CPSC 221: Algorithms and Data Structures
Midterm Exam, 2015 October 21

Name: _____ Student ID: _____

Signature: _____ Section (circle one): **MWF(101)** **TTh(102)**

- You have **90 minutes** to solve the **8** problems on this exam.
- A total of **75 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 **one** sheet of paper (US Letter size or metric A4 size) of notes. **Besides this, 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	P7	P8	Total
12	11	9	8	5	10	10	10	75

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 Asymptotic Relationships [12 marks]

Consider the following functions:

A. 100

B. $\lg(n)/\log(n)$

C. $3n^5$

D. $\frac{5n^{5+\lg n}}{n^{0.1}}$

E. $\lg(n!)$

F. $100n + 2n\lg(n)$

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 B	A B
B	A B C D E F	A B	A B
C	C D	A B C E F	C
D	D	A B C D E F	D
E	C D E F	A B E F	E F
F	C D E F	A B E F	E F

2 Big-Θ Proof [11 marks]

For this problem, you must **formally prove** that $\lfloor n/10 \rfloor \in \Theta(n)$. Since this is a formal proof, it will be based on the definition of big-Θ, and since that definition has two main parts, **you should clearly label those two major parts of your proof.**

(If you don't remember, recall that the floor function $\lfloor x \rfloor$ computes the largest integer less than or equal to x. In other words, it "rounds x down" to the closest integer. You might find it helpful in your proof to note that $x - 1 < \lfloor x \rfloor \leq x$.)

O: $f(n) \in O(g(n))$ means there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

$\lfloor n/10 \rfloor \leq n/10 = \left(\frac{1}{10}\right) n$ for all $n \geq 1$ ← c

Ω: $f(n) \in \Omega(g(n))$ means there exist positive constants c and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$

$\lfloor n/10 \rfloor > n/10 - 1 \geq \left(\frac{1}{20}\right) n$ for all $n \geq 20$ ← c

(since $2n - 20 \geq n$ for all $n \geq 20$)

Since $\lfloor n/10 \rfloor \in O(n)$ and $\lfloor n/10 \rfloor \in \Omega(n)$,

$\lfloor n/10 \rfloor \in \Theta(n)$

3 Big Factoring [9 marks]

Computing *prime factorizations* (i.e., the set of prime numbers that multiply together to equal the given number) is an important problem in cryptography.

(Recall that a *prime number* is a number that is evenly divisible by only 1 and itself. For example, 2, 3, and 5 are all prime numbers, but 6 isn't, because 6 is evenly divisible by 2 (as well as by 3).)

The following code computes the prime factorization of a number. (In real life, more efficient algorithms are used, but this is simple enough for us to analyze.)

```
void factor(int n) {
  while (n>1) {
    for (int i=2; i<=n; i++) {
      // The next line checks whether n is evenly divisible by i
      if (n%i == 0) {
        // If so, than i is a prime factor.
        cout << i << endl;  // Print i out
        n = n/i;    // Factor i out of n
        break;      // and break out of the for loop
      }
    }
  }
  return;
}
```

1. Give a big-Θ runtime bound in terms of n for `factor`, **under the assumption that n is prime**. You do not need to show your work (but it might help with partial credit). (Hint: Think carefully about how the loops and if statements will behave when n is prime.)

When n is prime, $n \% i == 0$ only when $i = n$.

$\Theta(n)$

2. Give a big-Θ runtime bound in terms of n for `factor`, **under the assumption that n is a power of 2** (in other words, that $n = 2^k$ for some integer k). You do not need to show your work (but it might help with partial credit). (Hint: Think carefully about how the loops and if statements will behave when n is a power of 2.)

When $n = 2^k$, the for loop decreases n to $n/2$ in one step.

$\Theta(\log n)$

4 Tribonacci Numbers [8 marks]

The *Tribonacci numbers* are a straightforward extension to Fibonacci numbers, where

$\text{Trib}(n) = \text{Trib}(n-1) + \text{Trib}(n-2) + \text{Trib}(n-3)$

and $\text{Trib}(0) = 0$, $\text{Trib}(1) = 0$, and $\text{Trib}(2) = 1$.

1. Write down the next three Tribonacci numbers:

   Trib(3) = 1 Trib(4) = 2 Trib(5) = 4

2. The following code computes Tribonacci numbers:

```
int trib(int n) {
  if (n==0) return 0;
  if (n==1) return 0;
  if (n==2) return 1;
  return trib(n-1) + trib(n-2) + trib(n-3);
}
```

   Write down recurrence relations for the **runtime** of `trib`. Your answer **must** reflect the recursive structure of the code.

$$T(n) \leq \begin{cases} c & \text{for } n \leq 2 \quad (c \text{ constant}) \\ c + T(n-1) + T(n-2) + T(n-3) & \text{for } n > 2 \end{cases}$$

3. Write down recurrence relations for the **space** used by `trib`. Your answer **must** reflect the recursive structure of the code. (You may assume that the space used by `trib(n-1)` is greater than the space used by `trib(n-2)`, and so on.) "Space" here means "maximum call stack size"

$$S(n) \leq \begin{cases} 0 & \text{for } n \leq 2 \\ 3 + S(n-1) & \text{for } n > 2 \end{cases}$$

4. Give a tight big-Θ bound on the **space** complexity of `trib` in terms of n. You do not need to explain or prove your answer, but if you do provide a brief explanation, we might be able to give you partial credit.

   $\Theta(n)$

5. Which recursive calls are tail calls? Circle your answer(s) here:

   `trib(n-1)` `trib(n-2)` `trib(n-3)` none of them

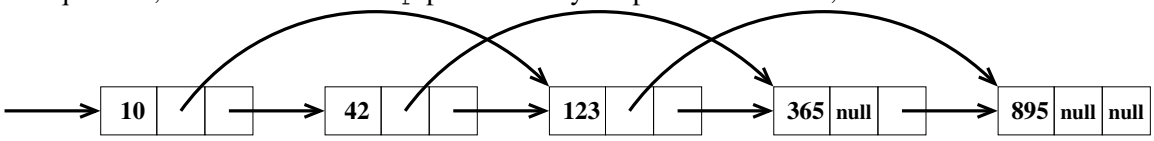
5 Express Lists I [5 marks]

One of the drawbacks of linked lists is that there's no fast way to get through the list, other than by visiting each node one-by-one, in order. In this question (and subsequent questions), we'll explore an idea to have a linked list that can run faster by hopping over some nodes in the list. (What we'll do on this exam isn't a great idea, but it's similar to something we might see later in the course, which really is cool.)

The idea is that each node will have — in addition to the usual `next` pointer that points to the next node in the list — a pointer `hop` that points to a node farther down the linked list. So, the declaration for the Node datatype is:

```
struct Node {
  int data;
  Node * hop;
  Node * next;
};
```

In this question, let's assume the `hop` pointer always hops over one node, so the list would look like this:



Your task on this question is to write code that will correctly delete a node from a list like this, restoring all the pointers to point to the correct places. To make your programming easier, we will assume that your function will receive as parameters a pointer to the node to delete, as well as pointers to the (immediate) predecessor of the node to delete, and the "express" predecessor, i.e., the node whose `hop` pointer points to the node being deleted.

Your code must run in constant time and space. Be sure not to leak memory.

```
void delete_node(Node * target, Node * pred, Node * hop_pred) {
\\ target is the node to be deleted
\\ pred is the node that points to target through its next pointer
\\ hop_pred is the node that points to target through its hop pointer

\\ YOUR CODE GOES HERE
        if ( target == NULL ) return ;
        if ( pred ) { pred->next = target->next ;
                      pred->hop = target->hop ; }
        if (hop_pred) { hop_pred->hop = target->next ; }
        delete target;
}
```

6 Express Lists II [10 marks]

In this question, we will continue working with the "Express List" data structure from Question 5, but you don't need to have solved that question to do this one. (You do need to read it, though!)

In addition, in this question, we will assume that the integers in the list are in increasing order.

The following function returns a node in the express list whose data field matches the specified key, or NULL if there is no such node:

```
Node * find_key(Node * list, int key) {
  if (list == NULL) return NULL;

  // Take the hop pointers through the list as far as possible.
  Node * cur = list;
  while ( (cur->hop != NULL) && (cur->hop->data < key) ) {
    cur = cur->hop;
  }

  // Now, follow the next pointers as usual.
  while ( (cur != NULL) && (cur->data <= key) ) {
    if (cur->data == key) return cur;
    cur = cur->next;
  }
  return NULL;
}
```

1. Give tight big-O and big-Ω bounds on the **worst**-case running time of `find_key`, in terms of the size n of the list. You do not need to prove your answer, but briefly explain your reasoning.

   $O(n)$ and $\Omega(n)$ since following a hop or next pointer advances the cur position in the list by a constant amount (≤ 2) and, in the worst case, we search for a very large key.

2. Give tight big-O and big-Ω bounds on the **best**-case running time of `find_key`, in terms of the size n of the list. You do not need to prove your answer, but briefly explain your reasoning.

   $O(1)$ and $\Omega(1)$ we could find the key in the first Node.

3. Now, assume that instead of hopping over 1 node, the `hop` pointer hopped over 9 nodes. Give tight big-O and big-Ω bounds on the **worst**-case running time of `find_key`, in terms of the size n of the list. You do not need to prove your answer, but briefly explain your reasoning.

$O(n)$ and $\Omega(n)$

4. Now, assume that instead of hopping over 9 nodes, the `hop` pointer hopped over $\sqrt{n}$ nodes, where n is the total length of the list. Give tight big-O and big-Ω bounds on the **worst**-case running time of `find_key`, in terms of n. You do not need to prove your answer, but briefly explain your reasoning.

$O(\sqrt{n})$ and $\Omega(\sqrt{n})$

Only $\sqrt{n}$ "hops" can be executed in the first while loop (since $\sqrt{n} \cdot \sqrt{n} = n$) and only $\sqrt{n}$ "nexts" can be executed in the second while loop (since at most $\sqrt{n}$ Nodes lie between cur and cur->hop and key must lie in this range).

5. Now, assume that instead of hopping over $\sqrt{n}$ nodes, the `hop` pointer hopped over $n/2$ nodes, where n is the total length of the list. Give tight big-O and big-Ω bounds on the **worst**-case running time of `find_key`, in terms of n. You do not need to prove your answer, but briefly explain your reasoning.

$O(n)$ and $\Omega(n)$

$n/2$ Nodes lie between cur and cur->hop
So the second while loop takes $\Theta(n)$
in worst case.

7 Express Lists III [10 marks]

Convert the `find_key` function from the preceding question (Question 6) so that it doesn't use any iteration — no while-loops, no for-loops, no do-loops, no loops of any kind. (Hint: Use recursion. Your solution should have a total of three functions.)

```
Node *hop(Node *curr, int key) {
    if ((curr->hop != NULL) &&
        curr->hop->data < key)
        return hop(cur->hop, key);
    else
        return cur;
}

Node *next(Node *curr, int key) {
    if (curr != NULL && curr->data < key)
        return next(curr->next, key);
    else if (curr == NULL || curr->data > key)
        return NULL;
    else
        return curr;
}

Node *find_key(Node *list, int key) {
    if (list == NULL) return NULL;
    return next(hop(cur, key), key);
}
```

8 Priority Queues Using Queues [10 marks]

For this question, your challenge is to implement a priority queue with only a **single** (normal, non-priority) **queue** in which to store the data. You are also allowed to declare `int` variables, but not arrays, vectors, pointers, or any other data structures. You also are not allowed to call `new`, nor use recursion.

The `Queue` class provides these methods:

```
void enqueue(int);  // Puts an int into the queue.
int dequeue();  // Takes the int from the front of the queue and returns it.
int size();  // Returns the number of elements stored in the queue.
```

Here's most of the code for the `PriorityQueue` class, including an implementation of the `insert` method. Your task is to write the `deleteMin` method.

```
class PriorityQueue {
  Queue q; // Storage for items in the priority queue
  // Note that you are NOT allowed to declare additional Queues
public:
  PriorityQueue();
  ~PriorityQueue();
  void insert(int);
  int deleteMin(); // Takes the smallest int out of the queue and returns it.
  int isEmpty();
}
...
void PriorityQueue::insert(int n) {
  // I'll just enqueue it, not worrying about order.
  // deleteMin can worry about finding the smallest item.
  q.enqueue(n);
}

int PriorityQueue::isEmpty() { return (s.size() == 0); }

int PriorityQueue::deleteMin() {
  // YOUR CODE GOES HERE (AND POSSIBLY ON THE NEXT PAGE)...
```

```
    int n = q.size();
    int x = q.dequeue();
    for(int i=1; i<n; ++i) {
      int y = q.dequeue();
        if(y < x) {q.enqueue(x); x=y;}
        else      q.enqueue(y);
    }
    return x;
}
```

(additional space if needed for Priority Queues Using Queues)

Briefly explain why this question would be impossible to solve if you had been allowed only a **single stack** instead of a single queue (with the same restrictions about no other data structures and no recursion).

deleteMin may require access to the bottom element in the stack. All elements must be removed (popped) to access the bottom element but there is no additional memory to hold them.

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.