1 Problems as Languages

For each of the following problems:

i. Formulate the problem as a language (give an example of the problem instances and how they are encoded, you don't have to write every problem instance).

Note that how you encode the language matters for the regular expression you end up with.

1. Checking whether (or not) a number is divisible by 4). You are given a binary number and need to output if this number is divisible by 4.

Solution: Part(i) Intuition: Note that if a binary number is divisible by 4, then it must have 2 zeroes in the suffix.

Strategy: Assume we want to formulate this as the language: ($L_{Div4?}$). For every binary number x, we:

- Add the string $w = x \cdot$"|1" to $L_{Div4?}$ if the two-character suffix of $x[0:1] = 00$. (x is divisible by 4). We add the "|" to separate the input from the output of the problem.
- Add the string $w = x \cdot$"|0" to $L_{Div4?}$ if the two-character suffix of $x[0:1] =$ 01 or 10 or 11. (x is *not* divisible by 4).

Part(ii) Formulating the language like we did above makes the regular expression very easy:

$$r_{DIV4?} = \underbrace{(0+1)^*00|1}_{\text{all output 1 instances}} + \overbrace{((0+1)^*(01+10+11)|0"}^{\text{all output 0 instances}}$$

Alternatively, formulating a regular expression that accepts only the binary strings divisible by 4 is also acceptable. Therefore, any equivalents of $(0+1)^*00$ is a valid solution.

■

2. The sum of two *unary* integers.

Solution: Let's first refresh on what unary numbers are. A unary number is base 1. In other words, each digit in a unary number represent $1^{digit\_position}$. Example: $5_{10} = 101_2 = 11111_1$

So the language would be similar to the example discussed in lecture. Let's assume $\Sigma = \{1, +, =\}$. We don't particularly need the "+" and "=" symbols but I added them for visual clarity. Therefore we can describe the language as:

$$L_{+unary} = \left\{ \begin{array}{ccc} +=, & +1=1, & +11=11,\ldots \\ 1+=1, & 1+1=11, & 1+11=111,\ldots \\ 11+=1, & 11+1=111, & 11+11=1111,\ldots \\ \vdots & \vdots & \vdots \\ n+=1....1(\text{n times}), & \ldots, & n+11=1....1(\text{n+2 times}),\ldots \end{array} \right\} \quad (1)$$

■

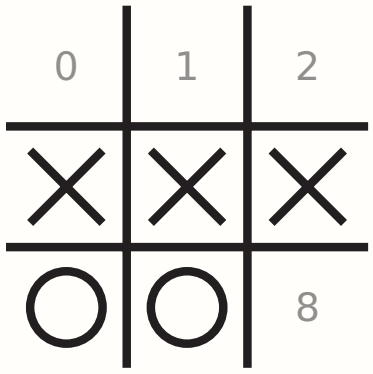
3. The game of TicTacToe. You are given a completed tic-tac-toe board and you need to determine who won. Hint: think about how many games of TicTacToe there are.

Solution: Part (i):

Intuition: So with the game of TicTacToe, first thing to notice is that there are a finite number of games. This means that the language that recognizes a TicTacToe board and calculates it's winner must **have a finite number of strings making it automatically regular.**

Strategy: Let's first focus on formulating the strings in the language. First notice that a TicTacToe board has nine spaces and each space can have one of three values (blank, X, O). We can construct an encoding where each square corresponds to a position on the string like follows:

- The language is composed of 9-character strings. Every charcater on the string corresponds to a space on the TicTacToe board.
- We consider a alphabet $\Sigma = \{\square, \times, \circ\}$ to represent a empty-space, X-mark, O-mark.
- Next we concatenate a "|" symbol to note the end of the board encoding and add a character to mark the winner.



This game of TicTacToe would be encoded as
$w = [\square, \square, \square, \times, \times, \times, \circ, \circ, \square, |, \times]$

There are at most 3^9 possible boards (but remember not all boards are valid since the number o X's and O's must differ by at most 1. **We construct the language (L_{TTT}) by including a string for all the possible games of TicTacToe.**

Part(ii)

I'm not going to type out the regular language explicitly but let's define the regular expression as:

$$r_{TTT} = r_{TTT} + w \qquad \forall\ w \in L_{TTT}$$

As mentioned earlier $|L_{TTT}| < 3^9$ which is finite so our above construction is valid.

■

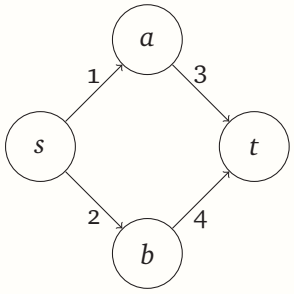
4. The shortest path of a weighted (only positive integer values), undirected graph between 2 nodes s and t.

Solution: Again, this is just a issue of embedding a graph as a string which can be accomplished a multitude of ways. Big thing is to remember what a graph is, a set of nodes and a set of edges. So how do we encode a graph as a string? We simply list out the nodes and edges.

Each string in a language represents an *instance* of the problem. In this case, the problem input is the weighted, undirected, graph and the output is the shortest path. Knowing this, we can formulate a string embedding such as:

$$\langle < listofnodes > | < listofedges > | < shortestpath > \rangle$$

Hence if we have a graph such as:



would yield a sting embedding like:

$$\langle \underbrace{a,b,c,d}_{\text{Nodes in Graph}} | \underbrace{s1a,s2b,a3t,b4t}_{\text{Edges in Graph}} | \underbrace{sat}_{\text{Shortest Path}} \rangle \tag{2}$$

Again I'm including a bunch of extaneous notation for the sake of visual clarity. So the language that defines the shortest path problem would simply be a collection of all these strings.

■

2 Recursive Definitions

Give the recursive definition of the following languages. For both of these you should concisely explain why your solution is correct.

1. A language that contains all strings.

Solution: So the thing to remember with recursive definitions is that you have to have a base case, and a set of rules that could build on so let's do that:

Base case: $\varepsilon \in L_a$

Next we got the inductive step where we define rules to make other strings. The good news is that since this language is any strings, we can simply say:

- $w = 0x$ for some $x \in L_a$, is in L_a
- $w = 1x$ for some $x \in L_a$, is in L_a

■

2. A language which holds all the strings containing the substring 000.

Solution: Very similar to the last problem expect we need to ensure all the strings in L_b have a particular substring. We can accomplish this by simply changing the base case:

- **Base case:** $000 \in L_a$
- $w = 0x$ for some $x \in L_a$, is in L_a
- $w = 1x$ for some $x \in L_a$, is in L_a

But this isn't completely correct because it means there will always be a suffix. Easy to fix with a few more rules:

- $w = x0$ for some $x \in L_a$, is in L_a
- $w = x1$ for some $x \in L_a$, is in L_a

■

3. A language L_A that contains all palindrome strings using some arbitrary alphabet Σ.

Solution: A string $w \in \Sigma^*$ is a palindrome if and only if:

- $w = \varepsilon$, or
- $w = a$ for some symbol $a \in \Sigma$, or
- $w = axa$ for some symbol $a \in \Sigma$ and some palindrome $x \in \Sigma^*$

■

4. A language L_B that does not contain either three 0's or three 1's in a row. E.g., 001101 $\in L_B$ but 10001 is not in L_B.

Solution: We are going to define two languages, L_{B1} and L_{B0} using a mutually recursive definition. L_{B1} contains all strings in L_B that start with 1, and L_{B0} contains all strings in L_B that start with 0. We are also going to have $\varepsilon \in L_{B0}$ and $\varepsilon \in L_{B1}$.

Definition:

- $\varepsilon \in L_{B0}$
- $\varepsilon \in L_{B1}$
- If $x \in L_{B0}$, then $1x$ and $11x$ are in L_{B1}
- If $x \in L_{B1}$, then $0x$ and $00x$ are in L_{B0}

Then $L_B = L_{B1} \cup L_{B0}$ ■