# More hashing examples

Example 1: 49 - Group Anagrams

Given an array of strings `strs`, group the anagrams (a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once) together.

For example, given `strs = ["eat","tea","tan","ate","nat","bat"]`, return (in any order) `[["bat"],["nat","tan"],["ate","eat","tea"]]`.

The cleanest way to know if two strings are anagrams of each other is by checking if they are equal after both being sorted. Also, all strings in a group will be the same when sorted, so we can use the sorted version as a key. We can map these keys to the groups themselves in a hash map, and then our answer is just the values of the hash map.

Essentially, every group has its own "identifier" (the sorted string), and we can use this identifier to group them in a hash map easily.

```
from collections import defaultdict

class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        groups = defaultdict(list)
        for s in strs:
            key = "".join(sorted(s))
            groups[key].append(s)

        return groups.values()
```

Note for Python: `dictionary.values()` doesn't actually return a list, but actually a view object. However, the LeetCode judge accepts it as a valid format.

Given $n$ as the length of `strs` and $m$ as the average length of the strings, we iterate over each string and sort it, which costs $)(nmlogm)$. Then, we need to iterate over the keys. In the worst case scenario, when there are no matching anagrams, there will be $n$ groups, which means this will cost $O(n)$, giving an overall time complexity of $O(nmlogm)$ (the final $+n$ is dominated). The space complexity is $O(nm)$ if you consider $m$ not to be a constant as each string will be placed in an array within the hash map.

Another way to solve this problem is to use a tuple of length 26 representing the count of each character as the key instead of the sorted string. This would technically solve the problem in $O(nm)$ because the 26 is a constant defined by the problem, but for test cases with smaller strings it would be slower due to the constant factor which is hidden by big O.

It also assumes that the strings can only have 26 different characters, which is valid but less general and less resistant to follow-ups.

---

Example 2 : 2260 - Minimum Consecutive Cards to Pick Up

Given an integer array `cards`, find the length of the shortest subarray that contains at least one duplicate. If the array has no duplicates, return `-1`.

We can use sliding window and/or a hash map. This question is equivalent to: what is the shortest distance between any two of the same element? If we go through the array and use a hash map to record the indices for every element, we can iterate over those indices to find the shortest distance. For example, given `cards = [1, 2, 6, 2, 1]`, we would map `1: [0, 4]`, `2: [1, 3]`, and `6: [2]`. Then we can iterate over the values and see that the minimum difference can be achieved from picking up the `2`s.

Iterate over the array once and record the position of each element in a hash map. The keys to the hash map will be the element, and the value will be an array of all the indices it appears at.

Because we iterate on the indices in ascending order, each array within the hash map will also be sorted ascending.

Now we can check each element individually. To find the minimum distance, we just need to check all adjacent pairs because the array is sorted.

```
from collections import defaultdict

class Solution:
    def minimumCardPickup(self, cards: List[int]) -> int:
        dic = defaultdict(list)
        for i in range(len(cards)):
            dic[cards[i]].append(i)

        ans = float("inf")
        for key in dic:
            arr = dic[key]
```

```
            for i in range(len(arr) - 1):
                ans = min(ans, arr[i + 1] - arr[i] + 1)

        return ans if ans < float("inf") else -1
```

The time complexity is still $O(n)$ even though we have a nested loop in the algorithm. This is because the inner loop in the nested loop can only iterate $n$ times in total, since it's iterating over indices of elements from the array, where $n$ is the length of the input array.

We can improve this algorithm slightly by observing that we don't need to store all the indices, but only the most recent one that we saw for each number. This improves the average space complexity. The current algorithm has $O(n)$ space complexity always, but with the improvement, it is only $O(n)$ in the worst case, when there are no duplicates.

```
from collections import defaultdict

class Solution:
    def minimumCardPickup(self, cards: List[int]) -> int:
        dic = defaultdict(int)
        ans = float("inf")
        for i in range(len(cards)):
            if cards[i] in dic:
                ans = min(ans, i - dic[cards[i]] + 1)

            dic[cards[i]] = i

        return ans if ans < float("inf") else -1
```

This algorithm also runs faster because we save on an iteration, although the time complexity of both algorithms is $O(n)$, where $n$ is the length of the input array.

---

Example 3: 2342 - Max Sum of a Pair With Equal Sum of Digits

Given an array of integers `nums`, find the maximum value of `nums[i] + nums[j]`, where `nums[i]` and `nums[j]` have the same **digit sum** (the sum of their individual digits). Return `-1` if there is no pair of numbers with the same digit sum.

This problem is similar to the first example we looked at with grouping anagrams. In the first example, groups were identified by their sorted string. In this problem, we can identify a group

by its digit sum. We iterate through the array and group all the numbers with the same digit sum together in a hash map. Then we can iterate over that hash map and for each group with at least 2 elements, find the 2 max elements by sorting.

```
from collections import defaultdict

class Solution:
    def maximumSum(self, nums: List[int]) -> int:
        def get_digit_sum(num):
            digit_sum = 0
            while num:
                digit_sum += num % 10
                num //= 10

            return digit_sum

        dic = defaultdict(list)
        for num in nums:
            digit_sum = get_digit_sum(num)
            dic[digit_sum].append(num)

        ans = -1
        for key in dic:
            curr = dic[key]
            if len(curr) > 1:
                curr.sort(reverse=True)
                ans = max(ans, curr[0] + curr[1])

        return ans
```

This algorithm is inefficient due to the sorting, which can potentially cost $O(nlogn)$ if every number has the same digit sum, where $n$ is the length of the input array. But we don't need to store all the numbers in the group. We can improve the time complexity and average space complexity by only saving the largest number seen so far for each digit sum.

```
from collections import defaultdict

class Solution:
    def maximumSum(self, nums: List[int]) -> int:
        def get_digit_sum(num):
```

```
            digit_sum = 0
            while num:
                digit_sum += num % 10
                num //= 10

            return digit_sum

        dic = defaultdict(int)
        ans = -1
        for num in nums:
            digit_sum = get_digit_sum(num)
            if digit_sum in dic:
                ans = max(ans, num + dic[digit_sum])
            dic[digit_sum] = max(dic[digit_sum], num)

        return ans
```

The first algorithm always uses $O(n)$ space because we store all the elements in the hash map's values, but with the improvement, the average case will use much less space since each key only stores an integer. We also save on an extra iteration and a sort in each iteration, giving us a time complexity of $O(n)$, where $n$ is the length of the input array.

---

SEE PROBLEMS 383, 771, 3