsquigarrow \text{if } 3 > (\dot{A}) \text{ then } A \text{ else } A \\
&\rightsquigarrow \text{if } 3 > (\dot{A} + A) \text{ then } A \text{ else } A \\
&\rightsquigarrow \text{if } 3 > (3 + \dot{A}) \text{ then } A \text{ else } A \\
&\rightsquigarrow \text{if } 3 > (3 + 5) \text{ then } \dot{A} \text{ else } A \\
&\rightsquigarrow \text{if } 3 > (3 + 5) \text{ then } 3 \text{ else } \dot{A} \\
&\rightsquigarrow \text{if } 3 > (3 + 5) \text{ then } 3 \text{ else } 5
\end{aligned}$$

Similarly, we can have a *rightmost* derivation of the above, meaning that at each step the non-terminal replaced is the rightmost one.

$$\begin{aligned}
\dot{A} &\rightsquigarrow \text{if } B \text{ then } A \text{ else } \dot{A} \\
&\rightsquigarrow \text{if } B \text{ then } \dot{A} \text{ else } 5 \\
&\rightsquigarrow \text{if } \dot{B} \text{ then } 3 \text{ else } 5 \\
&\rightsquigarrow \text{if } A > \dot{A} \text{ then } 3 \text{ else } 5 \\
&\rightsquigarrow \text{if } A > (\dot{A}) \text{ then } 3 \text{ else } 5 \\
&\rightsquigarrow \text{if } A > (A + \dot{A}) \text{ then } 3 \text{ else } 5 \\
&\rightsquigarrow \text{if } A > (\dot{A} + 5) \text{ then } 3 \text{ else } 5 \\
&\rightsquigarrow \text{if } \dot{A} > (3 + 5) \text{ then } 3 \text{ else } 5 \\
&\rightsquigarrow \text{if } 3 > (3 + 5) \text{ then } 3 \text{ else } 5
\end{aligned}$$

Each of these derivations can be summarized by the following *derivation tree*.



A derivation tree is also called a *parse tree*. You may see it written with the root at the bottom, or with several edges instead of a triangle.

Note the principles:

- At the root we place the Start nonterminal.
- Each triangle has a nonterminal above and a string of terminals and nonterminals below, following one of the productions in the grammar.
- At each leaf, we have a terminal.

The desired word appears by reading the leaves from left to right.

Let's look at a "Natural Language" example. The alphabet is

{ the, a, cat, dog, happy, tired, slept, died, ate, dinner, and, . }

The grammar is

Sentence	$\Rightarrow$ S	::=	C.
Clause	C	::=	NP VP | C and C
Noun phrase	NP	::=	Art N | dinner
Noun	N	::=	Adj N | cat | dog
Adjective	Adj	::=	happy | tired
Verb phrase	VP	::=	VI | VT NP
Intransitive verb	VI	::=	slept | died
Transitive verb	VT	::=	ate
Article	Art	::=	a | the

This grammar accepts "words" such as

the happy tired happy dog died and the cat slept.
the tired tired cat ate dinner.
dinner ate a happy dog.

Try writing derivations and derivation trees for these sentences.

2 The matching problem for a context free language

Given a context free grammar, is the matching problem decidable? In other words, is there some program

```
boolean f (string w) {
 ...
}
```

that, when given a word w over our alphabet, returns True if w is derivable and False otherwise? The answer is Yes; the CYK algorithm (which we shall not learn) is a way of doing this. But, for some grammars, it is not efficient (cubic complexity).

Happily, for certain kinds of grammar, there are efficient ways of solving this problem. When people design a grammar for a programming language, they try to design it to fit one of these special kinds.

A program that constructs a derivation tree for a given word (if possible) is called a *parser*. Tools such as Yacc and Antlr are called *parser generators*; you supply a grammar (which must be of the right kind) and the tool will produce an efficient parser. You'll learn more about parsing when you study Compilers.

3 Designing Context-free Grammars

The following example (taken from Sipser), showcases that in order to get the grammar for the language $\{0^n1^n|n \geq 0\} \cup \{1^n0^n|n \geq 0\}$, we first construct the grammar:

$$\Rightarrow S_1 ::= \quad 0S_11|\varepsilon$$

for the language $\{0^n1^n|n \geq 0\}$ and the grammar

$$\Rightarrow S_2 ::= \quad 1S_20|\varepsilon$$

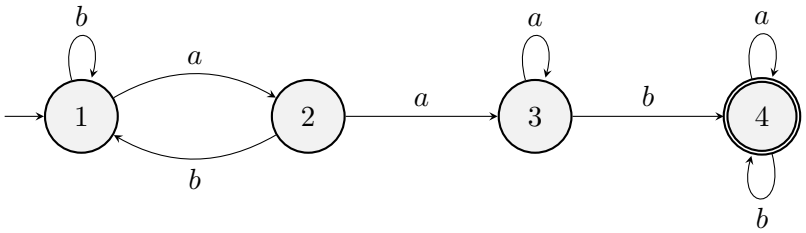
for the language $\{1^n0^n|n \geq 0\}$ and then add the rule $S ::= \quad S_1|S_2$ to give the grammar:

$$\begin{aligned} \Rightarrow S &::= \quad S_1|S_2 \\ S_1 &::= \quad 0S_11|\varepsilon \\ S_2 &::= \quad 1S_20|\varepsilon \\ \text{Start:} & \quad\quad S \end{aligned}$$

For regular languages, the task of constructing equivalent CFG is relatively easy. We can construct the DFA for the given language and then convert the DFA into an equivalent CFG as follows:

1. Create a variable R_i for each state q_i of the DFA.
2. Add the rule $R_i ::= \quad aR_j$ to the CFG, if $\delta(q_i, a) = q_j$ is a transition in the DFA.
3. Add the rule $R_i ::= \quad \varepsilon$, if q_i is an accepting state of the DFA.
4. Make R_0 the start variable of the grammar, where q_0 is the start state of the machine.

You can verify the resultant CFG and the fact that it generates the same language as the given DFA quite easily. Let's consider a DFA that accepts any string that contains the substring "aab".



1. We can make the variables R_1, R_2, R_3 and R_4, corresponding to the states of above DFA.
2. We add the rules using the pattern $R_i ::= \quad aR_j$ to the CFG, for each transition in the DFA.
3. Add the rule $R_4 ::= \quad \varepsilon$, as state 4 is an accepting state of the DFA.
4. Make R_1 the start variable of the grammar.

We get the following grammar after applying the above steps:

$$\begin{aligned} \Rightarrow R_1 &::= aR_2 \mid bR_1 \\ R_2 &::= aR_3 \mid bR_1 \\ R_3 &::= aR_3 \mid bR_4 \\ R_4 &::= aR_4 \mid bR_4 \mid \varepsilon \end{aligned}$$

We will let you verify the above CFG generates the same language that the DFA recognizes.

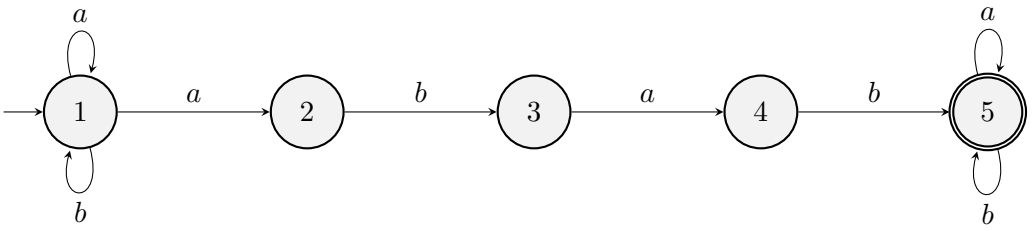
Let us given another grammar for arithmetic expressions. In this example, $G = (V, \Sigma, R, E)$, with $V = \{E, T, F\}$ and Σ is $\{3, 5, +, \times, (,)\}$ is shown below.

$$\begin{aligned} \Rightarrow E &::= E + T \mid T \\ T &::= T \times F \mid F \\ F &::= (E) \mid 3 \mid 5 \end{aligned}$$

In the above example, any time the symbol E appears, an entire parenthesized expression might appear recursively instead. To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear *i.e.* the $F ::= (E)$ production.

3.1 Test Your Understanding

1. Let's consider an NFA that accepts any string that contains the substring "abab".



   (a) Convert the above NFA into its equivalent total DFA.

   (b) Convert the resultant DFA in an equivalent CFG.

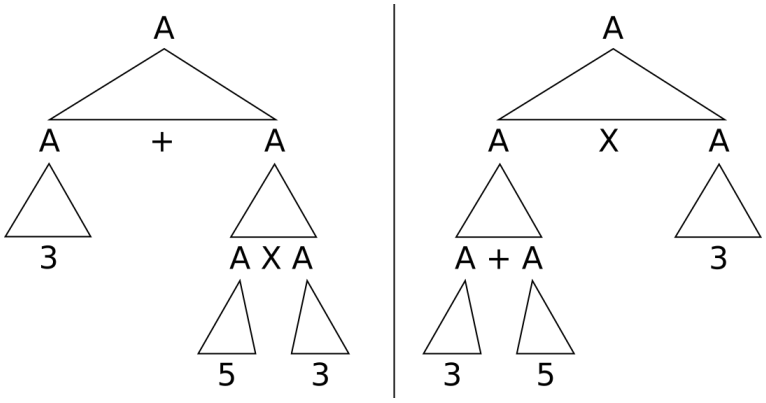
2. Give a context free grammar for the set of palindromes over the alphabet $\{\texttt{a}, \texttt{b}\}$.

4 Ambiguity

We have seen that one derivation tree can arise from several different derivations (although only one leftmost derivation), but that's not a serious problem. Much more serious is that a word can have more than derivation tree. For example, using the following grammar:

$$\Rightarrow A ::= A + A \mid A \times A \mid (A) \mid 3 \mid 5$$

the word $3 + 5 \times 3$ has derivation trees



The above grammar does not take into account the order of precedence of operators, that is, it does not ensure that $\times$ operation must be applied before $+$. In contrast, the following grammar generates exactly the same language, but every generated string has a unique derivation tree. Hence, this grammar is unambiguous, whereas the one above is ambiguous. The following grammar takes into account the *precedence* of operators, by moving the higher priority operations lower in the grammar e.g. the $\times$ operation comes later than $+$ operation.

$$\begin{aligned} \Rightarrow A &::= A + B \mid B \\ B &::= B \times C \mid C \\ C &::= (A) \mid 3 \mid 5 \end{aligned}$$

It is usually desirable to design a grammar to be unambiguous. Therefore it would be useful to have a program that, when we provide a context free grammar, tells us whether that grammar is ambiguous or not. But that is impossible: ambiguity of context free grammars is *undecidable*. (We shall not prove this.)

4.1 Test Your Understanding

1. Try deriving the string $3 + 5 \times 3$ in two different ways using leftmost derivation only, using the grammar given above.

2. Show that the following grammar is ambiguous. The alphabet is $\{\mathtt{a}, \mathtt{b}\}$.

$$\begin{aligned} \Rightarrow P &::= \varepsilon \mid Q\mathtt{a} \mid \mathtt{a}Q \\ Q &::= \mathtt{aa}P \mid \mathtt{b}R \\ R &::= Q\mathtt{a} \end{aligned}$$

5 Chomsky Normal Form (CNF)

In this section we are going to learn how to convert a CFG into a special form known as *Chomsky Normal Form*. It only allows rules of the following kind

$$\begin{aligned} A &::= BC \\ A &::= a \end{aligned}$$

where a is any terminal and A, B, and C are any variables - except that B and C may not be the start variable. In addition, there can be a rule $S ::= \varepsilon$, where S is the start variable.

Chomsky Normal Form has various uses, but one is especially noteworthy. This is the fact that using a grammar in this form, a derivation of a nonempty word involves $2n - 1$ steps, where n is the word's length. This can be proved using course-of-values induction on n.

And so, given a grammar G and a word w, we can test mechanically whether the word is accepted by G. First we convert G to Chomsky Normal Form (in the manner we shall see below). Then we write out all derivations of length $2n - 1$ and see if any of them work. Highly inefficient, and much worse than the CYK algorithm, but it does the job.

To convert any given CFG into CNF, use the steps outlined below:

1. We begin by introducing a new start symbol (variable) to the grammar.

2. In the second step, we remove all of the ε-rules of the form $A ::= \varepsilon$.

3. In the third step, we remove all of the unit productions of the form $A ::= B$.

4. We may need to patch-up/fix the grammar to make sure that it still produces the original language.

5. In the end, we will convert the remaining rules into proper form.

Let's convert the following CFG (taken from Sipser) into Chomsky normal form by using the conversion steps outlined above. We will go through the conversion process, step by step, and at each stage show you the new grammar along with the newly added rules shown in bold.

$$\begin{aligned} \Rightarrow S &::= ASA \mid aB \\ A &::= B \mid S \\ B &::= b \mid \varepsilon \end{aligned}$$

1. Add a new start variable S_0

$$\begin{aligned} \Rightarrow \boldsymbol{S_0} &\ \boldsymbol{::=} \quad \boldsymbol{S} \\ S &::= \quad ASA \mid aB \\ A &::= \quad B \mid S \\ B &::= \quad b \mid \varepsilon \end{aligned}$$

2a. Remove ε-rule $B ::= \varepsilon$

$$\begin{aligned} \Rightarrow S_0 &::= \quad S \\ S &::= \quad ASA \mid aB \mid \boldsymbol{a} \\ A &::= \quad B \mid S \mid \boldsymbol{\varepsilon} \\ B &::= \quad b \end{aligned}$$

2b. Remove ε-rule $A ::= \varepsilon$

$$\begin{aligned} \Rightarrow S_0 &::= \quad S \\ S &::= \quad ASA \mid aB \mid a \mid \boldsymbol{SA} \mid \boldsymbol{AS} \mid \boldsymbol{S} \\ A &::= \quad B \mid S \\ B &::= \quad b \end{aligned}$$

3a. Remove the unit rule $S ::= S$

$$\begin{aligned} \Rightarrow S_0 &::= \quad S \\ S &::= \quad ASA \mid aB \mid a \mid SA \mid AS \\ A &::= \quad B \mid S \\ B &::= \quad b \end{aligned}$$

3b. Remove the unit rule $S_0 ::= S$

$$\begin{aligned} \Rightarrow S_0 &::= \quad \boldsymbol{ASA} \mid \boldsymbol{aB} \mid \boldsymbol{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\ S &::= \quad ASA \mid aB \mid a \mid SA \mid AS \\ A &::= \quad B \mid S \\ B &::= \quad b \end{aligned}$$

3c. Remove the unit rule $A ::= B$

$$\begin{aligned} \Rightarrow S_0 &::= \quad ASA \mid aB \mid a \mid SA \mid AS \\ S &::= \quad ASA \mid aB \mid a \mid SA \mid AS \\ A &::= \quad \boldsymbol{b} \mid S \\ B &::= \quad b \end{aligned}$$

3d. Remove the unit rule $A ::= S$

$$\begin{aligned} \Rightarrow S_0 &::= \quad ASA \mid aB \mid a \mid SA \mid AS \\ S &::= \quad ASA \mid aB \mid a \mid SA \mid AS \\ A &::= \quad b \mid \boldsymbol{ASA} \mid \boldsymbol{aB} \mid \boldsymbol{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\ B &::= \quad b \end{aligned}$$

4. Convert the remaining rules into the proper form by adding additional variables and rules. The final grammar in Chomsky normal form.

$$\begin{aligned} \Rightarrow S_0 &::= \quad AC \mid DB \mid a \mid SA \mid AS \\ S &::= \quad AC \mid DB \mid a \mid SA \mid AS \\ A &::= \quad b \mid AC \mid DB \mid a \mid SA \mid AS \\ C &::= \quad SA \\ D &::= \quad a \\ B &::= \quad b \end{aligned}$$

5.1 Test Your Understanding

Convert the following CFG into an equivalent CFG in Chomsky normal form

$$\begin{aligned} \Rightarrow A &::= BAB \mid B \mid \varepsilon \\ B &::= 00 \mid \varepsilon \end{aligned}$$

6 Emptiness and Fullness

To test whether a CFG accepts *some* word, we mark each variable that is able to turn into a word. For example, if we see a production

$$A ::= BCaB$$

and we know that B and C can turn into a word, then A can too. Just repeat this until you can't go any further, and see whether the start variable can turn into a word.

Can we test whether a CFG accepts *every* word over the given alphabet? No, this is an undecidable problem. Therefore, it is undecidable whether two CFGs accept the same words.

7 Beyond context free languages

We know that not all languages are context free, because there are uncountably many languages, yet only countably many context free grammars. But are there useful examples of languages that are not context free? Yes. Here's an example.

When you write a program, it's essential that every variable is declared, because a program with an undeclared variable can't run. A compiler will always check that the code being compiled has this property. But the set of words with this property is not context free. (We won't prove this, but the basic problem is that a context free grammar requires the set of nonterminals to be finite.)

Complexity of Programs

1 Mathematical Preliminaries

1.1 Laws of probability

The probability of an event A is written $\mathbb{P}(A)$. It is an element of $\{x \in \mathbb{R} \mid 0 \leqslant x \leqslant 1\}$.

The basic laws of probability are as follows:

- An impossible event has probability 0.
- A certain event has probability 1.
- For an event A, we have $\mathbb{P}(\text{not } A) = 1 - \mathbb{P}(A)$. For example, suppose the probability that it's raining is $\frac{1}{3}$. Then the probability that it's not raining is $\frac{2}{3}$.
- For events A and B that are mutually exclusive, we have $\mathbb{P}(A \text{ or } B) = \mathbb{P}(A) + \mathbb{P}(B)$. For example, suppose the probability that it's raining is $\frac{1}{3}$ and the probability that it's sunny is $\frac{1}{5}$. If these are mutually exclusive events, then the probability that it's either raining or sunny is $\frac{8}{15}$.
- For events A and B that are independent, we have $\mathbb{P}(A \text{ and } B) = \mathbb{P}(A) \times \mathbb{P}(B)$. For example, suppose the probability that it's raining is $\frac{1}{3}$ and the probability that John is happy is $\frac{1}{5}$. If these are independent events, then the probability that it's raining and John is happy is $\frac{1}{15}$.

1.2 Important summations

Here are some summations that come up again and again, so make sure you know them.

$$\begin{aligned} 0 + 1 + 2 + \cdots + (n-1) &= \frac{1}{2}n(n-1) \\ 1 + 2 + 3 + \cdots + n &= \frac{1}{2}n(n+1) \\ 1 + b + b^2 + \cdots + b^{n-1} &= \frac{b^n - 1}{b - 1} \qquad (b \neq 1) \\ 1 + b + b^2 + \cdots + b^n &= \frac{b^{n+1} - 1}{b - 1} \qquad (b \neq 1) \end{aligned}$$

1.3 Upper and lower bounds

- It will take me at least an afternoon to clear my office. *Lower bound.*
- Clearing the office will take me a week at most. *Upper bound.*
- Building the new railway will cost no more than 70 billion pounds. *Upper bound.*
- For the café to be viable, we need at least 30 customers a day, maybe more. *Lower bound.*

Note that an upper bound gives a guarantee.

2 Running time of a program

2.1 Best, average and worst cases

Consider a sample problem, e.g. sorting (arranging items in order)

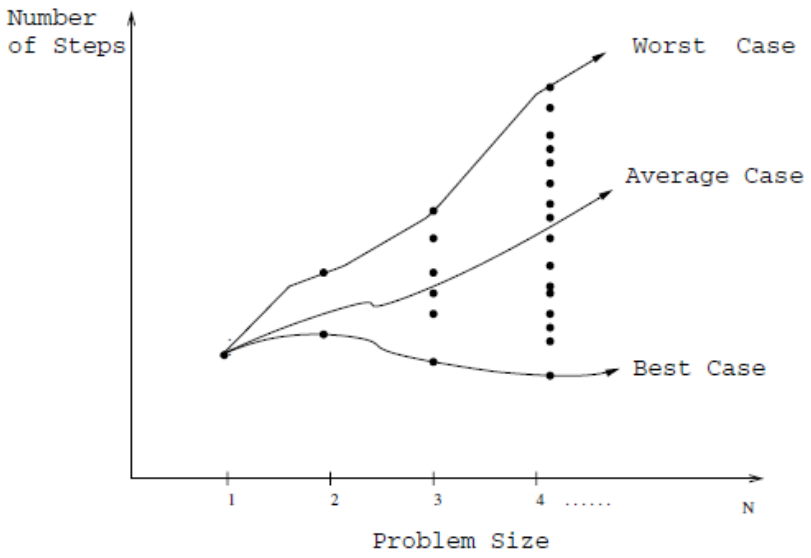


Figure 2.1: Best, worst, and average-case complexity

The above plot illustrates three functions:

- **Worst-case complexity**: It gives us an *upper bound* on the cost. It is determined by the *most difficult* input and provides a guarantee for all inputs.
- **Best-case complexity**: It gives us a *lower bound* on the cost. It is determined by the *easiest input* and provides a goal for all inputs.
- **Average-case complexity**: It gives us the *expected cost* for a random input. It requires a model for *random input* and provides a way to predict performance.

We will mainly focus on worst/average case complexity analysis.

Let's consider the following program that operates on an array of characters that are all `a` or `b` or `c`.

```
void f (char[] p) {
  elapse(1 second);
  for (nat i = 0; i<p.length(), i++) {
    if (p[i]=='a') {
      elapse(1 second);
    } else {
      elapse(2 seconds);
    }
    elapse (1 second);
  }
}
```

For an array of length 0, the running time is 1 second.
For an array of length 1, assuming `a`, `b`, `c` are equally likely:

Array contents	Probability	Time	Probability $\times$ time
`a`	$\frac{1}{3}$	$3s$	$1s$
`b`	$\frac{1}{3}$	$4s$	$1\frac{1}{3}s$
`c`	$\frac{1}{3}$	$4s$	$1\frac{1}{3}s$
Worst case: `b`		$4s$	
Average case			$3\frac{2}{3}s$

For an array of length 2, assuming `a`, `b`, `c` are equally likely and the characters are independent:

Array contents	Probability	Time	Probability $\times$ time
`aa`	$\frac{1}{9}$	$5s$	$\frac{5}{9}s$
`ab`	$\frac{1}{9}$	$6s$	$\frac{2}{3}s$
`ac`	$\frac{1}{9}$	$6s$	$\frac{2}{3}s$
`ba`	$\frac{1}{9}$	$6s$	$\frac{2}{3}s$
`bb`	$\frac{1}{9}$	$7s$	$\frac{7}{9}s$
`bc`	$\frac{1}{9}$	$7s$	$\frac{7}{9}s$
`ca`	$\frac{1}{9}$	$6s$	$\frac{2}{3}s$
`cb`	$\frac{1}{9}$	$7s$	$\frac{7}{9}s$
`cc`	$\frac{1}{9}$	$7s$	$\frac{7}{9}s$
Worst case: `bb`		$7s$	
Average case			$6\frac{1}{3}s$

Now consider an array of length n.

1. When does the worst case arise? (Just give one example.) What is its running time?

2. Assuming `a`, `b`, `c` are equally likely and the characters are independent, what is the average case running time?

Consider another example:

```
void g (char[] p){
  elapse(8 seconds);
  for (nat i=0; i<p.length(); i++){
    elapse(5 seconds);
    for (nat j=i; j<p.length(); j++){
      elapse(2 seconds);
    }
  }
}
```

3. What's the running time of `g` for an array of length 4?

4. What's the running time of `g` for an array of length n?

For this program, the running time for an array of given length is always the same.

2.2 Running time in terms of argument size

Running time is always expressed in terms of the *size* of the argument. Computer scientists use different definitions of size in different settings, but in this module, we always treat the argument as a word over Σ and its size is its length. (Remember that Σ is a finite set of size at least 2.)

For example, suppose we are studying the problem of sorting a list of natural numbers. Some computer scientists would take the size to be the length of the list, and treat comparison of two numbers as a single step, ignoring the fact that comparison of large numbers takes longer. But we shall take $\Sigma = \{0, 1, [,], ,\}$, and then, for example, represent the list $[5, 2, 6]$ as the word "[101,10,110]" of length 12. Alternatively, we can use base ten; but what we cannot do is to treat each natural number as a single character, because the alphabet must be finite.

Here are two widely used algorithms that are fast in the average case but slow in the worst case.

- Quicksort, for sorting an array.
- The simplex algorithm, for solving an optimization problem called "linear programming". In the worst case it is exponential (i.e. very slow), yet it is efficient in practice.

Which is more important, the worst case or the average case? Usually it is the average case that is more important; an occasional slow run doesn't matter so much if the program runs fast on average. Nevertheless it's certainly better if you can guarantee that your program always runs quickly. And there are situations where a slow run would be catastrophic.

3 What do we care about?

We don't usually work out the precise running time of a program; indeed we don't usually have the information needed to do so. Instead we work out the *complexity*, which is a kind of rough estimate of the running time that ignores two things: small arguments and constant factors.

To help us grasp the idea, let's meet four people, with different opinions on the subject of running time.

- **Little Tim** is the most discriminating. He cares about the time taken for inputs of all sizes, even small inputs.
- **Big Tim** is less discriminating than Little Tim. He cares only about the time taken for big inputs.
- **Constance** is less discriminating than Big Tim. She cares only about the time taken (for big inputs) up to a constant factor.
- **Polly** is the least discriminating. She cares only whether the time taken (for big inputs) is polynomial in the size of the input.

4 Small arguments

4.1 Basic examples

Look at the following three programs:

```
void g (char[] p){
  elapse (8 seconds);
  for (nat i=0; i<p.length(); i++){
    elapse (5 seconds);
    for (nat j=i; j<p.length(); j++){
      elapse (2 seconds);
    }
  }
}

void g2(char[] p){
  if (p.length()<1000) {
    elapse(1000000 seconds);
  } else {
    elapse (8 seconds);
    for (nat i=0; i<p.length(); i++){
      elapse (5 seconds);
      for (nat j=i; j<p.length(); j++){
        elapse (2 seconds);
      }
    }
  }
}

void g3 (char[] p){
  if (p.length()<1000) {
    elapse(1 second);
  } else {
    elapse (8 seconds);
    for (nat i=0; i<p.length(); i++){
```

```
      elapse (5 seconds);
      for (nat j=i; j<p.length(); j++){
        elapse (2 seconds);
      }
    }
  }
}
```

On arrays of size < 1000, the programs have very different running times. But since our alphabet is $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, there are only finitely many such arrays, specifically $\frac{1}{2}(3^{1000} - 1)$ of them. On all other arrays, the three programs have the same running time.

Little Tim regards `g3` as better than `g`, and `g2` as worse than `g`, but Big Tim regards them all as equivalent.

4.2 Comparing functions for sufficiently large n

Let's say we have two programs. The running time (in seconds) of Program 1A on all inputs of size $n > 0$ is

$$f(n) \;=\; 12n^3 + 300n^2 - 29n + 4$$

The running time (in seconds) of Program 1B on all inputs of size $n > 0$ is

$$g(n) \;=\; 12.01n^3 - n^2 + 7n - 5$$

Which is preferable? Little Tim points out that $f(1) = 287$ and $g(1) = 13.01$, so there are some inputs for which Program 1B is faster. But for Big Tim, who doesn't care about small inputs, it's Program 1A that is faster. He says: "When the input is large, only the term of highest degree matters, and $12n^3 < 12.01n^3$."

Let's make Big Tim's argument precise. We want to show that, for sufficiently large n, we have

$$f(n) \;<\; g(n)$$

Let's note that we have

$$\begin{aligned} f(n) &\leqslant 12n^3 + 300n^2 + 4 \\ &\leqslant 12n^3 + 300n^2 + 4n^2 \\ &= 12n^3 + 304n^2 \\ \text{and } g(n) &\geqslant 12.01n^3 - n^2 - 5 \\ &\geqslant 12.01n^3 - n^2 - 5n^2 \\ &= 12.01n^3 - 6n^2 \end{aligned}$$

So we have $f(n) < g(n)$ if we have

$$\begin{aligned} & 12n^3 + 304n^2 < 12.01n^3 - 6n^2 \\ \Leftrightarrow \quad & 310n^2 < 0.01n^3 \\ \Leftrightarrow \quad & 31000n^2 < n^3 \\ \Leftrightarrow \quad & n > 31000 \end{aligned}$$

And the alphabet is finite, so there are only finitely many inputs of size $\leqslant 31000$.

The same argument works for any two polynomials: all that matters for sufficiently large inputs is the term of largest degree, just as Big Tim said.

Now let's say that Program 2A has running time (in seconds) of

$$h(n) \;=\; n^2 + 17n + 2$$

and Program 2B has running time (in seconds) of 1.05^n. Big Tim says that Program 2A is faster, because "for large n, exponential is larger than polynomial".

Let's make this argument precise. The binomial theorem tells us that

$$\begin{aligned} 1.05^n &= 1 + n \times 0.05 + \frac{n(n-1)}{2!} \times 0.05^2 + \frac{n(n-1)(n-2)}{3!} \times 0.05^3 + \cdots \\ &\geqslant 1 + n \times 0.05 + \frac{n(n-1)}{2!} \times 0.05^2 + \frac{n(n-1)(n-2)}{3!} \times 0.05^3 \end{aligned}$$

which is cubic and therefore $> h(n)$ for sufficiently large n, as we saw before. In summary, an exponential function is always larger than a polynomial function for sufficiently large inputs, just as Big Tim said.

Now recall that $n^{0.001}$ is the thousandth root of n. Let's say that the running time (in seconds) of Program 2C is $1.05^{(n^{0.001})}$. So Program 2C is faster than Program 2B, but is it slower than Program 2A? Yes it is. To see this, put $m = n^{0.001}$. Then $h(n)$ is polynomial in m, whereas $1.05^{(n^{0.001})}$ is exponential in m, and therefore $h(n) < 1.05^{(n^{0.001})}$ for sufficiently large m. So this is also true for sufficiently large n.

Now let's say that the running time (in seconds) of Program 3A is $\log_{1.05} n$, and that of Program 3B is $n^{0.001}$. Big Tim says that Program 3A is faster because "Logarithmic is always less than polynomial". To make his argument precise, note that the statement $\log_{1.05} n < n^{0.001}$ is equivalent to the statement $n < 1.05^{(n^{0.001})}$, which we've already seen is true for sufficiently large n.

To sum up, here are Big Tim's slogans:

- For polynomials, it's only the term of highest degree that matters.
- An exponential function is larger than every polynomial.
- A logarithmic function is smaller than every polynomial.
- All this is on the assumption that the input is sufficiently large.

5 Constant factors

5.1 Basic examples

Look at the following three programs:

```
void g (char[] p){
  elapse (8 seconds);
  for (nat i=0; i<p.length(); i++){
    elapse (5 seconds);
    for (nat j=i; j<p.length(); j++){
      elapse (2 seconds);
    }
  }
}

void g4 (char[] p){
  elapse (8000 seconds);
  for (nat i=0; i<p.length(); i++){
    elapse (5000 seconds);
    for (nat j=i; j<p.length(); j++){
      elapse (2000 seconds);
    }
  }
}
```

```
void g5 (char[] p){
  elapse (0.008 seconds);
  for (nat i=0; i<p.length(); i++){
    elapse (0.005 seconds);
    for (nat j=i; j<p.length(); j++){
      elapse (0.002 seconds);
    }
  }
}
```

The running times of these programs differ by a constant factor: `g4` is a thousand times slower than `g`, and `g5` is a thousand times faster. So Big Tim regards `g5` as better than `g`, and `g4` as worse than `g`, but Constance (and also Polly) regards them all as equivalent.

5.2 Steps

Now consider the following code:

```
void g6 (char[] p){
  elapse (8 steps);
  for (nat i=0; i<p.length(); i++){
    elapse (5 steps);
    for (nat j=i; j<p.length(); j++){
      elapse (2 steps);
    }
  }
}
```

Constance regards `g6` as equivalent to `g`. Whether a step is a second, 1000 seconds or 0.001 seconds doesn't matter to her. All that matters is that a step is a fixed length of time.

Usually in complexity theory, we follow the opinion of Constance, which allows us to give the running time in steps, not seconds. This is helpful, because we can often look at a program and estimate the number of steps by making some reasonable assumptions. You will see this when studying Algorithms.

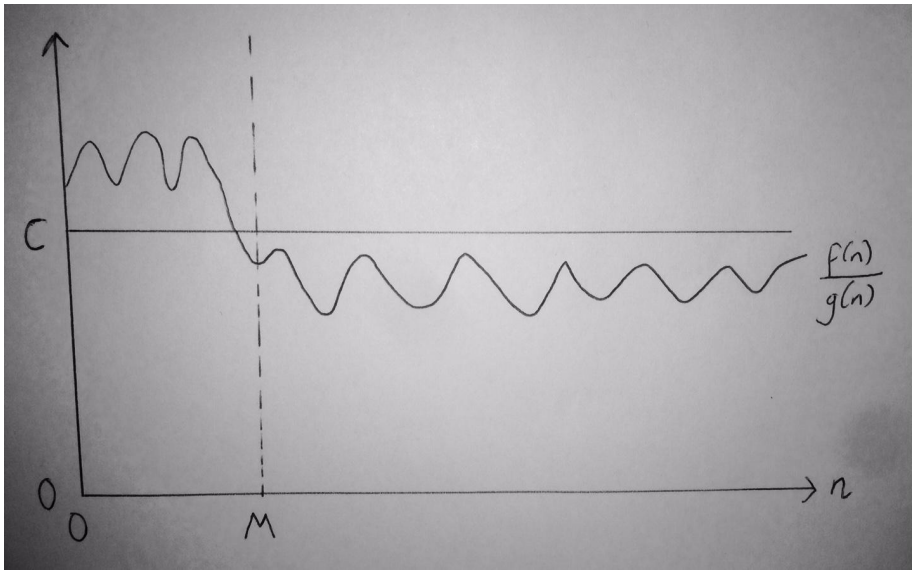
5.3 Time Complexity using Big-O Notation

Now we are ready for the most important definition in complexity theory, **Big O** notation. Let f be a function from $\mathbb{N}$ to the positive reals. (Think of $f(n)$ as the running time for an argument of size n. It could be worst case or it could be average case.) Let g be another such function. We say that $f \in O(g)$, or more informally that $f(n)$ is $O(g(n))$, when the following condition holds: there is a natural number M and constant factor C, such that for all $n \geqslant M$, we have $\frac{f(n)}{g(n)} \leqslant C$.

In logical symbols:

$$\exists M.\, \exists C.\, \forall n \geqslant M.\, \frac{f(n)}{g(n)} \leqslant C$$

A helpful slogan to remember is "Big O means proportional or less". Here is a picture:



Note: In many cases $f(n)$ or $g(n)$ may be undefined or negative or zero (contrary to what I said), but only for small n. An example is $g(n) = \log_2 \log_2 n$, which is undefined for $n = 0$ and $n = 1$ and zero for $n = 2$ but positive for all $n \geqslant 3$. This is not a problem because, provided we take $M \geqslant 3$, the inequality makes sense.

Tip: If you want to compare two functions f and g for complexity, try dividing $f(n)$ by $g(n)$ and see what happens as n gets large.

5.4 Complexity Notations

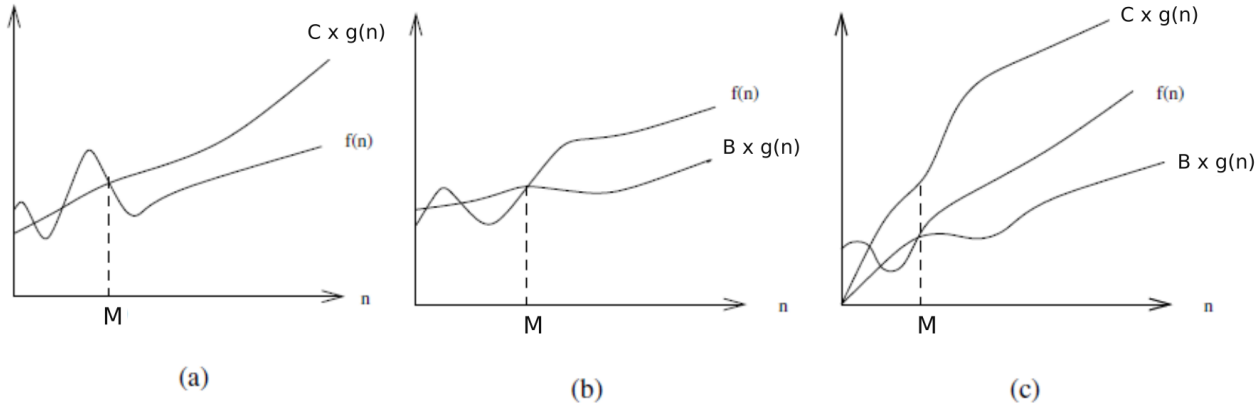
Let f and g be functions from $\mathbb{N}$ to the nonnegative reals.

(a) We say that $f \in O(g)$, or informally "$f(n)$ is $O(g(n))$", when g is an upper bound for f up to a constant factor. That is: there are numbers M and C such that for $n \geqslant M$ we have $f(n) \leqslant C \times g(n)$.

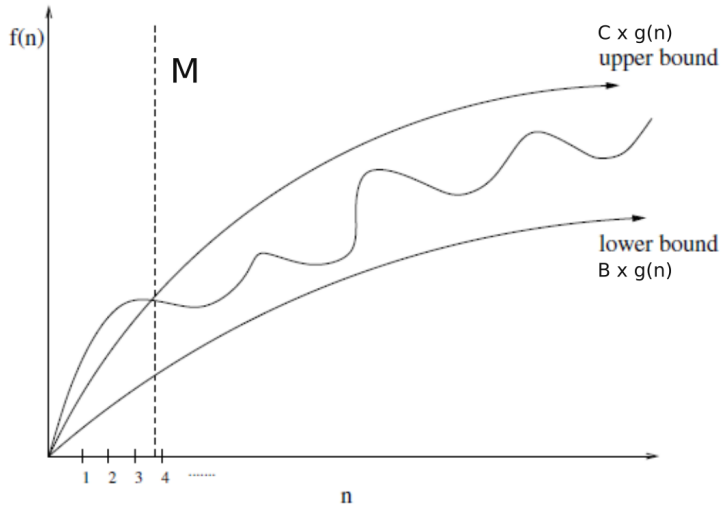
(b) We say that $f \in \Omega(g)$, or informally "$f(n)$ is $\Omega(g(n))$", when g is a lower bound for f up to a constant factor. That is: there are numbers M and $B > 0$ such that for $n \geqslant M$ we have $B \times g(n) \leqslant f(n)$.

(c) We say that $f \in \theta(g)$, or informally "$f(n)$ is $\theta(g(n))$", when both of the above conditions hold. That is: there are numbers M and C and $B > 0$ such that for $n \geqslant M$ we have $B \times g(n) \leqslant f(n) \leqslant C \times g(n)$.

The following figure illustrates the above complexity notations.



With these notations, we can be more precise about complexity. For example, if we say that the worst case running time is $O(n^2)$, it might in fact be linear, but if we know that it is $\theta(n^2)$ then it really is no better than quadratic, because it will be within the lower and upper bounds of complexity. The upper and lower bounds that are valid for $n > M$ smooth-out the behavior of complex functions, we would like to have a tight-bound to ensure our estimations are precise, as shown below:



6 Polynomial time

We have seen that in complexity theory we ignore small arguments and constant factors, to get a rough estimate of running time. But people like Polly take this further and ask just one question: is the running time polynomial? That is a *very* basic requirement of a program: a polynomial time program might be slow, but a program that isn't polynomial time is regarded as utterly infeasible.

Definition. Let the running time of a program be given by a function from $\mathbb{N}$ to the positive reals. (This could be worst case or average case.) The program is *polynomial time* when there is k such that $f(n)$ is $O(n^k)$. In detail, it is polynomial time when there is k and M and C such that if $n \geqslant M$ then $f(n) \leqslant C \times n^k$.

In 2002, Agrawal, Kayal and Saxena published a polynomial time algorithm for testing whether a number p is prime. Its running time is in $O(n^{13})$, where n is the length of p. This was surprising; people had previously suspected that no such algorithm existed. The result has since been improved to an algorithm whose running time is $O(n^7)$.

6.1 Test Your Understanding

1. The running time of my program, on an argument of size n, is $3n^2 + 9n + 8$. Is this $O(n^2)$? Is it $O(n)$? Is it $O(n^3)$?

2. The running time of my program, on an argument of size n, is 5^n for $n < 1000$, and $3n^2 + 9n + 8$ for $n \geqslant 1000$. Is this $O(n^2)$? Is it $O(n)$? Is it $O(n^3)$?

3. On an argument of size n, I first run a program whose running time is in $O(n^2)$, and then run a program whose running time is in $O(n^3)$. Show that the total running time is in $O(n^3)$.

7 Space Complexity

We can study the space (memory) usage of a program in a similar way. For example, suppose my program, on an argument of size n, uses $8n^2 + 5$ bytes of memory. We then say that the space usage is quadratic.

Turing Machines

1 Quick Review – How many steps?

We have seen how to analyze the running time of a program by counting the number of steps.
For example:

```
void g6 (char[] p){
  elapse (8 steps);
  for (nat i=0; i<p.length(); i++){
    elapse (5 steps);
    for (nat j=i; j<p.length(); j++){
    }
      elapse (2 steps);
  }
}
```

Remember that a "step" is supposed to be a fixed amount of time.

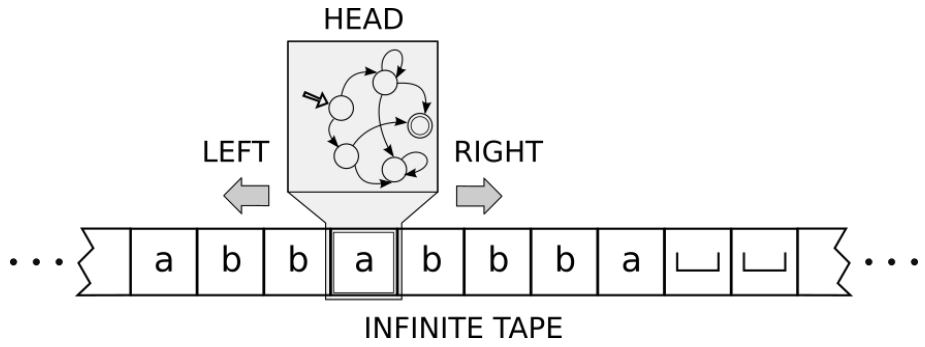
This kind of analysis is widely used and convenient. But how do we get the basic step counts in each part of the code? They are just assumptions (or guesses). For example, many people analyze sorting algorithms by assuming that each comparison of two values is "one step". That's certainly a helpful assumption but it ignores the fact that comparing two big numbers takes longer than comparing two small numbers.

To reason about running time in a rigorous way, and avoid the risk of sweeping any time costs under the carpet, we need a precise *Model of Computation* that fully specifies the steps. The model we'll be looking at is the *Turing Machine (TM)*, invented by Alan Turing in 1936. A TM takes a very conservative view of what constitutes a step, so it serves as a gold standard. If your algorithm is fast on a Turing Machine, it's indisputably fast!

2 What is a Turing Machine?

> *"A Turing machine can do everything that a real computer can do. Nonetheless, even a Turing machine cannot solve certain problems. Similar to a finite automaton but with an unlimited and unrestricted memory, a Turing machine is a much more accurate model of a general purpose computer. In a very real sense, these problems are beyond the theoretical limits of computation.*
> *The Turing machine model uses an **infinite tape** as its unlimited memory. It has a tape **head** that can read and write symbols and move around on the tape. Initially the tape contains only the input string and is blank everywhere else. If the machine needs to store information, it may write this information on the tape. To read the information that it has written, the machine can move its head back over it. The machine continues computing until it decides to produce an output. The outputs accept and reject are obtained by entering designated accepting and rejecting states. If it doesn't enter an accepting or a rejecting state, it will go on forever, never halting."* (Sipser, 2013).

In simple words, a Turing machine is a simple formal model of mechanical computation, and a universal Turing machine can be used to compute any function, which is computable by any other Turing machine. A Turing machine has finitely many states (like a DFA) but it also has an external memory: an infinite tape, divided into cells. The machine has a *head* that sits over one cell of the tape. The Turing machine can read and write symbols on the tape and move left or right (or stay put) after each step. Unlike a DFA, once a Turing machine enters an accept or reject state, it stops computing and halts. The following diagram shows a general representation of a Turing machine:



Before giving a Turing machine, we first specify two finite sets:

- The *Tape Alphabet* T, which includes a "blank" character ␣ (also shown above). At any time, each cell contains a character in T, and there are only finitely many cells with a non-blank character. We will usually take $T = \{\mathtt{a}, \mathtt{b}, ␣\}$. The set of non-blank characters $\{\mathtt{a}, \mathtt{b}\}$ is called the *input alphabet* (Σ).
- The set V of *Return Values*.

For $V = \{\mathsf{true}, \mathsf{false}\}$, the instructions available are the following:

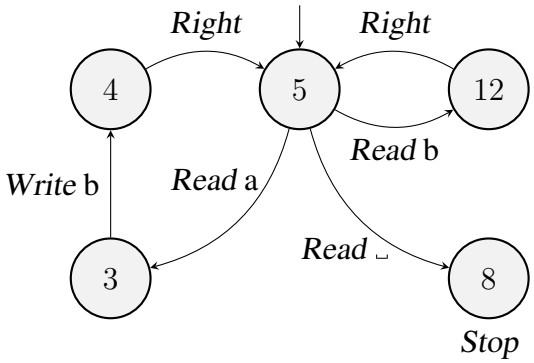
- *Read*, which may result in `a` or `b` or ␣
- *Write* `a`
- *Write* `b`
- *Write* ␣
- *Move Left*
- *Move Right*
- *No-op*, which does nothing

- *Return* true (accept)
- *Return* false (reject)

If V is singleton then the Return instruction is usually just written Stop. It is important to keep note of the following points:

- Where is the head at the start?
- Where should the head be at the end?

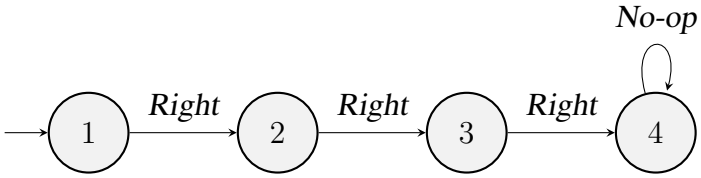
The following machine starts on the leftmost cell of an a, b-block on an otherwise blank tape. It moves to the right, converting every a to b, and halts on the cell to the right of the block.



For example:

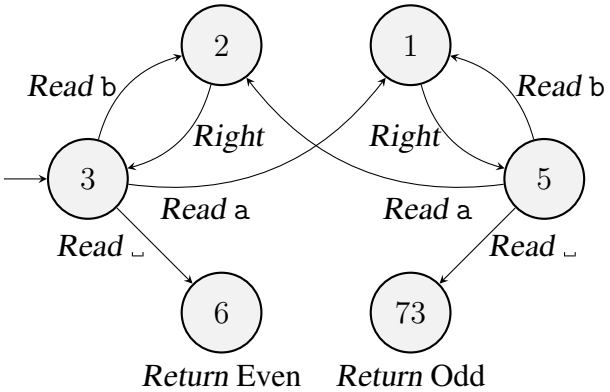
ḃaba	5	*Read* b
ḃaba	12	*Right*
bȧba	5	*Read* a
bȧba	3	*Write* b
bḃba	4	*Right*
bḃba	5	*Read* b
...		

The following machine moves three steps to the right and waits forever.



2.1 Parity Checking Example

The following "parity checking" machine has:
Tape Alphabet: $T = \{␣, \mathtt{a}, \mathtt{b}\}$, Return Set: $V = \{\text{Even}, \text{Odd}\}$
It starts on the leftmost cell of an a, b-block on an otherwise blank tape, and it ends on the cell to the right of the block, saying whether the number of a's is even or odd.



More formally, a Turing machine consists of the following, over T and V:

- A finite set of X states
- An initial state $p \in X$.
- A transition function δ from X to

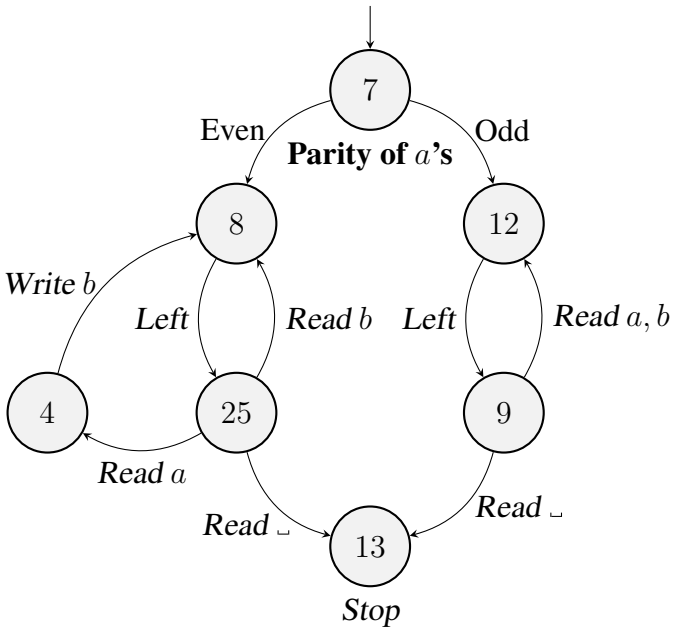
$$
\begin{aligned}
&\{\textit{Read}(f) \mid f \in X^T\} && \text{(Read instructions and change state)}\\
\cup\, &\{\textit{Write}(l, s) \mid (l, s) \in T \times X\} && \text{(Write instructions and change state)}\\
\cup\, &\{\textit{Left}\ s \mid s \in X\} && \text{(Move head left and change state)}\\
\cup\, &\{\textit{Right}\ s \mid s \in X\} && \text{(Move head right and change state)}\\
\cup\, &\{\textit{No-op}\ s \mid s \in X\} && \text{(No-op and change state)}\\
\cup\, &\{\textit{Return}\ v \mid v \in V\} && \text{(Return a value from set } V\text{)}
\end{aligned}
$$

The above parity-checking TM can be formally described as:

$$
\begin{aligned}
X \triangleright \quad & (\{3, 2, 6, 73, 5, 1\},\\
p \triangleright \quad & \{3\},\\
\delta \triangleright \quad & \{3 \longmapsto \textit{Read}(a \mapsto 1, b \mapsto 2, \text{␣} \mapsto 6)\\
& \ 2 \longmapsto \textit{Right}\ 3\\
& \ 6 \longmapsto \textit{Return}\ \text{Even}\\
& \ 5 \longmapsto \textit{Read}(a \mapsto 2, b \mapsto 1, \text{␣} \mapsto 73)\\
& \ 73 \longmapsto \textit{Return}\ \text{Odd}\\
& \ 1 \longmapsto \textit{Right}\ 5\\
& \})
\end{aligned}
$$

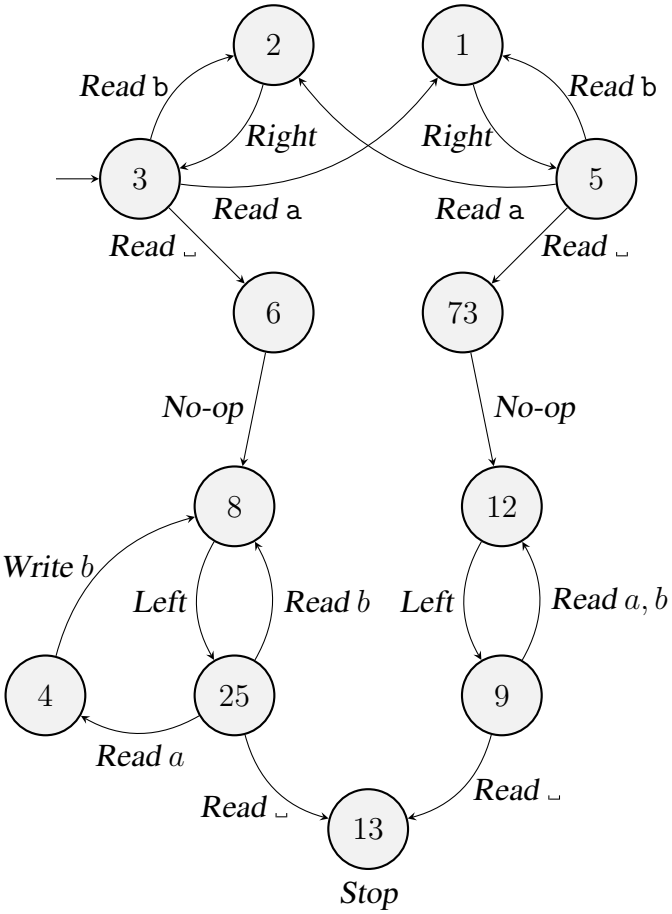
2.2 Macros

A convenient way of writing a program is using *macros*, which is a single instruction that abbreviates a whole program. To get the full program out of a *program with macros*, we need to *expand* all of them. Here is an example, using the parity checker that we saw above:



We can see that the state 7 is a macro, which abbreviates the parity checking of a i.e. whether the number of a's is even or odd. To obtain the full program, we expand this macro, which means that we replace "Parity

of a's" with the parity checker's definition. Anything that points to the macro, will now point to the initial state of the definition. Likewise, if the state with the macro is initial, the initial state of the definition is initial state of the expanded program. For example, the arrow pointing to the starting state 7 will now point to the starting state 3 of the macro definition. Each Return instruction of the parity checker is replaced by a No-op, leading to the appropriate next state of the main program i.e. the next state will be the one that results from V after the macro.



2.3 Example: Reverse copy on a single tape machine

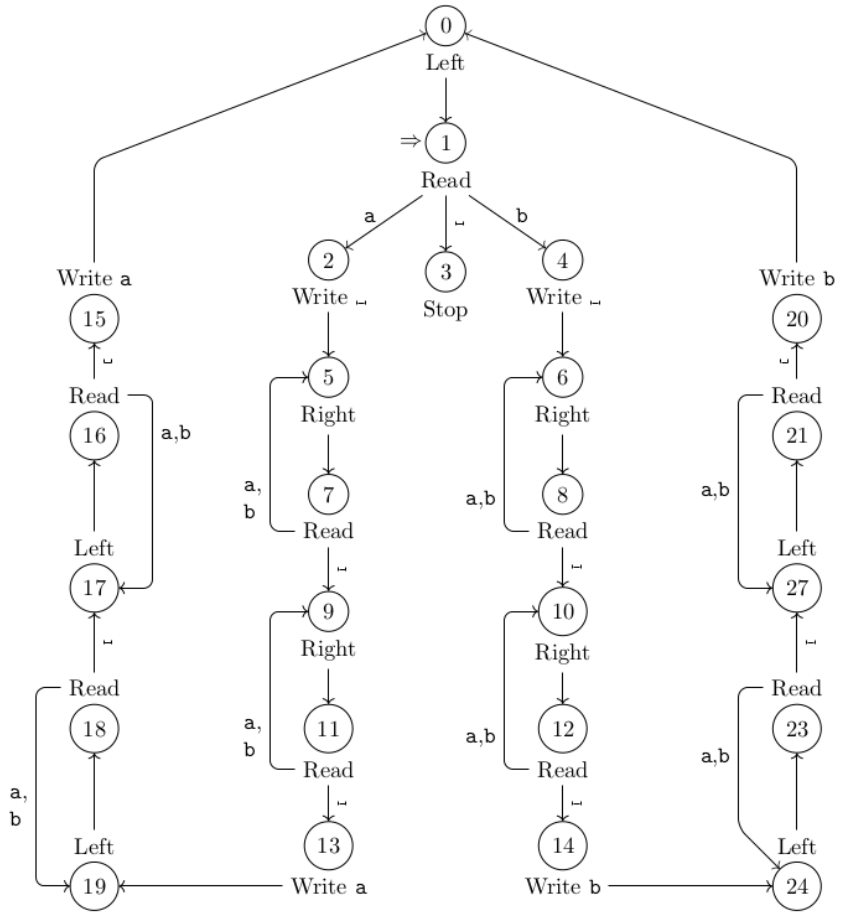
For example, we would like to build a Turing machine that starts at the rightmost character of an a, b-block on an otherwise blank tape and places a reversed copy of the input string to the right, with a $blank$ character in between the original and reversed strings. The machine should halt with the head on the blank cell to the left of the original block. For example, at the start:

abbaḃ

We expect to get the following output:

␣abbab␣babba

The following Turing machine implements the desired program:



Note how this machine is forced to use a ␣ character to record the position currently being copied whether the copied character is a or b is included in the state. This is somewhat an abuse of the ␣ character which would normally denote the start/end of the input — *hacking!*

We can work out the precise number of steps of this program, in terms of the length of the initial block. The general outline of the steps undertaken by this TM are given as under:

- Record the current character, whether its a or b
- Replace it with a ␣ (*hacking!*)
- Move two steps to the right and write this character
- Then go back and replace the ␣ with a or b (whatever was there before)
- Move one step to the left
  - Record the current character, whether its a or b
  - Replace it with a ␣ (*hacking!*)
  - Move right to the central ␣ character
  - Move right to the next ␣ and write this character
  - Move left to the central ␣ character
  - Move left to the character we just blacked-out
  - Replace the ␣ with a or b (whatever was there before)
  - Move one step to the left, if its ␣ then stop
  - Otherwise, begin the cycle again.

The running time is evidently *quadratic*: $1 + 2 + 3 + \cdots + n$ times a constant! It can be shown that the copy-reverse task cannot be solved any faster than this.

3 Turing Machine Variants

As we have seen, because the notion of Turing machine is so conservative, programs can be intricate and run slowly. Let us consider some more liberal variants.

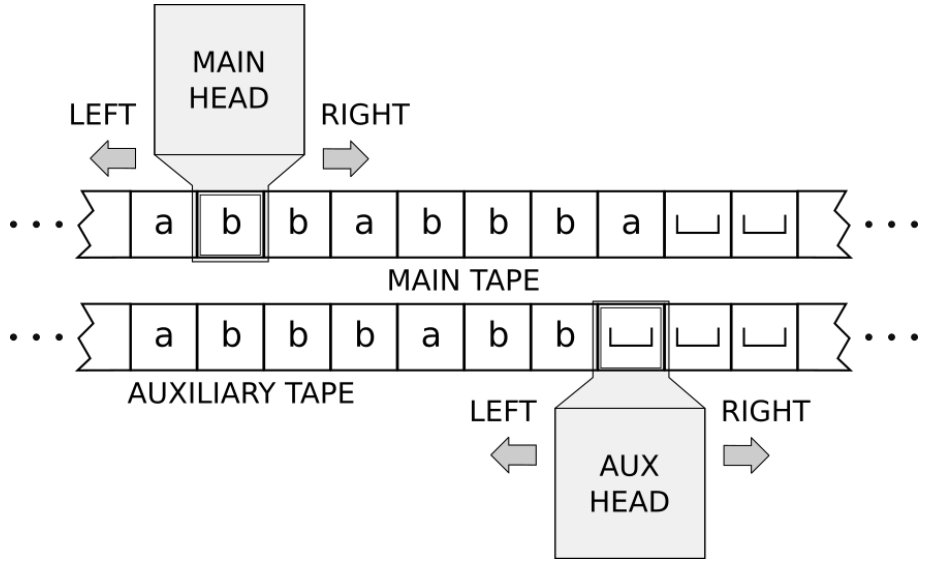
3.1 Auxiliary Characters

Suppose that, in addition to the input alphabet and the blank, we have a finite set of auxiliary characters. A program may assume that initially these do not appear, and must guarantee that finally they don't appear, but in the middle of execution they can be used. For example, suppose the input alphabet is $\{\mathtt{a}, \mathtt{b}\}$ and the auxiliary alphabet is $\{\mathtt{a}', \mathtt{b}'\}$. Then we can write a more straightforward program for copy-reverse, using $\mathtt{a}'$ to indicate a currently being copied, and $\mathtt{b}'$ to indicate b currently being copied (rather than using the blank as previously).

3.2 Auxiliary/Multitape Turing Machines

A two-tape Turing machine has a main tape and an auxiliary tape, with a head on each tape. The input to the two-tape TM is provided on the main tape and a program may assume that initially the auxiliary tape is blank and must ensure that finally it is blank. The original single-tape TM and its reasonable variants all have the same power i.e. they are able to recognize the same class of languages.

The available instructions are *Write Main* x, *Write Aux* x, *Read Main*, *Read Aux*, *Left Main*, *Left Aux*, *Right Main*, *Right Aux*, *No-op*, and *Return* v.

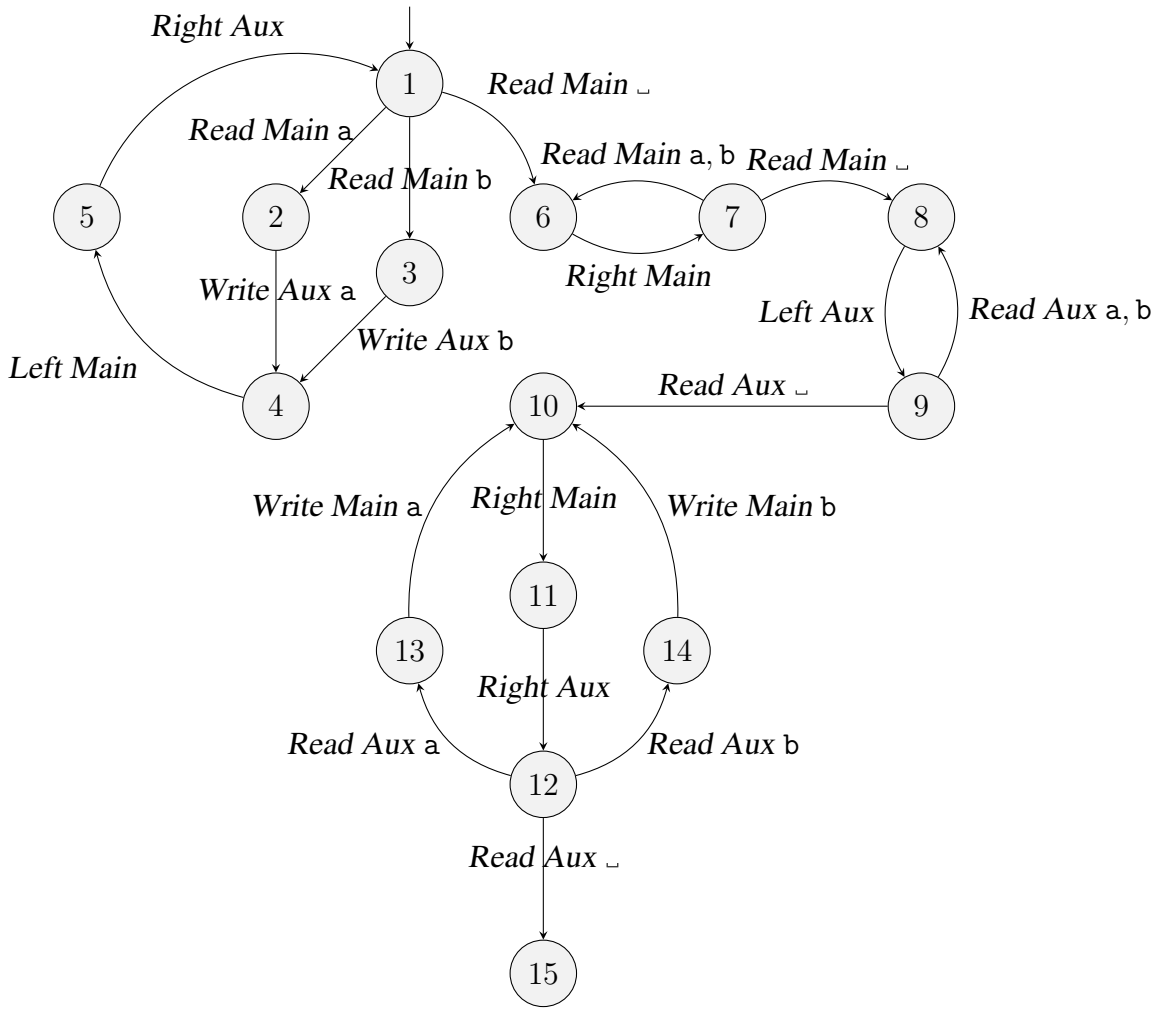


Details on the available instructions are given below:

- *Read Main*, which may result in a or b or ␣
- *Read Aux*, which may result in a or b or ␣
- *Write Main* x (x = a or b or ␣)
- *Write Aux* x (x = a or b or ␣)
- *Left Main*
- *Left Aux*
- *Right Main*
- *Right Aux*
- *No-op*, which does nothing
- *Return* true (accept)
- *Return* false (reject)

3.2.1 Example: Reverse copy on a two-tape machine

The following two-tape TM shows how the copy-reverse problem can be solved in linear time:



Move left through 2 blocks on main tape and stop

For example, if this two-tape TM starts with: ␣abbab␣$\cdots$ on the main tape (main head at the first blank, after the string) and ␣␣␣␣␣␣␣$\cdots$ on the auxiliary tape (auxiliary head at the first blank, on an overall empty tape). In the first phase, from states 1 to 5, the TM reads from the main tape (a's and b's) and writes them to the auxiliary tape, while moving the main head to the left and the auxiliary head to the right. Once it reads a *blank* from the main tape, it resets both heads, by moving the main head to the right and auxiliary head to the left, until both read a *blank* (states 6 to 9).

In the second phase, from states 10 to 14, it moves both of the heads to the right, and reads from the auxiliary tape (a's and b's, which are now in reverse order) and writes them to the main tape. Once it reads a *blank* from the auxiliary tape, the TM knows that it is done with the program, and can reset its both heads, as desired. We should also erase the auxiliary tape, during the resetting process (details omitted here). We expect to get the following output on the main tape: ␣abbab␣babba

4 Summary

In this handout, we have quickly reviewed the complexity notations and understood the need to study Turing machines. We have seen the general model of a Turing machine, which contains a head, an infinite tape, and can move its head left/right while executing a program. We have understood that a TM is a simple formal model of mechanical computation, and it can do everything that a real computer can do. We have studied some examples of Turing machines, including parity checking, macros and hacking. We have also

discussed Turing machine variants, including auxiliary characters, multi-tape and two-dimensional TMs. We have understood that the "append reversed copy" problem can be solved in:

- quadratic time $O(n^2)$ on a TM.
- linear time $O(n)$ on 2-tape TM.

Do you think that we can solve this problem on a TM in linear time? The answer is No! Do you think which of these machine models of computation is more appropriate? You could argue that a 2-tape TM is *unrealistic* because it allows for instant communication between the two heads that may be far apart.

We have seen that the same problem can be solved with different complexity on different machines e.g. quadratic on one machine can be linear on another. We haven't however seen that:

- a problem that can be solved in polynomial time on one kind of machine but not on another.
- a problem can be solved in one kind of machine but not on the other.

Actually, these things can't happen for all the kinds of machines we have looked at. So, for Constance, it matters whether we use a TM or a 2D-TM, but for Polly, it doesn't matter because it is still polynomial whether its $O(n)$ or $O(n^2)$.

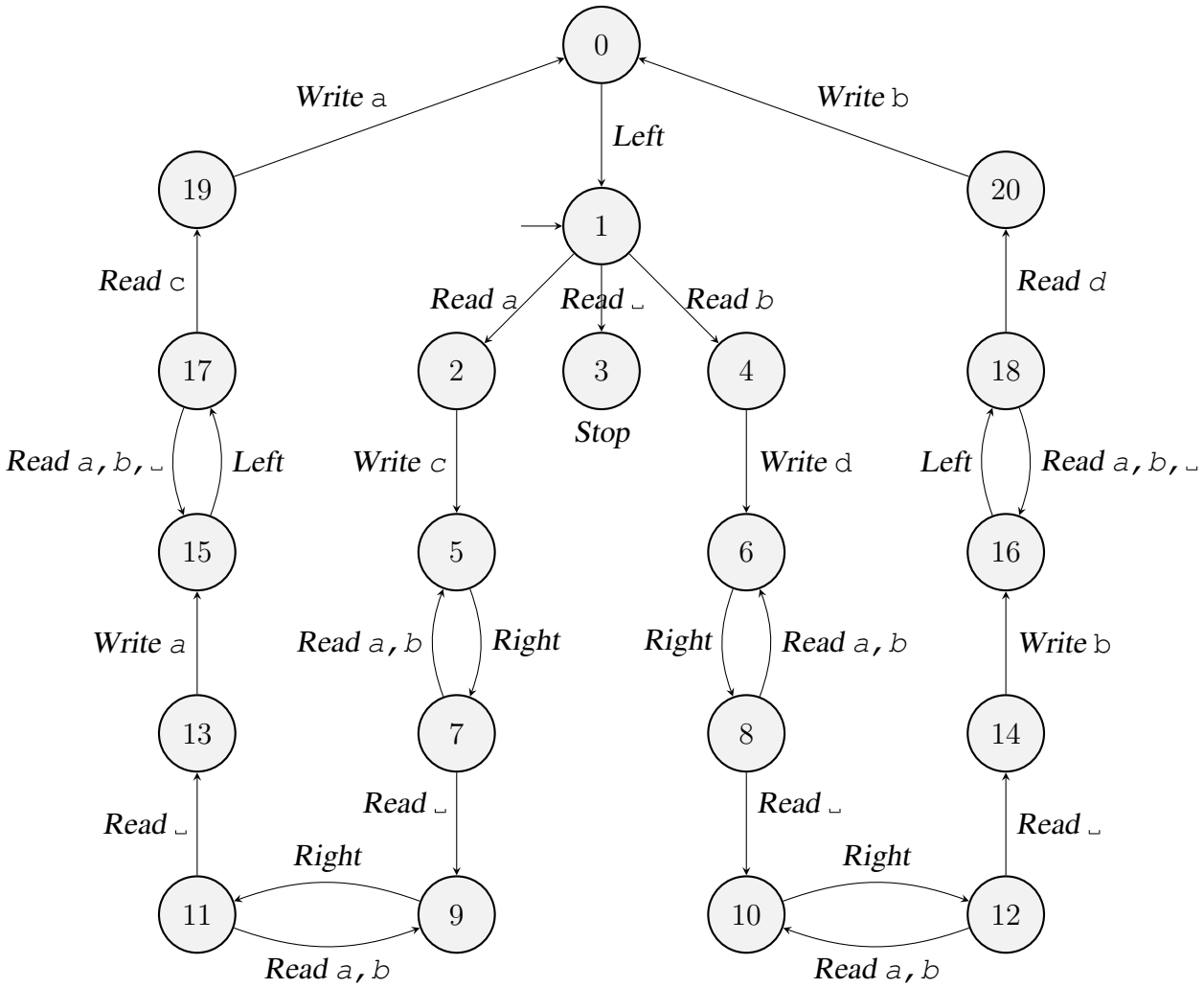
5 Further Readings / References

- Sipser, M. (2013) *Chapter #3: The Church–Turing Thesis, Introduction to the Theory of Computation*, 3rd Edition, CENGAGE Learning Custom Publishing, Mason, USA

Converting Fancy Turing Machines to Simple Machines

1 Turing Machines using Auxiliary Characters

Suppose that, in addition to the input alphabet and the blank, we have a finite set of auxiliary characters. A program assumes that initially these do not appear, and must guarantee that finally they don't appear, but in the middle of execution they can be used. For example, suppose the input alphabet is $\{\mathtt{a}, \mathtt{b}\}$ and the auxiliary alphabet is $\{\mathtt{c}, \mathtt{d}\}$. This means that we can have instructions *Write* `a`, *Write* `b`, *Write* `c`, *Write* `d` and *Write* ␣, and each *Read* instruction has 5 possible outcomes (`a`, `b`, `c`, `d` and ␣). We can now write a more straightforward (but still quadratic time) program for copy-reverse problem, using `c` to indicate that character `a` is currently being copied, and `d` to indicate that character `b` is currently being copied (rather than using the blank, as previously). For example, here is a fancy TM (a TM including auxiliary characters) for the copy-reverse problem discussed earlier.



We're going to learn a general method for converting a fancy TM to an ordinary TM. This general method will involve the following steps:

1. Give a relation between the tape configurations of fancy TM and the tape configuration of the simple TM. In simple words, define a way to represent the tape configuration (i.e. tape contents plus head position) of the fancy TM as a configuration of the simple TM. This is a *creative step!*
2. Give a program to convert the initial configuration of the fancy TM into a corresponding configuration of the simple TM (The "Setting-up" program).
3. For each instruction of the fancy TM, show how to simulate it on a simple TM. In other words, show how to "perform" each step of the fancy TM on a simple TM (The "Simulating" programs).
4. Give a program to convert the result of simple TM to the fancy TM result (The "Finishing" program).

Typically all of the above programs are polynomial time.

Key Point: *A polynomial time program on a Fancy Turing machine can be converted into a polynomial time program on a Simple Turing machine.*

We will now show the details of the above steps in the following sections.

1.1 Defining Relation between Fancy & Simple Tape Configurations

Let's say we have a fancy TM using the input alphabet {a, b}, and auxiliary characters {c, d}. It means that we **assume** only a, b and ␣ at the start of TM, but we are **allowed** to use c and d in the middle of execution. However, we must **ensure** that only a, b, ␣ are present in the output of the fancy TM at the end. When defining the relationship between fancy and simple tapes, the key point will be to assume that the fancy and simple tape configurations are related at the start of each simulation step, we may violate this relationship in the middle of the step, but we must ensure it at the end.

In this step, we shall define a relation between the fancy and simple TMs' tapes configurations. Lets consider the following fancy tape configuration:

$$\mathtt{acbccda}$$

Note: *Obviously, the above configuration is not the initial configuration of the fancy TM, as the auxiliary symbols can only appear on the tape during the execution, but not at the start or at the end.*

In order to represent the fancy TM's tape configuration as a simple TM's tape configuration, we define the following relation (which is a *creative suggestion*):

Character on Fancy tape	Represented on Simple tape
a	aa
b	bb
c	ab
d	ba
␣	␣␣

The simple TM's head position will be the leftmost of the two characters representing the fancy TM's head position. For example, the fancy tape configuration:

$$\mathtt{acbccda} \tag{1}$$

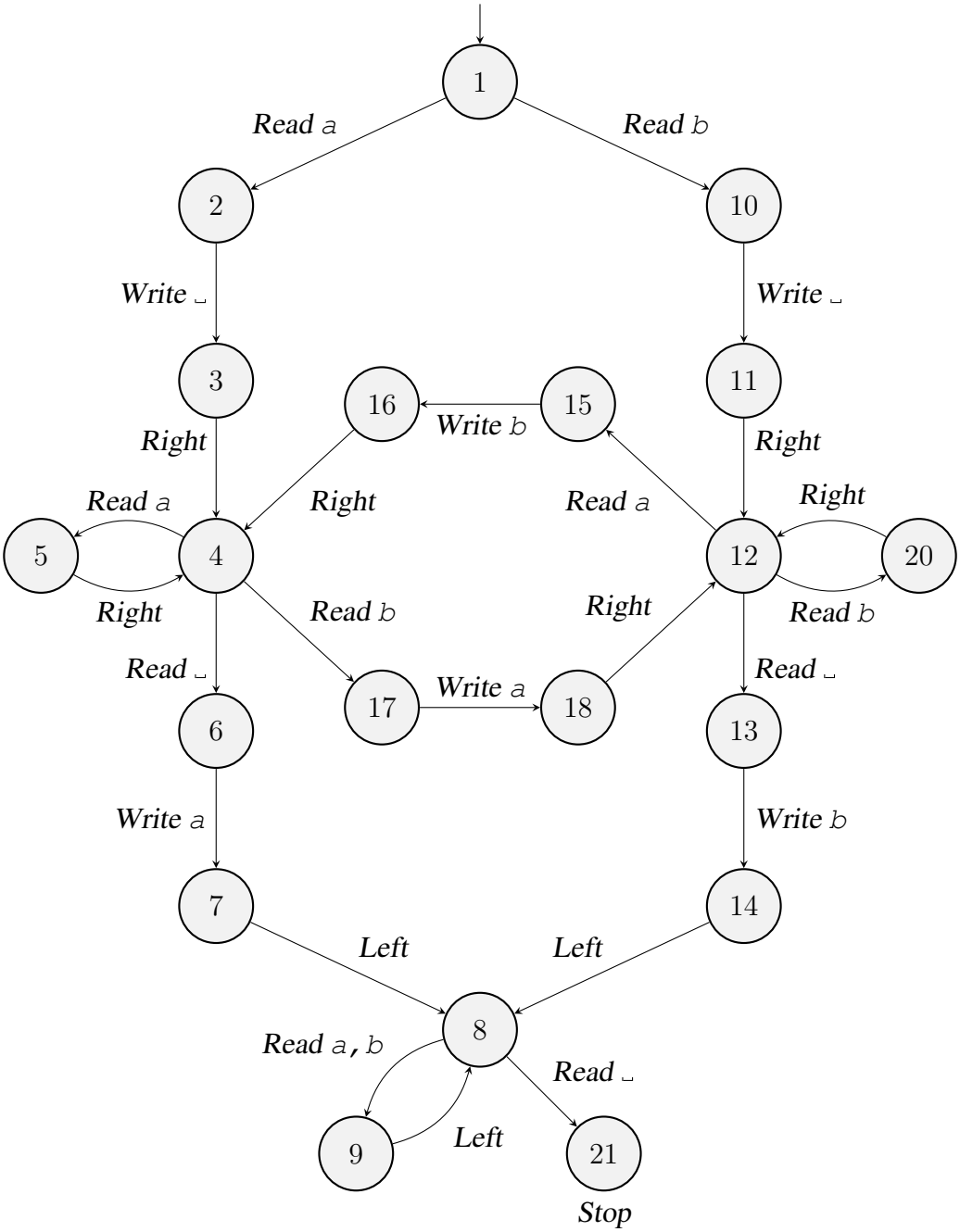
is represented as the simple tape configuration:

$$\mathtt{aaabbbababbaaa} \tag{2}$$

1.2 Converting Initial Configurations – The Setting-up Program

In this step, we will setup the simple TM to represent the fancy TM *i.e.* we want a program that stretches a fancy *input* tape — recall that this will contain only `a`, `b`, ␣ — into a corresponding simple tape. For example, if the input is initially `ȧbbab`, the simple TM will start by *stretching* the given input to `ȧabbbbaabb`.

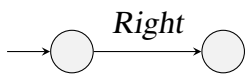
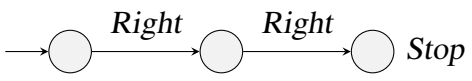
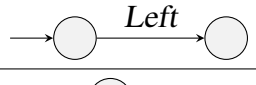
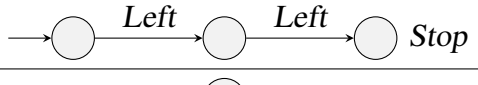
One way to write a stretching program is to add one character at a time, which would require $O(n)$ steps, and then repeat it for each of the characters in the input, requiring a total of $O(n^2)$ steps. As an example, consider the modified version of the TM seen during last week, which makes a space at the current head position, e.g. given the input $\dot{a}bab$ it will result in the output $\dot{\sqcup}abab$. We can further modify this TM to achieve the stretching program (left as an exercise for you).



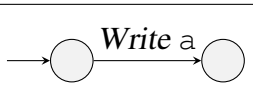
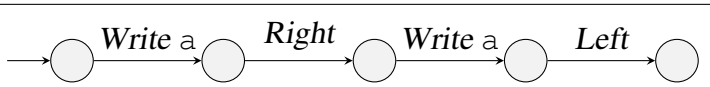
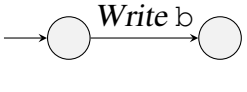
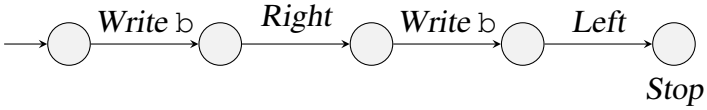
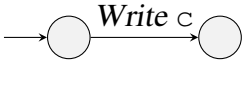
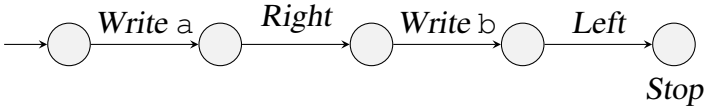
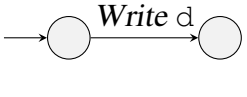
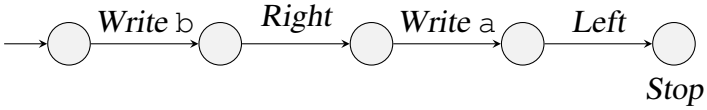
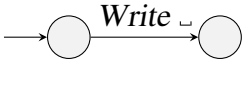
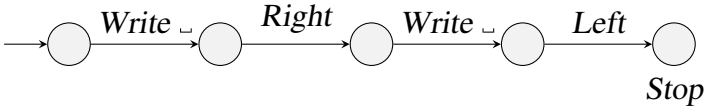
You can easily see that the above TM takes $O(n)$ steps for creating a space; we will have a similar complexity for the modified version to duplicate a single character on the tape. The full stretching program will repeat the above steps for each of the characters on the tape, therefore, it will take $O(n^2)$ steps.

1.3 Performing Fancy TM Instructions – The Simulating Programs

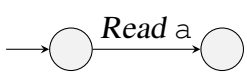
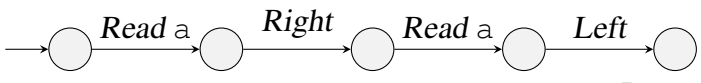
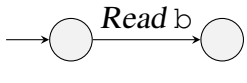
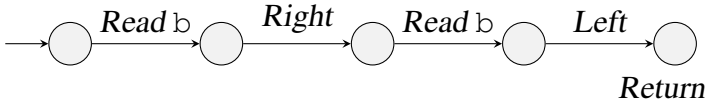
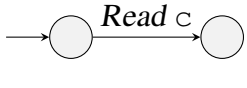
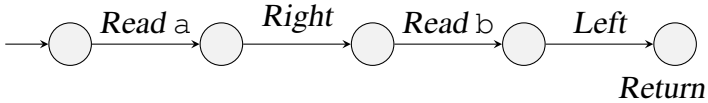
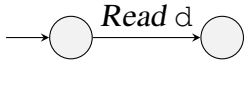
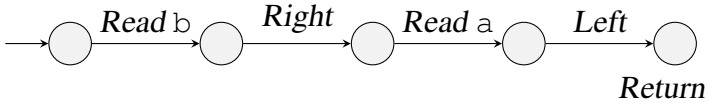
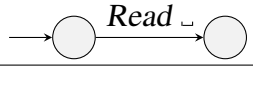
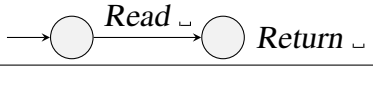
In this step, we simulate each instruction of the fancy TM by a program of the simple TM. For example, we simulate the fancy *Right*, *Left* and *Stop* as the following simple programs:

Fancy TM Steps	Simple Simulating TMs
	
	
	

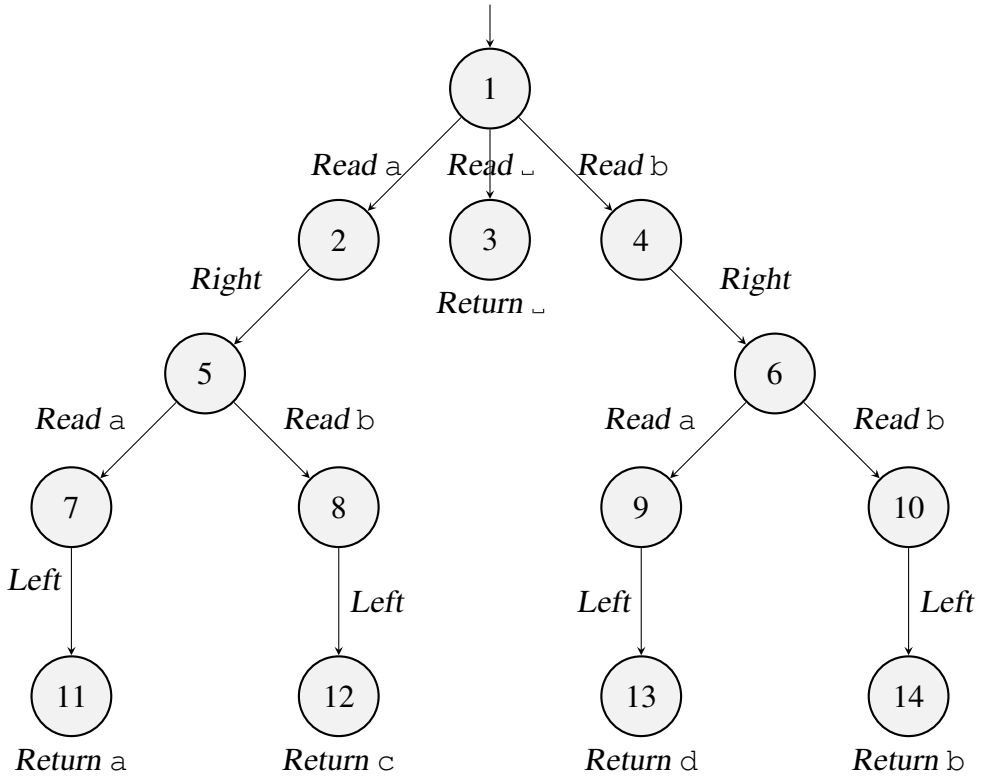
Similarly, we can create the following simple TM programs for the *Write* operations.

Fancy TM Steps	Simple Simulating TMs
	
	
	
	
	

It is important to note that the tape head on the simple TM is moved to the left, after writing the second character. Similarly, we can create the following simple TM programs for the *Read* operations.

Fancy TM Steps	Simple Simulating TMs
	
	
	
	
	

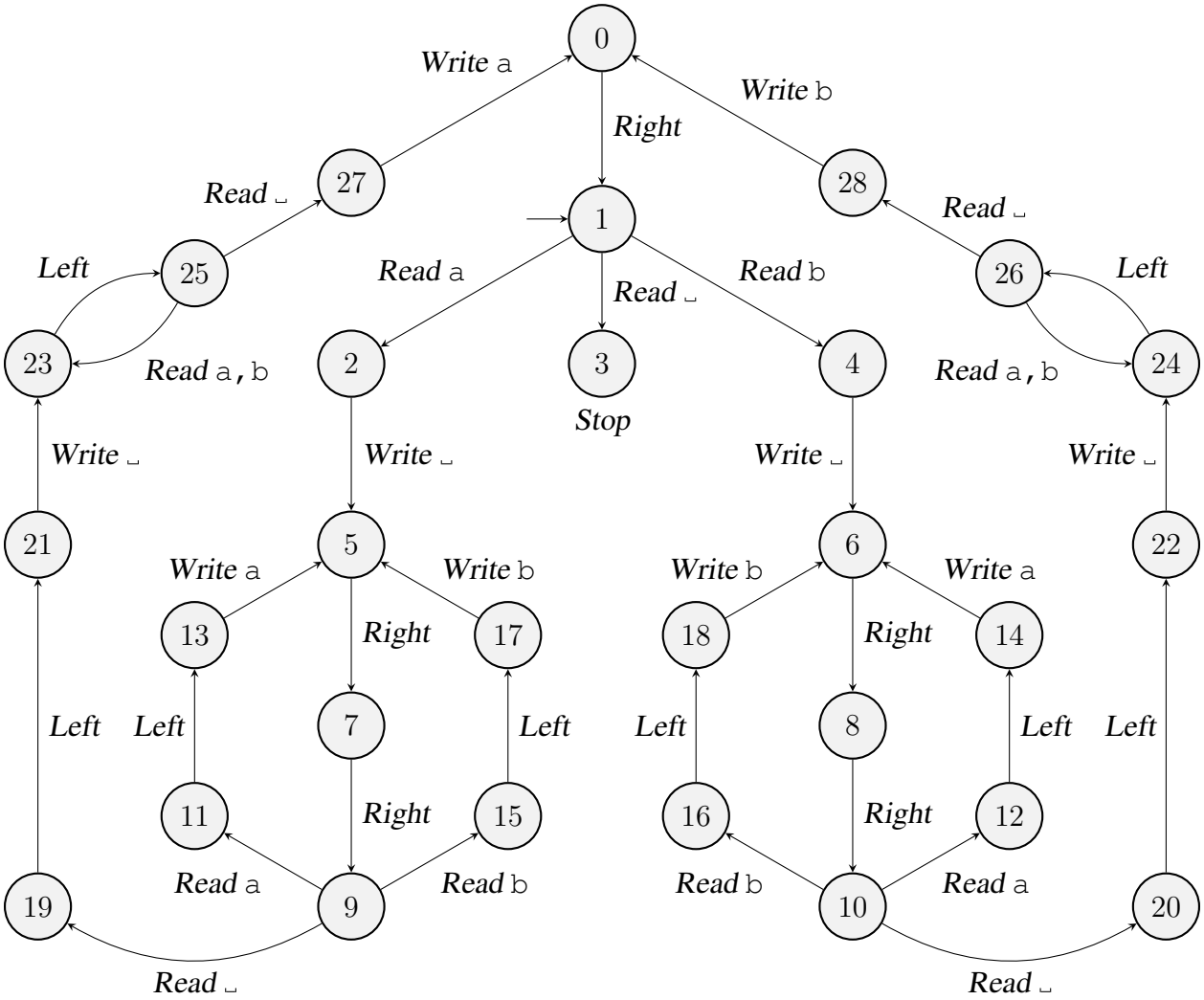
We can combine the simple simulating TMs for the read operations as shown below:



All of these simple TM programs are constant times e.g. each *Read/Write* operation takes 4 steps, therefore, all of these have polynomial time complexity.

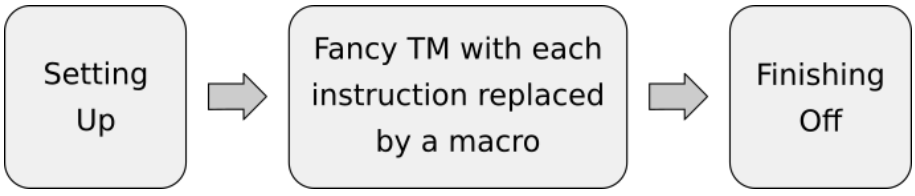
1.4 Converting Simple to Fancy TM's Output – The Finishing Program

In the last step, we want a program that *squashes* a simple tape back into a fancy one that serves as the output. For example, it would squash ȧabbaabbbb into ababb␣. Similar to the *stretching* program, the squashing program will do it one character at a time and its complexity will be quadratic i.e. $O(n^2)$. Here is one possible program for squashing the output tape of a simple (simulating) TM back to the fancy tape.



1.5 Fancy to Simple TM Conversion – Discussion

Given a fancy TM, the corresponding simple TM (expressed using macros) is as follows:



Suppose that the fancy program runs in polynomial time, then consider the following arguments:

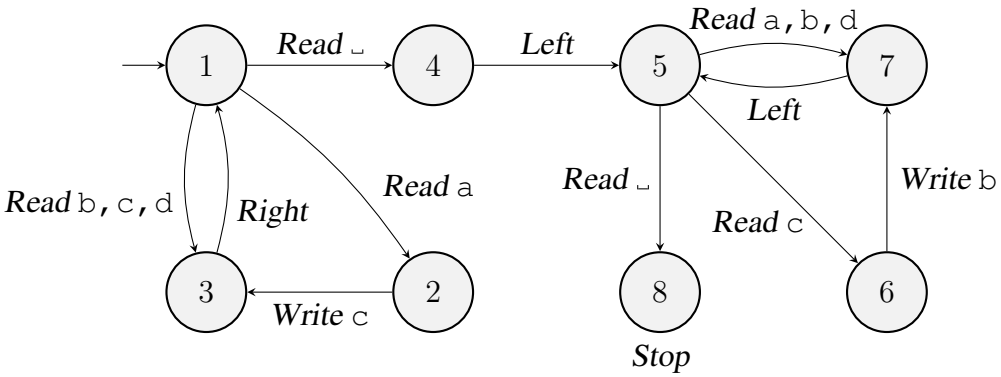
- The stretching program is quadratic time, as discussed earlier.
- Each simulating instruction is constant time, and polynomially many of them happen.
- The squashing program is also quadratic time, and gets applied to the simple tape contents that's twice the size of the corresponding fancy tape contents.

Therefore, the simple TM runs in polynomial time as well.

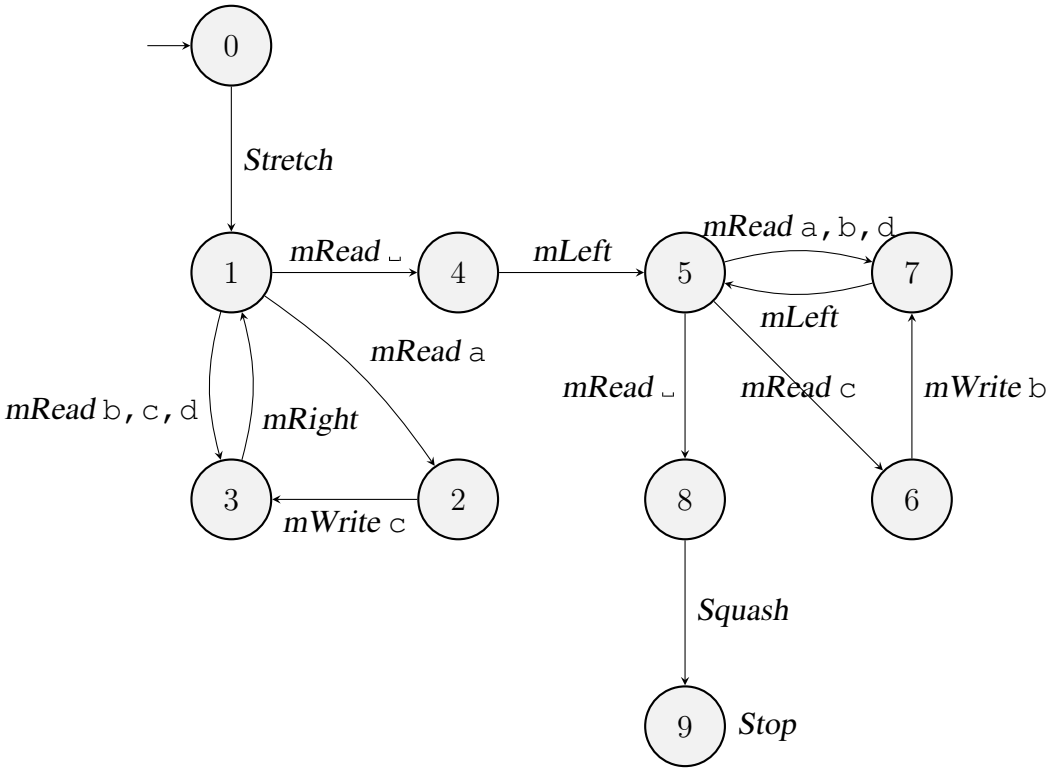
1.5.1 Fancy to Simple TM Conversion – Example

Now lets consider how we can convert a fancy program into a simple program. Let's assume that we have a fancy TM that starts on the leftmost character, changes all `a`'s to `b`'s and ends-up on the cell to the left of input block. Lets assume that the program accomplishes this in two steps:

1. When going to the right, the fancy TM changes all `a`'s to `c`'s
2. When going to the left, the fancy TM changes all `c`'s to `b`'s.



This becomes the following simple TM written using macros (that you can expand, if interested). Each macro name starts with m, in order to differentiate it from the fancy machine instruction, e.g. *mRead* is a macro that replaces the *Read* instruction of the fancy TM.



From the above discussion and examples, we can conclude the following:

1. We can convert a fancy TM using auxialiary characters into a simple TM, so allowing the extra characters does not give us any more power to express functions.
2. If the fancy TM is polynomial time (setting-up, simulating and finishing programs are all polynomial), then the resulting simple TM is also polynomial time.

Nondeterministic Turing machines and NP

1 The complexity class P

So far we've seen various kinds of fancy Turing machines (allowing extra symbols, providing an extra tape, two-dimensional, ...). In each case, we've seen how to convert a fancy Turing machine M into a simple machine $\theta(M)$, in such a way that if M is polytime, then so is $\theta(M)$. This means that the question "is there a polytime machine that solves this problem?" doesn't depend on which kind of machine we use.

As before, we'll write Σ for our (input) alphabet.

Let f be a function from Σ^* to Σ^*. Saying that a Turing machine *computes* the function f means the following. For any word w, if the machine starts with just w on the tape and the head on the leftmost character (or just on a blank if $w = \varepsilon$), the machine eventually halts with $f(w)$ on the tape. For a given function f, we can ask: is there a polytime TM that computes it?

Let L be a language, i.e. a subset of Σ^*. Saying that a Turing machine *decides* the language L means the following. For any word w, if the machine starts with just w on the tape and the head on the leftmost character (or just on a blank if $w = \varepsilon$), the machine eventually returns True if $w \in L$, and False if $w \notin L$. For a given language L, we can ask: is there a polytime TM that decides it? A language thus corresponds to a *decision problem*, i.e. a problem where the answer is True or False.

The complexity class $\mathbf{P}$ is the set of all languages that can be decided in polytime. According to Polly, these are the decision problems that can be solved "fast".

A famous example is the set of prime numbers (represented in binary). Note here that the size of an input is the length of the bitstring, not the number itself. In 2002, it was shown to be in $\mathbf{P}$ by two computer science undergraduates (Kayal and Saxena) and their supervisor (Agrawal).

A machine is said to run in *exponential time* when its running time is $O(2^{n^k})$ for some $k \in \mathbb{N}$. The class of languages that can be decided in exponential time is called $\mathbf{EXP}$.

2 Representing data as words

This is all very well for problems involving words. But what if you want to study computational problems involving other kinds of data? For example, an *ordered pair* of words? Or a graph? Or an integer matrix? Or a database of student records?

In each case, we can devise a way of representing the data as a word. For example, for integer matrices, we can represent all the numbers in decimal, using a minus symbol for negative numbers, a comma to separate entries, and a newline character at the end of each line.

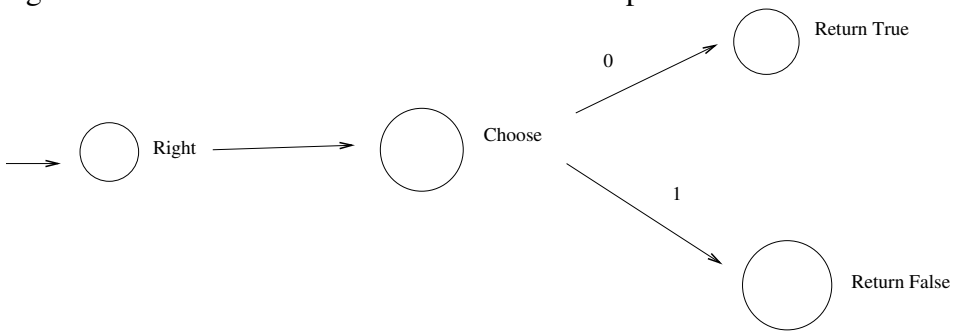
Of course this is just one way of representing an integer matrix as a word. You might invent another one to use instead. But I expect you will find that (a) the two representations differ in length

by a constant factor and (b) you can convert in each direction using a polytime machine. So the choice of representation will not affect what counts as polytime.

In the same way, an ordered pair of words can be encoded as a word, as can a graph, or a database of student records. In summary, any kind of data that can be written out as a finite amount of data can be encoded as a word.

3 Nondeterministic Turing machines

Now we're going to look at a *nondeterministic* Turing machine, which can choose whether to follow an edge labelled 0 or one labelled 1. Here's an example:



Given a word w, we start the machine with just w on the tape and the head on the leftmost character. (It's on a blank symbol if $w = \varepsilon$.) The word is *acceptable* if the machine may return True, and *unacceptable* otherwise. The set L of acceptable words is the *language* of the machine.

A NDTM is *polynomial time* if there's M and C and k such that, for every word w of size $n \geqslant M$, when the machine starts with just w on the tape and the head on the leftmost character, the machine is *guaranteed* to terminate in time at most $C \times n^k$, regardless of the choices made.

The complexity class $\mathbf{NP}$ is the set of all languages that are given by a polytime NDTM. Since a TM is also a NDTM, we can see that $\mathbf{P} \subseteq \mathbf{NP}$. It's also the case that $\mathbf{NP} \subseteq \mathbf{EXP}$. To see this, suppose we have a NDTM that runs in time $C \times n^k$, for large enough n. Then to check all possible choices will take at most $2^{C \times n^k} \times C \times n^k$ steps, which is exponential.

Is it the case that $\mathbf{P} = \mathbf{NP}$? That is, can we convert a polytime nondeterministic machine M into a polytime deterministic machine N with the same language? Thus a word will be accepted by N iff[1] it is acceptable to M.

This is an open question. The accepted hypothesis is that $\mathbf{P} \neq \mathbf{NP}$, but we don't know this for sure. We shall see why this is an important question.

4 Example:Sudoku

Some of you may have tried to solve a 3-Sudoku puzzle.

[1] 'iff' means 'if and only if'

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

It consists of 3×3 subgrids consisting of 3×3 cells. A solution is given in red. Every column, row and subgrid contains all the digits from 1 to 9. A general Sudoku puzzle is similar, but with $n \times n$ subgrids consisting of $n \times n$ cells; you fill it with numbers from 1 to n^2.

There are three computational problems associated with Sudoku.

- The *Sudoku checking* problem. Given a puzzle and a candidate solution, check that the solution is correct.
- The *Sudoku solvability* problem. Given a puzzle, decide whether it's solvable.
- The *Sudoku solution* problem. Given a puzzle, find a solution, or return Impossible.

The first two are decision problems, the last one is not.

The Sudoku checking problem is in $\mathbf{P}$. To check a candidate solution, there are three phases: checking the rows, checking the columns and checking the subgrids. Each row has n^2 entries. To check a row, we check each entry against every other entry, overall $n^2 \times n^2 = n^4$ comparisons (or a bit less). There are n^2 rows, so checking all the rows is $n^2 \times n^4 = n^6$ comparisons. The same for checking the columns and the subgrids. So we have $O(n^6)+O(n^6)+O(n^6) = O(n^6)$ comparisons. The time taken for each comparison is proportional to the length of the numbers, so it is $O(\log n)$ steps. Overall we have $O(n^6) \times O(\log n) = O(n^6 \log n)$ steps, which is $O(n^7)$ steps. Note that we don't bother to write the base on $\log$, because up to a constant factor it doesn't matter.

Furthermore the size of a candidate solution is polynomial in the size of the puzzle. There are at most n^4 cells and each one is $O(\log n)$ bits.

Hence the Sudoku solvability problem is in $\mathbf{EXP}$. Given a puzzle, we can consider all possible candidate solutions, and check each one in polynomial time.

But it's also in $\mathbf{NP}$. Given a puzzle, we *guess* the solution bit by bit, which takes polynomially many steps, then check it, which takes polynomially many steps. A puzzle is acceptable to this machine iff it has a solution.

Thus if $\mathbf{P} = \mathbf{NP}$, then the Sudoku existence problem can be solved in polynomial time.

The solution problem (which is the practically useful one) is not a decision problem at all. But if the existence problem can be solved in polytime, then so can the search problem, by filling the bits one by one.

5 Example: Hamiltonian paths

Given a directed graph, a *Hamiltonian path* from vertex s to vertex t is a path that visits each vertex exactly once. So we have three problems:

- The *Hamiltonian path checking* problem. Given a graph with designated s and t, and a path from s to t, say whether it's Hamiltonian.
- The *Hamiltonian path existence* problem. Given a graph with designated s and t, is there a Hamiltonian path?
- The *Hamiltonian path search* problem. Given a graph with designated s and t, find a Hamiltonian path, or return Impossible.

The checking problem is in $\mathbf{P}$. The existence problem is in $\mathbf{NP}$, since we can guess the path, and its length is linear in the input size. The search is not a decision problem, but if the existence problem in $\mathbf{P}$, then the search problem can be solved in polytime, by filling the bits one by one.

6 Example: Factorization

Given a number N we can check in polytime whether it's prime or composite. But if it's composite, this procedure will not give us the factors. We have three problems.

- The *factor checking problem*. Given N and c, say whether c is a factor of N. This can be done in polytime, using the long division algorithm.
- The *factor existence problem*. Given N and $k < N$, say whether N has a factor that is $\leqslant k$.
- The *factorization problem*. Given composite N, obtain a factor.

The checking problem is in $\mathbf{P}$, by implementing the long division algorithm. The existence problem is in $\mathbf{NP}$, as we guess a factor $\leqslant k$. The factorization problem is not a decision problem, but if the existence problem is in $\mathbf{P}$, then the factorization problem can be solved in polytime. To see this, we can use binary search to find the least prime factor. The number of iterations is linear (proportional to the length of N), and in each one we apply the polynomial algorithm for existence.

7 Example: SAT

A *propositional formula* is built from atoms and connectives $\vee$ (disjunction, "or"), $\wedge$ (conjunction, "and") and $\neg$ (negation, "not"). For example, the formula $(\neg(q \vee p) \wedge r) \vee (p \wedge q)$. An *assignment* says whether each atom is true or false. For example the assignment $p = \mathrm{True}, \mathrm{q} = \mathrm{True}, \mathrm{r} = \mathrm{False}$. This is a *satisfying assignment* for the formula.

Again we get three computational problems.

- The *Formula satisfying assignment checking* problem. Given a formula ψ and an assignment, say whether it's a satisfying assignment for ψ.

- The *Formula-SAT* problem. Given a formula ψ, say whether it's satisfiable.
- The *Formula satisfying assignment search* problem. Given a formula ψ, find a satisfying assignment, or return Impossible.

Here are examples.

- $(p \vee \neg q \vee r) \wedge (\neg p \vee q \vee \neg r)$.
- $(p \vee \neg q) \wedge (p \vee q) \wedge (\neg p \vee \neg q)$

Are they satisfiable?

The checking problem can be performed in linear time. This uses an algorithm called "Shunting Yard" to parse the formula so that it can be easily evaluated.

Furthermore the size of a candidate assignment is linear in the size of the formula.

Therefore Formula-SAT is in $\mathbf{NP}$ because we can guess an assignment, which takes linear time, and then check whether it's a satisfying assignment, which takes linear time.

The search problem (which is the practically useful one) is not a decision problem at all. But if Formula-SAT can be solved in polytime, then so can the search problem, by filling the bits one by one.

SAT is useful for solving constraint problems. Let's see an example. Andy and Barbara each want to meet Chris between 15:00 and 17:00 for an hour, and they also want to meet each other for half an hour. The meeting times are 15:00, 15:30, 16:00 and 16:30. How do we turn this into an instance of SAT? We first give a list of 12 propositional variables:

AC(15:00)	Andy meets Chris at 15:00
AB(15:30)	Andy meets Barbara at 15:30

and so forth. Then we give a formula that expresses all the constraints. Scheduling the meetings is the same as finding a satisfying assignment.

8 The other definition of $\mathbf{NP}$

Apart from the definition using NDTMs, there's another way of defining $\mathbf{NP}$. A language L belongs to $\mathbf{NP}$ when there is a polynomial p and a polytime Turing machine M—called the *checking machine*—such that, for any word w (we write $|w|$ for its length), the following are equivalent:

- $w \in L$
- there is a word x of length $p(|w|)$ such that M accepts the ordered pair $\langle w, x \rangle$.

When this holds, we say that x *certifies* that $w \in L$.

Notice that the Sudoku solvability meets this definition. The word x is the candidate solution and its size is polynomial in $|w|$. (To be more precise, it's bounded by a polynomial p, but we can pad it out with 0's to give it length $p(|w|)$.) And, as we said at the outset, checking whether x is a solution to the puzzle w can be done in polyime.

Let's see why the two definitions are equivalent. (This is an outline argument, not a detailed proof.)

- Suppose that L is the language of a NDTM that, for an input w, runs in time $p(|w|)$. Then the number of choices is at most $p(|w|)$. Given a word w and suggested list of choices x of this length, we can check in time $p(|w|)$ whether this list leads to True being returned.
- Suppose that we have polynomial p and checking machine M. Then we can form a machine that, given w, guesses a word x of length $p(|w|)$ one bit at a time, which takes $p(|w|)$ steps, and then checks it in polytime. Altogether this is a polytime NDTM that recognizes L.

For each part, it's helpful to use auxiliary tapes. These can be removed using the methods we have learnt.

9 Completeness

Let L and L' be languages. A *reduction* from L to L' is a function $f : \Sigma^* \to \Sigma^*$ such that for any bitstring x, we have $x \in L$ iff $f(x) \in L'$. It's a *polytime reduction* when this is done on a (deterministic) polytime machine.

The idea is that if we know how to decide membership of L', then the reduction gives us a way to decide membership of L. If we have a polytime reduction from L to L', and $L' \in \mathbf{P}$, then $L \in \mathbf{P}$.

A language L is said to be $\mathbf{NP}$-*hard* when every language in $\mathbf{NP}$ reduces to it in polytime. It is $\mathbf{NP}$-*complete* when it is in $\mathbf{NP}$ and also $\mathbf{NP}$-hard.

10 Completeness of Formula-SAT and 3CNF-SAT

An important result is that Formula-SAT is $\mathbf{NP}$-complete. Here is an outline of the proof. Suppose $L \in \mathbf{NP}$, i.e. there's a nondeterministic machine M that recognizes L in time bounded by the polynomial p. Given a word w, our task is to construct (in polytime) a formula that's satisfiable iff w is acceptable to M.

If w is acceptable, then there's a trace of machine configurations ending with Return true. There are at most $p(|w|)$ configurations, each of which uses at most $p(|w|)$ characters. So we have approximately $p(|w|)^2$ cell contents, as well as the states and the head positions. Each of these can be expressed by atoms. For example, we have an atom saying that at step 5, cell 17 contains 1; and another saying that at step 5 the head is over state 14; and another saying that at step 5, we're in state 17. Then we write down a load of constraints saying that the first configuration is the one that should be given by w, and each configuration (except the first) follows from its predecessor by following the rules of M, and the last constraint results in returning True.

This construction takes polytime, and the resulting formula is satisfiable iff w is acceptable. As required.

In fact, we can do better, and transform the formula into *3CNF* form. This refers to a formula that is a conjunction of clauses, each of which is a disjunction of three literals. (A *literal* is either an atom p or a negated atom, written $\overline{p}$.) For example:

$$(p \vee \overline{q} \vee r) \wedge (\overline{p} \vee q \vee \overline{s})$$

To sum up, 3CNF-SAT is an $\mathbf{NP}$-complete problem. This is called the *Cook-Levin theorem*. As for 2CNF, it is known to be in $\mathbf{P}$.

11 Other problems

Any problem that we can reduce 3CNF-SAT to in polytime must be $\mathbf{NP}$-hard. By this method, many problems have been shown to be $\mathbf{NP}$-complete, e.g. Sudoku solvability and the Hamiltonian path existence problem. Planning and scheduling problems are examples of $\mathbf{NP}$-complete problems that arise in AI and robotics.

As for factorization, the accepted hypothesis is that this is neither in $\mathbf{P}$ nor $\mathbf{NP}$-hard, but neither of these things is known for sure. The question is critical because RSA encryption is built on the assumption that factorizaion is hard. If it turns out to be easy, then RSA can be cracked.

Decidability

1 Decidable properties and computable functions

Recall the definition we saw earlier. A decision problem is said to be *decidable* when there is some program that, when given an argument, says whether the answer is Yes or No. But what is a "program"? A program in what language?

Let's say we have a decision problem concerning words in our language Σ^*. This can be expressed in two ways:

- As a function Σ^* to $\{\mathrm{Yes}, \mathrm{No}\}$, sending good words to Yes, and bad words to No.
- As a subset of Σ^*, viz. the set of good words.

Let's first think about solving our problem on a Turing machine. The tape alphabet is $\Sigma \cup \{\textvisiblespace\}$ and the set of return values is $\{\mathrm{Yes}, \mathrm{No}\}$. We start execution with a word w on an otherwise blank tape; the head is on the cell to the left of the word. At the end of execution, the head is where it began, the tape is blank, and the machine returns Yes if w is a good word and No if it's a bad word.

Is there such a Turing machine? If there is, we say that the problem is *decidable*. But surely Turing machine gives a rather restrictive notion of an algorithm? Maybe we should be more liberal. Maybe we should allow two-tape Turing machines, or two-dimensional Turing machines. Maybe we should allow programs written in a language with infinitely many integer variables and string variables, and `for` loops and `while` loops and recursive procedure calls and pointers and storable labels and ...

Many different notions of "algorithm" have been proposed that, at first sight, appear to be more expressive than a Turing machine. But every time, it has turned out that—as far as *decision problems on words* are concerned—the fancy notion is no more expressive than a Turing machine. If, in my fancy new programming language, I can solve such a decision problem, I could already solve it on a Turing machine.

This is remarkable. It tells us that the notion of decidability is *robust*. All these very different definitions turn out to be equivalent.

Perhaps somebody in the future will invent a new notion of "algorithm" that clearly counts as algorithmic and yet can solve decision problems on words that a Turing machine cannot? *Church's thesis* (named after Alonzo Church) says that this will not happen.

Church's thesis states: "any decision problem on words that can be solved by an algorithm can be solved by a Turing machine".

Another version is for functions $f\colon \Sigma^* \to \Sigma^*$. In this case the Turing machine must start with a word w on an otherwise blank; the head is to the left of the word. At the end of execution, the machine must stop with the word $f(w)$ on an otherwise blank; the head is to the left of the word. When there is such a machine, we say that f is *computable*. (Older literature uses the word

"recursive" instead.) Church's thesis for functions states: "any functions on words that can be computed by an algorithm can be computed by a Turing machine".

Let me again emphasize that natural numbers, integers, lists of words etc. can all be encoded as words, just as they can all be encoded as natural numbers. Also, we may talk about undecidable *problems*, undecidable *properties*, undecidable *subsets* and undecidable *languages*. But the idea is the same in all cases, despite the varying terminology.

2 Primitive and Basic Java

Here is a different viewpoint. Let's give the name *Primitive Java* to a Java-like language with only the type `nat`, for (unlimited) natural numbers. It has the the following facilities.

- Create a variable `nat i = 0`.
- Increment a variable `i++`.
- Decrement a variable `i--`. This does nothing if `i==0`.
- Conditionals `if i == 0` $\{M\}$ `else` $\{N\}$.
- Repetition a fixed number of times `repeat i times` $\{M\}$.

Just to get started, here are some useful encodings. We can encode `nat j = i` as follows:

```
nat j = 0;
repeat i times {j++;}
```

We can encode `j = 0` as follows:

```
repeat j times {j--;}
```

We can encode `j = i` as follows:

```
j = 0;
repeat i times {j++;}
```

We can encode `if i <= j` $\{M\}$ `else` $\{N\}$ as follows:

```
nat k = i;
repeat j times {k--;}
if k == 0 {M} else {N}
```

We can encode `if i < j` $\{M\}$ `else` $\{N\}$ as `if j <= i` $\{N\}$ `else` $\{M\}$. We can encode `if i == j` $\{M\}$ `else` $\{N\}$ as follows:

```
if i < j {N} else {if j < i {N} else {M}}
```

A program has several (immutable) parameters `input0`, `input1`, `input2`, etc., and a variable `output` that's initialized to 0. For example, here is a program that computes the sum of `input0` and `input1`.

```
nat output = 0;
repeat input0 times {
   output++;
}
repeat input1 times {
   output++;
}
```

If `input0` $== 3$ and `input1` $== 5$, then execution terminates with `output` $== 8$.

Note what's lacking: while-loops and recursion. Let's give the name *Basic Java* to Primitive Java extended with while-loops: `while i > 0` $\{M\}$. For example, we can encode `hang` as follows:

```
nat i = 0;
i++;
while i > 0 { }
```

To encode `while` c $\{M\}$ for a condition c we write

```
nat k = 0;
if c { k++; } else { }   // Sets k to be 1 if c is true, else 0.
while k > 0 {
   M
   k = 0;
   if c { k++; } else {}
}
```

where `k` is a *fresh* variable, i.e. one that doesn't appear in M.

3 Computing functions

Consider a program that is k-*ary*, i.e. has k inputs. If the program is in Primitive Java, every list of inputs $\mathbf{x} \in \mathbb{N}^k$, it will terminate with an output. Thus the program computes a (total) function from $\mathbb{N}^k$ to $\mathbb{N}$. If the program uses while-loops, there's a possibility of nontermination. So the program computes a *partial* function. For example, the following unary program

```
nat output=0;
nat j=0;
j++;
j++;
while (j > 0) {
  if (input0 <= j){
    j--;
    output++;
  }
}
```

will return an output of 2 if the input is 0 or 1, but otherwise will hang. So it computes the partial function that sends 0 and 1 to 2 and is undefined on all numbers $>= 2$.

We can translate between Basic Java and Turing machines, in such a way that the translation preserves the meaning: i.e., the translated program computes the same partial function as the original one. Thus the partial functions that are expressible in Java are precisely the computable ones.

Any function that can be computed by a Primitive Java program is said to be *primitive recursive*. The important case is where $k = 1$, because a k-tuple of natural numbers can be encoded as a single one. Instead of (or as well as) natural numbers, we could use integers or strings or lists of natural numbers, provided we modify Primitive Java to include suitable operations.

The primitive recursive functions include all the familiar functions such as comparison, multiplication, exponentiation, factorial and so on. They also include all functions that can be computed in a Turing machine in polynomial time, exponential time and much more. People used to think that they were the only functions on $\mathbb{N}$ that could be computed algorithmically.

However in 1928, Wilhelm Ackermann made a shock discovery: a function that can be computed algorithmically, but is not primitive recursive. The Ackermann function (actually a simplified version due to Rózsa Péter) takes two arguments, and is given as follows:

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)) \end{aligned}$$

This is a program using recursion. Here is a proof that it terminates.

For $m, n \in \mathbb{N}$, let $Q(m,n)$ be the statement that the evaluation of $A(m,n)$ terminates. We prove $\forall m \in \mathbb{N}.\, \forall n \in \mathbb{N}.\, Q(m,n)$ by induction.

- The base case says $\forall n \in \mathbb{N}.\, Q(0,n)$. It holds since $A(0,n)$ returns $n+1$.
- For the inductive step, we suppose $\forall n \in \mathbb{N}.\, Q(m,n)$ and want to prove $\forall n \in \mathbb{N}.\, Q(m+1,n)$. We do this by induction.
  - The base case says $Q(m+1,0)$. This holds because $A(m,1)$ returns a value p, by the outer inductive hypothesis *i.e.* $Q(m,n)$, so $A(m+1,0)$ also returns p.
  - For the inductive step, we suppose $Q(m+1,n)$ and we want to prove $Q(m+1,n+1)$. Then $A(m+1,n)$ returns an answer p, by the inner inductive hypothesis *i.e.* $Q(m+1,n)$, and $A(m,p)$ returns a value q, by the outer inductive hypothesis *i.e.* $Q(m,n)$, and so $A(m+1,n+1)$ also returns q.

The Ackermann function cannot be written in Primitive Java (though I won't prove this). However, it *can* be written in Basic Java: the rough idea is to represent the stack of recursive calls as an extra parameter. This shows the importance of while-loops even for the purpose of computing total functions on natural numbers.

Church's thesis says that we cannot extend Basic Java to a language that can express *more* partial functions from $\mathbb{N}^k$ to $\mathbb{N}$ and yet is still a *programming language* i.e. the programs written in it can be computed algorithmically.

A language like Basic Java that can express all computable total functions is said to be *Turing-complete*. We have seen that Primitive Java is not Turing-complete, as it cannot express the Ackermann function. On the other hand it has the advantage of being a *total* language, i.e. all programs terminate. Would you rather use a language that is Turing-complete, or one that is total? Unfortunately, it turns out that no language can be both! Most languages used in practice (if they allow unlimited natural numbers) are Turing-complete, like Basic Java. But a few, such as Agda, are total and Turing-incomplete, like Primitive Java.

4 Examples of decidable and undecidable properties

Here are some examples of problems and their decidability status.

- The problem of saying whether two regular expressions are equivalent. This is decidable, as we learnt earlier in the term.
- The problem of saying whether a given context-free grammar accepts a given word. This is decidable.
- The problem of saying whether a given context-free grammar accepts any word at all. This is decidable.
- The problem of saying whether a given context-free grammar accepts every word (for the alphabet Σ). This is undecidable.
- The problem of saying whether a given context-free grammar is ambiguous. This is undecidable.
- The problem of saying whether a given integer-coefficient polynomial in one variable has a rational solution.[1] This is decidable.
- The problem of saying whether a given integer-coefficient polynomial in several variables has a rational solution. It is currently unknown whether this is decidable or not.
- The problem of saying whether a given integer-coefficient polynomial in several variables has an integer solution. This is undecidable, as proved in 1970 by Matiyasevic, Robinson, Davis and Putnam. The question was originally asked by Hilbert in a famous lecture in 1900, although nobody at that time had a precise definition of "decidable".
- The problem of saying whether a given propositional formula is true for all interpretations of the propostional atoms. (Such a formula is called a *tautology.*) This is decidable—just evaluate the formula for each assignment.
- The problem of saying whether a given first-order formula is true for all interpretations of the predicate symbols. This is undecidable.

There are many other examples of decidable and undecidable problems. We'll see some of them in the coming weeks.

[1] A *solution* of a polynomial is an assignment of values to the variables that makes the polynomial equal to zero.

5 Semidecidable properties

A property P of natural numbers is said to be *semidecidable* when there's a program M that, when executed on a number n, returns True if n satisfies P, and runs forever otherwise. (For decidability, the program would have to return False otherwise.)

Here's an example: the problem of saying whether a given integer-coefficient polynomial p in several variables has an integer solution. Let's see that this is semidecidable.

First of all, recall that we can treat lists of integers as natural numbers. So consider the following program. Given polynomial p in k variables, apply p to the first list of k integers, then to the second list of k integers, then to the third list, and so on. If any of these calculations returns 0, stop and return True. Since every list of k integers appears somewhere in this enumeration, the program will stop and return True if p has an integer solution . But if p has no integer solution, then the program will run forever.

What is the relationship between semidecidability and decidability?

- Any property P that is decidable must be semidecidable. For if M is a program the decides it, then here is a program that semidecides it. For any natural number n, run M on it, and if this returns True, then return True, but if it returns False, then run forever.

- If both P and the negation of P are semidecidable, then P is decidable. For if M is a program that semidecides P, and N is a program that semidecides the negation of P, then here is a program that decides P. For any natural number n, run M for one step, then N for one step, then M for another step, then N for another step, and so forth. If one of the steps of M returns True, then return True, but if one of the steps of N returns True, then return False.

We conclude that a property P is decidable iff both P and the negation of P are semidecidable. (To see the forwards direction, note that if P is decidable, then the negation of P must also be.)

The Halting Problem

1 Introducing the halting problem

Is there any decision problem, any set of words, that is undecidable? The answer is clearly yes. There are only countably many Java programs (or Turing machines), but uncountably many sets of words. But we'd like to see a specific example of an undecidable problem. And so far, I've given you some examples, such as ambiguity of context free grammars. But I haven't proved that these properties are undecidable.

Now I'm going to tell you the most famous example of an undecidable problem in computer science: the halting problem: to distinguish between those nullary programs that halt and those that hang. (A *nullary* program is one with no arguments.)

For example, the following program halts:

```
void programA () {
  nat sum = 0;
  nat count = 0;
  while (count <= 5) {
    sum = sum + count;
    count = count + 2 - 1;
  }
  if (count < 100) {
    return;
  } else {
    while (true) {}
  }
}
```

The following program hangs:

```
void programB () {
  nat sum = 0;
  nat count = 0;
  while (count <= 5) {
    sum = sum + count;
    count = count + 1 - 1;
  }
  if (count < 100) {
    return;
  } else {
```

```
    while (true) {}
  }
}
```

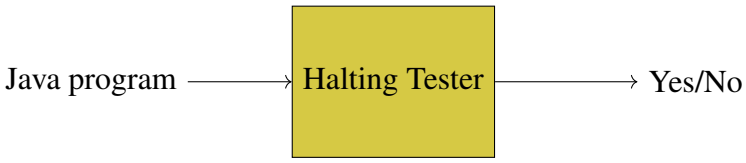
The next example is based on *Goldbach's conjecture*—the statement that every even number $\geqslant 4$ is a sum of two primes. It is currently unknown whether this is true or false.[1] Here's a program that hangs if Goldbach's conjecture is true, and halts if it's false:

```
boolean isprime(nat n) {
   for (nat i = 2, i < n, i++) {
      if (n mod i == 0) {return false;}
   }
   return true;
}

boolean isasumoftwoprimes(nat n) {
   for (nat i = 2, i < n, i++){
      if (isprime(i) and isprime(n-i)) {return true;}
   }
   return false;
}

void programC (){
   nat k = 4;
   while(isasumoftwoprimes(k)){
      k = k+2;
   }
}
```

Our question is whether the halting problem is decidable. Thus we seek a program



For example, if we feed in Program A, it returns True. If we feed in Program B, it returns False. If we feed in Program C, it return False if Goldbach's conjecture is true, and True otherwise.

Turing proved that the halting problem is undecidable, i.e. there is no such tester. This is called the *Halting Theorem*.

2 Proof of the Halting Theorem

The proof of the Halting Theorem is as follows. Suppose it is decidable.

[1] It has been checked mechanically that every even number from 4 to 4×10^{18} is a sum of two primes. But, for all we know, there might be a larger even number that isn't.

1. Now consider the *unary halting problem*: given a unary Java method

```
void f (String x){
   ...
}
```

   and a string y, does `f` terminate when called with argument y? We can reduce the unary halting problem to the nullary one: given a unary method `f` and a string x, obtain a nullary method `g` by taking the code of `f` and replacing `x` with y; then `g` terminates when called if `f` terminates when called with argument y. So since the nullary problem is (we're assuming) decidable, the unary one is too. So we have a program

```
boolean haltcheck (String somemethod, String y)
```

   where `somemethod` is the body of a unary method. When applied to M and y, this method returns true if M applied to y terminates, otherwise it returns false.

2. Next we turn this into a program

```
void hangcheck (String somemethod, String y){
  if haltcheck (somemethod, y) {
    while true {}
  } else {
    return;
  }
}
```

   This method, when applied to M and y, hangs if M applied to y terminates, otherwise it returns.

3. Next we turn this into a program

```
void doublehang (String y){
   hangcheck(y,y)
}
```

   This method, when applied to y (the body of a unary method), will hang if y applied to y terminates, otherwise it returns.

4. Finally let z be the body of `doublehang`. We see that `doublehang`, when applied to z, terminates iff it hangs. Contradiction.

Rice's theorem

1 Undecidability by Reduction

Let P and Q be problems. To *reduce* problem P to problem Q means to give a way of solving P using a black box that solves Q. For example, I saw a recipe for profiteroles that said "take some pastry balls" and "take some chocolate sauce". The author reduced the problem of making profiteroles to the problems of making pastry balls and making chocolate sauce.

Suppose we've reduced problem P to problem Q.

- If Q is decidable, then P is decidable
- If P is undecidable, then Q is undecidable.

Now that we know the halting property is undecidable, we can deduce the undecidability of many other properties. Here's an example. A program

```
void f () {
  ...
}
```

is *orange* when it both halts and contains (in the body code) more occurrences of "a" than "b". Thus

```
int a = 3;
return;
```

is orange, but

```
int b = 3;
return;
```

is not, and

```
int a = 3;
while (true) {}
```

is not. Orangeness is undecidable: to prove this fact, we reduce the halting problem to the orangeness problem. For any program C of type `void`, let $F(C)$ be the same code as C with an extra comment line at the end consisting of just enough occurrences of "a" so that there are more occurrences of "a" than "b". Then C halts iff $F(C)$ is orange. So if orangeness were decidable, then the halting property would be too.

2 Semantic and non-semantic properties

As you may have noticed, orangeness is a rather strange property. Just look at the examples above: the program

```
int a = 3;
return;
```

is orange, but

```
int b = 3;
return;
```

is not. Yet these programs have exactly the same *semantics*, i.e. the same behaviour observable by an external user. While orangeness is a property of code, it isn't a property of behaviour.

When you want to test that code is good or bad, you typically aren't interested in properties like orangeness. The properties you are interested in are properties of behaviour. They are called *semantic properties*. For example:

- Does this software print only polite words?
- Does this software provide good advice to all users, and excellent advice to premium users?
- Does this software, when supplied with two positive integers, always return the highest common factor?

All these questions concern only the behaviour of the program and nothing else. They are semantic properties. It would be great if we could check them automatically.

Well, we can't. *Rice's Theorem* says that (except in two cases, which I'll come to later), every semantic property is undecidable.

Recap To show a property (such as orangeness) is not semantic, give two programs that have the same semantics (behaviour), but one of them has the property and the other one doesn't. In the absence of such a pair, the property is semantic.

Exercise A program

```
void f () {
  ...
}
```

is *red* when it prints a friendly greeting, and the code comments include a poem. Show that redness is not semantic.

3 The two exceptions

Look at the following example:

- Does this software (of type `nat`) return a number that is greater than itself?

This is a semantic property: it concerns the behaviour of a program. But the answer is always No. So this property is decidable: we can test for it by a one-line program that just returns False.

Now look at the following example:

- Does this software (of type `nat`) either hang, or return a number that is equal to itself?

This is a semantic property: it concerns the behaviour of a program. But the answer is always Yes. So this property is decidable: we can test for it by a one-line program that just returns True.

Now look at the following example:

- Does this software print only polite words?

This is a semantic property: it concerns the behaviour of a program. There's some program that satisfies it (i.e. prints only polite words), and some program that doesn't. Rice's Theorem tells us that it is undecidable:

To summarize, there are three kinds of semantic property:

- A property that never holds. This is decidable.
- A property that always holds. This is decidable.
- A property that holds in some case and fails to hold in some case. This is undecidable.

Rice's Theorem *Any semantic property of code that holds in some case and fails to hold in some case is undecidable.*

4 Proving Rice's Theorem

To prove Rice's Theorem, we use the same method that we've seen before: reduction of the Halting Problem. There are actually two kinds of situation. Look at these examples:

1. Consider the following example:

   - A method `void f ()` is *polite* when it prints only polite words.

   Note that the code

   ```
   while (true) {}
   ```

   is polite but the code

   ```
   System.out.println("You #$@&%*!");
   ```

   is not. So given any *plain* program code P of type `void` (no I/O, exceptions etc.), we see that P halts iff the code

   P

   ```
   System.out.println("You #$@&%*!");
   ```

is not polite.

2. Now look at this example:

   - The software `void f (boolean premium)` is *helpful* when it prints helpful advice to all users, and excellent advice to premium users.

   Note that the code

   ```
   while (true) {}
   ```

   is not helpful but the code

   ```
   if premium {
     System.out.println("Unpack all definitions before "
         + "attempting a proof");
   } else {
     System.out.println("Check your answer before "
         + "handing it in.");
   }
   ```

   is helpful. So given any *plain* code P of type `void` (no I/O, exceptions etc.), we see that P halts iff the code

   ```
   P
   if premium {
     System.out.println("Unpack all definitions before "
        + "attempting a proof");
   } else {
     System.out.println("Check your answer before "
        + "handing it in.");
   }
   ```

   is helpful.

Now in general: given a semantic property $\mathcal{R}$ that holds in some case and fails to hold in some case, we ask whether the code `while (true)` that just hangs has the property.

- If it does, then there must be some other piece of code M that doesn't. Now for any *plain* code P of type `void`, let $F(P)$ be the code

  $$P$$
  $$M$$

  If P halts, then $F(P)$ has the same semantics as M, so $F(P)$ doesn't satisfy $\mathcal{R}$, just like M doesn't. If P hangs, then $F(P)$ has the same semantics as `while (true)` , so $F(P)$ satisfies $\mathcal{R}$, just like `while (true)` . To summarize, P halts iff $F(P)$ does not have the property $\mathcal{R}$. So we have reduced the halting problem to $\mathcal{R}$. So $\mathcal{R}$ is undecidable.

- If it doesn't, then we apply the same argument the other way round.

5 Decidability in Practice

Let us look at how undecidability questions crop up in practice.

Verification. Because it is so common to write bugs when programming, some people have developed a more formal approach to software development. Someone first writes a *specification*, which describes properties of the code. (This may be written in a specification language such as Z.) Then the program is written. Finally we want to check the program automatically to see if it meets the specification. However, this is undecidable. In some cases, the verifier will say "Yes, the code meets its spec." In other cases it will find bugs. But necessarily there will be cases where the verifier cannot establish whether the program meets its spec or not.

Type reconstruction. In some programming languages, such as Haskell and ML, variables don't need to be declared, because the system works out the correct type automatically. However, while this works for these languages, it would not work for a stronger language than Haskell, because general type inference is undecidable. Knowing this, the designers of Haskell deliberately constrained the language so as to have automatic type inference for it.

Mobile code and viruses. If you allow code from an external source to run on your machine, then you run the risk of that code performing destructive actions on your data. It would be extremely useful if we could test code automatically for potential malicious behaviour. Again, this is impossible.

There are two partial solutions to this problem.

- Virus checking software detects some viruses but it is incomplete. No matter how sophisticated it is, there will always be damaging code that it is not able to recognize.
- Alternatively, we can be overly conservative: run code in a *sandbox* that guarantees that it can't access any important data, except in cases where we know that the access is safe. But there will always be some safe kinds of execution that we disallow.

Logic. Proving theorems is hard! Wouldn't it be great if we could determine automatically which statements are provable? But even in very simple kinds of logic (e.g. predicate logic), this is undecidable. And the reason is familiar: reduce the Halting Problem to provability of sentences in predicate logic.