Problems and Algorithms

Eric Sohel

1 General Problem Solving Strategies

1.1 Understand the Problem

1. What is the unknown? 2. What is the data? 3. What is the condition? 4. Is it possible to satisfy the condition? 5. Is the condition sufficient to determine the unknown? Or is it insufficient? Or redundant? Or contradictory? 6. Draw a figure. Introduce suitable notation. 7. Separate the various parts of the condition. Can you write them down?

1.2 Devising a Plan

1. Have you seen it before? Or have you seen the same problem in a slightly different form? 3. Look at the unknown! And try to think of a familiar problem having the same or a similar unknown. 4. Here is a problem related to yours and solved before. Could you use it? Could you use its result? Could you use its method? Should you introduce some auxiliary element in order to make its use possible? 5. Could you restate the problem? Could you restate it still differently? Go back to definitions. 6. If you cannot solve the proposed problem try to solve first some related problem.

1.3 Carrying Out the Plan

1. Carrying out your plan of choice, check each step. 2. Can you see clearly that each step is correct? 3. Can you prove it is correct?

1.4 Looking Back

1. Can you check the result? 2. Can you derive the result differently? 3. Can you use the result for another problem?

1.5 Good Luck

Always believe you can solve the problem.

2 Contains Duplicates

```python
    def containsDuplicate(self, nums: List[int]) -> bool:
        myset = set()

        for num in nums:
            if num in myset:
                return True
            myset.add(num)

        return False
```

2.1 Complexity Analysis

The time complexity of this algorithm is O(n), where n is the number of elements in the list. This is because each element is visited once, and set operations (checking existence and adding a new element) have an average-case time complexity of O(1). The space complexity is also O(n) in the worst case, as in the worst scenario, the set might need to store every element from the list if there are no duplicates.

3 Valid Anagram

```python
    def isAnagram(self, s: str, t: str) -> bool:
        if len(s) != len(t):
            return False

        scount = {}
        tcount = {}

        for i in range(len(s)):
            scount[s[i]] = scount.get(s[i],0) + 1
            tcount[t[i]] = tcount.get(t[i],0) + 1

        return scount == tcount
```

3.1 Complexity Analysis

The time complexity of this algorithm is O(n) as we iterate through the string and perform constant time operations such as adding to a hashmap. The space complexity is also O(n) as we have to store every element from the strings to the hashmaps.

4 Two Sum Problem

```python
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        pos = {}

        for i in range(len(nums)):
```

```
            temp = target - nums[i]
            if temp in pos:
                return [i,pos[temp]]

            pos[nums[i]] = i
```

4.1 Complexity Analysis

The time complexity of this algorithm is O(n) as we are iterating through the string and performing constant time operations such as adding to a hashmap while we are doing that. The space complexity is O(n) as in the worst case, we might have to add every single number and still not find our pair.

5 Group Anagrams

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

        groups = defaultdict(list) #sets every value to empty list

        for i in strs:
            identifier = [0] * 26
            for j in i:
                identifier[ord(j) - ord('a')] += 1

            groups[tuple(identifier)].append(i)

        return groups.values()
```

5.1 Complexity Analysis

The main for loop will run n times where n is the size of the list strs. There is also a for loop in the inside which will run k times where k is the average length of the words in strs. As a result, our overall time complexity is O(n * k). The space complexity on the other hand is O(n * k) as we not only have to store n strings but each of these strings have average length k. Strings are not like ints which can be stored simply.

6 Top K Frequent Elements

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

        count = {}
        for i, e in enumerate(nums):
            count[e] = count.get(e, 0) + 1

        freq = [[] for _ in range(len(nums))] # not [[]] * len(nums)
        for i in count:
            freq[count[i] - 1].append(i)
```

```
10
11          res = []
12          i = len(freq) - 1
13          while len(res) != k:
14              res += freq[i]
15              i -= 1
16
17          return res
```

6.1 Complexity Analysis

The process of scanning through the the list itself is an O(n) process and then we do a bucket sort and have another for loop which are also worst case O(n). As a result, our overall time complexity is O(n). Our space complexity is also O(n) as we have to create a hashmap which will hold n elements in the worst case where n is the size of the list nums.

7 Product of Array Expect Self

```
1       def productExceptSelf(self, nums: List[int]) -> List[int]:
2
3           res = [1] * len(nums)
4           prefix,postfix = 1,1
5
6           for i in range(len(nums)):
7               res[i] *= prefix
8               prefix *= nums[i]
9               res[len(nums) - 1 - i] *= postfix
10              postfix *= nums[len(nums) - 1 - i]
11
12          return res
```

7.1 Complexity Analysis

The time complexity of this solution is O(n) which is because we have to loop through nums while performing constant time operations such as multiplication. The space complexity ignoring the output array is O(1) as we only have the variables prefix and postfix regardless of the size of n.

8 Valid Sudoku Board

```
1       def isValidSudoku(self, board: List[List[str]]) -> bool:
2           iden = set()
3           for i in range(9):
4               for j in range(9):
5                   element = board[i][j]
6                   if element != ".":
```

```
                    if any(((i, element) in iden, (element, j) in iden, (
    i // 3, j // 3, element) in iden)):
                        return False
                    iden.update({(i, element), (element, j), (i // 3, j
    // 3, element)})
        return True
```

8.1 Complexity Analysis

This is a constant time and memory algorithm as the size of the sudoku board never changes.

9 Longest Consecutive Sequence

```
    def longestConsecutive(self, nums: List[int]) -> int:

        elements = set()
        for i in nums:
            elements.add(i)

        longest = 0
        for i in elements: #Careful: big diff b/w nums & elements here
            if i - 1 not in elements:
                length = 0
                while i + length in elements:
                    length += 1
                longest = max(length, longest)

        return longest
```

9.1 Complexity Analysis

The time complexity of this algorithm is O(n) since we have to add n elements to our set. For our second loop, while it looks complicated we actually still have a time complexity of O(n) as there's only so many chains that could exist. Our space complexity for this problem is also O(n) as we have to create a set which holds all of our elements.

10 Is Valid Palindrome

```
    def isPalindrome(self, s: str) -> bool:
        l, r = 0, len(s) - 1
        while l < r:
            while l < r and not s[l].isalnum():
                l += 1
            while l < r and not s[r].isalnum():
                r -= 1

```

```
            if s[l].lower() != s[r].lower():
                return False

            l += 1
            r -= 1

        return True
```

10.1 Complexity Analysis

This algorithm utilizes the two pointers strategy which is typically O(n) using O(1) space. The max our pointers can move is a total of n elements before they meet. We also only have to store the information about our two pointers.

11 Two Sum II - Input Array is sorted

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

        l,r = 0, len(numbers) - 1

        while(l < r):
            total = numbers[l] + numbers[r]
            if(total< target):
                l += 1
            elif(total > target):
                r -= 1
            else:
                return [l + 1, r + 1]
```

11.1 Complexity Analysis

We have the two sum problem once again but this time the input array is sorted. Recall that the original two sum problem was solved in O(n) time using O(n) memory. Since the input array is sorted this time, we can put two pointers at the end of the arrays, and take the correct move each time. This will have a time complexity of O(n) and a space complexity of O(1).

12 Three Sum

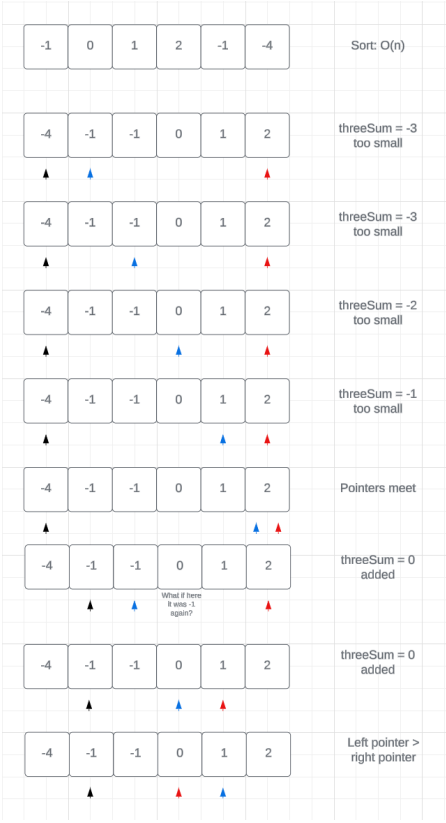
```

def threeSum(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        res = []

        for i, a in enumerate(nums):
            if i > 0 and a == nums[i - 1]:
                continue
```

```
            l, r = i + 1, len(nums) - 1
            while l < r:
                threesum = a + nums[l] + nums[r]
                if threesum > 0:
                    r -= 1
                elif threesum < 0:
                    l += 1
                else:
                    res.append([a, nums[l], nums[r]])
                    l += 1
                    while nums[l] == nums[l - 1] and l < r:
                        l +=1

        return res
```



12.1 Complexity Analysis

Our intuition can lead us to predict that if two sum was O(n), three Sum will probably be O(n squared). However, since we know our time complexity is likely to be O(n squared) we can perform a sort with no cost since it is O(nlog(n)). This will allow us to decrease our memory complexity to O(1). The strategy used is creating a for loop and then breaking the problem down to two sum with sorted inputs again. The hard part is not

having any duplicates. Which we have to make certain moves to avoid. One of these is in the for loop itself where we have to use continue if the index we are on is the same as the previous index. The other is after adding an element if the next element is same as the previous one, it will create a duplicate, so we have to continuously increment the left pointer until it is a fresh value or it is greater than r. We can do something similar for the right pointer, but it is not necessary.

13 Container with Most Water

```
    def maxArea(self, height: List[int]) -> int:

        capacity = 0
        l,r = 0, len(height) - 1
        while l < r:
            amount = (r - l) * min(height[l], height[r])
            capacity = max(capacity, amount)
            if height[r] > height[l]:
                l += 1
            else:
                r -= 1

        return capacity
```

13.1 Complexity Analysis

This is similar to two sum II where we make the localized best choice to try to maximize the amount of rain water. These types of problems are called greedy. Other than that we again used the two pointers methodology which is typically O(n) time complexity and O(1) memory complexity.

14 Trapping Rain Water

```
    def trap(self, height: List[int]) -> int:
        maxright = 0
        maxleft = 0
        capacity = 0

        l,r = 0, len(height) - 1
        while l <= r:
            if maxleft < maxright:
                amount = maxleft - height[l]
                if amount > 0:
                    capacity += amount
                maxleft = max(height[l], maxleft)
                l += 1
            else:
                amount = maxright - height[r]
                if amount > 0:
```

```
                    capacity += amount
                maxright = max(maxright, height[r])
                r -= 1

        return capacity
```

14.1 Complexity Analysis

We are using the two pointers method once again therefore our memory complexity will be O(1) and our time complexity will be O(n). In this problem, however, the pointer strategy is a little bit different. In our usual cases the stop condition for the loop was that l < r, yet in this case it is l <= r. Why is that? Well this time when the pointers meet we cannot immediately kill the loop since we have to analyze the index that it is standing on as well. In previous problems we are considering and comparing two different things when using the pointers in this problem we are looking at space not two different object.

15 Merge Sorted Array

```
def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) ->
    None:
        p1,p2,p3 = m - 1, n - 1, m + n - 1
        while p2 >= 0:
            if p1 >= 0 and nums1[p1] > nums2[p2]:
                nums1[p3] = nums1[p1]
                p1 -= 1
                p3 -= 1
            else:
                nums1[p3] = nums2[p2]
                p2 -= 1
                p3 -= 1
```

15.1 Complexity Analysis

We are again using the two pointers strategy so we can expect a linear time complexity which we do get into this case where our time complexity is O(n + m) since in the worst case both pointers have to go to zero. For space, as usual, the space complexity is O(1).

16 Valid Parenthesis

```
def isValid(self, s: str) -> bool:
        pairs = {"{": "}", "[": "]", "(": ")"}

        stack = []
        for c in s:
            if c in pairs:
```

```
                stack.append(c)
            else:
                if(len(stack) == 0):
                    return False
                if(pairs[stack.pop()] != c):
                    return False

        return len(stack) == 0
```

16.1 Complexity Analysis

This is a stack questions, and surprisingly in python the list can double as a stack. So in this problem we iterated through s and then appended to a stack and we also popped from the end of a stack. Both of these actions are O(1), so overall our runtime is O(n) and our space complexity is O(1) as we had to use a hashmap but the hashmap is the same regardless of our input.