Problem 1 (20 points)

Problem 4-7 (Monge arrays) at pages 88-89 in the textbook. This multipart problem is long, but not particularly hard. It is meant to test mostly your ability to clearly write simple proofs using methods you should already know. (E.g., induction, recurrences, substitution method, asymptotic notation, etc.) Here are some comments/extra hints.

1. Part a (5 points). Ignore the hint in the textbook. Use the following hint instead: For the “if” part, use induction on the size \((k - i) \cdot (l - j)\) of the rectangular region defined by the columns and rows.

2. Part b (0 points). You can skip this part.

3. Part c (5 points).

4. Part d (5 points). Assume for simplicity that \(m = 2^i\) for some \(i \geq 1\). You can use the result from Part c, even if you didn’t prove that part. In addition to explaining how to compute \(f\) for the odd rows (as asked in the textbook), you should also give more details about the main recursive procedure. In particular, you should explain how to efficiently “construct a submatrix \(A’\)”. (Notice that copying the content of the odd rows to some other area of memory is not a good solution because it would result in \(\Omega(mn)\) running time.)

5. Part e (5 points). Hint: write the running time \(T(m, n)\) as a function of both \(m\) and \(n\). Then use the substitution method to prove that \(T(m, n)\) is \(O(m + n \log_2 m)\).

Problem 2 (20 points)

Consider the closest pair of points problem in 3-dimensional space: given an array \(P[1, \ldots, n]\) of points where each \(P[i] = (x, y, z)\), find \(i\) and \(j\) such that \(P[i]\) and \(P[j]\) are as close as possible. Give as efficient an algorithm as possible to solve this problem. (Hint: Can you achieve \(O(n \log n)\) running time as we did in class for points in the plane?)

Optional part: What about 4 dimensions? 5-dimensions? etc. Can you give an efficient generic algorithm that takes the dimension \(d\) of the space as one of the inputs? What is the performance of this algorithm as a function of \(n\) and \(d\)?
Problem 3 (20 points)

Some applications require the use of arbitrary precision arithmetic, i.e., numbers are not stored using a predetermined fixed amount of memory (e.g., 2 bytes or 6 bytes), but variable amount of memory depending on the size of each number. E.g., you can represent an integer as an array of \( n \) digits \( D[0, \ldots, n-1] \) where each \( D[i] \) is a number between 0 and 9. The integer represented by array \( D \) is \( \sum_{i=0}^{n-1}(10)^i D[i] \).

When arbitrary precision integers are used, it is no longer justified to assume that arithmetic operations can be performed in constant time, and the time required by an operation depends on the size of the arrays. Assume we want to multiply to \( n \)-digit numbers. The grade-school method to multiply two integers gives an algorithm to perform multiplication in \( \Theta(n^2) \) time.

Use the divide and conquer approach to devise an algorithm to multiply integers in \( O(n^{\log_2 3}) \leq O(n^{1.585}) \) time. (Hint: divide each array in half, and use a method similar to that used in class to perform fast matrix multiplication.)

Your solution should include a clear description of the recursive algorithm, proof of correctness, and running time analysis.

Problem 4 (20 points)

Consider the “Top-k list” problem: given an array \( A[1, \ldots, n] \) of \( n \) integers and a parameter \( k \leq n \), output the list of the biggest \( k \) elements in \( A \) sorted in descending order. Give as efficient an algorithm as possible to solve the above problem.

Problem 5 (20 points)

Implement the classic \( \Theta(n^3) \) matrix multiplication procedure to multiply \( n \times n \) square matrices, and Strassen algorithm. (You can assume \( n \) is a power of 2.) The matrices should have fixed size entries, e.g., words (integers), or floating point numbers, and you should ignore overflow or numerical stability problems.

Run experiments on matrices of size 2, 4, 8, 16, 32, 64, 128, \ldots to determine the crossover point, i.e., smallest value of \( n \) for which Strassen algorithm becomes faster than the standard one.

Implement a hybrid algorithm that follows Strassen’s recursion for big matrices, but performs the standard algorithm when matrices are smaller than a given threshold \( t \). What is the optimal value of \( t \)? (Notice, this is not necessarily the same as the crossover point as defined above.) Comment your results are express your opinion about the practicality of Strassen multiplication algorithm.

You are not required to submit code for this problem, but you should describe the characteristics of your implementation (programming language, compiler, processor, etc.).