1 (text) Recurrence [10 points] Solve the following recurrence relation using repeated substitution.

$$T(n) = 3T(n/4) + 4n$$

Then solve it by using the Master Method, showing in detail which rule applies

Note: Show your work. You will get 4 points if you identify the pattern, 3 points if you do the proof work necessary to show what it resolves to, and 3 points for solving it using the master theorem.

We have $T(n) = 3T(n/4) + 4n$

$\frac{\frac{n}{4}}{4} = \frac{n}{16}$

Substitute $T(n/4)$ to get: $T(n/4) = 3T(\frac{n/4}{4}) + \frac{4n}{4}$

$= 3T(n/16) + \frac{4n}{4}$

$T(n) = 3[3T(\frac{n}{16}) + \frac{4n}{4}) + 4n$

$= 9T(n/16) + 3n + 4n$

Next, $T(n/16) = 3T(\frac{n/16}{4}) + \frac{4n}{16}$

$= 3T(n/64) + \frac{4n}{16}$

$T(n)$ then be: $9[3T(n/64) + n/4] + 3n + 4n$

The pattern becomes $T(n) \to 3^K T(n/4^K) + \sum_{i=0}^{K-1} \left(\frac{3}{4}\right)^i \cdot 4n$

Geometric Series $\to S = \frac{1 - r^K}{1 - r}$

$S = \frac{1 - (3/4)^K}{1 - 3/4} = \boxed{4\left(1 - \left(\frac{3}{4}\right)^K\right)}$

$T(n) = 3^K T(n/4^K) + 4n \cdot \left[1 - \left(\frac{3}{4}\right)^K\right]$ Substitute this back $T(n)$

As n goes to $\infty \to n/4^K = 0 \to 3^K \cdot 0 = 0$

Now we only have $4n \cdot \left[1 - \left(\frac{3}{4}\right)^K\right]$

Now this becomes $O(n)$ because as n is our input, as n increase, the time also increase.

$\Rightarrow T(n) = O(n)$

Master Thm: $T(n) = aT(n/b) + f(n)$

If $f(n) = \Theta(n^d)$, where $d \geq 0$, then

1. $T(n) = \Theta(n^d)$ if $a < b^d$
2. $T(n) = \Theta(n^d \log n)$ if $a = b^d$
3. $T(n) = \Theta(n^{\log_b a})$ if $a > b^d$

$T(n) = 3T(n/4) + 4n$

$a = 3$

$b = 4 \Rightarrow a < b^d \Rightarrow 3 < 4^1 \Rightarrow$ case 1:

$d = 1$

$\boxed{\Theta(n^1)}$

2 (text) Master Theorem [20 points] Apply the Master method to solve each of the following recurrences, or state that the Master method does not apply. Justify your answers. Note that the Master method covers all the cases

a. $T(n) = 3T(\frac{n}{5}) + n^2$

b. $T(n) = 4T(\frac{n}{3}) + 7n$

c. $T(n) = 5T(\frac{n}{4}) + 10$

d. $T(n) = 9T(\frac{n}{3}) + n^4$

e. $T(n) = 6T(\frac{n}{8}) + n^3$

Master Thm: $T(n) = aT(n/b) + f(n)$

If $f(n) = \Theta(n^d)$, where $d \geq 0$, then

1. $T(n) = \Theta(n^d)$ if $a < b^d$
2. $T(n) = \Theta(n^d \log n)$ if $a = b^d$
3. $T(n) = \Theta(n^{\log_b a})$ if $a > b^d$

a. $T(n) = 3T(\frac{n}{5}) + n^2$

$a = 3, b = 5, d = 2$

$b^d = 5^2 > 3 \Rightarrow$ Case 1: $a < b^d \Rightarrow \Theta(n^2)$

b. $4T(n/3) + 7n$

$a = 4, b = 3, d = 1$

$b^d = 3^1 < 4 \Rightarrow$ Case 3: $a > b^d \Rightarrow \Theta(n^{\log_3 4})$

c. $T(n) = 5T(n/4) + 10$

$a = 5, b = 4, d = 0$

$b^d = 4^0 = 1 < 5 \Rightarrow$ Case 3: $a > b^d \Rightarrow \Theta(n^{\log_4 5})$

d. $T(n) = 9T(n/3) + n^4$

$a = 9, b = 3, d = 4$

$b^d = 3^4 = 81 > 9 \Rightarrow$ Case 1: $a < b^d \Rightarrow \Theta(n^4)$

e. $6T(n/8) + n^3$

$a = 6, b = 8, d = 3$

$b^d = 8^3 = 512 > 6 \Rightarrow$ Case 1: $a < b^d \Rightarrow \Theta(n^3)$

3 (text) Radix Sort [10 points] Lexicographical ordering means order of the dictionaries to sequences of ordered symbols therefore (a < b < c < d < e < f < … < m < n < o < … < y < z). The same logic applies to Uppercase letters.

Illustrate the operation of Radix-Sort on the following list of strings using lexicographic ordering.

CAP, COL, USD, SUN, JPY, VEE, ROW, JOB, COX, LOL, RAT, WOW, DOD, CAR, FIG, PIG, VIS, LOW, LOX, VEA, CAD, DOG, TSL

1st (3rd char)

A: VEA
B: JOB
C: N/A
D: USD, DOD, CAD
E: VEE
F:
G: FIG, PIG, DOG
H:
I:
J:
K:
L: COL, LOL, TSL
M:
N: SUN
O:
P: CAP
Q:
R: CAR
S: VIS
T: RAT
U:
V:
W: ROW, WOW, LOW
X: COX, LOX
Y: JPY
Z:

2nd (2nd char)

A: CAD, CAP, CAR, RAT
B:
C:
D:
E: VEA, VEE
F:
G:
H:
I: FIG, PIG, VIS
J:
K:
L:
M:
N:
O: JOB, DOD, DOG, COL, LOL, ROW, WOW, LOW, COX, LOX
P: JPY
Q:
R:
S: USD, TSL
T: N/A
U. SUN
V:
W:
X:
Y:
Z:

3rd (1st char)

A:
B:
C: CAD, CAP, CAR, COL, COX
D: DOD, DOG,
E:
F: FIG
G:
H:
I:
J: JOB, JPY
K:
L: LOL, LOW, LOX
M:
N:
O:
P: PIG
Q:
R: RAT, ROW
S: SUN
T. TSL
U. USD
V: VEA, VEE, VIS
W: WOW
X:
Y:
Z:

After 3 iterations, the sorted array is:

[CAD, CAP, CAR, COL, COX, DOD, DOG, FIG, JOB, JPY, LOL, LOW, LOX, PIG, RAT, ROW, SUN, TSL, USD, VEA, VEE, VIS, WOW]

4 (text) Double Hashing [15 points] Consider a hash table consisting of M = 13 slots, and suppose nonnegative integer key values are hashed into the table using the hash function h1() and that collisions are resolved by using double hashing with the secondary hash function Reverse(value), which reverses the digits of v and returns that value; for example, Reverse(3652) = 2563.

```
int h1 (int key) {
    int x = (key + 19) * (key + 11);
    x = x / 15;
    x = x + key;
    x = x % M;
    return x;
}
```

Add the following items [25, 14, 9, 7, 5, 3, 0, 21, 6, 33, 25, 42, 24, 107] to the HashTable in order.

For each key being inserted to the HashTable, show:
(1) The home slot (the initial hashed slot)
(2) The number of collisions and the probe sequence (if collisions occur)
(3) The final contents of the hash table

Hint: You will have to re-size and rehash once

$(25+19)*(25+11) = 1584/15 = 105.6 + 25 = 130.6 \% 13 = 0$

$(14+19)*(14+11) = 825/15 = 55 + 14 = 69 \% 13 = 4$

$(9+19)*(9+11) = 560/15 = 37.3 + 9 = 46.3 \% 13 = 7$

$(7+19)*(7+11) = 468/15 = 31.2 + 7 = 38.2 \% 13 = 12$

$(5+19)*(5+11) = 384/15 = 25.6 + 5 = 30.6 \% 13 = 4$ Collision

$(3+19)*(3+11) = 308/15 = 20.53 + 3 = 23.53 \% 13 = 10$

$(0+19)*(0+11) = 209/15 = 13.93 + 0 = 13.93 \% 13 = 0$ Collision infinite

$(21+19)*(21+11) = 1280/15 = 85.3 + 21 = 106.3 \% 13 = 2$

$(6+19)*(6+11) = 425/15 = 28.3 + 6 = 34.3 \% 13 = 8$

$(33+19)*(33+11) = 2288/15 = 152.53 + 33 = 185.53 \% 13 = 3$

$(25+19)*(25+11) = 1584/15 = 105.6 + 25 = 130.6 \% 13 = 0$ Collision infinite

$(42+19)*(42+11) = 3233/15 = 215.53 + 42 = 257.53 \% 13 = 10$ Collision

$(24+19)*(24+11) = 1505/15 = 100.3 + 24 = 124.3 \% 13 = 7$ Collision → $(7 + 1 \cdot \text{Reverse}(24)) \% 13 = 3$
$(7 + 2 \cdot \text{Reverse}(24)) \% 13 = 0$
$(7 + 3 \cdot 42) \% 13 = 1$ ✓

$(107+19)*(107+11) = 14868/15 = 991.2 + 107 = 1098.2 \% 13 = 6$

0: 25
1: 24
2: 21
3: 33
4: 14
5:
6: 107
7: 9
8: 6
9: 5
10: 3
11: 42
12: 7

Hash Table

$12/14 = 0.85 > 0.7$

Resize

Collision at 5 ⇒ Home slot: 4 ⇒ $(4 + 1 \cdot \text{Reverse}(5)) \% 13 = 9$

Collision at 0 ⇒ Home slot: 0 ⇒ $(0 + 1 \cdot \text{Reverse}(0)) \% 13 = 0$ Collision
0 ⇒ Home slot: 0 ⇒ $(0 + 2 \cdot \text{Reverse}(0)) \% 13 = 0$ Collision
: 0 ⇒ $(0 + 3 \cdot \text{Reverse}(0)) \% 13 = 0$ Collision
: 0 ⇒ $(0 + 4 \cdot \text{Reverse}(0)) \% 13 = 0$ Collision
} ∞ infinite probing

Collision at 25 ⇒ Home slot: 0 ⇒ $(0 + 1 \cdot \text{Reverse}(25)) \% 13 = 0$ Collision, Probe = 2
Collision at 0 ⇒ Home slot: 0 ⇒ $(0 + 2 \cdot \text{Reverse}^{52}(25)) \% 13 = 0$ Collision, Probe = 3
Collision at 0 ⇒ Home slot: 0 ⇒ $(0 + 3 \cdot \text{Reverse}(25)) \% 13 = 0$ Collision, Probe = 4
} ∞ infinite Prob

Collision at 42 : Home slot : 10 $\Rightarrow$ (10 + 1 · Reverse(42)) % 13 = 8 Collision, Probe = 2

Collision at 42 $\Rightarrow$ (10 + 2 · Reverse(42)) % 13 = 6 Collision, Probe = 3

Collision at 42 : $\Rightarrow$ (10 + 3 · Reverse(42)) % 13 = 4 Collision, Probe = 4

$\Rightarrow$ (10 + 4 · 24) % 13 = 2 Collision, Probe = 5

$\Rightarrow$ (10 + 5 · 24) % 13 = 0 Collision, Probe = 6

$\Rightarrow$ (10 + 6 · 24) % 13 = 11 ✓

Since our hash table is too full $\Rightarrow$ resize

$2 \times 13 = 26 \rightarrow$ The next prime numbe = 29

Key	Rehash h_1(Key) mod 29	New home slot
25	$h_1(25) \% 29$	14
14	$h_1(14) \% 29$	11
9	$h_1(9) \% 29$	17
7	$h_1(7) \% 29$	9
5	$h_1(5) \% 29$	1
3	$h_1(3) \% 29$	23
0	$h_1(0) \% 29$	13
21	$h_1(21) \% 29$	19
6	$h_1(6) \% 29$	5
33	$h_1(33) \% 29$	11
25	~	14
42	~	25
24	~	6
107	~	25

Collision $\Rightarrow$ (14 + 1 · 52) % 29 = 8 ✓

Collision $\Rightarrow$ (8 + 1 · 42) % 29 = 17

$\Rightarrow$ (8 + 2 · 42) % 29 = 5

$\Rightarrow$ (8 + 3 · 42) % 29 = 18 ✓

7 (text) Algorithm Analysis [5 points]

For each of the algorithms you wrote for problems 4-6, explain their time complexity and space complexity using Big-O notation. Explain how you arrived at your answer.

```java
public void radixSort(String[] arr) {  3 usages  Peter
    int maxLength = getMaxStringLength(arr);
    for (int pos = maxLength - 1; pos >= 0; pos--) {
        countSort(arr, pos);
    }
}
```

O(m)

Time: $O(m \cdot n)$ where m is the length of the largest word and n is the # string in the array.

The radix sort loops over the length of the largest string then in the body it runs countSort m-times

→ Therefore, $O(m \cdot n)$

```java
private static void countSort(String[] arr, int charIndex) {  1 usage  Peter
    int n = arr.length;
    Map<Integer, List<String>> buckets = new HashMap<>();

    // 53 buckets: 1-26 for A - Z, 27-52 for a-z, 0 for short words.
    for (int i = 0; i < 53; i++) {
        buckets.put(i, new ArrayList<>());
    }

    for (String s : arr) {
        int bucketIndex;
        if (charIndex < s.length()) {
            char ch = s.charAt(charIndex);
            if (Character.isUpperCase(ch)) {
                bucketIndex = ch - 'A' + 1;
            } else if (Character.isLowerCase(ch)) {
                bucketIndex = ch - 'a' + 27;
            } else {
                bucketIndex = 0;
            }
        } else {
            bucketIndex = 0; //this bucket handles the short words.
        }
        buckets.get(bucketIndex).add(s); //add
    }

    int index = 0;
    for (int i = 0; i < 53; i++) {
        List<String> bucket = buckets.get(i);
        for (String word : bucket) {
            arr[index++] = word;
        }
    }
}
```

Space: $O(n)$ where n is the # of strings in the input array. This is because we store strings in buckets of the hash map. Each bucket contains a list of string, worst case, all n-strings will be store into the same bucket.

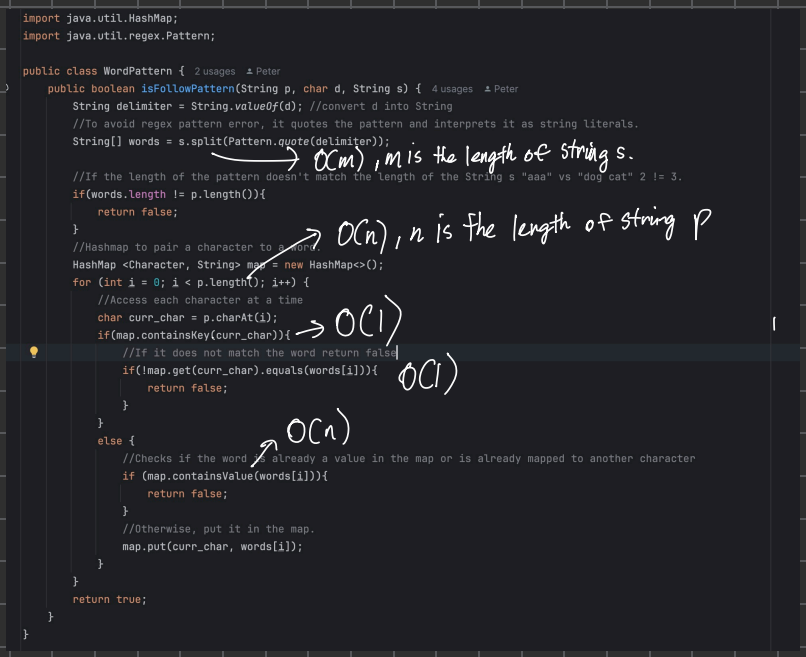
```java
private static int getMaxStringLength(String[] arr) {  1 usage  Peter
    int maxLength = 0;
    for (String s : arr) {
        if (s.length() > maxLength) {
            maxLength = s.length();
        }
    }
    return maxLength;
}
```

O(1)

O(1)

Space Complexity is O(1) since the code doesn't make variables the input size increase.

5.



```java
import java.util.HashMap;
import java.util.regex.Pattern;

public class WordPattern {  2 usages  Peter
    public boolean isFollowPattern(String p, char d, String s) {  4 usages  Peter
        String delimiter = String.valueOf(d); //convert d into String
        //To avoid regex pattern error, it quotes the pattern and interprets it as string literals.
        String[] words = s.split(Pattern.quote(delimiter));

        //If the length of the pattern doesn't match the length of the String s "aaa" vs "dog cat" 2 != 3.
        if(words.length != p.length()){
            return false;
        }
        //Hashmap to pair a character to a word.
        HashMap <Character, String> map = new HashMap<>();
        for (int i = 0; i < p.length(); i++) {
            //Access each character at a time
            char curr_char = p.charAt(i);
            if(map.containsKey(curr_char)){
                //If it does not match the word return false
                if(!map.get(curr_char).equals(words[i])){
                    return false;
                }
            }
            else {
                //Checks if the word is already a value in the map or is already mapped to another character
                if (map.containsValue(words[i])){
                    return false;
                }
                //Otherwise, put it in the map.
                map.put(curr_char, words[i]);
            }
        }
        return true;
    }
}
```

O(m), m is the length of string s.

O(n), n is the length of string p

O(1)

O(1)

O(n)

Time Complexity: $O(m+n^2)$ because the method .split the delimiter, in worst case the entire string s needed to scanned to know where the delimiter happen.

First for loop runs n-times, where n is length of string p. Then the containValue takes n-times to scan the entire map.

Space: O(n), where n is the length of string p because the map stores each unique character from pattern p. Which means it can store up to n-key-value pairs.

4. Double hashing

Time: O(n), where n is the # of keys to hash

Space: O(n), where n is the # of keys because it stores in HashTable.