University of Waterloo
Midterm Examination Solutions
Winter 2015

Student Name:	
Student ID Number:	
Course Section (circle one):	**BME 122 | MTE 140 | SYDE 223**

Instructors:	Alexander Wong and Igor Ivkovic
Date of Exam	March 2nd 2015
Time Period	5:00pm-7:00pm
Duration of Exam	2 hours
Pages (including cover)	11 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.

Question 1: (10 marks)	Question 2: (10 marks)	Question 3: (15 marks)	Question 4: (15 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)

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. $3 + 9 + 27 + 81 + 243 + \cdots + 3^{n-1} + 3^n$
2. $15234n \log_{897124}(5551212^{5551212n})$
3. $42232^2 + 42233n^7 + 42234n^5 + 42235$
4. $12342n \log_{234235}(43325n)$.

Solution:

1.

$$\sum_{i=1}^{n} 3^i = \frac{3^{n+1}-1}{3-1} - 1 = \frac{3^{n+1}-3}{2} = O(3^n) \tag{7}$$

2.

$$\begin{aligned} 15234n \log_{897124}(5551212^{5551212n}) &= \\ 15234n \frac{\log_{5551212}(5551212^{5551212n})}{\log_{5551212}(897124)} &= \\ \frac{15234 * 5551212n^2}{\log_{5551212}(897124)} &= O(n^2) \end{aligned} \tag{8}$$

3.

$$42232^2 + 42233n^7 + 42234n^5 + 42235 = O(n^7) \tag{9}$$

4.

$$\begin{aligned} 12342n \log_{234235}(43325n) &= \\ 12342n \log_{234235}(43325) + 12342n \log_{234235}(n) &= \\ O(n + nlog(n)) &= O(nlog(n)) \end{aligned} \tag{10}$$

Ordering: (4) $O(nlog(n))$ < (2) $O(n^2)$ < (3) $O(n^7)$ < (1) $O(3^n)$.

Grading scheme: 4 marks for the correct ordering, 2 marks for 2 equations being ordered correctly, 1 mark for 1 equation ordered correctly.

b. (6 marks)

Suppose that the runtime efficiency of an algorithm is defined as:
$T(n) = 14n^3 + 14n(n-1)^2 + ... + 126n + 56n + 14n$.
Determine the algorithm's order of growth in terms of the big-O notation, and prove that this is indeed the true order of growth using the big-O formal definition. Show all steps in your proof.

Solution:

$$\begin{aligned} T(n) = 14n^3 + 14n(n-1)^2 + ... + 126n + 56n + 14n = \\ 14n\sum_{i=1}^{n} i^2 = \\ \frac{14n^2(n+1)(2n+1)}{6} = \\ \frac{28}{6}n^4 + \frac{42}{6}n^3 + \frac{14}{6}n^2 \end{aligned} \tag{11}$$

Let $\frac{28}{6}n^4 + \frac{42}{6}n^3 + \frac{14}{6}n^2 \leq Kn^4$.

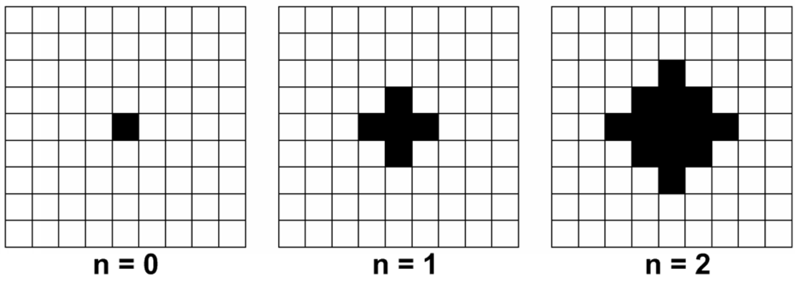
Then, let $n_0 = 1$, so it follows that $\frac{28}{6} + \frac{42}{6} + \frac{14}{6} = 14 \leq K$.

Hence, for $K = 14$ and $n \geq 1$, it follows that $T(n) = O(n^4)$.

Grading scheme: 1 mark for each correct step, for the total of 6 marks.

2 Question 2. Algorithm Design (10 marks)

You are asked to implement a non-recursive $O(n^2)$ function named *drawCells* that starts by drawing one cell on its first pass, and each of its iterations adds new cells all around its sides. For n = 0, it just draws one cell. For n = 1, it draws the cells adjacent (top/bottom/left/right) to the cell created for n = 0. For n = 2, it draws the cells adjacent to the cells created for n = 1. It then continues iterating until it reaches the given value of n. See the illustration of this algorithm below.



Write code to implement this function as *void drawCells*(*int n*). The function will start drawing cells at coordinates $(0, 0)$. To draw a cell at coordinates (x, y) and fill its pixels, you are being provided with the function *void* $fill(x, y)$ that you can call as needed. You may only use *iostream* library in your implementation, and no other external libraries. If you need any other helper functions, you will need to implement them.

Solution:

```
int abs (int val) {
    return val > 0 ? val : -val;
}

void drawCells(int n) {
    for (int i = -n; i <= n; ++i) {
        for (int j = -n; j <= n; ++j) {
            if (abs(i) + abs(j) <= n)
                fill(i,j);
        }
    }
}
```

Grading scheme: 3 marks for the correct outer loop, 3 marks for the correct inner loop, 3 marks for the correct if statement (including 1 mark for the absolute function implementation), and 1 mark for the correct use of fill(x,y), for the total of 10 marks.

3 Question 3. Divide and Conquer Algorithms (15 marks)

Consider the following function named *woof*:

```
void woof(int K, int firstDog, int lastDog) {
    int segment = (lastDog - firstDog) / 3;
    if (lastDog <= firstDog)
        cout << "Dog " << firstDog << " says WOOF!\n";
    else {
        if (K <= firstDog + segment)
            woof(K, firstDog, firstDog + segment);
        else if (K <= firstDog + segment * 2)
            woof(K, firstDog + segment + 1, firstDog + segment * 2);
        else
            woof(K, firstDog + segment * 2 + 1, lastDog);
    }
}
```

a. (5 marks)

Draw the call tree for this function when $woof(35, 9, 47)$ is called.

Solution:

```
woof(35,9,47)
|
woof(35,34,47)
|
woof(35,34,38)
|
woof(35,34,35)
|
woof(35,35,35)
```

Grading scheme: 1 mark for each correct step, for the total of 5 marks.

b. (4 marks)

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

Solution:
$T(0) = a$
$T(n) = T(n/3) + b$

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, 3 marks for the recursive case, for the total of 4 marks.

c. (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:

$$\begin{aligned} T(n) = T(n/3) + b = T(n/3^2) + 2b = \\ ... = T(n/3^i) + ib \end{aligned} \tag{12}$$

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,

$$\begin{aligned} T(n) = T(n/3^i) + ib = T(1) + \log_3(n)b = \\ T(0) + b + \log_3(n)b = a + b + \log_3(n)b = O(log(n)). \end{aligned} \tag{13}$$

Grading scheme: 2 marks for the unfolding to n = 1 case, 2 marks for the computation of $c = \log_3(n)$, and 2 marks for the final derivation of O(log(n)), for the total of 6 marks.

4 Question 3. Data Structure Design (15 marks)

Automated parking systems are a reality in many cities around the world. The idea is simple: maximize the utility of space available for parking, and minimize the time required to find a parking spot, by using an automated and mechanical system. How does the system work? A driver deposits their car at one of the entrance locations, and instead of a human moving the car, it is the robotic trolley that moves the car to an appropriate location. When the driver comes back to pick up their car, the same robotic trolley moves the car to the entrance location. The driver then enters the car and drives away.

Your job is to implement data structures with methods that will allow this single-level automated parking garage to function. Your implementation needs to be based on a doubly linked list, and it needs to ensure that the cars that have been deposited first are also retrieved first. To that end, the class Car is a container data type that stores details about the car, such as $ownerID$, *licencePlate*, and $manufacturer$. All of these attributes are stored as *string*s. Each Car also stores the details of the vehicle's current location and the new/destination location for when it is being moved. The current location is represented as integers *curRow* and *curColumn* for the current location's row and column respectively, and the new location is represented as integers *newRow* and *newColumn* for the destination location's row and column respectively. Finally, each Car object also stores relevant attributes needed for the linked data implementation to function.

In addition to the class Car, you are being asked to implement the class *RoboticTrolley*, which includes pointers to the linked data structure of Car objects, and which controls the robotic machine. *RoboticTrolley* also includes pointers to a data structure used to keep track of free spaces in the garage, but you are not responsible for implementing this data structure since it is out of scope of this question.

a. (2 marks) Based on the abstract data types (ADT) presented in class, which ADT would be most appropriate for this problem? Explain your answer.
Solution:

Answer: Queue ADT. The question specified First-In First-Out (FIFO) principle of storing and retrieving cars, and queues are based on the FIFO principle.

Grading scheme: 1 mark for the correct answer, and 1 mark for the explanation, for the total of 2 marks.

b. (6 marks) Fill in the missing information for the following interfaces using appropriate linked data representations for Car and *RoboticTrolley*. Include all of the necessary member attributes and methods. You should include at least one constructor, and getter/setter methods as needed.
Solution:

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

```
        string licencePlate, manufacturer;
        int curRow, curColumn;
        int newRow, newColumn;
        Car* next, prev;

    public:
        Car();
        Car(string newID, string newPlate, string newManf,
            int newCRow, int newCCol, int newNRow, int newNCol);
        ~Car();

        string getOwnerID();
        string getLicencePlate();
        string getManufacturer();
        int getCurRow(); int getCurColumn();
        int getNewRow(); int getNewColumn();

        void setOwnerID(string newID);
        void setLicencePlate(string newPlate);
        void setManufacturer(string newManf);
        void setCurRow(int newCRow); void setCurColumn(int newCCol);
        void setNewRow(int newNRow); void setNewColumn(int newNCol);
    };

    class RoboticTrolley {
        // fill-in other required member attributes and methods below
        // members should be declared as public or private as appropriate
        Car* head, tail;
        int size;
    public:
        RoboticTrolley();
        ~RoboticTrolley();

        int getSize();
        // Enqueue(), Dequeue(), and Peek() may also be inserted

        bool AssignLocation(Car* car);
        bool MoveCar(Car* car);
        bool DepositCar(Car* car);
    };
```

Grading scheme: 2 marks for *Car* attributes including *prev* and *next*, 2 marks for the *Car* methods, and 2 marks for the *RoboticTrolley* attributes and methods including *head* and *tail*, for the total of 6 marks.

To assign a vehicle to a location, the method *bool RoboticTrolley* :: *AssignLocation*(*Car* car*) is used. The method takes a car object pointer as input, and changes its new location values (*newRow* and *newColumn*) to a free location inside the garage. If the car is being moved back to the entrance, it will assign a free location at the entrance instead. If the location assignment fails, for example due to the lack of space, the location values are unchanged, and the method returns *false*; otherwise, the method returns *true*.

To move the vehicle to a location, the method *bool RoboticTrolley* :: *MoveCar*(*Car* car*) is used. The method takes a car object pointer as input, and moves the car to the new location (*newRow* and *newColumn*). After the car has been moved, its current location equals its new location. If the car could not be moved to its destination (e.g., due to a mechanical fault), the method returns *false*, and the car is returned back to its current location; otherwise, the method returns *true*.

You do not have to implement *AssignLocation* and *MoveCar* methods, and you may just call them in your implementations below.

c. (7 marks) Write the method *bool RoboticTrolley* :: *DepositCar*(*Car*∗ *car*), which takes a *Car* object pointer as input. It assigns the vehicle to a new physical location, moves the vehicle into the assigned location, and inserts the data item into the linked list. If the car could not be deposited, the method returns *false*; otherwise it returns *true*.

You may only use *iostream* and *string* libraries in your implementation, and no other external libraries. 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 RoboticTrolley::DepositCar(Car* car) {
```

Solution:

```
    // Check if the car pointer is NULL [1 mark]
    if (!car)
        return false;

    // Assign a new location to the car [1 mark]
    if (!AssignLocation(car))
        return false;

    // Move the car to the new location [1 mark]
    if (!MoveCar(car))
        return false;

    // Insert the car into the linked list
    // Insert at the start if the list is empty [2 marks]
    if(!head) {
        head = car;
        tail = car;
    } else {
    // Insert at the end if the list is not empty [2 marks]
        car->prev = tail;
        tail->next = car;
        tail = car;
    }
    // Increment the size
    ++size;

    return true;
}
```

Grading scheme: See the grading scheme above, for the total of 7 marks.