Problem 1

i

	Lookup	Insert	Delete	Space
Array	$O(\lvert w\rvert)$	$O(\lvert w\rvert + \lvert\Sigma\rvert)$	$O(\lvert w\rvert)$	$\Theta(N \cdot \lvert\Sigma\rvert)$
BST	$O(\lvert w\rvert \log \lvert\Sigma\rvert)$	$O(\lvert w\rvert \log \lvert\Sigma\rvert)$	$O(\lvert w\rvert \log \lvert\Sigma\rvert)$	$O(N)$
Hash Table	$O(\lvert w\rvert)$	$O(\lvert w\rvert)$	$O(\lvert w\rvert)$	$O(N \cdot \lvert\Sigma\rvert)$

ii In order to search the trie for $\sigma(w)$, we begin by using a standard trie lookup to find the node that represents w in the trie. The algorithm can be split into two cases: one in which w is in the trie, and when it is not.

If w is in the trie, we find it, and then look in two directions: the children of w, and the children of ancestors of w. We start by looking at children of w. Any word that has a letter represented by a child of w must be lexicographically after w due to trie properties: the children of w have w as a prefix. If w has multiple children, we choose the one that is lexicographically before the others. If w has no children we move back to the parent of w and look at its children (excluding w) in the same way as before, choosing the one that is lexicographically before the others but after the letter in w represented by how deep we are in the tree. We repeat this process, traversing up parent nodes, until we find a match.

If w is not in the trie, we go as far as we can, and then treat the node that we are on as if we were on a "parent" node in the previous scenario, looking at the children to find one that is lexicographically before the others, but after the letter in w we were looking for during the traversal. We can then continue as in the other case to find a match, traversing up the tree if necessary.

In both cases, once a match is found, traverse the tree from that node, taking the child that is lexicographically first every time until the end of a word is found. That word is the successor.

The further we traverse back up the tree towards the node, the shorter the prefix shared with w, so we are exploring the possibilities in an order such that $\sigma(w)$ will be the successor of w. If we reach the root node, check its children, and still cannot find a match, then w does not have a successor.

Since in the worst case, we can traverse at most $|w|$ nodes, the runtime of this algorithm is $O((|w| + |\sigma(w)|) \cdot |\Sigma|)$ for Array/HashMap implementations and $O((|w| + |\sigma(w)|) \log |\Sigma|)$ for the BST.

iii Proof. Let T be any weight-balanced tree with weight W. Since the weight of every node in T is positive, the root is nonzero and must have a weight. Thus, it is not possible for either the right or the left subtree to have the full weight of W. It is also not possible for them to have a weight greater than W since W is the total weight of the tree. Thus, there must exist an $0 < \epsilon < 1$ such that each subtree has weight of at most ϵW.

Among all possible weight-balanced trees, there exists an ϵ for each. The greatest ϵ among these trees satisfies the property that for any weight-balanced tree $0 < \epsilon < 1$ and the left subtree and right subtree each have a weight of at most ϵW. Thus, ϵ exists. ■

iv Proof. The lookup operation in this representation consists of walking down the trie and traversing multiple weight-balanced trees (WBT) in the process to follow the trie's edges. Since there are n words in total, the weight of the first WBT that is traversed (representing the children of the root node of the trie) is n.

We can use our result from (iii) that in any WBT, the weight of both the right subtree and the left subtree is bounded by ϵW where $0 < \epsilon < 1$. As we traverse the trie, the weights of the WBT's traversed become lower and and lower since we must choose one of the children each time and they all have nonzero weights. The weight of the child chosen each time indicates the number of strings that begin with the prefix represented by the node, so it will be the weight of the next WBT searched.

Note: In the case that the root of the WBT is chosen, we do not have a bound and thus cannot show that the next WBT will be bound by ϵW, but choosing the root requires no traversal of the WBT and can be done in constant time. The sum of the total number WBT traversals also cannot exceed $|w|$ since that is the number of trie nodes that will be traversed. Thus, these traversals are summarized as $O(|w|)$ time.

For the WBT that require traversal, the size shrinks such that the total traversal time is as follows:

$$\begin{aligned} \log n + \log \epsilon n + \log \epsilon^2 n + \cdots &= \log n + (\log \epsilon + \log n) + (2 \log \epsilon + \log n) + \cdots \\ &= k_1 \log n + k_2 = O(\log n) \end{aligned}$$

where the last step follows since ϵ is a positive constant less than 1, which means that the geometric series using a ratio equal to ϵ will converge and thus become a constant. We can then sum all the $\log n$ terms to get $O(\log n$ for lookup in the WBTs.

Including the root case mentioned above as well as the time taken to perform node operations while traversing the trie (at most $|w|$ nodes), the time becomes $O(|w| + \log n)$ ■

v The space usage of the static trie is $O(N)$ since each of the nodes in the WBTs correspond to an edge on the trie and there are N nodes on the trie. Each edge connects two nodes, making this bound possible.

Problem 2

i Because there is only one pattern string, our automaton will be in the shape of a line. (Think linked-list)

ii We need trie edges because we still need to advance character by character. We still need suffix links because suffixes can still be preserved within the same string on a mismatch. For example, if we look for "ababe" in "abababab" we will match up to index 4 and fail when 'e' is compared with 'a'. Instead of falling back to index 0, we can fall back to index 2 and preserve the suffix "ab" so we don't miss anything.

However, we don't need output links. Output links only fix the case where there are multiple pattern strings that are substrings of each other. Because we only deal with a single pattern string, this is not a problem.

iii See code.

iv Let P be the pattern string. We chose to represent the matching automaton as an array of integers A with a length 1 greater than $length(P)$. This way, each index in A corresponds to a node in the automaton. The associated character of every trie link can be accessed by indexing into P based on your current index in A. This makes moving to the next node along the trie links very easy - just increment your current index in A. The values in A correspond to suffix links. That is, the value of $A[i]$ is the index of the node pointed to by node i's suffix pointer (this also means that $A[i] < i$). Therefore, by utilizing P and A, we can access all the necessary information to perform string matching.

Problem 3

Let $LCE_{S_1,S_2}(i,j)$ be the length of the longest common prefix between any two strings S_1 and S_2 starting at positions i and j. Our solution to the k-approximate matching problem will utilize the Longest Common Extension solution present in lecture.

Our solution for the k-approximate matching problem, where T is the text string and P is the query string:

Keep an error count C. For every character position i in T, calculate $L_1 = LCE_{T,P}(i, 0)$ using the algorithm presented in lecture. If $L_1 < length(P)$, increment the error count C and calculate $L_2 = LCE_{T,P}(i + L_1 + 1, L_1 + 1)$. Note that this essentially skips the mismatched character, utilizing one of our k-allowed mismatch errors. If $C + L_1 + L_2$ is still $< length(P)$, skip the mismatched character and calculate the next LCE and so on. Repeat this process until either:
1) The total length of the extensions $(C + L_1 + L_2 + ...) >= length(P)$. Return true because we have found a appropriate solution.
2) The error count C exceeds the allowed k or we run off the end of T before $>= length(P)$ consecutive characters have been matched. Then, move on to the next character position i. If every character position i in T is evaluated without success, we return false.

Proof. We will first prove the correctness of this algorithm. Let $T[i : j]$ be the notation for the substring of T starting at index i and ending at j (for convenience, if $j > length(T)$ just stop the substring at the end of T).

For a given character index i in T, if $T[i : i + length(P)]$ is the correct match for P, then $T[i : i + length(P)]$ will match exactly with P with at most k character mismatches. This is exactly what our validation step does utilizing LCE. By definition, LCE counts the number of consecutive matching characters until a mismatch occurs. And when a mismatch occurs, we simply increment our error count and skip past the offending mismatch, rerunning LCE starting at the new index. Therefore, we are guaranteed that if $T[i : i+length(P)]$ is a correct match for P, the sum of the lengths returned by our LCE calls and the mismatches we skipped over (tallied in our error count) will be $>= length(P)$, which is our success condition.

We are also guaranteed that if $T[i : i + length(P)]$ is not a correct match of P, the error count will exceed the allowed k (or we run off the end of T) before we finish matching the entire string, which is our failure condition.

Because this verification starts at every index i in T, if a solution exists, we are guaranteed to find it. And if a solution does not exist, we will fail at every i and eventually return false. Therefore, our algorithm will yield the correct answer. ■

Proof. We will now prove the runtime of this algorithm. In order to run LCE, we must preprocess the strings T and P, which takes linear time: $O(m + n)$. However, LCE itself runs in constant time. When we run our algorithm, we look at $O(m)$ starting indexes in T, and evaluate each potential matching substring in $O(k)$. It is $O(k)$ because LCE runs in $O(1)$ time and we can run LCE a maximum of k times before we exceed the allowed error count. Therefore, running the entire algorithm will take $O(mk)$. Adding in the $O(m + n)$ preprocessing for LCE, we get a final runtime of $O(mk + n)$. ■

Problem 4

i DC2 does not always compute the suffix array in $O(m)$ since there could arise a case in which during the merging phase, two suffixes have many of the same letters but we only have relative rankings in different sets (T_0 and T_1 for example). These sets can't be compared, so we need to extend our suffixes; however, since there is an equal number of ranked T_0 and T_1 characters in DC2, every extension of the suffix will still result in two relative rankings that cannot be compared. Thus, we cannot place a bound on how much we need to extend the suffixes before we can finish the comparison. In DC3, extending the relative rankings checked will guarantee that we get relative rankings in the same domain since there is only one T_0 for two T_1 and T_2. This places an $O(1)$ bound on comparing any two suffixes in the merge phase, but does not hold for DC2.

ii DC4 works the same way as DC3 would, but with initial metacharacters that contain four characters each and an initial sort of T_1, T_2, T_3, leaving T_0 to the second phase. The sorting in both phase 1 and phase 2 work correctly regardless of the number of relative ranking sets. The merge phase is also guaranteed to find relative rankings from the same set between any two suffixes in $O(1)$ time unlike DC2. This is because there are many more in the T_1, T_2, T_3 set than there are in the T_0 one. The recurrence relation is

$$R(m) = R(3m/4) + O(m)$$

Using Master Theorem, we have $\log_b a = \log_4 1 = 0$ so $f(m) = O(m) = \Omega(m^{.5})$ where $c = .5 > 0$. We also have $O(\frac{m}{4}) \leq kO(m)$ for $k = 1$. Thus, this is a Case 3 equation, and we can conclude that $R(m) = \Theta(m)$ and that DC4 correctly computes a suffix array in $O(m)$ time.