After doing this tutorial you should be able to:

1. design a regular expression given an English specification,
2. write an English description of the language of a given regular expression,
3. reason and prove results about regular expressions,
4. write and justify recursive definitions/processes.

Problem 1. Let $\Sigma = \{a, b\}$. Write regular expressions for the following:

1. The set of strings ending in a.
2. The set of strings that have aba as a substring.
3. The set of strings that do not contain ab as a substring.
4. The set of strings of even length.
5. The set of strings with an even number of a's.
6. The set of strings not containing bab as a substring (hard).

Solution 1.

1. $(a \mid b)^* a$.
2. $(a \mid b)^* aba (a \mid b)^*$
3. $b^* a^*$
4. $((a \mid b)(a \mid b))^*$
5. $(b^*((ab^*)(ab^*))^* \cup b^*$, also written $(b^*ab^*ab^*)^* \cup b^*$.
6. $a^*(b^*aaa^*)^* b^* a^*$.

Problem 2. Let $\Sigma = \{0, 1\}$. We use the following shorthands: Σ instead of $(0 \mid 1)$, and R^+ to mean RR^*.

Write English descriptions for the language of each of the following regular expressions:

1. $\Sigma^* 1 \Sigma\Sigma$.
2. $(\Sigma\Sigma\Sigma)^*$
3. $(\Sigma^* 0 \Sigma^* 1 \Sigma^*) \mid (\Sigma^* 1 \Sigma^* 0 \Sigma^*)$.
4. $(1^*01^*01^*0)^*1^*$
5. $(0^*10)^*0^*110^*(010^*)^*$ (hard)

Solution 2.

1. The set of strings whose third symbol from the right is 1.
2. The set of strings whose length is a multiple of three.
3. The set of strings containing at least one 0 and at least one 1.
4. The set of strings whose number of 0's is a multiple of three.
5. The set of strings with exactly one pair of consecutive 1s.

Problem 3. Let $\Sigma = \{a, b, c\}$. Which of the languages represented by the following regular expressions are infinite? Give reasons.

1. $a(bc^*)$
2. $(a \mid b)c$
3. $(a \mid b)^*$
4. $\emptyset$
5. $\emptyset^*$
6. ϵ^*

Solution 3.

1. Infinite since the language consists of the strings of the form $abc, abcc, abccc, etc.$
2. Finite since it only contains the strings ac and bc
3. Infinite since its language contains all the strings over the alphabet.
4. Finite since it contains no string
5. Finite since it contains just the empty string
6. Finite since it contains just the empty string

Problem 4. Argue that every finite language can be represented by a regular expression.

Solution 4. For every string $s = a_1 a_2 \cdots a_k \in \Sigma^*$, we have $L(a_1 \cdot a_2 \cdot a_3 \cdots a_k) = \{s\}$, but since we don't need to write the concatenation symbol we can write $L(s) = \{s\}$. The language $\{s_1, s_2, \cdots, s_n\}$ is the language of the regular expression

$$s_1 \mid s_2 \mid \cdots \mid s_n$$

Problem 5.

1. Show that $(R^*)^*$ and R^* represent the same language.
2. Show that R^* and $\epsilon \mid RR^*$ represent the same language.

Solution 5.

1. We show that $L(R^*) \subseteq L((R^*)^*))$ and that $L((R^*)^*) \subseteq L(R^*)$.
   (a) For a regular expression S, we have that $L(S) \subseteq L(S^*)$, just take $k = 1$ in the definition of $*$. So, taking $S = R^*$, we have that $L(R^*) \subseteq L((R^*)^*)$.
   (b) If w matches $(R^*)^*$, then $w = u_1 u_2 \cdots u_k$ where each $u_i \in R^*$. So each $u_i = v_i^1 \cdots v_i^{j_i}$ with $v_i^l \in L(R)$ for every l. So $w = v_1^1 \cdots v_1^{j_1} v_2^1 \cdots v_2^{j_2} \cdots v_k^1 \cdots v_k^{j_k}$, and so $w \in L(R^*)$.
2. Note that $w \in L(R^*)$ iff w is either the empty string or of the form $v_1 v_2 \cdots v_k$ where each $v_i \in L(R)$ and $k \geq 1$, iff w matches ϵ or RR^* iff w matches $\epsilon \mid RR^*$.

Problem 6. Give a recursive procedure that decides, given R, if $L(R)$ contains the empty string. You should explain why each case is correct.

Solution 6. We define a procedure EmptystringRE which takes a regular expression R as input, and returns 1 if $\epsilon \in L(R)$ and 0 otherwise, as follows:

EmptystringRE(R):

1. if $R = \emptyset$ or $R = a$ for $a \in \Sigma$, return 0.

   This is correct because $L(\emptyset)$ is empty, and $L(a)$ only contains a.

2. if $R = \epsilon$ return 1.

   This is correct because $L(\epsilon) = \{\epsilon\}$.

3. if $R = R_1 \cup R_2$ return 1 if either EmptystringRE(R_1) $= 1$ or EmptystringRE(R_2) $= 1$ (or both).

   This is correct because $L(R) = L(R_1) \cup L(R_2)$. In a little more detail, for any languages A, B and any string x we have that $x \in A \cup B$ if and only if (i.e., exactly when) $x \in A$ or $x \in B$.

4. if $R = R_1 R_2$ return 1 if both EmptystringRE(R_1) $= 1$ and EmptystringRE(R_2) $= 1$.

   This is correct because for any languages A, B, we have that $\epsilon \in AB$ if and only if ϵ can be split into two pieces $x \in A, y \in B$ such that $xy = \epsilon$, but the only way this can happen is if $x = y = \epsilon$.

5. if $R = S^*$ return 1.

This is correct because the star of every language contains the emptystring.

Problem 7. In this problem you will design an algorithm for solving the membership problem for regular expressions (over a fixed alphabet Σ), i.e., write a recursive procedure $Reg(R, x)$ that takes as input a regular expression R over Σ and a string $x \in \Sigma^*$, and returns 1 if $x \in L(R)$, and 0 otherwise.

For instance, say $\Sigma = \{0, 1\}$ and $R = (0^*1^*) \,|\, (1^*0^*)$. If $x = 00111$ then the algorithm returns 1, and if $x = 001100$ then the algorithm returns 0.

1. Provide (high-level) pseudocode for your algorithm and very briefly describe the main idea(s) in plain English.
2. Briefly argue that your algorithm is correct.

Hint: do not worry about complexity. Just get an algorithm that works.

Solution 7. We will give a straightforward recursive algorithm that runs in time exponential in $|R|$ and $|x|$. However, there is an algorithm to solve this that runs in worst case time $O(|R||x|)$ and space $O(|R|)$. We will see how to do this after we learn about machine models of computation.

1. Very briefly, the algorithm recurses over the syntactic structure of the given regular expression, and for each case mimics the semantic definition of the language represented by that expression. In particular, for the concatenation and star operations, it exhaustively iterates over all 'break points' and 'partitions' of the string.

   Here is high-level pseudocode. Let's use `Python` indexing and slice notation – recall this means that for a string x, we have that $x[0]$ is the first symbol in x, and $x[i : j]$ is the substring of x from position i (inclusive) to position j (exclusive).

   $Reg(R, x)$:

   (a) if $R = \emptyset$ then return 0.
   (b) if $R = \epsilon$ then return 1 if $x = \epsilon$ and otherwise return 0.
   (c) if $R = a$ for $a \in \Sigma$ then return 1 if $x = a$ and otherwise return 0.
   (d) if $R = R_1 \,|\, R_2$ then return

   $$\max\{Reg(R_1, x), Reg(R_2, x)\}$$

   (e) if $R = R_1 R_2$ then return iterate through $i \in [0 : len(x)]$ and return

   $$\max_{i \in [0:len(x)]} \min\{Reg(R_1, x[0 : i]), Reg(R_2, x[i : len(x)])\}$$

(f) if $R = S^*$ then if $x = \epsilon$ return 1, else return

$$\max_{i \in [1:len(x)]} \min\{Reg(S, x[0:i]), Reg(R, x[i:len(x)])\}$$

2. Each case mimics the definition of the semantics of that case.

   (a) If $R = \emptyset$ then the definition of $L(R)$ is that that $L(R) = \emptyset$, and so no string is in $L(R)$. Thus in line (a) we return 0 (ignoring x).

   (b) if $R = \epsilon$ then $L(R) = \{\epsilon\}$ and so in line (b) we check if $x = \epsilon$.

   (c) if $R = a$ for $a \in \Sigma$ then $L(R) = \{a\}$ and so in line (c) we check if $x = a$.

   (d) if $R = R_1 \mid R_2$ then $L(R) = L(R_1) \cup L(R_2)$ and so in line (d) we check if the input string is in $L(R_1)$ or in $L(R_2)$ by recursively calling the procedure on (R_1, x) and (R_2, x).

   (e) If $R = R_1R_2$ then $L(R) = L(R_1)L(R_2)$, and so in line (e) we check if there is a partition of $x = u_1u_2$ and recursively check if $u_1 \in L(R_1)$ and $u_2 \in L(R_2)$ by recursively calling the procedure on (R_1, u_1) and (R_2, u_2).

   (f) If $R = S^*$ then $L(R) = L(S)^*$ and so in line (f) we check if the input string is the empty string (and return 1 if it is, since $\epsilon \in L(S^*)$ for any regular expression S), and if not, check there is a partition of $x = uv$ and recursively check if $u \in L(S)$ and $v \in L(S^*)$. This is correct since a non-empty string is in $L(S^*)$ exactly when it is in $L(SS^*)$.

   - Note that we make two different types of recursive calls in the star case. One type is like in the other cases and is of the form $Reg(S, x)$ on the same string x but on a subexpression S of R; and the other is of the form $Reg(R, y)$ on the same expression R but on a substring y of x. In either case, something is getting "smaller", and so the recursion must eventually stop (i.e., there cannot be infinite regress).

Problem 8. Discuss whether focusing on input strings is a serious restriction to studying computation? For instance, how would you encode an integer as a string? Or a finite set of integers? or a graph? Is there any computational object that we can't encode as a string?

Solution 8. It seems we can encode almost anything as a string. For instance, an integer can be encoded as a binary string. A graph can be encoded as an adjacency matrix, which itself can be encoded as a string, e.g., the matrix with rows 011, 110, and 111, can be encoded by the single string 011#110#111.

Note also that programs themselves are strings.