STUDENT NAME: _____

STUDENT ID: _____

FIRST MIDTERM

COMP-250: Introduction to Computer Science - Winter 2008

March 10, 2008

You are allowed one double sided cheat sheet.

There are 4 questions, for a total of 100 points. Please read all the questions first. Please make sure to **write your name** and ID number on the exam booklet!

Answer all questions on the exam booklet

Good luck!

1. [20 points] **Big-oh**

   For each of the questions below, provide a true or false answer and explain your reason.

   (a) $1000000n + 10^{10} \in O(n)$
   True. Apply the definition of $O(n)$ with $c = 1000001$ and $n_0 = 10^{10}$.

   (b) $2^{n+1} \in O(2^n)$?
   True. Note that $2^{n+1} = 2 \cdot 2^n$ and apply the definition of $O()$ with $c = 2$, $n_0 = 0$.

   (c) $2^{2n} \in O(2^n)$?
   False because $\lim_{n\to\infty} \frac{2^{2n}}{2^n} = \lim_{n\to\infty} 2^n = \infty$.

   (d) $2^{\log_{10}(n)} \in O(n)$?
   Re-write $\log_{10}(n) = \log_2(n) \cdot \log_1 0(2)$, so $2^{\log_{10}(n)} = 2^{\log_2(n)\cdot\log_1 0(2)} = n^{\log_1 0(2)} < n$ (because $2 < 10$, so $\log_1 02 < 1$. Hence we can use the definition of $O()$ with $c = 1$, $n_0 = 1$.

2. [30 points] **More Big-Oh**

   For each of the pieces of pseudocode below, state what $O()$ is.

   (a) **Algorithm** f1(n)
   $i \leftarrow 1$
   while $i < n$
   print(i)
   $i \leftarrow i + 10$
   $O(n)$ (loop with incrementing counter)

   (b) **Algorithm** f2(n)
   $i \leftarrow 1$
   while $i < n$
   print(i)
   $i \leftarrow i * 10$
   $O(\log n)$ (loop where the variable gets multiplied)

   (c) **Algorithm** f3(n)
   $i \leftarrow 1$
   while $i < n$
   print(i)
   $i \leftarrow i * 10 + 37$
   $O(\log n)$ (same as above, the multiplication is the part that matters)

(d) **Algorithm** f4(n)
$i \leftarrow n$
while $i! = 0$
  print(i)
  $i \leftarrow i$ **mod** 10

The **mod** operator takes the remainder of the integer division of i to 10, so it returns a result between 0 and 9. If the result is 0, the loop terminates right away, so we have $O(1)$. If the result is not 0, subsequent operations will not change the value of i, so the program will loop forever.

(e) **Algorithm** f5(n)
if $n = 0$ **return**
print(n)
f5($n-1$)

$O(n)$ (you can convert the recursion into a loop with decrementing index).

(f) **Algorithm** f6(n)
if $n = 0$ **return**
print(n)
f6($n/10$)

$O(\log n)$ (you can convert the recursion in a loop with an index that gets divided by 10 every time).

(g) **Bonus 5 points**
Algorithm f7(n)
if $n = 0$ **or** $n = 1$ **return**
print(n)
f7($n-1$)
f7($n-2$)

This algorithm is essentially the same as generating the Fibonacci numbers recursively. If you unfold the recursive calls, you get a binary tree of height n, which means that the complexity is exponential: $O(2^n)$.

3. [30 points] **Pseudocode**

   Write the pseudocode for an algorithm which receives as input an array of positive integers a and a positive integer x. If there are two integers p and q in the array such that $2p+q=x$, the algorithm should return p and q. Otherwise it should return $(-1,-1)$. *Your algorithm should work in* $O(n \log n)$. Hint: you can call as subroutines any of the algorithms we discussed in class.

   Example: CrazyFind($\{2,5,1,7\}$, 9) should return (1, 7)
   Example: CrazyFind($\{2,5,1,7\}$, 100) should return (-1, -1)

   Algorithm CrazyFind (a,n, x)
   Input: a is an array of positive integers of size n and x is a positive integer
   Output: A pair (p,q) of numbers from array a such that $2p+q=x$, if such a pair exists; $(-1,-1)$ otherwise

   Sort(a)
   $i \leftarrow 1$
   while $i \leq n$
   if BinarySearch(a, $x-2 \cdot a[i]$) **then return** ($a[i]$, $x-2 \cdot a[i]$)
   return (-1, -1)

   Sorting runs in $O(n \log n)$ and the loop executes n times, calling binary search, which is an $O(\log n)$ algorithm, so the whole algorithm is $O(n \log n)$ (using the rules we studied about looping, subroutine calls and sequences of instructions).

4. [20 points] **Crazy sort**

   Consider the sorting algorithm described by the following pseudocode:

   Algorithm CrazySort (a,i,j)
   Input: An array of integers a and indices i and j in the array
   Output: The array a will be sorted

   if $a[i] > a[j]$ **then** swap($a[i]$,$a[j]$)
   if $i+1 \geq j$ **then return**
   $k \leftarrow \lfloor \frac{j-i+1}{3} \rfloor$
   CrazySort(a,i,$j-k$) //recursive call on the first two-thirds of the array
   CrazySort(a,$i+k$,j) //recursive call on the last two-thirds of the array
   CrazySort(a,i,$j-k$) // recursive call again on the first two-thirds of the array

   The algorithm is called with: CrazySort(a,1,n)
   where n is the length of the array.

   (a) [10 points] Prove by induction that the algorithm is correct.

   Base case: If there is just one element the array is already sorted and the algorithm exist right away. If there are two elements, the if-swap statement will ensure that they are put in the right order. Hence, an array of size 2 will also be sorted correctly.

   Induction step: Suppose that the recursive call works and arrays up to size n are sorted correctly. Now consider an array of size $n+1$, $n \geq 2$. After the first recursive call, we have $a[i] \leq a[j], \forall 1 \leq i < j \leq \frac{2}{3}n$ (by induction hypothesis). This means, more specifically, that for any $1 \leq i \leq \frac{1}{3}n$ and any $\frac{1}{3}n \leq j \leq \frac{2}{3}n$, we ahve $a[i] \leq a[j]$. Now the elements in the middle and last thirds will get sorted. After the second recursive call, we have $a[i] \leq a[j], \forall \frac{1}{3}n \leq i < j \leq n$. Now we know the following important things:

   - In the last third of the array, the elements are sorted correctly: $a[i] \leq a[j] \forall \frac{2}{3}n \leq i < j \leq n$
   - The elements in the last part of the array are all bigger than the elements in the first part: $a[i] \leq a[j] \forall 1 \leq i \leq \frac{1}{3}n$ and $\forall \frac{2}{3}n \leq j \leq n$. This is because after our first call, we had put the "bigger" elements in the middle. So either all these ended up in the last third after the second recursive call (in which case we have the relationship already), or even bigger elements got in the last third (in which case this relationship is still true).
   - The elements in the last part are bigger than the elements in the second part: $a[i] \leq a[j] \forall \frac{1}{3} \leq i \leq \frac{2}{3}n$ and $\forall \frac{2}{3}n \leq j \leq n$ (because this call worked).

   So after the first two calls, the last part of the array is correct. The last call will make sure the first two parts are correct. This concludes our proof.

(b) [5 points] Write down a recurrence for the running time of the algorithm, $T(n)$. You may use a constant, C, to cover for all the $O(1)$ operations.
We have $T(1) = T(2) = C$. For $n > 2$, we have:

$$\begin{aligned} T(n) &= C + 3T\left(\frac{2}{3}n\right) = C + 3\left[C + 3T\left(\left(\frac{2}{3}\right)^2 n\right)\right] \\ &= C(1+3) + 3^2 T\left(\left(\frac{2}{3}\right)^2 n\right) = C(1+3+3^2) + 3^3 T\left(\left(\frac{2}{3}\right)^3 n\right) = \ldots \\ &= C(1+3+\cdots+3^{m-1}) + 3^m T\left(\left(\frac{2}{3}\right)^m n\right) \end{aligned}$$

(c) [5 points] What is $O()$ for the algorithm? Hint: to justify this, you may need to use the fact that, for a constant k, $1 + k + k^2 + \cdots + k^m = \frac{k^{m+1}-1}{k-1}$.

First, note that the argument of $T()$ becomes 1 when we have: $m = \frac{\log_3 n}{\log_3(3/2)}$. Since $T(1) = C$, we have:

$$T(n) = C(1+3+3^2+\cdots+3^{m-1}+3^m) = C\frac{3^{m+1}-1}{3-1} = \frac{C}{2}(3 \cdot 3^{\frac{\log_3 n}{\log_3(3/2)}} - 1) = \frac{C}{2}(3n^{\frac{1}{\log_3(3/2)}} - 1)$$

Hence, $T(n) \in O(n^{\frac{1}{\log_3(3/2)}})$.