Problem 1 (10 points).

In the KMP algorithm we defined spm_i for pattern P as the length of the longest substring of P that ends at position i, $i > 1$, and matches a prefix of P and that $P[i+1] \neq P[spm_i + 1]$. This definition does not apply to the case $i = |P|$. How should this case be defined? Justify your answer.

Problem 2 (10 points).

Prove that a suffix tree for string s has $O(|s|)$ nodes and $O(|s|)$ edges.

Problem 3 (10 points).

Prove that (1) each node in a suffix trie has a suffix link; and (2) each node in a suffix tree has a suffix link.

Problem 4 (10 points).

Design an algorithm to count the number of distinct substrings of a given string s in $O(|s|)$ time.

Solution 1. *Algorithm.* Construct the suffix tree of s using Ukkonen's algorithm. Note that in the resulting suffix tree, each edge $e \in T$ is labeled with a pair $[e_1, e_2]$, indicating the string associated with edge e is $s[e_1 \cdots e_2]$. The algorithm then traverses all edges of the suffix tree T, through either BFS or DFS, and returns the total lengths of the strings on edges, i.e., $\sum_{e \in T}(e_2 - e_1 + 1)$.

Correctness. The concatenation of letters following a path of the suffix tree T from the root to any letter of an edge gives a substring of s, and any substring of s can be spelled out following a path starting from the root. This is because the suffix tree represents all suffixes of s and any substring of s is a prefix of a suffix. Additionally, the spell-outs (contatenations of letters) of any two paths from the root must be different, as the out-edges of a node must have their first letter labeled differently. Combined, there is one-to-one correspondence between all distinct substrings of s and all paths of T from the root to some letter on an edge. Since there is a unique path from root to a letter on an edge, the total number of letters on all edges gives the number of distinct substrings of s.

Another equivalent way to show the correctness is to use suffix trie. There is a one-to-one correspondence between nodes of the suffix trie (except the root, which corresponds to the empty substring) and distinct substrings of s. Since suffix trie is also a tree, the total number of nodes minus one (the root) is exactly the total number of edges. Each edge in a suffix tree may correspond to multiple edges in the suffix tree, but the number of such edges is exactly the length of the string labeled on the edge of the suffix tree.

Running time. Constructing suffix tree costs $O(|s|)$ time; traversing the suffix tree costs $O(|s|)$ time, and it takes constant time to calculate the length of the string of an edge. Hence, this algorithm runs in $O(|s|)$ time.

Common issues found in grading. (a), explicitly construct the suffix trie—this cannot be done in $O(|s|)$ time; deducting 50% points. (b), scan the string on each edge to calculate its length, or fail to specify it; deducting 10% points. (c), miss the correctness/proof; deducting 10% points. (d), whether the empty substring or the \$ is counted–this depends on the definition; no points deducted.

Solution 2. An alternative algorithm is to use the suffix array and the LCP array. The idea is to count the number of substrings and then remove the "duplicates", i.e., when seeing them more than once. Let S be the suffix array and let L be the LCP array, both of which can be constructed in $O(|s|)$ time. We consider the suffixes of s following the suffix array, i.e., from the smallest suffix to the largest suffix. Consider the smallest suffix, i.e., $S[1]$. (Hereinafter I also use $S[1]$, which is a number, to denote the suffix, i.e., the one starting from $S[1]$ in s.) Its length is $(|s| - S[1] + 1)$ and therefore contains $(|s| - S[1] + 1)$ substrings (spelling out from the beginning of this suffix). We are seeing them for the first time so they are all counted. Now consider the next suffix, $S[2]$, which contains $(|s| - SA[2] + 1)$ substrings. How many of them have already been seen before? It is $L[1]$, i.e., the longest common prefix of suffix $S[2]$ and $S[1]$. Why? Consider a general case when we process $S[i]$, with an example where $S[i-1]$ is *aabcaa* and $S[i]$ is *aabbcaa*, hence $L[i-1] = 3$. It is clear that $L[i-1] = 3$ of the substrings of $S[i]$, namely *a*, *aa*, and *aab*, indeed appear in $S[i-1]$, which have already been considered when processing $S[i-1]$. But will *aabb* be possibly also seen before in another suffix? This is not possible, as otherwise *that* suffix will be the suffix right before $S[i]$ because suffixed are sorted in the suffix array. Combined, the total number of distinct substrings will be $|s| - S[1] + 1 + \sum_{i=2}^{|s|}(|s| - S[i] + 1 - L[i-1]) = n(n+1)/2 - \sum_{i=1}^{n-1} L[i]$. (Neat, righ?)

Problem 5 (10 points).

Given a set S of strings, design an algorithm to find every string in S that is a substring of some other string in S; your algorithm should run in $O(M)$ time, where M is the total length of all strings is S.

Solution. Assume $S = \{s_1, s_2, \cdots, s_k\}$. Build the generalized suffix tree T for S, by constructing the suffix tree for string $s = s_1\#_1 s_2\#_2 \cdots s_k\#_k$, following by keeping only one # in each leaf node. For each string s_i ($\#_i$ is not included), search it against T. Note that s_i will always be exhausted as s_i is a substring of s. If the searching ends in the middle of an edge, then the next letter of that edge must be $\#_i$ (then reaching a leaf node), as otherwise $s_i\#_i$ will not be found in T which contradicting with that $s_i\#_i$ is also a substring of s. In this case, s_i is not a substring of any other string, since $s_i\#_i$ is the only substring found in T that contains s_i. If the searching ends at a node, then this node must be an internal node, as otherwise again $s_i\#_i$ cannot be found in T. In this case, s_i is a substring of some other string in S, because the internal node has at least two out-edges, one of which must be labeled with $\#_i$ and any other out-edge indicates a substring of some other string that contains s_i. The correctness is given in the description. The running time is $O(M)$ since building the generalized suffix tree takes $O(M)$ time and searching each s_i takes $O(|s_i|)$ time and hence searching all takes $O(\sum_{i=1}^{k} |s_i|) = O(M)$ time.

Common issues found in grading. (a), for each s_i construct a generalized suffix tree T_i for $S \setminus \{s_i\}$ and search s_i against T_i; this algorithm takes $O(kM)$ where k is the number of sequences and you cannot assume that k is a constant; deducting 50% points. (b), construct the suffix trie for above s—this again cannot be done in $O(M)$ time; deducting 50% points. (c), use the same algorithm as in above solution but (implicitly) assume that the searching s_i always ends at a node; deducting 20% points.

Problem 6 (10 points).

Let T be a suffix tree for string s. Let $str(u)$ be the string represented by node u in T, i.e., the string spelled out when walking from the root of T to node u. A node in T is called left-aligned if the occurrences of $str(u)$ in s are always preceded by the same character. For example, if $s = ababacb$ then the node representing ba is left-aligned since ba is always preceded by a, but the node with $str(u) = b$ is not left-aligned as sometimes b is preceded by a and sometimes by c. Give an algorithm runs in $O(|s|)$ time to find all the left-aligned nodes in T.

Solution. Consider all occurrences of $str(u)$ in s for node u. Each occurrence correponds to a suffix of s, i.e., the substring of s from the beginning of this occurrence to the end of s. Clearly, the letter precedes each occurrence is the same letter precedes the suffix. Hence, we can decide whether u is left-aligned by examining if all its correponding suffixes are proceded by the same letter. Where are the corresponding suffixes for node u? They are the leaves in the subtree of T rooted at u.

Following above analysis, the algorithm therefore consists of two steps. In step 1, we label each leaf with the letter that precedes the corresponding suffix. One way to achieve this is to traverse the tree (in preorder) to calculate the length of the string from the root to each node, i.e., $|str(u)|$ for each node u. Once we have $|str(u)|$ for each leaf u, the letter that precedes leaf u would be $s[|s| - |str(u)|]$. (We assume s contains a \$ as its last letter.) If u represents the longest suffix (i.e., s), in which case we will have $|s| - |str(u)| = 0$, we define the letter precedes u is a unique letter #. In step 2, we traverse the suffix tree T in postorder to give each internal node a label. Note that all leaves are already labeled with a letter in step 1. For each internal node u, if all its children are labeled with the same letter, label u with the same letter and output u as we know that it is left-aligned; if not all children are labeled the same, label u with "not left-aligned"; if one of its children is labeled with "not left-aligned", also label u with "not left-aligned". Each step runs in $O(|s|)$ time since T has $O(|s|)$ nodes and edges and traversal takes linear time, hence the entire algorithm runs in $O(|s|)$ time.

Common issues found in grading. (a), for each suffix link pointing from node u to v, where $str(u)$ is x followed by $str(v)$, add x to the label-set for v; then report a node is not left-aligned if the size of its label-set is larger than 1; this approach is on the right direction but in fact not every node has incoming suffix links; deducting 30% points. (b), print the entire $str(u)$ when outputing if u is left-aligned; this may exceed $O(|s|)$; instead, you may output a pointer for each left-aligned node; no point deducted.

Problem 7 (16 points).

Let s and t be two strings; design an algorithm runs in $O(|s| + |t|)$ time to find the longest suffix of s that exactly matches a prefix of t.

1. Design such an algorithm using Z values (introduced in the Z-algorithm).
2. Design such an algorithm using suffix trie / suffix tree / generalized suffix tree.

Solution for Part 1. Construct a new string $z = t\#s$ and then build the Z array of string z. Scan the Z array starting from index $i = |t| + 2$, i.e., the s-portion: if we find $Z[i] + i - 1 = |t| + |s| + 1$, then the algorithm terminates and returns $Z[i]$, which gives the longest suffix-prefix match of s and t; if no such i can be found, then no suffix-prefix match between s and t exists. This is correct because $Z[i]$ gives the length of longest substring of z starting from index i that matches a prefix of z. The prefix of z must be a prefix of t as the match cannot exceed the unique # letter. The condition that $Z[i] + i - 1 = |t| + |s| + 1$ implies the Z-box reaches the end of z, i.e., that matching substring is also a suffix of s. The smallest/first index i corresponds to the largest $Z[i]$ as the sum of these two is a constant. Both the construction of Z array for z and the consequent scanning takes linear time.

Common issues found in grading: constructing string $z = s\#t$ which does not work; deducting 75% points.

Solution for Part 2. Build a suffix tree T for $s\$$. Search string t letter by letter against T. When being in the middle of an edge check if the next letter labeled on that edge is \$; when reaching a node check if there exists an out-edge that is just labeled \$. In either case, store the current position of t, which is also the total number of letters on the searching path from the root to the current location. When t gets exhausted or the searching gets stuck, return the largest stored number if any, or 0 otherwise. *Correctness:* above algorithm examines all prefixes of t and for each prefix the algorithm determines if it is also a suffix of s. *Running time:* constructing T takes $O(|s|)$ time and searching takes $O(|t|)$ time.

Alternative solutions include using the generalized suffix tree for s and t, or using the generalized suffix tree for s and the reverse of t.

Common issues found in grading: (a), using suffix trie, which cannot be done in $O(|s| + |t|)$ time; deducting 50% points. (b), searching t but not (or incorrectly) checking if reaching a suffix of s; deducting 50% points.