GitHub Username:

MY470 Computer Programming

Mock Problem Set 5, AT 2023

Practicing order of growth analysis for time complexity

This assessment takes the form of a more traditional problem set, where each problem stands by itself and is unrelated to the others.

You are expected to complete the problems on your own.

You have 25 minutes to complete the problem set.

Instructions for Problems 1–5

Give the order of growth for the function and explain your reasoning in a couple of sentences.

Problem 1

In [1]:
```python
def sum_array(array):
    """Calculate the sum of all elements in a 3-dimensional array
    (list of lists of lists)."""

    total = 0 # O(1)
    for layer in array: # O(l)
        for row in layer: # O(r)
            for element in row: # O(c)
                total += element # O(1)
    return total

# Your answer: O(lrc)
#
# number of layers = l
# number of rows = r
# number of columns (elements in each row) = c
# It takes l*r*c steps to loop over all elements
# Time complexity: O(1 + l * r * c * 1)
# = O(lrc)
#
#
```

Problem 2

In [ ]:
```python
def is_power_of_two(n):
    """Check if an integer is a power of two.
    (i.e., 1, 2, 4, 8, 16, 32, 64, etc.)"""

    while n > 1: # How many times will this while loop run?
        n = n / 2 # O(1)

    if n == 1: # O(1)
        return True
    else:
        return False

# Your answer: O(log(n))
#
# The while loop will run as many times as it takes to get n <= 1.
# Each step, n is divided by 2.
# So, the number of times the loop runs is the number of times n
# can be divided by 2 before it is <= 1.
# So 2^x ~ n, where x is the number of times the loop runs.
# Solve for x: x = log2(n), (log base 2 of n)
# Time complexity is then: O(log(n))
```

Problem 3

In [ ]:
```
def is_sorted(int_list):
    """Check if a list of ints is sorted."""
    sorted_list = sorted(int_list) # Sorting ints: O(l * log(l))
    if sorted_list == int_list: # Check all elements the same: O(l)
        return True
    return False
# Your answer: = O(l * log(l))
# length of int_list = l
# Time complexity from sorting and checking: O(l * log(l) + l)
# = O(l * log(l))
```

Problem 4

In [1]:
```
def is_sorted2(string_list):
    """Check if a list of strings is sorted
    (different approach to P3)."""
    for i in range(len(string_list) - 1): # iterating: O(l)
        # comparing strings requires checking every character!
        if string_list[i] > string_list[i + 1]: # O(s)
            return False
    return True
# Your answer: O(ls)
# length of string_list = l
# length of each string = s
# "worst case scenario" is that the list is sorted, so we have to
# check every string in the list, and every char in each string.
# Time complexity: O(ls)
```

Problem 5

In [2]:
```
def sum_nd_array(ndarray):
    """Calculate the sum of all elements in a numeric
    n-dimensional array. Each dimension is of the same size "d"."""
    if isinstance(ndarray, (int, float)): # O(1)
        return ndarray
    total = 0 # O(1)
    for item in ndarray: # O(d)
        # Addition = O(1), but how many recursive calls are made?
        total += sum_nd_array(item)
    return total
# Your answer: O(d^n)
# Number of dimensions = n, Size of each dimension = d
# For each extra dimension, make d times more recursive calls
# e.g. 1 dimension: d recursive calls
#      2 dimensions: d * d recursive calls
#      3 dimensions: d * d * d recursive calls
#      ...
# So, for n dimensions, d^n recursive calls are made = O(d^n)
# Or, there are d*d...d = d^n total elements to sum = O(d^n)
```

Instructions for Problems 6–7

Write the function as described in the docstring and called under. Then, give the order of growth for the function and explain your reasoning in a couple of sentences.

Problem 6

In [3]:
```
def fizz(n):
    """Print out all numbers from 1 to n (inclusive),
    replacing multiples of 7 with "Fizz"."""
    # complete function here

    for i in range(1, n + 1): # iteration: O(n)
        if i % 7 == 0: # O(1)
            print('Fizz') # O(1)
        else:
            print(i) # O(1)

fizz(16)
# Your answer for order of growth: O(n)
#
# Size of max number to iterate to = n
# Time complexity = O(n)
#
#
#
#
#
```

```
1
2
3
4
5
6
Fizz
8
9
10
11
12
13
Fizz
15
16
```

Problem 7

In [1]:
```
def three_char_words(l1, l2, l3):
    """Given three lists, first identify the strings in each
    list. Assume all strings are a single character and return
    a list of all possible three-character "words" that can be
    made from combining one character from each list (allowing
    for repeats). You will need to remove any non-strings
    (ints, floats, etc.) in advance, but do not alter the
    case of any letters or remove non-letters.
    """
    filtered_l1 = [char for char in l1 if type(char)==str] # O(l1)
    filtered_l2 = [char for char in l2 if type(char)==str] # O(l2)
    filtered_l3 = [char for char in l3 if type(char)==str] # O(l3)

    word_list = [] # O(1)
    for char1 in filtered_l1: # O(l1)
        for char2 in filtered_l2: # O(l2)
            for char3 in filtered_l3: # O(l3)
                word_list.append(char1 + char2 + char3) # O(1)
                word_list.append(char1 + char3 + char2) # O(1)
                word_list.append(char2 + char1 + char3) # O(1)
                word_list.append(char2 + char3 + char1) # O(1)
                word_list.append(char3 + char1 + char2) # O(1)
                word_list.append(char3 + char2 + char1) # O(1)

    return word_list


l1 = ['a']
l2 = ['?', 1, 'C']
l3 = ['c', 'a']
ls = three_char_words(l1, l2, l3)
print(ls)
# Your answer for order of growth: O(l1*l2*l3)
#
# length of l1 = l1
# length of l2 = l2
# length of l3 = l3
# Worst case scenario is that all characters are letters, so
# filtered lists are the same length as original lists.
# To create every combo, we need to loop over l1*l2*l3 triples
# Time complexity:
# O(l1 + l2 + l3 + l1*l2*l3*6)
# O(l1*l2*l3)
```

```
['a?c', 'ac?', '?ac', '?ca', 'ca?', 'c?a', 'a?a', 'aa?', '?aa', '?
aa', 'aa?', 'a?a', 'aCc', 'acC', 'Cac', 'Cca', 'caC', 'cCa', 'aCa
', 'aaC', 'Caa', 'Caa', 'aaC', 'aCa']
```