CSOR W4231: Homework 2

Problem 1 (18 points, Graded by Niloofar and Rabia)

a. $T(n) = 4T(n/2) + n^2$

We use the master theorem. The parameters are $a = 4, b = 2, f(n) = n^2$ and $n^{\log_b a} = n^{\log_2 4} = n^2$. Case II of the master theorem applies, giving us $T(n) = \Theta(n^2 \lg n)$.

b. $T(n) = 3T(n/4) + n$

Again applying the master theorem, the parameters are $a = 3, b = 4, f(n) = n$ and $n^{\log_b a} = n^{\log_4 3}$. Since $\log_4 3 < 1$, there exists $\epsilon > 0$, such that $f(n) = \Omega(n^{\log_4 3+\epsilon})$. Checking $af(n/b) \leq cf(n) \Rightarrow 3n/4 \leq cn$ for $c \geq 3/4$. Thus, using Case III of the master theorem, we get $T(n) = \Theta(n)$.

c. $T(n) = 3T(n/2) + n\log^2 n$

We have $a = 3, b = 2, f(n) = n\log^2 n$ and $n^{\log_b a} = n^{\log_2 3} = O(n^{1.585})$. Since $f(n) = O(n^{\log_2 3-\epsilon})$ for some $\epsilon > 0$. Case I applies, giving us $T(n) = \Theta(n^{\log_2 3})$.

d. $T(n) = 2T(n-3) + 1$

Let $c = T(n \mod 3)$ (a constant). Expanding (unfolding) the recursion, we have:

$$\begin{aligned}
T(n) &= 2T(n-3) + 1 \\
&= 4T(n-6) + 2 + 1 \\
&= 8T(n-9) + 4 + 2 + 1 \\
&= \dots \\
&= 2^i T(n-3i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 \\
&= \dots \\
&= 2^{\lfloor n/3 \rfloor} T(n \mod 3) + 2^{\lfloor n/3 \rfloor - 1} + \cdots + 2 + 1 \\
&= 2^{\lfloor n/3 \rfloor}(c+1) - 1 \\
&= \Theta(2^{n/3})
\end{aligned}$$

e. $T(n) = T(\sqrt{n}) + 1$

One way is to observe that when we unfold the recurrence, at every iteration $\log n$ is halved, suggesting that the number of iterations until we reach the base case is $\log\log n$, and hence $T(n) = \Theta(\log\log n)$. We can show this also using a change of variables. Let $m = \log n$.

$$T(2^m) = T(2^{m/2}) + 1$$

Similar to the notes we rename $S(m) = T(2^m)$:

$$S(m) = S(m/2) + 1$$

$$S(m) = \Theta(\log m)$$

Thus, substituting our changed variables back in:

$$S(\log n) = \Theta(\log \log n)$$

$$T(2^{\log n}) = \Theta(\log \log n)$$

$$T(n) = \Theta(\log \log n)$$

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

Notice that if we expand the recursion tree, we see that at each subsequent level, the n operations performed at each level is reduced by a constant proportion, namely $\frac{1}{2} + \frac{1}{3} = \frac{5}{6}$. This implies that we can guess $T(n) = \Theta(n)$ and prove it using the substitution method.

First we prove $T(n) = O(n)$. We would like to show $T(n) \leq cn$ for some constant c, using induction. We have:

$$\begin{aligned} T(n) &\leq cn/2 + cn/3 + n \\ &= \frac{5}{6}cn + n \\ &\leq cn \quad \text{for } c \geq 6 \end{aligned}$$

For the lower bound we will check $T(n) = \Omega(n)$. It suffices to show $T(n) \geq c'n$ for some constant c', by induction. We have:

$$\begin{aligned} T(n) &\geq c'n/2 + c'n/3 + n \\ &= \frac{5}{6}c'n + n \\ &\geq c'n \quad \text{for } c' \leq 6 \end{aligned}$$

With the upper and lower bound verified, we have proven that $T(n) = \Theta(n)$.

Problem 2 (10 points, Graded by Malhar)

1.

The probability that $A[1] > A[2]$ is 1/2. To see this, pair up all the permutations of $1, \ldots, n$, where each permutation is paired with the permutation that is the same except that it swaps the first two elements $A[1], A[2]$. In each pair, one permutation has $A[1] < A[2]$ and the other has $A[1] > A[2]$. Hence 1/2 of the permutations have $A[1] > A[2]$. Therefore, the probability that a random permutation has $A[1] > A[2]$ is 1/2.

2.

Let us consider a collection of indicator random variables I_{ij}, with $i, j \in \{1, 2, \ldots, n\}$, $i < j$, to indicate whether $A[i] > A[j]$. That is, I_{ij} has value 1 if $A[i] > A[j]$, and $I_{ij} = 0$ otherwise. As in part 1, the probability that $I_{ij} = 1$, i.e., that $A[i] > A[j]$, is 1/2 for every pair i, j. Hence the expected value of I_{ij} is $E[I_{ij}] = 1/2$.

The number X of pairs of indices i, j such that $1 \leq i < j \leq n$ and $A[i] > A[j]$ is $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} I_{ij}$. By linearity of expectation, the expectation of X is

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[I_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} = \frac{1}{2} \sum_{i=1}^{n-1} (n-i) = \frac{1}{2} \sum_{k=1}^{n-1} k = \frac{n(n-1)}{4} \ .$$

Problem 3 (15 points, Graded by Ananth)

1. Construct a new array $B[2 \dots n]$ where $B[i] = |A[i] - A[1]|$ for every $i = 2, \dots n$.
2. Use the linear-time deterministic SELECT algorithm from class to find the k-th smallest element of $B[2 \dots n]$, and let x be that element.
3. Iterate through the array A, and for each $i = 2, \dots, n$ do the following: if $|A(i) - A(1)| < x$ then add $A[i]$ to the solution list Sol of elements that will be output, and if $|A(i) - A(1)| = x$ then add $A[i]$ to a list L of tied elements. Output the set Sol and $k - |Sol|$ elements from the list L to get k elements.

Runtime: Corresponding to the steps above:

1. $\Theta(n)$ - a simple iteration through an array of size n, doing O(1) operations at each element.
2. $\Theta(n)$ - we know from class that SELECT runs in time $\Theta(n)$.
3. $\Theta(n)$ - a simple iteration through an array of size n + an additional at most k operations.

Total: $\Theta(n) + \Theta(n) + \Theta(n)$ gives us a worst-case runtime of $\Theta(n)$.

Correctness: The k elements $A[i]$ that are closest to $A[1]$ are precisely those that correspond to the k smallest elements $B[i] = |A[i] - A[1]|$. The SELECT algorithm correctly computes the k-th smallest element x of B. The k smallest elements of B are (i) all the elements that are strictly smaller than x, and (ii) enough elements with value x to obtain exactly k elements. Step 3 of the algorithm traverses the array A and outputs the corresponding elements of A, which consist of the set Sol of elements that have distance less than x from $A[1]$ and a sufficient number of elements from L that have distance x to get exactly k elements.

Problem 4 (17 points, Graded by Ziqing and Zhenchen)

We present an algorithm below using divide and conquer. The basic idea is find a k-quantile (one of the elements of rank $\lceil \frac{n}{k} \rceil, \lceil \frac{2n}{k} \rceil, \dots, \lceil \frac{(k-1)n}{k} \rceil$) of the input array A using the selection algorithm seen in class and then recurse on the left and right partitioned subarrays to get the k-quantiles with rank smaller than and larger than the current k-quantile respectively. In order to evenly split the problem into two, we use the middle k-quantile to get a balanced partition, so that each of the two subproblems calls for finding $(k-1)/2$ k-quantiles; this will lead to time complexity proportional to $\log k$.

```
input : An array $A[1, \dots, n]$ and an integer $k < n$ representing the quantiles we want to find
function find-quantiles($A, k$):
    $n = A.length$;
    find-quantiles-helper($A, 1, n, 1, k-1$);

function find-quantiles-helper($A, i, j, p, q$):
    if $p \leq q$ then
        $mid = \lceil (p+q)/2 \rceil$;
        $r = \lceil \frac{mid \cdot n}{k} \rceil$;
        $x$ = select($A$, $i$, $j$, $r$ - $i$ + $1$);
        find-quantiles($A, i, r-1, p, mid-1$);
        print($x$);
        find-quantiles($A, r+1, j, mid+1, q$);
```

The recursive subroutine find-quantiles-helper(A, i, j, p, q) prints (in order) from the pth to the qth $k-$quantile of A, where $1 \leq p \leq q \leq k-1$, i.e., it prints the elements of rank $\lceil pn/k \rceil$, $\lceil (p+1)n/k \rceil$, ..., $\lceil qn/k \rceil$ of A. The select(A, i, j, t) routine is the linear-time deterministic Select algorithm, called on the subarray $A[i \dots j]$ to find the t-th smallest element x of the subarray. In the algorithm as written above, it is assumed that the Select routine partitions during its execution the elements of the subarray with respect to x (as is the case with the algorithm in the book). Otherwise, we would have to partition the subarray $A[i \ldots j]$ around the element x returned in line 8.

Correctness
An invariant of the algorithm is that whenever the find-quantile-helper function is called with arguments (A, i, j, p, q), the index i is $\lceil (p-1)n/k \rceil + 1$, the index j is $\lceil (q+1)n/k \rceil - 1$ if $q < k-1$, and $j = n$ if $q = k-1$, and the subarray $A[i \ldots j]$ of A contains the elements of A with rank i through j (but not necessarily in sorted order). The invariant trivially holds for the first call find-quantile-helper$(A, 1, n, 1, k-1)$. It is straightforward to verify that it continues to hold in every subsequent recursive call. Indeed suppose it holds for a call with arguments (A, i, j, p, q). Since the subarray $A[i \ldots j]$ contains the elements of A with rank i through j, when the select routine finds the element x of rank $r - i + 1$ in the subarray $A[i \ldots j]$, this is the element of rank $r = \lceil mid \cdot n/k \rceil$ of the whole array A. The subarray $A[i \ldots j]$ is partitioned with respect to x, hence $A[i \ldots r-1]$ contains the elements of the array A with rank i through $r-1$ and $A[r+1 \ldots j]$ contains the elements of A with rank $r+1$ through j. The first recursive call has arguments $(A, i, r-1, p, mid-1)$ which satisfy the properties $i = \lceil (p-1)n/k \rceil + 1$ and $r - 1 = \lceil mid \cdot n/k \rceil - 1$ by the induction hypothesis and the definition of r, and similarly with the arguments of the second recursive call. Hence the properties of the invariant continue to hold for the recursive calls that are generated.

The Correctness now follows from the recursive structure of the algorithm. We are guaranteed to find correctly all the quantiles since we simply get correctly the middle quantile by the correctness of the Select algorithm, we partition the array and then recurse on the left and right halves to find the remaining quantiles.

Time Complexity
The find-quantile-helper function generates two recursive calls, where each of the two calls involves half of the quantiles, and the sum of the sizes of the two subarrays in the two calls is equal to the size of the subarray in the original call minus 1 (for the element that is printed). Since the number of quantiles is halved in each recursive call, the recursion tree is $\log k$ levels deep. The select algorithm runs in linear time in the size of the subarray. Therefore the work done at each level is $O(n)$, since we are running select on a set of disjoint subarrays of combined size n (at most). Thus, the worst-case complexity of this algorithm is $O(n \log k)$.

Problem 5 (20 points, Graded by Shenzhi and Xingyu)

1.

It is $\Theta(n^2)$. To see why notice that the PARTITION phase will always put all the elements of the current sub-array on the left of the pivot element. Therefore, if the current sub-array has size n the two sub-arrays created from PARTITION will have size $n-1$ and 0 respectively. Hence, the recurrence relation describing the running time of QUICKSORT will be $T(n) = T(n-1) + \Theta(n)$ whose solution is $T(n) = \Theta(n^2)$.

2.

```
$x = A[r]$
$q_1 = p - 1$
$q_2 = p - 1$
for $j$=$p$ to $r$-1 do
    if $A[j] == x$ then
        $q_2 = q_2 + 1$
        exchange $A[q_2]$ with $A[j]$
    end
    else if $A[j] < x$ then
        $q_2 = q_2 + 1$
        exchange $A[q_2]$ with $A[j]$
        $q_1 = q_1 + 1$
        exchange $A[q_1]$ with $A[q_2]$
    end
end
$q_2 = q_2 + 1$
exchange $A[q_2]$ with $A[r]$
return $q_1$, $q_2$
```

Algorithm 1: PARTITION($A[p..r]$)

Correctness: We will use the loop-invariant shown in figure 1.

At the beginning of each iteration holds that:

1. $q_1, q_2 < j$
2. $A[k] < x$ for all $k \leq q_1$
3. $A[k] = x$ for all $q_1 < k \leq q_2$
4. $A[k] > x$ for all $q_2 < k \leq j - 1$

Figure 1: Loop invariant for PARTITION.

Initialization: Before the first iteration the invariant trivially holds because initially $q_1 = q_2 = p - 1 < j = p$ and there is no k in the range of the array such that $k < q_1$, $k \leq q_2$, or $k \leq j - 1$.

Maintenance: Assume that before some iteration the loop invariant is satisfied, i.e. $q_1, q_2 < j$, $A[k] < x$ for all $k \leq q_1$, $A[k] = x$ for all $q_1 < k \leq q_2$, and $A[k] > x$ for all $q_2 < k \leq j - 1$. We will prove that after the iteration the invariant is still true, i.e. $q_1, q_2 < j + 1$, $A[k] < x$ for all $k \leq q_1$, $A[k] = x$ for all $q_1 < k \leq q_2$, and $A[k] > x$ for all $q_2 < k \leq j$. Indeed, we have:

1. If $A[j] > x$ then q_1 and q_2 are not increased and no element is affected. Notice also that since $A[j] > x$ it now holds that $A[k] > x$ for all $q_2 < k \leq j$. Hence the invariant is still true after the iteration.
2. If $A[j] = x$ then q_2 is increased by 1 while q_1 is not increased and no element before position q_1 is affected. Therefore we still have $A[k] < x$ for all $k \leq q_1$ and it now holds that $q_1, q_2 < j + 1$ since previously he had $q_1, q_2 < j$. Notice also that after q_2 is increased $A[q_2]$ is exchanged with $A[j] = x$, hence, we still have that $A[k] = x$ for all $q_1 < k \leq q_2$. Finally, if $q_2 < j - 1$ then after q_2 is increased by 1 we have $A[q_2] > x$ because of the assumption that the invariant holds before the iteration. Hence, after exchanging $A[j]$ with $A[q_2] > x$ it holds that $A[k] > x$ for all $q_2 < k \leq j$. On the other hand, if $q_2 = j - 1$ then after increasing it $A[k] > x$ for all $q_2 < k \leq j$ is trivially satisfied. It follows that in this case the invariant is still true after the iteration.

3. If $A[j] < x$ then both q_1 and q_2 are increased by 1, therefore we now have $q_1, q_2 < j+1$ since previously he had $q_1, q_2 < j$. Also, after q_1 is increased $A[q_1]$ gets, through a double exchange with $A[q_2]$ and $A[j]$, a value less than x, hence, it still holds that $A[k] < x$ for all $k \leq q_1$. Furthermore, after we increase q_1 and q_2 by 1 and before exchanging $A[q_2]$ with $A[q_1]$ we have $A[q_1] = x$ because of the assumption that the invariant holds before the iteration. Hence, after exchanging $A[q_1] = x$ with $A[q_2]$ it holds that $A[k] = x$ for all $q_1 < k \leq q_2$. Finally, we can prove that after the iteration $A[k] > x$ for all $q_2 < k \leq j$ with exactly the same proof as for the $A[j] = x$ case. Hence, the invariant is still true after the iteration.

It follows that if the invariant is true before an iteration then it will also be true after it.

Termination: From the maintenance rule we get that when the loop terminates it holds that $q_1, q_2 < r$, $A[k] < x$ for all $k \leq q_1$, $A[k] = x$ for all $q_1 < k \leq q_2$, and $A[k] > x$ for all $q_2 < k \leq r-1$. Therefore, after we increase q_2 and swap $x = A[r]$ with $A[q_2]$ it holds that $A[k] < x$ for all $k \leq q_1$, $A[k] = x$ for all $q_1 < k \leq q_2$, and $A[k] > x$ for all $q_2 < k \leq r$. The correctness of PARTITION follows.

Running Time: The loop iterates through the array only once, therefore PARTITION runs in $\Theta(n)$ time.

3.

```
if $p < r$ then
    i=RANDOM(p,r)
    exchange $A[r]$ with $A[i]$

    $q_1, q_2$=PARTITION($A[p..r]$)

    R-QSORT(A[$p..q_1$])
    R-QSORT($A[q_2 + 1..r]$)
end
```

Algorithm 2: R-QSORT($A[p..r]$)

We modify RANDOMIZED-QUICKSORT so that it now uses the PARTITION routine from part (2) on $A[p..q]$ and then it recursively continues on sub-arrays $A[p..q_1]$ and $A[q_2 + 1..r]$. Algorithm 2 implements this modification. R-QSORT is correct because after the PARTITION routine is called every element in $A[q_1 + 1..q_2]$ is equal to x, every element in $A[p..q_1]$ is less than x, and every element in $A[q_2 + 1..r]$ is larger than x, hence, recursively sorting $A[p..q_1]$ and $A[q_2 + 1..r]$ results in a sorted $A[p..q]$.

Notice now that the analysis of the running time of RANDOMIZED-QUICKSORT given in book uses the fact that the elements are distinct only once: When it argues that once a pivot $z_i < x < z_j$ is chosen then z_i and z_j cannot be compared at any subsequent time, it doesn't need to argue about the case $z_i = x = z_j$ because the elements are distinct. In our case, even if $z_i = x = z_j$ we know that z_i and z_j cannot be compared at any subsequent time because after PARTITION they will both end up in the $A[q_1 + 1..q_2]$ part of the array that contains all elements equal to x and on which the algorithm does not recurse. It follows that the analysis given in book also applies in R-QSORT. Therefore, R-QSORT runs in expected $O(n \log n)$ time for every input array.

Problem 6 (20 points, Graded by Aditya)

(a)

Suppose there are k good chips where $k \leq \frac{n}{2}$. Let A be the set of good chips and divide the set of bad chips into two subsets B, C where $|B| = k$ and $|C| = n - 2k$. By definition, every chip in A answers GOOD if it is paired with another chip in A and BAD otherwise. If the chips in the set B mimic the chips in A pretending to be the good chips, then the tester will not be able to decide whether the set of good chips is A or B. More precisely, suppose the bad chips behave as follows:

1. When a bad chip in B is paired with another chip in B it marks it as good, and when it is paired with a chip in A or C it marks it as bad.
2. A bad chip in C marks every other chip as bad.

Note that all chips in A are marked good by the chips in A and bad by all the other chips; similarly all chips in B are marked good by all chips in B and bad by all the other chips. Therefore, the subsets A and B are indistinguishable, and the tester cannot tell whether the set of good chips is A or B.

(b)

We pair the chips in couples, leaving one chip aside if n is odd, i.e. we make $\lfloor \frac{n}{2} \rfloor$ couples and we perform mutual tests between the chips in each couple. We then partition the pairs according to their answers into two sets. G is the set of the pairs of chips that both answered GOOD and B contains all other pairs. Notice that every pair in B must contain at least one bad chip. For every pair in B we eliminate both chips while for every pair in G we choose one chip arbitrarily and eliminate it. If n is odd and the number of remaining chips is even, then we add the odd chip that was not paired, otherwise we discard it.

Let R be the set of remaining chips. We claim that $|R| \leq \lceil \frac{n}{2} \rceil$, and furthermore that more than half of the chips in R are good, i.e. this process reduces the original problem to one of half the size.

Proof. It is trivial to see that at most $\lceil \frac{n}{2} \rceil$ chips survive this elimination process because from every pair at least one chip is eliminated.

Every pair in B must contain at least one bad chip, hence after we eliminate the chips in bad pairs, there are still more good chips than bad. In every pair of G, either both chips are good or both are bad. Let y_g be the number of good pairs in G and y_b the number of bad pairs. Since the number of good chips is greater than the number of bad, we must have $y_g \geq y_b$, and furthermore, if n is even, then $y_g > y_b$. Thus, if n is even, then the number y_g of good chips in R is greater than the number y_b of bad chips.

If n is odd and the number of pairs is odd, i.e., if $y_g + y_b$ is odd, then we must have $y_g > y_b$, hence R contains more good chips than bad.

Suppose n is odd and $y_g + y_b$ is even, i.e., we add the unpaired chip. If $y_g > y_b$, then $y_g > y_b + 1$ since $y_g + y_b$ is even, and thus the number of good chips in R exceeds the number of bad even if the unpaired chip is bad. On the other hand, if $y_g = y_b$, then the unpaired chip must be good because the number of good chips in the original set excceeds the bad. Therefore, in this case also, the remaining set R contains more good chips (namely, $y_g + 1$) than bad (y_b). Thus, in all cases, more than half of the chips that survive are good and this completes our proof.

□

(c)

We can recursively apply the technique described in part (b) until we end up with only one chip. This chip is guaranteed to be good, hence, we can use it to test all other chips, which requires $n-1$ tests, and identify all the good ones.

Let $T(n)$ be the number of tests we need to find one good chip when initially we have n chips. Since we first do $\lfloor \frac{n}{2} \rfloor$ tests to reduce the size of the problem by half and we then recursively solve it, we can write $T(n)$ as follows:

$$T(n) = T\left(\lceil \frac{n}{2} \rceil\right) + \lfloor \frac{n}{2} \rfloor$$

From the third case of Master Theorem we then get $T(n) = \Theta(n)$. It follows that the total number of tests to identify all good chips is $T(n) + n - 1 = \Theta(n)$.