MIDTERM (with SOLUTIONS)

Problem 1 (Java) [5+5 points]
Consider the following code for a class `ModernArtist`:

```
public abstract class ModernArtist {
    protected String name;
    ModernArtist(String s){
        name = s;
    }
    ...
}
public class Cubist extends ModernArtist {
    ...
}
Cubist firstOne = new Cubist("Pablo Picasso");
```

(a) What does the keyword "**abstract**" here imply about the class `ModernArtist`? In particular, how is it different from an ordinary class?
(b) In what sense do we say that you cannot create a stack (the ADT) of `ModernArtists`. Describe situations where this is an advantage, and where this is a disadvantage.

ANSWER:
(a) You are not allowed to instantiate an abstract class. You need to first extend it to some non-abstract class (like Cubist.)
(b) You cannot create a stack of `ModernArtists` since `ModernArtist` is an abstract class. But you can always create a stack of `Objects` which are Modern Artists. Using abstract class here may be a disadvantage when you really want to collect all kinds of ModernArtists into one stack. It is an advantage if the kinds of ModernArtists are so diverse that you (as code designer) really want to prevent any such collection from happening.

Problem 2 (Big-Oh Notation) [8 points]
(a) State a big-Order bound on the following two sums:

$$S_1 = \sum_{i=0}^{n-1} (1.5)^i, \qquad S_2 = \sum_{i=1}^{n} \frac{n}{i}.$$

(b) Prove that 4^n is not $O(2^n)$. HINT: You may use a proof by contradiction.

ANSWER: (a) $S_1 = O(1.5^n)$. $S_2 = O(n \log n)$.
(b) Suppose, by way of contradiction, there is some $C > 0$ and n_0 such that

$$4^n \leq C.2^n,$$

for all $n \geq n_0$. Then $2^n \leq C$ for all $n \geq n_0$. This is wrong if we choose $n = \max\{n_0, C, 2\}$ (expand the justification on this yourself).

Problem 3 (Asymptotics) [15 points]

Order the following functions in increasing order of growth. NOTE: log means $\log_2$.

(a) $n!$ (b) $n^2 \log\log n + n \log^8 n$ (c) $(1/2)^n$ (d) $n^{1/\log n}$
(e) $4^{\log n}$ (f) $n \log^{15} n$.

ANSWER: Note that you can ignore the second term in (b); the function in (c) is $o(1)$ and the function in (d) is really the constant 2. Hence:

$$(c) < (d) < (f) < (e) < (b) < (a)$$

Problem 4 (Analysis) [4+6+4+6 points]

Consider the following pseudo-code:

```
for i := 1 to n do begin
      min := array[i];
      index := i;
      for j := i+1 to n do begin
            if array[j] < min then begin
                  min := array[j];
                  index := j;
            end;
            array[j] := array[i];
            array[i] := min;
      end;
end;
```

(a) Let $T(n)$ be the worst case running time of this code. Write an upper bound expression for $T(n)$ in terms of a double summation (corresponding to the 2 for-loops). You need to get your constants correct (count 1 for each array access (`array[i]`), count 1 for each comparison in the for-loop, count 1 for each assignment, etc).
(b) Use your answer in (a) to show that $T(n) = O(n^2)$.
(c) Similarly, write a lower bound expression on $T(n)$.
(d) Use your answer in (b) to prove that $T(n) = \Omega(n^2)$.

ANSWER:
(a) $T(n) \le \sum_{i=1}^{n}(6 + \sum_{j=i+1}^{n} 13)$. Here is how I get the first constant "6": each iteration of the first for-loop needs (1) an increment of i; (2) an assignment of $i+1$ to i; (3) a test to see if this new value is more than n; (4) an array access `array[i]`; (5) an assignment to `min` and (6) an assignment to `index`. Similarly, the second for-loop has a constant of "13": each iteration has the analogues of operations (1)–(3) above; (4) an array access `array[j]`; (5) a comparison with `min`; (6-8) possibly another 3 operations inside the if-statement; and (9-13) another 5 operations for updating the array.
(b) $T(n) \le \sum_{i=1}^{n}(6 + 13(n-i)) \le 13n(n+1) = O(n^2)$.
(c) $T(n) \ge \sum_{i=1}^{n}(6+\sum_{j=i+1}^{n} 10)$. The constant "6" is as in part (a) but now the inner loop may execute only "10" operations in case the if-statement fails (this is a lower bound).
(d) $T(n) \ge \sum_{i=1}^{n}(6 + 10(n-i)) \ge \sum_{i=1}^{n/2}(6 + 10(n/2)) \ge (n/2)(11n/2)$ (for $n \ge 12$). Hence $T(n) = \Omega(n^2)$.

Problem 5 (Trees) [5+10 points]

Here is the pre-order and post-order traversal of the nodes in a binary tree T on the nodes $a, b, c, \ldots, h, i$:

$$PRE(T) = (a, b, d, e, c, f, h, i, g);$$
$$POST(T) = (d, e, b, h, i, f, g, c, a)$$

(a) Please reconstruct the tree T from these 2 traversals. HINT: how can you detect the right child of the root in *any* postorder traversal?
(b) Prove that you can always reconstruct a tree T from $PRE(T)$ and $POST(T)$. HINT: Use induction, by expressing $PRE(T)$ as $(root(T), PRE(T_L), PRE(T_R))$ where T_L, T_R are the left and right subtrees of T. Do a similar decomposition of $POST(T)$.

ANSWER:
(a) The tree is $a(b(d, e), c(f(h, i), g))$. Answer to HINT: the right child of the root is always the last but one node in $POST(T)$.
(b) Using the previous HINT, we can inductively reconstruct T: BASIS: if T has only one node, this is trivial. Otherwise, since we know the left and right children of the root, we can use this information to partition $PRE(T) = (root(T), PRE(T_L), PRE(T_R))$. Similarly, $POST(T) = (POST(T_L), POST(T_R), root(T))$. Then by induction, from $PRE(T_L), POST(T_L)$ we can reconstruct T_L. Similarly for T_R. Hence we can reconstruct T.

Problem 6 (Linked Lists) [10+5 points]

(a) Assume you have the singly-linked list structure. Write a Java method called `Reverse(Node H)` to reverse a linked list where `Node H` is a reference to the head (i.e., first node) of the list. Note: You may assume that this method is a member of the class `Node`. Recall that the class `Node` has a public method called `getNext()` that returns a reference to the next node in the list.
(b) What is the running time of your algorithm on a list of n nodes?

ANSWER:
(a) Assume `Reverse` returns `H`, which now points to the head of the reversed list. In the following, we assume H is not a **null** pointer. The algorithm keeps track of a "back" and a "forward" node (`H` is updated to be the "current node", which is between the back and forward nodes). The `next` field of the current node is always updated to the "back" node.

```
public Node Reverse(Node H) {
    Node back, forward;
    forward = H.getNext();
    while (forward != null) {
        back = H;
        H = forward;
        forward = H.getNext();
        H.setNext(back);
    }
    return(H);
} }
```

(b) Clearly, this algorithm is $O(n)$.