Counting

Counting (keeping track of the frequency of things) is a very common pattern with hash maps. This means our hash maps will be mapping keys to integers. Anytime you need to count anything, think about using a hash map to do it.

With the sliding window problems we've done so far, if we've had to say limit the window to having at most `k` `0`s, we got away with using one variable as a counter. But now with hash maps we can keep track of any amount of things.

Example 1: Given a string `s` and an integer `k`, find the length of the longest substring that contains **at most** `k` distinct characters.

For example, given `s = "eceba"` and `k = 2`, return `3`. The longest substring with at most 2 distinct characters is "ece".

Since this problem deals with a constraint on substrings, we can use a sliding window. The brute force way to check for this constraint would be to check the entire window every time, which would take $O(n)$ time. Using a hash map, we can check the constraint in $O(1)$.

In Python, the `collections` module provides very useful data structures. `defaultdict` behaves like a hash map, but if for instance we are counting letters, you don't have to worry about initializing the counts for characters not seen yet.

```
from collections import defaultdict

def find_longest_substring(s, k):
    counts = defaultdict(int)
    left = ans = 0
    for right in range(len(s)):
        counts[s[right]] += 1
        while len(counts) > k:
            counts[s[left]] -= 1
            if counts[s[left]] == 0:
                del counts[s[left]]
            left += 1

        ans = max(ans, right - left + 1)
```

```
    return ans
```

Using a hash map to store the frequency of any key we want allows us to solve sliding window problems that put constraints on multiple elements. The hash map takes $O(k)$ space since the algorithm will delete elements from the hash map once it grows beyond *k*.

Example 2: 2248 - Intersection of Multiple Arrays

Given a 2D array `nums` that contains `n` arrays of distinct integers, return a sorted array containing all the numbers that appear in all `n` arrays.

For example, given `nums = [[3, 1, 2, 4, 5], [1, 2, 3, 4], [3, 4, 5, 6]]`, return `[3, 4]`. `3` and `4` are the only numbers that are in all arrays.

The problem states that each individual array contains **distinct** integers. This means that a number appears `n` times if and only if it appears in all arrays.

```
from collections import defaultdict

class Solution:
    def intersection(self, nums: List[List[int]]) -> List[int]:
        counts = defaultdict(int)
        for arr in nums:
            for x in arr:
                counts[x] += 1

        n = len(nums)
        ans = []
        for key in counts:
            if counts[key] == n:
                ans.append(key)

        return sorted(ans)
```

You may be thinking, since our keys are integers, why can't we just use an array instead of a hash map? Well the array would need to be as least as large as the max element. `[1, 2, 3, 1000]` would waste a lot of space. The array might be slightly more efficient because of the overhead of a hash map, but a hash map is much safer. Even if `99999999999` is the input, it doesn't matter - the hash map handles it like any other element.

Let's say that there are n lists and each list has an average of m elements. To populate our hash map, it costs $O(nm)$ to iterate over all the elements. Then, there can be at most m elements inside `ans` when we perform the sort (because can't be larger than the minimum size row and m is greater than or equal to the minimum size of a row), which means in the worst case, the sort will cost $O(mlog(m))$. Adding with the term before, our time complexity is $O(m(n + log(m)))$. If every element in the input is unique, then the hash map will grow to a size of nm, which means the algorithm has a space complexity of $O(nm)$.

Example 3: 1941 - Check if All Characters Have Equal Number of Occurrences

Given a string `s`, determine if all characters have the same frequency.

After getting the counts of each character, since a set ignores duplicates, we can put all the frequencies in a set and check if the length is `1` to verify if the frequencies are all the same.

```
from collections import defaultdict

class Solution:
    def areOccurrencesEqual(self, s: str) -> bool:
        counts = defaultdict(int)
        for c in s:
            counts[c] += 1

        frequencies = counts.values()
        return len(set(frequencies)) == 1
```

Given n as the length of `s`, it costs $O(n)$ to populate the hash map, then $O(n)$ to convert the hash map's values to a set. In total, the time complexity is $O(n)$. If the number of unique characters is k, then the space complexity is the $O(k)$, the space that the hash map and set take up.

Here is a one-liner solution to the problem:

```
from collections import Counter

class Solution:
```

```
    def areOccurrencesEqual(self, s: str) -> bool:
        return len(set(Counter(s).values())) == 1
```

Count the number of subarrays with an "exact" constraint

In the sliding window section from chapter 1, we talked about a pattern "find the number of subarrays/substrings that fit a constraint". In those problems, if you had a window between `left` and `right` that fits the constraint, then all windows from `x` to `right` also fit the constraint, where `left < x <= right` and the constraint was something like:

"Find the number of subarrays that have a sum less than k" with an input of **only positive numbers**.

Here we will discuss problems of the form:
"Find the number of subarrays that have a sum **exactly equal** to `k`"

Recall the concept of prefix sums.

Given a prefix sum array that we calculate in $O(n)$, any difference in the prefix sum equal to `k` represents a subarray with a sum equal to `k`. So how do we find these differences?

First declare a hash map `counts` that maps prefix sums to how often they occur (a number could appear multiple times in a prefix sum if the input has negative numbers; for example, given `nums = [1, -1, 1]`, the prefix sum is `[1, 0, 1]` and `1` appears twice). We need to initialize `counts[0] = 1` because the empty prefix `[]` has a sum of `0`. We'll see why this is necessary in a second.

We now declare our `answer` variable and `curr`. `curr` keeps track of the current prefix sum.

If `curr` is the current prefix sum and we need to figure out how many subarrays **ended** at the current index, we need a previous prefix sum to have been `curr - k` since we need a difference in the prefix sums to be `k` (`curr - (curr - k) == k`). If we keep track of the frequency of `curr` as we go, we can check how many times we saw `curr - k` before (when `curr - k` was `curr` at a previous stage) in $O(1)$ with a hash table `counts`. It can occur more than once if there are negatives. After incrementing the `answer` by the amount of `curr - k`s, we can add the present `curr` to `counts`.

Example 4: 560 - Subarray Sum Equals K

Given an integer array `nums` and an integer `k`, find the number of subarrays whose sum is equal to `k`.

Let's go through an example to see why the algorithm above works. Say `nums = [1, 2, 1, 2, 1], k = 3`. There are four subarrays with sum `3` - `[1, 2]` twice and `[2, 1]` twice.

The prefix sum, which is what `curr` represents during iteration, is `[1, 3, 4, 6, 7]`. There are three differences in this array equal to `3`: `(4 - 1)`, `(6 - 3)`, `(7 - 4)`.

But there are four valid subarrays. This is why we needed to initialize our hash map with `0: 1`, considering the empty prefix. This is because if there is a prefix with a sum equal to `k` (in this step our `curr` value), then without initializing `0: 1`, `curr - k = 0` wouldn't show up in the hash map and we would "lose" this valid subarray.

In this case the numbers were all positive so each `curr - k` only showed up once. But in general with non-positive numbers being possible inputs, like if `nums = [1, -1, 1, -1]` with `k = 1` and thus prefix sum = `[1, 0, 1, 0]`, when `curr` is at the third index of the prefix sum, `curr - k = 1 - 1 = 0` is seen twice before (once at the second index and before the first because we needed to handle the case with `curr = k` from the last paragraph). So we won't be able to get away with a hash set this time (considering non-positive numbers in addition to positive ones).

In the following code, the prefix sum is calculated "on the fly' as the current value of `curr`, it is not done beforehand to completion as a pre-processing step.

```
from collections import defaultdict

class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        counts = defaultdict(int)
        counts[0] = 1
        ans = curr = 0

        for num in nums:
            curr += num
            ans += counts[curr - k]
            counts[curr] += 1

        return ans
```

To summarize:

- We use `curr` to track the prefix sum.
- At any index `i`, the sum up to `i` is `curr`. If there is an index `j` whose prefix is `curr - k`, tthen the sum of the subarray from `j + 1` to `i` is `curr - (curr - k) = k`.
- Because the array can have negative numbers, the same prefix can occur multiple times. We use a hash map `counts` to track how many times a prefix has occurred.
- At every index `i`, the frequency of `curr - k` is equal to the number of subarrays whose sum is equal to `k` that end at `i`. Add it to the answer.

The time and space complexity of this algorithm are both $O(n)$, where n is the length of `nums`. Each for loop iteration runs in constant time and the hash map can grow to a size of n elements.

Example 5: 1248 - Count Number of Nice Subarrays

Given an array of positive integers `nums` and an integer `k`, find the number of subarrays with exactly `k` odd numbers in them.

For example, given `nums = [1, 1, 2, 1, 1], k = 3`, the answer is `2`. The subarrays with `3` odd numbers in them are [1, 1, 2, 1, 1] and [1, 1, 2, 1, 1].

In the previous example, the constraint metric was a sum, so we had `curr` record a prefix sum. In this problem, the constraint metric is **odd number count**. Let's have `curr` track the count of odd numbers so far. We will keep track of the `curr`s at each index in a hash map. If there are `curr = 4` odd numbers at an index and `curr = 1` at a prior index, then the subarray formed similar to a prefix sum difference gives a value of `4 - 1 = 3` odd numbers between them. At every element, we can query `curr - k` again. We add each differences to the result if the difference equals `k` , i.e. (`curr - (curr - k) = k`).

```
from collections import defaultdict

class Solution:
    def numberOfSubarrays(self, nums: List[int], k: int) -> int:
        counts = defaultdict(int)
        counts[0] = 1
        ans = curr = 0

        for num in nums:
            curr += num % 2
```

```
            ans += counts[curr - k]
            counts[curr] += 1

        return ans
```

The time and space complexity is also $O(n)$.