Problem 1. A polynomial is a function $C(x) = \sum_{i=0}^{n} c_i x^i$, where $x$ is the variable and $c_0, \ldots, c_n$ are the constant coefficients. The only information we need to define a polynomial is this sequence of coefficients. As such, we usual represent polynomials in algorithms as coefficient arrays. The coefficients are stored so that $c_i$ is stored at index $i$.

The product of two polynomials $A, B$ (of the same variable) is

$$(AB)(x) = \sum_{i=0}^{n} \sum_{j=0}^{n} a_i b_j x^{i+j}.$$ 

At a glance, an algorithm to compute $AB$ is straightforward. In Algorithm 1, we multiply two degree $n$ polynomials together. To multiply two different degree polynomials together, we can represent the smaller polynomial as a larger degree polynomial by padding its coefficient array with 0’s.

\begin{algorithm}
\begin{algorithmic}
  \STATE \textbf{Input:} Polynomials $A, B$ of degree $n$
  \STATE Let $C$ be a polynomial of degree $2n$ initialized to all 0 coefficients
  \FOR{$i = 0$ \TO $n$}
    \FOR{$j = 0$ \TO $n$}
      \STATE Add $a_i b_j$ to $c_{i+j}$
    \ENDFOR
  \ENDFOR
  \STATE \textbf{return} $C$
\end{algorithmic}
\end{algorithm}

a) What is the runtime of this multiplication algorithm?

b) Adapt a divide and conquer algorithm of similar purpose to improve the runtime.

c) State precisely what your algorithm returns.

d) Prove your algorithm is correct (that is it returns what you claimed it should).

e) Give a recurrence relation for your algorithm.

f) Determine the runtime of your algorithm using a work tree to justify it.
Problem 2. The discrete Fourier transform (DFT) of a function \( f : \mathbb{Z}_n \rightarrow \mathbb{C} \), where \( \mathbb{C} \) is the set of complex numbers, is given by

\[
\mathcal{F}(f)(k) = \sum_{j=0}^{n-1} f(j) e^{-\frac{2\pi i}{n} jk}.
\]

This transformation is used just about everywhere, even in places you would not expect it, though we will omit a lengthy discussion of its utility here.

Notice that the output of the Fourier transform \( \mathcal{F}(f) \) is itself a function from \( \mathbb{Z}_n \) to \( \mathbb{C} \). If we want to compute all of them, the naive approach is to compute \( \mathcal{F}(f)(k) \) directly for each value of \( k \) in \( O(n^2) \) time. However, \( e^{-\frac{2\pi i}{n} jk} \) happens to traverse a circle of radius one in the complex plane and thus has a number of symmetries we can take advantage of.

The fast Fourier transform (FFT) is the go-to algorithm we use for computing each of \( \mathcal{F}(f)(0), \ldots, \mathcal{F}(f)(n-1) \) simultaneously in \( O(n \log n) \) time (it returns an array/function/vector/etc containing these values). The key insight is the observation that

\[
\mathcal{F}(f)(k) = \sum_{j=0}^{n-1} f(2j) e^{-\frac{2\pi i}{n} jk} + e^{-\frac{2\pi i}{n} k} \sum_{j=0}^{n-1} f(2j+1) e^{-\frac{2\pi i}{n} jk} = \mathcal{F}(f_o)(k) + e^{-\frac{2\pi i}{n} k} \mathcal{F}(f_e)(k)
\]

We obtain this by separating the even and odd index terms into two smaller DFTs of size \( \frac{n}{2} \). These smaller functions are denoted \( f_e \) and \( f_o \) respectively above. This division into half size problems lends itself to divide and conquer algorithms. However, there is an issue here. The smaller DFTs are not properly defined for \( k \geq \frac{n}{2} \). Luckily, symmetry saves the day. When \( n \) is even,

\[
\mathcal{F}(f)\left(k + \frac{n}{2}\right) = \mathcal{F}(f_e)(k) - e^{-\frac{2\pi i k}{n}} \mathcal{F}(f_o)(k).
\]

For the purposes of this problem, you may freely assume \( n \) is a power of two. There are a number of ways to mitigate this assumption in application, so it is largely an unimportant restriction.

a) Give an efficient divide and conquer algorithm for the FFT.

b) Show that the runtime of your algorithm is \( O(n \log n) \).