CMPSC 465 Data Structures & Algorithms
Fall 2021 Antonio Blanca and David Koslicki

HW 4

1. (20 pts.) Problem 1

(a) Similar to binary search, start by examining $A[\lfloor \frac{n}{2} \rfloor]$.

- If $A[\lfloor \frac{n}{2} \rfloor]$ is $\lfloor \frac{n}{2} \rfloor$, then we have a satisfactory index.
- If $A[\lfloor \frac{n}{2} \rfloor] > \lfloor \frac{n}{2} \rfloor$, then no element in the second half of the array can possibly satisfy the condition. To check this, note that all integers in the array are distinct, so each integer in the array is at least one greater than the previous element. Since $A[\lfloor \frac{n}{2} \rfloor]$ is already greater than $\lfloor \frac{n}{2} \rfloor$, all the elements on its right are greater than their indices as well. Thus, the second half can be discarded in this case.
- if $A[\lfloor \frac{n}{2} \rfloor] < \lfloor \frac{n}{2} \rfloor$, then by the same logic no element in the first half of the array can satisfy the condition.

We discard the half of the array that cannot hold an answer and repeat the same check until the satisfactory index has been found or all the elements in the array have been discarded.

(b) At each step we do a single comparison and discard at least half of the remaining array (or terminate), so the running time of this algorithms is given by the recurrence $T(n) = T(n/2) + O(1)$ and hence $T(n) = O(\log n)$ time by the master theorem.

2. (20 pts.) Problem 2

(a)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^2 = \begin{bmatrix} a^2+bc & ab+bd \\ ca+dc & cb+d^2 \end{bmatrix} = \begin{bmatrix} a^2+bc & b(a+d) \\ c(a+d) & bc+d^2 \end{bmatrix}$$

Hence, the 5 multiplications $a^2, d^2, bc, b(a+d)$ and $c(a+d)$ suffice to compute the square.

(b) Two possible answers:

1) We have:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^2 = \begin{bmatrix} A^2+BC & AB+BD \\ CA+DC & CB+D^2 \end{bmatrix} \neq \begin{bmatrix} A^2+BC & B(A+D) \\ C(A+D) & BC+D^2 \end{bmatrix}.$$

Note that matrices don't commute! This means $BC \neq CB$ in general, so we cannot reuse that computation. Also, matrix multiplication only has left-distributivity and right-distributivity. As a result, $B(A+D) = BA+BD \neq AB+BD$ and $C(A+D) = CA+CD \neq CA+DC$. Thus, we end up getting more than 5 subproblems, and the recurrence $T(n) = 5T(n/2) + O(n^2)$ does not make sense.

2) Originally, our problem is to square a matrix. However, to compute $\begin{bmatrix} A & B \\ C & D \end{bmatrix}^2$, there is no way of making all the subproblems the same as our initial problem (squaring a matrix). So, this can't be claimed as a divide-and-conquer approach.

(c) Given two $n \times n$ matrices X and Y, create the $2n \times 2n$ matrix A:

$$A = \begin{bmatrix} 0 & X \\ Y & 0 \end{bmatrix}$$

Now, it suffices to compute A^2 to obtain the result of the matrix multiplication, as its upper left block will contain XY:

$$A^2 = \begin{bmatrix} XY & 0 \\ 0 & YX \end{bmatrix}$$

Let $S(n)$ be the time to square a $n \times n$ matrix. The general matrix multiplication XY can be calculated in time $S(2n)$. Given that $S(n) = O(n^c)$, $S(2n) = O((2n)^c) = O(2^c n^c) = O(n^c)$ since c is a constant as well as 2^c which can be ignored as a coefficient.

Note: matrix A in the solution is not unique. The other possibilities for matrix A are $\begin{bmatrix} 0 & X \\ 0 & Y \end{bmatrix}$ ($A^2 = \begin{bmatrix} 0 & XY \\ 0 & Y^2 \end{bmatrix}$), $\begin{bmatrix} X & Y \\ 0 & 0 \end{bmatrix}$ ($A^2 = \begin{bmatrix} X^2 & XY \\ 0 & 0 \end{bmatrix}$), $\begin{bmatrix} Y & 0 \\ X & 0 \end{bmatrix}$ ($A^2 = \begin{bmatrix} Y^2 & 0 \\ XY & 0 \end{bmatrix}$), $\begin{bmatrix} 0 & 0 \\ Y & X \end{bmatrix}$ ($A^2 = \begin{bmatrix} 0 & 0 \\ XY & X^2 \end{bmatrix}$), and $\begin{bmatrix} 0 & Y \\ X & 0 \end{bmatrix}$ ($A^2 = \begin{bmatrix} YX & 0 \\ 0 & XY \end{bmatrix}$).

3. (20 pts.) Problem 3

It is sufficient to show how to find n in time $O(\log n)$, as we can use binary search on the array $A[0, \cdots, n-1]$ to find any element x in time $O(\log n)$. To find n, query elements $A[0], A[1], A[2], A[4], A[8], \cdots, A[2^i], \cdots$,until you find the first element $A[2^k]$ such that $A[2^k] = \infty$. Then, $2^{k-1} \leq n < 2^k$. We can then do binary search on $A[2^{k-1}, \cdots, 2^k]$ to find the last non-infinite element of A. This takes time $O(\log(2^k - 2^{k-1})) = O(\log n)$.

4. (20 pts.) Problem 4

We will assume $k \leq n$. If this is not the case, then for $k' = 2n + 1 - k$, we are trying to find the k'-th largest element, and $k' \leq n$. But finding the k-th largest element and finding the k-th smallest element are symmetric problems, so with some minor tweaks we can use the same algorithm. So for brevity, we will only give the answer for when $k \leq n$.

Since $k \leq n$, the k-th smallest element can't exist past index k of either array. So we cut off all but the first k elements of both arrays. Now, we just want to find the median of the union of the two arrays. To do this, we check the middle elements of both arrays, and compare them. If, the median of the first list is smaller than the median of the second list, then the median can't be in the left half of the first list or the right half of the second list. If the median of the second list is smaller, the opposite is true. So we can cut these elements off, and then recurse on the resulting arrays.

Pseudocode

```
procedure TwoArraySelection(a[1..n], b[1..n], element rank k)
    Set a := a[1, ···, k], b := b[1, ···, k],
    Let ℓ1 = ⌊k/2⌋ and ℓ2 = ⌈k/2⌉
    while a[ℓ1] ≠ b[ℓ2] and k > 1 do
        if a[ℓ1] > b[ℓ2] then
            Set a := a[1, ···, ℓ1]; b := b[ℓ2 + 1, ···, k]; k := ℓ1
            Let ℓ1 = ⌊k/2⌋ and ℓ2 = ⌈k/2⌉
        else
            Set a := a[ℓ1 + 1, ···, k]; b := b[1, ···, ℓ2]; k := ℓ2
            Let ℓ1 = ⌊k/2⌋ and ℓ2 = ⌈k/2⌉
```

```
        end if
    end while
    if k = 1 then
        return min(a[1], b[1])
    else
        return a[ℓ_1]
    end if
end procedure
```

Proof of correctness

Our algorithm starts off by comparing elements $a[\ell_1]$ and $b[\ell_2]$. Suppose $a[\ell_1] > b[\ell_2]$.
Then, in the union of a and b there can be at most $k-2$ elements smaller than $b[\ell_2]$, i.e. $a[1,\cdots,\ell_1-1]$ and $b[1,\cdots,\ell_2-1]$, and we must necessarily have $s_k > b[\ell_2]$. Similarly, all elements $a[1,\cdots,\ell_1]$ and $b[1,\cdots,\ell_2]$ will be smaller than $a[\ell_1+1]$; but these are k elements, so we must have $s_k < a[\ell_1+1]$.
This shows that s_k must be contained in the union of the subarrays $a[1,\cdots,\ell_1]$ and $b[\ell_2+1,\cdots,k]$. In particular, because we discarded ℓ_2 elements smaller than s_k, s_k will be the ℓ_1th smallest element in this union.
We can then find s_k by recursing on this smaller problem. The case for $a[\ell_1] < b[\ell_2]$ is symmetric.
If we reach $k = 1$ before $a[\ell_1] = b[\ell_2]$, we can cut the recursion a little short and just return the minimum element between a and b. You can make the algorithm work without this check, but it might get clunkier to think about the base cases.

Alternatively, if we reach a point where $a[\ell_1] = b[\ell_2]$, then there are exactly k greater elements, so we have $s_k = a[\ell_1] = b[\ell_2]$.

Running time analysis

At every step we halve the number of elements we consider, so the algorithm will terminate in $\log(k)$ recursive calls. Assuming the comparison takes constant time, the algorithm runs in time $\Theta(\log k)$. Note that this does not depend on n.

5. (20 pts.) Problem 5

(a) Suppose that x is the majority element in the original array and suppose first that n is even. Assume, by contradiction, that x is not the majority on either half of the array. If this is the case, the count of x should be $\leq n/4$ in the first half, and $\leq n/4$ in the second half. So overall, x appears at most $n/2$ times which implies that it is not the majority element and this leads to a contradiction. Similarly, if n is odd and x is not the majority in either half, the count of x should be $\leq (n-1)/4$ and $\leq (n+1)/4$ in each half, respectively. So, x appears at most $n/2$ times which is a contradiction.

(b) We scan the array counting how many entries equal x. If the count is more than $n/2$ we declare x to be a majority element. The algorithm scans the array and spends $O(1)$ time per element, so $O(n)$ time overall.

(c) We break the input array A into two halves, recursively call the algorithm on each half, and then test in A if either of the elements returned by the recursive calls is indeed a majority element of A. If that is the case we return the majority element, otherwise, we report that there is "no majority element". To argue the correctness, we see that if there is no majority element of A then the algorithm must return "no majority element" since no matter what the recursive call returns, we always check if an element is indeed a majority element. Otherwise, if there is a majority element x in A we are guaranteed that one of our recursive call will identify x and the subsequent check will lead the algorithm to return x. Regarding time complexity, breaking the main problem into the two subproblems and testing the two

candidate majority elements can be done in $O(n)$ time. Thus we get the following recurrence for the running time:

$$T(n) = \begin{cases} T(n) = 2T(n/2) + O(n), & \text{for } n>1 \\ O(1), & \text{for } n=1 \end{cases} \tag{1}$$

It follows from the master theorem that $T(n) = O(n \log n)$.

(d) After this procedure, there are at most $n/2$ elements left as at least one element in each pair is discarded. Now, let m be the majority element in the initial array. After we pair the elements, there will be pairs of fours types:

(i) We have pairs of the form $\{m,m\}$, where both elements of the pair are the majority element;

(ii) Pairs of the form $\{m,a\}$, where a is some element distinct from m;

(iii) Pairs of the form $\{a,a\}$ for some $a \neq m$; and

(iv) Pairs of the form $\{a,b\}$ for some a and b such that $a \neq m$, $b \neq m$ and $a \neq b$.

Let c_1, c_2, c_3 and c_4 be the number of pairs of each type, respectively. Let's assume first that n is even. Since pairs of the types $\{m,a\}$ and $\{a,b\}$ are discarded, it suffices for us to show that $c_1 > c_3$. Now, $2c_1 + c_2 > n/2$ and $c_1 + c_2 + c_3 + c_4 = n/2$. Combining these two facts, we get $c_1 + (n/2 - c_3 - c_4) > n/2$. This implies that $c_1 > c_3 + c_4$ and that $c_1 > c_3$ as desired. If n is odd, it again suffices to show that $c_1 > c_3$. Let $z = 1$ if the not-paired element is equal to m and 0 otherwise. Now, $c_1 + c_2 + c_3 + c_4 = (n-1)/2$ and $2c_1 + c_2 + z \geq (n+1)/2$. Then, $c_1 + z + ((n-1)/2 - c_3 - c_4) \geq (n+1)/2$ which implies that $c_1 + z \geq c_3 + c_4 + 1$. This implies that $c_1 + z > c_3 + c_4$. Now note that if $c_4 > 0$ or $z = 0$, then we have $c_1 > c_3$. Thus, the only case which remains is the case that $z = 1$ and $c_4 = 0$. In this case we may have $c_1 = c_3$. Remember that we defined A as initial array, and L as the elements left after pairing procedure. Note that so far, we showed if n is even then majority of A is same as majority of L, also if n is odd, and $c_4 > 0$ or $z = 0$, again the majority of A is same as L. Now, let's assume that n is odd, and $c_1 = c_3$, $c_4 = 0$ and $z = 1$. In this case L has c_1 number of m, and c_3 elements different than m, so the size of L would be $c_1 + c_3 = 2c_1$. Note that m would be a majority in L, if there are at least two elements different than m in L. Therefore, the only remaining case is when L has c_1 elements of same value like m, and c_1 elements of another element like a. In this case, we say we have a tie, and we use the unpaired element as the tie breaker. If we keep calling the pairing procedure on this array L, eventually L will be an empty array, and then we return the tie breaker as the majority.

(e) Let $\text{Maj}(A[0,\ldots,n-1], \text{tie-breaker})$ be the function that returns the majority element of A if there is one and if A is empty it returns the tie-breaker, and "no majority element" otherwise. Our implementation of $\text{Maj}(A[0,\ldots,n-1], \text{tie-breaker})$ will work as follows:

1. If $A.length == 1$, output $A[0]$. If $A.length == 0$, output tie-breaker.
2. Otherwise, pair up the elements of A arbitrarily, to get $n/2$ pairs. (If n is odd, update the value of tie-breaker with the unpaired element)
3. Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them. Let L be the resulting set of elements (as an array).
4. Let $m = \text{Maj}(L, \text{tie-breaker})$. If $m =$"no majority element", then there was no majority element in the recursive call, which means can only mean that there was no majority to start with, so we output "error".
5. If $m \neq$"error", we use the method from part (b) to check that m is indeed the majority. If it is, we output m; if not, we output "no majority element".

The correctness of the algorithm follows from parts (b) and (d). The running time is given by the recurrence $T(n) = T(n/2) + O(n)$ which is $T(n) = O(n)$ by the master theorem.

Rubric:

Problem 1, 20 pts

(a) 15 points
6 points: provide an algorithm that achieves what we want
4 points: it's truly faster than the brute force algorithm
5 points: explain why it's correct

(b) 5 points
3 points: provide a running time analysis that makes sense
2 points: reach to a correct conclusion on the running time
Note:

- Some students might analyze the running time correctly, but the given algorithm is not faster than the brute force way. In this case, they'll get full points for part (b) but lose 4 points for part (a).
- Some other students might provide an algorithm that's indeed faster than the brute force approach, but their running time analysis has certain issues. In this case, they'll have the 4 points for part (a) second rubric but lose points for part (b).

Problem 2, 20 pts

(a) 5 points
3 points: correctly expand the squaring of a 2×2 matrix
2 points: correctly identify the 5 needed multiplications

(b) 6 points as long as the student provides either of the possible answers

1) 3 points: correctly expand the squaring of a matrix with 4 sub-matrices and point out the non-commutativity property
3 points: explain how the non-commutativity property introduces an issue for the statement
2) 3 points: point out the sub-problems of squaring a matrix are not necessarily squaring a matrix
3 points: point out it's no longer divide-and-conquer if the main problem and its sub-problems are different

(c) 9 points
3 points: the explanation of squaring a $2n \times 2n$ matrix can also be done in $O(n^c)$ time makes sense
2 points: specify a reasonable $2n \times 2n$ matrix A
4 points: explain how they can obtain the result of a general matrix multiplication from A^2

Problem 3, 20 pts

(a) 5 pts for pointing out that to find any element in a sorted array, the running time is $O(\log n)$

(b) 15 pts for solution how to find the last non-infinite element in the array

Problem 4, 20 pts

(a) 8 pts for designing correct divide and conquer algorithm, and clear description of their algorithm. Note that correct algorithm should give us the kth element in $O(\log k)$, however, if their runtime was $O(\log n)$, they will still get $7 pts$ for this part.

(b) 6 pts for showing the correctness of their algorithm. They need to show that why their algorithm outputs k_{th} element

(c) 6 pts for analyzing the running time. Two ways of doing this are: 1) either showing their algorithm has $\log k$ recursive calls, and each call takes $O(1)$, so the total runtime will be $O(\log k)$. 2) or they can write the recurrent relation ($T(n) = T(\frac{n}{2}) + O(1)$), and derive runtime from it. In this case writing recurrence gets 4 pts, and deriving the final runtime gets 2 pts.

Problem 5, 20 pts

(a) 6 pts for right proof

(b) 5 pts for appropriate explanation for running time $O(n)$

(c) 5 pts for right algorithm and right recurrence

(d) 2 pts for right proof

(e) 2 pts for right algorithm description