Computation

This course studies computation as a task and attempts to understand the hardness of computation. As we will go on to see, some computational tasks are trivial, some are easy, some are hard and some are even impossible. The hardness of a task is of course a function of the computational model we choose to work in. Many a time, the model we choose will serve as being pretty insightful into the structure and hardness of the problem itself. To start off, we need to define computational problems. For this purpose, we introduce the notion of an alphabet $\Sigma$. $\Sigma$ is the set of symbols used to encode the computational problems and their solutions. For instance, in the world of computers, we would think of $\Sigma = \{0, 1\}$. In fact, going forward, unless stated otherwise, we will work over this binary alphabet. Transitioning to a larger alphabet, in most cases, will be trivial. The unary alphabet $\{1\}$ is of special interest and will pop up occasionally.

For a huge portion of the class, the kind of computational problems we will consider are what we call decision problems. Decision problems are those problems whose solution is a simple yes or no, 1 or 0, a single bit. It is possible for us to think of yes and no as the response to a membership question. We use this idea to define a language. A language $L$ is a subset (finite or infinite) of strings over the alphabet $\Sigma$. Put mathematically, $L \subseteq \Sigma^*$, where $\Sigma^*$ is the set of all strings over the alphabet $\Sigma$ (this notation will become clear as we move on). Now, given a computational problem, two languages associated to the language become pertinent – the set of all instances of the problem whose solution is yes, and the set of all instances of problem whose solution is no. This idea will be useful going ahead.

Finite Automata – Getting to Know You

In class, we looked at the first, and simplest, model of computation, namely the finite automaton. A finite automaton is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where $Q$ denotes the set of possible states of the automaton, $\delta : Q \times \Sigma \rightarrow Q$ denotes the transition function, $q_0 \in Q$ denotes the start state of the automaton and $F \subseteq Q$ denotes the set of accepting states of the automaton. Recall that, informally, the finite automaton accepts those strings that make that automaton reach an accepting state when the automaton reads them, and this set of strings is called the language of the finite automaton, that is said to be recognized by the automaton. Every other string is said to be rejected by the automaton. Languages recognized by finite automata are called regular languages. The goal of this section will be to better understand the finite automaton (or equivalently, regular languages) and how it works. Consider the finite automaton described diagrammatically below.
Mathematically, we would describe this as $M = (Q, \Sigma, \delta, q_0, F)$ where $Q = \{q_0, q_1\}$, $F = \{q_1\}$ and $\delta$ described by

$$
\delta(q_0, 0) = q_1; \delta(q_0, 1) = q_0 \\
\delta(q_1, 0) = q_0; \delta(q_1, 1) = q_1
$$

Going forward, we will use the mathematical and diagrammatic descriptions of finite automata interchangeably.

A little inspection will reveal that the language of the above automaton is the set of all binary strings with an odd number of 0s. Let us see why. Notice that both states are oblivious to 1s. Since they are the only states, the strings accepted or rejected by the automaton would have nothing to do with 1s. Furthermore, the two states toggle on reading a 0. So, the state of the automaton reflects the parity of the number of 0s read thus far. It is easy to see that $q_0$ maintains the information – “An even number of 0s have appeared thus far and this string shall be rejected”, while $q_1$ maintains the information – “An odd number of 0s have appeared thus far and this string shall be accepted”. This leads us to make the following observations:

- At any point in time, the finite automaton maintains only the following information – the state $q \in Q$. It has no idea what has been read thus far, the sequence of states it passed through, etc. the finite control, $\delta$, is responsible for appropriately changing the state, and the state reached at the end tells us whether the string is accepted or not. Based on this intuition, and the fact that $Q$ is finite, it is instructive to think of finite automata as machines with finite memory\(^1\). Going forward, in automata design, it will be extremely helpful to answer the question – “To solve this problem, what must I remember?”. In the above example, it suffices to remember the parity of the number of 0s read thus far, and that is precisely the function of the two states.

- Each state can be thought of as a representative for a certain set of strings, that is, the set of all strings that make the automaton reach that state when they are read. In the example above, $q_0$ represents the set of all strings with an even number of 0s and $q_1$ represents all those strings with an odd number of 0s.

- By designing an automaton, we actually design automata for more than one language. For instance, the language of the automaton below is the set of all binary strings with an even number of 0s.

\(^1\)We can also think of this as bounded memory, where the bound is independent of the input.
The language of the automaton below is the set of all binary strings with either an even number of 0s or an odd number of 0s, that is, the set of all binary strings.

The language of the automaton below is the set of all binary strings with neither an even number of 0s nor an odd number of 0s, that is, the empty set.

Thus, by varying the set of accepting states, we obtain automata for many different languages.

- The sets of strings represented by various states are disjoint. This follows from definition as on reading a string, the automaton can only be in one state. But this is a very useful observation. Put differently, the states of the automaton partition the set of all strings over the alphabet. Thus, the number of states one needs in an automaton is really the number of partitions of the set of all strings one has to make. For instance, in the example above, the partitions are the set of all strings with an even number of 0s and the set of all strings with an odd number of 0s. It is easy to see that if there is at least one string accepted and one string rejected by the automaton, the automaton must have at least two states. Given a language, determining how large the automaton recognizing it must be is the so called state minimization problem. This is will come up later in the course.

- Although the automaton may reach accepting states while reading a string, it may still reject the string if the state it ends up in is not an accepting state. Thus reaching an accept state in the middle of the string has no bearing on whether the string will be accepted.
Regular Operations

Recall that union, concatenation and Kleene star were introduced as being the regular operations. For the sake of completeness, recall that concatenation is defined as follows: For languages $A, B$,

$$A \circ B = AB = \{xy : x \in A \land y \in B\}$$

where $xy$ denotes the concatenation of the strings $x$ and $y$. We denote $A \circ A = A^2$, $A \circ A^2 = A^3$ and so on. Finally,

$$A^* = \bigcup_{k=0}^{\infty} A^k$$

where $A^0 := \{\varepsilon\}$. This explains why $\Sigma^*$ is the set of all possible strings over the alphabet $\Sigma$. Recall also that we have already shown that regular languages are closed under union and intersection and that we will eventually show that they are closed under all the regular operations.

We make the following observation on the proof done in class that showed that regular languages were closed under union. Firstly, the proof was constructive. In other words, given the automata for the languages under consideration, one can construct explicitly the automaton for the union. It has a number of states equal to the product of the numbers of states in the given automata. The same is true of the proof that showed that regular languages were closed under intersection.

Building Finite Automata

**Automaton for the set of all strings, $\Sigma^*$**. Since, we don’t need to treat any string differently, that is we accept all strings, we would need only one state. The automaton is shown below.

![Automaton for the set of all strings, $\Sigma^*$](image)

**Automaton for the empty set, $\phi$**. Since, we don’t need to treat any string differently, that is we reject all strings, we would need only one state. The automaton is shown below.

![Automaton for the empty set, $\phi$](image)

**Automaton for the empty string, $\{\varepsilon\}$**. Recall that the empty string $\varepsilon$ is the string of length 0. Notice that we need at least two states. We need a state that stores the information – “No symbols have appeared thus far and this (empty) string shall be accepted”. That will be our start state $q_0$ as that is the state the automaton begins in. We let the automaton transition to a so called dead state, $q_1$, on reading any symbol. The dead state is called so because once the automaton lands in a dead state, there is nothing it can read that will take it to an accept state. In other words, we use it as a sink for the automaton. The automaton is shown below.

![Automaton for the empty string, $\{\varepsilon\}$](image)
Automaton for all strings but the empty string, $\Sigma^* \setminus \{\varepsilon\}$. From our earlier observations, note that the state $q_1$ in the automaton above represents all strings but the empty string. Thus making $q_1$ an accepting state and $q_0$ a non-accepting state suffices. The automaton is shown below.

On closer inspection of all the examples we have seen thus far, we note that if we have an automaton $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes a language $L$, the automaton $M' = (Q, \Sigma, \delta, q_0, Q \setminus F)$ obtained by swapping the accepting and non-accepting states recognizes the complement language $\overline{L} = \Sigma^* \setminus L$. This further informs us that regular languages are closed under complementation or simply, complement.

Automaton for the string 00, $\{00\}$. The automaton is shown below.

Let us describe how the automaton works. The sequence of states $q_0, q_1, q_2$ is meant to track the exact reading of the string 00. Anything that does not conform to this is immediately sent to the dead state $q_3$. Note that $q_0$ represents the empty string, $\{\varepsilon\}$, $q_1$ represents the string 0, $\{0\}$, $q_2$ represents the string 00, $\{00\}$ and $q_3$ represents all strings but the empty string, 0 and 00, that is, $\Sigma^* \setminus \{\varepsilon, 0, 00\}$.

Automaton for the string 01, $\{01\}$. The automaton is shown below.
Automaton for the an arbitrary string $b_1 b_2 \ldots b_k$, \{b_1 b_2 \ldots b_k\}. The automaton is shown below.

Let us describe how the automaton works. The sequence of states $q_0, q_1, \ldots, q_k$ is meant to track the exact reading a sequence of $k$ symbols from $\Sigma$. After reading $k$ symbols, a read immediately takes the automaton to the dead state $q_{k+1}$. Note that $q_0$ represents the empty string, $\Sigma^0 = \{\varepsilon\}$,
$q_1$ represents the strings of length 1, $\Sigma = \Sigma^1 = \{0,1\}$, $q_2$ represents the strings of length 2, $\Sigma^2 = \{00,01,10,11\}$ and in general for $0 \leq i \leq k$, $q_i$ represents all strings of length $i$, $\Sigma^i$ and $q_{k+1}$ represents all strings of length greater than $k$, that is

$$\Sigma^* \setminus \bigcup_{i=0}^{k} \Sigma^i = \bigcup_{i=k+1}^{\infty} \Sigma^i$$

**Automaton for the strings** $00$ and $01$, $\{00,01\}$. Note that we have $\{00,01\} = \{00\} \cup \{01\}$. Furthermore, we have automata for $\{00\}$ and $\{01\}$. Using the construction we developed to show that regular languages are closed under union, we can construct an automaton for $\{00,01\}$. As noted above, it would have $4 \cdot 4 = 16$ states. In fact, we can generalize this observation and see that any finite language is regular. If $L$ is a finite language consisting of strings $s_1, \ldots, s_n$ where the length of the string $s_i$ is $k_i$, we know that there is automaton for $L$ that has

$$N = \prod_{i=1}^{n} (k_i + 2)$$

states.

We also, however, note that is possible to design a simpler automata for finite languages. Let us start with our initial example. Consider $L = \{00,01\}$. Thinking constructively, we see that the strings we accept are precisely those that start with a 0 and are of length 2. Using our techniques, we arrive at the automaton below.

![Automaton](image)

Notice that this automaton has only 4 states, while the automaton obtained from the union construction would have had 16 states. Let us extend this a bit further and attempt to construct an automaton for the language $\{00,01,010\}$. We arrive at the automaton below.
It is easy to see how this approach generalizes to any finite language. In fact, the automaton essentially mirrors what is called the trie data structure of the language, along with a dead state, \( q_{\text{dead}} \). It is constructed by keeping track of common prefixes. It is also instructive to observe that an automaton for a finite language constructed this way is much smaller than one obtained from the union construction.

**Automaton for the strings that start and end with the same symbol.** One thing that is clear at the outset is that we must remember the start symbol. We will thus designate two states, \( q_0 \) and \( q_1 \) that store the information the string started with the symbol 0 and 1 respectively. To complete the construction, we simply ensure that in these two branches, the strings accepted are those that end with 0 and 1 respectively. The automaton is shown below.

Note that $q_{01}$ represents strings that start with 0 and end with 1 while $q_{10}$ represents strings that start with 1 and end with 0. As required, $q_0$ represents strings that start and end with 0 while $q_1$ represents strings that start and end with 1. Of course, $q_{\varepsilon}$ represents the empty string, $\{\varepsilon\}$.

**Arithmetic with Finite Automata**

We have already seen how we can use finite automata to compute the parity of the number of the number of 0s in a string. We extend this idea in this section. As a warm-up, we construct an automaton that accepts strings which contain a number of 0s that is a multiple of 3. The automaton is shown below.
Notice that all states are *oblivious* to 1s. Since they are the only states, the strings accepted or rejected by the automaton would have nothing to do with 1s. Furthermore, the states toggle in a cycle of size 3 on reading a 0. So, the state of the automaton reflects the number of 0s read thus far modulo 3. It is easy to see that \(q_0\) represents all strings whose number of 0s is 0 modulo 3, \(q_1\) represents all strings whose number of 0s is 1 modulo 3 and \(q_2\) represents all strings whose number of 0s is 2 modulo 3. The automaton below recognizes the set of strings whose number of 0s is either 2 or 3 modulo 4.

![Automaton Diagram]

It is easy to see how the above ideas can be generalized in order to “count” the number of 0s in a string modulo any fixed \(k\). The automaton would essentially be a \(k\) cycle.

As a final exercise, we see how finite automata can be used to perform division. We will design a finite automaton that computes the remainder obtained when a binary number, the input, is divided by some fixed number \(k\). It is instructive to note that this is a well-defined problem\(^3\). As a starting point, we recall our well known method of long division. Consider the example shown below.

\[
\begin{array}{c|c|c}
13 & 17352 & 1334 \\
13 & \underline{43} & \\
39 & \underline{45} & \\
39 & \underline{62} & \\
52 & \underline{10} & \\
\end{array}
\]

We now attempt to mathematically understand this procedure. One point to note at the outset is that the procedure proceeds by reading the input, in this case 17352, symbol by symbol from left

---

\(3\)Also note that other versions of essentially the same problem may not be well-defined, such as, computing the quotient. It is important to understand why this is the case, and that problem definition is key.
to right. On reading the symbol 1, the quotient is 0 and the remainder is 1. On reading 7, the quotient becomes 01 or 1 and the remainder 4. We reinterpret this as follows. On reading 1, the remainder is 1. On reading 7, the remainder is 17 modulo 13, which is 4. Notice that this indeed captures the process. Next, on reading 3, the remainder is 43 modulo 13, which is 4. On reading 5, the remainder is 45 modulo 13, which is 6, and on reading 2, the remainder is 62 modulo 13, which is 10.

To unpack this further, from Horner’s rule (or simple logic),

\[ 17352 = ((((1 \cdot 10 + 7) \cdot 10 + 3) \cdot 10 + 5) \cdot 10 + 2) \]

Using modular arithmetic,

\[
17352 \mod 13 = ((((1 \mod 13) \cdot 10 + 7) \mod 13 \cdot 10 + 3) \mod 13 \cdot 10 + 5) \mod 13 \cdot 10 + 2) \mod 13 \\
= (((((1 \cdot 10 + 7) \mod 13 \cdot 10 + 3) \mod 13 \cdot 10 + 5) \mod 13 \cdot 10 + 2) \mod 13 \\
= (((17 \mod 13 \cdot 10 + 3) \mod 13 \cdot 10 + 5) \mod 13 \cdot 10 + 2) \mod 13 \\
= (((4 \cdot 10 + 3) \mod 13 \cdot 10 + 5) \mod 13 \cdot 10 + 2) \mod 13 \\
= (43 \mod 13 \cdot 10 + 5) \mod 13 \cdot 10 + 2) \mod 13 \\
= (4 \cdot 10 + 5) \mod 13 \cdot 10 + 2) \mod 13 \\
= (6 \cdot 10 + 2) \mod 13 \\
= 62 \mod 13 \\
= 10
\]

This explains why long division works the way it does. To simplify this, we can hence say that if the current remainder modulo \( k \) is \( r \), and we read the symbol \( s \), and we are working in base \( B \), the new remainder is going to be \( r' = (r \cdot B + s) \mod k \). We just saw this with \( B = 10 \) and \( k = 13 \). This tells us how to build an automaton that performs binary division, that is, computes the remainder when a binary number is divided by any fixed number \( k \). Formally, the automaton would be \( M_{\text{div},k} = (Q_k, \Sigma, \delta_k, q_0, F) \) where \( Q_k = \{q_0, \ldots, q_{k-1}\} \) and \( \delta_k \) would be defined by

\[
\delta_k(q_i, b) = q_{(i \cdot 2+b) \mod k}
\]

for every \( 0 \leq i \leq k - 1 \) and \( b \in \Sigma = \{0, 1\} \). Notice that we have basically created one state for every possible remainder from 0 through \( k - 1 \) and state \( q_i \) represents the set of all binary strings which produce remainder \( i \) when divided by \( k \). We will define \( F \) as required, that is, depending on which remainders would be accepted. It is also important to note the power of the formalism that we have developed in expressing this finite automaton so succinctly. As an illustration of this, the diagrammatic representation of the automaton accepting binary strings with remainder 5 when divided by 7 is shown below.
Structure versus Hardness

It is important to understand that the more intricate structure a language has, the harder the task of recognizing it. For instance, languages such as $\Sigma^*$ and $\phi$ only need a finite automaton with a single state, while the automata we constructed for some of the languages has 6 or more states. Of course, one must deliberate on whether these automata are minimal. However, a little reflection will show that it is indeed the case that certain languages require finite automata with many states. Some, as seen in class such as the set of all strings with an equal number of 0 and 1s, cannot even be recognized by a finite automaton. In some sense, these languages are “harder” to recognize because of their finer, more “complex” structure, and demand stronger computational models. Throughout this course, we will study many such classes of languages and models of computation in our effort to study the structure versus hardness relationship of languages.