1. **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" of $O(f(n))$ for some function $f(n)$. That is, the algorithm’s total running time satisfies the recurrence $T(n) \leq 2T(n/2) + O(f(n))$. For simplicity, assume that $n$ is a power of 2.

Prove the following using a recursion tree analysis:

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

2. **Uneven divide and conquer.** The following recurrence relation appears in divide-and-conquer algorithms in which the problem is divided into unequal size parts:

$$T(n) \leq \sum_{i=1}^{k} a_i T([n/b_i]) + cn.$$  

Here, the $a_i$’s and $b_i$’s, as well as $c$, are positive integer constants. You may assume the above recursively defined bound for $T(n)$ holds for $n \geq \max_i b_i$, and that $T(n)$ is bounded by a constant $t$ for $n < \max_i b_i$.

The goal of this exercise is to prove that $T(n) = O(n)$, assuming $\delta := \sum_{i=1}^{k} a_i/b_i < 1$. To this end, instead of using a recursion tree analysis, try using a proof by induction. To do this, introduce a constant $d$ (to be determined by your analysis), and prove by (strong) induction that $T(n) \leq dn$ for all positive integers $n$. You should be able to identify exactly where in the proof you use the assumption that $\delta < 1$.

3. **Roots of unity.** Let $R$ be a ring and assume $n = 2^k > 1$ and that $2 \in R^\star$. In class, we defined $\omega \in R$ to be a primitive $n$th root of unity if $\omega^{n/2} = -1$. This exercise asks you to prove some properties that were stated without proof in class. Throughout, assume that $\omega \in R$ is a primitive $n$th root of unity.

(a) Show that if $n > 2$, then $\omega^2$ is a primitive $(n/2)$th root of unity.
(b) Show that if $i$ is an odd integer, then $\omega^i$ is a primitive $n$th root of unity.
(c) Show that $\omega - 1 \in R^\star$. What is $(\omega - 1)^{-1}$?
(d) Show that $\omega^i - 1 \in R^\star$ for all $i \not\equiv 0 \pmod{n}$. Hint: this follows rather easily from parts (a)–(c).

4. **Product trees.** For this and the remaining exercises, we assume that $R$ is a ring and that we can compute the product of two polynomials in $R[X]$ whose degrees are less than $\ell$ using $M(\ell)$ operations in $R$.

For this problem, the input is a list of ring elements $a_1, \ldots, a_n \in R$. 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 $P_s^\ell := \prod_{i=s}^{s+\ell-1}(X - a_i)$:

- the root stores $P_1^n$;
- for every internal node in the tree, if that node stores the polynomial $P_s^{2\ell}$, its left child stores $P_s^\ell$ and its right child stores $P_{s+\ell}^\ell$.

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

Using the given polynomial multiplication algorithm as a subroutine, design a recursive algorithm to solve this problem and analyze its running time $T(n)$. In particular, show that:

$$T(n) = \begin{cases} 
\text{if } M(\ell) = O(\ell^\alpha) \text{ for constant } \alpha \text{ with } 1 < \alpha \leq 2, \text{ then } T(n) = O(n^\alpha); \\
\text{if } M(\ell) = O(\ell \log \ell), \text{ then } T(n) = O(n(\log n)^2). 
\end{cases}$$

(1)

Note: this problem is actually very easy using results from class and from Exercise 1.
5. **Multi-point evaluation.** For this problem, the input is a list of ring elements $a_1, \ldots, a_n \in R$ and a polynomial $f \in R[X]$ of degree less than $n$. For simplicity, assume that $n$ is a power of 2. The output is $f(a_1), \ldots, f(a_n)$.

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

Using the algorithm from previous exercise, and the fact that polynomial division is no harder than multiplication (up to constant factors), show how to solve this problem recursively in time $T(n)$ satisfying (1).

**Hint:** use the fact that for $a \in R$, we have $f(a) = f \mod (X - a)$.

6. **Polynomial interpolation (I).** For this problem, the input consists of two lists of ring elements: $a_1, \ldots, a_n \in R$ and $c_1, \ldots, c_n \in R$. 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 $P_i^* := P/(X - a_i)$. The output is the polynomial $\sum_{i=1}^{n} c_i P_i^*$.

Using results from class and previous exercises, show how to solve this problem recursively in time $T(n)$ satisfying (1).

7. **Polynomial interpolation (II).** The problem in the previous exercise is related to Newton’s interpolation formula. If we are given the list $((a_1, b_1), \ldots, (a_n, b_n))$, where each $a_i$ and $b_i$ is in $R$, and $a_i - a_j \in R^*$ for all $i \neq j$, then the unique polynomial $f \in R[X]$ of degree less than $n$ that satisfies $f(a_i) = b_i$ for $i = 1, \ldots, n$ is given by the formula $f = \sum_{i=1}^{n} c_i P_i^*$, where $c_i = b_i / P_i^*(a_i)$.

So to solve the interpolation problem, it suffices to solve the following problem. The input is a list of ring elements $a_1, \ldots, a_n \in R$, where as usual we assume that $n$ is a power of 2 to simplify things. The output is the list $P_1^*(a_1), \ldots, P_n^*(a_n)$.

Show how to solve this problem in time $T(n)$ satisfying (1).