Problem 1. (25 points)

Suppose you have $k$ sorted arrays, each with $n$ elements, and you want to combine them into a single sorted array of $kn$ elements. Describe a divide-and-conquer algorithm that takes $O(kn \log k)$ time. Explain carefully why your algorithm takes $O(kn \log k)$ time.

**Answer:** Here is the divide-and-conquer algorithm

1. Partition the set of $k$ arrays into $k/2$ pairs
2. Merge each pair of arrays using the linear-time merge procedure from MergeSort
3. Recurse on the remaining set composed of $k/2$ arrays

The recursion is $T(k) = 2T(k/2) + k\Theta(n)$ which has solution $O(nk \log k)$ using Master Thm case II. This is essentially doing the top $k$ levels of the recursion tree from mergesort on a partially sorted $kn$-element array.

Problem 2. (25 points)

A CS 141 student has been trying to speed-up Strassen’s divide-and-conquer for matrix multiplication algorithm. Recall that Strassen’s algorithm computes the product of two $n \times n$ matrices in $O(n^{2.81})$, and uses the fact that one can multiply two $n/2 \times n/2$ matrices with only 7 multiplications instead of 8 with the naive matrix-multiplication algorithm. Suppose the student came up with a variant of Strassen’s algorithm based on the fact that the product of two $n/3 \times n/3$ can be found with only $m$ multiplication instead of the normal 27. How small would $m$ have to be for this algorithm to be asymptotically faster than Strassen’s algorithm covered in class? You can assume $m > 9$. Justify your answer.

**Answer:** We have $T(n) = mT(n/3) + dn^2$. We are in case one of the Master theorem $dn^2 \in O(n^{\log_3 m - \epsilon})$ because $m > 9$, and therefore we can choose $\epsilon = \log_3 m - 2$ and it will be positive. In order for the new algorithm to be faster, we would need to have $\log_3 m < 2.81$, that is $m < 3^{2.81} \approx 22$. 

Problem 3. (25 points)
Give a divide-and-conquer algorithm for multiplying two polynomials of degree \(n\) in time \(O(n^{\log_2 3})\). This algorithm is very similar to Karatsuba’s integer multiplication algorithm we covered in class.

Answer: Suppose the two polynomials we want to multiply are \(A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}\) and \(B(x) = b_0 + b_1x + b_2x^2 + \ldots + b_{n-1}x^{n-1}\). We assume that \(n\) is a power of two (otherwise, we can always pad the coefficients with zeros to reach the “next” power of two). Let us break \(A(x)\) and \(B(x)\) into two polynomials as follows.

\[
A(x) = A_0(x) + x^{n/2}A_1(x) \\
B(x) = B_0(x) + x^{n/2}B_1(x)
\]

where

\[
A_0(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n/2-1}x^{n/2-1} \\
A_1(x) = a_{n/2} + a_{n/2+1}x + a_{n/2+2}x^2 + \ldots + a_{n-1}x^{n/2-1} \\
B_0(x) = b_0 + b_1x + b_2x^2 + \ldots + b_{n/2-1}x^{n/2-1} \\
B_1(x) = b_{n/2} + b_{n/2+1}x + b_{n/2+2}x^2 + \ldots + b_{n-1}x^{n/2-1}
\]

Then the problem of multiplying \(A(x)B(x)\) can be decomposed in the problem of multiplying \(A_0(x), A_1(x), B_0(x), B_1(x)\) as follows. We omit “\(x\)” to reduce the clutter.

\[
AB = (A_0 + x^{n/2}A_1)(B_0 + x^{n/2}B_1) \\
= A_0B_0 + x^{n/2}(A_0B_1 + A_1B_0) + x^nA_1B_1 \\
= A_0B_0 + x^{n/2}((A_0 - A_1)(B_1 - B_0) + A_0B_0 + A_1B_1) + x^nA_1B_1
\]

Therefore, we need 3 multiplications of two polynomials of degree \(n/2\) (namely, \(A_0B_0, A_1B_1\) and \((A_0 - A_1)(B_1 - B_0)\)) and \(O(n)\) additional work for the sum and the differences.

The recurrence relations is

\[
T(n) = \begin{cases} 
\Theta(1) & n = 1 \\
3T(n/2) + O(n) & n > 1
\end{cases}
\]

which can be solved using the Master Theorem, concluding that \(T(n) \in O(n^{\log_2 3})\).

Problem 4. (25 points)
An array \(A\) is said to have a majority element if more than half of the entries in \(A\) are exactly the same. Describe an \(O(n \log n)\) divide-and-conquer algorithm that determines whether an array \(A\) of \(n\) items has a majority element, and if so, returns that item. The only comparison operation allowed on the items is equality. That is, your algorithm can determine whether “\(A[i] == A[j]\)” or not in \(O(1)\) time, but it cannot, for example, compare the items to sort them, or hash the items into buckets. Explain why your algorithm takes \(O(n \log n)\) time.
**Answer 1:** Split $A$ into $A_1 = A[1..\lfloor n/2 \rfloor]$ and $A_2 = A[\lceil n/2 \rceil + 1..n]$, then recursively find the majority $m_1$ of $A_1$ (if any) and the majority element $m_2$ of $A_2$ (if any).

(If $A$ has a majority element $m$, then at least one of $A_1$ or $A_2$ will also have to have $m$ as the majority element, so $m \in \{m_1, m_2\}$. Note that this is true whether or not $n$ is even or odd.)

Once $m_1$ and $m_2$ are determined, scan $A$ in linear time and count the occurrences of each to see whether it is a majority element.

The running time $T(n)$ on $n$ elements satisfies the recurrence $T(n) \leq 2T(n/2) + O(n)$, which (as for mergesort) gives $T(n) = O(n \log n)$.

**Answer 2:** Pair up the elements of the array (say, pair each even element $A[2i]$ with its odd neighbor $A[2i + 1]$), then, in each pair, if the two elements are the same, discard one of them, and otherwise, discard neither. Then, recursively find the majority element among the remaining elements.

(Note that this can be done only if number of elements is even — otherwise you can’t pair them. We consider this issue further later.)

Next we prove that if we do the above process, and the original array had a majority element $m$, then the elements remaining after the pairing/discarding process will also have majority element $m$.

To see this, consider the discarding process in two steps:

1. discard the differing pairs
2. from each of the remaining pairs, discard one element

Since step 1 discards at least one non-majority element in each pair, it discards at least as many non-majority elements as majority elements. Thus, among the elements remaining after step 1, the majority must still be $m$.

Because of this, in step 2, the number (say $p$) of pairs that contain two $m$’s must exceed the number (say $q$) of pairs that contain non-$m$’s. (Otherwise there would not be more of the majority element than other elements.) That is, $p > q$.

Since one majority element remains from each majority pair, and one non-majority element remains from each other pair, the number of elements remaining is $p + q$, among which $p$ equal $m$.

Since $p > q$, it must be that more than half of the elements remaining equal $m$.

This proves that the majority element after the pairing/discarding process will be the same as the majority element before the process (if any).

Now, how do we deal with the case when there are an odd number of elements? If $n$ is odd, simply check the whether the particular element $A[n]$ is the majority element, by scanning through the array in $O(n)$ time and counting its occurrences. If it is, you are done. Otherwise, discard $A[n]$ (reduce $n$ by 1), and proceed as above for $n$ even.

Then the running time $T(n)$ satisfies $T(n) \leq O(n) + T(n/2)$, giving $T(n) = O(n \log n)$. 