CMPT 307
Fall 2007
T. Shermer

Midterm

100 points. 75 minutes. No calculators, books, or notes allowed.

1. (15 points; 3 each)
   a. Define $\Omega(g(n))$.
   b. Express $6n^2 + 4n^2 \log n + 2n^{3/2} + 35$ in O-notation.
   c. Express $4^{\log n} + 5n^{1.63} + 3^n$ in Ω-notation.

2. (15 points) The following is the routine PLOOVIN, a critical component of nothing in particular. Derive a recurrence for the time complexity of PLOOVIN, and express that time complexity in Θ-notation. Do not worry about floors and ceilings.

```
int PLOOVIN(A, left, right)
     if (right - left < 4) {
          return 1;
     }
     int sum = 0;
     for(int i=left; i<right; i++) {
          for(int j = i+1; j<right; j++) {
               sum = sum + A[i] * A[j];
          }
          sum = sum + A[i];
     }

     int mid =  (left + right) / 2;
     int qone = (left + mid) / 2;
     int qtwo = (mid + right) / 2;

     plooA = PLOOVIN(A, left, mid);
     plooB = PLOOVIN(A, qone, qtwo);
     plooC = PLOOVIN(A, mid, right);

     int min = plooA;
     if(plooB < min) {
          min = plooB;
     }
     if(plooC < min) {
          min = plooC;
     }
     return sum - min;
```

3. Express the following recurrences in O-notation.
    a. (5 points) $T(n) = 9T(n/3) + cn^2$
    b. (10 points) $T(n) = (\log n)T(n/2) + c(\log n)$

4. (20 points; 10 each) A max-heap can be used as a *max-priority queue*, which is an abstract data type having the operations *Insert*, *Maximum*, *Extract-Max*, and *Increase-Key*.

    a. Describe how to implement a FIFO queue with a priority queue, and analyze the time complexity of the implementation (*Insert* and *Delete* operations), assuming that a heap is used for the priority queue.
    b. Describe how to implement a LIFO queue (i.e. a stack) with a priority queue, and analyze the time complexity of the implementation (*Insert (Push)* and *Delete (Pop)* operations), assuming that a heap is used for the priority queue.

5. (10 points) What is the reason that we choose a random pivot element in *Randomized Quicksort*? How can using randomization in this way possibly help?

6. (25 points) On planet Triblinki, everyone has three fingers on each of their three hands, one of which is at the end of each of their three arms. They watch over their three hands with their three eyes. Naturally, when Triblinkians designed computers and programs, the comparisons they used were three-way.

We can think of the Triblinkian comparison as a function `COMPARE` that takes three arguments and returns one of eight codes that indicate which permutation of the arguments is increasing, as follows:

```
switch(COMPARE(a,b,c)) {
case abc:                // a <= b <= c

case acb:                // a <= c <= b

case bac:                // b <= a <= c

     ...

case cba:                // c <= b <= a

default:                 // error condition
}
```

(On Triblinki, of course, the basic branching statements of their computer languages are nine-way, with the different branches named after their nine different fingers; the switch and cas es were just to help make it understandable to you.)

Noz Guzbop is a professional banjo player on Triblinki, and for reasons known only to him, he needs an algorithm that, given an array of *n* numbers, returns the smallest, the second smallest, and the largest of these numbers. His computer has a very slow implementation of `COMPARE`, but unfortunately that's the only way to compare elements that he has available.

Help poor Noz out: design an algorithm for his problem, using as few calls to `COMPARE` as possible (the fewer calls, the better your mark). Analyze the number of calls your algorithm makes to `COMPARE`. Code is neither necessary nor desired in your response: describe the algorithm at a high level.

Your algorithm must be based on `COMPARE` (no radix or counting sorting, for instance), but, at no cost, you may use pointers or lists or any data structures that don't use comparisons.