University of Waterloo
Midterm Examination Solutions
Spring 2018

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

Instructors:	Dr. Igor Ivkovic
Date of Exam	Spring 2018
Time Period	9:00am-11:00am
Duration of Exam	120 minutes
Pages (including cover)	13 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 15 minutes of the exam. Plan your time wisely to finish the exam on time.

Question 1: (11 marks)	Question 2: (15 marks)	Question 3: (9 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),
  $$S = n\left(\frac{a_1 + a_n}{2}\right) \tag{1}$$
- $$\sum_{i=k}^{n} 1 = (n - k + 1) \tag{2}$$
- $$\sum_{i=1}^{n} i = \left(\frac{n(n+1)}{2}\right) \tag{3}$$
- $$\sum_{i=1}^{n} i^2 = \left(\frac{n(n+1)(2n+1)}{6}\right) \tag{4}$$
- $$\sum_{i=0}^{n} r^i = \left(\frac{r^{n+1} - 1}{r - 1}\right) \tag{5}$$
- $$\log_b x = \frac{\log_c x}{\log_c b} \tag{6}$$

1 Question 1. Algorithm Complexity (11 marks)

(approx. 25 minutes)

a. (3 marks)

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

1. $2 + 4 + 8 + 16 + \cdots + 2^{n-1} + 2^n$
2. $n^2 \log_{4242}(1234^{4321n})$
3. $4 + 8 + 12 + 16 + 20 + \cdots + 4n$
4. $\log(\log(n^2))$

Solution:

1.

$$\sum_{i=1}^{n} 2^i = \frac{2^{n+1} - 1}{2 - 1} - 1 = 2^{n+1} - 2 = O(2^n) \tag{7}$$

2.

$$n^2 \log_{4242}(1234^{4321n}) = 4321n^3 \log_{4242}(1234) = O(n^3) \tag{8}$$

3.

$$4\sum_{i=1}^{n} i = 4\frac{n(n+1)}{2} = 4\frac{n^2 + n}{2} = O(n^2) \tag{9}$$

4.

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

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

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

b. (4 marks)

Provide a code fragment in pseudocode that would exhibit the runtime of $O(log(n^3)+log(n^2))$. 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:

```
int n = 45, step_count = 0;
for (int k = 1; k < n * n; k = k * 2) { // 0.5 marks for k < n^2, 1 mark for k = k*2
    ++step_count; // 0.5 marks for constant steps
    cout << "step" << endl;
}
cout << step_count << endl; // corresponds to log_2(n^2)

step_count = 0;
for (int k = 1; k < n * n * n; k = k * 2) { // 0.5 marks for k < n^3, 1 mark for k = k*2
    ++step_count; // 0.5 marks for constant steps
    cout << "step" << endl;
}
cout << step_count << endl; // corresponds to log_2(n^3)
```

c. (4 marks)

Suppose that the runtime efficiency of an algorithm is defined as:

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} i^2 \; log(n).$$

Determine the algorithm's order of growth in terms of the Big-O notation by simplifying the given expression. You do not have to provide a formal proof using the Big-O formal definition, but you need to show all steps of your work.

Solution:

$$\begin{aligned}
T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} i^2 \; log(n) = \\
[1 mark] \sum_{i=1}^{n} i^2 \; log(n) \sum_{j=1}^{n} 1 = \\
[1 mark] \sum_{i=1}^{n} i^2 \; log(n)(n-1+1) = \\
n \; log(n) \sum_{i=1}^{n} i^2 = \\
n \; log(n) \frac{n(n+1)(2n+1)}{6} = \\
n \; log(n) \frac{(n^2+n)(2n+1)}{6} = \\
[1 mark] \frac{1}{6} log(n)(2n^4+3n^3+n^2) = \\
[1 mark] O(n^4 log(n)).
\end{aligned} \qquad (11)$$

2 Question 2. Algorithm Design (15 marks)

(approx. 40 minutes)

a. (11 marks)

You are given an array A that contains integers. Write the function

```
void find_equilibrium(int* A, int size_A)
```

that will print all equilibrium indices for this array such that the sum of elements at indices lower than equilibrium is equal to the sum of elements at indices higher than equilibrium.

For example, if A equals $[-4, 2, 3, 4, -6]$, then its only equilibrium index is 2 where the sum of $A[0] + A[1] = -2$ equals $A[3] + A[4] = -2$.

Similarly, if A equals $[-4, -2, 3, 4, -5]$, then its only equilibrium index is 0 where the sum of no elements (0) equals the sum of remaining four elements $A[1] + A[2] + A[3] + A[4] = 0$.

Also, if A equals $[0, 0, 0, 0, 0]$, then its equilibrium indices are 0, 1, 2, 3, and 4 since for all of them both sums are equal to 0.

Note that equilbrium index must be a valid array index, and cannot be outside of array boundaries. If the array is empty, then it cannot contain valid indices. If no equilibrium indices are available, then nothing is printed.

For full marks, the run-time performance of your algorithm needs to be linear (i.e., $O(n)$ where n is the size of the array), and the memory performance has to be constant (i.e., $O(1)$).

For partial marks, the run-time performance of your algorithm can be worse than linear (e.g., $O(n^2)$), and you may use more than constant memory space. [The amount of partial marks awarded will be decided when grading.]

Solution:

```
void find_equilbrium(int* A, int size_A) {
    if(!A)
        return;

    int array_sum = 0, left_sum = 0;

    for (int i = 0; i < size_A; ++i)
        array_sum += A[i];

    for(int i = 0; i < size_A; ++i) {
        array_sum -= A[i];

        if(left_sum == array_sum)
            cout << i << endl;

        left_sum += A[i];
    }
}

int main() {
    int A[5] = {-4, 2, 3, 4, -6};
    int B[5] = {-4, -2, 3, 4, -5};
    int C[5] = {0, 0, 0, 0, 0};
    int D[6] = {1, 1, 1, 1, 1, 1};
    cout << "testing A" << endl;
    find_equilbrium(A, 5);
    cout << "testing B" << endl;
    find_equilbrium(B, 5);
    cout << "testing C" << endl;
    find_equilbrium(C, 5);
    cout << "testing D" << endl;
    find_equilbrium(D, 6);
    cout << "testing E" << endl;
    find_equilbrium(NULL, 0);
    return 0;
}

/* Output:
testing A
2
testing B
0
testing C
0
1
2
3
4
testing D
testing E
*/
```

Grading scheme: 1 mark for checking if A is NULL, 2 marks for declaring array-sum and left-sum, 2 marks for the first loop including array-sum updating, 2 marks for the second loop including array-sum updating, 2 marks for index output, 2 marks for left-sum updating. Award up to 5 marks for the solutions that are worse than O(n) or that use more than O(1) space. 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 comprehensively 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, one of the test case given above could be explained as "test non-empty array of size 5 with all 0 values". You do not have to modify the code to pass the test cases.

Solution:

```
[Test for A being NULL or empty and size_A is zero; 0.5 mark]
1. A is NULL, size_A is 0
2. A is not NULL and empty, size_A is 0

[Test for cases where the array size and size_A value do not match; 0.5 mark]
3. A is NULL, size_A is not 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 contains only one type of values and at least one equilibrium; 0.5 mark]
6. A is not NULL, size_A is 7, all values are positive
7. A is not NULL, size_A is 15, all values are negative
8. A is not NULL, size_A is 18, all values are zero

[Test for A contains only one type of values and no equilibrium; 0.5 mark]
9. A is not NULL, size_A is 8, all values are positive
10. A is not NULL, size_A is 16, all values are negative

[Test for A contains mixed values and at least one equilbrium; 0.5 mark]
11. A is not NULL, size_A is odd
12. A is not NULL, size_A is even

[Test for A contains mixed values and no equilibrium; 0.5 mark]
13. A is not NULL, size_A is odd
14. A is not NULL, size_A is even

[Test for A contains mixed values whose sum exceeds maximum integer value; 0.5 mark]
15. A is not NULL, size_A is odd, values are large and positive
16. A is not NULL, size_A is even, values are large and positive

[Test for A contains mixed values whose sum exceeds minimum integer value; 0.5 mark]
17. A is not NULL, size_A is odd, values are large and negative
18. A is not NULL, size_A is even, values are large and negative

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

3 Question 3. Pointers and Dynamic Memory (9 marks)

(approx. 15 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 SuperStar {
    string name; SuperStar* next;
    SuperStar(string nName) : name(nName), next(NULL) {}
};

void fun_with_pointers() {
    SuperStar* ss = new SuperStar("Dark Sky");
    cout << &ss << endl;
    cout << ss->name << endl; // [Question1. Where is ss->name located (stack or heap)?]
    delete ss;

    int** data = new int*[3]();
    data[0] = new int[10]();
    data[1] = new int[20];
    data[2] = new int; // [Question2. Will this line compile and run?]

    int* q = data[0];
    *q = 5;
    q = q + 2;
    *q = 8;
    cout << data[0][1] << data[0][2] << endl; // [Question3. What is the output of this line?]
    cout << data[1][0] << endl; // [Question4. What is the output of this line?]
    cout << *data[2] << endl;

    delete [] data[1];
    delete [] data[0];
    delete [] data; // [Question5. Will a memory leak occur after this line?]
}
```

Solution:

[*Question*1] *ss* was allocated space using new, so $ss->name$ should be on the heap. [2 marks]

[*Question*2] Yes, this line will compile and run. It is just a pointer being set. [1 mark]

[*Question*3] Output: 08. [2 marks]

[*Question*4] Output is undefined value since $data[1][0]$ was not initialized. [2 marks]

[*Question*5] Yes, this will cause a memory leak. The array of pointers will be freed with $delete[]data$ but not $data[2]$ which was allocated with $data[2] = new int;$. [2 marks]

4 Question 4. Data Structure Design (15 marks)

(approx. 40 minutes)
You recently joined a software development company on a co-op term. Another co-op student worked there before you in the same role that you presently occupy. They developed a web application for a small art auction house that handles various types of artwork.

Each artwork was represented as an instance of *Artwork* class. For each piece of artwork, they stored the artist name, year it was made, title, and type of art (e.g., painting, sculpture, photograph); year it was made was stored as an unsigned integer while all other attributes were stored as *string* values.

ArtManager class was used as a container to handle *Artwork* objects, and it included relevant attributes/fields; *ArtManager* was modeled as a doubly linked list.

a. (3 marks) Your supervisor asked you to extend the existing design and provide both doubly linked list and sequential list access to *Artwork* objects stored inside *ArtManager*. That is, you should implement combined array-list access to *Artwork* objects, where objects can be accessed using array index (think *Assignment* #2 using $data[i]$ where *data* points to a corresponding array of pointers) as well as using linked list (think *Assignment* #3 using $first$ and *last* pointers that point to corresponding *Artwork* objects).

Unfortunately, the files implemented by the previous student were corrupted, and many key elements are missing. Using *Assignment* #2 and *Assignment* #3 as the basis for your implementation, modify the declarations given below, and provide corresponding member attributes and methods that are needed to make *Artwork* and *ArtManager* function.

Solution:

```
class Artwork {
    unsigned int yearMade;
    string artistName, artTitle, artType;
    friend class ArtManager;
public:
    Artwork(string nArtistName, unsigned int nYearMade,
        string nArtTitle, string nArtType) :
        artistName(nArtistName), yearMade(nYearMade),
        artTitle(nArtTitle), artType(nArtType)  {}

    unsigned int getYearMade();
    string getArtistName();
    string getArtTitle();
    string getArtType();

    void setYearMade(unsigned int nYearMade);
    void setArtistName(string nArtistName);
    void setArtTitle(string nArtTitle);
    void setArtType(string nArtType);

    // fill in other required member attributes and methods below
    // members should be declared as public or private as appropriate
private:
    Artwork* prev; // [0.5 marks]
    Artwork* next; // [0.5 marks]
};

class ArtManager {
    int size;

public:
    ArtManager();
    ~ArtManager();

    int getSize();
    void print() const;

    bool insertArtworkUnsorted(string nArtistName, unsigned int nYearMade,
                string nArtTitle, string nArtType, unsigned int index);
    bool removeArtwork(string artistName, string title);

    // fill in other required member attributes and methods below
    // members should be declared as public or private as appropriate
private:
    Artwork* first; // [0.5 marks]
    Artwork* last; // [0.5 marks]
    Artwork** data; // [0.5 marks]
    int capacity; // [0.5 marks]
};
```

b. (4 marks) Implement default constructor and destructor for the *ArtManager* class. Include appropriate initialization and deinitialization of member variables, respectively. Set default capacity to 128.
Solution:

```
    ArtManager() { // 2 marks for the constructor
        size = 0;
        first = NULL;
        last = NULL;
        capacity = 128;
        data = new Artwork*[capacity];
    }

    ~ArtManager() { // 2 marks for the destructor
        while (first) {
            Artwork* temp = first->next;
            delete first;
            first = temp;
        }
        delete [] data;
    }
```

c. (8 marks) Write the method $ArtManager :: insertArtworkUnsorted$ as specified below. The method inserts the given artwork record into $ArtManager$ at the given index, and updates sequential list and linked list structures as needed. That is, both *data* array and corresponding doubly linked list pointers need to be updated.

There must be no fragmentation in the list; that is, no clusters of unallocated elements except at the list end. You are not responsible for growing or shrinking the sequential list. You are not responsible to check for duplicate values. If there is enough capacity to insert the new element and there is no list fragmentation, the function returns *true*; else, the function returns $false$.

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 ArtManager::insertArtworkUnsorted(string nArtistName, unsigned int nYearMade,
                string nArtTitle, string nArtType, unsigned int index) {

    // create appropriate record [1 mark]
    Artwork* record = new Artwork(nArtistName, nYearMade, nArtTitle, nArtType);
    record->prev = NULL;
    record->next = NULL;

    if (size == capacity) // check if at capacity [0.5 mark]
        return false;

    if (index > size) // check for fragmentation [0.5 mark]
        return false;

    if(!first) { // handle empty list [0.5 mark]
        data[0] = record;
        first = record;
        last = record;

    } else { // handle non-empty list
        for (int i = size; i > index; --i) // move all pointers ahead by one [1 mark]
            data[i] = data[i - 1];

        if (index == 0) { // insert at the front [1 mark]
            data[index] = record;
            first->prev = record;
            record->next = first;
            first = record;

        } else if (index == size) { // insert at the back  [1 mark]
            data[index] = record;
            last->next = record;
            record->prev = last;
            last = record;

        } else { // insert in the middle [2 mark]
            data[index] = record;
            record->prev = data[index-1];
            record->next = data[index+1];
            data[index-1]->next = record;
            data[index+1]->prev = record;
        }

    }
    ++size; // increment the size [0.5 mark]

    return true;
}
```