CS 513
Solutions to #2

1. Find a closed form for the recurrence:

   $T(1) = 1$
   $T(n) = 2T(n/2) + \log n$ (for $n \geq 2$)

   You may assume n is a power of 2. Give a tight "big-oh" bound on T. Show your derivation and prove your answer is correct.

   Guess:

$$\begin{aligned}
T(n) &= 2T(n/2) + \log n \\
&= 2[2T(n/2^2) + \log(n/2)] + \log n \\
&= 2^2T(n/2^2) + 2\log(n/2) + \log n \\
&= 2^3T(n/2^3) + 2^2\log(n/2^2) + 2\log(n/2) + \log n \\
&\vdots \\
&= 2^iT(n/2^i) + \sum_{j=0}^{i-1} 2^j \log(n/2^j) \\
&= 2^{\log n}T(1) + \sum_{j=0}^{\log(n)-1} 2^j \log n - \sum_{j=0}^{\log(n)-1} j2^j \qquad n/2^i = 1 \Rightarrow i = \log n \\
&= n + \log n(n-1) - [(\log(n) - 2)2^{\log n} + 2] \qquad \text{By Problem 4.b of Homework \#1} \\
&= n + n\log n - \log n - n\log n + 2n - 2 \\
&= 3n - \log n - 2
\end{aligned}$$

   Theorem 1 $T(n) = 3n - \log n - 2$.

   Proof: By induction.

   Base case: $T(1) = 1 = 3 - \log 1 - 2$.

   Inductive Hypothesis: Suppose that for all $n' < n$, $T(n') = 3n' - \log n' - 2$. Then $T(n) = 2T(n/2) + \log n = 2[3n/2 - \log(n/2) - 2] + \log n$. Therefore $T(n) = 3n - 2\log n + 2 - 4 + \log n = 3n - \log n - 2$.□

   Finally, we can sumarize by saying that $T(n)$ is $\Theta(n)$, since $\lim_{n\to\infty} T(n)/n = 3$.

2. Find a closed form for the recurrence:

   $T(2) = 1$

$T(n) = \sqrt{n}T(\sqrt{n}) + n$ (for $n > 2$)

You may assume n is of the form 2^{2^k}. Give a tight "big-oh" bound on T. Show your derivation and prove your answer is correct.

Guess:

$$\begin{aligned}
T(n) &= n^{1/2}T(n^{1/2}) + n \\
&= n^{1/2}[n^{1/2^2}T(n^{1/2^2}) + n^{1/2}] + n \\
&= n^{1-1/2^2}T(n^{1/2^2}) + 2n \\
&\vdots \\
&= n^{1-1/2^i}T(n^{1/2^i}) + in \\
&= n^{1-1/2^{\log\log n}}T(2) + n\log\log n \qquad n^{1/2^i} = 2 \Rightarrow i = \log\log n \\
&= n^{1-1/\log n} + n\log\log n \\
&= n/2 + n\log\log n
\end{aligned}$$

Theorem 2 $T(n) = n/2 + n\log\log n$.

Proof: By induction.

Base case: $T(2) = 1 = 2/2 - 0$.

Inductive Hypothesis: Suppose that for all $n' < n$, $T(n') = n'/2 - n'\log\log n'$. Then $T(n) = n^{1/2}T(n^{1/2}) + n = n^{1/2}[n^{1/2}/2 + n^{1/2}\log\log n^{1/2}] + n$. Therefore $T(n) = 3n/2n\log(1/2\log n) = 3n/2 + n\log\log n - n = n/2 + n\log\log n$.□

Finally, we can sumarize by say that $T(n)$ is $\Theta(n\log\log n)$, since $\lim_{n\to\infty} T(n)/n\log\log n = 1$.

3. A *Weight Balanced Search Tree* maintains the following invariant. For each node u,

$$\text{weight(left-child)} + 1 \leq 2\text{weight(right-child)} + 1.$$

(Recall that the weight of a node is the number of descendants.)

(a) Show that the tree can be searched in $O(\log n)$ time.

Solution: From the balancing condition, we have that the number of nodes in the right subtree is at most twice the number of nodes in the left subtree. This gives us the following recurrence for the running time of search:

$$T(n) \leq T(2n/3) + O(1)$$

It is easy to check that the recurrence results in $T(n) = O(\log n)$.

(b) Suppose that on an update, whenever a node goes out of balance, you rebuild the entire subtree rooted at that node. What is the cost of an update?

Solution: To rebuild the subtree at vertex v, we do an in-order traversal of the subtree and extract the elements in sorted order. From here, it is straightforward to construct a balanced tree recursively. Set the middle element to be the root, and recurse on the left and right side.

The whole operation would take $O(\text{weight}(v))$ time.

(c) How many insertions are needed to unbalance a freshly balanced tree?

Solution: For a freshly balanced tree, we have that:

$$|\text{weight(left-child)} - \text{weight(right-child)}| \leq 1$$

Assume that every node inserted goes into the right subtree. Then we need at least weight(left-child) insertions to unbalance the tree.

(d) What is the cost of n insertions? (Hint: an insertion can contribute to the unbalancing of more than one node. How many nodes can an insertion help unbalance?)

Solution: Each insertion can unbalance at most $O(\log n)$ nodes. These would be the nodes along the path to where the new vertex is inserted.

Inserting the node itself takes $O(\log n)$ time. We will argue that rebalancing does not happen too often and can be charged to previous insertions. This would imply that the total cost of n insertions is $O(n \log n)$.

Say that we start from a balanced tree and at some point a node v needs a rebalance. This means that v has a child with weight k and another child with weight $2k$. It took at least k insertions to make that happen and the time to rebuild is $O(k)$. This gives constant amortized time for balancing.

(e) What is the amortized cost of an insertion?

Solution: The cost of an insertion would be the cost of inserting the element, plus a "charge" for each future rebalancing. We already know that an insertion can unbalance at most $O(\log n)$ nodes.

Hence, the total amortized insertion cost would be $O(\log n)$.

4. Consider a rectangular array of distinct elements. Each row is given in increasing sorted order. Now sort (in place) the elements in each column into increasing order. Prove that the elements in each row remain sorted.

Proof: Let $A_k[i]$ be the ith element of the kth row in the initial array. Let $S_k[i]$ be the ith element of the kth row in the array after it is sorted.

Claim 1 *For all $i, j, k \in [1, n]$, $i < j$ implies that $S_k[i] \leq S_k[j]$.*

Proof of claim: Let C_i be the set of elements in the ith column, that is $C_i = \{A_k[i] | 0 \leq k \leq n\}$. The claim will follow if we can show that at least k elements are less than $S_k[j]$ in C_i. To see this, notice that the kth smallest element of C_i will end up as $S_k[i]$.

Now, consider the elements $S_1[j], S_2[j], \ldots, S_k[j]$. Each such $S_l[j]$ started out as some $A_{l'}[j]$. But $A_{l'}[i] < A_{l'}[j] < S_k[j]$. Therefore, we have shown that there are at least k elements in C_i which are no greater than $A_k[j]$ and we have therefore established the claim.□

But this claim is the same as saying that each row is sorted.□

5. Show that if you have a subroutine for $HamP$ which runs in polynomial time, you can solve the $HamC$ problem in polynomial time.

Answer: Suppose we want to determine if G has a hamiltonian cycle. If it does, it must contain an edge. So, for every edge (x, y) (of which there are only $\binom{n}{2}$), we can add a new node x' connected to x and a new node y' connected to y. Call the new graph G'. Call $HamP(G', x', y')$. If any call $HamP(G', x', y')$ comes back true, then we know there is a hamiltonian cycle going through the edge (x, y). If there is a hamiltonian cycle going through (x, y), then $HamP(G', x', y')$ will come back true. So this algorithm will solve the $HamC$ problem.□

6. Show that if you have a subroutine for $HamP$ which runs in polynomial time, you can actually *find* a hamiltonian path in polynomial time.

Answer: We want to find a hamiltonian path from x to y in G (supposing one exists). Then perform the following loop:

For every edge (u, v) in G,

- construct G' be removing (u, v) from G.
- Call $HamP(G', x, y)$.
- If G' still has a hamiltonian path, remove (u, v) from G. Otherwise leave the edge in.

Now, this proceedure throws away edges not needed for the hamiltonian path, but keeps the ones needed. If runs in $\binom{n}{2}$ calls to $HamP$. Thus if $HamP$ takes polynomial time, so does our routine.

Extra Credit: Consider the following recurrence:

$$T(1) = 1$$

$$T(n) = T(n/5) + T(7n/10) + n.$$

You need not give an exact closed form for this recurrence, but give a tight "big-oh" bound on T. As always, prove your answer is correct.

Theorem 3 *The above recurrence $T(n)$ is $\Theta(n)$.*

Proof: We first prove it is $O(n)$. If it is $O(n)$, then there is a c and n_0 such that for all $n > n_0$, $T(n) < cn$. We prove by induction.
Base case: Set $n_0 = 1$. Then for any $c > 1$, $T(1) < c$.
Inductive Hypothesis: Suppose that for all $n' < n$, $T(n') < cn'$. Now $T(n) = T(n/5) + T(7n/10) + n < cn/5 + 7cn/10 + n$ by the inductive hypothesis. Now, we must simply find a c that satisfies the condition that $9cn/10 + n \leq cn$, which is to say that $9c/10 + 1 \leq cn$, or that $1 \leq c/10$. Simply picking any $c \geq 10$ suffices, and the induction goes through.
Now, $T(n)$ is trivially $\Omega(n)$, since the definition of $T(n)$ shows that $T(n) \geq n$. Therefore $T(n)$ is $\Theta(n)$.$\Box$