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

SAMPLE SOLUTIONS

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!

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.

The basic marking scheme was 1 point for each box filled in correctly, with a few special cases for certain systematic mistakes. The downside of this scheme is that someone can easily lose several points for a bit of carelessness. On the positive side, someone who understands the concepts can derive answers easily, or cross-check his/her work (e.g., if X is $O(Y)$, then Y is $\Omega(X)$). One of my goals in giving you the first row as an example was to remind you all that a function is big-O, -Ω, and -Θ of itself.

The best way to approach a problem like this is to simplify each expression to its asymptotic order:

A. $42 \in \Theta(1)$

B. $\lg(n^2) = 2\lg n \in \Theta(\lg n)$

C. $n^{100} \in \Theta(n^{100})$

D. $2^{\lg n} = n \in \Theta(n)$

E. $2^n \in \Theta(2^n)$

F. $5n + \lg n - 22 + \frac{1}{n} \in \Theta(5n) = \Theta(n)$

Once you have these, you can immediately see that A is asymptotically smaller than B, is asymptotically smaller than D and F (which are asymptotically equal), are asymptotically smaller than C, is asymptotically smaller than E. And from that, you can just fill in the blanks:

The function...	... is big-O of:	... is big-Ω of:	... is big-Θ of:
A	A, B, C, D, E, F	A	A
B	BDFCE	AB	B
C	CE	ABDFC	C
D	DFCE	ABDF	DF
E	E	ABDFCE	E
F	DFCE	ABDF	DF

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;
}
```

Solution : $\Theta(1)$.

This was meant to be some easy points: do you know what "tight big-Θ" means? Have you looked at a C++ program?

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;
}
```

Solution : $\Theta(\lg n)$.

To do this problem, you first need to be able to read the program and understand what it does: it's going to divide n by 2 each iteration, and this will be an integer division, so it rounds down, and then it loops until n is 0. Then, you either remember that the number of times you can divide by 2 gives you the log time bound, or you can crank through some formulas: each time through the loop, you divide n by 2, so after i times through the loop, n will be equal to $n_0/2^i$, where n_0 is the original value of n. What value of i (in other words, how many iterations) will it take for $n_0/2^i < 1$ (at which point it will round down to zero)?. Multiplying both sides by 2^i and taking the log, we find that $\lg n_0 < i$.

You really want to be comfortable enough with this code to do this problem quickly, without resorting to a bunch of algebra. Also, some people were confused becaues they didn't remember that integer division in C++ (and in Java and C and most programming langauges) rounds down. I had originally written this problem with `n >>= 1;` instead of `n /= 2;`. I thought this way would be easier for people to understand.

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);
  }
}
```

Solution : $T(0) = c_1$ and $T(n) = c_2 + T(n-1)$. It was also fine if you put in specific constants like 1 instead of symbolic constants like c_1. One could also write (indeed, it'd be a more formally correct solution) something like $T(m, 0) = c_1$ and $T(m, n) = c_2 + T(m, n-1)$.

4. Is `mult2` 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.

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.

Solution :

$$\begin{aligned}
T(n) &= c_2 + T(n-1) \\
&= c_2 + c_2 + T(n-2) \\
&= c_2 + c_2 + c_2 + T(n-3) \\
&\vdots \\
&= ic_2 + T(n-i) \\
&= nc_2 + T(0) \quad \text{when } i = n \\
&= nc_2 + c_1 \\
&\in \Theta(n)
\end{aligned}$$

3 Big-O Proofs [15 marks]

For this problem, you must give formal proofs.

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

Solution : Proof: By the definition of big-O, we must find positive constants c and n_0 such that for all $n > n_0$, we have $n + 1000 < cn$. Consider $c = 2$ and $n_0 = 1000$. Then, since $n > n_0$, we know that $1000 < n$. Adding n to both sides gives us the desired result. QED

The preceding paragraph is a model solution. Obviously, many other values will work. Also, if you named your constants differently (as done in Epp, for example), that'd be completely fine. This problem was meant to be an easy assessment of the most fundamental big-O proofs.

By the way, some people who appeared to understand how to do the problem lost points for not stating clearly what their proof actually was. Remember that the goal here is NOT just to earn points on an exam by writing things that resemble a correct solution; the goal is to write a proper proof. A proof is really just a formalized argument, so you should write with the same level of explanation as you would if you were trying to convince someone of something. If I just wrote down, "Strawberries are $4.99 per pound," you'd have no idea what I'm talking about. Instead, if I said, "Summer fruits are more expensive when out of season. For example, last week at my local grocery store, Strawberries were \$4.99 per pound, whereas in the summer, they sell them for less than \$1.99 per pound," you'd understand the point I'm making. Similarly, an attempted proof that just said "$n_0 = 1000$, $c = 2$" wouldn't make any sense. What are you talking about? What does n_0 represent? What does c represent?

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

Solution : This is a harder problem, since you now must disprove an "exists" property (that there exists n_0 and c that would make $3^n \in O(2^n)$). To do this proof, you have to show that no matter what n_0 and c someone picks, they won't work. It's not good enough to just show some particular values of n_0 and c that don't work. Here's a model solution:

Proof: The proof is by contradiction (or indirect proof). Suppose that $3^n \in O(2^n)$. Then by the definition of big-O, there must exist positive constants c and n_0 such that for all $n > n_0$, $3^n < c2^n$. However, dividing both sides by 2^n, this implies that $3^n/2^n = (3/2)^n < c$ for all $n > n_0$. But $(3/2)^n$ goes to infinity as n goes to infinity, so there can't be a constant c that is always larger $(3/2)^n$ for all $n > n_0$, so this is a contradiction. Therefore, $3^n \notin O(2^n)$. QED

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:**

Solution: There are really two "ideal" solutions, because the two parts of the proof are interchangeable: L PNHCDEKGJ QOIABM F is one solution, or L QOIABM PNHCDEKGJ F is the other. However, there are other solutions that are logically just as good (although if you wrote them out, they might be harder to read). Accordingly, the marking scheme took off 1 point for each statement that was listed where the necessary statements it depends on haven't been listed yet, as well as 1 point off for each missing statement (except for L, P, and Q, which are there for the benefit of a human reader, but don't fulfill a logical role), as well as 1 point off for certain orderings that are particularly bad. For example, many people tried to put M before O, which doesn't make sense, since M is concluding the thing that O is trying to prove. If you were a lawyer trying to convince a jury (which isn't that different from a proof), would it work to say: "Therefore, my client is innocent. What I am going to show you today is that my client is innocent." A proof is supposed to be an argument that makes sense, and I'm getting the impression that many of you struggling with proofs have never really gotten to the point where any proof ever felt like an argument that made sense (probably because the examples were all math formulas, and proofs were taught as just manipulating math formulas without any intuition).

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.**

Solution : This problem was to test your understanding of stacks and queues **as abstract data types**. There's also a bit of programming and problem solving, but not too much. It might have been easier if I had given more of a hint, e.g., "I am enqueueing by pushing onto the stack, so when I dequeue, I need to get at the element that's at the **bottom** of the stack. How can I get to that element?" The answer is exactly how you would do it in real life: you'd take things off of the stack and pile them up onto a second stack, until you can get to the thing at the bottom of your stack. Then, you'd pile everything back onto the original stack. Similarly, the trick here is to push everything onto the second stack until we get to the bottom element, which we can return to the caller, then push everything back onto the stack.

Here's a solution with a second stack:

```
int IntQueue::dequeue() {
  IntStack t; // temp storage to unpack stack.
  int result, temp;

  assert (!s.isEmpty()); // dequeue undefined if stack is empty

  // Unpack stack from s onto t
  t = new IntStack();
  while (!s.isEmpty()) {
    temp = s.pop();
    t.push(temp);
  }

  result = t.pop();

  // Pack everything from t back onto S.
  while (!t.isEmpty()) {
    temp = t.pop();
    s.push(temp);
  }

  return result;
}
```

Alternatively, we could think of this problem recursively. If s has only one element, then the dequeue is easy, since the top and bottom of the stack are the same. That's the base case. Otherwise, we pop one item off the stack, and dequeue from what's left. Before we return, we need to push the item that we popped off earlier. This is basically using the system call stack in place of the temporary stack t in the preceding solution.

```
int IntQueue::dequeue() {
  int result, temp;

  assert (!s.isEmpty()); // dequeue undefined if stack is empty

  temp = s.pop();
  if (s.isEmpty()) {
    // We've reached the bottom.
    // temp contains the item we're supposed to dequeue.
    return temp;
  }

  // Recursive call of the method.
  result = dequeue();

  // Put the item we popped back on the stack.
  s.push(temp);

  return result;
}
```

(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.

   Solution : $T(-1) = c_1$ and $T(h) = T(h-1) + c_2$. For this problem, one challenge was simply to understand the code (important skill in real life!). The other key point was that this was in terms of the **height** of the tree, not the total number of nodes. So, in the worst case, the subtree we recurse into has height $h-1$. Also, the base case is for an empty tree, which is defined to have height -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.

   Solution : $\Theta(h)$. Again, this is in terms of the height, not the total number of nodes.

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:

   Solution : In the base case, the tree is empty, so it vacuously obeys the heap-order property. After the insertion, there is only a single node, so it also vacuously obeys the heap-order property.

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

   Solution : Proof: We are given a tree with the heap-order property and must prove that after the `insert` function finishes, the heap-order property still holds. (Aside: this text really should have been written at the beginning of the proof, before the base case.) In the inductive case, we are at a non-leaf node. By the heap-order property, the priority at this node is currently less than or equal to the priorities at its child(ren). The code calls `insert` recursively on one of the subtrees. (The two cases are symmetric and can be reasoned about identically. Without loss of generality, we can assume we recurse on the left subtree.) By the inductive hypothesis, the resulting subtree still obeys the heap-order property after the recursive call returns, i.e,. it's smallest element is at its root. Therefore, the only place the heap-order property might be violated is between the original node's priority and the priority at the root of the subtree. If the subtree's root's priority value is larger or equal to the node's priority value, then the heap-order property holds already, and the code does nothing. If the subtree's root's priority value is smaller than the node's priority value, the priority at the subtree's root must be the newly inserted value (because the heap order property held before the insertion). Therefore, the code exchanges these priority values, which guarantees that the heap order property holds between the node and the subtree root (because of the exchange), as well as within the subtree (because the heap order property held before the insertion). The other child of the node is not affected, since the heap order property already held there, and the exchange results in an even smaller value at the node. Hence, in all cases, the code restores the heap order property. QED

Discussion: The key here is to trust the inductive hypothesis, exactly as you would trust the recursion! Many people tried to explain their "proofs" in terms of percolate-up or percolate-down, but there's **nothing** in the code that says "percolate". If you wanted to do a proof that relies on the percolate functions we learned in lecture, you'd have to prove that the code here performs `percolate-up`. The other challenge in this proof is to reason about the various cases of how the parent's priority can compare to its children's priorities.

Discussion 2: BTW, some people might not be happy with the structural induction (on the structure of the tree) that I used above. If you insist on doing high-school style induction, you could use the exact same proof I gave, except you'd add a few sentences like, "The induction is on the height of the tree. In the base case, the tree is empty, so it has height -1.... In the inductive case, the tree has some height h, and by the inductive assumption, we assume that `insert` preserves the heap-order property on all trees of height less than h. The code calls `insert` recursively on one of the subtrees, which must have height less than h, so the inductive hypothesis holds....

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.

Solution : The trick here is first to understand what the code does, then understand how the heap order property works, which shows why the code can fail to find the largest priority value in the heap. The problem is that the code is "greedy" – it picks which child search based on the priority at the root of the subtree, which doesn't say how much bigger values might be further down the tree. For example, here's a simple solution:

```
    1
   / \
  2   3
 /     \
9       4
```

The code will compare the 2 and the 3 and recurse down the right child, where it will end up returning 4, but the 2 is "hiding" the much larger value 9.

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.)

Solution : The most important point here is to realize that the function is tail-recursive, so the conversion to iteration is easy. Then, it's just a matter of applying the conversion: making the code loop, and simulating the effect of passing the parameter in by assigning the new value. I guess one other tricky thing is making sure the control flow does the right thing: either by understanding the `continue` statement in C++ (and C and Java), or by making some of the `if` statements `else ifs`

```
// Pre-Condition:  Never called on an empty tree (NULL)
int buggyFindMax(Node * heap) {
  while (1) {
    if ((heap->left==NULL) && (heap->right==NULL)) return heap->priority;
    if (heap->left==NULL) heap = heap->right;
    else if (heap->right==NULL) heap = heap->left;
    else if (heap->left->priority > heap->right->priority)
      heap = heap->left;
    else
      heap = heap->right;
    }
}
```