Automata Theory

Aditya Neeraje

June 2024

1 Introduction

Automata is the study of abstract computing devices that are capable of passing between states when faced with certain inputs along. Automata theory deals with concepts such as finite automata - which are very useful in checking whether models satisfy certain properties, regular expression checking - widely used to search for expressions when the relevant database can rapidly change, etc., Turing machines - which help us understand what can and cannot be computed, complexity theory - which helps understand what cannot tractably be computed for all but very small input sizes.
Some applications of automata theory include:

1. Software for designing and checking behaviour of systems that can be modeled, such as digital circuits.
2. The "lexical analyzer" of a typical compiler, which breaks down the input program into a series of tokens and can generate parse trees.
3. Software for finding occurences of words, phrases or other patterns.
4. Software for vertifying that certain finite systems possess properties expected of them, such as communications protocols or cryptographic protocols.

1.1 Alphabets

An alphabet is a finite, non-empty set of symbols, conventionally represented by Σ. Examples include the binary alphabet $\{0, 1\}$ and the set of lowercase ASCII letters, etc.

Definition 1.1. A word is a sequence of (not necessarily non-zero) characters taken from an alphabet.

The length of a word is defined as the number of characters it has.

Definition 1.2. Powers of an alphabet:
We define Σ^k to be the set of all letters of length k made using characters from

the alphabet Σ. Σ^0 is conventionally the set consisting of the empty string ϵ, whose length is 0.

The set of all strings over an alphabet is conventionally denoted $\Sigma*$. For instance, $\{0,1\}* = \epsilon, 0, 1, 00, 01, 10, 11, 000, \cdots$.

$$\Sigma* = \bigcup_{i=0}^{\infty} \Sigma^k$$

Similarly, we define Σ^+ as

$$\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^k$$

Alternatively,

$$\Sigma* = \Sigma^+ \cup \{\epsilon\}$$

Definition 1.3. Concatenation of strings:
Let x and y be strings. Then xy is the concatenation of x and y.

Note that the concatenation operator is not commutative but is associative.

1.2 Languages

A language is a set of strings chosen from the closure $\Sigma*$ of an alphabet Σ.

1. $\varnothing$, the empty language, is a language over any alphabet.
2. $\{\epsilon\}$, the language consisting of only the empty string, is also a language over any alphabet. Note the difference between this and $\varnothing$. $\varnothing$ has no strings, whereas there does exist one string in this set - the empty string.

2 Deterministic Finite Automatons

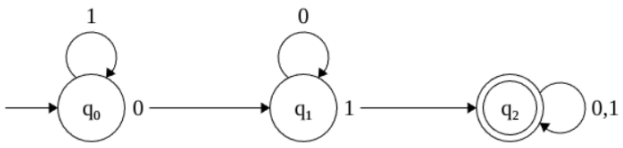
Definition 2.1. A DFA consists of:

1. A finite set of states, Q. In a pictorial representation, each state is generally represented as a circle.
2. An alphabet, Σ
3. A transition function that takes as an argument a state and a symbol and returns a state. In a pictorial representation, we denote this by arcs from the previous state q to $f(q, a)$, labeled by a, for each symbol a in the language[1].

[1]Often, it is of use to neglect pictorially certain transitions that make the system go to a "dead" state from which no action results in a change of state and which is itself not an accepting state.

4. A start state, which is one of the states in Q.
5. A set of accepting states F. If a particular run of a DFA terminates at an accepting state, we consider that the property we were checking for has been satisfied.

A DFA can also be represented pictorially as expressed above, or as a table, with the states on one axis, appropriate symbols to indicate start and accepting states, the characters in the language on the other axis and the corresponding resulting states as the elements of the matrix.
Example: Design a DFA which accepts all binary strings which contain 01 in consecutive positions.
Solution:



Definition 2.2. The Extended Transition Function:
The transition function gives the result of a particular (state,action) pair. The extended transition function $\tilde{\delta}$ returns the unique final state for a given pair of initial state and input string. $\tilde{\delta}(q, \epsilon) = q$, $\tilde{\delta}(q, w) = \delta(\tilde{\delta}(q, x), a)$, if a is the last letter of w and x is the string w without the last letter.

Example: Design an automaton to model the language which accepts all strings which have an even number of both zeros and ones.
Solution:

State	**0**	**1**
$\rightarrow *0,0$	$1,0$	$0,1$
$0,1$	$1,1$	$0,0$
$1,0$	$0,0$	$1,1$
$1,1$	$0,1$	$1,0$

Table 1: Transition table for the DFA

Exercise: Give a DFA which accepts all strings ending in 00.
Solution:

State	**0**	**1**
$\rightarrow 1$	2	1
2	3	1
$*3$	3	1

Exercise: Give a DFA which accepts all strings which contain 000.
Solution:

State	**0**	**1**
$\rightarrow 1$	2	1
2	3	1
3	4	1
$*4$	4	4

Exercise: Give a DFA which accepts all strings which contain 011.
Solution:

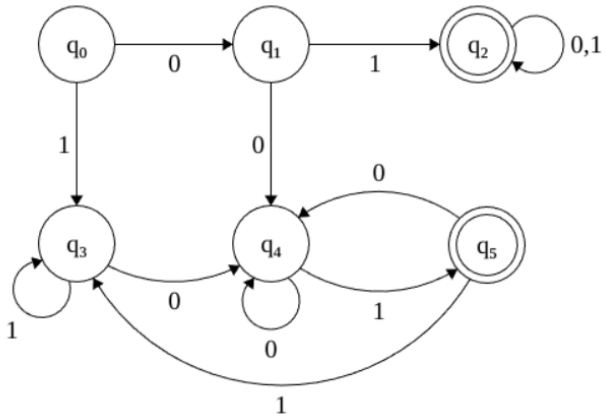
State	**0**	**1**
$\rightarrow 1$	2	1
2	2	3
3	2	4
$*4$	4	4

Exercise: Give a DFA accepting all strings such that each block of 5 consecutive symbols contains 2 zeros.
Solution: Note that here, it is easier to find a DFA which accepts all strings which have at least 4 letters and contain a sequence of 4 consecutive 1s and then take its negation.

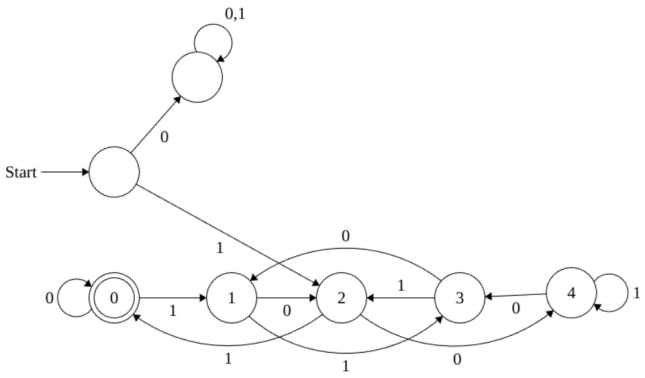
State	**0**	**1**
$\rightarrow *1$	1	2
$*2$	1	3
$*3$	1	4
$*4$	1	5
5	5	5

Exercise: Find a DFA which accepts the set of all strings whose tenth symbol from the right is 1.
Solution: We claim that no DFA having fewer than 2^{10} states can satisfy this. If there was such a DFA having fewer than 2^{10} states, and we consider all 2^{10} binary strings of length 10, two of them must end at the same state. But since they are distinct, they have to differ at, say, the i^{th} position from the left. If we extend both these strings by the same i-1 characters, they both end up at the same state. However, one of them has a 1 at the tenth last digit and the other has zero, which is not possible.
Hence, any DFA has to have at least 2^{10} states. Let the start state be state 0. For any input k, transition from state i to state $(10 * i + k) \mod 1024$. States 512 to 1023 are the accepting states. This is the required DFA.

Exercise: Give a DFA accepting all strings that either begin or end (or both) with 01.
Solution:

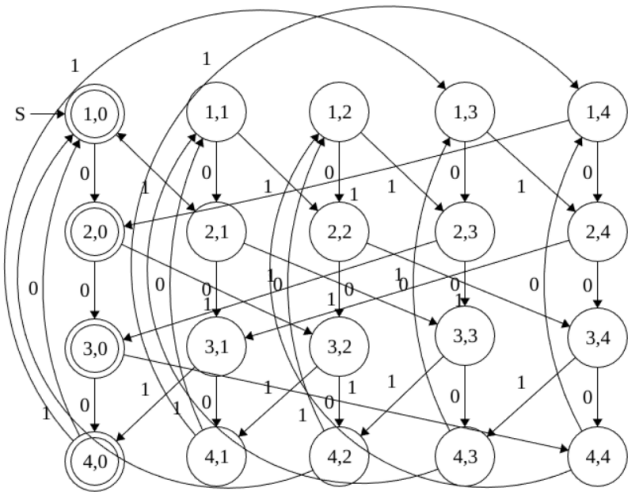


We claim that the above automaton, with q_0 as the start state, satisfies this condition. If the string starts with 0, and the next letter is 1, the automaton immediately moves to an accepting state which loops back to itself until the string terminates. If the second letter is 0, the string moves to the branch of the automaton checking if the string ends with 01. If the first letter is 1, the automaton immediately moves to the branch checking if the string ends with 01.
Exercise: Design a DFA accepting the set of strings which begin with 1 and are interpretable as a multiple of 5 expressed in binary. Eg. 101, 1010, 1111, 10100, etc.
Solution:

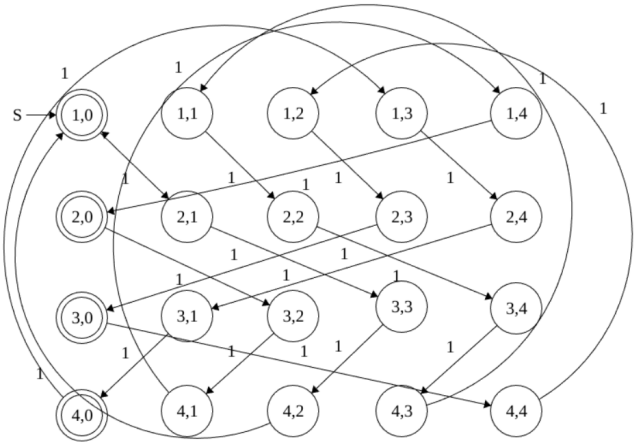


The above DFA, whose bottom 5 states represent the string read so far modulo

5, is the required DFA. The accepting state is the one where the string has concluded with us reading a number which is 0 mod 5.
Exercise: Give a DFA which accepts the set of strings whose reverse can be interpreted as representing multiples of 5. Eg. 101, 10011, 0,1001100,0101.
Solution:



The above picture represents the full automaton. For extra clarity. I have removed the transitions corresponding to 0 in the picture below. All those transitions go from state (a,b) to ((a+1) mod 5, b).



Exercise: Give a DFA accepting all binary strings with the number of 0's divisible by 5 and the number of 1's divisible by 3.

Solution: Similar to the DFA for all binary strings with an even number of 0s and 1s. The solution is left to the reader.
Exercise: Describe in natural language (english) the language accepted by the DFA, and prove the same by induction on the length of an input string.

State	0	1
$\rightarrow *0$	0	1
$*1$	1	0

Solution: The number of 1s in the input string is odd. Proof left to the reader.

3 Non-Deterministic Finite Automata

A non-deterministic finite automaton, or NFA, does not necessarily have a unique output for every state, action transition. Instead, the transition function can be viewed as mapping from a state and action to the power set of states. Some state, action pairs need not even have a resulting state, which we interpret as the result of the transition function being the empty set. A NFA accepts a word if at least one of the states at which the word can terminate is an accepting state. In a sense, the NFA repeatedly guesses that a word could go along one path, and backtracks if it finds that that path ends at a non-accepting state. If all paths end at non-accepting states, the NFA does not accept the word.
It is easy to observe that all DFAs can be converted to NFAs. The transition function of a DFA can be modified to map to the singleton set containing the resultant state of a state, action pair.
A very powerful property of automata is that all NFAs (and in fact all ϵ-NFAs, which we will discuss later), are equivalent to some DFAs.

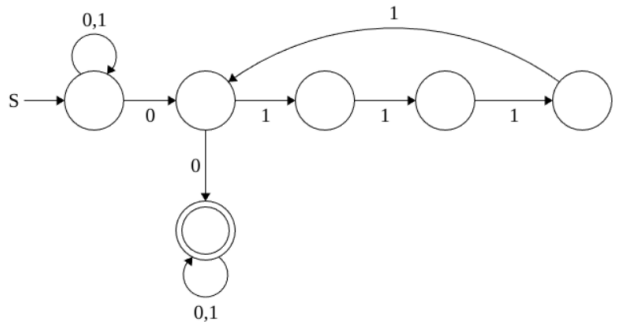
3.1 Equivalence of DFAs and NFAs

Although there are many languages for which an NFA is easier to construct than a DFA, every language that can be described by an NFA can be described by a DFA. In the worst case, this DFA has 2^n states if the smallest NFA for the language has n states. This can be proven using *subset construction*. If an NFA is $(Q_N, \Sigma, \delta_N, q_0, F_N))$, the corresponding DFA is $(2^{Q_N}, \Sigma, \delta_D, \{q_0\}, F_D)$. F_D is the set of all subsets of Q_N which contain at least one accepting state of N. The states of D are the subsets of the power set of the set of states of N.
It only remains to explain how the transition function of D works. For each set $S \subset Q_N$ and for each input symbol a in Σ,

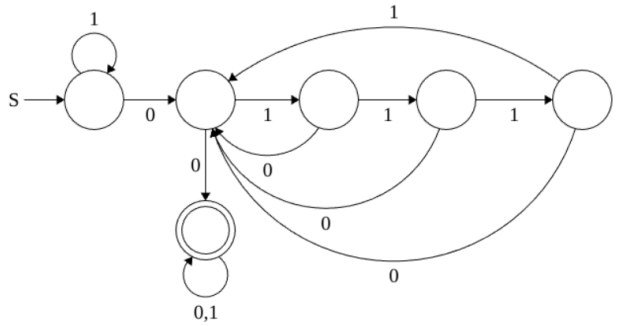
$$\delta_D(S, a) = \bigcup_{pinS} \delta_N(p, a)$$

This construction yields a DFA, but it need not be the simplest DFA we can achieve for the language. There might be states which we cannot achieve by

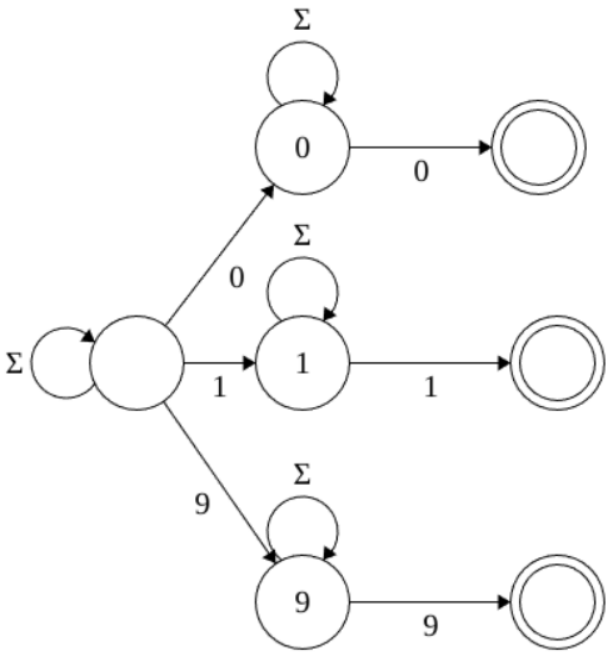
following the directed arrows out of the start state, and these can be found using a BFS or DFS in $O(n^2)$ time at worst and deleted. NFAs often make it very convenient to represent certain languages in a more concise way than DFAs do. The following NFA accepts all binary strings in which some two consecutive zeros are separated by a number of positions which is a multiple of 4.



In this particular case, the DFA is fairly simple too:



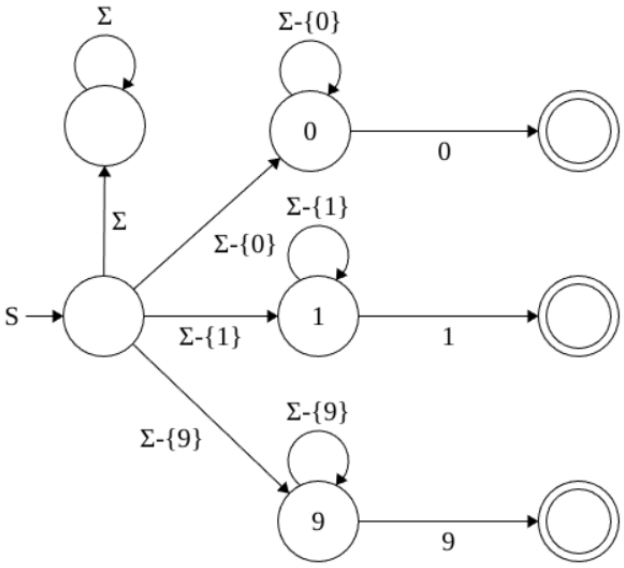
Exercise: For a case with a simple NFA but more complex DFA, construct an NFA accepting all strings over the alphabet {0,1,2,3,4,5,6,7,8,9} such that the final digit has appeared before.



In the above figure, only the paths for 0,1 and 9 are shown for conciseness. The same structure applies for 2,3,4,5,6,7,8.

Exercise: Construct a NFA which accepts all decimal strings where the last letter does not appear earlier.

Solution: As an NFA, this can be represented with just around 20 states (again, only 0,1 and 9 are shown for conciseness):



However, no DFA can model this language without at least 1024 states. We can show using PHP that this is the case. If not, if we consider the binary string whose i^{th} bit is 1 if i-1 has already been seen in our string and 0 otherwise, there are two different subsets which must end up at the same state. Take any element which is in their symmetric difference (in binary, the xor operator). Adding that element as the last letter of the decimal string should ideally cause one of the strings to go to an accepting state and the other to a non-accepting state, but these two strings are currently in an identical state and have received the same input, which is a contradiction.

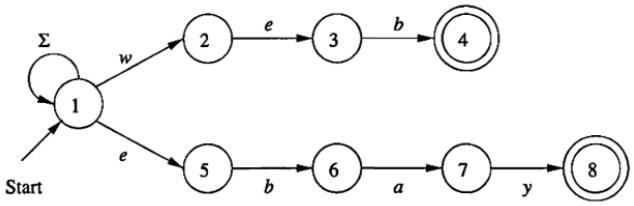
Exercise: Construct an NFA which accepts all strings which have a 1 at the 10th position from the right.

3.2 Text Searching

Suppose we are given a set of keywords, and we want to find occurences of any of these words. A NFA can achieve this.

1. There is a start state with a transition to itself on every printable ASCII character. Intuitively, the start state represents a guess that we have not yet begun to see any of the keywords, even if the guess is actually wrong.
2. For each keyword $a1 \cdots a_k$, there are k states, say $q1, q_2 \cdots q_k$. There is a transition from the start state to q_1 on a_1, and from q_i to q_{i+1} on a_{i+1}. The state q_k is an accepting state to indicate that $a1 \cdots a_k$ has been found.

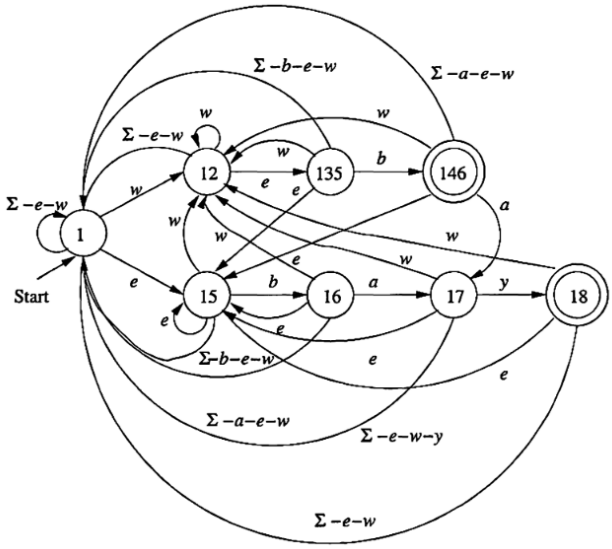
Example: Consider the following NFA which searches for the words web and ebay:



It turns out, converting these NFAs to DFAs actually does not cause blowup in the number of states. The rules for constructing the DFA is as follows:

1. If q_0 is the start state of the NFA, then $\{q_0\}$ is in the DFA.
2. Suppose p is one of the NFA states, and it is reached from the start state along a path $a_1 \cdots a_m$. Then one of the DFA states is the set of NFA states consisting of q_0, p and every other state of the NFA that is reachable from q_0 by following a path whose labels are a suffix of $a_1 \cdots a_m$.

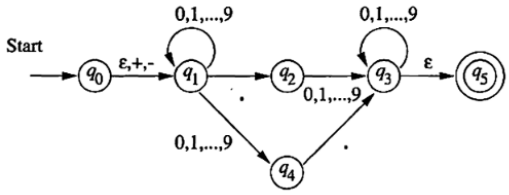
Note that, in general, there will be one DFA state for every NFA state p. However, in step 2, two states may actually lead to only one state in the DFA (if the paths from the start state to these two states have the same letters, i.e two keywords have a common prefix). Hence, there might even be fewer states in the DFA than in the NFA. The DFA corresponding to the above NFA is given below.



In the above picture, the transitions labeled $\Sigma - x$ for some symbol(s) x denote transitions labelled by all printable characters except those denoted by x. The numbers in the DFA circles denote the set of states from the NFA to which that DFA state corresponds. For instance, 135 corresponds to the NFA states 1, 3 and 5.

3.3 ϵ-Transitions

We introduce to NFAs the feature of allowing transitions on an empty string. This transition in fact does not expand the class of languages that can be accepted by finite automata, but it does give us some added conciseness and convenience, especially while dealing with optional symbols.
Example: The following is an ϵ-NFA which accepts decimal numbers consisting of an optional + or - sign, a string of digits, a decimal point and another string of digits. Either one of the strings of digits might be empty, but not both (for instance .12 is a valid decimal number, as is 12., but not +.).



We can represent an ϵ-NFA the same way we represent an NFA, with the exception that the transition function maps (q, ϵ) to a subset of Q (possibly empty) for all states q as well.

3.3.1 ϵ-Closures

We now give a formal defintion of the extended transition function on ϵ-NFAs. We define the Epsilon-closure of a state q (ECLOSE(q)) by considering only the arcs labeled by ϵ, and conducting a DFS or BFS from q to identify all the states in the automaton that are reachable from q using ϵ transitions alone.
To define the extended transition function, $\delta(q, \epsilon)$=ECLOSE(q). If w is xa, where a is the last symbol of w and a is not ϵ, then

$$\delta(q_0, w) = \bigcup_{p_i \in \delta(q_0, x)} ECLOSE(\delta(p_i, a))$$

To create the DFA which accepts the same language as an arbitrary ϵ-NFA, we use the same subset construction as we did with NFAs. If the ϵ-NFA was $(Q_E, \Sigma, \delta_E, q_0, F_E)$, the resulting DFA is $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$ where:

1. Q_D is the set of subsets of Q_E.

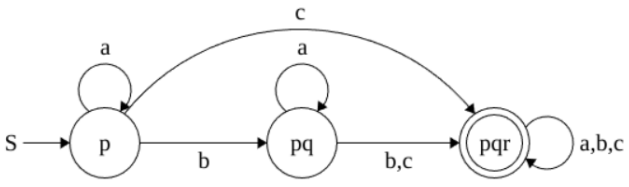
2. q_D is ECLOSE(q_0).
3. F_D is those sets of states which contain at least one accepting state of E.
4. δ_D(S,a) is computed for all a in Σ and all sets S in Q_D as:
   (a) Let S=$\{p_1, p_2 \cdots p_k\}$
   (b) Compute $\bigcup_{i=1}^{k} \delta_E(p_i, a)$. Let this set be $\{r_1, r_2 \cdots r_m\}$
   (c) $\delta_D(S, a) = \bigcup_{j=1}^{m} ECLOSE(r_j)$

Exercise:

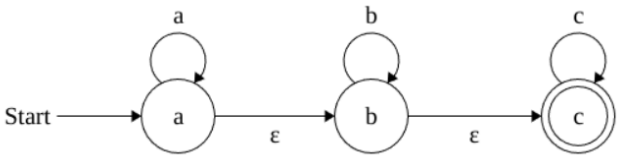
Consider the following ϵ-NFA.

	ϵ	a	b	c
$\rightarrow p$	$\emptyset$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\emptyset$
$*r$	$\{q\}$	$\{r\}$	$\emptyset$	$\{p\}$

Convert it to a DFA, and list all strings of length 3 or less accepted by the automaton.
Solution: First, we think about explaining this NFA using natural language. From p, an 'a' makes the new state p. A 'b' goes to q, from where it can immediately come back to p, and a 'c' takes it to r, from where it can immediately come back to p. Thus, any string can terminate at p. Similarly, we observe q is the termination point of all strings with at least one of b or c. r is the termination point of all strings with one or more c's or two or more b's.
Using the construction outlined above, the corresponding DFA is:



Exercise: Design an ϵ-NFA for the set of binary strings with at least one of the last 10 positions having a 1.
Exercise: Design an ϵ-NFA for the set of strings consisting of 'a' zero or more times, followed by 'b' zero or more times, followed by 'c' zero or more times.
Solution:



Each state represents a "guess" that we are done with all of the previous states (i.e state b represents a guess that we have seen all 'a"s and state c represents a guess that we have seen all a and b's).

4 Regular Expressions

Regular expressions are a way of depicting as a logical expression the language accepted by a finite automaton. For instance, search commands like grep take keywords in the form of regular expressions with a particular syntax, and then convert them to an appropriate DFA or NFA, and simulate that automaton on the file being searched.
We can define certain rules for constructing regular expressions

1. The constants ϵ and $\varnothing$ are regular expressions. L(ϵ)=$\{\epsilon\}$ and L($\varnothing$)=$\varnothing$.
2. If a is any symbol in the language Σ, **a** is a regular expression denoting {a}.
3. If E and F are regular expressions, so is E|F, denoting their union (the set of all strings in language E or F or both).
4. If E and F are regular expressions, so is EF (the set of all strings which are formed by concatenating a string from E and a string from F in that order).
5. If E is a regular expression, so is E^*, and it denotes $L(E)^*$. That is, $L(E^*) = L(E)^*$, where L(E) is the set of strings in the language of the expression E.
6. If E is a regular expression, it is the same regex as (E)
7. For convenience, we add the symbols En to denote the expression E repeated n times.
8. For convenience, we also use E? to denote one or zero appearances of the expression E. It is equivalent to (E|ϵ).

In terms of precedence, the star operator E^* has the highest precedence, followed by concatenation and then union.
Exercise: The command grep also allows for the regex symbol E,m to denote "E repeats up to m times" and En,m to denote "E repeats between n and m

times". Express these complex operators using the | and * operators and concatenation alone.
Exercise: Write a regular expression for the set of all binary strings not containing the substring 101.
Solution: 0*(1000*)*1*
Exercise: Write regex for the set of all binary strings with an equal number of 0's and 1's, with no prefix having two more 0's than 1's, or two more 1's than 0's.
Exercise: Write regex for the set of strings with a number of 0's divisible by 5 and the number of 1's being even.
Hint: Might take some casework.

4.1 Converting DFAs to Regular Expressions

Theorem 1. *If L=L(A) for some DFA A, then there is a regular expression R such that L=L(R).*

Proof: Let us suppose that A has states $\{1, \cdots n\}$. We use induction to prove that A can generate a regular expression. Let us use R_{ij}^k as the name of a regular expression whose language is the set of strings w such that w denotes a path from state i to state j in A, and the path never goes through any state with an index higher than k, except maybe at the endpoints i and j.
For the base case, k = 0. Since all states are numbered 1 or above, there must not be any intervening states between i and j. Hence R_{ij}^0 is:

1. $\epsilon + a_1 + a_2 \cdots a_m$ if i=j and $\delta(i, a_l) = i$ for all $1 \leqslant j \leqslant m$.
2. $\varnothing$ if there exists no symbol a such that $\delta(i, a) = j$.
3. $a_1 + a_2 \cdots a_m$ if $\delta(i, a_l) = j$ for all $1 \leqslant j \leqslant m$.

Now, for the induction part, suppose there is a path from i to k that goes through no state higher than k. If this path does not pass through k at all, the label of this path is in the language of R_{ij}^{k-1}. Else, the label of this path is in $R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1}$. Therefore,

$$R_{ij}^k = R_{ij}^{k-1} | R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1}$$

After finding all such R_{ij}^k, set i to the index of the start state, k=n, and let j iterate over all the accepting states. The complete regular expression accepted by the automaton is the union (—) of all the regular expressions for these values of j.
However, this is a very tedious task, and it is often more convenient to proceed by eliminating all states but one or two states.
Exercise: Let A be an ϵ-NFA with no transitions into the start state q_0 and a single accepting state q_f with no transitions out of it. Describe the language accepted by each of these modified automata, in terms of the language L(A) accepted by A:

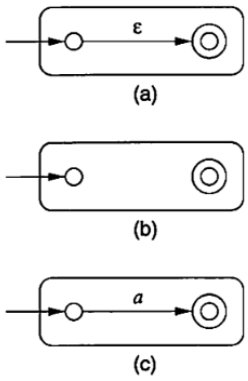
1. The automaton constructed by adding an ϵ-transition from q_f to q_0.
2. The automaton constructed by adding an ϵ-transition from q_0 to every state reachable from q_0.
3. The automaton constructed by adding an ϵ-transition to q_f from every state from which q_f is attainable.
4. The automaton constructed by doing both operations 2 and 3 above.

Exercise: In natural language or in pseudocode, give an algorithm that takes a DFA A and a number n, and computes the number of strings of length n accepted by A.
Hint: Use dynamic programming concepts, bringing down the time complexity to $O(m * n)$, if m is the number of edges (always less than the square of the number of states) in the DFA. Note that the brute force approach would be $O(k^n n)$ where k is the size of the language.

4.2 Converting regular expressions to automata

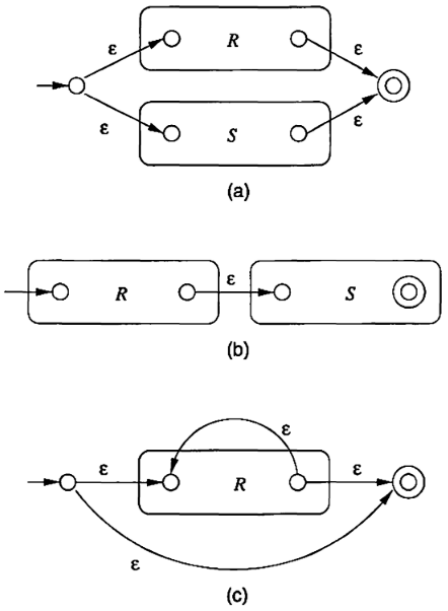
Theorem 2. *Every language defined by a regular expression is also defined by an automata.*

Proof: Suppose L=L(R) for some regex R. We show L=L(E) for some ϵ-NFA E with one accepting state, no arcs in to the initial state and no arcs out of the accepting state.
For the base case where R is ϵ, $\varnothing$ or a, we have the following three automata:



Now, assume two regular expressions R and S have equivalent automata[2]. For the induction step, we have to show that $R|S$, RS and R^* can be converted to ϵ-NFAs[3]. For that, we use the following three constructions:

[2]Depicted in the pictures as rectangles with R and S as labels
[3]Note that the {n}, {,m}, {n,m} and ? operations can be expressed in terms of the above three operators, so we do not need to showcase separate proofs for them in the inductive step.



4.3 Algebraic laws for regular expressions

Exercise: Prove the following for regular expressions R, S, T:

1. $R|S = S|R$ (commutativity of union)
2. $(R|S)|T = R|(S|T)$ (associativity of union)
3. $(RS)T = R(ST)$ (associativity of concatenation[4])
4. $R(S|T) = (RS)|(RT)$ (Distributivity of concatenation over union)
5. $(R^*)^* = R^*$ (Idempotent property of closure)
6. $(\epsilon + R)^* = R^*$
7. $(R^*S^*)_* = (R|S)^*$

Exercise: Prove or disprove each of the following:

1. $(R|S)^* = R^*|S^*$
2. $(RS|R)^*R = R(SR|R)^*$
3. $(RS|R)^*RS = (RR^*S)^*$
4. $(R|S)^*S = (R^*S)^*$
5. $S(RS|S)^*R = RR^*S(RR^*S)^*$

[4]Concatenation is not commutative

5 Properties of Regular Languages

Theorem 3. ***Pumping Lemma:***
Let L be a regular language. Then there exists a constant n (which depends on L) such that for every string w in L with $|w| \geqslant n$, we can break w into three strings w=xyz such that y is nonempty, $|xy| \leqslant n$ and for all $k \geqslant 0$, xy^kz is also in L.

Proof: This is equivalent to saying that if L can be modeled by a DFA with a finite number of states, and we set n higher than the number of states, by the pigeonhole principle there must be a loop in the path from the start state to the accepting state for any string of length greater than or equal to n. This gives us an obvious choice for x, y and z, and it is also easy to observe how looping over the loop multiple (maybe 0) times gives us xy^kz for all non-negative k.
Exercise: Prove that the following are not regular languages:

1. $\{0^n1^n{:}n \geqslant 1\}$
2. The set of regular bracket sequences.
3. $\{0^n10^n{:}n \geqslant 1\}$
4. $\{0^n1^M{:}n \leqslant m\}$
5. $\{0^n1^{2n}{:}n \geqslant 1\}$
6. $\{0^n$:n is a perfect square$\}$
7. $\{0^n$:n is a power of 2$\}$
8. The set of binary strings whose length is a perfect square.
9. The set of binary strings which are of the form ww, that is, some string repeated.
10. The set of binary strings of the form ww^R, that is, some string followed by itself in reverse.
11. The set of binary strings of the form $w\tilde{w}$, where $\tilde{w}$ is obtained by replacing all 1s with 0 and 0s with 1 in w.
12. The set of strings of the form $w1^n$, where w is a binary string of length n.

Solution :

1. FTSOC, assume L is regular. Let N be the value of n required by the pumping lemma. Let $w = 0^N1^N$. By the pumping lemma, w can be divided into x, y and z with y being non-empty, and $|xy| \leqslant N$ (hence y has to have a non-zero number of 0s and no 1s) with xy^kz accepted by L. But xy^kz does not have an equal number of 0s and 1s if $k \neq 1$, hence L does not satisfy the pumping lemma.
2. Similar to the last case, for a particular N, let $w = (^N)^N$. L does not satisfy the pumping lemma.

3. Similar to the above questions. Solution left to the reader.
4. For a given N, let $w = 0^N 1^{N+1}$. For appropriately large k, $xy^k z$ cannot satisfy the number of 0s being less than or equal to the number of 1s.
5. Similar to the above questions. Solution left to the reader.
6. For a particular N, let n be some number which is a perfect square greater than N and $w = 0^n$. Then, if y is of length j, n-j+aj is a perfect square for all non-negative a. However, successive perfect squares get arbitrarily distant from each other, hence there can only be a finite sequence of squares with the difference of two consecutive elements being bounded by j.
7. Similar to the above questions. Solution left to the reader.
8. Similar to the above questions. Solution left to the reader.
9. For a particular N, let $w = 0^N 10^N 1$. It is easy to observe that for any choice of $k > 1$, $xy^k z$ will have both its 1s to the right of its middle.
10. For a particular N, consider $w = 0^N 110^N$.
11. Similar to the above questions. Solution left to the reader.
12. Similar to the above questions. Solution left to the reader.

5.1 Closure Properties of Regular Languages

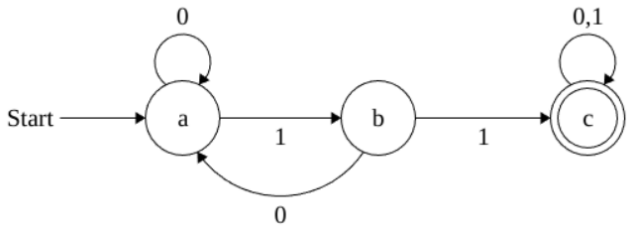
1. The Union of two regular languages is regular.
2. The intersection of two regular languages is regular.
3. The complement of a regular language is regular.
4. The difference of two regular languages is regular.
5. The reversal of a regular language is regular.
6. The closure of a regular language is regular.
7. The concatenation of regular languages is regular.
8. A homomorphism (substitution of strings for symbols) of a regular language is regular.
9. The inverse homomorphism of a regular language is regular.

Proof: The proofs for union, closure and concatenation follow from the fact that these operators are defined in the structural induction used to build up regular expressions themselves.
We will first prove the closure of regular languages over complementation. That is, for a suitable Σ, if L is a set of accepted words, the set of all words over Σ not in L is also regular. For proof, construct a DFA corresponding to L. Every word in Σ^* either ends up in an accepting state or not. If we make all non-accepting states accepting and accepting states non-accepting, then the new DFA accepts

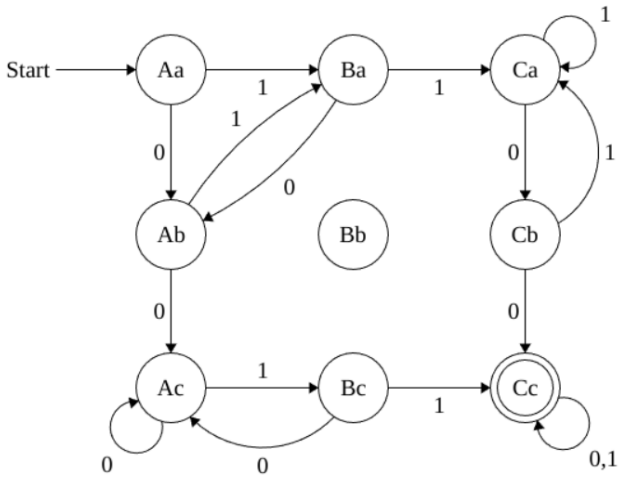
all words over Σ^* which are not in L. Intersection of two regular languages is simply the negation of the union of their negations, hence it is also regular. Similarly, we observe the difference is regular.
However, it is instructive to notice that from two NFAs, the intersection can be found with much fewer states than the worst case predicted by taking the negation of the union of their negations (which is $O(2^{2^n})$).

Theorem 4. *If L and M are regular languages, then so is $L \cap M$.*

Proof: Let L and M be the languages of NFAs $A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$ and $A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$. Notice that the alphabets of both can quite easily be assumed to be the union of their alphabets, WLOG. Let $N = (Q_L \times Q_M, \Sigma, \delta', (q_L, q_M), F_L \times F_M)$. That is, the states of the new NFA are pairs of states, one from A_L and one from A_M. The new start state has the start state of the two NFAs, and the new accepting states are those in which both states were accepting states in the NFA they are from. The new transition function satisfies $\delta((p, q), a) = \delta_L(p, a) \times \delta_M(q, a)$.
Exercise: Let L be the language accepting all binary strings containing a 11 and M be the language accepting all binary strings containing a 00. Construct a finite automaton modelling this.
Solution: Since the DFAs for L and M are very simple (3 states each), and the product construction is obviously much simpler with DFAs (and, usefully, also returns a DFA), we start with the DFAs accepting L and M. Given below is the DFA for L, the DFA for M can be obtained by replacing 0 with 1 and 1 with 0:



From the subset construction, we get the following DFA, with Aa as the start:



Intuitively, Bb is never reached by any run of the DFA, since B is only attained if the last string was a 1 and b is only attained if the last string was a 0, hence we can delete state Bb.

5.1.1 Closure under Reversal:

The reversal of a string is the string written backwards. The reversal of a language is the language consisting of the reversal of all its strings. Given a ϵ-NFA A that accepts a language,

1. Reverse all the arcs in the transition diagram of A.
2. Make the start state of A be the only accepting state for the new automaton.
3. Make one of the accepting states of A the new start state, with ϵ-transitions to all the other accepting states of A. If there are no accepting states in A, create a new start state with no transitions out of it, and which is not accepting.

More formally, we can prove that reversals of regular are regular languages by induction, using the fact that $(E_1|E_2)^R = E_1^R|E_2^R$, $(E_1E_2)^R = E_2^RE_1^R$ and $(E^*)^R = (E^r)^*$.

5.1.2 Closure under Homomorphisms

A string homomorphism is a function on strings that works by substituting a particular string (maybe empty) for each symbol. The homomorphism of a language is the set obtained by applying the homomorphism on every string in the

set L.
Proof: The proof involves proving that $L(h(E)) = h(L(E))$ for any regular expression E.
For the base case, if E is ϵ or $\varnothing$, h(E) is the same as E. Thus, L(h(E))=L(E). L(E)=h(L(E)) since L(E) is ϵ or $\varnothing$ respectively. If E = a for some symbol a in Σ, L(E)={a} and h(L(E))={h(a)}. Also, h(E) is the regular expression that is the string h(a). Thus, L(h(E)) is also {h(a)}, and we conclude $L(h(E)) = h(L(E))$.
For structural induction, we need to showcase that $L(h(E)) = h(L(E))$ holds if E is the union, concatenation or closure of expressions to which the proposition applies. For the union, we know $L(F|G) = L(F) \cup L(G)$ and $L(h(E)) = L(h(F)|h(G)) = L(h(F)) \cup L(h(G))$. Also, $h(L(F|G)) = h(L(F) \cup L(G)) = h(L(F)) \cup h(L(G))$. Now, using the inductive hypothesis, $L(h(F)) = h(L(F))$ and $L(h(G)) = h(L(G))$. Hence, $L(h(F|G)) = h(L(F|G))$.
The proof when E is a concatenation is similar.

5.1.3 Closure under Inverse Homomorphism

Suppose h is a homomorphism from some alphabet Σ to the (possibly same) alphabet T. Let L be a language over T. Then $h^{-1}(L)$ is the set of strings w in Σ^* such that h(w) is in L.
Exercise: Let L be the language (00|1)*. Let h be the homomorphism defined as h(a)=01 and h(b)=10. Find $h^{-1}(L)$.
Solution: $h^{-1}(L)$ is the set of all strings of the form $(ba)^*$. The forward direction of the proof is simple: if M is the language $(ba)^*$, $h(M) = (1001)^*$, which is a subset of L.
For the other direction, we need to prove that unless w is of the form $(ba)^*$, h(w) is not in L. Consider all the cases for w to not be of this form:

1. If w starts with a, h(w) starts with 01. This has an isolated zero, and hence is not in L
2. If w ends with b, h(w) ends with 10. This also has an isolated zero, and hence is not in L.
3. If w has aa or bb as a substring, h(w) has 0101 or 1010 respectively. Thus, h(w) would have an isolated zero, and hence would not be in L.

Notice that the above might seem counterintuitive. A lot of strings in L, for instance, 001, have no inverse under the homomorphism h. But we have proven that no other string in $(a|b)^*$ has its homomorphism in L, hence $h^{-1}(L) = (ba)^*$.

Theorem 5. *If h is a homomorphism from Σ to T, and L is a regular language over T, then $h^{-1}(L)$ is a regular language over Σ.*

Proof: Intuitively, the proof involves first constructing a DFA A for L and a method to translate the input string w to h(w) before sending it to the DFA.
Formally, let $A = (Q, T, \delta, q_0, F)$. Define a DFA $B = (Q, \Sigma, \delta', q_0, F)$ where the

transition function δ' is defined as

$$\delta'(q, a) = \tilde{\delta}(q, h(a))$$

where $\tilde{\delta}$ represents the extended transition function of A. This is the required DFA accepting $h^{-1}(L)$.

Exercise: Prove that if L is a regular language with a DFA A, the subset of L consisting of strings w such that each state of A is visited at least once while accepting w is a regular language.

Proof: Let the new language so created be L'. First, define a new alphabet T consisting of symbols that are of the form [paq], where:

1. p and q are states in Q
2. a is a symbol in Σ
3. $\delta(p, a) = q$

Now, define the homomorphism h([paq])=a for all [paq]. The language $M = h^{-1}(L)$ is regular by the construction we have just shown in the current section. Now, our goal is to repeatedly use the fact that the intersection of regular operations is regular to eliminate "bad" strings from M. First, we impose the condition that the first symbol in the string must start with q_0. That is, we take the regular expression $([q_0 a_i p_i]| \cdots |[q_0 a_n p_n])T^*$[5] for all symbols a_i and intersect it with M.

Next, we add the condition stating that the last symbol in $h^{-1}(w)$ must depict a transition to an accepting state of A. If there are k accepting states of A, create regular expressions similar to the one above for each of them, and take their union to generate a regular expression imposing the condition that the string terminates at an accepting state. Furthermore, impose the condition that the last state of the current symbol and the first state of the next symbol cannot be different (note that this is the negation of a finite regular expression - that finite regex can be obtained by consider all $2 * \binom{n}{2}$ ordered pairs of unequal n and all transitions reaching the first element of the pair, and all transitions leaving the second element of the pair, and stating that these two transitions occur simultaneously somewhere in the string). Now, intersect the regular language until now with the intersection of the n conditions stipulating that there is some state of the form $[qap_i]$ for all i (note that each condition can be stipulated as the union of every transition into p_i). Finally, apply the homomorphism h again on M to get L'.

Intuitively, through this process, we have stipulated that the string representing the transition history of w must satisfy the following conditions:

1. It must show that w began at a start state
2. It must show that w ends at an end state

[5]Here, I am using T^* as shorthand for the regular expression consisting of the closure of the union of all the symbols in T.

3. No two consecutive symbols can say, w ended at p after i symbols were read and moved out of q in the $(i+1)^{th}$ move, where $p \neq q$
4. No state can exist which was never visited

These are the necessary and sufficient conditions for a string to be in L', and we have constructed L' through a series of moves preserving the regular expression property, hence we are done.

Exercise: Suppose h is the homomorphism from {0,1,2} to {a,b} defined by h(0)=a, h(1)=ab, h(2)=ba. Give h(w), h(L) or $h^{-1}(M)$ as appropriate for the following:

1. w=0120
2. w=21120
3. L=01*2
4. L=0|12
5. M=$a(ba)$*

Exercise: If L is a language, and a is a symbol, then L/a is the set of strings w such that wa is in L. Prove that if L is regular, so is L/a.

Solution: Assume WLOG that L is in the form of a DFA A. Make a new DFA with the same start states and transition function and whose accepting states are all those states from which there was a transition on a to an accepting state of A.

Exercise: If L is a language, and a is a symbol, then a/L is the set of strings w such that aw is in L. Prove that if L is regular, so is a/L. **Exercise:** Which of the following is true:

1. (L/a)a = L
2. a(a/L)= L
3. (La)/a=L
4. a/(aL)=L

Exercise: Solve the following

1. Show that $(R|S)/a = (R/a)|(S/a)$
2. Give the rule for (RS)/a
3. Give the rule for $(R^*)/a$
4. Find (0|1)*011/a for a= 0 and a=1
5. Characterize those languages for which L/0=$\varnothing$
6. Characterize those languages for which L/0=L

Exercise: Show that regular languages are closed under the following operations:

1. min(L)={w: w is in L but no proper prefix of w is in L}
2. max(L)={w: w is in L but for no x is wx in L}
3. init(L)={w: for some x, wx is in L}

Solution: Let h be the isomorphism that maps [qap] to a for state q, p and a symbol a with $\delta(q, a) = p$. We know the inverse of this isomorphism applied to L gives M, a regular language. We intersect M with the conditions that the string starts at a start state and consecutive transitions are consistent with the state it was in previously, and ends at an end state.

1. Take the difference of M and the union of $T^*[qa_ip_i|\cdots|qa_kp_k]T^*$ for all accepting states q. This basically amounts to saying that there is no transition out of an accepting state.
2. When we convert L to an automaton A, we have knowledge of all accepting states such that there is no path from that accepting state to another accepting state (or back to itself). Obviously, if there was such a path from accepting state q, any w ending at q could be expanded. Hence, intersect M with the condition that M ends at one of the particular end states which does not have a path out of it to another end state and we get the desired regular language.
3. Take the negation of the above solution.

Exercise: If w and b are strings of the same length, define alt(w,x) as being the string in which the symbols of w and x alternate, starting with w. If L and M are regular languages, define alt(L,M) as the set of strings of the form alt(w,x) where w is any string in L and x in M. Show that alt(L,M) is regular.
Solution: Let L and M have DFAs A and B respectively. Let us design a DFA whose set of states is the set of ordered pairs of states, one from A and one from B. That is, if A_i is a state of A and B_j is a state of B, (A_i, B_j) and (B_j, A_i) are part of the new set of states. The start state is (A_0, B_0), where A_0 is the start state of A and B_0 is the start state of B. The accepting states are those of the form (A_i, B_j) where A_i is accepting and B_j accepting. If you are at state (A_i, B_j) and get input a, go to state $(B_j, \delta_A(A_i, a))$. If you are at state (B_j, A_i) and get input a, go to state $(A_i, \delta_B(B_j, a))$. Intuitively, one of A or B is active and is reading the input. Once it reads the input, we toggle the "active" status of A and B. The string is accepted only if it has even length, and its symbols in odd indices[6] are accepted by A, and all symbols in even indices are accepted by B.
Exercise: Let L be a language. Define half(L) to be the set of first halves of strings in L. That is, {w: for some x with the same size as w, wx is in L}. Notice that odd-length strings do not contribute to half(L). Prove that if L is regular, so is half(L).
Solution: Let A be the DFA accepting L. Let S_n be the set of all states from which we can reach an accepting state with a string of length n. S_0 is simply the

[6]Forgive me, programmers, for using 1-based indexing

set of accepting states. Then, a string of length n, w is accepted iff $\tilde{\delta}(q_0, w) \in S_n$. We do this by constructing a DFA with $m * (2^m)$ states, if m is the number of states of A. Each state corresponds to a (state, subset of states of A) pair. A state (q, S) is accepting iff $q \in S$. From every state (q,S), on input of a, we move to the new state $(\delta(q, a), S')$ where S' is the set of neighbours of S. Note that if S is S_n, then S' is S_{n+1}. Let the start state be (q_0, S_0). This DFA accepts half(L), as can easily be proven.

Exercise: Suppose that L is any language, not necessarily regular, whose alphabet consists of only 0's. Prove that L^* is regular. At first, this may look either trivial or preposterous. However, consider the language L=$\{0^i$: i is prime$\}$, which is not regular[7]. We can show that $L^* = 000^*$. Similarly, the closures of all such languages can be proven to be regular.

Solution: Let i be the GCD of all the j's such that $0^j \in L$. Let M be the regular expression consisting of $(0i)^*$. We can claim that all but a finite number of multiples of the GCD are expressible as the sum of elements of the set $\{$j: $0^j \in L\}$. This is because of Bezout's identity.

First, we prove that the GCD of an infinite set of numbers is the GCD of some finite subset of that infinite set. Notice that we only need to consider division by p for those primes p which are smaller than or equal to the least element of the set x (this exists by the Well ordering principle). This means there are finitely many primes to consider. Assume WLOG that the GCD of the set is 1 (if the GCD were not 1, divide every element by the GCD, use Bezout's identity, and then multiply back the GCD). Hence, the infinite set has at least one element which is not divisible by prime p. Consider the statements "x_i is divisible by p" for all i. Divisibility over natural numbers is a statement that can be expressed in First Order Logic. By the compactness of FOL, if the theory consisting of all these statements is inconsistent, some finite subset is inconsistent. Hence, for every prime p, we can find a finite subset of which it is not the GCD. Take the union of these finite subsets for all primes p less than x. This will give us a finite subset whose GCD is 1 (or the GCD of the infinite set, if we have divided by that GCD before).

Given that finite subset, we apply Bezout's identity. Every natural number can be expressed as the sum of multiples of the elements of this subset. However, some of those weights might be negative. But these weights can be made to increase by a constant amount when the natural number in question is incremented by the LCM of this finite subset. Hence, there is some finite N such that for all $n \geqslant N$, the number can be expressed as a sum of non-negative multiples of the elements of this set. So, there is at most a finite number of whole numbers which are multiples of the GCD of the set and which cannot be part of the closure of L'.

However, $(L')^* \subset L^*$, since L' is a finite subset of L. Thus, if only finitely many j satisfy $0^j \notin (L')^*$, only finitely many j can satisfy $0^j \notin L^*$ **Exercise:** Show that regular languages are closed under the following operation: cycle(L)=$\{$w: we can write w=xy, such that yx is in L$\}$.

[7]Prove this using the pumping lemma

Solution: Let the alphabet Σ have k elements and let the DFA A of L have n states. For a particular state A_i, let $S(A_i, a)$ be the set of all states such that A_i is their successor on symbol a. Construct a ϵ-NFA as follows:
From an arbitrary start state, given an input symbol a, let there be transitions to n different "arms". Each arm has as its start state A_i. From each of the accepting states of A, there is an ϵ-transition to the start state of A. The new accepting states of this arm are the elements of $S(A_i, a)$.
In effect, this DFA is making a guess on reading the first symbol of w. If w was indeed a cyclic shift of some string accepted by A, then after reading a, that string would have been at A_i for some i. From that A_i, it would have been able to go to the end state. Furthermore, from the next symbol, we can assume we are at the start of y. That is, we are at the start state of A. If the next few symbols take the arm to a state from which a transition on a can get you to A_i, we have successfully identified that w is a cyclic shift.

5.2 Decision Properties of Regular Languages (Optional)

This deals with three main problems:

1. Is a particular language described by a regular expression empty?
2. Is a particular string w accepted by the regular expression?
3. Are two descriptions (either as automata or regular expressions) of languages actually describing the same language (equivalence)?

5.2.1 Testing emptiness of Regular languages

A regular language is empty if it is equivalent to $\varnothing$, and in no other situation. If we model a regular expression as an automaton, it is empty iff there is no path from the start state to any accepting state. The presence or absence of a path can be ascertained using DFS (note that for a DFA, that would take O(n+n*m) time if the number of symbols in the alphabet is m and the number of states in the DFA is n).
However, we can potentially solve this in a much easier way by looking at the regular expression. If R has no occurences of $\varnothing$, it is of course not empty. For the base case, L($\varnothing$) is empty, while L(ϵ) and L(a) for any symbol a is not.
If $R = R_1|R_2$, L(R) is empty iff L(R_1) and L(R_2) are both empty. If $R = R_1R_2$, L(R) is empty if either of L(R_1) or L(R_2) is empty. If $R = R_1^*$, L(R) is never empty, as it at least contains ϵ.

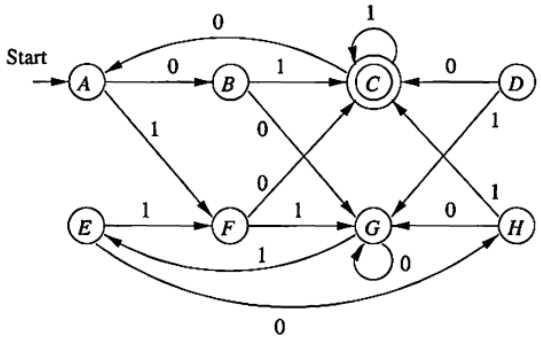
5.2.2 Testing membership in a Regular Language

If L is represented by a DFA, the algorithm is as simple as running the DFA on w and checking if it terminates at an accepting state.
If L is represented by an ϵ-NFA, we can run the following algorithm, which runs in O(ns^2) time, if the length of w is n and the number of states is s. Each input

symbol can be processed by taking the previous set of states, which numbers at most s, and looking at all their successors and taking the union of the same. We take the union of at most s of at most s states, which requires O(s^2) time (a factor of log(s) might be present based on the data structure implemented for finding the union).
If L is represented by a regular expression of size s, we can generate an ϵ-NFA with at most 2s states. We then perform the same action above, taking O(ns^2) time on an input of size n.
Exercise: Give an algorithm to tell whether a regular language L is infinite.
Solution: Consider the language as a DFA or NFA. Maintain a list of visited states and also the recursion stack while performing a DFS. If a state currently in the recursion stack is adjacent to the current state, we know there is a cycle. We can further find out the set of all states from which we can reach at least one accepting state (reverse the graph and perform DFS or BFS from each accepting state, for instance). If we notice that some vertex which is part of a cycle reachable from the start state is also part of a path to an accepting state, L is infinite. If not, all paths to accepting states are of bounded length (by the Pigeonhole Principle), and since the number of symbols is also finite, L is a finite set.
Exercise: Suppose L is a regular language with alphabet Σ. Give an algorithm to tell whether L=Σ^* (that is, every string over the alphabet is accepted).
Solution: From an NFA, we know a DFA can be constructed in O(2^n) time, if n is the number of states. For a DFA with m states, the complement can be found in O(m) time (or even O(1), if all you use is a boolean value). Iff the negation is empty, then L=Σ^*.

5.2.3 Equivalence of Automata

The question of whether two automata describe the same language involves constructing a DFA which has the minimum number of states possible whilst still being equivalent to these automata. If the two automata are equivalent, they will both reduce to the same DFA (up to a renaming of the states), which we prove by showing that the minimal equivalent automaton for a given automaton is unique.
Example: Consider the DFA below, with the transition function δ:



Certain pairs of states are obviously not equivalent. For instance, C and G are not equivalent because C is accepting and G is not. However, consider A and G. They are both non-accepting, and on the strings 0 and 1, they both go to states which are non-accepting. However, on the input 01, A goes to C and G goes to E. This means that the string 01 can distinguish between the two.

The above example gives an intuitive idea of how to find states that are equivalent: if there is some string on which one of the states leads to acceptance while the other leads to non-acceptance, they are non-equivalent. If not, the states are equivalent and we can reduce them to one state.

This table-filling algorithm proceeds as follows:

For the basecase, two states are non-equivalent if one of them is accepting and the other is not.

Let p and q be states that for some input symbol a, $\delta(p, a)$ and $\delta(q, a)$ are states known to be disntinguishable. Then p and q are a pair of distinguishable states, since there is some string which distinguishes $\delta(p, a)$ and $\delta(q, a)$. Let it be w. Then aw distinguishes p and q.

For instance, in the above image, this algorithm identifies that (A,E),(B,H),(D,F) are indistinguishable pairs of states. By induction on the length of a string which distinguishes two states, it is easy to observe that the table-filling algorithm actually identifies all pairs of equivalent states.

This gives us an easy way of testing equivalence of two regular languages. Construct a DFA consisting of the DFAs L and M "next" to each other, with no interactions between them. Since the start state does not affect the algorithm, any state can be taken to be the start state. Now, run the table-filling algorithm. If the start states of the two DFAs are equivalent, the initial two regular languages were equivalent.

Note that this can be accomplished in $O(n^2)$ time, by preprocessing the states to initialize, for each pair, a list of those pairs which would be rendered distinguishable if the first pair is found distinguishable. The total work of this algorithm is proportional to the sum of the lengths of these lists. Since each pair can be added to at most k lists, and k is a constant (for most purposes, the size of an alphabet is pretty small relative to the thousands or millions of states a DFA can have), the total work is $O(n^2)$.

Exercise: The above discussion should give an intuitive idea on how to construct a minimal equivalent DFA for any DFA. Even if you cannot prove the minimalism property, try to prove why the intuitive idea of creating equivalence classes actually satisfies the DFA properties (i.e on a symbol a, equivalence class p must transition to a single equivalence class, not multiple classes, and there must be a transition for each symbol). **Exercise:** If you can formalize how the minimum-state DFA equivalent to a DFA can be constructed, construct the minimal DFA equivalent to the DFA in the figure above.

6 Turing Machines (Optional)

While it may seem that ideal computers, with unlimited memory to utilize, should be able to solve any problem given to them, there is a huge category of functions that cannot be computed by even an ideal computer (uncomputable functions) and a huge category of decision problems that cannot be decided by a computer (undecidable problems). For instance, the question of whether an arbitrary program halts for a given input is undecidable, as is the question of whether a program's output starts with a particular letter (say, 'a').
However, to prove that a certain problem cannot be solved by a finite, deterministic algorithm in any programming language, we need a model of a machine which is capable of performing any task performed by any "algorithm". The yet-unproven Church hypothesis states that a function on the natural numbers can be calculated by an effective method if and only if it is computable by a Turing machine.
A Turing machine consists of a finite FSM called a control, and an infinite tape (memory) divided into squares or cells. Each cell can hold one symbol taken from a finite language. The input to the Turing machine, which is a finite string chosen from the input alphabet, is placed on the tape. All other tape cells, exending infinitely to the left and right, hold a special symbol called the blank to denote that they are empty. There is a tape head that is always positioned at one of the tape cells. The Turing machine is said to be scanning that cell. Initially, the tape head is at the leftmost cell that holds the input.
In one move, the Turing machine can:

1. Change the state of its control. Of course, the new state might even be the same as its old state.

2. Write a tape symbol in the cell it is scanning (again, can be the same as it was before).

3. Move the tape head left or right. It is unnecessary to consider a tape head which remains stationary in a particular move or which moves multiple spaces - in the first case, the tape head eventually has to move out of the cell, at which time the FSM would have moved to a state q'. We can just create a new Turing Machine which, in instances where the old machine would have remained stationary, changes state to q' and changes the tape

symbol to the last symbol the stationary head wrote, and moves the head in the direction it finally moves. In the second case, we can similarly divide the multiple moves into single moves.

Formally, a Turing Machine is represented by the 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

Where Q is the finite set of states in the control, Σ is the finite set of input symbols, Γ is the finite set of tape symbols[8], δ is the transition function. The arguments of $\delta(q, X)$ are a state q and a tape symbol X which is being scanned. The value of $\delta(q, X)$ is a triple (p, Y, D), where p is the new state, Y is the symbol written on the tape and D is a direction (left or right) that the head moves. q_0, as usual, is the start state, B is a blank symbol, which is in Γ but not in Σ, so it cannot be confused with a symbol in the input. F is the set of accepting states in Q.

6.1 Instantaneous Depictions

In order to compactly describe what a TM does for a particular input, we need to develop an instantaneous depiction, or ID, which tells us the exact state of the TM after it has read a particular input. While the tape a TM uses is theoretically infinite, it is possible to have finite IDs since in any finite number of moves (due to a finite input string), the TM can only have visited a finite number of cells. Thus, in every ID, we can show only the symbols between the leftmost and rightmost non-blank cells.
By convention, to include the state of the control in the ID, we write the state to the left of the cell being scanned in the ID. Thus, if we are at state q and we are scanning the second symbol when 10101 is written on the tape, we can represent it as 1q0101. If the cell being scanned is to the left of the first non-blank, we show the blank characters between the cell which the head is scanning (inclusive) and the leftmost non-blank. We proceed analogously if the head is to the right of the rightmost non-blank.
To represent a move of a TM, we use $\vdash$. To represent a non-negative sequence of moves that takes the TM from one state to another, we use $\vdash^*$. For instance, suppose $\delta(q, X_i) = (p, Y, L)$. Then,

$$X_1 X_2 \cdots q X_i X_{i+1} \cdots X_n \vdash X_1 \cdots p X_{i-1} Y X_{i+1} \cdots X_n$$

is the ID representation of the corresponding move.
Example: We construct a Turing machine which accepts the language $\{0^n 1^n | n \geqslant 1\}$. That is, it accepts the language with a number of 0s followed by an equal number of 1s.
Construction: One way to accept this language is to delete the first 0 and the last 1, pair by pair. If at some point after deleting a zero there is no 1 or there is no zero but a 1, the language is not accepted. If there is a 0 after a 1 or the

[8] Σ is a subset of Γ

first letter is 1, the language is not accepted. We can construct a model for this as follows:

1. q_0: This is the state we are in at the start, and it is at the leftmost non-blank. Finding a 1 there makes it go to a dead state q_5. Finding a 0 there makes it go to q_1, after converting the 0 to a blank. If there is neither 0 nor 1, that means we have deleted all zeros and ones, and there were an equal number initially, hence we can go to an accepting state q_4.
2. q_1 This state corresponds to going right in search of the first blank. Scanning 0 and 1 make it return to the same state, but proceed right. Scanning a blank makes it transition to q_2 and go left.
3. q_2 This is the state corresponding to the rightmost non-blank. Finding a zero makes it go to the dead state q_5. Finding a 1 makes it go to q_3 after converting the 1 to B.
4. q_3 The leftward analogue of q_1. Finding a blank makes it go right and become q_0.
5. q_4 The accepting state, with a halt command (i.e no transitions out of it).
6. q_5 The non-accepting dead state, with a halt command.

Example: Turing machines can also be used to compute integer-valued functions. Non-negative Integers can be represented on the strip as blocks of a single character of length equal to the magnitude of the integer. Construct a Turing machine that replaces m and n with m $\dot{-}$ n, where here $\dot{-}$ is defined as m-n if m $\geqslant$ n and 0 otherwise. We can interpret the input as a string on the tape consisting of 0^m10^n surrounded by blanks. At the conclusion, the tape should have $0^{m\dot{-}n}$ surrounded by blanks.
We can model this as follows: M repeatedly finds the leftmost 0 after a B and replaces it with B. It then proceeds right until it finds a 0 after 1. If there is no such 0 (i.e there is a B after 1), then return to the left and replace the blank right before the leftmost non-blank with a 0. If there is a zero, replace it with a 1 and go back to the first non-zero character earlier. If there is no such character (i.e the leftmost non-blank is a 1), then m $\leqslant$ n and we replace all 1s with blanks and halt.

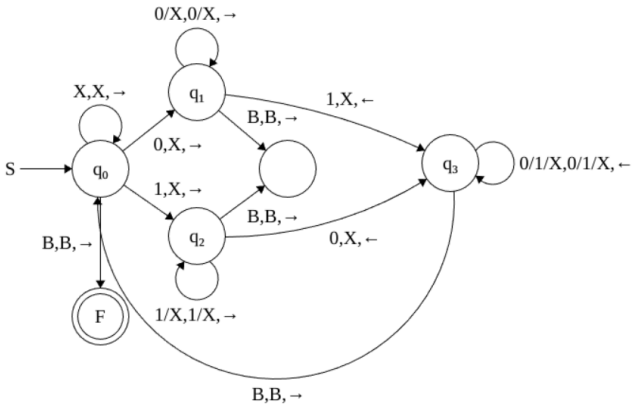
State	**0**	**1**	**B**
$\rightarrow q_0$	(q_1, B, R)	(q_0, B, R)	-
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	-
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	-	-	-

Exercise: Design Turing machines for the following languages:

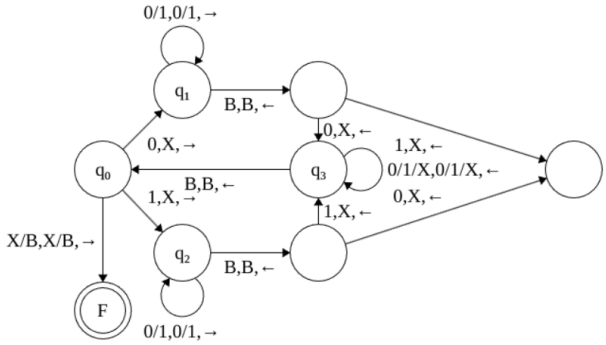
1. The set of binary strings with an equal number of 0s and 1s
2. $\{a^n b^n c^n | n \geqslant 1\}$
3. $\{ww^R |\ w \text{ is a binary string}\}$

Solution:

1. For the first TM, we will use the following states, with their role explained intuitively:
   (a) q_0: This state starts from the leftmost non-blank character and goes right until it finds a 0 or 1. If it does not find any before it encounters a blank, move to an accepting state. If it first encounters a 0, convert that to X and continue moving right in state q_1. Else, convert the 1 to X and go to stat q_2 and move right.
   (b) q_1 Keeps moving right in search of a 1. If it does not find any before a blank, go to a dead state. If it does, convert the 1 to X and go to state q_3.
   (c) q_2 Analogous to q_1, but searches for 0 instead of 1.
   (d) q_3 Keeps going left until it finds a blank. At that point, it moves right and goes to state q_0.

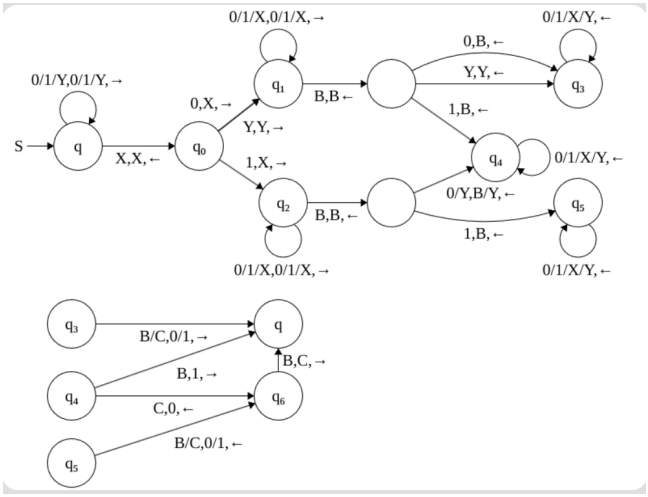


2. This is a bit more complex version of the TM that accepts strings of the form $0^n 1^n$. Replace the first instance of a, b, c respectively with X on each round.
3. The TM below accepts these strings:



Exercise: Two numbers n and m in binary are written on the tape as input, separated by some symbol which is not 0 and 1 (let this be X). Design a Turing machine that writes n+m in binary on the tape.
Solution: We write n+m in binary in the space to the left of n, from the rightmost bit, whilst erasing n and m. Note that we can also assume that to the left of n is a symbol Y, because it is trivial to add a new start state which just adds a Y in the slot before the leftmost bit in n before moving to the original start state.

1. q: This state keeps going to the right until it finds X. It then goes to q_0.
2. q_0 This state starts off from the last bit in n not set to X. If there is no such bit or if that bit is 0, it goes to q_1, else it goes to q_2.
3. q_1 and q_2 These states move to the right until they encounter B. At this point, they read the last bit of m. If it is 0 or X, q_1 goes to q_3, and q_2 goes to q_4. Else q_1 goes to q_4 and q_2 goes to q_5.
4. q_3, q_4, q_5: These states go left until they encounter B or C. q_3 replaces B with 0, C with 1 and goes to q. q_4 replaces B with 1 and goes to q. It replaces C with 0 and the B to the left of C with C. q_5 sets B to 0 and C to 1, and sets the leftward B to C.



The above picture has been split into two parts for the sake of space. q_6 is the state that sets the bit to the left of C to be C whenever needed (i.e the state that implements the carryover operation).

6.2 Modifications to TMs

6.2.1 Storage and Multiple Tracks

Without adding any functionality to TMs, we can view them as being able to store information about some finite number of variables whose values are from a predetermined finite set, since this is just equivalent to having new state variables for each combination of state, variable value. For instance, we can design a TM which remembers the first symbol (0 or 1) that is sees, and checks to ensure that it doesn't appear again. This accepts the language (01*)|(10*). This can be modeled for example as:

1. $\delta([q_0, B], a) = ([q_1, a], a, R)$ for a=0 or a=1. $[q_0, B]$ is the start state, and then it proceeds into one of two tracks depending on what it read first.
2. $\delta([q_1, \tilde{a}], a) = ([q_1, \tilde{a}], a, R)$ where $\tilde{a}$ is the complement of a. In state q_1, the TM skips over all characters which are the complement of the first character it read, as it should do.
3. $\delta([q_1, a], B) = ([q_1, B], B, R)$ for a =0 or a=1. $[q_1, B]$ is an accepting state of the TM.

Another useful trick which does not change the capabilities of a TM is to consider the tape as comprised of several tracks, each holding one symbol. This means we can view the tape alphabet as tuples, with one component for each track. This is useful when we view it as one track holding a variable and the other track holding a marker, which indicates something important about that position.

Exercise: Given a string x, check if x is of the form wcw for some string w and a character c not in w. You can imagine x as being on track one, and track 2 of a tape being empty. Use a marker in track 2 to indicate the symbols in w that have already been verified, (or equivalently to indicate the last symbol in w that has been verified).

6.2.2 Subroutines

As with programs, it further helps to think of TMs as having "functions" or subroutines which it can call. This set of states includes a start state and another state that serves as a "return state" to pass control to when the subroutine is over. Should our program require calling the subroutine from several states, we can make copies of the subroutine, using a new set of states for each copy. The "calls" are made to the start states of different copies of the subroutine, and each copy "returns" to a different state.
Exercise: Construct a subroutine "Copy" that can be used to implement a multiplication function. That is, the TM should start with 0^m10^n and end with 0^{mn} on the tape. It suffices to explain the states of both the subroutine and the program in natural languages, although constructing a transition diagram would be helpful for verification of the TM's correctness.

6.2.3 Multitape Turing Machines

A multitape Turing machine has a finite number of tapes instead of just one. Each tape is divided into cells, and each cell can hold a symbol of the tape alphabet.

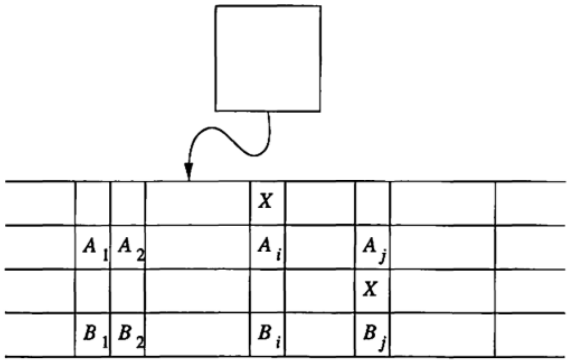
1. The input, a finite sequence of input symbols, is placed on the first tape.
2. All other tapes are filled with the blank alone.
3. The head of the first tape is at the left end of the input.
4. Since all other tapes are filled with the blank and infinite, the position of the head on these tapes does not matter. For convenience, however, we can assume that these tapes are all like the number line, with the head at position 0, and a move right representing a move to the successor of the current number, and a move left representing a move to the predecessor.

On each move, the heads on each tape can move independent of each other. Each head can move left, right or stay stationary.
Despite all these additional powers, each multitape TM is equivalent to a single-tape TM.

Theorem 6. *Every language accepted by a multitape TM is accepted by a single-tape TM. These languages are called the Recursively Enumerable (RE) languages.*

Proof: Suppose a language L is accepted by a k-tape TM M. We simulate M with a one-tape TM N whose tape we can think of as having 2k tracks. Half

these tracks hold the values in the tapes of M, and the other half of these tracks are blank everywhere but in one cell. Let the i^{th} cell on the j^{th} track have this special symbol. This corresponds to the TM M having its j^{th} head on the i^{th} cell of the tape corresponding to the $(j+1)^{th}$ track N has. To simulate a move of M, N's head must visit all k markers. We take as an invariant that the head is initially at the leftmost of the k markers (this is true for the base case, where by convention we can assume that all the k markers are on the same numbered cell in N, since all but the first track is empty). N can then go all the way until the last of the k markers is found (to know when the last marker is found, the control must have a finite storage storing the number of markers it has found). After each marker is found, since N knows what M would have read at this point, and N can also store which state M was in before this, N knows where M will go next. Thus, N knows exactly how all the tracks of M will look after M has made its move. On its way back to the leftmost marker (note that the leftmost marker might change, but N knows which position the new leftmost marker will be in), N changes all the columns accordingly. Thus, whatever M accepts, so can N.



Note that this new N also does not have exponentially more running time than M. In fact, for every n moves of M, N concludes in $O(n^2)$ moves. This can be proven by noticing that after n moves, the k heads of M can be at most 2n apart, and thus, on each move of N, the head has to move O(n) times to go from the first to the last and back whilst making the required changes to the tape.

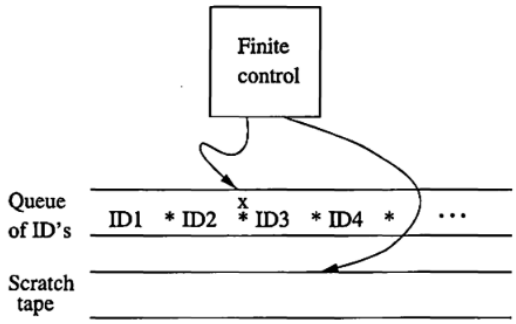
6.3 Non-Deterministic Turing Machines

A NTM differs from a deterministic TM (DTM) in that its transition function maps to a set of possible transitions instead of just one. As with other non-deterministic machines, an input is accepted if there is some sequence of transitions which accepts it, amongst all the possible sequences.

We prove that the NTMs accept no languages not accepted by a DTM. The proof involves showing that for every NTM M_N, we can construct a DTM M_D that explores the ID's that M_N can reach by any sequence of its choices. If M_D finds a path ending at an accepting state, then M_D enters an accepting state of its own. This can be viewed as an algorithm, putting new IDs on a queue so that after some finite time M_D has simulated all sequences of up to k moves of M_N.

Theorem 7. *If M_N is a non-deterministic TM, then there is a deterministic TM M_D with $L(M_N) = L(M_D)$.*

Proof: M_D can be designed as a multitape TM. The first tape of M_D holds a sequence of ID's of M_N. One ID of M_N is marked as the current ID. All IDs to the left of it have been explored and can be ignored subsequently.



To process the current ID, M_D does the following:

1. M_D examines the state and scanned symbol of the current ID. Built into the finite control of M_D is the knowledge of what choices of move M_N has for each state and symbol. If the state in the current ID is accepting, M_D accepts and simulates M_N no further.
2. However, if the state is not accepting, and the state-symbol combination has k moves, then M_D uses its second tape to copy the ID and then make k copies of that ID aT the end of the sequence of IDs on tape 1.
3. M_D modifies each of those k IDs to represent the IDs of each of the k choices of move that M_N has from its current ID.
4. M_D returns to the marked, current ID, erases the mark and moves the mark to the next ID to the right. The cycle then repeats with step 1.

Suppsose m is the maximum number of choises M_N has in any configuration. Then there is one initial ID of M_N and at most m ID's that M_N can reach after 1 move, m^2 ID's that M_N can reach after 2 moves, and so on. Since M_D essentially performs a breadth first search, if there is an accepting state, it; will find it in less than or equal to nm^n moves.

It is so far unknown if we can bring this bound to a polynomial bound in n rather than exponential. **Exercise:** Informally, but clearly, explain how you would construct the following NTMs:

1. A NTM which accepts all binary strings which have a sequence of length 100 repeated. i.e strings which are of the form xwywz where w is of length 100.
2. A NTM which takes as input a sequence of integers in binary, separated by a symbol X, and accepts the sequence if there is some number which is equal to its index (1-indexed) in the sequence.
3. A NTM which takes as input a sequence of integers in binary, separated by a symbol X, and accepts the sequence if there are two numbers which are equal to their indices (1-indexed) in the sequence.

Solution:

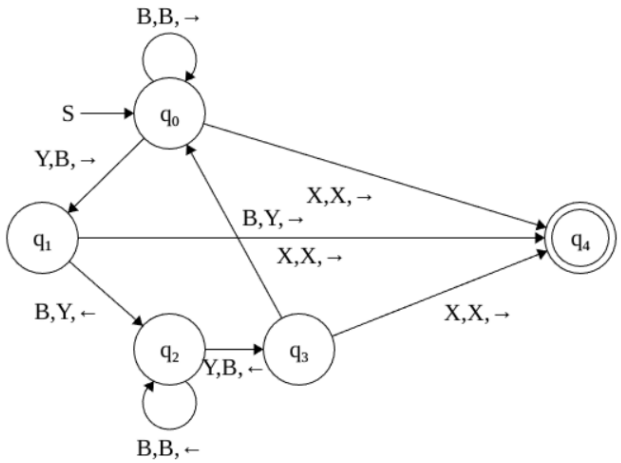
1. Assume the tape has two tracks. The start state has the options of remaining as the start state, not modifying variables and moving right, or of marking the position it is at and entering the first in a sequence of 99 states. Each of these 99 states shifts the head right, moves on to the immediate next state (i.e state 1 moves to state 2 and so on) on either 0 and 1, and moves to a dead state on encountering B. At the end of these 99 states, we mark the position we are in and enter a state which can continue to move right without modifying anything, or mark the current variable. With this, we currently have three markers set up, which represent the start of the substring of length 100, the end of that substring and our "guess" as to where the second occurence of the substring begins. It is now trivial to construct a subroutine which checks if the sequence between the first and second marks is equal to the sequence beginning at the third mark.
2. Again, we can consider this as having two tracks. The first track has the input. In the start state, we fill the second track with a 1 and move to a new state q. q is such that it makes a guess as to whether the binary number in the first track until the next X is the same as the binary number in the second track or not. If it guesses that the numbers are different, we proceed to call a subroutine that increments the number in the second track, and on return, we keep moving until we next encounter X. Then, we move right and again make a guess. If we encounter B before the next X, we are done with the sequence and that branch terminates in a dead state. Alternatively, if at some point we guess that the numbers are the same in binary, we proceed to check if the numbers in the first and second track are identical (they are both in binary). If they are, we move to an accepting state. If not, the guess was wrong and this path terminates.
3. One way is to modify the solution to the above problem by keeping three tracks instead of two. The second and third track are incremented by 1 (in

binary) whenever we make a guess that we have not arrived at a number which is the same as its index. The first time we guess, if the guess turns out to be right, instead of going to an accepting state, we return to a state which continues going right and makes another guess, this time using the number in track 3 (since the number in track 2 was erased while checking if it was the same as the number in track 1). If the second guess is right, we go to an accepting state.

Exercise: To truly understand the power of NTMs, consider an infinite blank tape with exactly one non-blank character X. The head is currently at some blank cell, and the state is q.

1. Write transitions that will enable the NTM ot enter state p after finding the symbol X.
2. Suppose the TM was deterministic instead. How would you enable it to find X?

Solution: For a non-deterministic TM, we simply need to have two possibilities to go from q - q_1, which keeps going right to search for X, and q_2, which keeps going left to search for X. If either of them finds X, we go to the accepting state p (technically, in order to make p scan X, we might need to move one step beyond X, to a new state, and then step back to the cell with X, entering the accepting state p in the process).
For a deterministic automaton, consider the following automaton:



Exercise: A k-head Turing machine has k heads reading cells of one tape. A move of this TM depends on the state and on the symbol scanned by each head. In one move, the TM can change state, write a new symbol on each cell being scanned and move each head right, left or keep it stationary. Since several heads may be scanning a cell, we assum ehte highest numbered head scanning a cell

gets priority while assigning new symbols to the cell. Prove that the languages accepted by the k-head Turing machines are the same as those accepted by ordinary TMs.

Exercise: We have seen an example of how to simulate a k-tape TM by a one-tape TM.

1. Suppose this technique if used to simulate a 5-tape TM with a tape alphabet of 7 symbols. How many tape symbols would the one-tape TM have?
2. An alternative way to simulate k tapes by one is to have a (k+1)st track to hold the head positions of all k tapes. Note that in the (k+1)st track, we must allow for two or more heads to be at the same cell and also be able to distinguish amongst the tape heads for different tapes. How many tape symbols does this need?
3. Another way to simulate k tapes by one is to avoid storing the head positions altogether. Rather, a (k+1)st track is used only to mark one cell of the tape. At all times, each simulated tape is positioned on its track so that its head is at the marked cell. If the k-tape TM moves the head of tape i, te simulation moves the entire non-blank content of the ith tape in the opposite direction. How many tape symbols does this need? What is the expected complexity of the operation of the onetape TM in terms of the number of operations of the k-tape TM?

Solution:

1. There are $7^5 \cdot 2^5$ tape symbols, since there need to be tape symbols for each combination of the 7 tape symbols in the 5 tracks, and also for each combination of X, B (assuming X is the chosen marker symbol) in the tracks which store the head positions.
2. This does not reduce the number of tape symbols. Proof left to the reader[9].
3. This does indeed reduce the number of tape symbols to just $7^5 \cdot 2$. The order of complexity is also $O(n^2)$.

6.4 Decidability of Problems

There are many problems for which no "algorithm" can ever be found. In fact, there are many Yes/No problems for which not even an ideal computer can guarantee that it will accurately answer "Yes" if the answer is yes. In close relation is the category of problems where no computer can compute an answer for any input in a finite number of steps. While some of these problems are very abstract (what is the maximum number of steps a Turing machine with n states can take?), some are more mathematical (given a set of n$\times$ n matrices, can the zero matrix be expressed as a finite product of these matrices?)

[9]Consider the subsets of the set of heads, and distinguishing power over the set of these subsets.

For instance, consider the problem of whether an arbitrary program prints "Successful" as its first 2 characters of output. Note than the difficulty stems from those programs which may take a very large amount of time. For instance, consider a brute force program that checks all tuples (x,y,z) (x, y and z are integers, in increasing order of z, and with x and y less than z) to check whether they satisfy $x^3 + y^3 = z^3$. If such a tuple was found, it would immediately print "Successful". If we had a program that could conclude whether or not this program prints "Successful", we could have just used that program to solve our question[10].

However, assume we had a hypothetical tester H for our proposition. That is, we assume H takes as input a program P and an input I, and tells whether P with input I prints "Successful" as its first word. In particular, H can output "Yes" or "No". Now, if instead of printing "No", we modify H to print "Successful" (since this can just be done by, for instance, replacing every instance of "No" as a string with "Successful" in the code of H, this should not affect the properties of H).

We will now create a new Turing Machine H_2. It takes as input not P and I, but only P. Instead of an input I, it feeds the code of P as input to P. That is, wherever P would have read from the input, P is made to read from its own code. If H_2 is fed itself as input, there are two possibilities: it prints "Yes" or "Successful". If the output of the H_2 is "Yes", that would mean the inner H_2 printed out "Successful" when passed H_2 as its argument. But that means the outer H_2 ought to have printed out "Successful" when passed H_2 as its argument. This reasoning can also be applied if the outer H_2 prints "Successful", leading us to conclude that there is no Turing Machine which can predict if an arbitrary code outputs a particular value of string as its first output.

Exercise: Now that we have proven one sort of decision problem is undecidable, rather than constructing H_2 as we did above, we can simply reduce later problems to examples of the decision problems we have earlier concluded are undecidable. For each of the following problems, find a TM such that if the problem were decidable, the TM could also solve the problem of checking whether the output of an arbitrary program is a particular string.

1. Given a program and an input, does the program eventually halt?
2. Given a program and an input, does the program ever produce any output?
3. Given two programs and an input, do the programs produce the same output for the given input?

[10]Unfortunately, no such algorithm existed, and thus mathematicians had to labor and ultimately proved Fermat's Last Theorem, a generalized version of our question.