[illegible]

CS 161 Summer 2009
Homework #1 Sample Solutions

Regrade Policy: If you believe an error has been made in the grading of your homework, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your homework a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire homework, so your final grade may either increase or decrease.

Problem 1 [5 points]

For arrays of types like byte, char, double, float, int, long, and short, java.util.Arrays uses a quicksort algorithm. For arrays of the general Object type, it uses a mergesort algorithm. The source of this information was

http://java.sun.com/j2se/1.4.2/docs/api/java/util/Arrays.html.

Problem 2 [10 points]

A_1: $\Theta(n^3)$. You can see this by noticing that when n doubles, the data points grow by a factor of 8 (which is 2^3), meaning that when x grows by a factor of n, $f(x)$ grows by a factor of n^3, meaning this is $\Theta(n^3)$.

A_2: $\Theta(1)$. Notice that no matter how x grows, $f(x)$ does not vary significantly, meaning that the function does not depend on the growth rate of x, so it must be $\Theta(1)$.

A_3: $\Theta(n)$. Notice that when n doubles, the data points also double, meaning that $f(x)$ grows as the same rate as x grows, so it must be $\Theta(n)$.

A_4: $\Theta(\log(n))$. If you plot this on a $\log - \log$ scale, you get a straight line, meaning this is $\Theta(\log(n))$.

A_5: $\Theta(n^2)$. You can see this by noticing that when n doubles, the data points grow by a factor of 4 (which is 2^2), meaning that when x grows by a factor of n, $f(x)$ grows by a factor of n^2, meaning this is $\Theta(n^2)$.

A_6: $\Theta(n \log n)$. When n doubles, the ratio $\frac{2n\log(2n)}{n\log n}$ is between 2.2 and 2.14, gradually declining when $1000 \le n \le 16000$. That is close to the real data points.

Alternative answer: $\Theta(n^{1.15})$. Notice that when n doubles, the data points grow by a factor of approximately $2^{1.15}$, meaning that when x grows by a factor of n, $f(x)$ grows by a factor of $n^{1.15}$, meaning this is $\Theta(n^{1.15})$. You can figure out the factor of 1.15 by looking at pairs of datapoints, and solving the equation factor $= \frac{log(y_2/y_1)}{log(x_2/x_1)}$, where (x_1, y_1) and (x_2, y_2) are two datapoints. Solving this for each consecutive pairs of points in the data gives factors of 1.13, 1.18, 1.11, 1.18, and 1.12, which are all around 1.15.

Problem 3 [10 points]

$(7, 10^6), \log\log n, 3^{\sqrt{\log n}}, (n, 5n+20), \frac{1}{2}n\log(n+5), (3n)^2, n^5, n^{\log n}, (\frac{3}{2})^n, 3^n, n3^n, n!, n^n, 3^{3^n}$
Most of the ranking is straightforward.
Asymptotic bounds for Stirling's formula are helpful. $n! = \Theta(n^{n+1/2}e^{-n})$

Problem 4 [15 points] Big O

(a) (5 points) $15n^3\log n + 10n^2 + 50 < 15n^3\log n + 10n^3\log n = 25n^3\log n$ when $n >= 4$. Thus by definition, $15n^3\log n + 10n^2 + 50 = O(n^3\log n)$.

(b) (10 points)
Upper bound, since $n! < n^n$, thus $\log(n!) < \log(n^n) = n\log n$.
$\log(n!) = O(n\log n)$.
Lower bound, since $(n!)^2 = (n)^2(n-1)^2 \ldots 2^2 1^2 >= (n/2)^n$ (just take the first n items), thus $(n!) >= (n/2)^{(n/2)}$, $\log(n!) >= n/2\log(n/2) = n/2\log n - n/2\log 2$, $\log(n!) = w(n\log n)$.
Therefore, $\log(n!) = \Theta(n\log n)$.

Problem 5 [40 points]

(a) (5 points) The pseudocode is shown in Algorithm 1.

Worst-case running time: $O(n\log n + m\log m + n + m)$, which equals $O(n\log n + m\log m)$.

(b) (10 points) The pseudocode is shown in Algorithm 2.

Worst-case running time: $O(m + n + l)$, where $l = length(A), m = length(B), n = length(C)$.

(c) (5 points) The pseudocode is shown in Algorithm 3.

Worst-case running time: $O(n\log_3 n)$, where $n = length(A)$.

Algorithm 1 INTERSECTION(A,B)

```
1:  MERGE-SORT(A)
2:  MERGE-SORT(B)
3:  Create a new output array, named C
4:  Let $a \leftarrow 1$; $b \leftarrow 1$
5:  while ($a <= length(A)$ && $b <= length(B)$) do
6:    if ($A[a] < B[b]$) then
7:      $a++$
8:    else if ($A[a] > B[b]$) then
9:      $b++$
10:   else
11:     Append $A[a]$ to the end of C
12:     $a++; b++$
13:   end if
14: end while
15: return C
```

Algorithm 2 MERGE3(A,B,C)

```
1:  Let $a \leftarrow 1$; $b \leftarrow 1$; $c \leftarrow 1$
2:  Extend lists A, B and C by 1 element each, and set those new elements to infinity
3:  Create a new output array, named D of size $lengh(A) + lengh(B) + lengh(C)$
4:  while ($d <= length(D)$) do
5:    if ($A[a] < B[b]$) then
6:      if ($A[a] < C[c]$) then
7:        $D[d] \leftarrow A[a]$
8:        $a++; d++$
9:      else
10:       $D[d] \leftarrow C[c]$
11:       $c++; d++$
12:     end if
13:   else
14:     if ($B[b] < C[c]$) then
15:       $D[d] \leftarrow B[b]$
16:       $b++; d++$
17:     else
18:       $D[d] \leftarrow C[c]$
19:       $c++; d++$
20:     end if
21:   end if
22: end while
23: return D
```

Algorithm 3 MERGE3-SORT(A)

```
if $length(A) <= 1$ then
    return A
end if
Let $a \leftarrow \lfloor \frac{1}{3} length(A) \rfloor$; $b \leftarrow \lfloor \frac{2}{3} length(A) \rfloor$
Create 3 new arrays B,C,D
B$\leftarrow A[1 \dots a]$; C$\leftarrow A[a+1 \dots b]$;D$\leftarrow A[b+1 \dots length(A)]$
//comment: a may be 0, in which case just let B be a zero length array.
MERGE-SORT(B)
MERGE-SORT(C)
MERGE-SORT(D)
return MERGE3(B,C,D)
```

Algorithm 4 FIND-IDENTITY(A[1, ..., n])

```
if $n == 0$ then
    return null
end if
if $n == 1$ then
    if $A[1] == 1$ then
        return 1
    else
        return null
    end if
end if
if $A[\lfloor n/2 \rfloor] < \lfloor n/2 \rfloor$ then
    return FIND-IDENTITY($A_1[1, \dots, \lfloor n/2 \rfloor]$)
else if $A[\lfloor n/2 \rfloor] > \lfloor n/2 \rfloor$ then
    return FIND-IDENTITY($A_2[(\lfloor n/2 \rfloor + 1), \dots, n]$)
else
    return $\lfloor n/2 \rfloor$
end if
```

(d) (10 points) The pseudocode is shown in Algorithm 4. Worst-case running time: $O(\log n)$, where $n = length(A)$.

(e) (10 points) The pseudocode is shown in Algorithm 5.

Algorithm 5 FIND-MAJORITY(A)

```
1: if $length(A) = 0$ then
2:   return $null$
3: end if
4: if $length(A) = 1$ then
5:   return 1
6: end if
7: Call FIND-MAJORITY recursively on the first half of A, and let $i$ be the result.
8: Call FIND-MAJORITY recursively on the second half of A, and let $j$ be the result.
9: if $i > 0$ then
10:   compare object $i$ to all objects in A(including itself), and let $k$ be the number of
      times that equality holds;
11:   if $k > length(A)/2$ then
12:     return $i$.
13:   end if
14: end if
15: if $j > 0$ then
16:   compare object $j$ to all objects in A(including itself), and let $k$ be the number of
      times that equality holds;
17:   if $k > length(A)/2$ then
18:     return $j$
19:   end if
20: end if
21: return $null$.
```

Worst-case running time:$O(n \log n)$.
Let $T(n)$ denote the largest number of equality tests required by FIND-MAJORITY on an array of n objects. If $n = 1$ then no equality tests are performed. If $n \geq 2$, then steps 2 and 3 require at most $T(n/2)$ equality tests each, which steps 4 and 5 each use n equality tests. Thus, $T(1) = 0$ and $T(n) \leq 2T(n/2) + 2n$ for all $n \geq 2$ that are powers of 2. Using the Master Theorem, it follows that $T(n) = O(n \log n)$.

Remarks. One common error is to run original MERGE-SORT algorithm first, and then count for consecutive and equal elements. This approach does not meet the requirements, since natural ordering $<$ or $>$ operations are needed in the original MERGE-SORT algorithm.

Extra Credit

We will give an algorithm to solve this problem which uses an auxiliary stack. The following is the pseudocode for the algorithm.

Algorithm 6 FIND-MAJORITY-ELEMENT(A)

```
for $i = 1$ to $length[A]$ do
    if stack is empty then
        push $A[i]$ on the stack
    else if $A[i]$ equals top element on the stack then
        push $A[i]$ on the stack
    else
        pop the stack
    end if
end for
if stack is empty then
    return null
end if
candidate $\leftarrow$ top element on the stack
counter $\leftarrow 0$
for $i = 1$ to $length[A]$ do
    if $A[i]$ equals candidate then
        counter $\leftarrow$ counter $+ 1$
    end if
end for
if counter $> \lfloor length[A]/2 \rfloor$ then
    return candidate
else
    return null
end if
```

The procedure first generates a candidate object from the array. It finds this element using the stack by looping over the array. If the stack is empty or the current element is the same as the top object on the stack, the element is pushed onto the stack. If the top object of the stack is different from the current element, we pop that object off the stack.

Claim: If a majority element exists, it will be on the stack at the end of this part of the procedure.

Proof: (by contradiction) Call the majority element x and say it appears $i > \lfloor n/2 \rfloor$ times. Each time x is encountered, it is either pushed on the stack or an element (different from

x) is popped off the stack. There are $n - i < i$ elements different from x. Assume x is not on the stack at the end of the procedure. Then, each of the i elements of x must have either popped another element off the stack, or been popped off by another element. However, there are only $n - i < i$ other elements, so this is a contradiction. Therefore, x must be on the stack at the end of the procedure.
Notice that this proof does not show that if an element is on the stack at the end of the procedure that it is the majority element. We can construct simple counterexamples ($A = [1, 2, 3]$) where this is not the case. Therefore, after obtaining our candidate majority element solution, we check whether it is indeed the majority element (lines 9-16). We do this by scanning through the array and counting the number of times the element *candidate* appears. We return *candidate* if it appears more than $\lfloor n/2 \rfloor$ times, else we report that no majority element exists. This procedure scans through the array twice and does a constant amount of computation (pushing and popping elements on the stack take constant time) at each step. Therefore the running time is $T(n) = cn = \Theta(n)$.

Alternative Solution: Divide and Conquer. Randomly pair up the elements. For each pair, if they match, keep one of them. If they don't, discard both of them. Continue this process until one or no element left.

Remarks: Some students may use hashmap/hashtabel to finish the task. Partial credits will be given for this approach. Note that *Lookup()* function in hashmap takes $O(1)$-time on average. But the worst-case lookup time can be as bad as $O(n)$. The worst-case expected running time of the algorithm would be linear if we view it as a randomized algorithm. But for this problem we are looking for a deterministic algorithm.