CS:5350 Midterm Exam, Spring 2016

Your answers will be graded primarily for correctness, but as in the homeworks, clarity, precision, and conciseness will also be important. Problem 1 is worth 70 points, and the remaining three problems are worth 60 points each.

1. You are presented three different "divide-and-conquer" algorithms for the same problem. Let us call these algorithms A, B, and C. Here are "high-level" descriptions of these algorithms:

   Algorithm A: In the *divide* step, Algorithm A starts by doing some work that results in five subproblems, each having size one-half of the size of the original problem. Algorithm A then recursively solves the five subproblems (in the *conquer* step) and *combines* the solutions of these subproblems to get a solution to the original problem. The divide and combine steps of Algorithm A run in $O(\log n)$ time, where n is the input size of the problem.

   Algorithm B: Given an input of size n, in the *divide* step, Algorithm B does some work to produce two subproblems, one of size $n-1$ and one of size $n-2$. It then recursively solves the two subproblems (in the *conquer* step) and *combines* the obtained solutions into a solution for the original problem. The divide and combine steps of Algorithm B take $O(1)$ time.

   Algorithm C: In the *divide* step, Algorithm C does some work to produce seven subproblems, each having size one-eighth of the size of the original problem. Algorithm C then recursively solves the seven subproblems (in the *conquer* step) and *combines* the solutions of these subproblems to get a solution to the original problem. The divide and combine steps of Algorithm C run in $O(n)$ time.

   Assume that there are constants c_A, c_B, and c_C (for each algorithm, respectively) such that, when the input size n (to Algorithm X) falls below c_X, then Algorithm X abandons recursion and solves the problem directly in some other manner, taking $O(1)$ time to solve the problem. (Here X can refer to either A, B, or C.) Thus $n < c_A$, $n < c_B$, and $n < c_C$ correspond to the base cases of the three algorithms.

   (a) Write down the recurrence for Algorithm C and then solve it to obtain the running time of the algorithm. Express your answer as $O(f(n))$, where $f(\cdot)$ is a function of the input size n. Show all your work.

   (b) Algorithms are considered "efficient" if their running time is bounded above by polynomial function of the input size. Are all three algorithms described above "efficient" in this sense? (Yes or No) Justify your answer.

   (c) How many function calls to base cases will result from a call to Algorithm A on an input of size n?

   (d) Based on your answers to parts (a)-(c), which algorithm would you pick to implement assuming that asymptotic running time is your only criteria? Briefly (in 1-2 sentences) justify your choice.

2. Suppose we are given an array $A[1..n]$ with the *boundary conditions* that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a local minimum if it is less than or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are six local minima in the following array:

9	**7**	7	2	**1**	3	7	5	**4**	7	**3**	**3**	4	8	**6**	9

   We can obviously find a local minimum in $O(n)$ time by scanning through the array. This problem leads you to a faster algorithm.

(a) Describe a *divide-and-conquer* algorithm that, given an array $A[1..n]$ with the boundary conditions mentioned above, finds and returns a local minimum in $O(\log n)$ time. You can assume that $n \geq 3$.
Hint: With the given boundary conditions, the array must have at least one local minimum. Why?

(b) Prove the correctness of your algorithm.

(c) Write down a recurrence for the running time of your algorithm. You do not have to solve this recurrence.

3. You are given an array $A[1..n]$ containing numbers (not necessarily integers), which can be positive or negative or zero. This problem has you construct a *dynamic programming* algorithm, running in $O(n)$ time, that finds indices q and p, $n \geq p > q \geq 1$ such that $A[p] - A[q]$ is *maximum*. Assume $n \geq 2$.

   (a) Let $OPT(j)$ denote the maximum difference $A[p] - A[q]$, where $j \geq p > q \geq 1$. Write a recurrence expressing $OPT(j)$ in terms of $OPT(\cdot)$ for smaller subproblems.
   Hint: As we have seen in some examples of dynamic programming solutions, you may need to define another (related) problem, whose solution may also participate in the recurrence. Make sure you write the recurrence relation for this related problem as well.

   (b) Using your recurrence explain (in pseudocode or in plain English) how one can solve the above problem in $O(n)$ time.

4. You are given matrices $A_1, A_2, \ldots, A_n$ and you want to compute the matrix product
$$A_1 \times A_2 \times \cdots \times A_n.$$
Each matrix A_i has dimensions $m_{i-1} \times m_i$. Note that the product is a matrix of dimensions $m_0 \times m_n$.

   Because matrix multiplication is associative, one can perform the multiplications in any order. However, different multiplication orders can have very different costs. Recall that multiplying an $a \times b$ matrix and a $b \times c$ matrix in the elementary fashion takes $a \cdot b \cdot c$ multiplications and we will use this as a measure of the cost of multiplying the matrices. For example, suppose that A_1 has dimensions 50×20, A_2 has dimensions 20×1, and A_3 has dimensions 1×10. Then multiplying in the order $(A_1 \times A_2) \times A_3$ will have cost $50 \cdot 20 \cdot 1 + 50 \cdot 1 \cdot 10 = 1000 + 500 = 1500$. This is because we first multiply $A_1 \times A_2$ and this has cost $50 \cdot 20 \cdot 1 = 1000$. Then we multiply a 50×1 matrix (the product of A_1 and A_2) with a 1×10 matrix (A_3) and this has an additional cost $50 \cdot 1 \cdot 10 = 500$. Now note that multiplying in the other order, i.e., $A_1 \times (A_2 \times A_3)$ has cost $20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 = 10200$. From this example it should be clear that performing $A_1 \times A_2$ first, before the other multiplication, is much cheaper than the other option of multiplying $A_2 \times A_3$ first.

   This problem has you construct a dynamic programing algorithm that takes as input the matrix dimensions $m_0, m_1, \ldots, m_n$ and finds the cost of a cheapest ordering of the multiplications. Note that your algorithm is not actually multiplying the matrices, just figuring out the order in which the matrices are to be multiplied.

   (a) Carefully define your subproblems.

   (b) Express the cost of a cheapest ordering of the multiplications in a subproblem in terms of costs of cheapest multiplication orderings for smaller subproblems.
   Hint: This should seem similar to a homework problem.

   (c) Write pseudocode for a dynamic programming algorithm (based on your recurrence(s) in (b)) to solve the problem of computing the cost of a cheapest multiplication ordering.

   (d) What is the running time of your algorithm?
