DSC 40B - Hashing

Problem 1.

Suppose a hash table with 1000 bins stores 2000 numbers. Collisions are resolved with chaining.

True or False: it is possible for one of the hash table's bins to contain zero elements.

Solution: True, imagine the case where all the numbers hash to the same bin.

Problem 2.

Suppose a hash table is implemented so that it has ten bins, and that this number of bins is not increased when new elements are inserted; that is, the hash table is not allowed to "grow" or "resize". Let n be the number of elements stored in the hash table.

What is the expected time complexity of querying an element in this hash table, as a function of n? You may assume that collisions are resolved with chaining, that the bins are linked lists, and that the hash function is "good" in that it appears to uniformly distribution elements among bins.

Solution: $\theta(n)$.
Given n elements, the expected number of elements in each bin is $n/10$ due to the assumption that the hash table doesn't resize and hash table hashes uniformly across the ten buckets. Therefore, the expected time complexity of querying an element is going to be cost of looking up the index + linear searching through the linked list, which takes $\theta(1) + \theta(n/10) = \theta(n)$.

Problem 3.

What is the expected time complexity of the code below, assuming that *numbers* is a Python *set* of size n? Recall that Python's sets are implemented as hash tables.

```python
def foo(numbers):
    count = 0
    for x in numbers:
        if -x in numbers:
            count += 1
    return count
```

Solution: $\theta(n)$
Checking if an element is in the set using the Python **in** operator takes time linear in the size of the set. Therefore, the if statement in the code takes expected $\theta(1)$ time. The for loop executes for $\theta(n)$ iterations. Hence, the total time complexity is $\theta(n)$.

Problem 4.

Recall that a *mode* of a collection is an element which occurs with the greatest frequency. For example, 4 is a mode of the collection $4, 5, 8, 3, 4, 2, 4, 5, 5, -2$. 5 is also a mode, since it occurs just as frequently as 4.

Describe an algorithm that finds the mode of a collection of numbers.

Solution: We can create an empty dictionary to keep track of the count of each number.

The keys are the numbers and the values are the count of each number.

We will then loop through the list of numbers and update the count of each number in the dictionary.

Finally, we will return the number with the highest count by looping through the items of the dictionary.

This algorithm has a expected time complexity of $\theta(n)$.

```
def mode(numbers: List[int]) -> int:
    counts = {}
    for i in numbers:
        if i not in counts:
            counts[i] = 0

        counts[i] += 1

    return max(counts, key=lambda k: counts[k])
```

Problem 5.

Given two strings `a` and `b`, determine if they are isomorphic.

Two strings `a` and `b` are isomorphic if the characters in `a` can be replaced to get `b`. They must also have the same length.

Eg. ”foo” and ”bar” are not isomorphic, but ”egg” and ”add” are. ”badc” and ”baba” are not isomorphic.

Describe an algorithm that determines if two strings are isomorphic.

Essentially, can the characters in `a` be mapped to the characters in `b` such that the characters in `a` can be replaced to get b.

Solution: The idea is that if too strings are isomorphic, then the characters in the first string can be mapped to the characters in the second string. And a character can only be mapped to one character.

We can create two dictionaries, `a2b` and `b2a` to keep track of the mapping of characters from the first string to the second string and vice versa. The keys of `a2b` are characters of `a`, and values are characters of `b`. The keys of `b2a` are characters of `b`, and values are characters of `a`.

We will loop through the characters of the strings simultaneously (they are the same length), and create a mapping for `a[i]` to `b[i]` and `b[i]` to `a[i]`.

During each iteration, whenever we see that an existing character already has a mapping, we will check if the existing mapping is the same as the current mapping to be assigned. If it is not, then the strings are not isomorphic.

If no mapping conflicts arise, then the strings are isomorphic.

```
def is_isomorphic(a: str, b: str) -> bool:
    a2b = {}
    b2a = {}
    N = len(a)
    M = len(b)

    # if they are different length isomophism is never possible
    if N != M:
        return False
    # for each lettter in A, map it to B, and vise versa
    for i in range(N):

        mapping_exists_for_a = a[i] in a2b
        mapping_exists_for_b = b[i] in b2a

        mapping_conflict_for_a = mapping_exists_for_a and a2b[a[i]] != b[i]
        mapping_conflict_for_b = mapping_exists_for_b and b2a[b[i]] != a[i]
        # if a mappping exists and they don't point to each other, it's doomed
        if mapping_conflict_for_a or mapping_conflict_for_b:
            return False

        a2b[a[i]] = b[i]
        b2a[b[i]]= a[i]

    return True
```