1a. f() is O(n)
g() is O(n)
1b. f() space complexity is O(1)
g() space complexity is O(n)

1c.
Int h (int n){
return n;}
This is O(1)

2 . f() is O(log(n)) and the loop in g() is O(log(n))
so g() is O(log^2(n))

3.

```
int kfunction(int n)
{
    std::string temp;
    char k = 1;
    int array[10] = {};
    bool complete = false;
    int tempint = n;


    while (!complete)
    {
        complete = true;
        temp = std::to_string(tempint);
        for (int i = 0; i<temp.length(); i++)
        {
            if (array[(int)temp.at(i)-48] != 1)
                array[(int)temp.at(i)-48] = 1;
        }
        for (int j = 0 ;j < 10; j++)
        {
            if (array[j] == 0)
            {
                complete = false;
                break;
            }
        }
        tempint += n;
        if (complete)
            return k;
        k++;
    }
    temp = std::to_string(n);
    return k;
}
```

This function is O(n) when we take into consideration the length of the string. We must loop through the length of the string to check each digit in the number.

4a. mod by 2 and check the remainder which is O(1)
4b. Assuming list isn't sorted or has any order, we need to go through the list which is O(n)
4c. Assuming list isn't sorted or has any order, we need to go through the list which is O(n)
4d. Two nested for loops, each for going through one of the lists. Since each for loop depends on n, then the algorithm would be O(n^2) if we looped through to compare every possible combination
4e. No duplicates with same values indicates same length. Since they are sorted, we can use one loop to iterate through both lists and compare as we go. Thus, the algorithm is O(n)
4f. As we go down a BST subtree, we cut the possibilities in half each time indicating that the algorithm is O(log(n))

5.
For(I = 0 - > string.length()) {
Counting sort s1
Counting sort s2
}
For(I = 0 -> 26){
Check counting sort array s1
Check counting sort array s2
}
Return true if arrays equal