Problem 3-1 (Stock Profit) 10 (+6) Points

Sometimes, computing “extra” information can lead to more efficient divide-and-conquer algorithms. As an example, we will improve on the solution to the problem of maximizing the profit from investing in a stock (page 68-74).

Suppose you are given an array $A$ of $n$ integers such that entry $A[i]$ is the value of a particular stock at time interval $i$. The goal is to find the time interval $(i, j)$ such that your profit is maximized by buying at time $i$ and selling at time $j$. For example, if the stock prices were monotone increasing, then $(1, n)$ would be the interval with the maximal profit ($A[n] - A[1]$). More formally, the current formulation of the problem has the following input/output specification:

Input: Array $A$ of length $n$.


(a) (6 Points) Suppose your change the input/output specification of the stock problem to also compute the largest and the smallest stock prices:

Input: Array $A$ of length $n$.

Output: Indices $i \leq j$ maximizing $(A[j] - A[i])$, and indices $\alpha, \beta$ such that $A[\alpha]$ is a minimum of $A$ and $A[\beta]$ is a maximum of $A$.

Design a divide-and-conquer algorithm for this modified problem. Make sure you try to design the most efficient “conquer” step, and argue why it works. How long is your conquer step? (Hint: When computing optimal $i$ and $j$, think whether the “midpoint” $n/2$ is less than $i$, greater than $j$ or in between $i$ and $j$.)

(b) (4 Points) Formally analyze the runtime of your algorithm and compare it with the runtime of the solution for the Stock Profit problem in the book.

(c) (Extra Credit; 6 Points) Design a direct, non-recursive algorithm for the Stock Profit problem which runs in time $O(n)$. Write its pseudocode. Ideally, you should have a single “for $i = 1$ to $n$” loop, and inside the loop you should maintain a few “useful counters”. Formally argue the correctness of your algorithm. (Hint: Scan the array left to right and maintain its running minimum and the best solution found so far. Under which conditions would the best current solution be improved when scanning the next array element?)
Problem 3-2 (Grid local minimum) 18 points

A local minimum of a two-dimensional array \( \{ A[i, j] \mid 1 \leq i, j \leq n \} \) is an index \( (i, j) \in \{1, \ldots, n\} \times \{1, \ldots, n\} \) which is less than or equal to all of its neighbors, where we say that two nodes are neighboring if they are either vertically or horizontally (but not diagonally) adjacent in the array. Note that every array has at least (and possibly more than) one local minimum, since the “global” minimum of the entire array is also a local minimum. The eventual goal of this problem is to design an efficient divide-and-conquer algorithm to find some local minimum of a given (unsorted) array \( A \) of size \( n \times n \).

(a) (2 points) Consider the following “greedy” algorithm. Start with any node \( v = (i, j) \). If \( v \) is a local minimum, then output \( v \). Else take the smallest neighbor of \( v \) (break ties arbitrarily) and repeat the above process with the neighbor until you find a local minimum. Prove that this algorithm always terminates in time \( O(n^2) \).

(b) (3 points) What is the exact length (number of nodes, not “edges”) of the “local minimum path” of the greedy algorithm on the following \( 7 \times 7 \) grid, starting with the initial point \( v = (1, 1) \) (equal to 30). By generalizing this picture from \( n = 7 \) to general \( n \), show that the worst case running time of the greedy algorithm is \( \Omega(n^2) \).

\[
\begin{pmatrix}
30 & 100 & 16 & 15 & 14 & 100 & 0 \\
29 & 100 & 17 & 100 & 13 & 100 & 1 \\
28 & 100 & 18 & 100 & 12 & 100 & 2 \\
27 & 100 & 19 & 100 & 11 & 100 & 3 \\
26 & 100 & 20 & 100 & 10 & 100 & 4 \\
25 & 100 & 21 & 100 & 9 & 100 & 5 \\
24 & 23 & 22 & 100 & 8 & 7 & 6
\end{pmatrix}
\]

(c) (3 points) For simplicity of calculation, assume that \( n = 2^k - 1 \) for some integer \( k \geq 1 \). Consider the following divide-and-conquer algorithm: at every step, find the minimum element \( v \) among all elements of the middle row \( 2^k - 1 \) and all elements of the middle column \( 2^k - 1 \). If \( v \) is a local minimum, output \( v \). Else take the smallest neighbor of \( v \) (call it \( w \)), and recurse in the quadrant (north-east, north-west, south-east or south-west) of \( A \) of size \( (n - 1)/2 = (2^{k-1} - 1) \) where \( w \) lies (not counting the middle row/column in the quadrant, so one row/column is eliminated before dividing by 2).

While this algorithm looks appealing, you will prove that it is not correct in general. For this, consider the following grid containing all the numbers from 1 to 49 exactly once, except three weights 10, 31, 39 are marked with the ?. Fill the missing numbers marked with ? with 10, 31, 39 in a way such that the algorithm given above gives the wrong answer, and state what this wrong answer is.
(d) (8 points) For simplicity of calculation, assume that \( n = 2^k + 1 \) for some integer \( k \geq 0 \) and all the numbers \( A[i,j] \) are distinct. Consider a slightly more complex divide-and-conquer algorithm. In addition to the middle row \((2^{k-1} + 1)\) and column \((2^{k-1} + 1)\), also look at the boundary nodes \{\((1,i),(n,i),(i,1),(i,n) \mid i = 1 \ldots n\}\), and find the global minimum \( a \) (located at node \( v \)) of all these \( 6n - 9 \) numbers \( (3 \text{ rows and 3 columns of size } n, \text{ except 9 "intersection points" are counted twice}) \) in \( O(n) \) time. If \( v \) is a local minimum, output \( v \). Else take the smallest neighbor of \( v \) (call it \( w \)), and recurse in the quadrant (north-east, north-west, south-east or south-west) of \( A \) of size \((n+1)/2 = 2^{k-1} + 1\) (where the quadrant includes the boundary, so you effectively duplicate the middle row/column before dividing by 2) where \( w \) lies.

Prove the correctness of this modified algorithm. To do that, prove the following stronger inductive statement: the algorithm computes an answer which is (a) correct local minimum \((i,j)\) and (b) \( A[i,j] \) is less or equal than every number on the boundary of the square. Make sure you stress where (i) you use the fact that a smaller neighbor \( w \) is selected; (ii) that all the numbers are distinct.

(e) (2 points) Write the recurrence equation and solve it to compute the running time of the algorithm in part (d).

**Problem 3-3 (Trominoes) 10 pts**

An \( n \)-tromino is a \( 2^n \times 2^n \) “chessboard of unit squares with one corner removed” (figure below drawn for \( n = 3 \)). Assume that initially you are given a 1-tromino (i.e., a simple \( L \)-shaped tile of area 3 drawn on the right), but you have a friendly genie whom you can ask to perform the following two operations in any order:

- **Duplicate**: This operation takes an object as input, and creates a second identical copy of this object.
- **Glue**: This operation takes two objects as input, and glues them together (along the sides, without any overlaps) in a manner specified by you.

(a) (6 points) Design a recursive algorithm \textsc{Tromino}(\( n \)) that creates an \( n \)-tromino from 1-tromino using the a minimum number of calls to the genie. You only need to specify the top level of
the recursion, without the need to explicitly “unwind” the recursion all the way to \( n = 1. \)  
(Hint: First step is to DUPLICATE the original 1-tromino, as otherwise you “lose” it in the recursive call(s), and you might need it in the “conquer” step.)

(b) (4 points) Give a recurrence relation for the number of calls \( T(n) \) to the genie, and solve it.

**Problem 3-4 (Tower of Hanoi)** 10 (+6) points

The Tower of Hanoi is a well known mathematical puzzle. It consists of three rods, and a number \( n \) of disks of different sizes which can slide onto any rod. The puzzle starts with all disks stacked up on the 1st rod in order of increasing size with the smallest on top. The objective of the puzzle is to move all the disks to the 3rd rod, while obeying the following rules.

- Only one disk is moved at a time
- Each move consists of taking one disk from top of a rod, and moving it on top of the stack on another rod
- No disk may be placed on top of a smaller disk.

A recursive algorithm that solves this problem is as follows: We first move the top \( n - 1 \) disks from rod 1 to rod 2. Then we move the largest disk from rod 1 to rod 3 and then move the \( n - 1 \) smaller disks from rod 2 to rod 3. Using the symmetry between the rods, the number of steps that this algorithm takes is given by the recurrence

\[
T(n) = 2T(n - 1) + 1,
\]

which can be solved to get \( T(n) = 2^n - 1. \)

(a) (5 points) Show that the above algorithm is optimal, i.e., there does not exist a strategy that solves the Tower of Hanoi puzzle in less than \( 2^n - 1 \) steps.

(b) (5 points) Suppose the moves are restricted further such that you are only allowed to move disks to and from rod 2. Give an algorithm that solves the puzzle in \( O(3^n) \) steps.

(c) (6 points (Extra credit)) Suppose the moves are restricted such that you are only allowed to move from rod 1 to rod 2, rod 2 to rod 3, and from rod 3 to rod 1. Give an algorithm that solves the puzzle in \( O((1 + \sqrt{3})^n) \) steps.