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 \( s_k \).

As an example, if \( P = \text{abcxabcde} \), 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 \( P = \text{abcxabcde} \) and \( T = \text{xyabcxabcxadcdqfeg} \). 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.

In this diagram,

- $\alpha$ and $\beta$ are the indicated substrings of $T$,
- the unshifted pattern $P$ matches $T$ up through position $k$ of $P$ and position $i - 1$ and of $T$,
- $P_{k+1} \neq T_i$.

**Task:** Answer the following questions to prove the above theorem.

(a) What is the relationship between $\beta$ and $P$? What is the length of $\beta$? Give justification for both answers.

(b) Which portion of the missed occurrence of $P$ matches $T$? Which portion of the unshifted $P$ matches $T$? What can you say about which portion matches between the unshifted $P$ and the missed $P$ as a result? Call this matching substring $\gamma$.

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

(d) What can we say about the length of $\alpha$?

(e) Combine your results from (a), (c), and (d) to derive a contradiction and complete the proof.
Problem 2: Burrows-Wheeler transform

(a) Task: Apply the Burrows-Wheeler transform to the string CATCATATG$. Show your work.

(b) Suppose you are given the following string which is the output of the Burrows-Wheeler transform: 
$TTCCTGACCGT. One of the following three options is the original string. Task: Without actually doing the inverse transform, determine which of the following is the only option that could possibly be the original string. Explain your answer.

1. ATCCTCCTGGT$
2. ATGGTCCTGGT$
3. ATCCTCCTCGT$

Problem 3: Suffix trees (OPTIONAL, WILL NOT BE GRADED)

One day Sorin was minding his own business (bioinformatics, Bob Dylan, beautiful algorithms, Beatles) when he was accosted by a magical snake. As everyone knows, snakes love suffixes. The snake wants to know all the suffixes of its favorite word, “abracadabra.” Sorin has better things to do than play mind games with a serpent, so the task has been passed on to you.

(a) Task: Construct the suffix tree for the word ABRACADABRA. You may use any algorithm that runs in quadratic time or better to build the tree, but be sure to show the first three or four steps in addition to the final tree.

(b) Task: Design an algorithm that uses your suffix tree to find the longest repeated substring of an input string in linear time, i.e. $O(n)$ where $n$ is the length of the input string. Provide a suitably detailed explanation of your answer, along with a justification of the runtime. Use your algorithm to identify the longest repeated substring of ABRACADABRA.

Problem 4: Finite automata for pattern matching

Task: Draw a deterministic finite automaton recognizing the pattern ACACG. Illustrate the computational history (i.e. the sequence of states that are visited) when the machine is run on the text ACACACGGACG.