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 of even length.
4. The set of strings with an even number of a's.
5. The set of strings that do not contain ab as a substring.
6. The set of strings not equal to ab. (Exam 2022)
7. 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. $((a \mid b)(a \mid b))^*$
4. $(b^*ab^*ab^*)^* \mid b^*$, also written $(b^*ab^*ab^*)^*b^*$
5. b^*a^*
6. $\epsilon \mid a \mid b \mid aa \mid ba \mid bb \mid (a \mid b)(a \mid b)(a \mid b)^+$
7. $a^*(b^*aaa^*)^*b^*a^*$.

Problem 2. Let $\Sigma = \{0, 1\}$.

We use the following shorthand: Σ to mean $(0 \mid 1)$

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; hint: it is testing if a certain string occurs a certain number of times).

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. (Assignment Question in 2023, extra)

For each of the following three languages over $\Sigma = \{a, b\}$, provide a regular expression for the language.

1. The set of strings that contain *abbab*, in that order, but not necessarily consecutively.

   Said differently, the set of strings such that one can delete some amount of letters, possibly zero, to obtain *abbab*.

   For example, *aaababaababba* is in the language, while *bbbababbbaa* is not.
2. The set of strings of length 4 that contain exactly two *b*s.

   For example, *abba* is in the language, while *abbb* and *abb* are not.
3. The set of strings that contain exactly three *a*s and an even number of *b*s.

   For example, *abababb* is in the language, while *ababab* is not.

Solution 3. Here are regular expressions for each language:

1. $(a|b)^*a(a|b)^*b(a|b)^*b(a|b)^*a(a|b)^*b(a|b)^*$
2. $(aabb)|(abab)|(baab)|(abba)|(baba)|(bbaa)$
3. Use the facts that 'even plus even = even', 'even plus odd = odd', and 'odd plus odd = even'. Let $E = (bb)^*$ and $O = b(bb)^*$.

   $$(EaEaEaE)|(OaOaEaE)|(OaEaOaE)|(OaEaEaO)|$$
   $$(EaOaOaE)|(EaOaEaO)|(EaEaOaO)|(OaOaOaO)$$

   A slightly more concise expression can be constructed based on the idea that if and only if the string is in the language, then the substrings to the left and right of the central *a* have one *a* each and they either both contain

an odd number of bs or they both have an even number of bs. We can then construct our answer R as follows:

$$\begin{aligned}
X_{odd} &= \big(b(bb)^*a(bb)^* \mid (bb)^*a(bb)^*b\big) \quad \because \text{odd+even=odd, even+odd=odd} \\
X_{even} &= \big((bb)^*a(bb)^* \mid b(bb)^*a(bb)^*b\big) \quad \because \text{even+even=even, odd+odd=even} \\
R &= X_{odd}aX_{odd} \mid X_{even}aX_{even} \\
&= \Big(b(bb)^*a(bb)^* \mid (bb)^*a(bb)^*b\Big)a\Big(b(bb)^*a(bb)^* \mid (bb)^*a(bb)^*b\Big) \mid \\
&\quad \Big((bb)^*a(bb)^* \mid b(bb)^*a(bb)^*b\Big)a\Big((bb)^*a(bb)^* \mid b(bb)^*a(bb)^*b\Big)
\end{aligned}$$

An even more concise answer:

$$(bb)^*\Big((a \mid bab)(bb)^*a(bb)^*(a \mid bab) \mid (ab|ba)(bb)^*a(bb)^*(ab \mid ba)\Big)(bb)^*$$

Problem 4. Give a recursive procedure that decides, given R, if $L(R)$ contains the empty string. Make sure you understand why each case is correct.

Recall the recursive definition of regular expressions: a regular expression is one of $\emptyset, \epsilon, a \in \Sigma, (R_1|R_2), (R_1R_2), R_1^*$. Hence your procedure should handle these cases separately.

Solution 4. 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 $R = \emptyset$ return 0.

   This is correct because $L(\emptyset)$ is empty.

2. if $R = a$ (where $a \in \Sigma$) return 0.

   This is correct because $L(a)$ only contains a.

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

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

4. if $R = R_1 \,|\, R_2$ 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 $x \in B$.

5. if $R = R_1R_2$ 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 $x = y = \epsilon$.

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

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

Let's run this on a bunch of different inputs. We write $E(R)$ instead of EMPTYSTRINGRE(R) for simplicity.

Let $R = (011)|(10)^*$. Then $E(R) = 1$ if either $E(011) = 1$ or $E((10)^*) = 1$. Now $E(0) = E(1) = 0$ by Case 2, and so $E(011) = 0$ by Case 5. But $E((10)^*) = 1$ by Case 6. So $E(R) = 1$.

Let $R = \emptyset(10)^*$. Then $E(R) = 1$ if both $E(\emptyset) = 1$ and $E((10)^*) = 1$. We already saw that $E((10)^*) = 1$. But $E(\emptyset) = 0$ by Case 1. So $E(R) = 0$.

Problem 5. Regular expressions can be used to search text for patterns. How? By solving the membership problem for regular expressions. In this problem you will design an algorithm for this problem (over a fixed alphabet Σ). You will write a recursive procedure Match(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. Make sure you understand why your algorithm is correct.

Hint: do not worry about finding an algorithm with good complexity (i.e., fast or memory efficient). Just get a correct algorithm. Later in the course we will see how to solve the membership problem in polynomial time. For a history of regular-expression matching see https://www.youtube.com/watch?v=NTfOnGZUZDk

Solution 5. 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).

Match(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 $R = R_1 \mid R_2$ then return

$$\max\{\text{Match}(R_1, x), \text{Match}(R_2, x)\}$$

(e) if $R = R_1R_2$ then return iterate through $i \in [0 : len(x)]$ and return

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

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

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

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 Match(S, x) on the same string x but on a subexpression S of R; and the other is of the form Match(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 6. (Assignment Question in 2022)

Let $\Sigma = \{a, b\}$. For each of the following regular expressions, provide a clear and concise English description for the language of that regular expression. No justification is necessary.

1. $(a|b)^*((aaa)|(bbb))(a|b)$
2. $b^*(abb^*)^*(a|\epsilon)$

Solution 6.

1. The set of strings whose 4th, 3rd, and 2nd last letters are all the same.
2. The set of strings that do not contain aa as a substring.

Problem 7. (Assignment Question in 2022)

Let $\Sigma = \{a, b\}$. For each of the following languages, provide a regular expression for that language. No justification is necessary.

1. The set of strings that contain at most two b.
2. The set of strings that have even length or contain no a, but not both.
3. The set of strings that contain exactly two out of the four length 2 strings (aa, ab, ba, bb) as substrings.

Solution 7.

1. $a^*(b|\epsilon)a^*(b|\epsilon)a^*$
2. The idea is that we check for two cases: odd length and containing no a, or even length and containing an a. The former is quite easy to do directly. For the latter, we note that if a string has even length and contains an a, the part before the a must be even and the part after odd or vice versa. Hence: $(b(bb)^*)|((\Sigma\Sigma)^*((a\Sigma)|(\Sigma a))(\Sigma\Sigma)^*)$
3. We analyse the cases separately:
   (a) aa and bb is impossible, since the 'transition point' between a and b must feature ab or ba.
   (b) aa and ab means the string starts with some number of a (at least 2), followed by exactly one b.
   (c) bb and ba is symmetric to the above.
   (d) aa and ba is the opposite of the above: the string starts with a single b, followed by some number of a (at least 2).
   (e) bb and ab is symmetric to the above.
   (f) ab and ba means the string contains alternating a and b. It also needs to have length at least 3.

   Hence: $(aaa^*b)|(bbb^*a)|(baaa^*)|(abbb^*)|(aba(ba)^*(b|\epsilon))|(bab(ab)^*(a|\epsilon))$

Problem 8. (Assignment Question in 2022) (hard) Give a recursive procedure that decides, given R, if $L(R)$ is infinite. Explain why each case is correct.

Hint: It is likely necessary to design a procedure that obtains and uses some additional information about the subexpressions beyond whether their languages are infinite. Be very careful with regards to edge cases, such as the empty set.

Solution 8. We provide a recursive procedure. We define three functions (from the regexes to $\{0,1\}$): INF, which decides whether the language of a given regex is infinite, CS ('contains string'), which decides whether the language of a given regex contains at least one string, and CNE ('contains non-empty string'), which decides whether the language of a given regex contains at least one non-empty string.

1. if $R = \emptyset$, $R = a$ for $a \in \Sigma$ or $R = \epsilon$, then $INF(R) = 0$

   This is clearly correct since the languages of each base case contain either 0 or 1 element, and so none are infinite.

2. if $R = R_1|R_2$, then $INF(R) = 1$ if $INF(R_1) = 1$ or $INF(R_2) = 1$ (or both), and $INF(R) = 0$ otherwise

   This is correct because the union of an infinite set with any set is infinite, while the union of two finite sets is finite.

3. if $R = S^*$, then $INF(R) = CNE(S)$

   This is correct because if $L(S)$ has a non-empty string x, $L(R)$ is infinite (as it contains x^n for every positive integer n), while if L(S) does not have such a string, $L(R)$ is just $\{\epsilon\}$ which is finite.

4. if $R = R_1R_2$, then $INF(R) = 1$ if ($CS(R_1) = 1$ and $CS(R_2) = 1$) and ($INF(R_1) = 1$ or $INF(R_2) = 1$, or both), and $INF(R) = 0$ otherwise

   This is correct because if one language is infinite and the other has at least one string, say x, then R has infinitely many strings (by concatenating any of the infinitely many strings with x), while if this is not the case, then either at least one of the languages is empty (so the concatenation is empty) or both are finite (so the concatenation is finite).

We now define our helper functions.

1. if $R = \emptyset$, then $CS(R) = CNE(R) = 0$
2. if $R = \epsilon$, then $CS(R) = 1, CNE(R) = 0$
3. if $R = a$ for $a \in \Sigma$, then $CS(R) = 1, CNE(R) = 1$

   The base cases are clearly correct.

4. if $R = R_1|R_2$, then $CS(R) = 1$ if $CS(R_1) = 1$ or $CS(R_2) = 1$ (or both), 0 otherwise, and similarly for CNE.

   This is correct because a union contains a string if and only if one of the underlying sets contains a string, and similarly for containing a nonempty string.

5. if $R = S^*$, then $CS(R) = 1, CNE(R) = CNE(S)$

   This is correct because $L(R)$ always contains ϵ, while it contains a nonempty string if and only if S does.

6. if $R = R_1R_2$, then $CS(R) = 1$ if $CS(R_1) = 1$ and $CS(R_2) = 1$, and 0 otherwise. Also, $CNE(R) = 1$ if ($CS(R_1) = 1$ and $CS(R_2) = 1$) and (either $CNE(R_1) = 1$ or $CNE(R_2) = 1$, or both), and 0 otherwise.

   This is correct because to produce a string from a concatenation, we need at least one string in each language, and to produce a nonempty string, we further need at least one of those strings to be nonempty.

As each recursive case only considers the lower subexpressions, this produces a correct recursive procedure to solve the given problem by evaluating $INF(R)$.