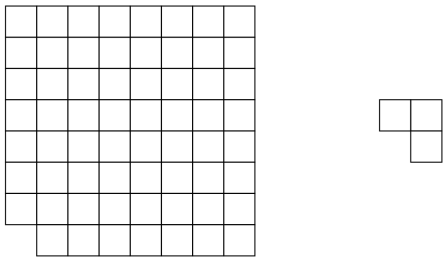
Fundamental Algorithms
Courant, Fall 2017

Homework 3

Instructor: Amir Shpilka
Due Date: Wed Sep 27, 11:55pm

1. 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 TROMINO(n) that creates an n-tromino from 1-tromino using as few calls to the genie as you can.

   Hint: Make sure you DUPLICATE the original 1-tromino first, 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.

   (c) (2 points) (**Bonus**[1]) If you haven't already solved question 1 (d) from Homework 1, can you solve it now?

2. An array $A[0 \dots (n-1)]$ is called *rotation-sorted* if there exists some cyclic shift $0 \le c < n$ such that $A[i] = B[(i + c) \bmod n]$ for all $0 \le i < n$, where $B[0 \dots (n-1)]$ is the sorted version of A.[2] For example, $A = (2, 3, 4, 7, 1)$ is rotation-sorted, since the sorted array $B = (1, 2, 3, 4, 7)$ is the cyclic shift of A with $c = 1$ (e.g. $1 = A[4] = B[(4 + 1) \bmod 5] = B[0] = 1$). For simplicity, below let us assume that n is a power of two (so that we can ignore floors and ceilings), and that all elements of A are distinct.

   (a) (4 points) Prove that if A is rotation-sorted, then one of $A[0 \dots (n/2-1)]$ and $A[n/2 \dots (n-1)]$ is fully sorted (and, hence, also rotation-sorted with $c = 0$), while the other is at least rotation-sorted. What determines which one of the two halves is sorted? Under what condition *both halves* of A are sorted?

   (b) (6 points) Assume again that A is rotation-sorted, but you are not given the cyclic shift c. Design a divide-and-conquer algorithm to compute the minimum of A (i.e., $B[0]$). Carefully prove the correctness of your algorithm, write the recurrence equation for its running time, and solve it. Is it better than the trivial $O(n)$ algorithm?

   Hint: Be careful with $c = 0$ an $c = n/2$; you might need to handle them separately.

3. (10 points) A local minimum of an array $A[1, \dots, n]$ is an index $i \in \{1, \dots, n\}$ such that either (a) $i = 1$ and $A[1] \le A[2]$; or (b) $i = n$ and $A[n] \le A[n-1]$; or (c) $1 < i < n$ and $A[i] \le A[i-1]$ and $A[i] \le A[i+1]$. Note that every array has at least (and possibly more than) one local minimum, since the

[1] Bonus questions are not mandatory, and not attempting them will not harm you in any way. They are often more difficult (so that you don't complain the homework isn't challenging enough) and/or teach extra material.

[2] Intuitively, A is either completely sorted (if $c = 0$), or (if $c > 0$) A starts in sorted order, but then "falls off the cliff" when going from $A[n-c-1] = B[n-1] = \max$ to $A[n-c] = B[0] = \min$, and then again goes in increasing order while never reaching $A[0]$.

global minimum of the entire array is also a local minimum. Describe an $O(\log n)$ divide-and-conquer algorithm to find some local minimum of a given (unsorted) array A of size n.

You must give pseudo-code, prove correctness, and prove the running-time bound to get full credit.

(Hint: Think of binary search for inspiration.)

4. (10 points) You're given a *single-peaked* array, which is an array $A[1, \ldots, n]$ such that there exists an index i such that $A[1, \ldots, i]$ is sorted in ascending order and $A[i+1, \ldots, n]$ is sorted in descending order. You can assume that A contains distinct numbers. Design a divide-and-conquer algorithm to find the index of the maximum element in $O(\log n)$ time.

   Again, you must give pseudo-code, prove correctness, and prove the running-time bound to get full credit.