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

Question 3 – Two sums

def twoSum(self, nums: List[int], target: int) -> List[int]:

n = len(nums)

for i in range(n):

for j in range(n):

if nums[i] + nums[j] == target and i != j:

return [i,j]

This solution is O(n^2), works alright but has an inefficient time complexity.

def twoSum(self, nums: List[int], target: int) -> List[int]:

# unpack elements in the list into a hashmap -> spacce and time of O(n)

nums\_map = {}

index = 0

for num in nums:

nums\_map[num] = index

index += 1

# we will iterate throught the array, take the first element, find the remainder, then look up the remainder in the hashmap then get the index

for i in range(len(nums)):

remainder = target - nums[i]

if remainder in nums\_map and i!= nums\_map[remainder]: # being very careful of not counting the same index twice.

return[i, nums\_map[remainder]]

second solution I believe personally is more efficient, since it works in an O(n ) time and space complexity.

This is still O(n), but just more compact

# unpack elements in the list into a hashmap -> spacce and time of O(n)

nums\_map = {}

# everything can be kept compact by using enumerate

for i, num in enumerate(nums):

remainder = target - num

if remainder in nums\_map:

return[i, nums\_map[remainder]]

nums\_map[num] = i

For char in countS:

If countS[char] != count.get(char, 0):

Return false

Return True

Question 4

class Solution:

def groupAnagrams(self, strs: List[str]) -> List[List[str]]:

# create a list to take the grouped anagrams

grouped\_anagrams = []

# create helper function to check if it is an anagram

def isAnagram(word1, word2):

return sorted(word1) == sorted(word2)

# iterate through the list

seen = set() # used to check if we have seen a particular word already

for i in range(len(strs)):

if strs[i] in seen:

continue

anagram\_list = [strs[i]]

seen.add(strs[i])

for j in range(i + 1, len(strs)):

if isAnagram(strs[i], strs[j]):

anagram\_list.append(strs[j])

seen.add(strs[j])

grouped\_anagrams.append(anagram\_list)

return grouped\_anagrams

For this I created a function to check if two words are anagrams, then I use two for loops and scan through the list to check for anagrams and grouped them, mand I used the seen set to ensure there are no multiple duplicates.

Next solution optimized using hashmaps of course with O(m\*n) runtime

def groupAnagrams(self, strs: List[str]) -> List[List[str]]:

# create a default dict hashmap with key of sequence of characters, and value grouped anagrams.

grouped\_anagrams = defaultdict(list)

# iterate throught a list of strings and iterate also through characters in each string hence O(m\*n), where m is the length of the list and n the average length of each string

for s in strs:

# have a counter array for each word with count for each character

count = [0] \* 26

for c in s:

count[ord(c) - ord('a')] += 1

grouped\_anagrams[tuple(count)].append(s)

return list(grouped\_anagrams.values())

Question topK frequent element

from collections import defaultdict

def topKFrequent(nums, k):

# count all the elements in the array using the hashmap

count = {}

for num in nums:

if num in count:

count[num] += 1

else:

count[num] = 1

sorted\_values = sorted(count.values(), reverse = True)

grouped = []

# for i in sorted\_values[:k]:

# num = list(count.keys())[list(count.values()).index(i)]

# grouped.append(i)

for num in count.keys():

if count[num] in sorted\_values[:k]:

grouped.append(num)

return grouped

nums = [1,1,1,2,2,3]

k = 2

print(topKFrequent(nums,k))

My solution which is nlogn because of the sorting algorithm, so all I did was to count the elements with the key being the number and value being its frequency. After that I extract the values and sort it in descending order and iterate through up to the kth element in this new array then I then run a loop to check for the if the key correspond to this value and then it appends it.

class Solution:

def topKFrequent(self, nums: List[int], k: int) -> List[int]:

count = {}

freq = [[] for \_ in range(len(nums) + 1)]

for num in nums:

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

for num, frequency in count.items():

freq[frequency].append(num)

res = []

for i in range(len(freq)-1, 0, -1):

for n in freq[i]:

res.append(n)

if len(res) == k:

return res

Most efficient algo, with O(n)

Encode and Decode problem , leetcode premium and lint code

This question just had me to convert a list of strings into a string, so just iterate and concartenate, with a special character after each string in the list. Then a condition is set that when we get to the last element if the list we add it without the special character.

After that we decode by using split and setting it ito the special character. However I tried to challenge my self by not using split

def encode(self, strs: List[str]) -> str:

"""Encodes a list of strings to a single string.

"""

new\_str = ""

# for word in strs:

# new\_str += f"{word}-"

for i in range(len(strs)):

if i == len(strs) - 1:

new\_str += f"{strs[i]}"

else:

new\_str += f"{strs[i]}!-!"

return new\_str

def decode(self, s: str) -> List[str]:

"""Decodes a single string to a list of strings.

"""

return s.split('!-!')

my new algo for implementing split is a two pointer technique with lpointer at 0 and rpointer at 1, so all I do is to continue increasing the r pointer till I get to my separator, when I get there I append from the left pinter to the rpointer , if r\_ptr is equal to the last element in the string, we append from left to right,

this is a success in both cases so we move the left to right + 1, then the right a step forward

if none of this happen we go to the else clause and increase the right pointer then continue the while loop to start again.

This fails some test cases so dawg.

**Optimal solution.**

class Codec:

def encode(self, strs: List[str]) -> str:

# iterate throught the list, encode it start with the length, then delimiter

new\_s = ""

for s in strs:

length = len(s)

new\_s += str(len(s))+ "#" + s

return new\_s

def decode(self, s: str) -> List[str]:

res = []

i = 0

while i < len(s):

j = i

while s[j] != "#":

j += 1

length = int(s[i:j])

res.append(s[j+1: j + 1 + length])

i = j + 1 + length

return res

this is the most optiman solution with no external fucntions.

So we begin with encoding, we first start we getting the length, then add a delimiter then the string,

For decode, use two while loops, first we start with an index I and keeping it to be less than

The length of the string, then we set I into j then initiat another while loop, keeping it not equal to the delimiter and incrementing it each instant the delimiter is not present..

When we get the delimeter, we start from the I and extend to the j and convert it to int since it is the length of the next string then we append j+ 1 to j+ 1 + length to the result variable to get that portion of the list .

**Product Except self,**

I first came up with bruteforce with O(n\*\*2) using two for loops and then chehck if I != j then multiply everything with our dummy variable product containing 1.

But for optimal solution

def productExceptSelf(nums):

res = [1] \* len(nums)

prefix = 1

for i in range(len(nums)):

res[i] = prefix

prefix \*= nums[i]

postfix = 1

for i in range(len(nums)-1, -1, -1):

res[i] \*= postfix

postfix \*= nums[i]

return res

print(productExceptSelf([1,2,3,4]))

this works, so we have a result array with initially 1 element in each since one is the identity element for multiplication.

Then we employ prefix and postfix. Throught pattern recorginitoin we realize that the first of both is 1 so we initialize prefix and postfix to 1, then beginning with prefix, we iteratie and multiple element positions to the prefix except the last element, and store it in result

For post fix, starting from the last element we multiply end omit the first element and store this in result straight up .

Valid Sodoku