Question 1 ():

For each of the following functions, give the **tightest upper bound** chosen from among the usual simple functions listed in Section 3.5 of the course readings. Answers should be expressed in big-O notation.

(a) (1 point) $f_2(n) = 42n + 9\sqrt{n} + \log_2 n$

(b) (1 point) $f_3(n) = 4n^{0.7} + 6n \log_2 n + 280$

(c) (1 point) $f_1(n) = n \log_2 n + \log_2 n^2 + 280n$

Function	Most quickly growing term	Big-Oh (tight bound)
$42n + 9\sqrt{n} + \log_2 n$	$42n$	$O(n)$
$4n^{0.7} + 6n \log_2 n + 280$	$6n \log_2 n$	$O(n \log_2 n)$
$n \log_2 n + \log_2 n^2 + 280n$	$n \log_2 n$	$O(n \log_2 n)$

Question 2 ():

Suppose the **exact** time required for an algorithm A in both the best and worst cases is given by the function

$$T_A(n) = \frac{1}{280}n^2 + 42\log n + 12n^3 + 280\sqrt{n}$$

(a) (2 points) For each of the following statements, indicate whether the statement is true or false.

1. Algorithm A is $O(\log n)$
2. Algoirthm A is $O(n^2)$
3. Algoirthm A is $O(n^3)$
4. Algoirthm A is $O(2^n)$

(b) (1 point) Can the time complexity of this algorithm be expressed using big-Θ notation? If so, what is it?

(a)

1. : False

2. : False

3. : True

4. : True

Because n^3 is the fastest growing term which should be the upper bound for the expression. However the $O(2^n)$ in Statement 4 is growing even faster than n^3, therefore, if n^3 is an upper bound, then 2^n must be upper bound too.

(b) Yes, it can be expressed as Big-Oh notation.

The notation is $\boxed{O(n^3)}$

Question 3 ():

If possible, simplify the following expressions. *Hint: See slide 11 of topic 3 of the lecture slides!*

(a) (1 point) $O(n^2) + O(\log n) + O(n \log n)$

(b) (1 point) $O(2^n) \cdot O(n^2)$

(c) (1 point) $42O(n \log n) + 18O(n)$

(d) (1 point) $O(n) + O(m)$ *(yes, that's an 'm', not a typo; note that m is independent of n)*

(a) $O(n^2) + O(\log n) + O(n \log n) = O(\max(n^2, \log n, n \log n)) = \boxed{O(n^2)}$

(b) $O(2^n) \cdot O(n^2) = \boxed{O(2^n \cdot n^2)}$

(c) $42O(n \log n) + 18O(n) = O(42n \log n) + O(18n) = O(n \log n) + O(n)$

$= O(\max(n \log n, n))$

$= \boxed{O(n \log n)}$

(d) $O(n) + O(m) = \boxed{O(n+m)}$

Question 4 ():

Consider the following Java code fragment:

```
1 // Print out all ordered pairs of numbers between 1 and n
2 for(i = 0; i <= n; i++) {                                  0
3     for(j = 0; j <= n; j++) {
4         System.out.println( i + ", " + j) ;
5     }
6 }
```

(a) (3 points) Use the *statement counting approach* to determine the exact number of statements that are executed when we run this code fragment as a function of n. Show all of your calculations.

(b) (1 point) Express the answer you obtained in part (a) in big-Θ notation (since the best and worst cases are the same – there is only one path of execution through this loop).

(a) For the inner loop:

① loop condition: 1
② print statement: 1
} + } 2 statements
X
③ the loop will execute n times: n
} 2n statements
+
④ An extra statement when the loop condition is false: 1
} 2n+1 statements

For the outer loop:

① # of inner loop statements: 2n+1
② loop condition: 1
} + } 2n+2 statements
X
③ the loop will execute n times: n
} n(2n+2)
+
④ an extra statement when the loop condition is false: 1
} n(2n+2)+1 statements

The total statements are: $\boxed{2n^2 + 2n + 1}$

(b) $O(2n^2+2n+1) = \boxed{O(n^2)}$

Question 5 ():

Consider the following pseudocode:

```
1  Algorithm roundRobinTournament(a)
2  This algorithm generates the list of matches that must be
3  played in a round-robin pirate-dueling tournament (a tournament where
4  each pirate duels each other pirate exactly once).
5
6  a is an array of strings containing names of pirates in the tournament
7
8  n = a.length
9  for i = 0 to n-1
10    for j = i+1 to n-1
11       print a[i] + " duels " + a[j] + ", Yarrr!"
```

Note: the pseudocode `for i = a to b` *means that the loop runs for all values of* `i` *between* `a` *and* `b`*, inclusive, that is, including the values* `a` *and* `b`*.*

(a) (6 points) Use the *statement counting approach* to determine the exact number of statements that are executed by this pseudocode as a function of `n`. Show all of your calculations.

(b) (1 point) Express the answer you obtained in part a) in big-Θ notation (since, again, the best and worst cases are the same).

(a) For the inner loop:

① loop condition: 1
② print statement: 1
} + } 2 statements

③ the loop will execute: $n-1-(i+1)+1 = n-i-1$

} X 2(n-i-1) statements + 1 statement

④ an extra statement when the loop condition is false: 1

} 2(n-i-1)+1 statements

For the outer loop:

① # of inner loop : $2(n-i-1)+1$

② the loop will execute n times: $n-1-i+1 = n$

④ Since this is Dependent Quadratic Nested Loops, we can use the summation notation as:

$$\sum_{i=0}^{n-1} [2(n-i-1)+1]$$

$$= 2 \times [(n-1)+(n-2)\cdots+1+0] + n$$

$$= 2 \sum_{i=0}^{n-1} (i+n)$$

$$= \frac{2(n-1)(n)}{2} + n$$

$$= n^2$$

(b) $O(n^2)$

Question 6 (3 points):

Using the active operation approach, determine the time complexity of the pseudocode in question 5. Show all your work and express your final answer in big-Θ notation.

The active operation is **line 10** from question 5

The cost of active operation is $O(1)$

The active operation will execute $(n-i-1)+1$ times

Therefore the total cost of active operation is:

$$\sum_{i=0}^{n-1} [O(1) \times ((n-i-1)+1)]$$

$$= \sum_{i=0}^{n-1} [(n-i-1)+1]$$

$$= \sum_{i=0}^{n-1} i + n$$

$$= \frac{n(n-1)}{2} + n$$

$$= \frac{n^2-n}{2} + n$$

$$= \frac{1}{2}n^2 - \frac{1}{2}n + n$$

$$= \frac{1}{2}n^2 + \frac{1}{2}n \in O(n^2)$$

$$\boxed{O(n^2)}$$

Question 7 (6 points):

Consider the following pseudocode.

```
1  Algorithm multiSearch( data, target ):
2  data:  a list of arrays of integers; in each array the
3         integers are sorted in ascending order; the list
4         'data' has a cursor.
5  target: an integer
6
7  // Iterate over the arrays in the list 'data' using
8  // its cursor:
9  data.goFirst()
10 found = false
11 while( !data.after() and !found ) {
12    // search for integer 'target' in A
13    found = binarySearch(data.currentItem(), target)
14    data.goForth()
```

Using the active operation approach to timing analysis determine the time complexity of this pseudocode in the **worst case**. Assume that the list of arrays contains n arrays and that each array has exactly m items in it. Be sure to clearly identify the line that is the active operation. Show all your work and express your final answer in Big-O notation (because we are doing a worst-case analysis).

line 12 is the active operation

The cost of active operation is $O(\log m)$

The worst case will be the target is not found in the arrays which means all n arrays need to be searched. Therefore, the active operation will execute n times. The corresponding big-Oh will be $O(n \cdot \log m)$

Question 8 (11 points):

A priority queue is a queue where a numeric priority is associated with each element. Access to elements that have been inserted into the queue is limited to inspection and removal of the elements with smallest and largest priority only. A priority queue may have multiple items that are of equal priority.

Give the ADT specification for a bounded priority queue using the specification method described in Topic 7 of the lecture notes. By "bounded", it is meant that the priority queue has a maximum capacity specified when it is created, and it can never contain more than that number of items.

Your specification must specify the following operations:

newPriorityQueue: make a new queue
insert: inserts an element with a certain priority
isEmpty: test if the queue is empty
isFull: test if the queue is full
maxItem: obtain the item in the queue with the highest priority
minItem: obtain the item in the queue with the lowest priority
deleteMax: remove from the queue the item with the highest priority
deleteAllMax: remove from the queue all items that are tied for the highest priority
deleteMin: remove from the queue the item with the lowest priority
frequency: obtain the number of times a certain item occurs in the queue (with **any** priority)

Queue Specifications

Name: PriorityQueue<G>

Sets:

Q: set of all queues containing elements from G

G: set of items can be in the queue Q

B: {true, false}

$\mathbb{N}$: set of positive integers

$\mathbb{N}_0$: set of non-negative integers

Signatures:

newPriorityQueue<G>: $\mathbb{N} \rightarrow Q$
Q.insert(g): $G \nrightarrow Q$
Q.isEmpty: $\rightarrow B$
Q.isFull: $\rightarrow B$
Q.maxItem: $\nrightarrow G$
Q.minItem: $\nrightarrow G$
Q.deleteMax: $\nrightarrow Q$

Q.deleteAllMax: ⇸ Q
Q.deleteMin: ⇸ Q
Q.frequency(g): ⟶ $\mathbb{N}_0$

Preconditions: $\forall q \in Q, g \in G$

new PriorityQueue<G>: None
q.insert(g): queue is not full
q.isEmpty: None
q.isFull: None
q.maxItem: queue is not empty
q.minItem: queue is not empty
q.deleteMax: queue is not empty
q.deleteAllMax: queue is not empty
q.deleteMin: queue is not empty
q.frequency(g): None

Semantics: $\forall q \in Q, g \in G, n \in \mathbb{N}$

new PriorityQueue<G>(n): create a new queue with capacity n
q.insert(g): insert item g into q with the largest priority number
q.isEmpty: return true if q is empty, false otherwise
q.isFull: return true if q is full, false otherwise
q.maxItem: return the highest priority item in q
q.minItem: return the lowest priority item in q
q.deleteMax: remove the highest priority item in q
q.deleteAllMax: remove all items in q that are tied for the highest priority
q.deleteMin: remove the lowest priority item in q
q.frequency(g): return the number of times item g occurs in the queue

---