CPSC-34000 Algorithms and Data Structures

Fall 2022 – Midterm Exam

**Question-1 (10%)**: 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)?

max, min = find\_max\_min(my\_list):

max = 0

min = 0

for i in my\_list:

if i > max: max = i

elif i < min: min = i

return {max, min}

*Time complexity is O(n).*

**Question-2 (10%)**: Write a Python function that takes in a list of integers (elements), checks whether the list is in descending order 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\_ descending (my\_list):

if my\_list[0] > my\_list[1]:

return False

else:

return True

*Time complexity is simply n = 2*

**Question-3 (15%)**: 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:

1. What is time complexity of two\_sum?
   1. LogO(n)
2. 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.
   1. *The only way would be to not have two variables (I and j) but reduce them down to just 1 which would improve the asymptotic cost. This would have to be down in a while loop with just one counter that continuesly loops through the array trying to add the two numbers together until the index reaches the length of the list. Then return as needed. This would make the algorithm slower but would improve the asymptotic cost.*

**Question-4 (10%):** 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.

**def** duplicates(nums: List[int]):

used\_elements = []

for i in nums:

for j in used\_elements:

if i != j:

for k in range (i + 1):

if i == nums[k]:

return True

break

return False

This may not be the most efficient but this is how I would check for duplicates in an array. I can also think of ways that would be faster using recursion that would up the space complexity. Alternatively one could simply use the build in list method **count** which would return the number of elements that match a value. So if it ever returns more than 1 then there would be duplicates.

**Question-5 (10%)**: 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))

*Time complexity would be O(n) \* n.*

*I get this from accounting for the linear format of the function through looping singularly through an array of N size. However the count function would require further looping of the array as a built in method which adds to the time complexity.*

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

1. In one statement, describe what does the function do.

**def** fun(nums: List[int], key):   
 count = 0

**for** v **in** nums:  
 if v == key  
 count = count + 1

**return** count

* 1. It loops through the list and counts how many of its elements are equal to the key value passed to the function.

1. Write an equivalent function using recursion.

**def fun(key, nums: List[int], count):**

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

**return count**

**if nums[0] == key:**

**count += 1**

**fun(key, nums[1:], count)**

1. Discuss advantages and disadvantages of both implementations in terms of time and space complexity.
   1. The complexity goes up with the recursion but it saves and improves on the time complexity allowing multiple iteration checks at once instead of 1 at a time. This forces the space complexity to go up though which with a massive list could theoretically break the program. The non recursion way though is slower.

**Question-7 (10%):** 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 🡪 valid

(1+2) 🡪 valid

(2 \* 3) + (4 \* 2 + (x – 1)) 🡪 valid

(1 + 1) + (2 \* x)) 🡪 invalid

((1 + 5) \* x 🡪 invalid

def is\_valid\_expr(expr: str):

op = Stack()

num = Stack()

par\_open = Stack()

par\_close = Stack()

for i in str:

if i.isnumeric() or i == “x”:

num.push(i)

if num.size() > 2:

return False

elif num.size() = 2:

if op.size() != 0:

op.pop()

num.pop()

else:

return False

elif i == “+” or i == “-“ or i == “\*” or i == “/”:

op.push(i)

if op.size() > 1:

return False

elif i == “(“ or i == “)”:

if “)”:

par\_close.push(i)

if par\_open.size() < par\_close.size():

return False

else:

par\_open.push(i)

else:

return False

if par\_close.size() != par\_open.size():

return False

if op.size() != 0 || num.size() != 1:

return False

return True

**Question-8 (10%):** 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):

if self.head == None: return 0

if self.head.next == None: return self.head.element

values = []

count = 0

current = self.head

while current != None:

values.push(current.element)

count += 1

current = current.next

if count % 2 == 0:

i = count / 2

j = i – 1

return value[i] – values[j]

else:

i = count % 2

return values[i]

**Question-9 (10%):** Write a Python function that will return True if A phrase is a palindrome and False otherwise. A phrase is palindrome if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string s, return true if it is a palindrome, or false otherwise.

**Ex 1:** Input: s = "A man, a plan, a canal: Panama"

Output: true

Explanation: "amanaplanacanalpanama" is a palindrome.

**Ex.2:** Input: s = "race a car"

Output: false

Explanation: "raceacar" is not a palindrome.

**Ex.3:** Input: s = " "

Output: true

Explanation: s is an empty string "" after removing non-alphanumeric characters.

Since an empty string reads the same forward and backward, it is a palindrome.

**class** Solution:

**def** isPalindrome(self, s: str) -> bool:

first = “”

second = “”

for i in str:

if i.isalpha():

first.push(i)

second.insert(0,i)

if first == “” or first == second:

return True

**return** False

**Question-10 (10%):** Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

**Ex 1:** Input: nums = [1,3,5,6], target = 5

Output: 2

**Ex.2:** Input: nums = [1,3,5,6], target = 2

Output: 1

**Ex.3:** Input: nums = [1,3,5,6], target = 7

Output: 4

**class** Solution:

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

if len(nums) == 0: return 0

for i in range(len(nums)):

if i == 0:

if nums[i] == target:

return i

elif nums[i+1] > target:

return I + 1

elif:

if nums[i] == target:

return i

elif nums[i-1] < target and nums[i+1] > targer:

return I + 1