CPSC 221: Algorithms and Data Structures
Midterm Exam, 2014 February 27

Name: _____ Student ID: _____

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

- You have **65 minutes** to solve the **6** 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!

UNIVERSITY REGULATIONS

1. Each candidate must be prepared to produce, upon request, a UBCcard for identification.
2. Candidates are not permitted to ask questions of the invigilators, except in cases of supposed errors or ambiguities in examination questions.
3. No candidate shall be permitted to enter the examination room after the expiration of one-half hour from the scheduled starting time, or to leave during the first half hour of the examination.
4. Candidates suspected of any of the following, or similar, dishonest practices shall be immediately dismissed from the examination and shall be liable to disciplinary action:
   - having at the place of writing any books, papers or memoranda, calculators, computers, sound or image players/recorders/transmitters (including telephones), or other memory aid devices, other than those authorized by the examiners;
   - speaking or communicating with other candidates; and
   - purposely exposing written papers to the view of other candidates or imaging devices. The plea of accident or forgetfulness shall not be received.
5. Candidates must not destroy or mutilate any examination material; must hand in all examination papers; and must not take any examination material from the examination room without permission of the invigilator.
6. Candidates must follow any additional examination rules or directions communicated by the instructor or invigilator.

P1	P2	P3	P4	P5	P6	Total
15	25	10	15	20	15	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]

In this problem, we will consider the asymptotic behaviour of functions as n goes to infinity. For each pair of functions, fill in the **LETTER** from this list that best describes their relationship:

A. ... big-O but not big-Ω (big-Omega) ...

B. ... big-Ω (big-Omega) but not big-O ...

C. ... big-Θ (big-Theta) ...

D. ... neither big-O nor big-Ω (big-Omega) ...

(Scoring will be based on how many correct answers you give *minus the number of wrong answers you give*, so random guessing isn't expected to be helpful unless you are more than 50% confident in your answer.)

We have done the first one for you, as an example.

1. **Example:** The function n is _____ C _____ of the function n.

2. The function $0.0001n$ is _____ of the function $(6.02 \times 10^{23})^{\lg(100!)}$.

3. The function 3^n is _____ of the function 4^n.

4. The function $n!$ is _____ of the function 4^n.

5. The function n^6 is _____ of the function n^5.

6. The function $\lg(n^6)$ is _____ of the function $\lg(n^5)$.

7. The function n^6 is _____ of the function $(1.001)^n$.

8. The function n^2 is _____ of the function $n \log n$.

9. The function $n^2 + 2^n + 37n^3 \log n$ is _____ of the function $n \log n + 2n^2 + 3n^3 + n!$.

10. The function $n^2(n - 2\lfloor n/2 \rfloor)$ is _____ of the function $2\lfloor n/2 \rfloor$.

11. The function $3n^3 + 18 \log n$ is _____ of the function $n \log n + 2n^2 + 4n^3$.

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

2 Analyzing Code [25 marks]

1. Give tight big-O and big-Ω runtime bounds for the following function in terms of n. You do not need to show your work (but it might help with partial credit).

```
int count_factors(int n) {
  int result = 0;
  for (int i=1; i<n; i++) {
    if (n%i == 0) result++;
  }
  return result;
}
```

2. Give tight big-O and big-Ω runtime bounds for the following function in terms of n. You do not need to show your work (but it might help with partial credit).

```
int count_perfect(int n) {
  int result = 0;
  for (int i=1; i<=n; i++) {
    int sum=0;
    for (int j=1; j<i; j++) {
      if (i%j == 0) sum += j;
    }
    if (sum==i) result++;
  }
  return result;
}
```

3. Give tight big-O and big-Ω runtime bounds for the following function in terms of n. You do not need to show your work (but it might help with partial credit).

```
int useless(int n) {
  int result = 0;
  for (int i=0; i*i < n; i++) {
    int j=1;
    while (j < n) {
      result++;
      j += j;
    }
  }
  return result;
}
```

4. Give tight big-O and big-Ω runtime bounds for the following function in terms of n. You do not need to show your work (but it might help with partial credit). For this problem, it's helpful to know that there are an infinite number of even numbers, and an infinite number of prime numbers. (A *prime number* is a number that is evenly divisible by only 1 and itself. For example, 7 is prime, because it's only divible by 1 and 7, but 6 is not prime, because it's also divisible by 2 or 3.) In other words, for any n_0, there are always even and prime numbers bigger than n_0.

```
int prime_tester(int n) {
  for (int i=2; i*i < n; i++) {
    if (n%i == 0) return 0;
  }
  return 1;
}
```

5. Give a big-Θ bound on the **space** used by the following function, in terms of n. You do not need to show your work (but it might help with partial credit).

```
int triangle_numbers(int n) {
  if (n < 2) return n;
  else return n + triangle_numbers(n-1);
}
```

6. Is the function `triangle_numbers` tail recursive?

7. Convert the function `count_factors` from part 1 of this problem into code *that has NO loops at all* — no `for`-loops, no `while`-loops, no `do`-loops, etc. (Hint: You'll want to use recursion!) You may define helper functions if you'd like.

3 Big-Θ Proofs [10 marks]

Here is the same `prime_tester` function that you saw in part 4 of the preceding question (Question 2 Analyzing Code):

```
int prime_tester(int n) {
  for (int i=2; i*i < n; i++) {
    if (n%i == 0) return 0;
  }
  return 1;
}
```

Prove that the asymptotic runtime of this function is *not* $\Theta(1)$. You must give a formal proof, based on the formal definition of big-Θ.

4 Cabbages and Goats and Wolves, Oh My! [15 marks]

In the first programming project, you developed code for a universal puzzle solver. One of the puzzles we gave you is the classic Wolf-Goat-Cabbage puzzle.[2]

Below is a numbered list of all legal states of the puzzle. In this problem, we will refer to the states by their numbers. For example, the puzzle always starts in State 9, where everything is on the left bank, and the goal is to reach State 0, where everything is on the right bank.

State Number	Left Bank	River	Right Bank
0		vvvvvvvvvv	Wolf Goat Cabbage Boat
1	Cabbage	vvvvvvvvvv	Wolf Goat Boat
2	Goat	vvvvvvvvvv	Wolf Cabbage Boat
3	Wolf	vvvvvvvvvv	Goat Cabbage Boat
4	Wolf Cabbage	vvvvvvvvvv	Goat Boat
5	Goat Boat	vvvvvvvvvv	Wolf Cabbage
6	Goat Cabbage Boat	vvvvvvvvvv	Wolf
7	Wolf Cabbage Boat	vvvvvvvvvv	Goat
8	Wolf Goat Boat	vvvvvvvvvv	Cabbage
9	Wolf Goat Cabbage Boat	vvvvvvvvvv	

You may find it helpful to draw a diagram of which states are connected to which states. For example, from the initial State 9, the only legal move is to go to State 4, so you could draw a circle labeled 9, with an arrow to a circle labeled 4. We will not mark you on this drawing (it is optional), but here is space to draw it if you'd like.

[2] For this problem, we have not changed the puzzle in any way. If you do not remember the puzzle, the rules are as follows. You are trying to get a wolf, a goat, and some cabbage across a river, with a boat that can carry yourself and at most one other item at a time. You're not allowed to leave the wolf and goat together unattended, nor the goat and cabbage.

Here is the `solvePuzzle` function from `solve.cpp` in the project (with some irrelevant parts omitted).

```
// This function does the actual solving.
void solvePuzzle(PuzzleState *start, TodoList<PuzzleState*> &active,
      PredDict<PuzzleState*> &seen, vector<PuzzleState*> &solution) {

  PuzzleState *state;
  PuzzleState *temp;

  active.add(start); // Must explore the successors of the start state.
  seen.add(start,NULL); // We've seen this state.  It has no predecessor.

  while (!active.is_empty()) {
    // Loop Invariants:
    // 'seen' contains the set of puzzle states that we know how to reach.
    // 'active' contains the set of puzzle states that we know how to reach,
    //    and whose successors we might not have explored yet.

    state = active.remove();    // ***** THIS IS THE MARKED LINE *****

    if (state->isSolution()) {
      // Found a solution!
      // ... omitted for brevity ...
      return;
    }

    vector<PuzzleState*> nextMoves = state->getSuccessors();
    for (unsigned int i=0; i < nextMoves.size(); i++) {
      if (!seen.find(nextMoves[i], temp)) {
        // Never seen this state before.  Add it to 'seen' and 'active'
        active.add(nextMoves[i]);
        seen.add(nextMoves[i], state);
      } else {
        delete nextMoves[i];
      }
    }
  }

  // Ran out of states to explore.  No solution!
  solution.clear();
  return;
}
```

In this problem, you will trace how the puzzle gets solved, depending on which ADT you use for `active`. You should assume that `start` is initialized to State 9, and that `seen` is a (correctly implemented) `BSTDict`. Also, assume that `getSuccessors()` always returns the possible successor states in increasing numerical order. For example, suppose that the legal successors of State 9 were states 1, 2, and 3, then the vector `nextMoves` would contain State 1, State 2, and State 3, in that order. For each part of the problem, you will write down the sequence of states that are removed from `active` at the line marked `THIS IS THE MARKED LINE` as the program runs.

1. What sequence of states do you get if `active` is a **stack**?

   Write your answer here: _____

2. What sequence of states do you get if `active` is a **queue**?

   Write your answer here: _____

3. What sequence of states do you get if `active` is a **(min) priority queue** that compares the state numbers?

   Write your answer here: _____

5 Thgieh [20 marks]

Trees are defined recursively. We can define a binary tree as either an empty tree, or a node with two subtrees, each of which is a binary tree.

Define the *thgieh* ("height" spelled backwards) of a tree to be the length of the **shortest** path from the root to a node with an empty subtree. (In contrast, a tree's height is the length of the **longest** such path.) Mathematically, the thgieh of an empty tree is -1, and the thgieh of any other tree is equal to 1 plus the smaller of the thgiehs of its two subtrees.

Now, consider this definition for a tree node:

```
// A tree is one of an empty tree (NULL) or a Node with two subtrees.
class Node {
public:
   // Constructor that takes at least two arguments (an initial key and
   // value) and optionally initial left and right subtree pointers.
   Node(int _key, int _value, Node* _left = NULL, Node* _right = NULL) {
     key = _key;
     value = _value;
     left = _left;
     right = _right;
   }
   int key;
   int value;
   Node* left;
   Node* right;
};
```

1. Complete the C++ function `thgieh` to calculate the thgieh of a tree:

```
// precondition: root points to a valid tree
// postcondition: the tree is unchanged; returns the thgieh of the
//                tree (defined to be -1 for empty trees)
int thgieh(Node* root) {
   // TODO: fill in this function!









}
```

2. Next, we will prove by structural induction that a tree t with $\mathrm{thgieh}(t) = k$ must have at least $2^{k+1} - 1$ nodes. *Your proof must be a structural induction, following the pattern we give here.* First, fill in the base case here, when the tree is empty.

3. Now, give the inductive case, for handling an arbitrary tree t.

6 Convertible Thgiehs [15 marks]

1. For this problem, assume that your `thgieh` from the preceding problem works correctly and has asymptotic performance in $\Theta(n)$, where n is the number of nodes in the tree.

   The following function converts a tree into a form with the invariant: the thgieh of the left subtree of a node is always at least as large as the thgieh of the right subtree.

```
void convert(Node* root) {
   if (root == NULL) {
     return;
   }
   else {
     int thgiehL = thgieh(root->left);
     int thgiehR = thgieh(root->right);
     if (thgiehR > thgiehL) {
       Node* temp = root->right;
       root->right = root->left;
       root->left = temp;
     }
     convert(root->right);
     convert(root->left);
   }
}
```

   Mark up the code with any annotations you need and then provide the recurrence $T(n)$ indicating the amount of time it takes to run `convert` on a tree with n nodes in the best case. (Note: the best case happens when the left and right subtrees have equal—or nearly equal for a tree with an even number of nodes total—sizes.)

   $T(0) =$ _____

   For $n > 0$, $T(n) =$ _____

2. Solve the following similar recurrence. (We do not ask you to solve your recurrence from the previous part in case it is incorrect.) (You may assume the division is either integer division or real division.)

$$\begin{aligned} T(0) &= 3 \qquad \text{for } n \leq 1 \\ T(n) &= 3T(n/3) + n/3 \qquad \text{for } n > 1 \end{aligned}$$

3. What is the *worst-case* asymptotic performance of `convert` when called on a tree of size n? Why?

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.

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.