CMPS 102: Homework #1

Due on Tuesday, April 7, 2015

John Allard 1437547

Problem 1

In a binary tree all nodes are either internal or they are leaves. In our definition, internal nodes always have two children and leaves have zero children. Prove that for such trees, the number of leaves is always one more than the number of internal nodes. There are many possible proofs based on induction, but there might be others.

Answer :

Let T be a binary tree subject to the definition above, T_I be the number of internal nodes, and T_L the number of leaves in T. Let $P(n)$ be the statement 'Given a binary tree T on n internal nodes, $T_L = T_I + 1 = n + 1$'.

Base Case

$P(n = 0)$: A tree with no internal nodes consists of only a root node, which would be a leaf. Thus

$$T_L = 1, T_I = 0, T_L = T_I + 1$$

Thus the base case is confirmed.

Inductive Case

Assume for any binary tree T on $n \geq 1$ internal nodes that $P(n)$ is true, i.e. $T_L = T_I + 1 = n + 1$. Now, pick an arbitrary binary tree on $n + 1$ internal nodes, call it T. Note that because $T_I = n + 1 \geq 1$, at least one internal node with 2 leaves as children must exist. Select any of these internal nodes, label that node k. Take a new leaf node, and put it in place of the internal node k. This exchange has two effects, it reduces the number of internal nodes by one (removal of k), and reduces the number of leaves by one (remove the two leaf-children of k, place a new leaf in place of k, $-2+1 = -1$). Label this new tree T'.

$$T'_I = T_I - 1$$

Because T was a binary tree on $n + 1$ internal nodes :

$$T_I = n + 1$$

$$T'_I = (n + 1) - 1$$
$$T'_I = n$$

Since T' is on n internal nodes, we can apply our inductive hypothesis, $T_L = T_I + 1$

$$T'_L = T'_I + 1$$

$$T'_L = (n) + 1$$

To get from T to T', we removed one leaf node in the process. Thus :

$$T_L = T'_L + 1$$

$$T_L = (n + 1) + 1 = n + 2$$
$$T_L = n + 2 = (n + 1) + 1 = (T_I) + 1$$

Which is exactly $P(n + 1)$.

Problem 2

For $n \geq 0$ consider $2^n \times 2^n$ matrices of 1s and 0s in which all elements are 1, except one which is 0 (The 0 is at an arbitrary position).
Operation: At each step, we can replace three 1s forming an L with three 0s (The Ls can have an arbitrary orientation).
Prove that for such matrices there always is a sequence of operations that transforms the matrix to the all 0 matrix.

Answer :

Let M_n designate the $2^n \times 2^n$ matrix of all ones except a single zero as described above for $n \geq 0$.
Let $P(n)$ be the statement 'Given a matrix M_n for $n \geq 0$, M_n can be reduced to the all-zero matrix by a sequence of L-removal operations which changes 3 ones to zeros in an L pattern of arbitray orientation'.
Base Case
$P(n = 0)$: If $n = 0$, then M_n is a $2^0 \times 2^0 = 1 \times 1$ matrix. Because all matrices of the above form must have a single 0, and a 1×1 matrix has only one component, the entire matrix is zero and thus we need no operations to reduce it to the zero matrix. Thus the base case is confirmed.
Inductive Case
Assume for any given $n \geq 1$ that $P(n)$ holds, i.e. that the matrix M_n can be reduced to the zero matrix via a sequence of L-removal operations. Construct a matrix of the form M_{n+1} (all ones except a single zero placed anywhere). M_{n+1} must be square and have sides that are a power of two, by its very definition (M_k is size $2^k \times 2^k$). This means we can divide it into four equal quadrants by dividing along the vertical and horizontal gaps between the middle rows and columns. Each quadrant will have sides of length 2^n, since M_{n+1} had sides of length 2^{n+1} and we divided each side-length in half to get M_n.
Now, regardless of where the single zero was initially placed, it must be in one and only one of these quadrants. Pick the other three quadrants that do not have a zero (consist of all ones). Place a single zero in the innermost component for each of the three quadrants that consist of all ones. What I mean by innermost are pieces touching the center of the matrix M_{n+1}, these 3 pieces form an L shape and thus this is a valid operation to perform.
Now, all of the 4 quadrants are of size $2^n \times 2^n$ and consist of all ones except a single zero, so we can apply $P(n)$. This means that each quadrant can be reduced to a zero matrix in a finite sequence of remove-L operations. Because it took us only a finite (single) valid operation (removing the L in the 3 one-filled quadrants) to get to this step from M_{n+1}, then the matrix M_{n+1} can also be reduced to the zero matrix in a finite sequence of steps. Thus we assumed $P(n)$ and showed that this implies $P(n + 1)$, concluding the proof.

Problem 3

Suppose you are given an array A of n distinct integers with the following property: There exists a unique index p such that the values of A[1...p] are increasing and the values of A[p...n] are is decreasing.
For instance, in the example below we have $n = 10$ and $p = 4$.

$$A = [2, 5, 12, 17, 15, 10, 9, 4, 3, 1]$$

Design a O(log n) algorithm to find p given an array A with the above property.

Algorithm :
Since the required run-time is $O(\log(n))$, I know that I have to find a way to reduce the search space for p by a constant multiple for each call. I applied a slight change to the typical binary search algrorithm to accomplish this task. My algorithm looks not for a specific key, but instead how the two middle-elements compare to one another. If they are increasing (low to high index), then I recurse on the right half of the array, if they are decreasing I recurse on the left half. When I have only one element, I know that I have found the index of the number where the numbers change from increasing to decreasing. If I have narrowed it to two items, I can perform 2 comparisons to determine which one corresponds to p.

```
// A = array of n elements, l = left element to sub-search
// r = right element to subsearch, return the index of p.
1.   findp(A, l, r)
2.        i = (r+l)/2 // get middle element
3.        if l == r :  // we found it
4.             return r
5.        else if r-l == 1 :  // it's one of the two we are searching
6.             if A[r] >= A[l]
7.                  return r
8.             else
9.                  return l
10.        if A[i] >= A[i-1] // if increasing
11.             return findp(A, i, r) // must be in right half
12.      else
13.             return findp(A, l, i) // must be in left half
```

Proof of Correctness :

To start, we must acknowledge that if an array holds the partial-sorting property given above, then so must any contiguous sub-section of that array. Any sub-section will contain elements either strictly to the left, strictly to the right, or on both sides of the p index, given the properties of the partial-sort, this sub-section then must also contain items that are either strictly increasing, strictly decreasing, or increasing and then decreasing from some given point, satisfying the definition of a valid partial-sort as defined above. This fact is used is the proof below.
Let $P(n)$ be the statement 'On any array (or sub-array) A of size n, the `findp` algorithm will find the correct index p.'.
Base Case :
There are two base cases, if the length of the sub-array we are searching is 1 or 2. $P(n = 1)$: If the sub-array is of size one, then the correct index to return would be of the only element. `findp` will be called with $l = r$, which will be caught by the `if` statement on line 3 and r would be returned, which is the item we are looking for.
$P(n = 2)$: If there are two elements in the sub-array, then the `else if` statement on line 5 with be caught.

This will examine the two elements and return the right index if it is greater than or equal to the left and return the left index otherwise. Since a two-element sub-array must either be decreasing or increasing, this `else if` statement will return the correct index.

Inductive Case :

Assume for $n > 2$ that $P(k)$ is true for $k = [1 \ldots n]$, i.e. that the correct index p can be found for any given array A on k items that is partially-sorted (as described above). I will use strong induction to show that this implies $P(n+1)$ is true.

Take a given array A on $n+1$ elements as described above. Since $n > 2$, $r > l+1$ so the first two if-statements don't execute. The algorithm then recurses on either $\lceil (r+l)/2 \rceil$ or $\lceil n - (r+l)/2 \rceil$ items depending on if it is in the left or right half respectively. Sicne both of these numbers of items are less than $n+1$, we can apply the inductive hypothesis on whichever half is taken. The inductive hypthesis says that the algorithm will be able to find the right p index on any array from $k = [1 \ldots n]$, and since we are passing it a subarray in that range, we know that thte algorithm will be able to find the right p index.

Proof of Runtime Complexity :

I will now attempt to prove that the run-time complexity for the above algorithm is of order $\log(n)$.

The `findp` takes an array A of $n >= 1$ elements that is partially sorted in such a way that all elements increase up to a certain index p and after that they all decrease. It also takes in a left and right index value between which the search will occur. For each call to the algorithm, we perform a constant number of comparisons, and for each recursive call we will be narrowing the search-space by about half. This can be formalized by the following recurrence relation :

$$T(n) = \left\{ \begin{array}{cc} 1 & \text{if n = 1} \\ \text{T(n/2) + c} & \text{if } n > 1 \end{array} \right\}$$

Let $P(n)$ be the statement 'On any array of size A of size n, the `findp` algorithm will find the index p in at most time $c * \log(n)$'.

Base Case :

$P(n = 1)$: If the array is of size one, then the correct index to return would be of the only element, namely 1. `findp` will be called with $l = r = 1$, which will be caught by the `if` statement on line 3 and $r = 1$ would be returned. Thus the base case is confirmed.

Inductive Case :

Assume for $n >= 1$ that $P(n)$ is true, i.e. that any given array that is partially-sorted (as described above) array A on n