CS325 Winter 2017: practice question set 2

This set of practice questions help you review the following concepts:

- Designing divide and conquer algorithms and characterize its run time using recurrence relations.
- More proof by induction, particularly using it to prove correctness of recursive algorithms

1. The well-known mathematician George Polya posed the following false "proof" showing through mathematical induction that actually, all horses are of the same color.

   Base case: If there's only one horse, there's only one color, so of course its the same color as itself.
   Inductive case: Suppose within any set of n horses, there is only one color. Now look at any set of $n+1$ horses. Number them: $1, 2, 3, \ldots, n, n+1$. Consider the sets $\{1, 2, 3, ..., n\}$ and $\{2, 3, 4, ..., n+1\}$. Each is a set of only n horses, therefore within each there is only one color. But the two sets overlap, so there must be only one color among all $n+1$ horses.

   Identify what is wrong with this proof.

   Solution:
   For the case of 2 horses, the two sets do not overlap. So we could not go from n=1 to n=2.

2. *DPV 2.17. You need to 1) describe the algorithm in clear pseudo-code; 2) prove its correctness (via induction) and 3) show that its run time is $O(\log n)$.*
   Solution:
   Algorithm:

   `algo`$(A[1, \ldots, n])$
   1. *if* $n == 1$, *return* $(A[n]==n)$
   2. $m = \lceil \frac{n}{2} \rceil$
   3. *if* $A[m] = m$ *return true*
   4. *else if* $A[m] > m$, *return* `algo`$(A[1, \ldots, m-1])$
   5. *else return* `algo`$(A[m+1, \ldots, n])$

   Each recursion reduces the problem size by half, and we do a constant amount of work in each function call. This gives us the following recurrence relation:

   $$T(n) = T(n/2) + O(1)$$

   which solves to $T(n) = O(\log n)$.

   Proof of correctness:
   Base case: for array of size 1, it is trivially correct because it directly compares the element with the index.
   Inductive assumption: Assume that the algorithm is correct for array of size 1,...,up to k.
   Inductive step: for array of size $k+1$.
   If line 3 is true, the algorithm is correct.
   If line 4 is true, i.e., $A[m] > m$, *we know all elements of the second half must be bigger than their indices, because* $A[i] \geq i - m + A[m] > i - m + m = i$. *The first inequality is because all elements are distinct and sorted. As such, we can eliminate the second half and only search the first half (via recursion).*
   Conversely, if $A[m] < m$, *then all elements of the first half must be smaller than their indices, and we only need to consider the second half.*
   In both cases, because the recursive call of algo has input size $\leq k$, *which will return the correct answer based on the inductive hypothesis. Thus the overall algorithm is correct. Q.E.D*

3. *Given two sorted arrays $a[1, ..., n]$ and $b[1, ..., n]$, given an $O(\log n)$ algorithm to find the median of their combined $2n$ elements. (Hint: use divide and conquer).*

Solution:

Algorithm: *For simplicity, we assume n is an exponential of 2.*

function *median2*(a, b)

if $n \leq 2$

explicitly find the median and return it

compute the median of a and b: m_1 and m_2 respectively

if $m_1 == m_2$*:*

return m_1

else if $m_1 > m_2$*:*

return median2(a_L, b_R)

else:

return median2(a_R, b_L)

Justification*:*

If m_1 and m_2 are equal, the overall median will be m_1.

If $m_1 > m_2$, then we know that at least half of the elements (all elements of a_L and b_L) $\leq m_1$, suggesting that $m_1 \geq$ the overall median. As such, all elements in a_R must be bigger than the median. Similarly, all elements of a_R and $b_R \geq m_2$, suggesting that $m_2 \leq$ the overall median. So all elements of $b_L \leq$ the overall median. In this case, we can remove a_R and b_L and recursively identify the median of the remaining two subarrays.

In contrast, If $m_1 < m_2$, then we know that all elements of a_R and $b_R \geq m_1$, suggesting that $m_1 \leq$ the overall median. As such, all elements in $a_L \leq$ the overall median. Similarly, all elements of a_L and $b_L \leq m_2$, suggesting that $m_2 \geq$ the overall median. So all elements of $b_R \geq$ the overall median. In this case, we can remove a_L and b_R and recursively identify the median of the remaining two subarrays.The correctness of the overall algorithm will be based on induction. Details omitted.

The run time of this algorithm can be described by $T(n) = T(n/2) + c$. Solving the recurrence relation, we have $T(n) = O(\log n)$.

4. *Given an array A of n distinct numbers whose values $A[1], A[2], ..., A[n]$ is unimodel: that is, for some index $p \in [1, n]$, the values in the array first increases up to position p, then decrease the remainder of the way. For example $[1, 2, 5, 9, 7, 3]$ is one such array with $p = 4$. Please design an $O(\log n)$ algorithm to find the peak p given such an array A.*

Solution:

`algo`$(A[1, \ldots, n])$

1. *if* $n == 1$*, return (A[n]==n)*

2. $m = \frac{n}{2}$

3. *if* $A[m] < A[m+1]$ *return* `algo`$(A[m+1, \ldots, n])$

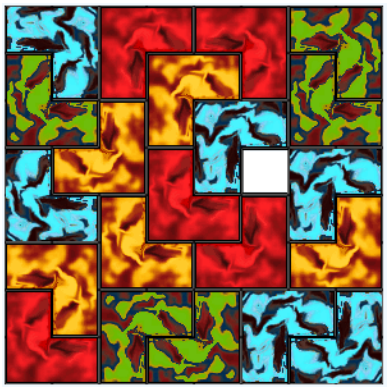
5. *else return* `algo`$(A[1, \ldots, m])$

Justification: If $A[m] < A[m+1]$, we know the peak cannot be on the left of m, we just need to recursively search $A[m, ..., n]$. Similarly, if $A[m] > A[m+1]$, we know the peak cannot possibly be on the right of m, we just need to recursively search in $A[1, ..., m]$. The overall correctness can be proved via induction. Details skipped.

For runtime, for each recursive function call, we do a constant amount of work and reduce the problem size by half. This gives us the following recurrence relation:

$$T(n) = T(n/2) + c$$

which solves to $T(n) = O(\log n)$.



5. *Divide and conquer for the Tromino Puzzle.*
   Use divide and conquer to design an algorithm/strategy to fill up any $2^n \times 2^n$ board with one square removed using tromino tiles. A tromino tile is a piece formed by three adjacent squares in the shape of an L and it can be rotated to fit the board. See the figure below for an example solution for a 8×8 board with cell $(4, 5)$ removed. The following page contains an interactive version of this puzzle (`https://www3.amherst.edu/~nstarr/trom/puzzle-8by8/`). You can play with it to get some idea. (Hint: assuming that you can solve the problem of $2^{n-1} \times 2^{n-1}$ with an arbitrary cell removed, how can you use that to solve the problem of $2^n \times 2^n$ size?)

   Solution:

   For this problem it is helpful to think about it from inductive-proof point of view to show that a puzzle is always solvable.

   Base case: *if the board size is 2×2, we can obvious always solve it regardless where the missing cell is.*

   Inductive hypothesis: *assume we can solve the puzzle for a board of size $2^k \times 2^k$*

   Inductive step: *now consider a board of size $2^{k+1} \times 2^{k+1}$. We can consider the four quadrants of the board, each will be of size $2^k \times 2^k$. One of the quadrant will contain the missing cell. We will simply place a tile in the center of the board with the three squares of L covering the three quadrants that do not have a missing cell. As such, we created four smaller puzzles of board size $2^k \times 2^k$. The inductive hypothesis says that we can solve each of these subproblems. Q.E.D*

   This suggests that our algorithm works as follows:
   If $n = 1$ directly solve the puzzle
   else
   place a single tile in the center of the board to make sure each quadrant has a single cell removed
   recursively solve the puzzle in each of the quadrant