Problem 1. [Analysis] (17 points)

Use the Master method (after an appropriate substitution) to give an asymptotic tight bound for $T(n)$ defined by the following recurrence relation

$$T(n) = \begin{cases} 
2 & n = 2 \\
3T\left(n^{1/3}\right) + (\log n)(\log \log n) & n > 2
\end{cases}$$

Answer: Let $n = 2^k$ (that is, log $n = k$). Then

$$T(n) = 3T\left(n^{1/3}\right) + (\log n)(\log \log n)$$

$$T(2^k) = 3T\left(2^{k/3}\right) + k \log k$$

Let $S(k) = T(2^k)$. We have

$$S(k) = 3S(k/3) + k \log k$$

We can apply case 2 of the Master Theorem. In fact,

$$k \log k \in \Theta\left(k^{\log_2 3} \log^t k\right)$$

for $t = 1$. Therefore $S(k) \in \Theta\left(k \log^2 k\right)$.

Hence, $T(2^k) \in \Theta\left(k \log^2 k\right)$, which implies that $T(n) \in \Theta\left((\log n)(\log^2 \log n)\right)$.

Problem 2. [Amortized Analysis] (17 points)

A sequence of Push and Pop operations is performed on a stack whose size never exceeds a given value $k$. After every $k$ operations, an operation Copy is automatically invoked which copies the entire stack for backup purposes (including the empty slots). In other words, Copy always costs $O(k)$ irrespective of the number of items in the stack. Show that the cost of a sequence of $n$ operations Push, Pop, Copy, is $O(n)$ by assigning suitable amortized costs to Push and Pop.

An answer: Charge $2$ for each Push and Pop operations and $0$ for each Copy. When we call Push, we use $1$ to pay for the operation, and we store the other $1$ on the item pushed. When we call Pop, we again use $1$ to pay for the operation, and we store the other $1$ in the stack itself. Because the stack size never exceeds $k$, the actual cost of a Copy operation is at most $\$k$, which is paid by the $\$k$ found in the items in the stack and the stack itself. Since there are $k$ Push and Pop operations between two consecutive Copy operations, there are $\$k$ of credit stored, either on individual items (from Push operations) or in the stack itself (from Pop operations) by the times a Copy occurs. Since the amortized cost of each operation is $O(1)$ and the amount of credit never goes negative, the total cost of $n$ operations is $O(n)$.

Problem 3. [Divide and Conquer] (17 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.

An answer: Split A into A₁ = A[1..[n/2]] and A₂ = A[[n/2] + 1..n], then recursively find the majority m₁ of A₁ (if any) and the majority element m₂ of A₂ (if any).

(If A has a majority element m, then at least one of A₁ or A₂ will also have to have m as the majority element, so m ∈ {m₁, m₂}. Note that this is true whether or not n is even or odd.)

Once m₁ and m₂ 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) ≤ 2T(n/2) + O(n), which (as for mergesort) gives T(n) = O(n log n).

Problem 4.
Write the pseudo-code of the direct (recursive) FFT and analyze its time complexity as a function of the input size n. You do not need to prove its correctness.

Answer: See slides.

Problem 5. (17 points)
Assume that you are given two unsorted arrays A = {a₁, a₂, . . . , aₙ} and B = {b₁, b₂, . . . , bₙ} of n positive integers. We want to determine an ordering of the elements of A and B such that

\[ W = \prod_{i=1}^{n} a_i^{b_i} \]

is maximized. Consider the following greedy algorithm.

Algorithm Greedy (A, B); sort A and B in decreasing order; return (A, B)

Show that the greedy-choice property holds for the algorithm Greedy (no need to prove that the problem has the optimal substructure property).

An answer: The usual exchange argument. Consider any indices i and j such that i < j, and consider the terms aᵢᵇᵢ and aⱼᵇⱼ. We want to show that the objective function W will not get worse by taking aᵢᵇᵢ and aⱼᵇⱼ instead. In other words, we need to show that aᵢᵇᵢaⱼᵇⱼ ≥ aⱼᵇⱼaᵢᵇᵢ. Since A and B are sorted in decreasing order and i < j we have aᵢ ≥ aⱼ and bᵢ ≥ bⱼ. Since aᵢ and aⱼ are positive and bᵢ − bⱼ is nonnegative, we have aᵢᵇᵢaⱼᵇⱼ ≥ aⱼᵇⱼaᵢᵇᵢ. Multiplying both sides by aᵢᵇᵢaⱼᵇⱼ yields aᵢᵇᵢaⱼᵇⱼ ≥ aᵢᵇᵢaⱼᵇⱼ.

Problem 6. (15 points)
Consider an implementation of union-find that uses union by rank and path compression. Draw the tree(s) that result at the end of each of the following set of operations, and indicate the rank of each node using subscripts.

MAKESET(a); MAKESET(b); MAKESET(c); MAKESET(d); MAKESET(e); MAKESET(f); MAKESET(g); MAKESET(h);

Union(a,b); Union(c,d); Union(e,f);
Union(b,g); Union(d,h);

Union(g,f);

Union(f,h);