CS 240 F2021 Final Practice

1. a) False b) False c) False d) False e) True f) False g) True h) False i) N/A j) True

2. a) ii b) ii c) iv d) iii e) ii f) iv g) iv :P

3. a) p-1, p^2-1. We first insert p-1, which is hashed to slot p-1. p^2-1 is also hashed to p-1. Using double hashing, we have that h1(p^2-1) = p-1+1 = p. Hence h1(p^2-1) is p-periodic, and so h0(p^2-1) + i*h1(p^2-1) is still p-1, so we fail to insert.

   b) 0, p-1, p^2-1, p^2-p all map to 0 for h1, and map to either 0 or p-1 for h0. We have 4 keys are 3 unique hashing locations, which by the pigeonhole principle, results in a failure.

   c) 0, 1, p-1, p^2-1, p^2-p, p^2-p+1
   k = 0, h0(k) = 0, h1(k) = 0
   k = 1, h0(k) = 1, h1(k) = 0
   k = p − 1, h0(k) = p − 1, h1(k) = 0
   k = p 2 − 1, h0(k) = p − 1, h1(k) = p − 1
   k = p 2 − p, h0(k) = 0, h1(k) = p − 1
   k = p 2 − p + 1, h0(k) = 1, h1(k) = p − 1

   We have 6 keys and 5 unique hashing locations, which by the pigeonhole principle, results in a failure.

4. a) 001 000 011 1010 1011 100 010 11 010 010
   pswd: lull

   b) u has the shortest code, but is not the most frequent character in the string.

5. a) No. Counterexample: P = aab, T = aaab. The first comparison is a mismatch, so we shift so that the mismatched character aligns with the first b in P. This causes us to shift past the text, returning failure when P is actually in T.

   b) Yes. If there is a mismatch, we still use the last-occurrence function to determine the shift, so we don't skip any potentially correct pattern shifts. The shift does not depend on which direction we scanned the pattern with.

   c) Yes. Since we shift forward, we will never end up shifting anything to the last character of P, since it is at the end.

6. a) The maximum height of T is n. This would occur if for example, we stored n strings all consisting of j repetitions of the same character, where j ranges from 1 to n.

b) The maximum height of T is 1. This would occur if for example, we stored n strings each differing in their first bit. Then we would only have the root, which would have n leaves.

7. a)

T	G	C	C	G	A	T	G	T	A	G	C	T	A	G	C	A	T
T	A																
	T																
		T															
								T	A	G	C	A					
										T							
												T	A	G	C	A	T

b) $h = ((h - T[i] \cdot s) \cdot 10 + T[i+m]) \bmod M$

c) Some T where all its substrings hash to the same hash value as P but are not equal to P.

8. Nope

9. We can stably sort T' in O(n) time using radix sort. The suffix array As of T is the sorting permutation of T. Hence we can iterate over each character in the sorted T' with index i, and we know that character is in the index As[i] of T. This is also O(n) time.

10. a) (0011)^(n/4) and (1100)^(n/4) have the worst compression ratio. There are n/2 runs, and each one compresses to 010. Hence the compression ratio is

$$\frac{\frac{3n}{2}+1}{n} = \frac{3}{2} + \frac{1}{n}$$

b) 0^n or 1^n have the best compression ratio. There is one run of length n, so the compressed string becomes 0 or 1, plus floor(logn) 0s and floor(logn) + 1 1s. Hence the compression ratio is

$$\frac{2\lfloor \log n \rfloor + 2}{n}$$

11. We range search the trie with query [b1, b2]. Assume, without loss of generality, that b1 < b2. We find the search path P1 for b1, and the search path P2 for b2. We have that a node is an inside node if it is to the right of P1 and to the left of P2. The inside nodes whose parents are boundary nodes are top inside nodes. If there exists a top inside node, return False; otherwise return True.

If b1 and b2 are consecutive, then the range-search would have no other keys in the range aside

from b1 and b2, so there would be no inside nodes. If b1 and b2 are not consecutive, then there exists at least one other key, b3, which lies between b1 and b2. All boundary nodes are internal nodes except for the two nodes at the ends of the paths P1 and P2, which correspond to the leaves containing b1 and b2. Since b3 is in the lexicographic range, it must be a leaf as an inside node.

We analyze the runtime. The boundary involving b1 has at most |b1| nodes, while the boundary involving b2 has at most |b2| nodes. It takes constant time to check for the existence of top inside nodes. Hence the runtime is O(|b1| + |b2|).

12. a) In the best case, we have the string s = a^n. In the ith iteration, we insert the string of length i into the table, and can use the code for the (i-1)-length string. Hence in the first w-1 iterations, we encode sum i (i=1 to w-1) characters. We have that

$$\sum_{i=1}^{w-1} i = \frac{w(w-1)}{2} < n \rightarrow w \in O(\sqrt{n})$$

In the (w-1)th iteration, we have n - <the sum above> characters left to encode. We add a string of length w to our dictionary. Since this is the second-last iteration, we must have w or less characters left to encode. Hence we have

$$1 \leq n - \sum_{i=1}^{w-1} i \leq w$$

$$n \leq \sum_{i=1}^{w-1} i + w = \sum_{i=1}^{w} i = \frac{w(w+1)}{2} \rightarrow w \in \Omega(\sqrt{n})$$

Hence we have

$$w \in \Theta(\sqrt{n})$$

b) The two consecutive code numbers 00100001 01000000 are repeated twice in the encoded string, which is impossible in LZW since they would have been added to the dictionary the first time, so would not appear a second time.

13. a) $AH^k\$A^k$

b) $A^tH^k\$$

c) $HA^{k-1}\$H^{t-1}A$

$AAAAAHHH
AHHH$AAAA
AAHHH$AAA
AAAHHH$AA
AAAAHHH$A
AAAAAHHH$
H$AAAAAHH
HH$AAAAAH
HHH$AAAAA

14. Not covered :P

15. We construct a BST with the key being the index in the array, and each node stores the value at that index, the minimum value in its subtree, and the maximum value in its subtree.

To range query, we BST range-search for the maximum value in the range of indices [i,j]. We do the same for the minimum, and return the difference between the two. Each range query takes O(log n) time. The BST takes O(n) space.

```
maxDiff(i, j) {
	return rangeMax(i, j) - rangeMin(i, j)
}
rangeMax(i, j) {
	P1 = boundary path to i
	P2 = boundary path to j
	Ins = topmost inside nodes
	return max(values in P1, values in P2, subtreeMax values in Ins)
}
rangeMin(i, j) {
	P1 = boundary path to i
	P2 = boundary path to j
	Ins = topmost inside nodes
	return max(values in P1, values in P2, subtreeMin values in Ins)
}
```