1. It was shown in class that the maximum of \( n \) elements can be found in \( O(1) \) time using \( n^2 \) common CRCW PRAM processors.

Consider the case when \( \epsilon = \frac{1}{2} \). Divide the elements into groups of size \( \sqrt{n} \). Assign the first \( \sqrt{n} \) elements to the first \( n \) processors and the second \( \sqrt{n} \) elements to the next \( n \) processors and so on. The maximum element in each group can be found in \( O(1) \) time. At this stage, we have \( \sqrt{n} \) elements and \( n\sqrt{n} \) processors. Hence, the maximum of these elements can be found in \( O(1) \) time. Total time = \( O(1) \).

Next, consider the case when \( \epsilon = \frac{1}{3} \). Here, divide the elements into groups of size \( n^{1/3} \). Assign the first \( n^{1/3} \) elements to the first \( n^{2/3} \) processors and the second \( n^{1/3} \) elements to the next \( n^{2/3} \) processors and so on. The maximum element of each group can be found in \( O(1) \) time and using \( n^{4/3} \) processors the maximum of these maximum elements can be found in \( O(1) \) time.

For the general case, partition the input into groups with \( n^\epsilon \) elements in each group. Find the maximum of each group assigning \( n^{2\epsilon} \) processors to each group. This takes \( O(1) \) time. Now the problem reduces to finding the maximum of \( n^{1-\epsilon} \) elements. Again, partition the elements with \( n^\epsilon \) elements in each group and find the maximum of each group. There will be only \( n^{1-2\epsilon} \) elements left. Proceed in a similar fashion until the number of remaining elements is \( \leq \sqrt{n} \). The maximum of these can be found in \( O(1) \) time. Clearly, the run time of this algorithm is \( O(1/\epsilon) \). This will be a constant if \( \epsilon \) is a constant.

2. Assign \( n^2 \) processors to each of the input keys. Let \( G_i \) be the collection of processors associated with \( k_i \). \( G_i \) identifies all the keys in the input that are greater than \( k_i \) and then finds the minimum of all these keys in \( O(1) \) time. This minimum is the right neighbor of \( k_i \).

For the randomized algorithm use the fact that we can find the minimum of \( n \) keys in \( \tilde{O}(1) \) time using \( n \) common CRCW PRAM processors.

3. Let the input sequence be \( k_1, k_2, \ldots, k_n \). The algorithm consists of \( 2d \) parallel comparison-exchange steps. In particular, in the first parallel step \( k_{2i-1} \) and \( k_{2i} \) are compared and exchanged if they are out of order (for \( 1 \leq i \leq (n/2) \)). In the second parallel step \( k_{2i} \) and \( k_{2i+1} \) are compared and exchanged if they are out of order (for \( 1 \leq i \leq (n/2-1) \)). In parallel step 3, \( k_{2i-1} \) and \( k_{2i} \) are compared and exchanged if they are out of order (for \( 1 \leq i \leq (n/2) \)). In parallel step 4, \( k_{2i} \) and \( k_{2i+1} \) are compared and exchanged if they are out of order (for \( 1 \leq i \leq (n/2-1) \)); and so on.

4. We begin by computing \( 1, w, w^2, \ldots, w^{n-1} \) where \( w \) is the primitive \( n \)th root of unity. This can be done using a prefix computation in \( O(\log n) \) time.
Looking at the details of the sequential FFT algorithm we realize that there are two recursive calls on an input size of \( n/2 \) each. This is because we can think of the FFT problem as that of evaluating the given degree \( n - 1 \) polynomial \( f(x) \) at the \( n \)th roots of unity. This problem is reduced to two subproblems, where each subproblem is that of evaluating a degree-\( n/2 - 1 \) polynomial at the \( n/2 \)nd roots of unity. Let these two polynomials be \( A(x) \) and \( B(x) \). We can do these evaluations in parallel by assigning \( n/2 \) processors to each subproblem. Once \( A(x) \) and \( B(x) \) have been evaluated at the \( n/2 \)nd roots of unity, it takes an additional \( O(n) \) time to compute the FFT of interest. Given the \( n \)th roots of unity, it is easy to see that these additional steps can be completed in \( O(1) \) time using \( n \) processors.

Thus, if \( T(n) \) is the run time of the algorithm (excluding the computation of the \( n \)th roots of unity), then we have: \( T(n) = T(n/2) + O(1) \) which solves to: \( T(n) = O(\log n) \).

5. We can use an array \( a[1 : n] \) to solve this problem. At the beginning processor 1 sets \( \text{ThereAreRepeatedElements} := 0 \). Assume that we have \( n \) arbitrary CRCW PRAM processors. Let the input sequence be \( k_1, k_2, \ldots, k_n \). We assign one key per processor. In one parallel write step, for \( 1 \leq i \leq n \), processor \( i \) tries to write \( i \) in \( a[k_i] \). In the next step processor \( i \) reads from \( a[k_i] \). If it does not find \( i \) there it tries to write a 1 in \( \text{ThereAreRepeatedElements} \). At the end of this step, we have the result in \( \text{ThereAreRepeatedElements} \).

6. The essence of Strassen’s algorithm is the fact that two \( 2 \times 2 \) scalar matrices can be multiplied using 7 multiplications and 18 additions. One applies the same formulas to multiply two given \( n \times n \) matrices by partitioning each matrix into four \( n/2 \times n/2 \) submatrices. The same technique can be applied in parallel as well. The seven submatrix multiplications can be done recursively in parallel using enough processors. Once these multiplications have been completed, the 18 submatrix additions can be completed in \( O(1) \) time given \( O(n^2) \) processors.

If \( T(n) \) is the parallel time needed to multiply two \( n \times n \) matrices, then we have: \( T(n) = T(n/2) + O(1) \) which solves to: \( T(n) = O(\log n) \).

If \( P(n) \) is the processor bound, then we have: \( P(n) = 7P(n/2) + O(n^2) \) which solves to: \( P(n) = O(n^{\log_2 7}) \).