University of Waterloo
Midterm Examination
Winter 2017

Student Name:	______________________
Student ID Number:	______________________
Course Section (circle one):	**BME 122 | MTE 140**

Instructors:	Alexander Wong and Igor Ivkovic
Date of Exam	February 16th 2017
Time Period	2:30pm-4:20pm
Duration of Exam	110 minutes
Pages (including cover)	12 pages
Exam Type	Closed Book

NOTE: No calculators or other electronic devices are allowed. Do not leave the exam room during the first 30 minutes and the last 10 minutes of the exam. Plan your time wisely to finish the exam on time.

Question 1: (10 marks)	Question 2: (14 marks)	Question 3: (12 marks)	Question 4: (14 marks)
Total: (50 marks)			

Useful Formulas

- For $S = a_1 + (a_1 + d) + ... + (a_n - d) + a_n$
  (S is a series that goes from a_1 to a_n in d-size increments),

$$S = n\left(\frac{a_1 + a_n}{2}\right) \tag{1}$$

- $$\sum_{i=k}^{n} 1 = (n - k + 1) \tag{2}$$

- $$\sum_{i=1}^{n} i = \left(\frac{n(n+1)}{2}\right) \tag{3}$$

- $$\sum_{i=1}^{n} i^2 = \left(\frac{n(n+1)(2n+1)}{6}\right) \tag{4}$$

- $$\sum_{i=0}^{n} r^i = \left(\frac{r^{n+1} - 1}{r - 1}\right) \tag{5}$$

- $$\log_b x = \frac{\log_c x}{\log_c b} \tag{6}$$

1 Question 1. Algorithm Complexity (10 marks)

(approx. 20 minutes)

a. (1 mark)

How would you perform a swap of two integer values, a and b, without using a temporary variable? Circle the correct answer.

1. a = a + b; b = a - 2b; a = b - a;
2. a = 2a + b; b = a - 2b; a = b - a;
3. a = a + b; b = a - b; a = a - b;
4. None of the above

b. (1 mark)

Based on the algorithms studied in class, provide a code fragment in pseudocode that would exhibit the runtime of $O(n^3 log^2(n))$. For instance, for O(n) runtime, you could write the following:

```
loop from 1 to n {
    call swap(a, b) that performs constant number of steps
}
```

c. (8 marks)

Let the runtime efficiency of an algorithm be defined as: $T(n) = 14log_2^2(n) + 56log_2^2(n) + 126log_2^2(n) + ... + 14(n-1)^2log_2^2(n) + 14n^2log_2^2(n)$. Prove that $T(n)$ is $O(n^3log^2(n))$ using the Big-O formal definition or by using limits as $n \to \infty$. Show all steps in your proof.

2 Question 2. Algorithm Design (14 marks)

(approx. 35 minutes)

a. (10 marks)

You are given two unsorted arrays of integers, A and B. Each array stores a random number of integers drawn from the $[0, 999]$ range, inclusive of the boundary values. There are no duplicates inside A or B, respectively. Write a function

```
void find_repeated_values(int* A, int size_A, int* B, int size_B)
```

that will iterate through the two arrays, check to see what numbers from A are repeated in B, and then output on separate lines the repeated numbers and the "total count" of repeated numbers. For instance, if A equals $[42, 1, 57, 5, 317]$ and B equals $[5, 57, 11, 111, 256, 317]$, your function should output the following in no particular order:

```
5
57
317
total count: 3
```

For full marks, the run-time performance of your algorithm needs to be linear (i.e., $O(n+m)$ where n is the size of one array and m is the size of the other array), and the memory performance has to be constant.

For partial marks, the run-time performance of your algorithm can be worse than linear (e.g., $O(nm)$), and you may use more than constant memory space. [The amount of partial marks awarded will be decided when grading.]

b. (4 marks)

Once you have designed the algorithm, explain in natural language — not code — how would you test the algorithm. That is, list and briefly explain at least ten test cases that you would use to ensure correctness of your algorithm and enhance its robustness. For example, the test case given above could be explained as ”test non-empty arrays of size 5 and 6 where 3 values are repeated between A and B”.

3 Question 3. Divide and Conquer Algorithms (12 marks)

(approx. 20 minutes)
Consider the following function:

```
void woofNG(int K, int firstWoof, int lastWoof) {
    int segment = (lastWoof - firstWoof) / 3;
    if (lastWoof <= firstWoof) {
        cout << "WOOF Complete!" << endl;
        return;
    }

    for (int i = firstWoof; i <= lastWoof; ++i) {
        cout << "WOOF ";
    }
    cout << endl;

    if (K <= firstWoof + segment)
        woofNG(K, firstWoof, firstWoof + segment);
    else if (K <= firstWoof + segment * 2)
        woofNG(K, firstWoof + segment + 1, firstWoof + segment * 2);
    else
        woofNG(K, firstWoof + segment * 2 + 1, lastWoof);
}
```

a. (3 marks)

Draw the call tree for this function when $woofNG(42, 11, 85)$ is called.

b. (3 marks)

Determine the recurrence relationship for the runtime efficiency $T(n)$ of this function, where $n = lastWoof - firstWoof$ for $n \geq 0$.

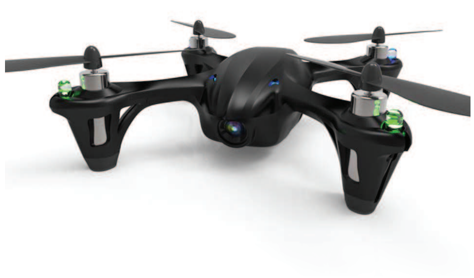
b. (6 marks)

Solve the recurrence relationship for this function by unrolling the recurrence (use back substitution), and give the $T(n)$'s order of growth in terms of the Big-O notation as a function of n. You do not have to provide a formal proof using the Big-O formal definition. Show all steps in deriving your solution.

4 Question 4. Data Structure Design (14 marks)

(approx. 35 minutes)

Our company is in a possession of a small fleet of drones that we would like to make available to customers. We are tasked with programming a data structure for storage of drone records.



Each drone will be recorded as an instance of *DroneRecord* class. Each record has a unique drone ID, which is stored as an unsigned integer. In addition, each *DroneRecord* also stores drone type, manufacturer, description, range, battery type, and year bought; range and year bought are to be stored as unsigned integers while all others are to be stored as *string* values.

DronesManager class will be used as a container to handle *DroneRecord* objects, and it will include relevant attributes/fields. *DronesManager* needs to provide fast access to individual drone records based on their location in the collection (index), and it needs to keep *DroneRecord* objects sorted by drone IDs in ascending order.

a. (4 marks) Using sequential list from *Assignment* #1 as the basis for your implementation, provide the declarations for member attributes and methods, including getter/setter methods, that are needed to make *DroneRecord* and *DronesManager* function. The capacity for *DronesManager* should be 1024, and once that capacity is reached, no more drones can be stored. For *DroneRecord*, you should also include at least one parametric constructor.

```
class DroneRecord {
    unsigned int droneID;
    // fill in other required member attributes and methods below
    // members should be declared as public or private as appropriate
```

```
};

class DronesManager {
// fill in other required member attributes and methods below
// members should be declared as public or private as appropriate
```

```
    bool insertDrone(DroneRecord* record);
    bool removeDrone(unsigned int droneID);
    DroneRecord* findDrone(unsigned int droneID);
    DroneRecord* findDroneByIdx(unsigned int idx);
};
```

b. (3 marks) Implement one of the constructors and the destructor for the $DronesManager$ class. Include appropriate initialization and deinitialization of member variables, respectively.

c. (7 marks) Write the method $DronesManager :: removeDrone(unsigned\ int\ droneID)$. The method removes the drone with the given drone ID. After the drone record is removed, other records are adjusted so that there is no fragmentation and the list remains sorted by drone ID. If the drone ID is not found, the method returns $false$. Otherwise, if the record is found and the removal succeeds, the method returns $true$.

You may only use *iostream* and *string* libraries in your implementation, and no other external libraries. However, you may write helper functions of your own. Include appropriate error checking in your code, and adequately document your code. Marks will be deducted for implementations that are difficult to read, or that are not adequately documented.

```
bool DronesManager::removeDrone(unsigned int droneID){
```