Neetcode

Arrays and hashing

Hash table has a unique way of storing items using a hash function in case where a hash function computes the same thing for two items we have what we call the hash collision.

To solve this we use collision resolution( an example is separate chaining where we attach a linked list to each index ) -> open addressing

In python a hash map is a dictionary, that has a key value pair

NB: keys in a hash map are immutable.

In Python, a **defaultdict** is a specialized dictionary from the **collections** module. It behaves like a regular dictionary but has a key difference: if you try to access or modify a key that doesn't exist in the dictionary, the **defaultdict** will automatically create an entry for that key with a default value. The default value is specified when the **defaultdict** is created and can be any type such as int, list, set, etc.

First Question – Contains Duplicate

My Brute force solution

class Solution:

def containsDuplicate(self, nums: List[int]) -> bool:

# brute force approah iterate to every single element twice and ignore the element of comparison using two for loops

n = len(nums)

for i in range(n):

for j in range(n):

if i != j:

if nums[i] == nums[j]:

return True

return False

This solution I know has O(n^2) hence it is very inefficient and not the best, but it is always good to comprehend the question from the brute force perspective then begin to optimize from there.

Second solution

class Solution:

def containsDuplicate(self, nums: List[int]) -> bool:

# using the dictionary to count each element then we and find if a count is more than 1

dict = {}

for i in nums:

if i in dict:

dict[i] += 1

else:

dict[i] = 1

for i in dict.values():

if i > 1:

return True

return False

This solution used hashmaps -> dictionary and has a time complexity of O(n ) with the most expensive computation being checking each elements, however this must be done hence it is time efficient, the only problem might be with the space complexity.

def containsDuplicate(nums):

# I believe this should be the most efficient one

# here we will use a set and then check if an element is in the set, once we find it the code ends and returns saving space and time

# create a set called seen to keep track of the elements

seen = set()

#itereate throguh and add elements to seen if there are new

for num in nums:

if num in seen:

# once we see the number already in the seen set, we end implying there is a duplicate

return True

seen.add(num)

print(containsDuplicate([1,2,1]))

This is the most efficient since it has a space and time complexity of O(n), hence in an interview this is what I will go for in the end as the most optimized solution.

Question 2 – Valid Anagrams

def isAnagram(s,t):

new\_s = sorted(s)

new\_t = sorted(t)

if new\_s == new\_t:

return True

else:

return False

this is my first solution which is not so optimal with time complexity of O(nlogn)

The function you provided checks if two strings, `s` and `t`, are anagrams by sorting the strings and then comparing them. To determine the time complexity of this function, let's analyze each part:

1. Sorting the strings:

- Sorting a string using `sorted()` function in Python generally uses Timsort, which has a time complexity of \(O(n \log n)\), where \(n\) is the length of the string being sorted.

- Since the function sorts both strings `s` and `t`, and if we assume `n` is the length of `s` and `m` is the length of `t`, then sorting `s` takes \(O(n \log n)\) and sorting `t` takes \(O(m \log m)\).

2. Comparing the sorted strings:

- The comparison of two lists (which are the results of the sorted strings) is linear with respect to the number of elements in the lists. Thus, if both strings are of the same length (say the shorter string has length \(k\)), this comparison takes \(O(k)\), where \(k = \min(n, m)\).

Combining these complexities, the dominant factor here is the sorting step, so the overall time complexity of the function `isAnagram(s, t)` is \(O(n \log n + m \log m)\). If the strings are of similar length, this simplifies to \(O(n \log n)\) assuming \(n \approx m\).

This is not too efficient, the other method is using a hashmap and couting the elements in the two strings and then comparing the count of eeach string, also with the base case, compare the length of the two strings, once not equal then they can’t be anagrams.

def isAnagram(s,t):

#compare the length of string s and string t

if len(s) != len(t):

return False

# count elemets in s and t using hashmaps

countS, countT = {}, {}

# go throught the two strings and count them into the hashmap

for char in s:

if char in countS:

countS[char] += 1

else:

countS[char] = 1

for char in t:

if char in countT:

countT[char] += 1

else:

countT[char] = 1

for c in countS:

if countS[c] != countT.get(c, 0):

return False

return True

another method is to create two hash maps and count the elemnts in both strings, when done compare the count of each character if not equal then it isn’t an anagram.

def isAnagram(self, s: str, t: str) -> bool:

if len(s) != len(t):

return False

count = {}

for char in s:

count[char] = count.get(char, 0) + 1

for char in t:

if char not in count:

return False

else:

count[char] -= 1

for i in count.values():

if i != 0:

return False

else:

return True

most efficient is to use two hashmaps, if the character exist you subtract, then after that you iterate through all the values of the hashmap and if not 0, then it is false else true

NB: from collections import Counter, does everything for us