University of Waterloo
Midterm Examination
Spring 2019

Student Name:	_____
Student ID Number:	_____
Course Section:	**MTE 140**

Instructors:	Dr. Igor Ivkovic
Date of Exam	Spring 2019
Time Period	12:30pm-2:20pm
Duration of Exam	110 minutes
Pages (including cover)	15 pages (one page of scrap paper)
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: (15 marks)	Question 3: (10 marks)	Question 4: (15 marks)
Total: (50 marks)			

Useful Formulas

You may remove this page from the exam. If you do remove it, write your Student Number on it, and hand it in with your exam.

- 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),

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

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

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

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

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

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

- $$x^{\log_b y} = y^{\log_b x} \tag{7}$$

- $$\log_b xy = \log_b x + \log_b y \tag{8}$$

- $$\log_b \frac{x}{y} = \log_b x - \log_b y \tag{9}$$

1 Question 1. Algorithm Complexity (10 marks)

(approx. 20 minutes)

a. (4 marks)

Order the following functions from smallest to largest based on their order of growth in terms of the Big-O notation:

1. $\log(\log(n^2))$
2. $1245^2 + 3547n^4 + 42234n^6 + 151515$
3. $3 + 9 + 27 + 81 + 243 + \cdots + 3^{n-1} + 3^n$
4. $n^2 \log_{1567}(2222^{4545n})$
5. $4 + 8 + 12 + 16 + 20 + \cdots + 4n$

Solution:

1.

$$\log(\log(n^2)) = O(\log(\log(n^2))) \tag{10}$$

2.

$$1245^2 + 3547n^4 + 42234n^6 + 151515 = O(n^6) \tag{11}$$

3.

$$3 + 9 + 27 + 81 + 243 + \cdots + 3^{n-1} + 3^n = \sum_{i=1}^{n} 3^i = \frac{3^{n+1} - 1}{3 - 1} - 1 = \frac{3^{n+1} - 3}{2} = O(3^n) \tag{12}$$

4.

$$n^2 \log_{1567}(2222^{4545n}) = 4545n^3 \log_{1567}(2222) = O(n^3) \tag{13}$$

5.

$$4 + 8 + 12 + 16 + 20 + \cdots + 4n = 4\sum_{i=1}^{n} i = 4\frac{n(n+1)}{2} = 4\frac{n^2 + n}{2} = O(n^2) \tag{14}$$

Ordering: (1) $O(\log(\log(n^2)))$ < (5) $O(n^2)$ < (4) $O(n^3)$ < (2) $O(n^6)$ < (3) $O(3^n)$.

Grading scheme: Subtract 1 mark for each equation that is out of place, down to 0 marks.

b. (6 marks)

Let the runtime efficiency of an algorithm be defined as: $T(n) = \sum_{i=3}^{n}\sum_{j=1}^{n}\sum_{k=7}^{n}(i+2j)$.

Derive and express using Big-O notation a tight upperbound for $T(n)$ using limits as $n \to \infty$ or using the Big-O formal definition. Show all steps in deriving your solution. **Solution:**

$$
\begin{aligned}
T(n) = \sum_{i=3}^{n}\sum_{j=1}^{n}\sum_{k=7}^{n}(i+2j) = \\
\sum_{i=3}^{n}\sum_{j=1}^{n}(i+2j)(n-6) = \\
(n-6)\sum_{i=3}^{n}(i\sum_{j=1}^{n}1 + 2\sum_{j=1}^{n}j) = \\
(n-6)\sum_{i=3}^{n}(in + n(n+1)) = \\
(n-6)n(\sum_{i=3}^{n}i + (n+1)\sum_{i=3}^{n}1) = \\
(n-6)n(\frac{n(n+1)}{2} - 2 - 1 + (n+1)(n-2)) = \\
(n-6)n\frac{(n^2+n-6+2n^2-2n-4)}{2} = \\
\frac{(n^2-6n)(3n^2-n-10)}{2} = \\
\frac{(3n^4-19n^3-4n^2+60n)}{2} = \\
\frac{3}{2}n^4 - \frac{19}{2}n^3 - 2n^2 + 30n = \\
O(n^4).
\end{aligned}
\tag{15}
$$

(Option A.)
$\lim_{n\to\infty}\frac{\frac{3}{2}n^4-\frac{19}{2}n^3-2n^2+30n}{n^4} = \lim_{n\to\infty}(\frac{3}{2} - \frac{19}{2n} - \frac{2}{n^2} + \frac{30}{n^3}) = \lim_{n\to\infty}\frac{3}{2} = constant.$

Hence, it follows that $T(n) = O(n^4)$.

(Option B.)

We must show that $\frac{3}{2}n^4 - \frac{19}{2}n^3 - 2n^2 + 30n \leq Kn^4$ for all $n >= n_0$.

From $\frac{3}{2}n^4 - \frac{19}{2}n^3 - 2n^2 + 30n \leq Kn^4$, it follows that $\frac{3}{2} - \frac{19}{2n} - \frac{2}{n^2} + \frac{30}{n^3} \leq K$.

Let $n >= n_0 = 1$. It follows that as $n \uparrow$, $\frac{3}{2} - \frac{19}{2n} - \frac{2}{n^2} + \frac{30}{n^3} \leq \frac{3}{2} + \frac{30}{n^3} \leq \frac{3}{2} + 30 < 32$.

Hence, for $K = 32$ and $n \geq n_0 = 1$, it follows that $T(n) = O(n^4)$.

Grading scheme: 4 marks for the derivation of n^4 as tight upper bound. 2 marks for the correct verification using limits as $n \to \infty$ or using the Big-O formal definition.

2 Question 2. Algorithm Design (15 marks)

(approx. 40 minutes)

a. (10 marks)

You and your friend are playing a game in order to hone your mathematical skills.

They generate an unsorted list of *values* drawn from the range $[1, 1000]$ inclusive of 1 and 1000; the list may contain duplicate values. Your task is to read through the list, and extract the minimum and maximum value. After that, your friend would get to ask you for the I-th missing number from the range that starts with the minimum value and ends with the maximum value; if there is no such value, you would tell them -1. If you answer their question correctly three times, you win the game; if not, your friend wins. After that, you switch roles, and then you get to generate the list.

For example, your friend generates the following: $[14, 2, 3, 4, 6]$. The minimum value is 2, the maximum value is 14, and the missing values are $[5, 7, 8, 9, 10, 11, 12, 13]$. Your friend asks you for the $5th$ missing number to which you reply 10, $7th$ missing number to which you reply 12, and the $20th$ missing number to which you reply -1. Hence, you win the game.

Write the function

```
int find_ith_missing_value(vector<int> values, unsigned int missing_index)
```

that will help you play this game and that returns the missing I-th value. The *values* vector is the list that your friend generates while *missing_index* is the index of the missing value. If the index is zero or if it exceeds the range of the given values, return -1.

Hint: Use a temporary (buffer) array of integers to record number of occurrences of integers in the *values* list. Use the temporary array at the end to find the missing values. Record minimum and maximum indices as separate variables as you are iterating through the temporary array. If the *missing_index* is zero or if it exceeds the maximum index, return -1.

You may also write your function in pseudocode without focusing on syntactic details (e.g., you may write "$for\ i\ =\ 0\ to\ (n-1)$"), but you need to specify each step and each function call unambiguously.

For full marks, the run-time performance of your algorithm needs to be linear $O(n)$ where n is the size of the *values* list. That is, you may iterate through the list multiple times using loops but the loops cannot be nested. For partial marks, the run-time performance of your algorithm can be worse than linear (e.g., $O(n^2)$). Note that comparison-based sorting will require $O(n\ log(n))$ time. [The amount of partial marks awarded will be decided when grading.]

Solution:

```
// PURPOSE: Finds the I-th missing value in a given list from the range
//          defined by the minimum and maximum values in the same list
//    INPUTS: values - a list of integer values to be examined
//            missing_index - index of the value that needs to be found
//    RETURNS: I-th missing value in a given list if the value is found
//             -1 if the value cannot be found
int find_ith_missing_value(vector<int> values, unsigned int missing_index) {
    // intialize variables
    int missing_value = -1, min_index = 0, max_index = 0, count = 0;
    vector<int> data_count(1001, 0); // align indices to numbers

    // iterate through the values array
    // increment corresponding positions in data_count [1 mark]
    for (unsigned int index = 0; index < values.size(); index++) {
        ++data_count[values[index]];
    }

    // iterate through the data_count vector [1 mark]
    for (unsigned int index = 0; index < data_count.size() ; index++) {
        // store min_index as the first location that is not zero [1 mark]
        if (data_count[index] > 0 && min_index == 0)
            min_index = index;

        // store max_index as the last location that is not zero [1 mark]
        if (data_count[index] > 0)
            max_index = index;

        // count locations that are equal to zero [2 marks]
        if (data_count[index] == 0 && min_index > 0)
            ++count;

        // store missing_value if the corresponding count is reached [2 marks]
        if (data_count[index] == 0 && count == missing_index)
            missing_value = index;
    }

    // if the missing_value exceeds the max_index or if it is zero, return -1
    // else, return the missing_value [2 marks]
    return (missing_value > max_index || missing_index == 0? -1 : missing_value);
}
```

Grading scheme: See above. Award up to 5 marks for the solutions that are worse than linear (e.g., $O(n + m)$ where n is the size of the values list and m is the size of the buffer array). Subtract marks for solutions that are very hard to read or understand, or that do not provide all the required code steps.

```
int main() {
    vector<int> test1 = {14, 2, 3, 4, 6}; // missing 5, 7, 8, 9, 10, 11, 12, 13

    cout << find_ith_missing_value(test1, 5) << endl;
    cout << find_ith_missing_value(test1, 7) << endl;
    cout << find_ith_missing_value(test1, 20) << endl;
    cout << find_ith_missing_value(test1, 1) << endl;
    cout << find_ith_missing_value(test1, 9) << endl;
    cout << find_ith_missing_value(test1, 10000) << endl;

    vector<int> test2 = {1};
    cout << find_ith_missing_value(test2, 5) << endl;

    return 0;
}
```

b. (5 marks)

Once you have designed the algorithm, explain in natural language — not code — how would you comprehensively test the algorithm. That is, list five or more equivalence classes with ten or more test cases in total that you would use to ensure correctness of your algorithm and enhance its robustness. For each equivalence class, write a short description.

For example, one of the examples given above could be explained as "test a scenario where the values list contains no duplicates and the missing index exceeds the range." You do not have to modify the code to pass the test cases.

Solution:

```
[Test for cases where the values list is empty; 1 mark]
1. values list is empty, missing_index is zero
2. values list is empty, missing_index is not zero

[Test for cases where the values list contains values outside the [1,1000] range; 1 mark]
3. values list equals [5,5000], missing_index is zero
4. values list equals [3402,-3411], missing_index is not zero

[Test for cases where the values list contains one integer; 1 mark]
5. values list equals [3], missing_index is zero
6. values list equals [5], missing_index is one
7. values list equals [125], missing_index is two

[Test for cases where the values list contains multiple integers in order; 1 mark]
8. values list equals [3,4,5], missing_index is zero
9. values list equals [3,5,7], missing_index is one
10. values list equals [2,30,455], missing_index is 25

[Test for cases where the values list contains multiple integers out of order; 1 mark]
11. values list equals [122,3,45,5], missing_index is zero
12. values list equals [5,3,2,55], missing_index is one
13. values list equals [32,30,255], missing_index is 34

[Test for cases where the values list contains multiple integers and some duplicates;
1 mark]
14. values list equals [4,3,4,5], missing_index is zero
15. values list equals [5,5,2,55], missing_index is one
16. values list equals [30,2,255,30], missing_index is 34

[Test for cases where the values list contains multiple integers and all duplicates;
1 mark]
17. values list equals [4,4,4,4,4], missing_index is zero
18. values list equals [5,5,5,5,5,5,5], missing_index is one
19. values list equals [30,30], missing_index is 15

[Test for cases where the values list contains multiple integers while missing index
is negative; 1 mark]
20. values list equals [122,3,45,5], missing_index is -5
21. values list equals [5,3,5,55], missing_index is -17
22. values list equals [1,1,1], missing_index is -22
```

Grading scheme: If the description along with two or more test cases are provided, award 1 mark for that category. Some categories are mandatory, such as testing with the empty list, testing with duplicate values in the list, and testing with missing indices that exceed the range. Other categories may be accepted as appropriate. To qualify for full marks, at least ten distinct cases need to be provided.

3 Question 3. Pointers and Dynamic Memory (10 marks)

(approx. 10 minutes)
Examine the following code fragment. For each line marked with the [*Question*∗] tag, answer the related question in the space below, and provide a short explanation. If a line represents a compilation error, state that instead, and briefly explain why the error occurs. Assume that no lines will be skipped in execution.

```
struct BasketballStar {
    string name; BasketballStar* next;
    BasketballStar(string new_name) : name(new_name), next(NULL) {}
};

BasketballStar** with_pointers_to_championships() {
    BasketballStar** roster = new BasketballStar*[15]();
    BasketballStar star1("Kyle"), star2("Danny"), star3; // [Question1. Will this compile?]

    roster[0] = &star1; // [Question2. Given that roster[0] is passed outside
                        // this function via the return statement,
                        // is this going to result in a dangling pointer?]

    roster[1] = new BasketballStar(star2); // [Question3. Is roster[1] pointing
                        // to heap memory or stack memory after this line?]

    cout << (*(roster + 1)).name  << endl; // [Question4. What is the output of this line?]
    cout << roster[2]->name << endl; // [Question5. What is the output of this line?]

    return roster;
}
```

Solution:

[*Question*1] No, this will not compile since the empty constructor is missing for star3 given that an explicit parametric constructor was included. [2 marks]

[*Question*2] Yes, this will result in a dangling pointer. One way to address this would be to use copy constructor as it was done for star2. [2 marks]

[*Question*3] It is pointing to heap memory since the new operator was used. [2 marks]

[*Question*4] This line will not compile due to ".name" syntax error. For partial marks, if there was no syntax error (e.g., ".name" was replaced with "→ *name*"), then "Danny" would be outputted. [2 marks]

[*Question*5] This line should crash the program since roster[2] was never allocated memory and it is pointing to NULL. Some IDEs may stop the execution and just return an error code instead. For partial marks, the output is unknown/undefined since roster[2] was never initialized. [2 marks]

4 Question 4. Data Structure Design (15 marks)

(approx. 40 minutes)
We are tasked with developing statistical software that stores information about basketball players and basketball teams currently playing in a basketball league (e.g., NBA, WNBA). For each player, the software will store their unique player ID as an unsigned integer, full name as a string, position played as a string, height as an integer value in centimeters, weight as an integer value in kilograms, and date of birth as a string. For each team, we will store the active roster and backup roster along with the size for each roster. Both rosters are not sorted.

a. (4 marks) As the reference model for our design, we will make use of the **sequential list implementation of List ADT from Practice Exercise #1 and Lab Assignment #1** where *BasketballPlayer* objects will be managed by the corresponding *BasketballTeam* instances. As internal storage inside *BasketballTeam*, you may use the $< vector >$ library or a dynamically-sized array. The default capacity for each roster is 15. Do not create a linked-list implementation.

To that end, complete the C++ class declaration provided below to make *BasketballPlayer* and *BasketballTeam* function. For *BasketballPlayer*, include the empty constructor, parametric constructor, and *operator* ==. For *BasketballTeam*, include the empty constructor, *void insert_new_player*(*BasketballPlayer new_player*), *void remove_player*(*unsigned int playerID*), *void activate_player*(*unsigned int playerID*) and *void deactivate_player*(*unsigned int playerID*). Provide only method signatures at this point and no method implementation. Include the *size* variables if applicable. You do not have to include getters and setters.
Solution:

```
class BasketballPlayer {
    unsigned int playerID;
    friend class BasketballTeam;
    // fill in other required member attributes and methods below
    // members should be declared as public or private as appropriate
    string player_name, position_played, player_dob; // [1 mark] for attributes
    unsigned int player_height, player_weight;

public:
    BasketballPlayer(); // [1 mark] for the constructors and operator==
    BasketballPlayer(unsigned int new_playerID);
    BasketballPlayer(unsigned int new_playerID, string new_player_name,
        unsigned int new_player_height, unsigned int new_player_weight,
        string new_position_played, string new_player_dob);
    bool operator==(const BasketballPlayer& rhs) const;
};

class BasketballTeam {
    // fill in other required member attributes and methods below
    // members should be declared as public or private as appropriate
    vector<BasketballPlayer> active_roster; // [0.5 mark]
    vector<BasketballPlayer> passive_roster; // [0.5 mark]
public:
    BasketballTeam(); // [1 mark] for the constructor and service methods
    int get_size() const;
    void print();
```

```
    void insert_new_player(BasketballPlayer new_player);
    void remove_player(unsigned int playerID);
    void activate_player(unsigned int playerID);
    void deactivate_player(unsigned int playerID);
};
```

b. (5 marks) Implement the method insert_new_player(BasketballPlayer new_player) that will insert the *BasketballPlayer* element at the end of the backup roster list, and update the *size* variable if applicable. If the provided *playerID* is already in the list, then no change will be made.

If you are using the $< vector >$ library, you may use *push_back*(*element*) or *resize*(*value*) functions as needed. You must use exact C++ syntax for this function. **Solution:**

```
void BasketballTeam::insert_new_player(BasketballPlayer new_player) {
    // base case: empty list [2 marks]
    if (!passive_roster.size()) {
        passive_roster.push_back(new_player);

    // general case: non-empty list
    } else {
        for (int index = 0; index < passive_roster.size(); ++index) { // 2 marks
            if (passive_roster[index].playerID == new_player.playerID)
                return;
        }

        passive_roster.push_back(new_player); // 1 mark
    }
}
```

Grading scheme: See above.

c. (6 marks) Implement the method *void remove_player*(*unsigned int playerID*). The method will take as input a *playerID* value, remove the corresponding player from the backup roster, free up its memory, move other players ahead in the roster by one so that the list remain contiguous, and update the *size* variable.

For example, if the backup roster stores the following player IDs: $[8, 25, 15, 13]$ and if $playerID = 25$, then the function will remove the second player from the list and produce the following: $[8, 15, 13]$. If the *playerID* is not in the list, then no change will be made.

You must use exact C++ syntax for this function. If you are using the $< vector >$ library, you may use *pop_back*() or *resize*(*value*) functions but not *erase*() or *swap*(). You may also use *iostream* and *string* libraries in your implementation but no other external libraries. However, you may write helper functions of your own. You also need to document your code and explain the rationale for key steps. **Solution:**

```
void BasketballTeam::remove_player(unsigned int playerID) {
    // base case: empty list [1 mark]
    if (!passive_roster.size()) {
        return;

    } else {
        // general case: non-empty list
        bool found_the_player = false;
        int index = 0;
        // iterate through the list
        while (index < passive_roster.size()) {
            // check if the specified player is in the list [2 marks]
            if (!found_the_player && passive_roster[index].playerID == playerID)
                found_the_player = true;

            // if the player is in the list, move other elements ahead by one [2 marks]
            if (found_the_player && index < passive_roster.size() - 1)
                passive_roster[index] = passive_roster[index + 1];

            // increment current index by one
            ++index;
        }

        // if the player is in the list, remove the last element [1 mark]
        if (found_the_player)
            passive_roster.pop_back();
    }
}
```

Grading scheme: See above.

```
// additional code for testing; otherwise, not needed
BasketballPlayer(unsigned int new_playerID) :
    playerID(new_playerID), player_height(0), player_weight(0) {};
BasketballTeam(){};

void BasketballTeam::print() {
    for (int index = 0; index < passive_roster.size(); ++index) {
        cout << passive_roster[index].playerID << endl;
    }
}

int main() {
    BasketballTeam raptors;
    BasketballPlayer p1(10), p2(2), p3(54);
    raptors.insert_new_player(p1);
    raptors.insert_new_player(p1);
    raptors.insert_new_player(p1);
    raptors.insert_new_player(p2);
    raptors.insert_new_player(p3);
    raptors.insert_new_player(p1);
    raptors.insert_new_player(p1);
    raptors.insert_new_player(p3);
    raptors.insert_new_player(p3);
    raptors.insert_new_player(p3);
    raptors.print();

    raptors.remove_player(10);
    raptors.remove_player(10);
    raptors.print();
    raptors.remove_player(2);
    raptors.print();
    raptors.remove_player(54);
    raptors.print();

    return 0;
}
```