# COMP 1405Z – Fall 2023
# Introduction to Computer Science I
# Final Exam – Sunday, October 29th 2023

## Exam Details

The exam will be submitted through Brightspace. Put your solutions to each of the problems in the corresponding .py file from the provided 1405-exam-materials.zip file. To submit your exam solutions, create a single .zip file containing all of your exam materials (including all test resources) and submit it to the '1405 Exam' assignment submission on Brightspace. The official end time for the exam is 12:00pm (Eastern Time) but you will have until 12:15pm (Eastern Time) to submit your exam. **Do not wait until the last minute to submit the exam. Brightspace's clock may not be exactly synchronized with your own and no late exam submissions will be accepted**. After submitting and before the submission cut-off, you should download your own zip file, extract its contents, and ensure the files have the correct content.

The exam is divided into 3 sections consisting of 3 questions each: Section #1 (Problems 1-3), Section #2 (Problems 4-6), and Section #3 (Problems 7-9). Each problem will be marked out of 10. Your mark for each section of the exam will be calculated using the best two marks on problems within that section, for a total of 60 marks. For example, if you get 8/10, 10/10, and 3/10 on Problems #1-3, your mark for Section #1 of the exam will be 18/20.

You are not allowed to consult other students, TAs, or anybody else while completing the exam. You may use any IDE or text editor you generally use to write your code. **You may not use any additional modules for any problems**. **You may not use the built-in sorting functionality for any problems. You may use any other standard Python code you want, unless the question specifically mentions that you cannot**. You can execute your programs as many times as you want to debug and fix your code. **The solutions you create will be evaluated based on correctness, efficiency, and overall code quality.** Your code should use quality variable names, make effective use of functions, be well organized and easy to read. You should also try to minimize the computational/space complexity for each of your solutions.

## Problem 1 (10 marks)

**Write the code for this part in the problem1.py file**. Write a function called `median_word_length` that takes a single string input argument representing a file name. This function must read the file with the given name and return the median word length of all words within the file. This program should handle the case where there are 2 median/middle word lengths by returning the average of the two. You can assume that a space separates each word in the file. As an example, the median word length for the provided `problem1-example-input.txt` file would be 3.5.

## Problem 2 (10 marks)

**Write the code for this part in the problem2.py file**. For this problem, you will be working with strings stored as lists. For example, the string "`hello`" would be stored as a list in the form `["h", "e", "l", "l", "o"]`. You must work with the list data and cannot transform it into another type of data (e.g., string). You must implement a `count(list1, list2)` function. This function must return the number of times that the string represented by `list2` occurs in `list1`. For example, if `list1` were `["b", "a", "n", "a", "n", "a", "s"]` and `list2` were `["a", "n"]`, `count(list1, list2)` would return 2. If `list1` were `["b", "a", "a", "a", "h"]` and `list2` were `["a", "a"]`, `count(list1, list2)` would also return 2.

## Problem 3 (10 marks)

**Write the code for this part in the problem3.py file**. For this problem, you will be writing code that will compute probability information. Your code will remember information about numbers as they are observed and will be able to return the probability of any given number based on the current data. Remember that the probability of X is calculated as:

(# of times X has been observed) / (total number of observations)

You must implement the following three functions in the problem3.py file:

1. `observe(int)` – Accepts a single integer value and updates the state of your program as required (you may use global variables to store the state).
2. `probability_of(int)` – Accepts a single integer and outputs the probability in decimal form of the given integer based on the current data.
3. `reset()` – Resets all data.

For example, if `reset()` is called and then the `observe` function is called ten times with the integers 0, 1, 0, 1, 0, 2, 0, 2, 1, and 0, then:

`probability_of(0)` → 0.5
`probability_of(1)` → 0.3
`probability_of(2)` → 0.2
`probability_of(3)` → 0.0

The provided problem3-tester.py file can be used to perform some basic tests on your code. It is advised that you test your code further with some of your own examples. **For full marks, your observe and probability_of functions must run in O(1) time.**

## Problem 4 (10 marks)

**Write the code for this part in the problem4.py file**. For this problem, you will be working with data from an auto show. The first line of the file will contain the auto show name ("AutoRama 2023" in the included autoshow-example.txt). The rest of the file will contain five lines of data for each car. In order, these lines represent the make (string), model (string), color (string), price (integer), and 'for sale' value (either "true" or "false") of that car. You must write the following functions:

1. `most_common_color(filename)` – Accepts a filename and returns the most common car color found in the given file. The most common color for the provided file should be "Green".
2. `sorted_prices(filename)` – Accepts a filename and returns a sorted list of the prices of cars for sale. Cars that are not listed for sale (i.e., have a for sale value of "false") should not be included in the result. The list should be sorted from highest sale price to lowest. The expected output for the included example file is [260, 220, 180, 180, 160, 160, 110, 10]. **You may not use the built-in sorting functionality Python provides for this problem.**

## Problem 5 (10 marks)

**Write the code for this part in the problem5.py file**. For this problem, you must work with the list and integer data provided. You cannot convert the list to another type (e.g., a string). In this problem, you will be working with a list representation of the contents of a box. Within a box list, there will be a mix of integer values and lists. Integer values within a box list represent the weight of an item inside of that box. If one box list (A) contains another list (B) within it, this represents a smaller box (B) within the larger box (A). The boxes may be nested to any depth. You must write a function called `box_weight` that accepts a single box list representation and outputs the total weight of all items contained within the box (including items within inner boxes). For example, the representation below would have a total weight of 50:

```
    [2, 1, [[4], 2, [9, 7]], 2, [1, 3, [1, [2], [3]], [9], [], 4]]
```

Note that to solve this problem you may need a way of determining whether a value is either an integer or a list. You can use the Boolean expression `isinstance(some_value, list)`, which will return True if `some_value` is a list and False otherwise. The problem5-tester.py file can be used to execute your function with some example inputs.

## Problem 6 (10 marks)

**Write the code for this part in the problem6.py file**. For this problem, you must write the following two functions:

1. `process(alist)` – Accepts a list of integer values. Clears any existing data and performs any computation necessary to support the `within_range` function described below. Once the `process` function is called, any subsequent calls for `within_range` should use only `alist`'s data until the `process` function is called again (i.e., you only work with one list's data at a time). You may use global variables to allow data to be accessed by each function.

2. `within_range(a, b)` – Returns the number of values in the list that are within the range from `a` to `b` (inclusive). You may assume the value of `a` will always be less than or equal to the value of `b`. For example, if the initial list was `[1, `**`3`**`, `**`4`**`, 2, `**`4`**`, 2, `**`3`**`, `**`5`**`, 1, `**`4`**`]` and the values `a=3` and `b=5` were input, the function would return 6 (the bolded numbers are within the given range).

For full marks, you should optimize the runtime complexity of both functions. The `process` function can be implemented in O(n) time, where n is the number of elements in `alist`. It is possible to implement an O(k) solution for `within_range`, where k is the size of the range given as input. You can assume the list will contain only integer values, but cannot assume anything else about the list (size, order, min/max values, etc.). The problem6-tester.py file can be used to execute some basic tests on your code.

## Problem 7 (10 marks)

**Write the code for this part in the problem7.py file**. For this problem, you must write a function called `coldest_interval` that will accept a list and an integer (X) as input. The list input will contain integer values representing daily temperature measurements. Your function must find the X-day interval (i.e., X sequential entries in the list) in which the average temperature was coldest and return the average temperature for that interval. You may assume that 1 <= X <= N, where N is the length of the list. The problem7-tester.py file can be used to test your function with some example data.

## Problem 8 (10 marks)

**Write the code for this part in the problem8.py file**. For this problem, you must write a function called `find_largest_distance` that will accept a string representing a filename as input. You may assume the filename represents a file that contains city location data in the following format:

```
City #1 Name
City #1 X Location
City #1 Y Location
City #2 Name
City #2 X Location
City #2 Y Location
City #3 Name
City #3 X Location
City #3 Y Location
…and so on…
```

Within the file, the city names will be strings and the X/Y locations will be integers. Your function must read the given file and return the largest distance between any two cities contained in the file. Distance should be calculated using the Manhattan distance. The Manhattan distance can be calculated using the equation below, where |x| represents the absolute value of x. That is, the magnitude of x regardless of whether it is positive or negative (e.g., |5| = 5 and |-5| = 5):

$$manhattan\_distance = |x1 - x2| + |y1 - y2|$$

The problem8-tester.py file can be used to test your function with some example data.

## Problem 9 (10 marks)

**Write the code for this part in the problem9.py file**. For this problem, you must write a function called `sort_volumes` that will accept a string representing a filename as input. You may assume the filename represents a file that contains box information in the following format:

Box #1 ID
Box #1 X Dimension
Box #1 Y Dimension
Box #1 Z Dimension
Box #2 ID
Box #2 X Dimension
Box #2 Y Dimension
Box #2 Z Dimension
Box #3 ID
Box #3 X Dimension
Box #3 Y Dimension
Box #3 Z Dimension
…and so on…

Your function must read the given file and return a list containing each box ID. The ID entries in the returned list must be sorted from the box with highest volume to the box with lowest volume. You can assume the ID values and dimensions are all integers. As an example, the list [3, 1, 2] would be returned if there were 3 boxes with the following IDs and volumes:

ID = 1, Volume = 12
ID = 2, Volume = 9
ID = 3, Volume = 18

The volume of a box can be calculated by multiplying the X, Y, and Z dimensions together. The problem9-tester.py file can be used to test your function with some example data.