Checking for existence

One of the most common applications of a hash table/set is determining if an element exists in $O(1)$.

Example 1: 1 - Two Sum

Given an array of integers `nums` and an integer `target`, return indices of two numbers such that they add up to `target`. You cannot use the same index twice.

In the brute force solution, the first for loop focuses on a number `num` and does a second for loop which looks for `target - num` in the array. With an array, looking for `target - num` is $O(n)$, but with a hash map, it is $O(1)$.

We can build a hash map as we iterate along the array, mapping each value to it's index. At each index `i`, where `num = nums[i]`, we can check our hash map for `target - num`. Adding key-value pairs and checking for `target - num` are all $O(1)$, so our time complexity will improve to $O(n)$.

The problem wants us to return the indices instead of the numbers themselves, so we can associate each number with its index.

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        dic = {}
        for i in range(len(nums)):
            num = nums[i]
            complement = target - num
            if complement in dic: # This operation is O(1)!
                return [i, dic[complement]]

            dic[num] = i

        return [-1, -1]
```

If the question wanted us to return a boolean indicating if a pair exists or to return the numbers themselves, then we could just use a set. However, since it wants the indices of the numbers, we need to use a hash map to "remember" what indices the numbers are at.

The time complexity is $O(n)$ as the hash map operations are $O(1)$. This solution also uses $O(n)$ space as the number of keys the hash map will store scales linearly with the input size.

Example 2: 2351 - First Letter to Appear Twice

Given a string `s`, return the first character to appear twice. It is guaranteed that the input will have a duplicate character.

Checking for existence can be done with a hash set in $O(1)$. So the overall runtime will be $O(1)$.

The space complexity is $O(m)$ where m is the number of allowable characters in the input.

```
class Solution:
    def repeatedCharacter(self, s: str) -> str:
        seen = set()
        for c in s:
            if c in seen:
                return c
            seen.add(c)

        return " "
```

Example 3: Given an integer array `nums`. find all the **unique** numbers `x` in `nums` that satisfy the following: `x + 1` is not in `nums`, and `x - 1` is not in `nums`.

In this case we need to check not just the previously seen items but all of the items in the array. So creating a set out of the entire array first is a good way to go. This is an example of preprocessing. Thus all the checks will take $O(1)$.

```
def find_numbers(nums):
    ans = []
    nums = set(nums)

    for num in nums:
        if (num + 1 not in nums) and (num - 1 not in nums):
            ans.append(num)
```

```
    return ans
```

The total time will be $O(n)$. The set will take up $O(n)$ space.

Anytime you find your algorithm running `if ... in ...` then consider using a hash map or set to store elements to have these operations run in $O(1)$.

SEE PROBLEMS 1832, 268, 1426