Introduction to Algorithms October 1, 2004
Massachusetts Institute of Technology 6.046J/18.410J
Professors Piotr Indyk and Charles E. Leiserson Handout 9

Problem Set 2 Solutions

Reading: Chapters 5-9, excluding 5.4 and 8.4

Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered in the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date and the names of any students with whom you collaborated.

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudo-code.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to correct algorithms which are *which are described clearly*. Convoluted and obtuse descriptions will receive low marks.

Exercise 2-1. Do Exercise 5.2-4 on page 98 in CLRS.

Exercise 2-2. Do Exercise 8.2-3 on page 170 in CLRS.

Problem 2-1. Randomized Evaluation of Polynomials

In this problem, we consider testing the equivalence of two polynomials in a finite field.

A *field* is a set of elements for which there are addition and multiplication operations that satisfy commutativity, associativity, and distributivity. Each element in a field must have an additive and multiplicative identity, as well as an additive and multiplicative inverse. Examples of fields include the real and rational numbers.

A *finite field* has a finite number of elements. In this problem, we consider the field of integers modulo p. That is, we consider two integers a and b to be "equal" if and only if they have the same remainder when divided by p, in which case we write $a \equiv b \bmod p$. This field, which we denote as $\mathbf{Z}/p$, has p elements, $\{0, \ldots, p-1\}$.

Consider a polynomial in the field $\mathbf{Z}/p$:

$$a(x) = \sum_{i=0}^{n} a_i x^i \bmod p \tag{1}$$

A *root* or *zero* of a polynomial is a value of x for which $a(x) = 0$. The following theorem describes the number of zeros for a polynomial of degree n.

Theorem 1 *A polynomial $a(x)$ of degree n has at most n distinct zeros.*

Polly the Parrot is a very smart bird that likes to play math games. Today, Polly is thinking of a polynomial $a(x)$ over the field $\mathbf{Z}/p$. Though Polly will not tell you the coefficients of $a(x)$, she will happily evaluate $a(x)$ for any x of your choosing. She challenges you to figure out whether or not a is equivalent to zero (that is, whether $\forall x \in \{0, \ldots, p-1\} : \ a(x) \equiv 0 \bmod p$).

Throughout this problem, assume that $a(x)$ has degree n, where $n < p$.

(a) Using a randomized query, describe how much information you can obtain after a single interaction with Polly. That is, if a is *not* equivalent to zero, then for a query x chosen uniformly at random from $\{0, \ldots, p-1\}$, what is the probability that $a(x) = 0$? What if a *is* equivalent to zero?

Solution:

If a is not equivalent to zero, then the probability that $a(x) = 0$ is at most n/p. By Theorem 1, a has at most n distinct zeros in $\{0, \ldots, p-1\}$. As we are choosing x uniformly at random from $\{0, \ldots, p-1\}$, the probability of choosing any given x in the field is $1/p$. Thus, the probability that the chosen x is a zero of a is at most n/p.

If a is equivalent to zero, then the probability that $a(x) = 0$ is 1. By definition, if a is equivalent to zero, then $a(x) = 0$ for all $x \in \{0, \ldots, p-1\}$.

(b) If $n = 10$ and $p = 101$, how many interactions with Polly do you need to be 99% certain whether or not a is equivalent to zero?

Solution:

Our strategy is to send k queries to Polly, each time evaluating $a(x)$ for a random x as described in Part (a). If each query evaluates to zero, then we report that a is equivalent to zero. Otherwise, we report that a is not equivalent to zero.

In the event that a is equivalent to zero, then our report will always be correct (even for $k = 1$). Thus, we focus on the case when a is not equivalent to zero. If we find that any query does not evaluate to zero, then our report will also be correct (since a is not equivalent to zero for a certain x).

The problem thus becomes: if a is not equivalent to zero, choose k such that the probability that all k queries evaluate to zero is no more than 1%. Let ϵ denote the margin of error in the general case ($\epsilon = 1\%$ in this part), and let Q_i be a random variable indicating the result of the ith query. The constraint is as follows:

$$\begin{aligned} \epsilon &\geq \Pr[Q_1 = 0 \text{ and } Q_2 = 0 \text{ and } \ldots \text{ and } Q_k = 0] \\ &= \Pr[Q_1 = 0]\Pr[Q_2 = 0] \ldots \Pr[Q_k = 0] \\ &\geq (n/p)^k \end{aligned}$$

The first step follows from the fact that all of the queries are independent. The second step utilizes the bound from Part (a). Solving for k, we have:

$$\begin{aligned} (n/p)^k &\leq \epsilon \\ k \lg(n/p) &\leq \lg \epsilon \\ k &\geq \lg \epsilon / \lg(n/p) \end{aligned}$$

The last step above utilizes the assumption that $n < p$, and thus $\lg(n/p) < 0$.

Substituting $\epsilon = 0.01$, $n = 10$ and $p = 101$ gives $k \geq 1.991$. Thus, $k = 2$ ensures that we are 99% certain whether or not a is equivalent to zero.

Later, you are given three polynomials: $a(x)$, $b(x)$, and $c(x)$. The degree of $a(x)$ is n, while $b(x)$ and $c(x)$ are of degree $n/2$. You are interested in determining whether or not $a(x)$ is equivalent to $b(x) * c(x)$; that is, whether $\forall x \in \{0, \ldots, p-1\} : \; a(x) \equiv b(x) * c(x) \bmod p$.

Professor Potemkin recalls that in Problem Set 1, you showed how to multiply polynomials in $\Theta(n^{\lg_2(3)})$ time. Potemkin suggests using this procedure to directly compare the polynomials. However, recalling your fun times with Polly, you convince Potemkin that there might be an even more efficient procedure, if some margin of error is tolerated.

(c) Give a randomized algorithm that decides with probability $1 - \epsilon$ whether or not $a(x)$ is equivalent to $b(x) * c(x)$. Analyze its running time and compare to Potemkin's proposal.

Solution:

The algorithm checks the equivalence of $a(x)$ and $b(x) * c(x)$ by testing if $a(x) - b(x) * c(x)$ is equivalent to zero. The testing is done using the randomized procedure developed in Part (b), choosing k to ensure the desired level of certainty. Pseudocode for the algorithm is as follows:

```
▷ Returns whether a(x) ≡ b(x) * c(x) mod p  ∀x ∈ {0, ..., p − 1}
▷ Correct with probability at least 1 − ε
EQUIV(a[0 ... n], b[0 ... n/2], c[0 ... n/2], p, ε)
1  k ← ⌈lg ε/ lg(n/p)⌉
2  for i ← 1 to k
3       do x ← RANDOM(0, q − 1)
4          a′ ← a(x)
5          b′ ← b(x)
6          c′ ← c(x)
7          if a′ − b′ * c′ ≢ 0  ( mod p)
8            then return false
9  return true
```

Correctness. For a given value of x, $a(x) = b(x) * c(x)$ if and only if $a(x) - b(x) * c(x) = 0$. Thus, $a(x)$ is equivalent to $b(x) * c(x)$ if and only if $a(x) - b(x) * c(x)$ is equivalent to zero. Our solution to Part (b) shows how to determine with probability at least $1 - \epsilon$ whether or not a given polynomial is equivalent to zero. Using this same procedure, we test whether or not $a(x) - b(x) * c(x)$ is equivalent to zero, thereby determining whether or not $a(x)$ is equivalent to $b(x) * c(x)$.

Running time. We count the number of arithmetic operations in EQUIV. In this class, we assume that steps 1 and 7 are $\Theta(1)$, as they are arithmetic operations on scalar values; also, we assume that the call to RANDOM on line 3 is $\Theta(1)$. Each polynomial evaluation on lines 4-6 is $\Theta(n)$, as an n-degree polynomial can be evaluated in $\Theta(n)$ time using Horner's rule. The loop runs $k = \lceil \lg \epsilon / \lg(n/p) \rceil$ times, performing $\Theta(n)$ work on each iteration. The total runtime is thus $T(n) = \Theta(n \lceil \lg \epsilon / \lg(n/p) \rceil)$.

Consider Potemkin's proposal, with a runtime $P(n) = \Theta(n^{\lg_2(3)})$, and let us evaluate the conditions under which $T(n) = O(P(n))$. Note that, in the runtime of our algorithm, we cannot consider p to be fixed as n grows, since we require that $n < p$.

$$\begin{aligned} T(n) &= O(P(n)) \\ n\lceil \lg \epsilon / \lg(n/p) \rceil &\leq c n^{\lg_2(3)} \\ \lceil \lg \epsilon / \lg(n/p) \rceil &\leq c n^{\lg_2(3)-1} \end{aligned} \tag{2}$$

That is, the running time of our algorithm is bounded above by the running time of Potemkin's algorithm so long as p and ϵ satisfy Equation 2 for sufficiently large n.

For example, consider that $p = c'n$ for some $c' > 1$ and ϵ is a fixed constant. Then $\lceil \lg \epsilon / \lg(n/p) \rceil = \lceil \lg \epsilon / \lg(1/c') \rceil = \Theta(1)$. Thus, the loop executes $\Theta(1)$ times and the algorithm is has a running time of $\Theta(n)$, which is asymptotically faster than Potemkin's proposal.

On the other hand, if $p = n + 1$ while ϵ remains fixed, then $\lceil \lg \epsilon / \lg(n/p) \rceil = \lceil \lg \epsilon / \lg(n/(n+1)) \rceil = \lceil \lg \epsilon / (\lg n - \lg(n+1)) \rceil$. Intuitively, one can see that this

is $\Theta(n)$ because $\frac{d}{dn}(\lg n) = 1/n$, which means that $\lg(n+1) - \lg n \approx 1/n$ and $\lceil \lg \epsilon/(\lg n - \lg(n+1)))\rceil \approx \lceil \lg \epsilon/(-1/n)\rceil = \Theta(n)$. Thus, the loop executes $\Theta(n)$ times and the algorithm has a running time of $\Theta(n^2)$, which is asymptotically slower than Potemkin's proposal.

We can prove more rigorously that $\lceil \lg \epsilon/(\lg n - \lg(n+1)))\rceil = \Theta(n)$ by appealing to the following identity [CLRS, p.53]:

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = e$$

Then, using the definition of limit, there exist positive constants c_1, c_2, and n_0 such that for all $n > n_0$:

$$\begin{aligned}
c_1 e \le \left(1+\frac{1}{n}\right)^n &\le c_2 e \\
\ln(c_1 e) \le n\ln\left(1+\frac{1}{n}\right) &\le \ln(c_2 e) && \text{[take natural log]} \\
\ln(c_1 e) \le n\ln\left(\frac{n+1}{n}\right) &\le \ln(c_2 e) && \text{[simplify]} \\
\ln(c_1 e) \le n(\ln(n+1) - \ln(n)) &\le \ln(c_2 e) && \text{[simplify]} \\
\ln(c_1 e)/n \le \ln(n+1) - \ln(n) &\le \ln(c_2 e)/n && \text{[divide by } n\text{]} \\
n/\ln(c_1 e) \ge 1/(\ln(n+1) - \ln(n)) &\ge n/\ln(c_2 e) && \text{[take inverse]} \\
n/\ln(c_1 e) \ge -1/(\ln(n) - \ln(n+1)) &\ge n/\ln(c_2 e) && \text{[simplify]}
\end{aligned}$$

Thus, $-1/(\ln(n) - \ln(n+1)) = \Theta(n)$ because it is bounded above and below by a constant factor times n. By adjusting the constants, this implies that $\lceil \lg \epsilon/(\lg n - \lg(n+1)))\rceil = \Theta(n)$.

Finally, we point out the desirable property that the algorithm is logarithmic in ϵ for fixed values of p and n. Decreasing the error margin by a given factor results in only an additive increase in the runtime.

Problem 2-2. Distributed Median

Alice has an array A[1..n], and Bob has an array B[1..n]. All elements in A and B are distinct. Alice and Bob are interested in finding the median element of their combined arrays. That is, they want to determine which element m satisfies the following property:

$$|\{i \in [1,n] : A[i] \le m\}| + |\{i \in [1,n] : B[i] \le m\}| = n \tag{3}$$

This equation says that there are a total of n elements in both A and B that are less than or equal to m. Note that m might be drawn from either A or B.

Because Alice and Bob live in different cities, they have limited communication bandwidth. They can send each other one integer at a time, where the value either falls within $\{0,\ldots,n\}$ or is drawn

from the original A or B arrays. Each numeric transmission counts as a "communication" between Alice and Bob. One goal of this problem is to minimize the number of communications needed to compute the median.

(a) Give a deterministic algorithm for computing the combined median of A and B. Your algorithm should run in $O(n \log n)$ time and use $O(\log n)$ communications. (Hint: consider sorting.)

Solution:

The algorithm works as follows. Alice and Bob begin by sorting their arrays using a deterministic $\Theta(n \log n)$ algorithm such as HeapSort or MergeSort. Then, Alice assumes the role of the master and Bob the role of the slave. Alice considers an element $A[i]$ and sends $n - i$ to Bob, who returns two elements: $B[n - i]$ and $B[n - i + 1]$. Because A is sorted, $A[i]$ is the combined median if and only if there are exactly $n - i$ elements in B that are less than $A[i]$. Because B is sorted, this condition is reduced to checking whether or not $B[n - i] < A[i] < B[n - i + 1]$. Note that when $i = n$, Bob needs to return $B[0]$, which is out of range; in this case, he returns $B[0] = -\infty$ to satisfy the above condition.

Alice checks whether $B[n - i] < A[i] < B[n - i + 1]$ and returns $A[i]$ as the median if the condition holds. If the condition fails, then she proceeds to do a binary search within her array. The search is on i, with an initial range of $[1, n]$. On each step, she descends into the top half of the range if $A[i]$ is too small (i.e., $A[i] < B[n - i]$); she descends into the bottom half if $A[i]$ is too big (i.e., $A[i] > B[n - i + 1]$). If the combined median is stored within A, the search is guaranteed to terminate at the right spot, with $B[n-i] < A[i] < B[n-i+1]$, because $A[i] - B[n-i]$ is a monotonically increasing function of i while $B[n-i+1]-A[i]$ is a monotonically decreasing function of i.

If the combined median is not held in A, then Alice's binary search terminates after $1 + \lceil \lg n \rceil$ steps and returns a value of NIL. In this case, Alice and Bob swap roles, with Bob becoming the master and Alice the slave. The procedure is repeated, and this time the binary search returns the combined median because it must be stored within B.

For clarity, pseudocode for this algorithm[1] is given below.

[1]Note that the DONE value used on line 15 of MASTER and line 3 of SLAVE can be implemented by preceding each communication with a 0 or 1 to indicate whether or not the following value indicates DONE. This does not impact the asymptotic running time or communication cost.

```
ALICE(A[1...n])
 1  HEAPSORT(A)
 2  median ← MASTER(A)
 3  if median = NIL
 4     then median ← SLAVE(A)
 5  return median
```

```
BOB(B[1...n])
 1  HEAPSORT(B)
 2  median ← SLAVE(B)
 3  if median = NIL
 4     then median ← MASTER(B)
 5  return median
```

```
MASTER(M[1...n])
 1  lower ← 1
 2  upper ← n
 3  median ← NIL
 4  while lower ≤ upper and median = NIL
 5        do i ← ⌈lower + (upper − lower)/2⌉
 6           send n − i
 7           receive b_1
 8           receive b_2
 9           cur ← M[i]
10           if b_1 < cur < b_2
11              then median= cur
12           elseif cur < b_1
13              then lower ← i +1
14              else upper ← i −1
15  send DONE
16  send median
17  return median
```

```
SLAVE(S[1...n])
 1  while true
 2        do receive j
 3           if j = DONE
 4              then receive median
 5                   return median
 6           if j = 0
 7              then send −∞
 8              else send S[j]
 9           send S[j + 1]
```

Running Time. Each individual statement is $\Theta(1)$ except for the calls to HeapSort (line 1 of ALICE and BOB), which requires $\Theta(n \lg n)$ time. The loop in MASTER performs a binary search, which (as we saw in lecture) requires $\Theta(\lg n)$ iterations. Each iteration does $\Theta(1)$ work, so the total running time for the loop is $\Theta(\lg n)$. The loop in SLAVE terminates when it receives a DONE value, which happens exactly when the loop in MASTER terminates; thus, SLAVE is also $\Theta(\lg n)$. Alice and Bob each execute MASTER, SLAVE and HeapSort; HeapSort dominates, yielding a final running time of $\Theta(n \lg n)$.

Communication cost. Most of the communication is in the loop of MASTER, in which three items are relayed between Alice and Bob per each iteration. Since this loop executes $\Theta(\lg n)$ times, it contributes $\Theta(\lg n)$ communications. The items sent and received at the end of MASTER contribute $\Theta(1)$ communications, leaving the total at $\Theta(\lg n)$.

Alternate Solution:

An alternate solution is simpler in some ways, but cannot be adapted to solve Part (b).

As before, Alice and Bob begin by sorting their arrays using a deterministic $\Theta(n \log n)$ algorithm such as HeapSort or MergeSort. Then, Alice assumes the role of the master and Bob the role of the slave. When Alice sends a value $A[i]$ to Bob, Bob returns the number of elements, *count*$(A[i])$, in his array that are less than $A[i]$. Because A is sorted, the element $A[i]$ is the combined median if and only if $i +$ *count*$(A[i]) = n$. Alice checks this condition and returns $A[i]$ as the median if the condition holds. If the condition fails, then she proceeds to do a binary search within her array. The search is on i, with an initial range of $[1, n]$. On each step, she descends into the top half of the range if $i +$ *count*$(A[i]) < n$; she descends into the bottom half if $i +$ *count* $> n$. Because the quantity $i+count(A[i])$ is a monotonic function of i, the search terminates with $A[i] = i +$ *count*$(A[i])$ if the combined median is stored within A.

If the combined median is not held in A, then Alice's binary search terminates after $1 + \lceil \lg n \rceil$ steps and returns a value of NIL. In this case, Alice and Bob swap roles, with Bob becoming the master and Alice the slave. The procedure is repeated, and this time the binary search returns the combined median because it must be stored within B.

For clarity, pseudocode for this algorithm is given below.

```
ALICE(A[1...n])
 1  HEAPSORT(A)
 2  median ← MASTER(A)
 3  if median = NIL
 4     then median ← SLAVE(A)
 5  return median
```

```
BOB(B[1...n])
 1  HEAPSORT(B)
 2  median ← SLAVE(B)
 3  if median = NIL
 4     then median ← MASTER(B)
 5  return median
```

```
MASTER(M[1...n])
 1  lower ← 1
 2  upper ← n
 3  median ← NIL
 4  while lower ≤ upper and median = NIL
 5        do i ← ⌈lower + (upper − lower)/2⌉
 6           send A[i]
 7           receive count
 8           if i + count = n
 9              then median= M[i]
10           elseif i + count < n
11              then lower ← i +1
12              else upper ← i −1
13  send DONE
14  send median
15  return median
```

```
SLAVE(S[1...n])
1  while true
2        do receive val
3           if val = DONE
4              then receive median
5                   return median
6           else send |i ∈ [1, n] : S[i] ≤ val|
```

Running Time. All but three statements are $\Theta(1)$ time. Both Alice and Bob call HeapSort, which is $\Theta(n \lg n)$. Line 6 of SLAVE counts how many elements in S are at most *val*. This can be implemented in $\Theta(n)$ time with a brute-force comparison or, because the array is sorted, in $\Theta(\lg n)$ time using a binary search. The last statements of interest are lines 6-7 of MASTER, which wait for one iteration of SLAVE. Since the slave executes $\Theta(\lg n)$ operations between a receive and send statement, lines 6-7 of MASTER are also $\Theta(\lg n)$.

It remains to account for the loops. The loop in MASTER is performing a binary search, which (as we saw in lecture) requires $\Theta(\lg n)$ iterations. Each iteration does $\Theta(\lg n)$ work, so the total running time for the loop is $\Theta(\lg^2 n)$. The loop in SLAVE terminates when it receives a DONE value, which happens exactly when the loop in MASTER terminates; thus, SLAVE is also $\Theta(\lg^2 n)$. Alice and Bob each execute MASTER, SLAVE and HeapSort; HeapSort dominates, yielding a final running time of $\Theta(n \lg n)$.

Communication cost. Most of the communication is in the loop of MASTER, in which two items are relayed between Alice and Bob per each iteration. Since this loop executes $\Theta(\lg n)$ times, it contributes $\Theta(\lg n)$ communications. The items sent and received at the end of MASTER contribute $\Theta(1)$ communications, leaving the total at $\Theta(\lg n)$.

(b) Give a randomized algorithm for computing the combined median of A and B. Your algorithm should run in expected $O(n)$ time and use expected $O(\log n)$ communications. (Hint: consider RANDOMIZED-SELECT.)

Solution:

The algorithm is almost identical to Part (a). As before, Alice starts as the master and conducts a binary search through Bob's elements, looking for $A[i]$ such that $B[n-i] < A[i] < B[n-i+1]$. The only difference is that, instead of finding the ith largest element by sorting the arrays and referencing $A[i]$ directly, we use RANDOMIZED-SELECT to find the ith largest element in expected linear time. As the algorithm is so close to Part (a), we describe only the differences below; in the pseudocode, the modified lines are denoted by $\star$.

```
ALICE(A[1...n])
 1  HEAPSORT(A)
 2  median ← MASTER(A)
 3  if median = NIL
 4     then median ← SLAVE(A)
 5  return median
```

```
BOB(B[1...n])
 1  HEAPSORT(B)
 2  median ← SLAVE(B)
 3  if median = NIL
 4     then median ← MASTER(B)
 5  return median
```

```
MASTER(M[1...n])
 1  lower ← 1
 2  upper ← n
 3  median ← NIL
 4  while lower ≤ upper and median = NIL
 5      do i ← ⌈lower + (upper − lower)/2⌉
 6         send n − i
⋆7         send n − upper
⋆8         send n − lower
 9         receive b1
10         receive b2
⋆11        cur ← RANDOMIZED-SELECT(M, lower, upper, i − lower + 1)
12         if b1 < cur < b2
13           then median= cur
14         elseif cur < b1
15           then lower ← i +1
16           else upper ← i −1
17  send DONE
18  send median
19  return median

SLAVE(S[1...n])
 1  while true
 2      do receive j
 3         if j = DONE
 4           then receive median
 5                return median
⋆6         receive bottom
⋆7         receive top
 8         if j = 0
 9           then send −∞
⋆10          else send RANDOMIZED-SELECT(S, bottom, top, j − bot + 1)
⋆11        send RANDOMIZED-SELECT(S, bottom, top, j − bot + 2)
```

Correctness. We need to show that all calls to RANDOMIZED-SELECT return the corresponding values from Part (a). On line 11 of MASTER, we have replaced $M[i]$, the ith smallest value of the sorted array M, with RANDOMIZED-SELECT(M, *lower*, *upper*, $i-$ *lower*$+1$). To show that these values are equivalent, we need the following loop invariant I: before each iteration of the loop, $M[$*lower*$\ldots$*upper*$]$ contains the ith smallest element in M for all $i \in \{$*lower*$, \ldots,$ *upper*$\}$. This invariant is true at the beginning, since *lower* $= 1$, *upper* $= n$ and all of the elements are within the range.

For the inductive step, assume I is true on the current iteration. Then the call to RANDOMIZED-SELECT will partition around the ith smallest element in M, because 1) (by the inductive hypothesis) the smallest element in $M[\textit{lower} \ldots \textit{upper}]$ is the *lower*'th smallest element in M, 2) by our call to RANDOMIZED-SELECT, we are selecting for the $i - \textit{lower}$'th smallest element in $M[\textit{lower} \ldots \textit{upper}]$ (we also add 1 to compensate for the 1-based array indexing) and 3) $\textit{lower} + i - \textit{lower} = i$. Thus, the hypothesis will be satisfied for the range $\{\textit{lower}, \ldots, i\}$ and $\{i, \ldots, \textit{upper}\}$ because the PARTITION subroutine of RANDOMIZED-SELECT will place elements on the appropriate side of the pivot i. Finally, the inductive hypothesis will hold on the next iteration because we assign either *lower* or *upper* to be adjacent to i (but excluding i from the next range).

Using the invariant, we conclude that the call to RANDOMIZED-PARTITION on line 11 of MASTER returns the ith smallest element in M, which is equivalent to the expression $M[i]$ from Part (a).

It remains to show the equivalent property for lines 10 and 11 of SLAVE. This is done using the same loop invariant, but translating $j = n - i$, $\textit{bottom} = n - \textit{upper}$, and $\textit{top} = n - \textit{lower}$ across the call between MASTER and SLAVE. In this way, we can show that lines 10 and 11 of SLAVE return the $n-i$th and $n-i+1$th smallest elements of S, respectively.

We have shown that our changes from Part (a) preserve the behavior of the algorithm, and thus the algorithm remains correct.

Running Time. We can write a recurrence to model the running time of the main loop in MASTER. Let $m = \textit{upper} - \textit{lower}$. On each iteration, m decreases to at most $\lceil m/2 \rceil$ and RANDOMIZED-SELECT runs three times (once in MASTER, twice in SLAVE) over a segment of size m, with expected running time $\Theta(m)$. Thus $E[T(m)] = E[T(\lceil m/2 \rceil)] + \Theta(m)$, and by Case 3 of the Master Theorem, $E[T(m)] = \Theta(m)$. Finally, noting that $m = n$ at the beginning of the procedure, we have that the expected running time is $\Theta(n)$.

Communication cost. The communication cost is identical to Part (a), as the loop in MASTER still executes $\Theta(\lg n)$ iterations and sends $\Theta(1)$ items on each iteration. Thus, the total number of communications is $\Theta(\lg n)$. (Note that this algorithm gives a deterministic bound on the number of communications.)

Problem 2-3. American Gladiator

You are consulting for a game show in which n contestants are pitted against n gladiators in order to see which contestants are the best. The game show aims to rank the contestants in order of strength; this is done via a series of 1-on-1 matches between contestants and gladiators. If the contestant is stronger than the gladiator, then the contestant wins the match; otherwise, the gladiator wins the

match. If the contestant and gladiator have equal strength, then they are "perfect equals" and a tie is declared. We assume that each contestant is the perfect equal of exactly one gladiator, and each gladiator is the perfect equal of exactly one contestant. However, as the gladiators sometimes change from one show to another, we do *not* know the ordering of strength among the gladiators.

The game show currently uses a round-robin format in which $\Theta(n^2)$ matches are played and contestants are ranked according to their number of victories. Since few contestants can happily endure $\Theta(n)$ gladiator confrontations, you are called in to optimize the procedure.

(a) Give a randomized algorithm for ranking the contestants. Using your algorithm, the expected number of matches should be $O(n \log n)$.

Solution:

The problem statement does not describe exactly how the contestants and gladiators are specified, so we first need to come up with a reasonable representation for the input. Let's assume the contestants and gladiators are provided to us in two arrays $C[1 \ldots n]$ and $G[1 \ldots n]$, where we are allowed to compare elements across, but not within, these two arrays.

We use a divide-and-conquer algorithm very similar to randomized quicksort. The algorithm first performs a partition operation as follows: pick a random contestant $C[i]$. Using this contestant, rearrange the array of gladiators into three groups of elements: first the gladiators weaker than $C[i]$, then the gladiator that is the perfect equal of $C[i]$, and finally the gladiators stronger than $C[i]$. Next, using the gladiator that is the perfect equal of $C[i]$ we perform a similar partition of the array of contestants. This pair of partitioning operations can easily be implemented in $\Theta(n)$ time, and it leaves the contestants and gladiators nicely partitioned so that the "pivot" contestant and gladiator are aligned with each other and all other contestants and gladiators are on the correct side of these pivots — weaker contestants and gladiators precede the pivots, and stronger contestants and gladiators follow the pivots. Our algorithm then finishes by recursively applying itself to the subarrays to the left and right of the pivot position to sort these remaining contestants and gladiators. We can assume by induction on n that these recursive calls will properly sort the remaining contestants.

To analyse the running time of our algorithm, we can use the same analysis as that of randomized quicksort. We are performing a partition operation in $\Theta(n)$ time that splits our problem into two subproblems whose sizes are randomly distributed exactly as would be the subproblems resulting from a partition in randomized quicksort. Therefore, applying the analysis from quicksort, the expected running time of our algorithm is $\Theta(n \log n)$.

Interesting side note: Although devising an efficient randomized algorithm for this problem is not too difficult, it appears to be very difficult to come up with a deterministic algorithm with running time better than the trivial bound of $O(n^2)$. This

remained an open research question until the mid-to-late 90's, when a *very* complicated deterministic algorithm with $\Theta(n \log n)$ running time was finally discovered. This problem provides a striking example of how randomization can help simplify the task of algorithm design.

(b) Prove that any algorithm that solves part (a) must use $\Omega(n \log n)$ matches in the *worst case*. That is, you need to show a lower bound for any deterministic algorithm solving this problem.

Solution:

Let's use a proof based on decision trees, as we did for comparison-based sorting. Note that we can model any algorithm for sorting contestants and gladiators as a decision tree. The tree will be a ternary tree, since every comparison has three possible outcomes: weaker, equal, or stronger. The height of such a tree corresponds to the worst-case number of comparisons made by the algorithm it represents, which in turn is a lower bound on the running time of that algorithm. We therefore want a lower bound of $\Omega(n \log n)$ on the height, H, of any decision tree that solves part (a). To begin with, note that the number of leaves L in any ternary tree must satisfy

$$L \leq 3^H.$$

Next, consider the following class of inputs. Let the input array of gladiators G be fixed and consist of n gladiators sorted in order of increasing strength, and consider one potential input for every permutation of the contestants. Our algorithm must in this case essentially sort the array of contestants. In our decision tree, if two different inputs of this type were mapped to the same leaf node, our algorithm would attempt to apply to both of these the same permutation of contestants, and it follows that the algorithm could not compute a ranking correctly for both of these inputs. Therefore, we must map every one of these $n!$ different inputs to a distinct leaf node, so

$$\begin{aligned} L &\geq n! \\ 3^H &\geq n! \\ H &\geq \log_3 n! \\ H &= \Omega(n \log n) \quad \text{[Using Stirling's approximation]} \end{aligned}$$