BU CLA CS 530: Analysis of Algorithms
Solutions for Mid-Term Test (16 March 1999)

Problem 1

a. (3 pts) First, sort the set S in time at most $\mathcal{O}(n \log n)$. Second, for each entry y in S, perform binary search to find a number equal to $x - y$ in time at most $n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$. The total time for the two phases is at most $\mathcal{O}(n \log n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$.

b. (3 pts) We need to find two numbers $v, w \in S$ that are "farthest apart". Let $v = \min(S)$ and $w = \max(S)$, each can be found in time $\mathcal{O}(n)$.

c. (4 pts) We need to find two numbers $v, w \in S$ that are "closest together". First sort S in $\mathcal{O}(n \log n)$ time (using heapsort or mergesort or any other sorting algorithm that works in $\mathcal{O}(n \log n)$ time in the worst-case). Then scan the sorted numbers for the two adjacent numbers with the minimum difference, in time at most $\mathcal{O}(n)$. Total time is $\mathcal{O}(n \log n)$.

Problem 2

a. (4 pts) Here is a full solution when n is not necessarily a power of 2. The size of the problem is the number of entries in the array, $r - p + 1$. We prove by induction on $n = r - p + 1$ that $\textsc{Maximum}(A, p, r)$ will return the maximum value in $A[p..r]$. The algorithm is clearly correct when $n \leqslant 2$.

Now suppose $n > 2$, and that $\textsc{Maximum}(A, p, r)$ will return the maximum value in $A[p..r]$ whenever $r - p + 1 < n$. In order to apply the induction hypothesis to the first recursive call, we must prove that $\lfloor (p + r)/2 \rfloor - p + 1 < n$. There are two cases to consider. If $r - p + 1$ is even, then $r - p$ is odd, and hence $p + r$ is odd. In this case,

$$\left\lfloor \frac{p+r}{2} \right\rfloor - p + 1 = \frac{p+r-1}{2} - p + 1 = \frac{r-p+1}{2} = \frac{n}{2} < n \ .$$

The last inequality holds because $n > 2$. For the second case, if $r - p + 1$ is odd, then $r - p$ is even, and hence $p + r$ is even. In this second case,

$$\left\lfloor \frac{p+r}{2} \right\rfloor - p + 1 = \frac{p+r}{2} - p + 1 = \frac{r-p+2}{2} = \frac{n+1}{2} < n \ .$$

The last inequality holds because $n > 1$.

In order to apply the induction hypothesis to the second recursive call, we must prove that $r - (\lfloor (p + r)/2 \rfloor + 1) + 1 < n$. There are two cases to consider. If $r - p + 1$ is even, then

$$r - \left(\left\lfloor \frac{p+r}{2} \right\rfloor + 1 \right) + 1 = r - \frac{p+r-1}{2} = \frac{r-p+1}{2} = \frac{n}{2} < n \ .$$

If $r-p+1$ is odd, then

$$r-\left(\left\lfloor\frac{p+r}{2}\right\rfloor+1\right)+1=r-\frac{p+r}{2}=\frac{r-p+1}{2}-\frac{1}{2}=\frac{n}{2}-\frac{1}{2}<n\,.$$

Procedure MAXIMUM divides the array into two parts. By the induction hypothesis, the recursive calls correctly find the maxima in these parts. Therefore, because the procedure returns the maximum of the two maxima, it return the correct value.

b. (4 pts) Let $T(n)$ be the number of comparisons used by MAXIMUM(A, p, r), where $n = r-p+1$ is the size of the array chunk processed. Then if n is a power of 2, $T(1) = 0$, and for every $n > 1$ which is a power of 2, $T(n) = 2\cdot T(n/2)+1$. Hence, using any of the techniques in Chapter 4 of [CLR], we find $T(n) = n-1$.

c. (2 pts) The running time of the procedure is a constant times the number of comparisons. Hence, by part b, the procedure runs in time $\mathcal{O}(n)$.

Problem 3

a. (5 pts) If the answer returned is *yes*, then the randomly chosen x is a majority element, and hence there is a majority element in A. On the other hand, if the answer returned is *no*, it is still possible that A contains a majority element $a \neq x$. If A does indeed contain a majority element a, then the probability that the randomly chosen x is a minority element, i.e. the probability that $x \neq a$, is less than $1/2$ — because majority elements occupy more than half the locations in A. In conclusion,

If A has a majority element, MAJOR returns *yes* with probability $> 1/2$.

If A has no majority element, MAJOR returns *no* with probability 1 (i.e. with certainty).

Hence, in all cases, the error probability $p < 1/2$.

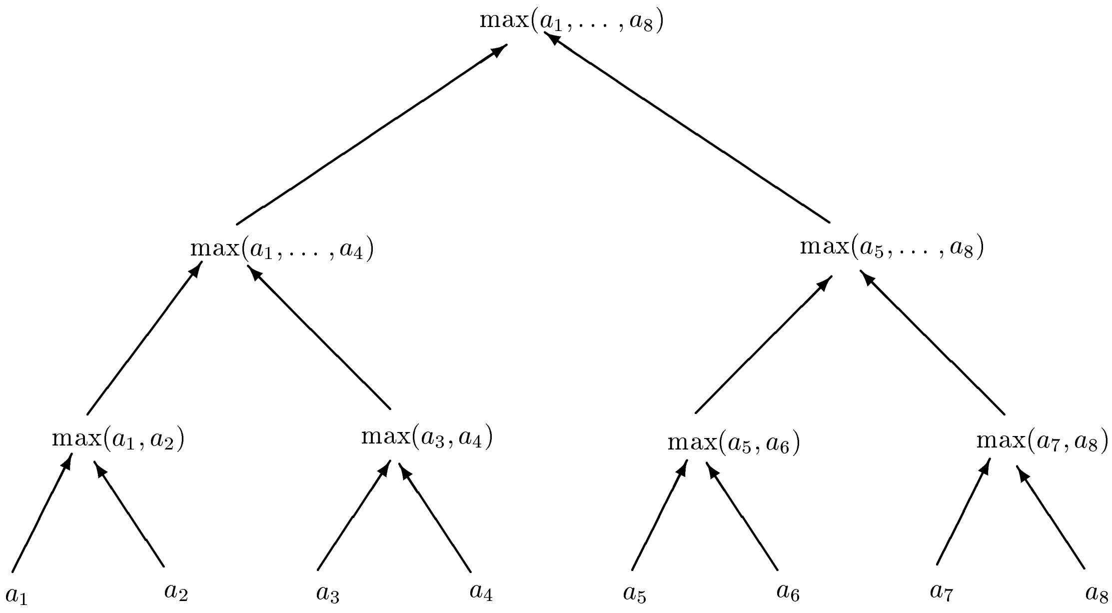
b. (5 pts) If A does not have a majority element, then each of the two calls to MAJOR is certain to return *no*, so that MAJOR$'$ also returns *no*. If A does have a majority element, the probability that the first call to MAJOR returns *yes* is $q > 1/2$, and in this case MAJOR$'$ also returns *yes*. On the other hand, if the first call to MAJOR returns *no*, which happens with probability $1-q$, the second call to MAJOR may still return *yes* with probability q, in which case MAJOR$'$ also returns *yes*. Hence, the probability that MAJOR$'$ returns *yes* when A contains a majority element is:

$$q+(1-q)\cdot q \;=\; 1-(1-q)^2 \;>\; 3/4$$

Hence, in all cases, the error probability $p' < 1/4$.

Problem 4 (10 pts)

To find the maximum value is equivalent to a tournament. The players (i.e. the entries in the array $A[1\,..\,n]$) are divided into pairs for the first round of the tournament, then all the winners of the first round are again divided into pairs, and so on until the final round. (At any round, if the number of players is odd, one of the players sits out.) The number of rounds is $\lceil \log n \rceil$. For the case $n = 8$ for example, we can describe the tournament in the form of a binary tree:



We obtain a parallel EREW algorithm by assigning a processor to every game in the tournament. However, each processor must know the two competing numbers, and for this we can put the winner of the game in the larger indexed position of the two players, i.e. if the game is played between a_i and a_j such that $i < j$, then $\max(a_i, a_j)$ is put in position j. Hence, in the first round, processor P_i compares a_{2i-1} and a_{2i}, for $i = 1, \ldots, \lfloor n/2 \rfloor$, and exchanges them if necessary. In the second round, P_i compares a_{4i-2} and a_{4i}, for $i = 1, \ldots, \lfloor n/4 \rfloor$, and so on. As each entry in A is involved in only one game at a time, the EREW model is adequate. The running time is $\mathcal{O}(\log n)$ and the number of prosessors utilized is $\mathcal{O}(n)$ (actually $\lfloor n/2 \rfloor$, so that the work performed is $\mathcal{O}(n \log n)$. Hence, this algorithm is *not* work-efficient. (You get 5 points if you reach this far.)

We can adjust the preceding algorithm to make it work-efficient. For this, we use only $\lceil n/\log n \rceil$ processors. We divide the entries in A into $\lceil n/\log n \rceil$ groups, with at most $\log n$ elements in each group and one processor assigned to each group. In a preliminary phase, each processor finds the maximum of its group using a sequential algorithm that makes no more than $\log n$ comparisons. In a second phase, we use the previously described EREW algorithm to find the maximum of the $\lceil n/\log n \rceil$ maximums. The number of comparisons carried out by this two-phase algorithm does not exceed $\log n + \log(n/\log n) < 2 \cdot \log n$, and the number of processors utilized is $\lceil n/\log n \rceil$. Hence, the work performed is $\mathcal{O}(n)$, and the algorithm is work-efficient. (You get another 5 points for this second part.)
