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

SAMPLE SOLUTIONS

These are sample solutions to the group version of the exam. The individual exam had some very small differences, but they are trivial enough that we won't be releasing separate solutions.

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 ...big-Ω (big-Omega) but not big-O... of the function $(6.02 \times 10^{23})^{\lg(100!)}$.

3. The function 3^n is ... big-O but not big-Ω (big-Omega) ... of the function 4^n.

4. The function $n!$ is ... big-Ω (big-Omega) but not big-O ... of the function 4^n.

5. The function n^6 is ... big-Ω (big-Omega) but not big-O ... of the function n^5.

6. The function $\lg(n^6)$ is ... big-Θ (big-Theta) ... of the function $\lg(n^5)$.

7. The function n^6 is ... big-O but not big-Ω (big-Omega) ... of the function $(1.001)^n$.

8. The function n^2 is ... big-Ω (big-Omega) but not big-O ... of the function $n \log n$.

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

9. The function $n^2 + 2^n + 37n^3 \log n$ is ... big-O but not big-Ω (big-Omega) ... of the function $n \log n + 2n^2 + 3n^3 + n!$.

10. The function $n^2(n - 2\lfloor n/2 \rfloor)$ is ... neither big-O nor big-Ω (big-Omega) ... of the function $2\lfloor n/2 \rfloor$.

11. The function $3n^3 + 18 \log n$ is ... big-Θ (big-Theta) ... of the function $n \log n + 2n^2 + 4n^3$.

Solution : The point of this problem was to test your knowledge of some common asymptotic complexities, the simple tricks for simplifying expressions to their asymptotic complexity order, a bit of commonly used algebra, and the definitions of big-O, -Ω, and -Θ.

As discussed on Piazza, we were quite surprised that a fraction of students were confused by the directionality of the answers. This is completely standard usage, so you need to be comfortable with how to read, write, say, and understand descriptions of big-O/Ω/Θ relationships, not just the mathematical notation. To make the answers easier to read, we have substituted the correct text directly into the problems.

If you don't understand the answers to any of these, **be sure to ask a professor or TA, in office hours, or on Piazza!** This is fundamental to the course.

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;
}
```

Solution : The loop goes around $n - 1$ times (with i going from 1 to $n - 1$), with a constant amount of work in each loop body, so this is $O(n)$ and $\Omega(n)$, or equivalently $\Theta(n)$.

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;
}
```

Solution : The outer loop has i going from 1 to n, and the inner loop has j going from 1 to $i-1$, with a constant amount of work in the inner loop body. If you were being super careful, you'd end up building an expression like (where C_1, C_2, and C_3 are constants):

$$\begin{aligned}
T(n) &= C_1 + \sum_{i=1}^{n} \left(C_2 + \sum_{j=1}^{i-1} C_3 \right) \\
&= C_1 + C_2 n + \sum_{i=1}^{n} \sum_{j=1}^{i-1} C_3 \\
&= C_1 + C_2 n + \sum_{i=1}^{n} (i-1) C_3 \\
&= C_1 + C_2 n + C_3 \sum_{i=1}^{n} (i-1) \\
&= C_1 + C_2 n + C_3 \frac{n(n-1)}{2} \\
&\in \Theta(n^2)
\end{aligned}$$

With practice, we'd hope most of you would just look at the code, see the pattern of the outer loop going from 1 to n, and the inner loop from 1 to i, know that you'd get the nested summation described above, and remember that we've done this summation several times in class, and that it's $\Theta(n^2)$ (and therefore $O(n^2)$ and $\Omega(n^2)$.

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;
}
```

Solution : Similarly to the previous problem, we're testing your ability to read code and understand what it's doing, in addition to your understanding of the asymptotics. You should see that the outer loop goes from $i = 0$ up until $i^2 \approx n$, or about $\sqrt{n}$ times. The inner loop has j doubling on each iteration, so it will go around about $\lg(n)$ times. So, the total runtime is $\Theta(\sqrt{n} \log n)$.

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;
}
```

Solution : Again, you need to be able to read short bits of code like this and understand what the code is doing. Most people saw that the i is counting up until it reaches approximately $\sqrt{n}$, so the code can't take more than $O(\sqrt{n})$ time. However, note that if n is evenly divisible by i, the loop terminates early. So, it might not take $\Omega(\sqrt{n})$ iterations. In fact, for *most* values of n, the loop will go around only a constant number of times! For example, for all even values of n, the loop will exit on the very first iteration. For odd values of n, 1 out of every 3 of those will be divisible by 3, so the loop will execute only twice, no matter how big n is. (BTW, notice that the "twice" doesn't change as n goes to infinity. It's a constant! In the proof in question 3, many people seemed to confuse $\Theta(1)$ with the loop going around exactly once. Two is also $\Theta(1)$, and so is 3, or 8, or a million. The key is that it doesn't depend on n.)

So, you can get that the runtime is $O(\sqrt{n})$, and you can see that this is tight, because there are an infinite number of prime numbers, so if you tried to get a tighter big-O bound, no matter what n_0 you used for that bound, you could always find a prime $p > n_0$ that forces the runtime to be proportional to $\sqrt{p}$.

Similarly, for the lower bound, notice that there are an infinite number of even numbers. So, you can't get any tighter than constant time. (E.g., how long does this code take when n is a thousand? How about when n is a million? A billion? Try it out!) Therefore, a tight lower bound on the runtime is $\Omega(1)$.

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);
}
```

Solution : $\Theta(n)$. Each call uses a constant amount of space on the stack, and the recursion unrolls from n down to 1 before returning.

6. Is the function `triangle_numbers` tail recursive?

Solution : No. Even though the recursive call is on the last line, you still have to do the addition after the recursive call returns. (Note that the *subtraction* to get $n - 1$ happens *before* the recursive call, so that alone would not be enough to make this not 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.

Solution : The key here is to recall that you can simulate iteration with recursion. Beyond that, it's just a matter of writing the recursive helper function properly. In particular, since n is not changing during the loop, you'll need a helper function that has both n and i as a parameter.

Note that there are many ways to do this correctly. This is just one solution:

```
int count_factors_helper(int i, int n) {
  if (i>=n) return 0;
  if (n%i == 0) return 1 + count_factors_helper(i+1,n);
  else return count_factors_helper(i+1,n);
}

int count_factors(int n) {
  return count_factors_helper(1,n);
}
```

BTW, is `count_factors_helper` here tail-recursive? Can you write a version where it is?

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.

Solution : We used the same code as in Problem 2, to reduce the amount of time you'd have to spend on the exam figuring out code.

It was possible to get full credit on this problem with a very short proof, as long as it hit all the key points and was sufficiently formal. Most full credit solutions were a bit longer, giving additional details and explanation.

Common errors included not being formal enough by not using the definitions of big-Θ and big-O, mistakenly thinking that going around the loop more than once disproved $\Theta(1)$, getting the negations wrong when disproving something, trying to prove a for-all property with some numerical examples, and wrongly assuming the runtime was some well-behaved mathematical function of n (like $\sqrt{n}$ or $\log n$).

Oh yeah, another common mistake was to say that by the definition of big-Θ, the runtime is $\Theta(1)$ *if* it is $O(1)$ and $\Omega(1)$. That's a true statement, but it's not strong enough to prove that the runtime is **not** $\Theta(1)$. You actually need the "only if" part of the definition of big-Θ, not the "if" part.

Many people wrote (or attempted to write) their proofs as indirect proofs (or proofs by contradiction). This is totally OK, *as long as you do it correctly!* In particular, in a proof by contradiction, you *assume* what you are trying to prove, and then show that that leads to a contradiction. So, your proof should have started with, "This will be a proof by contradiction. Assume that the runtime is $\Theta(1)$." Your proof by contradiction should **not** start with "This will be a proof by contradiction. Prove that the runtime is $\Theta(1)$." Failing to prove something is not the same as proving the negation!

More generally, be careful with your writing. It means something very different to say "Assume X" versus "Prove X". Or, it's often very helpful in proofs to remind the reader what you are trying to do, but you need to be clear that you are *trying* to do something, versus claiming or assuming that. So, in a proof, if you want to prove that the runtime is not big-O of 1, you might say something like, "Next, we will show that the runtime is not $O(1)$." Note that use of future tense. If you had just written (without having shown it yet), "$T(n) \notin O(1)$", that would be a wrong proof. By analogy, consider difference between the statements, "Assume I have won the lottery," versus "I want to win the lottery," versus "I won the lottery."

Anyway, here's a formal proof. Your formal proofs in the class should always have a similar style! Seriously, even if you have no clue what you are doing, at least copy the style.

By the definition of big-Θ, the runtime is $\Theta(1)$ if and only if it is $O(1)$ and $\Omega(1)$. Therefore, to prove that the runtime is not $\Theta(1)$, it suffices to show that the runtime is not $O(1)$ or that it is not $\Omega(1)$. Since the runtime actually is $\Omega(1)$, we will show that it is not $O(1)$.

Let $T(n)$ denote the runtime of the function called on n. By the definition of big-O, the runtime is $O(1)$ iff there exists positive constants c and n_0, such that for all $n > n_0$, the runtime $T(n) \leq c$. Therefore, $T(n) \notin O(1)$ iff

$$\forall c > 0, n_0 > 0,\ \exists n > n_0\ T(n) > c$$

For arbitrary c and n_0, choose a value of n that is prime, with $n > n_0$, and $n > c^2$. (For full formality, this should be multiplied by a constant that relates the number of loop iterations to runtime.) This is always possible because there are an infinite number of prime numbers. Now, since the chosen n is prime, the divisibility test in the `if` statement condition will never be true, so the loop never terminates early. Therefore, the loop will run approximately $\sqrt{n}$ times, and since $n > c^2$, the loop will run greater than c times. Thus the $T(n) > c$. QED

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.

[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.

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.

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];
```

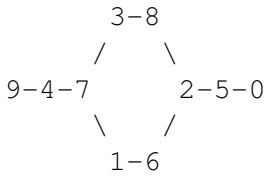
```
      }
    }
  }

  // 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.

Solution : There was a very similar problem to this on an old midterm that we provided (along with solution), where the traversals were on a tree. And this problem was discussed on Piazza. And since you did the programming project, you were already familiar with both the Wolf-Goat-Cabbage puzzle, as well as the `solvePuzzle` code. (And if not, we did provide everything you needed in the problem statement, but it would take more time to figure things out.) So, the main challenge here should only have been that it's time-consuming and detail-oriented to draw out the graph and then simulate the execution of `solvePuzzle`.

To save typesetting time, here's the figure in ASCII art:



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

   Write your answer here: 9 4 7 3 8 2 6 5 0

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

   Write your answer here: 9 4 7 1 3 6 8 2 5 0

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

   Write your answer here: 9 4 7 1 3 6 2 5 0

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!
   // SOLUTION
   // This is just one possible solution.
   if (root == NULL) return -1;
   int left_thgieh = thgieh(root->left);
   int right_thgieh = thgieh(root->right);
   return (left_thgieh < right_thgieh) ? left_thgieh+1 : right_thgieh+1;
}


}
```

2. Next, we will prove by structural induction that a tree t with $\text{thgieh}(t) = k$ 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.

Solution : Base case: Let the "thgieh" of the empty tree be k (which is -1 by definition). The empty tree has exactly 0 nodes. So, it has at least $0 = 1 - 1 = 2^0 - 1 = 2^{-1+1} - 1 = 2^{k+1} - 1$ nodes. This completes our base case.

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

   Solution : Induction hypothesis: Consider an arbitrary tree t. Assume the theorem applies to its well-formed, finite subtrees t_l and t_r. (So, t_l has a thgieh of k_l and at least $2^{k_l+1} - 1$ nodes; t_r has a thgieh of k_r and at least $2^{k_r+1} - 1$ nodes.)

   Inductive case: t has some thgieh k. By the definition of thgieh, $k = \min(k_l, k_r) + 1$. There are two cases: $k_l \leq k_r$ and $k_l > k_r$. In the latter case, note that $k_l \geq k_r$; so, the cases are symmetrical. Without loss of generality (since we could make the same argument with the subtrees reversed), assume $k_l \leq k_r$.

   Then, $k = k_l + 1$. By the IH, we know the left subtree has at least $2^{k_l+1} - 1 = 2^k - 1$ nodes and the right subtree has at least $2^{k_r+1} - 1 \geq 2^{k_l+1} - 1 = 2^k - 1$ nodes. Thus, the whole tree has at least the nodes in the left subtree, the nodes in the right subtree, and the root: $(2^k - 1) + (2^k - 1) + 1 = (2^k + 2^k) - 1 - 1 + 1 = 2^{k+1} - 1$ nodes.

   This completes our inductive case.

   Thus, t has at least $2^{k+1} - 1$ nodes.

   QED

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) { // BASE CASE: constant time
     return;
   }
   else {
     int thgiehL = thgieh(root->left); // Theta(n/2), which is Theta(n)
     int thgiehR = thgieh(root->right); // Also Theta(n)
     if (thgiehR > thgiehL) { // Swapping the children is Theta(1)
       Node* temp = root->right;
       root->right = root->left;
       root->left = temp;
     }
     convert(root->right); // Recursive call, approx. T(n/2)
     convert(root->left); // Recursive call, approx. T(n/2)
   }
}
```

   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.)

Solution : We have annotated the code above. For the recurrence relation, we just follow our annotations. To be precise, we can write this out by introducing symbolic constants, although they wouldn't matter for the asymptotic analysis. The important thing is to realize the base case is constant time, and that the recursive case does a linear amount of work, and then makes two recursive calls, on roughly half the number of nodes each.

$T(0) = C_1$ for some constant C_1.

For $n > 0$, $T(n) = 2T(n/2) + C_2 n + C_3$ for some constants C_2 and C_3

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.)

$$
\begin{aligned}
T(0) &= 3 \qquad \text{for } n \leq 1 \\
T(n) &= 3T(n/3) + n/3 \qquad \text{for } n > 1 \\
&= 3(3T(n/9) + n/9) + n/3 \\
&= 9T(n/9) + 2n/3 \\
&= 9(3T(n/27) + n/27) + 2n/3 \\
&= 27T(n/27) + 3n/3 \\
&\vdots \\
&= 3^k T(n/3^k) + kn/3 \quad \text{general pattern} \\
&= 3^{\log_3(n+1)} T(n/3^{\log_3(n+1)}) + (\log_3(n+1))n/3 \quad \text{when } k = \log_3(n+1) \\
&= (n+1)T(0) + (\log_3(n+1))n/3 \\
&= 3(n+1) + (\log_3(n+1))n/3 \\
&\in \Theta(n \log n)
\end{aligned}
$$

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

Solution : This will be $\Theta(n^2)$.

The worst case occurs if the tree is completely unbalanced, e.g., a single chain going off to the right. Then one recursive call would be $T(1)$ which would be constant time, but the other recursive call would be $T(n-1)$. And we'd still do linear time work for the two calls to `thgieh`. We've seen this recurrence several times (most

recently, with QuickSort), but you would get:

$$
\begin{aligned}
T(n) &= T(n-1) + cn \quad \text{for some constant } c \\
&= T(n-2) + c(n-1) + cn \\
&= T(n-3) + c(n-2) + c(n-1) + cn \\
&\vdots \\
&= c\sum_{i=1}^{n} i \\
&\in \Theta(n^2)
\end{aligned}
$$