CS 102: INTRO. TO THE ANALYSIS OF ALGORITHMS
ASSIGNMENT 3
SOLUTIONS

KRISHNA M. ROSKIN

Problem 1 (5 pt)**.** *Write your name, your Group, and your email address and your favorite color.*

Krishna Roskin, D, krish at soe.ucsc.edu, green.

Problem 2 (25pt)**.** *Assume you have an array $A[1..n]$ of n elements. A* majority element *of A is any element occurring in more than $n/2$ positions (so if $n = 6$ or $n = 7$, any majority element will occur in at least 4 positions). Assume that elements* cannot *be ordered or sorted, but can be compared for equality. (You might think of the elements as chips, and there is a tester that can be used to determine whether or not two chips are identical.)*

Design an efficient divide and conquer algorithm to find a majority element in A (or determine that no majority element exists).

Aim for an algorithm that does $O(n \lg n)$ equality comparisons between the elements. A more difficult $O(n)$ algorithm is possible, but may be difficult to find.

A $\Theta(n \log n)$ running time divide and conquer algorithm:

The algorithm begins by splitting the array in half repeatedly and calling itself on each half. This is similar to what is done in the merge sort algorithm. When we get down to single elements, that single element is returned as the majority of its (1-element) array. At every other level, it will get return values from its two recursive calls.

The key to this algorithm is the fact that if there is a majority element in the combined array, then that element must be the majority element in either the left half of the array, or in the right half of the array. There are 4 scenarios.

1. Both return "no majority." Then neither half of the array has a majority element, and the combined array cannot have a majority element. Therefore, the call returns "no majority."
2. The right side is a majority, and the left isn't. The only possible majority for this level is with the value that formed a majority on the right half, therefore, just compare every element in the combined array and count the number of elements that are equal to this value. If it is a majority element then return that element, else return "no majority."
3. Same as above, but with the left returning a majority, and the right returning "no majority."
4. Both sub-calls return a majority element. Count the number of elements equal to both of the candidates for majority element. If either is a majority element in the combined array, then return it. Otherwise, return "no majority."

The top level simply returns either a majority element or that no majority element exists in the same way.

To analyze the running time, we can first see that at each level, two calls are made recursively with each call having half the size of the original array. For the non-recursive costs, we can see that at each level, we have to compare each number at most twice (which only happens in the last case described above). Therefore, the non-recursive cost is at most $2n$ comparisons when the procedure is called with an array of size n. This lets us upper bound the number of comparisons done by $T(n)$ defined by the recurrence $T(1) = 0$ and

$$T(n) = 2T(n/2) + 2n.$$

We can then determine that $T(n) \in \Theta(n \log n)$ as desired using **Case 2** of the Master Theorem.

A $\Theta(n)$ running time algorithm:

We simply sketch the algorithm and analysis here, these are not complete proofs. The following algorithm is based on two ideas. First, if array A has a majority element, and elements $A[i]$ and $A[j]$

are different, then the new array created by discarding both $A[i]$ and $A[j]$ has the same majority element as A. Second, if A has a majority element and every element in A occurs in pairs, then after deleting one of each pair and keeping the other, we get a new array with the same majority element. The only tricky part is that if A has an unpaired element, then we must keep it, but weight it less than the other elements when determining the majority element.

```
MajGuess(A,n,part) -- returns the majority element of A if one exists
                   -- could return anything if no majority element exists
                   -- parameter ‘‘part’’ is a boolean which is true
                   -- if A[n] has less ‘‘weight’’ than the other elements.

if n=0 then return null;  -- there is no majority element
if n=1 then return A[1];  -- it is the majority element

i := 1  -- place in A[]
j := 1  -- first empty position in new array

while i<n-1 do
    if A[i]=A[i+1] then
    B[j] = A[i];   -- find matching pairs, through out unmatched pairs
    j := j+1;
    i := i+2;            -- move to next pair

-- now i is either n or n-1

if i = n then
    B[j] := A[n];
    part := true;  -- the last element counts for less than the others
else          -- i = n-1
    if part then
        B[j] = A[n-1]
    else      -- not part
        if A[n-1] = A[n] then
        B[j] = A[n-1];
    else
        j = j-1;    -- A[n-1] different from A[n] and they cancel

return MajGuess (B, j, part);
```

Now to determine if array A of n elements has a majority element, we call MajGuess(A, n,false), and get an element (or null). We then check this answer by counting how many times it occurs in the actual array $A[1..n]$. The total number of tests done by MajGuess is bounded by $T(n)$ having the recurrence: $T(0) = T(1) = 0$ and

$$T(n) = T(\lfloor n/2 \rfloor) + n/2.$$

The master theorem indicates that $T(n) \in \Theta(n)$. Since only an additional n comparisons are done at the end, the entire number of comparisons done by this method is in $\Theta(n)$.

Problem 3 (25 pts)**.** *Consider the problem of multiplying two large n-bit integers in a machine where the word size is one bit.*

1. (5 pts) Consider the problem of multiplying two large n-bit integers in a machine where the word size if one bit. Describe the straightforward algorithm that takes n^2 bit-multiplications.

   This algorithm is the algorithm that we all learned in grade school to multiply two numbers.

   ```
   procedure:  multiply(A,B)
     // assume A,B each have n bits
     // the lower order bit of A will be written A[0],
     // the highest will be A[n], same for B
     let result = 0
     let temp = 0
   ```

```
for j = 1 to n do
  for i = 1 to n do
    temp[i] = B[i] * A[j]
  end for
  shift temp by (j - 1) bits to the left
  set result to result plus temp
  set temp = 0
end for
return result
```

The two `for` loops that surround the bit multiplication makes this algorithm do n^2 bit-multiplications.

2. (5 pts) Find a way to compute the product of the two numbers using three multiplications of $n/2$ bit numbers (you will also have to do some shifts, additions, and subtractions, and ignore the possibility of carries increasing the length of intermediate results). Describe your answer as a divide and conquer algorithm.

   The algorithm that follows is motivated by the following example where two 2-digit (base 10) numbers are multiplied:

```
          a  b
        * c  d
        ------
         ad bd
     ca  cb
    ----------
ca (ad+bc) bd  (assuming no carries)
```

   In other words:

$$(a \times 10^1 + b \times 10^0) \times (c \times 10^1 + d \times 10^0) = ca \times 10^2 + (ad + bc) \times 10^1 + bd \times 10^0$$

   Converting now to bits, if a, b, c, d are all $n/2$-bit numbers then multiplying two n-but numbers (ab) and (cd) takes 4 multiplications of size $n/2$, 3 shifts and 3 addition/subtraction operations. We can reduce the number of multiplications by noticing that:

$$ad + bc \quad = \quad (a + b)(c + d) - ac - bd$$

   and therefore we have all the numbers we need with only three multiplications, and we can write down the divide and conquer algorithm (for binary numbers):

```
procedure:  DC_multiply(X,Y)
  // assume X,Y each have n bits, where n is a power of 2
  if X and Y are both 1 bit long, return X*Y;
  else
    let k = n/2;
    let a = X div 2^k and b= X mod 2^k (so a*2^k + b = X)
    let c = Y div 2^k and d= Y mod 2^k (so c*2^k + d = Y)
    // a,b,c,d are all n/2-bit numbers
    temp_1 = DC_multiply(c,a)
    temp_2 = DC_multiply(a+b,c+d)
    temp_3 = DC_multiply(b,d)
    return ( temp_1*2^(2k) + (temp_2 - temp_1 - temp_3)*2^k + temp_3 )
```

   In this algorithm we have 3 recursive calls on $n/2$-bit numbers, 2 shifts and 6 additions/subtractions on $\Theta(n)$-bit numbers (not counting the shifts/subtractions needed for the div and mod operations).

3. (5 pts) Assuming that adding/subtracting numbers takes time proportional to the number of bits involved, and shifting takes constant time.

   Using the above algorithm we can immediately write down the following recurrence for its running time. $T(1) = 1$ and

$$T(n) = 3T(\lceil n/2 \rceil) + G(n) \quad \text{for } n > 1$$

where $G(n) \in \Theta(n)$ is the time needed for the additions, shifts, and overhead. Applying the master theorem we get **Case 1**: $E = \lg 3 / \lg 2 = \lg 3$, so $T(n) \in \Theta(n^{\lg 3})$ (recall that $\lg 3 \approx 1.585$).

4. (10 pts) Now assume that we can find the product of two n-bit numbers using some number of multiplications of $n/3$-bit numbers (plus some additions, subtractions, and shifts). What is the largest number of $n/3$ bit number multiplications that leads to an asymptotically faster algorithms than the $n/2$ divide and conquer algorithm above?

   The running time of a 5-multiplication method for two n-figure numbers has the recurrence relation

   $$T(n) = 5T(n/3) + G(n)$$

   where $5T(n/3)$ is the time to do the 5 recursive multiplications, and $G(n) \in \Theta(n)$ is the time needed for the additions, shifts, and overhead. Applying case 1 of the Master Theorem, we get $T(n) \in \Theta(n^{\log_3 5})$, and $\log_3 5 \approx 1.465$, so this is better than the 2-multiplication method.

   A 6-multiplication method would have the recurrence relation

   $$T(n) = 6T(n/3) + G(n),$$

   where $6T(n/3)$ is the time to do the six recursive multiplications and $G(n) \in \Theta(n)$ is the time needed for the additions, shifts, and overhead. Using the the Master Theorem on this recurrence yields $T(n) \in \Theta(n^{\log_3 6})$, and $\log_3 6 = 1.631$, which is asymptotically worse than the algorithm we developed for our 2-multiplication method.

   Therefore 5 is the largest number of $n/3$ bit number multiplications that leads to an asymptotically faster algorithms than the $n/2$ divide and conquer algorithm above.

Problem 4 (20 pts)**.** *Consider the following variation on* Mergesort *(pg. 175 of the text) for large values of n. Instead of recursing until n is sufficiently small, recur at most a constant r times, and then use insertion sort to solve the 2^r resulting subproblems. What is the (asymptotic) running time of this variation as a function of n?*

Each call to Mergesort on n elements makes 2 recursive calls of size $n/2$, if we recurse at most r times and then revert to the basic (ad-hoc) method, the modified Mergesort creates 2^r subproblems of size $\frac{n}{2^r}$. If the ad-hoc algorithm runs in time $\Theta(n^2)$ (like selection or insertion sort), this modified mergesort will take at least $2^r(\frac{n}{2^r})^2 = n^2/2^r$ time for large instances ($n > 2^r n_0$).

Since r is constant, so is 2^r, and the running time is then $\Omega(n^2)$, which is asymptotically worse than the standard mergesort. In essence, this modification requires that we solve $\Theta(n)$ size problems with the ad-hoc method. If the ad-hoc method is polynomial, then the modification to divide-and-conquer method cannot be more than a constant factor better than simply applying the ad-hoc method to the original problem.

Problem 5 (25 pts)**.** *Consider a modified median of medians selection algorithm where the n elements are grouped into $n/3$ groups of three (rather than $n/5$ groups of five).*

1. First (5 pts), What is the (largest) numbers of elements examined by each of the two recursive calls? (Round your answers to simple fractions of n like $n/4$).

   The first recursive call in this version of the Selection Divide and Conquer algorithm is to find the *Median Of Medians*. This recursive call finds the median of $\lceil n/3 \rceil$ numbers, the medians of $\lceil n/3 \rceil$ groups of three (with the last group possibly being smaller). Therefore, the first recursive call is on roughly $n/3$ elements.

   The second recursive call is on one of the partitions after using the *Median Of Medians* as the pivot. We can get a lower bound on the number of elements in the smaller partition in the following way. Without loss of generality we assume that the smaller partition has elements greater than the *Median Of Medians* (the other case is similar). There are $\lceil n/3 \rceil (1/2)$ medians greater than or equal to the *Median Of Medians*. For each of these medians, at least 2 of the 3 elements in the group are greater than or equal to the *Median Of Medians* (except for the last group which could have only one greater element). Therefore there are at least $\lceil n/3 \rceil (1/2) * 2 - 1 = \lceil n/3 \rceil - 1$ in the (smaller) partition with elements greater than or equal to the *Median Of Medians* (this also counts the Median of Medians itself). The (-1) comes from the (possibly) incomplete last group of elements. Therefore there are at most $n - (\lceil n/3 \rceil - 1) = \lfloor 2n/3 \rfloor + 1$ elements in the larger partition. Rounding to a simple fractions give that, in the worst case, the second recursive call is on approximately $2n/3$ elements.

2. Next (5 pts) Use these numbers to write a recurrence for $T(n)$ approximating the number of comparisons done (in the worst case) by this selection method. (Assume that the local work done to find the medians of the sets of 3 and to do partition is $2n$).

$$T(n) = T(n/3) + T(2n/3) + 2n$$

Then, (5 pts) To complete the definition of $T(n)$, assume $T(n) = 3n \lg n$ for $1 \leq n \leq 10$. Find the largest constant c you can such that $T(n) \geq cn \lg n$ for all $n \geq 1$.

We use *Constructive Induction* to find a suitable value for the constant c: (The following is not a formal proof, only a sketch used to find a suitable value for c.)

From the base cases ($1 \leq n \leq 10$) we get that $c \leq 3$. In an Inductive Step we begin with the recurrence and apply the "inductive hypothesis" that $T(j) \geq cj \lg j$ for $j < n$.

$$\begin{aligned}
T(n) &= T(n/3) + T(2n/3) + 2n \\
&\geq \frac{cn}{3} \lg \frac{n}{3} + \frac{c2n}{3} \lg \frac{2n}{3} + 2n \\
&= \frac{cn}{3}(\lg \frac{n}{3} + 2 \lg \frac{2n}{3}) + 2n \\
&= \frac{cn}{3} \lg((\frac{2n}{3})^2 \frac{n}{3}) + 2n \\
&= \frac{cn}{3} \lg(\frac{4n^3}{27}) + 2n \\
&= \frac{cn}{3}(3 \lg n + \lg 4 - \lg 27) + 2n \\
&= cn \lg n + \frac{cn}{3}(2 - \lg 27) + 2n
\end{aligned}$$

We can show that $T(n) \geq cn \lg n$ only if $\frac{cn}{3}(2 - \lg 27) + 2n$ is non-negative. The largest possible value for c makes this expression 0, so we set it to 0 and solve for c. We first simplify the notation by defining $z = 2 - \lg 27$, which is *negative*. (For the readers: other ways to write z are: $\lg(27/4)$, or $\lg(3) + 2\lg(3/2)$, and some students may be using $-z$, changing the sign in front and inverting the arguments to lg.)

$$\begin{aligned}
\frac{cn}{3}(2 - \lg 27) + 2n &= 0 \\
\frac{cnz}{3} &= -2n \\
c &= -6/z
\end{aligned}$$

So whenever $c \leq -6/z$ we can push through the inductive step. Note that $-6/z \approx 2.178 < 3$, so that when $c = -6/z$ then the constraints on both the base cases and inductive step seem to be satisfied.

(10 pts) Now that we have found a particular value for c, namely $c = -6/z$, we prove formally that $T(n) \geq cn \lg n$ for all $n \geq 1$. //

Theorem For all $n \geq 1$, $T(n) \geq cn \lg n$

Proof: by Induction on n.

Base Cases: Since $c < 3$ and $T(n) = 3n \lg n$ for $1 \leq n \leq 10$, we have that $T(n) \geq cn \lg n$ for $1 \leq n \leq 10$. It is obvious that for $1 \leq n \leq 10$ that $T(n) \geq n \lg n$.

Inductive Step: Assume $n > 10$. Assume $T(j) \geq cj \lg j, \forall 1 \leq j < n$. Show $T(n) \geq cn \lg n$.

Start with the definition of $T(n)$. Since $n > 10$ we use the Recurrence Relation:

$$\begin{aligned}
T(n) &= T(n/3) + T(2n/3) + 2n \\
&\geq \frac{cn}{3}\lg\frac{n}{3} + \frac{c2n}{3}\lg\frac{2n}{3} + 2n \ : \text{ by Ind. Hyp. } j = n/3 \text{ and } j = 2n/3 \\
&= \frac{cn}{3}(\lg\frac{n}{3} + 2\lg\frac{2n}{3}) + 2n \\
&= \frac{cn}{3}\lg((\frac{2n}{3})^2\frac{n}{3}) + 2n \\
&= \frac{cn}{3}\lg(\frac{4n^3}{27}) + 2n \\
&= \frac{cn}{3}(3\lg n + \lg 4 - \lg 27) + 2n \\
&= cn\lg n + \frac{cn}{3}(2 - \lg 27) + 2n \\
&= cn\lg n + \frac{-6n}{3z}(z) + 2n \ \text{ def. } z \text{ and } c = -6/z \\
&= cn\lg n
\end{aligned}$$

so $T(n) \geq cn\lg n$ as desired, completing the proof.