Eric Bronner, Aedan Dispenza, Jason Davis, Timothy Yong
CS513 - Dr. Farach-Colton

Homework 2

Problem 1

Algorithm:

1. Sort the first k elements with merge sort
2. Perform insertion sort on previous $k-1$ elements based on binary search
   $k-1$, because, once the first k elements are sorted,
   the first element must be in the correct spot.
   We ignore one more element per each iteration.

Proof of Correctness by Contradiction:

Assume $\exists a \in A$, where A is the k-sorted array post-sort, s.t. a is not sorted.
Case 1: a is within range of the previous $k-1$ elements
Either merge sort or insertion sort must have failed.
This is a contradiction, as merge sort and insertion sort do not fail.
Case 2: a is not within k elements of where it started
This is a contradiction, as this means that a was not k-sorted to begin
Therefore, a cannot exist, and the algorithm must work $\square$

Running-Time Analysis:

Merge sorting the first k elements will take $klogk$ time
Binary search-based insertion sort for $n-k$ elements will take
$(n-k)log(k-1)$ time
In total, the running time is $klogk + nlogk - klog(k-1)$
Therefore, the running time is $O(nlogk)$, as $n \geq k$

Problem 2

Suppose we have a k-sorted array A
For every element $a \in A$ there are k possible placements in n total bins
So the number of posssibilities is $\prod_{i=0}^{n} k = k^n$
If we take the logarithm of this (because of the decision tree), we get:
$log(k^n) = nlog(k)$
So it's $\Omega(nlog(k))$

Problem 3

a) Suppose we have two subtrees, l and h of a tree T, where l is the lighter subtree, h is the heavier subtree and $w(h) = 2 \times w(l) + 1$. The number of nodes in the tree is n, so there are $n-3$ nodes beneath the root and its children.

$$w(h) = 2w(l) + 1$$
$$n - 3 = 2w(l) + 1 + w(l)$$
$$n - 3 = 3w(l) + 1$$
$$\frac{n-4}{3} = w(l)$$

So this means that $w(l) = \frac{n-4}{3}$ and $w(h) = \frac{2n-8}{3} + 1$

Thus, $h(h) = log(\frac{2n-8}{3} + 1)$ and $h(l) = log(\frac{n-4}{3})$

So, we have $O(logn)$ time to search both subtrees.

Adding the 3 removed nodes only adds a constant factor.

b) Suppose we have a tree T, s.t. T is the bare minimum tree that maintains the balance property, without the two subtrees being of equal weight.

If we add another node to h, then we would rebuild the subtree all the way up to the root via AVL rotation.

This rotation takes $O(log(n))$ time due to T having $log(n)$ levels.

c) 1 or 2 insertions are required to unbalance a freshly balanced tree.

It is 1 insertion if one subtree is complete, 2 in any other case.

d) The probability of unbalancing the tree in the worst case scenario is $\frac{1}{n}log(n)$

The number of nodes we need to add unbalance the tree again after rebalancing is $\sum_{i=0}^{log(n)} \frac{n}{2^i} = 2n$

So, the worst case cost of unbalancing and rebalancing the tree is $2nlog(n)$.

The cost to insert all nodes that do not lead to a worst case is $n - log(n)$.

So, the cost of n insertions is $nlogn + (n - log(n))$, or $O(nlog(n))$.

e) The amortized cost is $\frac{nlog(n)}{n} = O(logn)$

<u>Problem 4</u>

<u>Proof by Induction:</u>

Base: $\exists$ only one column

The row is sorted by definition and default, as it's length is 1

So, if we sort vertically, nothing can change.

Inductive Hypothesis:

Assume that for some number of sorted columns $\leq n$, we can maintain horizontal sort after vertically sorting.

Inductive Step:

Add a new column (the $n + 1^{th}$), keeping the rows sorted

Sort the new column; compare the new element at position i to the element that was previously there.

Case 1: Element is the same

Nothing has changed; the row must still be sorted.

Case 2: Position i now holds a greater value
Because the rows up to column n are still sorted, and
the value in position i has only increased, the row must be sorted
Case 3: Position i now holds a smaller value, x
x originated elsewhere in the column
x must belong at position i, proven by contradiction:
Assume x is less than the value in the column next to it, same row
If try to resort x by moving it up in the column, two cases emerge:
Case 1: x is always less than the value in the before it
This is a contradiction, as x doesn't belong in this column,
since the rows were originally sorted
Case 2: We find a new place for x
We must have displaced another element, y s.t. $y < x$
But, because y is less than everything below it,
as we've already sorted the column,
we have a new out-of-place element
Therefore, we have a contradiction $\square$

Problem 5

Say we have a graph G
We will call HamP on every pair of nodes in G.
If HamP returns true for any pair of nodes and those two nodes are connected, then we have found a Hamiltonian Cycle
This process will take $O(n^2) \in P$

Problem 6

Say we have a graph G
If we call HamP on every of nodes in G, we will either find a Hamiltonian Path or find that there is none.
This algorithm will take $\frac{n(n-1)}{2}$ or $O(n^2) \in P$