Problem 1 (KT 5.2)

Recall the problem of finding the number of inversions. As in the text, we are given a sequence of \( n \) numbers \( a_1, \ldots, a_n \), which we assume are all distinct, and we define an inversion to be a pair \( i < j \) such that \( a_i > a_j \).

We motivated the problem as a good measure of how different two orderings are. However, one might feel that this measure is too sensitive. Let’s call a pair a significant inversion if \( i < j \) and \( a_i > 2a_j \). Give an \( O(n \log n) \) algorithm to count the number of significant inversions between two orderings.

Problem 2

In this problem you are asked to give concrete bounds on the running time of algorithms, measured as the number of arithmetic operations performed. You do not need to determine the exact number of operations, any reasonable upper bound is fine.

Section 5.5 of the textbook presents a divide and conquer algorithm to multiply two \( n \)-digit integer numbers in time \( \approx O(n^{1.59}) \). In the textbook, numbers are written in binary, but to make the example more realistic, here we write numbers in base \( B = 256 \), so that each digit is represented by a byte, and assume that arithmetic operations on bytes can be performed in unit time. (On input two bytes \( x, y \), addition and multiplication produce two output bytes \( w, z \) such that \( w \cdot B + z \) equals \( x + y \) or \( x \cdot y \).) For simplicity, we will count only the number of arithmetic operations to measure the complexity of an algorithm. You can also assume that both input numbers \( a, b \) have the same length of \( n \) bytes, and that \( n \) is a power of 2, or has any other form that you find convenient for the analysis.

part (a) Give an upper bound on the number of arithmetic operations performed by the algorithm in Section 5.5. You know from Section 5.5 this should be an expression of the form \( O(n^{\log_2 3}) \). Your task here is to get a more accurate estimate, which includes a value for the hidden \( O(.) \) constant, assuming single byte arithmetic operations are used as the base case for the recursion.

part (b) Now we consider an asymptotically faster algorithm to compute the product of two integer numbers, using the Fast Fourier Transform to perform polynomial multiplication. Compute the product of \( a \) and \( b \) as follows:

- Interpret the digits of each number \( a = \sum_{i=0}^{n-1} a_i B^i \), \( b = \sum_{i=0}^{n-1} b_i B^i \) as the coefficients of a polynomial \( p_a(X) = \sum_i a_i X^i \) and \( p_b(X) = \sum_i b_i X^i \). Notice that the input numbers can be recovered from the polynomials as \( a = p_a(B) \) and \( b = p_b(B) \).
• Use the $O(n \log n)$ fast polynomial multiplication algorithm from the book to compute the coefficients $c_1, \ldots, c_{2n-2}$ of the product polynomial $p_c(X) = \sum c_i X^i = p_a(X) \cdot p_b(X)$. Notice that the result is a polynomial with positive integer coefficients of degree at most $2(n - 1)$. However, the coefficients of this polynomial $c_i$ may be larger than $B$, and be represented by more than one byte.

• Compute the product $c = a \cdot b$ by evaluating the polynomial $p_c(B) = p_a(B) \cdot p_b(B) = a \cdot b$ at $B$, i.e., compute the digits $d_i$ of $c = \sum d_i B^i$ from the polynomial coefficients $c_i$ of $p_c$, using arithmetic operations on bytes.

Assume for simplicity that the precision offered by standard floating point numbers is enough to compute the integer polynomial $p_c(X)$ by first using FFT with floating point numbers, and then rounding the (floating point) coefficients of the resulting polynomial to the closest integer. Assume also that a single arithmetic operation on (real or complex) floating point numbers is 10 times bigger than integer arithmetic operations on bytes, i.e., each floating point operation takes 10 units of time.

Give an upper bound on the cost of multiplying two $n$-digit integer numbers using this algorithm. You can use the FFT and fast polynomial multiplication algorithms as a black box. But, similarly to part (a), you should give a concrete upper bound on their cost, including a specific value of the constant hidden in the asymptotic notation. Your run time analysis should include both the cost of the floating point operations needed by the FFT (and fast polynomial multiplication) algorithms, and integer arithmetic operations used to compute the digits of the final result.

part (c) Give some exemplary values of cost estimates of the two algorithms for various values of $n$, assuming a 1GHz computer capable of performing $10^9$ byte arithmetic operations or $10^8$ floating point operations in one second. Determine the crossover point of the two algorithms, i.e., the smallest value of $n$ for which the FFT-based algorithm is faster that the algorithm from the textbook.

Problem 3

The diameter of a tree $T = (V, E)$ is given by

$$\max_{u, v \in V} \delta(u, v)$$

where $\delta(u, v)$ is the shortest path length between the vertices $u$ and $v$. That is, the diameter is the maximum distance between any two nodes in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

Problem 4

Consider the following problem. A library has $n$ books that must be stored in alphabetical order on adjustable height shelves. Each book has a height and a thickness. The width of the shelf is fixed at $W$, and the sum of the thicknesses of books on a single shelf must be at most $W$. The next shelf will be placed on top, at a height equal to the maximum height of a book in the shelf. Give an algorithm that minimizes the total height of shelves used to store all the books. You are given the list of books in alphabetical order, $b_i = (h_i, t_i)$, where $h_i$ is the height and $t_i$ is the thickness.