University of Waterloo
Midterm Examination Sample Solutions
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

Solution:

Answer: (3) a = a + b; b = a - b; a = a - b;

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
}
```

Solution:

```
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= n; ++j) {
        for (int k = 1; k <= n; ++k) {
            for (int l = 1; l < n; l = l * 2) {
                for (int m = 1; m < n; m = m * 2) {
                    // constant number of steps
                }
            }
        }
    }
}
```

Grading scheme: 0.5 marks for three outer loops for $O(n^3)$, and 0.5 marks for two inner loops for $O(log^2(n))$.

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.

Solution:

$$\begin{aligned} T(n) = 14log_2^2(n) + 56log_2^2(n) + 126log_2^2(n) + ... + 14(n-1)^2log_2^2(n) + 14n^2log_2^2(n) = \\ 14log_2^2(n)\sum_{i=1}^{n} i^2 = \\ 14log_2^2(n)\frac{n(n+1)(2n+1)}{6} = \\ \frac{14}{6}log_2^2(n)(n^2+n)(2n+1) = \\ \frac{14}{6}log_2^2(n)(2n^3+3n^2+n) \end{aligned} \quad (7)$$

(Option A.)

We must show that $\frac{14}{6}log_2^2(n)(2n^3+3n^2+n) \leq Kn^3log_2^2(n)$ for all $n >= n_0$.

Let $n_0 = 2$. It follows that $\frac{14}{6}(16+12+2) = \frac{14}{6}(30) \leq 8K$.

From there, $\frac{35}{4} \leq K$.

Hence, for $K = 9$ and $n \geq n_0 = 2$, it follows that $T(n) = O(n^3log_2^2(n)) = O(n^3log^2(n))$.

(Option B.)

$\lim_{n\to\infty} \frac{\frac{14}{6}log_2^2(n)(2n^3+3n^2+n)}{n^3log_2^2(n)} = \lim_{n\to\infty} \frac{14}{6}(2 + \frac{3}{n} + \frac{1}{n^2}) = \lim_{n\to\infty} \frac{28}{6} = constant.$

Hence, it follows that $T(n) = O(n^3log_2^2(n)) = O(n^3log^2(n))$.

Grading scheme: 2 marks for deriving $14log_2^2(n)\sum_{i=1}^{n} i^2$, 2 marks for deriving $\frac{14}{6}log_2^2(n)(2n^3+3n^2+n)$; 3 marks for deriving K and n_0, or computing the constant using limits; and 1 mark for the correct final answer.

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.]

Solution:

```
void find_repeated_value(int* A, int size_A, int* B, int size_B) {
    bool found_in_A[1000] = {0}; // create an array of 1000 booleans, all false

    for (int i = 0; i < size_A; ++i) {
        unsigned int val_A = A[i]; // grab the i-th value in A
        found_in_A[val_A] = true; // set the A[i]-th value to true
    }

    int total_count = 0; // store the count of repeated values

    for (int i = 0; i < size_B; ++i) {
        unsigned int val_B = B[i]; // grab the i-th value in B
        if (found_in_A[val_B]) { // if there is a match, output val_b
            cout << val_B << endl;
            ++total_count; // increment total count
        }
    }

    cout << "total count: " << total_count << endl;  // output total count
}

int main() {
    int A[5] = {42, 1, 57, 5, 317};
    int B[6] = {5, 57, 11, 111, 256, 317};
    find_repeated_value(A, 5, B, 6);
}

/* Output:
5
57
317
total count: 3
*/
```

Grading scheme: 2 marks for correctly declaring the boolean array, 3 marks for the first loop, 4 marks for the second loop including output, 1 marks for total count updating and output. Award up to 5 marks for the solutions that are worse than O(n + m). Subtract marks for solutions that are very hard to read or understand, or that do not provide all the required code steps.

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".

Solution:

```
[Test for A or B being NULL; 1 mark]
1. A is NULL, size_A is 0, B is NULL, size_B is 0
[Test for cases where the array size and size_A/_B values do not match; 0.5 mark]
2. A is NULL, size_A is not 0, B is NULL, size_B is not 0
3. A is not NULL, size_A is 0, B is not NULL, size_B is 0
4. A is not NULL, array size is 7, size_A is 5,
5. B is not NULL, array size is 11, size_A is 15

[Test for A or B having duplicate values internally; 0.5 mark]
6. A is not empty, size_A is 5, A contains some duplicates
7. B is not empty, size_B is 7, B contains all duplicates

[Test for A or B including values outside the [0,999] range; 0.5 mark]
8. A is not empty, size_A is 11, A contains values outside the range
9. B is not empty, size_A is 15, B contains values outside the range

[Test for A or B being empty; 0.5 mark]
10. A is not NULL and empty, size_A is 0, B is not NULL and empty, size_B is 0
11. A is not NULL and empty, size_A is 0, B is not empty, size_B is 5
12. A is not empty, size_A is 3, B is not NULL and empty, size_B is 0

[Test for A and B not empty with no/some/all repetition; 1 mark]
13. A is not empty, size_A is 5, B is not empty, size_B is 5, 0 repetitions
14. A is not empty, size_A is 5, B is not empty, size_B is 5, 5 (all) repetitions
15. A is not empty, size_A is 3000, B is not empty, size_B is 50, 5 repetitions
16. A is not empty, size_A is 35, B is not empty, size_B is 500, 15 repetitions

[Other equivalence classes may be accepted as appropriate]
```

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.

```
woofNG(42, 11, 85)
woofNG(42, 36, 59)
woofNG(42, 36, 43)
woofNG(42, 41, 43)
woofNG(42, 42, 43)
woofNG(42, 42, 42)
```

Grading scheme: 0.5 mark for each correct entry.

b. (3 marks)
Determine the recurrence relationship for the runtime efficiency $T(n)$ of this function, where $n = lastWoof - firstWoof$ for $n \geq 0$.

Solution:
$T(0) = a$
$T(n) = T(n/3) + n + b$
Also acceptable: $T(n) = T(n/3) + (n + 1) + b$
where a is a constant that represents the number of primitive operations used in the base case, and b is a constant that represents the number of primitive operations needed to execute each cycle.
Grading scheme: 1 mark for the base case. 2 marks for the recursive case.

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.

Solution:

$$T(n) = T(n/3) + n + b = T(n/3^2) + n/3 + b + n + b = T(n/3^3) + n/3^2 + b + n/3 + b + n + b = \\ ... = T(n/3^i) + n\sum_{j=0}^{i-1}(1/3)^j + ib = T(n/3^i) + n\frac{(1/3)^i - 1}{1/3 - 1} + ib = T(n/3^i) + n\frac{3(1-(1/3)^i)}{2} + ib \tag{8}$$

When $n/3^i = 1$, let $i = c$. It follows that $n/3^c = 1$ and $n = 3^c$, so $c = \log_3(n)$.

From there,

$$\begin{aligned} T(n) = T(n/3^i) + n\frac{3(1-(1/3)^i)}{2} + ib = T(1) + n\frac{3(1-(1/n))}{2} + \log_3(n)b = \\ T(0) + 1 + b + (3/2)(n-1) + \log_3(n)b = \\ (3/2)n + \log_3(n)b + b + a - (1/2) = O(n). \end{aligned} \tag{9}$$

Grading scheme: 2 marks for the unfolding to the general form, 2 marks for the computation of $c = \log_3(n)$, 1 mark for the computation of $(3/2)(n-1)$, and 1 mark for the final derivation of O(n).

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.
Solution:

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

    unsigned int range, yearBought;
    string droneType, manufacturer, description, batteryType;
    friend class DronesManager;

public:
    DroneRecord(); // required to make DronesManager constructor function
    DroneRecord(unsigned int newDroneID, unsigned int newRange,
        unsigned int newYearBought, string newDroneType,
        string newManufacturer, string newDescription, string newBatteryType);

    unsigned int getDroneID() const;
    unsigned int getRange() const;
    unsigned int getYearBought() const;
    string getDroneType() const;
    string getManufacturer() const;
    string getDescription() const;
    string getBatteryType() const;
```

```
    void setDroneID(unsigned int newDroneID);
    void setRange(unsigned int newRange);
    void setYearBought(unsigned int newYearBought);
    void setDroneType(string newDroneType);
    void setManufacturer(string newManufacturer);
    void setDescription(string newDescription);
    void setBatteryType(string newBatteryType);
};

class DronesManager {
    // fill in other required member attributes and methods below
    // members should be declared as public or private as appropriate
    DroneRecord* data;
    unsigned int capacity = 1024;
    unsigned int size;
public:
    DronesManager();
    ~DronesManager();

    unsigned int getSize() const;
    unsigned int getCapacity() const;
    bool empty() const;
    bool full() const;

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

Grading scheme: 1 mark for *DroneRecord* attributes, 1 mark for the *DroneRecord* methods, 1 mark for $DroneRecord * data$, 0.5 mark for *size* and *capacity* attributes and getter methods, and 0.5 mark for $DronesManager$ constructor that includes *cap* as input.

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

```
DronesManager::DronesManager() { // 0.5 mark
 data = new DroneRecord[capacity];  // 1 mark
}
DronesManager::~DronesManager() { // 0.5 mark
 delete [] data; // 1 mark
}
```

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.

Solution:

```
bool DronesManager::removeDrone(unsigned int droneID) {
        if(!size) // 1 mark
                return false;

        unsigned int i = 0; // 2 marks; find the required index
        for(; i < size; ++i) {
                if ((data+i)->droneID == droneID)
                        break;
        }

        if (i == size) // 1 mark; check if found
                return false;

        for (unsigned int j = i; j < (size - 1); ++j) { // 2 marks
                data[j] = data[j+1];
        }

        --size; // 1 mark

        return true;
}
```