CS 240 S2008 Final

1. A) False. MSD radix sort takes O(mnR) time, while LSD radix sort takes O(m(n+R)) time, where m is the number of digits of the largest key and R is the radix. If m or R are not constant, the runtime is not O(n).

B) False. The LZW dictionary is generated dynamically based on the text, so each code number from different pieces of text can correspond to different sequences of characters.

C) False. A Huffman code is only valid if we start at the beginning of a text, or at a break between characters. An arbitrary substring can begin at any character.

2. a) There are ceil(log n) bits in n. The algorithm iterates until there is 1 bit, and each recursive call splits the number in half by the number of bits. Hence the runtime is Theta(log n).

b) The outer for loop runs n times. The inner for loop runs 3^I times. Hence we have

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{3^i} 1 = \sum_{i=1}^{n} 3^i = \frac{3}{2}(3^n - 1) \in O(3^n)$$

3. a)

0	9690, 4090, 6070, 1020, 4070
1	
2	4372
3	1983
4	
5	
6	
7	
8	
9	

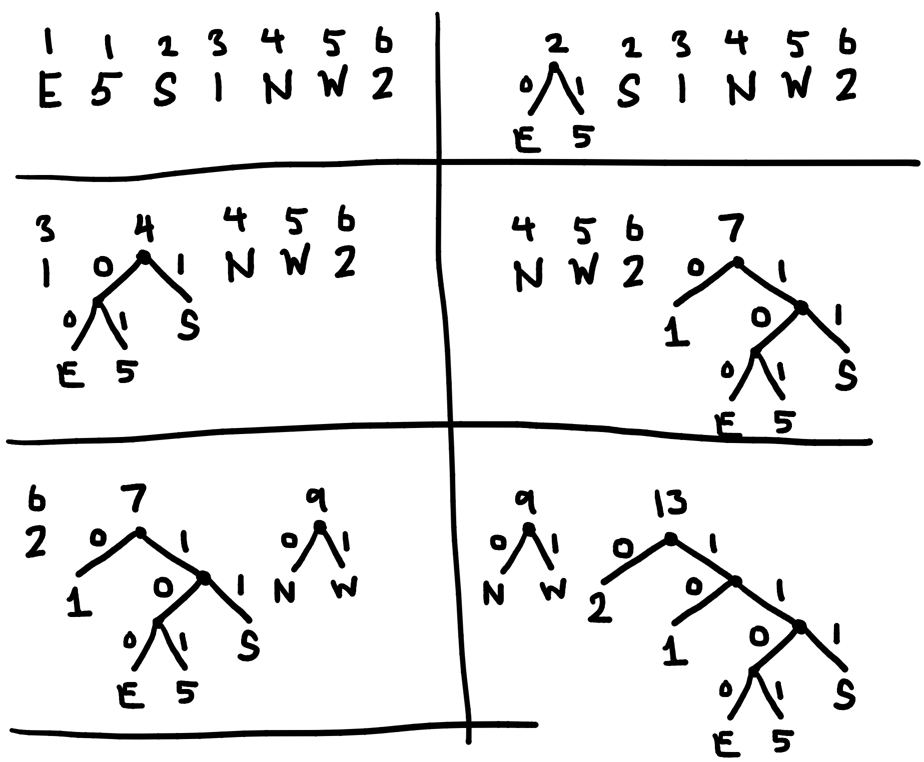
b)

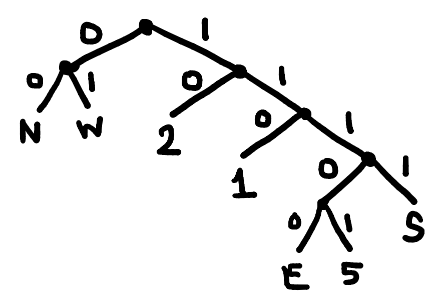
0	4070
1	1020
2	6070
3	4090
4	4372
5	9690
6	1983
7	
8	
9	

c)

0	4070
1	1020
2	6070
3	9690
4	4090
5	1983
6	4372
7	
8	
9	

4. a)





N: 00
W: 01
2: 10
1: 110
S: 1111
E: 11100
5: 11101

b) 00 01 10 00 110 01 10 00 110 10 00 01 10 01 110 1111 11101 01 10 1111 11100 10

5. an_ant_can_anti_eleplant

128: an
129: n_
130: _a
131: ant
132: t_
133: _c
134: ca
135: an_
136: _an
137: nt
138: ti
139: i_
140: _e
141: el
142: le
143: ep
144: pl

6\. BC$CCCAAB

ACBCCACB$	$ACBCCAC**B**
CBCCACB$A	ACB$ACBC**C**
BCCACB$AC	ACBCCACB**$**
CCACB$ACB	B$ACBCCA**C**
CACB$ACBC	BCCACB$A**C**
ACB$ACBCC	CACB$ACB**C**
CB$ACBCCA	CB$ACBCC**A**
B$ACBCCAC	CBCCACB$**A**
$ACBCCACB	CCACB$AC**B**

7\. A) Suffix trie (tree?). It takes O(n|Alphabet|) time to construct the trie, and O(m) time to query. Once the trie is constructed, we can query multiple different keys without needing to reconstruct the structure.

B) B-tree. It supports insertion, deletion, and fast range-searching.

C) Hash-table. It supports fast lookup by key, and can be implemented in a small amount of space since we don't need to support insertion or deletion.

D) Suffix trie (tree?). It takes O(n|Alphabet|) time to construct the trie, and O(m) time to query. Once the trie is constructed, we can query multiple different keys without needing to reconstruct the structure.

8\. Not covered :P

9\. a) We traverse the tree. At each node, we first traverse the children in lexicographical order. At each leaf, we print the suffix.

```
lexicographicalOrder(T: suffixTree) {
    if T is a leaf {
        print(T.string)
        return
    }
    for child in T.children { // Assume lexicographical order
        lexicographicalOrder(child)
    }
}
```

b) We traverse the tree until we reach an internal node with index >= length(P). If the substring differs from P by 2 characters or less, we return true. Assume we have a helper function to calculate the number of differing characters in two strings called strDiff

```
matchWithTypo(T: suffixTree, P: string) {
	if T is a leaf and T.string.length < P {
		return false
	}
	if T.string.length == P {
		if strDiff(T.string, P) <= 2 {
			return true
		}
		return false
	} else {  // T.string.length < P
		for child in T.children {
			if matchWithTypo(child, P) == false {
				return false
			}
		}
		return true
	}
}
```

c) We find the longest common string by finding the deepest common internal node between the two trees.

```
longestSubstring = ""
longestCommonString(T1, T2) {
	if (T1.string != T2.string) {
		return
	}
	If (T1.string.length > longestSubstring.length) {
		longestSubstring = T1.length
	}
	for child in T1.children {
		if (T2.children.contains(child) {
			longestCommonString(child, T2.children.find(child))
		}
	}
}
```

10. Yes. This would mean we calculate the last-occurrence array of P[0…m-2], ie P with its last character removed. Since we shift forward, we will never end up shifting anything to the last character of P, since it is at the end.