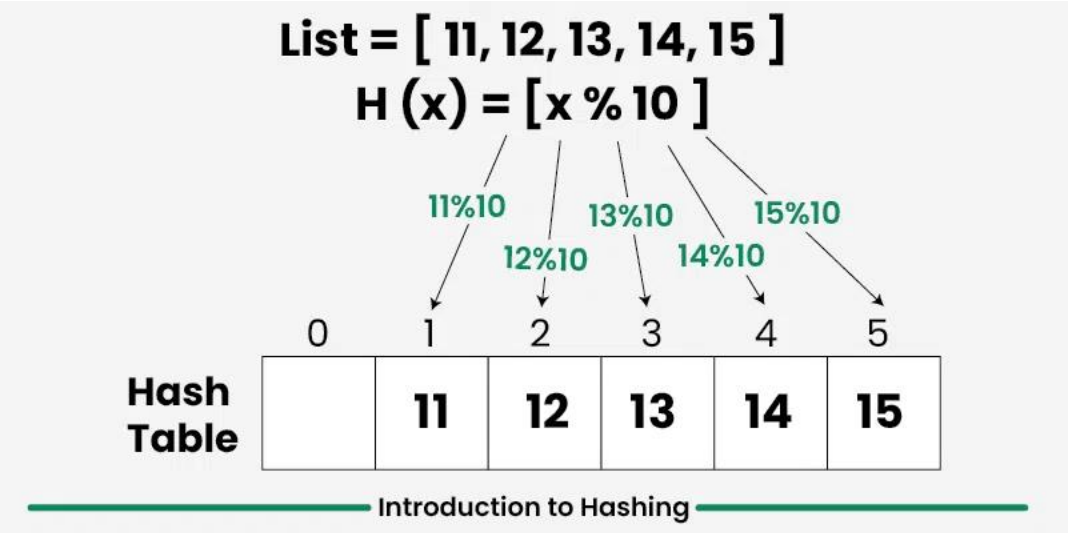
HASHING OR HASH TABLE

Hashing is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access. It involves mapping data to a specific index in a hash table using a hash function that enables fast retrieval of information based on its key. This method is commonly used in databases, caching systems, and various programming applications to optimize search and retrieval operations. The great thing about hashing is, we can achieve all three operations (search, insert and delete) in O(1) time on average.



→ **Hash Function:** You provide your data items into the hash function.
→ **Hash Code:** The hash function crunches the data and give a unique hash code. This hash code is typically integer value that can be used an index.
→ **Hash Table:** The hash code then points you to a specific location within the hash table.

A hash table is also referred as a hash map (key value pairs) or a hash set (only keys). It uses a hash function to map keys to a fixed-size array, called a hash table. This allows in faster search, insertion, and deletion operations.

Common hash functions include:

→ *Division Method:* Key % Hash Table Size
→ *Multiplication Method:* (Key * Constant) % Hash Table Size
→ *Universal Hashing:* A family of hash functions designed to minimize collisions

Hash Table operations in python

Hash tables in Python are implemented using dictionaries (dict), which provide an efficient way to store and retrieve key-value pairs. Here's a guide on common hash table operations using Python dictionaries:

→ Creating a Hash Table (Dictionary)

```
# Create an empty dictionary
hash_table = {}
```

```
# Create a dictionary with initial key-value pairs
hash_table = {'key1': 'value1', 'key2': 'value2'}
```

→ Inserting or Updating Elements

```
# Insert a new key-value pair
hash_table['key3'] = 'value3'

# Update the value of an existing key
hash_table['key1'] = 'new_value1'
```

→ Accessing Elements

```
# Access a value by its key
value = hash_table['key1']

# Using .get() to avoid KeyError if the key doesn't exist
value = hash_table.get('key4', 'default_value')  # Returns 'default_value' if 'key4' is not in the dictionary
```

→ Deleting Elements

```
# Remove a key-value pair using del
del hash_table['key2']

# Remove a key-value pair using .pop(), which also returns the removed value
value = hash_table.pop('key3', 'default_value')  # Returns 'default_value' if 'key3' is not found
```

→ Checking for Existence of a Key

```
# Check if a key is in the dictionary
exists = 'key1' in hash_table  # Returns True if 'key1' exists

# Check if a key is not in the dictionary
not_exists = 'key5' not in hash_table  # Returns True if 'key5' does not exist
```

→ Iterating Over a Hash Table

```
# Iterate over keys
for key in hash_table:
    print(key)

# Iterate over values
```

```
for value in hash_table.values():
    print(value)

# Iterate over key-value pairs
for key, value in hash_table.items():
    print(key, value)
```

→ Getting the Number of Elements

```
# Get the number of key-value pairs
size = len(hash_table)
```

→ Clearing the Hash Table

```
# Clear the entire dictionary
hash_table.clear()
```

→ Example of Common Operations:

```
# Create a hash table
hash_table = {'apple': 1, 'banana': 2, 'orange': 3}

# Insert a new item
hash_table['grape'] = 4

# Update an existing item
hash_table['apple'] = 5

# Access an item
print(hash_table['banana'])  # Output: 2

# Delete an item
del hash_table['orange']

# Check if a key exists
if 'grape' in hash_table:
    print('Grape exists')

# Iterate over the hash table
for key, value in hash_table.items():
    print(f"{key}: {value}")
```

Output:

Grape exists
apple: 5
banana: 2
grape: 4

LeetCode Problems

Problem: Find the Number of Winning Players

You are given an integer n representing the number of players in a game and a 2D array pick where pick[i] = [xi, yi] represents that the player xi picked a ball of color yi.

Player i wins the game if they pick strictly more than i balls of the same color. In other words,

Player 0 wins if they pick any ball.
Player 1 wins if they pick at least two balls of the same color.
...
Player i wins if they pick at leasti + 1 balls of the same color.
Return the number of players who win the game.

Note that multiple players can win the game.

Example 1:

Input: n = 4, pick = [[0,0],[1,0],[1,0],[2,1],[2,1],[2,0]]

Output: 2

Code:

```
class Solution:
    def winningPlayerCount(self, n: int, pick: List[List[int]]) -> int:

        result = 0
        hashmap = {}

        for player in range(n):

            hashmap[player] = [0]*11

        for index in range(len(pick)):

            hashmap[pick[index][0]][pick[index][1]] = hashmap[pick[index][0]][pick[index][1]] + 1

        for key in hashmap:
```

```
        for ball in hashmap[key]:

            if key < ball:
                result = result + 1
                break

    return result
```

Explanation:

→ n = 4 and pick = [[0,0],[1,0],[1,0],[2,1],[2,1],[2,0]].

hashmap[player] = [0]*11
{
  0: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  1: [2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  2: [1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  3: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
}

→ **Player 0**: 1 < 0 for all balls, so Player 0 doesn't win.

→ **Player 1**: 1 < 2 (for ball 0), so Player 1 wins.

→ **Player 2**: 2 > 1 (for ball 1), so Player 2 wins.

→ **Player 3**: No balls, so Player 3 doesn't win.

Output: 2

-→ Time complexity: O(n+m).
-→ Space complexity: O(n).

Problem: Find the Number of Good Pairs I

You are given 2 integer arrays nums1 and nums2 of lengths n and m respectively. You are also given a positive integer k.

A pair (i, j) is called good if nums1[i] is divisible by nums2[j] * k (0 <= i <= n - 1, 0 <= j <= m - 1).

Return the total number of good pairs.
Example 1:

Input: nums1 = [1,3,4], nums2 = [1,3,4], k = 1

Output: 5

Code:

```
class Solution:
    def numberOfPairs(self, nums1: List[int], nums2: List[int], k: int) -> int:
        count = 0
        for i in range(0, len(nums1)):
            for j in range(0, len(nums2)):
                if nums1[i] % (nums2[j]*k) == 0:
                    count += 1
        return count
```

Explanation:

nums1 = [1, 3, 4], nums2 = [1, 3, 4], k = 1
count = 0

→ (i = 0, nums1[i] = 1)
(j = 0, nums2[j] = 1):
Check: 1 % (1 * 1) == 0 -> True
count = 1.

Inner Loop (j = 1, nums2[j] = 3):
Check: 1 % (3 * 1) == 0 -> False
count = 1.

Inner Loop (j = 2, nums2[j] = 4):
Check: 1 % (4 * 1) == 0 -> False
count = 1.

→ (i = 1, nums1[i] = 3)

Inner Loop (j = 0, nums2[j] = 1):
Check: 3 % (1 * 1) == 0 -> True
count = 2.

Inner Loop (j = 1, nums2[j] = 3):
Check: 3 % (3 * 1) == 0 -> True
count = 3.

Inner Loop (j = 2, nums2[j] = 4):
Check: 3 % (4 * 1) == 0 -> False
count = 3.

→ (i = 2, nums1[i] = 4)
Inner Loop (j = 0, nums2[j] = 1):
Check: 4 % (1 * 1) == 0 -> True
count = 4.

Inner Loop (j = 1, nums2[j] = 3):
Check: 4 % (3 * 1) == 0 -> False
count = 4.

Inner Loop (j = 2, nums2[j] = 4):
Check: 4 % (4 * 1) == 0 -> True
count = 5.

Output: 5

-→ Time complexity: $O(n^2)$.
-→ Space complexity: O(1).

Problem: Maximum Length Substring With Two Occurrences

Given a string s, return the maximum length of a
substring
such that it contains at most two occurrences of each character.

Example 1:

Input: s = "bcbbbcba"

Output: 4

Code:

```
class Solution:
    def maximumLengthSubstring(self, s: str) -> int:
        n=len(s)
        r=0
        for i in range(n+1):
            for j in range(i+1,n+1):
                a=(s[i:j])
                c=0
                for k in a:
                    if(a.count(k)>2):
                        c+=1
                        break
                if(c==0):
                    r=max(r,len(a))
        return(r)
```

Explanation:

s = "bcbbbcba", n = len(s) = 8. r = 0

→ (i = 0), (j = 1 to 8):

s[0:1] = "b"
s[0:2] = "bc"
s[0:3] = "bcb"
s[0:4] = "bcbb"

→ (i = 1), (j = 2 to 8):

s[1:2] = "c"
s[1:3] = "cb"
s[1:4] = "cbb"
s[1:5] = "cbbb"

→ (i = 2), (j = 3 to 8):

s[2:3] = "b"
s[2:4] = "bb"
s[2:5] = "bbb"

→ (i = 3), (j = 4 to 8):
s[3:4] = "b"
s[3:5] = "bb"
s[3:6] = "bbc"
s[3:7] = "bbcb"

Output: 4

-→ Time Complexity: $O(n^3)$
-→ Space Complexity: O(n)

Problem: Points That Intersect With Cars

You are given a 0-indexed 2D integer array nums representing the coordinates of the cars parking on a number line. For any index i, nums[i] = [starti, endi] where starti is the starting point of the ith car and endi is the ending point of the ith car.

Return the number of integer points on the line that are covered with any part of a car.

Example 1:

Input: nums = [[3,6], [1,5], [4,7]]
Output: 7

Code:

```
class Solution:
```

```python
def numberOfPoints(self, nums: List[List[int]]) -> int:
    a = set()
    for i in nums:
        for j in range(i[0], i[1]+1):
            if j not in a:
                a.add(j)
    return len(a)
```

Explanation:

nums = [[3, 6], [1, 5], [4, 7]]
a = set()

→ Interval [3, 6], j from 3 to 6:
Add 3: a = {3}
Add 4: a = {3, 4}
Add 5: a = {3, 4, 5}
Add 6: a = {3, 4, 5, 6}

→Interval [1, 5], j from 1 to 5:
Add 1: a = {1, 3, 4, 5, 6}
Add 2: a = {1, 2, 3, 4, 5, 6}

→Interval [4, 7], j from 4 to 7:
Add 7: a = {1, 2, 3, 4, 5, 6, 7}

len(a) = 7

Output: 7

-→ Time Complexity: $O(n^2)$
-→ Space Complexity: O(n)

Problem: Most Frequent IDs

The problem involves tracking the frequency of IDs in a collection that changes over time. You have two integer arrays, nums and freq, of equal length n. Each element in nums represents an ID, and the corresponding element in freq indicates how many times that ID should be added to or removed from the collection at each step.

Addition of IDs: If freq[i] is positive, it means freq[i] IDs with the value nums[i] are added to the collection at step i.
Removal of IDs: If freq[i] is negative, it means -freq[i] IDs with the value nums[i] are removed from the collection at step i.
Return an array ans of length n, where ans[i] represents the count of the most frequent ID in the collection after the ith step. If the collection is empty at any step, ans[i] should be 0 for that step.

Example 1:

Input: nums = [2,3,2,1], freq = [3,2,-3,1]

Output: [3,3,2,2]

Code:

```
from sortedcontainers import SortedList, SortedDict, SortedSet
class Solution:
    def mostFrequentIDs(self, nums: List[int], freq: List[int]) -> List[int]:
        seen = SortedSet()
        mp = defaultdict(int)
        res = []

        for a , b in zip(nums , freq):
            if mp[a] == 0:
                mp[a] += b
                seen.add((b, a))
                res.append(seen[-1][0])
                continue
            p = mp[a]
            if b < 0:
                rem = seen.remove((mp[a] , a))

                if b < p:
                    seen.add((p + b , a))
            else:

                rem = seen.remove((p , a))
                seen.add((p + b , a))

            mp[a] += b
            mx_so_far = seen[-1][0]
            res.append(mx_so_far)
        return res
```

Explanation:

seen = SortedSet(), mp = defaultdict(int), res = []

→ (a = 2, b = 3):
mp[2] = 0, we add 3 to mp[2].
Add (3, 2) to seen.

res = [3]

→ (a = 3, b = 2):
mp[3] = 0, we add 2 to mp[3].
res =[3, 3]

→ (a = 2, b = -3):
mp[2] = 3, we remove (3, 2) from seen.
res =[3, 3, 2]

→ (a = 1, b = 1):
mp[1] = 0, we add 1 to mp[1].
Add (1, 1) to seen.
res = [3, 3, 2, 2]

Output: [3, 3, 2, 2]

-→ Time Complexity: O(n log m)
-→ Space Complexity: O(m + n)

Problem: Find the Longest Equal Subarray

You are given a 0-indexed integer array nums and an integer k.

A subarray is called equal if all of its elements are equal. Note that the empty subarray is an equal subarray.

Return the length of the longest possible equal subarray after deleting at most k elements from nums.

A subarray is a contiguous, possibly empty sequence of elements within an array.

Example 1:

Input: nums = [1,3,2,3,1,3], k = 3
Output: 3

Code:

```
class Solution:
    def longestEqualSubarray(self, nums: List[int], k: int) -> int:
        ans, idxs = 1, dict()
        for i, v in enumerate(nums):
            if v not in idxs: idxs[v] = list()
            idxs[v].append(i)
        for poses in idxs.values():
            i = 0
            for j in range(len(poses)):
                while i <= j and j - i + k < poses[j] - poses[i]: i += 1
```

```
            ans = max(ans, j - i + 1)
        return ans
```

Explanation:

nums = [1, 3, 2, 3, 1, 3] and k = 3.

→ For 1 at index 0: idxs = {1: [0]}
→ For 3 at index 1: idxs = {1: [0], 3: [1]}
→ For 2 at index 2: idxs = {1: [0], 3: [1], 2: [2]}
→ For 3 at index 3: idxs = {1: [0], 3: [1, 3], 2: [2]}
→ For 1 at index 4: idxs = {1: [0, 4], 3: [1, 3], 2: [2]}
→ For 3 at index 5: idxs = {1: [0, 4], 3: [1, 3, 5], 2: [2]}

→ For poses = [0, 4]
i = 0
For j = 0: ans = max(1, 0 - 0 + 1) = 1
For j = 1: ans = max(1, 1 - 0 + 1) = 2

→ For poses = [1, 3, 5]
Initialize i = 0
For j = 0: ans = max(2, 0 - 0 + 1) = 2
For j = 1: ans = max(2, 1 - 0 + 1) = 2
For j = 2: ans = max(2, 2 - 0 + 1) = 3

→ For poses = [2]
Initialize i = 0
For j = 0: ans = max(3, 0 - 0 + 1) = 3

Output: 3

-→ Time complexity: O(n).
-→ Space complexity: O(n).

Challenges faced to solve Hash Table leet code:

When working on hash table-related LeetCode problems, especially at the easy and medium levels, there are several challenges I faced:

→ **Collision Handling:**
Challenge: Understanding and implementing collision resolution strategies, such as chaining or open addressing, can be tricky. In some problems, need to handle scenarios where multiple keys hash to the same index.
Example: Problems that involve counting frequencies or detecting duplicates often require careful handling of collisions.

→ **Designing Custom Hash Functions**:
Challenge: For problems requiring a custom hash function, creating a function that efficiently distributes keys across the hash table can be challenging. Poor hash functions can lead to many collisions, affecting performance.

Example: Custom data structures or algorithms that rely on hashing often require you to design a robust hash function.

→ **Handling Edge Cases**:
Challenge: Identifying and correctly handling edge cases, such as empty inputs, very large numbers, or special characters in strings, can be difficult. Missing these cases often leads to incorrect or inefficient solutions.
Example: Problems like finding the first unique character in a string or checking for anagrams might require special attention to edge cases.

→ **Memory Management:**
Challenge: Hash tables can consume a lot of memory, especially if the keys or values are large objects. Managing memory efficiently without compromising performance can be a challenge.
Example: Problems that involve storing large amounts of data or performing operations on massive datasets might need optimizations to avoid memory issues.

→ **Dealing with Key-Value Pairs:**
Challenge: Some problems require careful manipulation of key-value pairs, such as updating values based on specific conditions or retrieving keys that meet certain criteria.
Example: Problems involving cache implementations or frequency counters often require efficient handling of key-value pairs.

→ **Optimizing Time Complexity:**
Challenge: While hash tables provide average O(1) time complexity for operations like insert and lookup, certain problems require careful optimization to maintain this performance, especially when dealing with large inputs.
Example: Optimizing solutions for problems like finding the longest substring without repeating characters can be challenging, as need to balance between space and time efficiency.

→ **Understanding Problem Requirements:**
Challenge: Some problems may have ambiguous requirements or tricky constraints that make it difficult to decide when and how to use a hash table effectively.
Example: Problems like group anagrams or two-sum variations often have multiple possible approaches and choosing the right one can be challenging.