1. **Mastering the Master Theorem.** Use the “Master Theorem,” as discussed in class, to solve the following recurrences:

   (a) \( T(n) = 2T(n/4) + \sqrt{n} \)
   (b) \( T(n) = 2T(n/4) + n^{0.51} \)
   (c) \( T(n) = 3T(n/3) + n/2 \)
   (d) \( T(n) = 16T(n/4) + n^{1.5} \)
   (e) \( T(n) = 7T(n/2) + n^2 \)

2. **Weird recursion tree analysis.** Suppose we have an algorithm that on problems of size \( n \), recursively solves two problems of size \( n/2 \), with a “local running time” bounded by \( t(n) \) for some function \( t(n) \). That is, the algorithm’s total running time \( T(n) \) satisfies the recurrence relation \( T(n) \leq 2T(n/2) + t(n) \). For simplicity, assume that \( n \) is a power of 2. Also, assume that \( n \geq 2 \) and the recursion stops when \( n = 2 \).

   Prove the following using a recursion tree analysis, similar to the one that was used to prove the Master Theorem in class.

   (a) If \( t(n) = O(n \log n) \), then \( T(n) = O(n(\log n)^2) \).
   (b) If \( t(n) = O(n/ \log n) \), then \( T(n) = O(n \log \log n) \).

   Note that the Master Theorem itself cannot be used here, because the functions \( t(n) \) above are not of the form required by the Master Theorem.

   **Hint:** for each level \( j \), estimate the number of subproblems at that level, and the local running time for each subproblem at that level. Also, note that for part (a), \( t(n) = O(n \log n) \) means \( t(n) \leq c \cdot n \log_2 n \) for some constant \( c \) and all \( n \geq 2 \). Similarly, for part (b), \( t(n) = O(n/ \log n) \) means \( t(n) \leq c \cdot n/ \log_2 n \) for some constant \( c \) and all \( n \geq 2 \).

3. **Uneven divide and conquer (1).** Suppose we have an algorithm that on problems of size \( n \), recursively solves two subproblems, one of size \([0.7n]\) and one of size \([0.2n]\). Furthermore, the “local running time” is \( O(n) \). That is, the algorithm’s total running time satisfies the inequality

   \[ T(n) \leq T([0.7n]) + T([0.2n]) + cn \]

   for some constant \( c \). You may assume the above inequality holds for all \( n \geq 1 \), and that \( T(0) = 0 \).

   Prove that \( T(n) = O(n) \) using a recursion tree analysis.

   **Hint:** Let \( S_j \) be the sum of all subproblem sizes at level \( j \). Show that \( S_{j+1} \leq 0.9S_j \) for all \( j \geq 0 \).

4. **Uneven divide and conquer (2).** Suppose we have an algorithm that on problems of size \( n \), recursively solves two subproblems, one of size \([0.9n]\) and one of size \([0.1n]\). Furthermore, the “local running time” is \( O(n^2) \). That is, the algorithm’s total running time satisfies the inequality

   \[ T(n) \leq T([0.9n]) + T([0.1n]) + cn^2 \]

   for some constant \( c \). You may assume the above inequality holds for all \( n \geq 1 \), and that \( T(0) = 0 \).

   Prove that \( T(n) = O(n^2) \) using a recursion tree analysis.

   **Hint:** Let \( S_j \) be the sum of the **squares** of the subproblem sizes at level \( j \). Show that \( S_{j+1} \leq 0.82S_j \) for all \( j \geq 0 \).

5. **Product trees.** For this and the following two exercises, we assume that \( F \) is a field and that we can compute the product of two polynomials in \( F[X] \) whose degrees are less than \( n \) using \( O(n^\alpha) \) operations in \( F \), where \( 1 < \alpha < 2 \) is a constant. You may also **freely use** the “Master Theorem” from class in these exercises.

   For this problem, the input is a list of field elements \( a_1, \ldots, a_n \in F \). For simplicity, assume that \( n \) is a power of 2. The output is a complete binary tree with \( n \) leaves, where each node in the tree stores a polynomial which is of the form \( P_s^{(t)} := \prod_{i=s}^{s+t-1} (X - a_i) \):

   - the root stores \( P_1^{(n)} \);
• for every internal node in the tree, if that node stores the polynomial $P_s^{(2t)}$, its left child stores $P_s^{(t)}$ and its right child stores $P_s^{(t+1)}$.

Note that the $n$ leaves of the tree store the linear polynomials $(X - a_1), \ldots, (X - a_n)$.

Example: on input $a_1, a_2, a_3, a_4$, the tree would look like this:

```
   (X - a_1)(X - a_2)(X - a_3)(X - a_4)
     /  \
(X - a_1)(X - a_2)    (X - a_3)(X - a_4)
      / \\    /  \n  X - a_1  X - a_2  X - a_3  X - a_4
```

Using the given polynomial multiplication algorithm as a subroutine, design a recursive algorithm to solve this problem and analyze its running time. In particular, show that the running time of your algorithm is $O(n^3)$ operations in $F$.

The recursive algorithm you design should just use the given polynomial multiplication algorithm as a “black box” — you do not need to worry about how it is implemented.

Notes: In this and the following two exercises, polynomials are always represented by their coefficient vectors. That is, if the exercise says a polynomial is input, output, or stored, what it means is that the coefficient vector of the polynomial is input, output, or stored. So in the above example with $n = 4$, the node that stores the polynomial $(X - a_1)(X - a_2)$ really stores the coefficient vector $(c_0, c_1, c_2)$, where $c_0 = a_1a_2$, $c_1 = -(a_1 + a_2)$, and $c_2 = 1$, which is the coefficient vector of the polynomial $(X - a_1)(X - a_2) = X^2 - (a_1 + a_2)X + a_1a_2$.

6. Multi-point evaluation. For this problem, the input is a list of field elements $a_1, \ldots, a_n \in F$ and a polynomial $f \in F[X]$ of degree less than $n$. For simplicity, assume that $n$ is a power of 2. The output is $f(a_1), \ldots, f(a_n)$. Show how to solve this problem with a recursive algorithm that uses $O(n^3)$ operations in $F$.

Note that unlike the FFT, we do not assume anything special about the evaluation points $a_1, \ldots, a_n$.

Use the algorithm from previous exercise as a preprocessing step.

Also, you may use as a subroutine an algorithm for polynomial division: on input $g, h \in F[X]$ where $h \neq 0$ and both $g$ and $h$ have degree less than $n$, you may assume this algorithm computes the quotient $q \in F[X]$ and remainder $r = g \mod h \in F[X]$ satisfying $g =hq + r$ and $\deg(r) < \deg(h)$ using $O(n^3)$ operations in $F$.

The recursive algorithm you design should just use the given division algorithm as a “black box” — you do not need to worry about how it is implemented.

Hint: use the following general fact. Suppose $b_1, \ldots, b_k \in F$. Let $g = (X - b_1) \cdots (X - b_k)$. Suppose we divide $f$ by $g$, obtaining a quotient $q$ and remainder $r$, so that $f = gq + r$. Then we have $f(b_i) = r(b_i)$ for $i = 1, \ldots, k$. This is easy to see: just plug $b_i$ into $f = gq + r$, and you see that $f(b_i) = g(b_i)q(b_i) + r(b_i) = 0 \cdot q(b_i) + r(b_i)$.

7. Polynomial interpolation.

(a) For this problem, the input consists of two lists of field elements: $a_1, \ldots, a_n \in F$ and $c_1, \ldots, c_n \in F$. For simplicity, assume that $n$ is a power of 2. Define the polynomial $P := \prod_{i=1}^n (X - a_i)$, and for $i = 1, \ldots, n$, define the polynomial $P_i^* := P/(X - a_i)$. That is,

$$P_i^* = \prod_{1 \leq j \leq n}^{j \neq i} (X - a_j) = (X - a_1) \cdots (X - a_{i-1}) (X - a_{i+1}) \cdots (X - a_n).$$

The output is the polynomial $\sum_{i=1}^n c_i P_i^*$.

\[\text{It is a general fact that we can perform polynomial division just as fast as polynomial multiplication, up to constant factors.}\]
Example: on inputs $a_1, a_2, a_3, a_4$ and $c_1, c_2, c_3, c_4$, the output should be the polynomial

$$
c_1(X - a_2)(X - a_3)(X - a_4) + c_2(X - a_1)(X - a_3)(X - a_4) +
\quad c_3(X - a_1)(X - a_2)(X - a_4) + c_4(X - a_1)(X - a_2)(X - a_3).
$$

Show how to solve this problem with a recursive algorithm that uses $O(n^\alpha)$ operations in $F$.

The recursive algorithm you design should just use the given polynomial multiplication algorithm as a “black box” — you do not need to worry about how it is implemented.

**Hint:** use the algorithm from Exercise 5 as a preprocessing step. Also, just think about what happens if you solve the problem recursively on two inputs: the first consisting of the two lists $a_1, \ldots, a_{n/2}$ and $c_1, \ldots, c_{n/2}$, and the second consisting of the two lists $a_{n/2+1}, \ldots, a_n$ and $c_{n/2+1}, \ldots, c_n$. How can you combine the results of those two recursive invocations (using a couple of polynomial multiplications and additions) to compute the result for the original problem.

(b) Use part (a) and the algorithm from Exercise 6 to implement Lagrange interpolation using $O(n^\alpha)$ operations in $F$. The input is a list $(a_1, b_1), \ldots, (a_n, b_n)$, where each $a_i$ and $b_i$ is in $F$, and $a_i \neq a_j$ for all $i \neq j$, and the output is the unique polynomial $f \in F[X]$ of degree less than $n$ that satisfies $f(a_i) = b_i$ for $i = 1, \ldots, n$. For simplicity, assume that $n$ is a power of 2.

**Hint:** first run the algorithm from part (a) with $c_i = 1$ for $i = 1, \ldots, n$ to get a polynomial $Q$ that will help you compute the values $P^*_i(a_i)$ for $i = 1, \ldots, n$, which arise in the Lagrange interpolation formula.