2 Weeks Grind TS

Contiguous : 1 Connected

Leetcode

DSA Cheatsheets

① Array: Arrays hold values of the same type at contiguous memory locations. In an array, we're usually concerned about two things — the position/index of an element and the element itself.

In some languages like Python, JavaScript, Ruby, PHP, the array (or list in Python) size is dynamic and you do not need to have a size defined beforehand when creating the array.

Advantages:

ⓐ Store multiple elements of the same type with one single variable name.
ⓑ Accessing elements is fast as long as you have the index, as opposed to linked lists where you have to traverse from the head.

Disadvantages:

ⓐ Addition and removal of elements into/from the middle of the array is slow because the remaining elements needs to be shifted to accomodate the new/missing element. An exception to this is if the position to be inserted/removed is at the end of the array.

For certain languages where the array size is fixed, it cannot alter its size after initialization. If an insertion causes the total number of elements to exceed its size, a new array have to be copied over. The act of creating a new array and transferring elements over takes O(n) time.

→ Definition of Array: Contiguous area of memory consisting of equal-size elements indexed by contiguous integers.

→ What's So Special about Arrays:

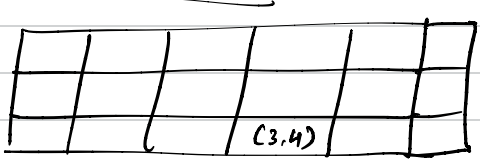
It has constant-time access to any particular element in array (Read/write).

How?

Computer uses this arithmetic formula to regulate array_addr:

array_addr + elem_size × (i − first_index)

→ Multi Dimensional array:

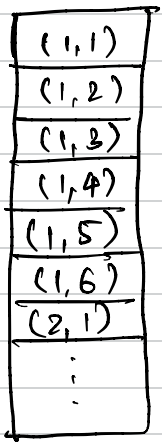


To get address of (3,4), we do:

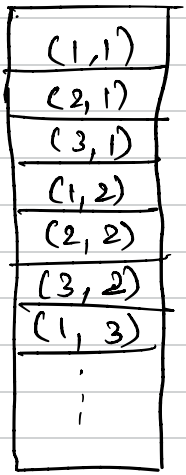
arr_addr + elem_size × ((3−1) × 6 + (4−1))

(3−1): row; ×6: no. of col; (4−1): column

Row major:



Column major:



Times for Common Operations

	Add	Remove
Beginning	O(n)	O(n)
End	O(1)	O(1)
Middle	O(n)	O(n)

Common terms:

Subarray: A range of contiguous values within an array.

Subsequence: A sequence that can be derived from the given sequence by deleting some or no elements without changing the order of the remaining elements.

Time complexity

Operation	Big-O	Note
Access	O(1)	
Search	O(n)	
Search (sorted array)	O(log(n))	
Insert	O(n)	Insertion would require shifting all the subsequent elements to the right by one and that takes O(n)
Insert (at the end)	O(1)	Special case of insertion where no other element needs to be shifted
Remove	O(n)	Removal would require shifting all the subsequent elements to the left by one and that takes O(n)
Remove (at the end)	O(1)	Special case of removal where no other element needs to be shifted

demarcate :- to distinguish

** Slicing and concatenating arrays would take O(n) time. Use start and end indices to demarcate a subarray/range where possible.

Corner Cases:

- Empty sequence
- Sequence with 1 or 2 elements
- Sequence with repeated elements
- Duplicated values in the sequence.

Techniques:

∵ both arrays and strings are sequences, most of the techniques here will apply to string problems.

① **Sliding Window:** In a sliding window, the two pointers usually move in the same direction will never overtake each other.

② **Two Pointers:** is a more general version of sliding window where the pointers can cross each other and can be on different arrays.

When you are given two arrays to process, it is common to have one index per array (pointer) to traverse/compare the both of them, incrementing one of the pointers when relevant.

Ex: Merge Sorted Array.

③ **Traversing from the right:**

Sometimes you can traverse the array starting from the right instead of the conventional approach of from the left.

④ **Sorting the array**

⑤ **Precomputation:** For questions where summation or

multiplication of a subarray is involved, pre-computation using hashing or a prefix/suffix sum/product might be useful.

(6) Index as a hash key: If you are given a sequence and the interviewer asks for O(1) space, it might be possible to use the array itself as a hash table.

(7) Traversing the array more than once: Traversing the array twice/thrice (as long as fewer than n times) is still O(n). Sometimes it helps in solving a problem while keeping time complexity to O(n).

(121)

Q Buy and sell stocks

Time Comp : O(n)
Space " : O(1)

There are multiple approaches to solve this question but the simplest and efficient code I got is using this approach:

Approach Related to Kaden's algo (DP)

1. Initialize 'buy' to the first element, profit as 0.
2. Iterate through prices starting from the second element.
3. If buy > sell, update buy ; buy = sell. else update the profit by checking if the difference b/w the curr. price and the buying price is greater than the current profit.
4. Return the final profit.

Ex: [7, 1, 5, 3, 6, 4] profit = 0

7 → buy; 1, 5, 3, 6, 4 → sell

(1) → buy = 7
sell = 1
buy > sell
∴ buy = sell
∴ sell = 1

(2) buy = 1
sell = 5
⇒ sell > buy:
profit = max (prof, sell − buy)
= max (0, 5−1) = 4

(3) buy = 1
sell = 3
profit = 4
⇒ sell > buy
⇒ profit = max (4, 2)
= 4

(4) buy = 1
sell = 6
profit = 4
⇒ sell > buy
⇒ profit = max (4, 6−1)
= max (4, 5)
= 5

(5) buy = 1
sell = 4
profit = 5
⇒ sell > buy
⇒ 4 − 1 = 3
⇒ curr profit < 5
⇒ profit = 5

Q → Two sum approach : (001)

Brute Force Approach :

```
class Solution:
    def twoSum(self, nums: List[int], target: int) ->
                                        List[int]:
        for i in range(0, len(nums)-1):
            for j in range(i+1, len(nums)):
                if nums[i]+nums[j]==target:
                    return [i,j]
        return []
```

→ Time Complexity : $O(n^2)$
Space " : $O(1)$

Two Pointer Approach :

T.C = O(n log n)
S.C > O(1) ↓ due to sorting

[, , , , ,]
↑ first pointer ↑ second pointer

first pointer + second pointer == target
↳ but array should be sorted .

o/p should be indices, so we enumerate nums . (attach value with index)

enum_nums = [(index 1, value 1), (index 2, value 2)...]

```
Class Solution:
    def twoSum(self, nums: List[int],
               target: int) -> List[int]:
        sorted_nums = sorted(enumerate(nums),
                             key = lambda x: x[1])
        left, right = 0, len(nums) - 1
        while left < right:
            current_sum = sorted_nums[left][1]
                        + sorted_nums[right][1]
            if current_sum < target:
                left += 1
            elif current_sum > target:
                right -= 1
            else:
                return [sorted_nums[left][0],
                        sorted_nums[right][0]]
        return []
```

→ Another Optimal way : Two pass Hash Table

T.C. → O(n)

```
class Solution:
    def twoSum(self, nums: List[int], target: int)
               -> List[int]:
        nums_Map = {}
        n = len(nums)
        # Hash Table
        for i in range(n):
            nums_Map[nums[i]] = i
```

```
# Find the complement
    for i in range(n):
        complement = target - nums[i]
        if complement in nums_Map and
            nums_Map[compliment] != i:
            return [i, nums_Map[complement]]

    return []
```

Final Sol^{n}: One-pass Hash Table

T.C $\rightarrow O(n)$

```
class Solution:
    def twoSum(self, nums: List[int], target:
                int) -> List[int]:
        nums_Map = {}
        n = len(nums)

        for i in range(n):
            complement = target - nums[i]
            if complement in nums_Map:
                return [nums_Map[comple
                        -ment], i]
            nums_Map[nums[i]] = i

        return []
```

Step by Step Execution:

Ex: ① Initialization

nums = [2, 7, 11, 15]

target = 9

nums_Map = {} (an empty dictionary)

n = 4

② Iteration I (i = 0)

- nums[i] = 2
- complement = target - nums[i] = 9 - 2 = 7
- check if 'complement (7)' is in 'nums_Map'. It is not.
- Update 'nums_Map' with 'nums[i]: nums_Map' = {2:0}

Current State: nums_Map = {2:0}

3. Iteration 2 (i = 1):
   - nums[1] = 7
   - complement = target − nums[1] = 9 − 7 = 2
   - Check if complement (2) is in nums_Map
     ↳ It is available (nums_Map[2] = 0)

- Since the complement is found, return the indices: [nums_Map[2], 1] = [0, 1].

Return : [0, 1]

Q → Majority Element (169)

↳ Given array "nums" → size = n
return the 'majority' element
↓
element that appears more than ⌊n/2⌋ times

↳ Assume majority element always exists in the array.

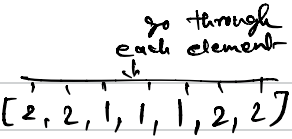
↳ follow up → make T.C. linear and space O(1)

→ we have multiple approaches:

① use Counter class from collections module

```
def majorityElement(self, nums: List[int]) ->
                                    int:
    s = Counter(nums).most_common()
    return s[0][0]
```

② Brute force approach:



We check frequency of each element one by one and check if it's larger than $\lfloor n/2 \rfloor$

↳ Costly · T.C = $O(n^2)$
S.C = $O(1)$

We need TC to be $O(n)$ and SC to be $O(1)$

So, next approach, which will be slightly better

③ Hashmap: We count frequency of elements using hashmap and the element having freq $> n/2$ is the majority element

nums = [2, 2, 1, 1, 1, 2, 2] n=7

freq ⇒ 2 = 4
1 = 3

∴ $4 > n/2$

∴ hashmap has both ② key and value, its easy to extract and return the key. [freq is value]

Here, TC ⇒ $O(n)$
SC ⇒ $O(n)$

Here, we go bad.

(4) To optimize it to make it a bit better

Let's do sorting.

So T.C. ⇒ $O(n \log n)$
S.C ⇒ $O(n \log n)$

Here, we simply sort the element, we will pick the element which is at the $n/2$ index.

As we know, there's always going to be a majority element, this will work.

for ex :—

$[2, 1, 2, 1, 2]$ $n = 5$

∵ we know there's going to be a M.E.

Sort the array:

$[1, 1, 2, 2, 2]_{n=5}$

∴ the no. at $\lfloor n/2 \rfloor$ ⇒ 2 is your ME

(5)★ Boyer-Moore's Voting Algo:

↳ Used to approach 'Majority' types question.

How to approach:

Here, Two elements: majority element = +1, minority element = -1 } keep track of sum

when sum > 0 then that is the majority element.

So, basically we initialize the first element as the Majority element and count +1 if the element occurs next, we do count -1 (we keep the same ME, until count = 0) if there's another element (next element) ↳ minority element

Ex: [2, 2, 1, 1, 1, 2, 2]n=7

① initialize majority element = -1, count = 0

② ME = 2, count = 1

③ ME = 2, count = 2

④ ME = 2, count = 1

⑤ ME = 2, count = 0 ∴ [2, 2, 1, 1, 1, 2, 2]n=7

⑥ New ME = -1, count = 0 ↓ just to explain

⑦ ME = 1, count = 0

⑧ ME = 2, count = 1, array ends.

So ME = 2

only traversing as much as N, once

T.C = O(n)
S.C = O(1)