CPSC 221: Algorithms and Data Structures
Midterm Exam, 2013 February 15

Name: ______________________ Student ID: ______________________

Signature: ______________________ Section (circle one): **MWF(201)** **TTh(202)**

- You have **60 minutes** to solve the **5** problems on this exam.
- A total of **100 marks** is available. You may want to complete what you consider to be the easiest questions first!
- Ensure that you clearly indicate a legible answer for each question.
- You are allowed up to three textbooks and (the equivalent of) a 3" 3-ring binder of notes as references. Otherwise, no notes, aides, or electronic equipment are allowed.
- Good luck!

P1	P2	P3	P4	P5	Total
15	15	15	15	40	100

This page intentionally left blank.
If you put solutions here or anywhere other than the blank provided for each solution, you must *clearly* indicate which problem the solution goes with and also indicate where the solution is at the designated area for that problem's solution.

1 The Big O[1] [15 marks]

Consider the following functions:

A. 42

B. $\lg(n^2)$

C. n^{100}

D. $2^{\lg n}$

E. 2^n

F. $5n + \lg n - 22 + \frac{1}{n}$

For each function, write down the LETTERs for the other functions which have that relationship to it. If no other functions have that relationship, draw a line through the box. I have done the first one for you as an example:

The function...	... is big-O of:	... is big-Ω of:	... is big-Θ of:
A	A, B, C, D, E, F	A	A
B			
C			
D			
E			
F			

[1]Feel free to ignore the problems' names!

2 Analyzing Runtime [15 marks]

1. Give a tight big-Θ runtime bound for the following function. You do not need to show your work.

```
int hello() {
  cout << "Hello, World!" << endl;
  return 42;
}
```

2. Give a tight big-Θ runtime bound for the following function in terms of n. You do not need to show your work.

```
int mult(int m, int n) {
  int sum = 0;
  while (n > 0) {
    if (n%2 == 1) sum += m;
    n /= 2;
    m *= 2;
  }
  return sum;
}
```

3. Give recurrence relations for the runtime for the following function. Your answer **must** reflect the recursive structure of the code.

```
int mult2(int m, int n) {
  if (n==0) return 0;
  else {
    return m+mult2(m, n-1);
  }
}
```

4. Is `mult2` tail recursive?

5. Solve the recurrence relation you gave above to get a tight big-Θ runtime bound for the function `mult2` in terms of n. Show your work if you want partial credit.

3 Big-O Proofs [15 marks]

For this problem, you must give formal proofs.

1. Prove that $n + 1000 \in O(n)$.

2. Prove that $3^n \notin O(2^n)$.

3. In lecture, we defined that $f(n) \in \Omega(g(n))$ iff $g(n) \in O(f(n))$. Call this definition Ω_1. In the Epp textbook, she defines that $f(n) \in \Omega(g(n))$ iff there exists a positive real number A and a nonnegative real number a such that: $A|g(n)| \leq |f(n)|$ for all numbers $n > a$. Call this definition Ω_2. Prove that these two definitions are the same, in other words that $f(n) \in \Omega_1(g(n))$ iff $f(n) \in \Omega_2(g(n))$. You may assume $f(n)$ and $g(n)$ are always positive.

   To make the problem easier, I am giving you the complete proof. However, the lines of the proof are scrambled. You must put them into a correct order. **Give your answer by listing the letters for each statement, in the order you use them.**

   (A) Because A, $g(n))$, and $f(n)$ are all positive, we can remove the absolute values and divide by A, yielding $g(n) \leq (1/A)f(n)$ for all $n > a$.

   (B) By choosing $n_0 = a$ and $c = (1/A)$, we see that $g(n) \in O(f(n))$

   (C) By the definition of big-O, there exists $n_0 > 0$ and $c > 0$ such that for all $n > n_0$, $g(n) \leq cf(n))$.

   (D) Dividing both sides by c, we get $(1/c)g(n) \leq f(n)$.

   (E) Let $A = 1/c$ and let $a = n_0$.

   (F) QED

   (G) Since $f(n)$ and $g(n)$ are always positive, we can add absolute values without changing the inequality.

   (H) Since $f(n) \in \Omega_1(g(n))$, then $g(n) \in O(f(n))$.

   (I) Since $f(n) \in \Omega_2(g(n))$, then there exists a positive real number A and a nonnegative real number a such that: $A|g(n)| \leq |f(n)|$ for all numbers $n > a$.

   (J) So we get $A|g(n)| \leq |f(n)|$ for all numbers $n > a$, which means that $f(n) \in \Omega_2(g(n))$.

   (K) Substituting in the above inequality, we get $Ag(n) \leq f(n)$ for all $n > a$.

   (L) The statement is an "if and only if", so we prove it in two separate parts.

   (M) Therefore, $f(n) \in \Omega_1(g(n))$.

   (N) We assume that $f(n) \in \Omega_1(g(n))$ and aim to prove that implies $f(n) \in \Omega_2(g(n))$.

   (O) We assume that $f(n) \in \Omega_2(g(n))$ and aim to prove that implies $f(n) \in \Omega_1(g(n))$.

   (P) **Proof of the $\Rightarrow$ Direction:**

   (Q) **Proof of the $\Leftarrow$ Direction:**

4 Stacks and Queues [15 marks]

In lecture, I mentioned a new method of computing using DNA molecules, in which demonstrating the ability to implement stacks was key to showing its computational generality. Imagine you are in the future, building the programming tools for such a machine. (You don't need to know anything about DNA for this problem!)

Because of the underlying molecular computing engine, this future computer lets you declare and use `int` variables, as well as an `IntStack` class that provides three methods (as well as the usual constructors and destructors):

```
void push(int);  // Pushes an int onto the stack.
int pop();  // Pops an int value off the stack and returns it.
int isEmpty();  // Returns 1 (true) if stack is empty; 0, otherwise.
```

The goal is to implement an `IntQueue` class, using nothing but `int` and `IntStack` variables. Here's most of the definition, including an implementation of the `enqueue` method:

```
class IntQueue {
  IntStack s; // Storage for items in queue
public:
  IntQueue();
  ~IntQueue();
  void enqueue(int);
  int dequeue();
  int isEmpty();
}
...
void IntQueue::enqueue(int n) {
  // I'll just push it onto the stack.
  // We can worry about how to dequeue later.
  s.push(n);
}
```

For this problem, your job is to implement the `dequeue` method. You may assume `dequeue` will not be called on an empty queue. (Hint: Declare a second stack, or else use recursion to use the call stack.) **Write your code here or on the next page.**

(You shouldn’t need the whole page.)

5 Pre-Treaps [40 marks]

In this problem, we will consider implementing a heap using an ordinary binary tree (with pointers), instead of using the "nifty storage trick" that we used to pack a nearly complete tree into an array. Specifically, we will build our binary tree out of these `Node` structures:

```
struct Node {
  int priority;
  Node * left;
  Node * right;
};
```

The resulting binary tree is required to obey the heap order property (parent has smaller or equal priority value than its children), but **not** the heap shape property (it's not necessarily a nearly complete tree).

The following code performs an `insert` into the heap. Because we don't have the nearly-complete shape, the insertion happens at a random leaf, and then a `percolate-up` operation is performed. We assume we have a function `rand()` that returns 0 or 1 randomly with equal probability:

```
void insert(Node *& heap, int p) {
  // Inserts a new node with priority p into the heap.
  if (heap==NULL) {
    heap = new Node();
    heap->priority = p;
    return;
  }
  if (rand()) {
    insert(heap->left, p);
    if (heap->left->priority < heap->priority) {
      int tmp = heap->left->priority;
      heap->left->priority = heap->priority;
      heap->priority = tmp;
    }
  } else {
    insert(heap->right, p);
    if (heap->right->priority < heap->priority) {
      int tmp = heap->right->priority;
      heap->right->priority = heap->priority;
      heap->priority = tmp;
    }
  }
}
```

1. Give a recurrence relation for the worst case run time of the `insert` function, in terms of the height h of the tree. Define the height of an empty tree (i.e., NULL) to be -1.

2. Give a tight big-O bound on the worst case run time for the `insert` function, as a function of the height h of the tree. You do not need to prove your result.

3. Prove that the `insert` function maintains the heap order property. (Hint: This is recursive, so your proof should be a short, inductive proof.) Give your base case here:

4. Now, complete your proof by giving the inductive case:

5. Consider this recursive function, which is supposed to find the largest priority value in the heap:

```
// Pre-Condition:  Never called on an empty tree (NULL)
int buggyFindMax(Node * heap) {
  if ((heap->left==NULL) && (heap->right==NULL)) return heap->priority;
  if (heap->left==NULL) return buggyFindMax(heap->right);
  if (heap->right==NULL) return buggyFindMax(heap->left);
  if (heap->left->priority > heap->right->priority)
    return buggyFindMax(heap->left);
  else
    return buggyFindMax(heap->right);
}
```

Draw a small tree that obeys the heap order property (min value at root), but where `buggyFindMax` will **not** return the largest priority in the tree.

6. Convert `buggyFindMax` to be iterative. (Do **not** try to fix the bug. Your code should do the exact same computation as the recursive `buggyFindMax`, but not use recursion at all.)

This page intentionally left blank.
If you put solutions here or anywhere other than the blank provided for each solution, you must *clearly* indicate which problem the solution goes with and also indicate where the solution is at the designated area for that problem's solution.