**Question-1 (10 pts): write a Python function that takes in a list of integers and returns maximum and minimum values in the list as a tuple. Note (Your function MUST NOT do more than one pass on the list., you are not allowed to use built-in min and max functions.) What is the time complexity of your function (i.e., Big-oh)?**

**def find\_max\_min(my\_list):**

**#set the first element of the list to be the min and max**

**element = 0**

**min = my\_list[element]**

**max = my\_list[element]**

**for element in range(len(my\_list)): #go through the list**

**if min > my\_list[element]: #if the element of the list is smaller than the min it is the new min**

**min = my\_list[element]**

**elif max < my\_list[element]: #if the element of the list is bigger than the max it is the new max**

**max = my\_list[element]**

**tuple = min,max #make a tuple of the min and max elements**

**return tuple**

**# time complexity: O(n)**

**Question-2 (10 pts): write a Python function that takes in a list of integers (elements), checks whether the list is sorted or not, and returns True of False accordingly. You MUST provide the most efficient algorithm in terms of its time and space complexity. What is the time and space complexity of your algorithm/function?**

**def is\_sorted(my\_list):**

**flag = 0 #we use this to see if list is sorted**

**for element in range(len(my\_list)-1): #we go through the whole list**

**if my\_list[element] <= my\_list[element+1]: #if the elements are sorted we don't flag the list**

**flag = flag + 0**

**else:**

**flag = flag + 1 #if the elements are not sorted we flag that**

**if flag !=0: #if there are flags**

**return False**

**else: #if there are no flags**

**return True**

**# time complexity: O(n)**

**# space complexity: O(1)**

**Question-3 (15 pts): two\_sum is a Python function that takes in a list of integers (elements) and an integer number (num) and returns True if there exist two values in elements that add up to num, otherwise, function returns False.**

**def two\_sum(elements: List[int], num: int):**

**for i in range(len(elements)):**

**for j in range(i + 1):**

**if elements[i] + elements[j] == num:**

**return True**

**return False**

**Study above function and answer below questions:**

**A- What is the time complexity of two\_sum? O(N^2)**

**B- Is it possible to improve on above algorithm performance (in terms of its asymptotic cost)? If yes, describe your algorithm, be detailed as much as possible.**

**Yes it is possible to reduce the O(N^2) time complexity of the current algorithm down to O(NlogN). The new algorithms works as follows: first sort the list (which takes O(NlogN) time); then for each term in the list starting with the first, compute the difference between the intended sum and the term; finally do a binary search (which takes O(logN) time) to see if the difference is there; if it is then we seek to return TRUE, otherwise we continue on; if we never returned TRUE after this process, we return FALSE.**

**Question-4 (10 pts): write a Python function that takes in a list of integers (nums) and finds whether the list consists of any duplicate numbers, the function returns True or False accordingly (Note: this is not a linked list, just a Python list). Function must run in place and cannot use Python’s built-in Set() structure. Make sure to provide the most efficient algorithm in terms of its time and space complexity.**

**There is a way to do this with time complexity O(n) but the space complexity would also be O(n), so I believe this is the most efficient way.**

**def repeated(nums):**

**nums.sort()**

**flag = 0**

**for element in range(len(nums)-1):**

**if nums[element] == nums[element+1]:**

**flag = 1**

**if flag !=0: #if there are flags**

**return True**

**else: #if there are no flags**

**return False**

**# time complexity: n logn**

**# space complexity: O(1)**

**Question-5 (10 pts): Study the following Python function and discuss its time complexity, make sure to account for time-cost of calling built-in python methods.**

| **def fun(nums: List[int]):**  **number\_of\_elements = len(nums)**  **for index in range(number\_of\_elements):**  **value = nums[index]**  **n = nums.count(value)**  **print('{v} appeared {n} times'.format(value, n))** |
| --- |
| **The time complexity is O(n^2). The nums.count goes as O(n), so when it's in a for loop which ranges over n elements, the resulting complexity is O(n^2).** |

**Question-6 (15 pts): study the following Python function carefully and answer the following questions.**

| **def fun(nums: List[int], key):**  **count = 0**    **for v in nums:**  **if v == key**  **count = count + 1**    **return count** |
| --- |

**A- In one statement, describe what the function does.**

**The function takes in a linked hash map/ dictionary / list and a key. It goes through the dictionary looking for an equivalent to the key fed into the function. Everytime that equivalent is found, the counter goes up by one. Then it returns the counter. It counts the number of a specified value in the list.**

**B- Write an equivalent function using recursion.**

**def count (list, index, key):**

**if list[index] == key:**

**if index == len(list)-1:**

**return 1**

**else:**

**return count(list, index+1, key)+1**

**else:**

**if index == len(list)-1:**

**return 0**

**else:**

**return count(list,index+1,key)**

**C- Discuss advantages and disadvantages of both implementations in terms of time and space complexity.**

**They both have the same time and space complexity. Usually, iteration takes less time and space complexity. But recursive code is much simpler and shorter. Sometimes recursion takes up less memory, and it is useful for searching trees (like a directory tree).**

**Question-7 (15 pts): Using a stack, implement a function is\_valid\_expr(expr: str), the function takes in a string representing a mathematical expression, which may include parenthesis, the function returns true if expr is valid, otherwise, it returns false. A valid expression is one that has a matching closing parenthesis for every opening one. Here are some examples:**

**1 + 10 is valid**

**(1+2) is valid**

**(2 \* 3) + (4 \* 2 + (x – 1)) is valid**

**(1 + 1) + (2 \* x)) is invalid**

**((1 + 5) \* x is invalid**

**def is\_valid\_expr(exp):**

**expr = list(exp)#turn string into list**

**stack = [] #using a regular list but acting like its a stack**

**for element in range (len(expr)):#go through the string**

**if expr[element]== "(":**

**stack.append(element) #using append as push**

**if expr[element]==")":**

**try: stack.pop()**

**except: return False**

**if len(stack) == 0:**

**return True**

**if len(stack) != 0:**

**return False**

**Question-8 (15 pts): Considering a singly linked list of integers implementation, assuming that elements of the list are always in ascending order, write a function (find\_median) that finds and returns median value. The function cannot scan the list more than ONCE.**

**class SinglyList:**

**class \_Node:**

**def \_\_init\_\_(self, e, next):**

**self.\_element = e**

**self.\_next = next**

**def \_\_init\_\_(self):**

**self.\_head = self.\_tail = None**

**def find\_median(self):**

**#assume we have a sorted list (list in ascending order)**

**head, num, mid = self.\_head, self.\_head, self.\_head #start by looking at the beginning**

**while num and num.\_next : #while the current number and the the next number exist**

**num = num.\_next.\_next #look ahead in the list, use this later to check if we need to divide the middle two numbers or return the regular middle number**

**mid = head #change your variable**

**head = head.\_next #set to new middle**

**if num: #if there is an odd amount of element**

**return head.\_element**

**else: #if there is an even**

**return (head.\_element + mid.\_element)/2**