Homework 3

CS 181, Fall 2025

Out: Oct. 14
Due: Oct. 24, 11:59 PM

Please upload your solutions on Gradescope. You can use LaTeX or a word document to write up your answers, but we prefer you use LaTeX. You can use the **tikz package** in LaTeX to draw the DFAs in this homework. This tool is a user-friendly way of generating tikz graphs. You may scan hand-written work or images for parts of solutions **only if** they are extremely clean and legible. Please ensure that your name does not appear anywhere in your handin.

Problem 1: Correctness of Knuth-Morris-Pratt

In class you have learned about the Knuth-Morris-Pratt algorithm for finding a pattern P in a larger text T. Recall that KMP's improvement over the naïve "sliding window" approach lies in the fact that in KMP we use the knowledge gained from earlier comparisons between P and T to avoid many unnecessary comparisons later on. To formalize this idea, we'll make the following definition.

Definition. *For each position k in the pattern P, let $s_k(P)$ denote the length of the longest proper suffix of $P_{1:k}$ that matches a prefix of P. (The notation $P_{1:k}$ is used to represent the first k characters of P.) If the pattern P is clear from context, we will simply write s_k. Note that $s_k(P)$ is essentially the failure function of P, which we will examine more later on.*

As an example, if $P = \texttt{abcxabcde}$, then $s_2 = s_3 = s_4 = 0$, $s_5 = 1$, $s_6 = 2$, $s_7 = 3$, and $s_8 = 0$.

If a mismatch between the pattern and the text is found at position $k + 1$ of P, then KMP responds by shifting the pattern $k - s_k$ places to the right. To see this rule in action, consider $P = \texttt{abcxabcde}$ and $T = \texttt{xyabcxabcxadcdqfeg}$. Suppose the left end of P is aligned with the third character of T. Then P and T match for 7 characters, but mismatch on the 8th character of P. So P is shifted to the right by $7 - s_7 = 7 - 3 = 4$ characters:

```
xyabcxabcxadcdqfeg
  abcxabcde
      abcxabcde
```

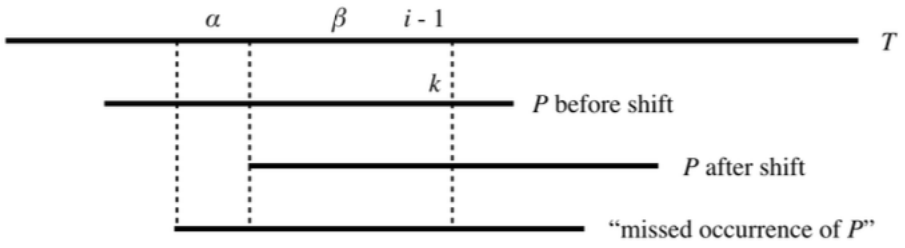
This shifting rule provides two advantages. First, we often shift the pattern by more than a single character, which is an improvement over the naïve algorithm. Second, after the shift is completed, we already know that the first s_k characters in P match their counterparts in T (see the example above). So we can start comparing P to T at position $s_k + 1$ of P, further saving ourselves from doing unnecessary work.

Everything sounds good so far. But hold on! How do we know that the KMP shift rule doesn't move the pattern too far to the right? In other words, how can we be sure that we don't inadvertently skip over the pattern we're looking for in the text? The purpose of this problem is to walk you through a proof of

the following theorem, which should put your mind at ease.

Theorem. *For any alignment of P with T, if characters 1 through k of P match the opposing characters of T but character $k+1$ mismatches with T_i, then P can be shifted by $k - s_k$ places to the right without passing any occurrence of P in T.*

Our proof will proceed by contradiction. In other words, we shall assume that there *is* in fact an occurrence of P in T starting strictly to the left of the shifted P and show that this assumption leads to a contradiction. Our proof will be guided by the following picture. For further explanation, you can refer to this article.



In this diagram,

- α and β are the indicated substrings of T,
- the unshifted pattern P matches T up to position k in P and position $i-1$ in T,
- $P_{k+1} \neq T_i$.

(a) What is the relationship between β and P? What is the length of β? Provide justification for both answers.

(b) Which portion of the missed occurrence of P matches T? Which portion of the unshifted P matches T? Define the matching substring between the unshifted P and the missed P as γ. What is the length of γ and why?

(c) Is a γ a prefix of $P_{1:k}$? a proper prefix? a suffix? a proper suffix? Explain.

(d) Looking at the unshifted P and the missed P, what can we say about the length of α?

(e) Combine your results from (a), (c), and (d) to derive a contradiction and complete the proof.

Problem 2: Failure Function

The following is the pseudocode for computing the failure function, f, for a pattern p:

```
1:  function FAILURE FUNCTION(p = p_1 ... p_l)
2:      f(1) ← 0
3:      i ← 0
4:      for j ∈ {2, ..., l} do
5:          i ← f(j − 1)
6:          while p_j ≠ p_{i+1} and i > 0 do
7:              i ← f(i)
8:          end while
9:          if p_j ≠ p_{i+1} and i = 0 then
10:             f(j) ← 0
11:         else
12:             f(j) ← i + 1
13:         end if
14:     end for
15: end function
```

(a) For each of the following strings, draw the failure function table (i.e. a table mapping i to $f(i)$ for all possible i). There's no need to write out the prefixes and suffixes for full credit but it may save you from some confusion :).

- `ALPALPACA`
- `ABRACADABRA`

(b) Consider line 5 of the pseudocode above. What does i represent in terms of p (1 sentence)?

(c) In each iteration of the while loop in line 6, i is re-assigned in line 7. On the nth iteration of the while loop, what does i represent in terms of p (1 sentence)?

(d) Consider the **if/else** block beginning in line 9. What can we conclude about p if we satisfy the condition "$p_j \neq p_{i+1}$ `and` $i = 0$"? What can we conclude about p if we do not satisfy this condition? Use the words suffix and prefix in your response (2 sentences).

Problem 3: Finite Automata for Pattern Matching

Congratulations! The Sorin's farm has recruited you to join his team and help take care of his animals. While making your rounds around the farm one morning, you discover a lost lamb named Mary wandering in the meadow, separated from the rest of the flock. Having been stuck outside overnight, Mary is disorientated and seems to be feeling unwell. Fortunately, Sorin is also a geneticist! Sorin says that Mary can be reunited with her family in the flock and will quickly recover from any sickness. Mary's family members should have multiple repeats of the following nucleotide sequences in their housekeeping genes (which are essential for general cell functions): CGCGAT, CTCGCGT, and CTCGCGA. Could you help construct a DFA that recognizes sheep that could be Mary's family based on their housekeeping genes?

(a) In particular, construct a DFA that will accept any input that contains at least one of the patterns below. Once any of the patterns is found, the DFA should reach an absorbing acceptance state. Remember to try out this tool.

```
CGCGAT
CTCGCGT
CTCGCGA
```

Note that there are many valid DFAs for this question. You should construct the simplest and most specific one possible (ie. your DFA should include the smallest number of states possible and should not accept any strings that do not contain at least one of the patterns above). You may omit edges only if you clarify which class of edges you omitted, the omitted edges do not create ambiguity in the DFA, and you don't omit additional edges not belonging to your specified class.

(b) Then, write the sequences of states visited when the machine is run on the text `CTCGCTCTCGCGAACC`. Assume that the finite automaton reads the whole string (i.e. there is no early stopping).

(c) In project 2, you will implement an algorithm to construct a DFA for matching a pattern, p, in a text, t. Why can't we construct such a DFA using the following steps: (1) build a skeletal DFA for our pattern such that we have read in $p_{1:j}$ when we reach state q_j, (2) compute the failure function, f for our pattern, (3) add failure transitions to our skeletal DFA such that we transition to $q_{f(j)}$ if we read in a character from t that does not equal p_{j+1} (2 sentences)?

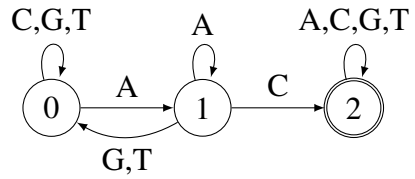
Problem 4: Languages

In the previous two problems, we have focused on building DFAs to recognize whether a pattern p appears anywhere in a text t. As a result, the DFAs constructed in these problems will accept languages of the form $L = \{\text{strings containing pattern } p\}$. However, what if we wanted to build a DFA that recognized a different language, such as $L = \{\text{strings ending with pattern } p\}$?

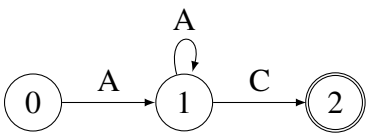
Our focus on DFAs accepting languages of the form $L = \{\text{strings containing pattern } p\}$ has enabled us to make two important assumptions to simplify our drawings of DFAs:

1. Once the entirety of p is found in t, the DFA should accept t regardless of what comes after p in t. Consequently, the acceptance state at the end of the DFA is also an *absorption state*, meaning that the DFA will never exit this state once it is entered. The assumption that all acceptance states are absorption states allows us to exclude arrows from acceptance states to themselves in DFA diagrams.
2. The DFA should accept t if p is found in t, regardless of what comes before p in t. Consequently, whenever the next letter in t is not consistent with p, we can move several states backwards in the DFA and continue searching for p in t (this is the failure function). Because many incorrect next letters in t will send us back to state 0 (the beginning of p), we typically exclude these arrows from DFA diagrams and assume that all missing arrows from non-acceptance states go to state 0.

As an example of these simplifications, consider a DFA accepting $L = \{\text{strings containing the pattern } AC\}$ over the alphabet $\Sigma = \{A, C, G, T\}$. The complete version of this DFA is:



However, the two assumptions above allow us to simplify this DFA to:



For each of the following languages L over the alphabet $\Sigma = \{A, C, G, T\}$, either (i) construct a DFA that accepts L, or (ii) explain why it is not possible to construct a DFA that accepts L. List any assumptions you make that allow you to exclude arrows. *Hint: Think about how, if a string has permanently failed the language's criteria, you can transition to a "dump" state that rejects no matter what comes next.*

(a) $L = \{\text{strings ending with pattern } CT\}$

(b) $L = \{\text{DNA strings of length 4}\}$

(c) $L = \{\text{palindromic DNA strings}\}$

(d) $L = \{\text{strings containing } AC\text{, but not as the first or last two characters}\}$

(e) **Bonus:** Recall from class that DFAs, NFAs, and regular expressions are equivalent in that they all represent the same set of languages—namely, the set of regular languages. In this problem, you will begin to prove the validity of this claim. Prove that every regular expression can be converted into a NFA that accepts the same language. *Hint: Consider each of the cases in the recursive definition of a regular expression to prove this claim recursively.*