Midterm
Introduction to Algorithms
601.433/633

Thursday, October 26th, 1:30-2:45pm

Name:

Section (circle one): undergraduate graduate

Ethics Statement
I agree to complete this exam without unauthorized assistance from any person, materials, or device.

Signature: Date:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Out of</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>2 a</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>b 12</td>
</tr>
<tr>
<td>3 a</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>b 9</td>
</tr>
<tr>
<td></td>
<td>c 7</td>
</tr>
<tr>
<td>4 a</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>b 12</td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
</tr>
</tbody>
</table>
1 Problem 1 (25 points): Preliminaries

Indicate whether each statement is true or false. You do not need to give a proof, nor do you need to give counterexamples.

(a) For the Union-Find problem, if we use union by rank and path compression, then the rank stored in each node is always the height of this node.
true   false

(b) Let $H$ be a universal family of hash functions. Then for any $h \in H$ and any sequence of $L$ insert, delete, and lookup operations, the total running time of those operations is at most $O(L)$.
true   false

(c) $\log \log n = O(\log^* n)$
true   false

(d) Let $T(n) = 3T(n/5) + 2n$. Then $T(n) = \Theta(n)$.
true   false

(e) The worst-case running time of Randomized Quicksort is $O(n \log n)$.
true   false

(f) Every leaf of a red-black tree has the same black-depth.
true   false

(g) The amortized cost of an Insert operation in a binomial heap is $O(1)$
true   false

(h) Let $H$ be a universal family of hash functions from a universe $U$ onto a table of size $m$. Then $\Pr_{h \sim H}[h(e) = i] = 1/m$ for all elements $e \in U$ and table positions $i \in \{1, 2, \ldots, m\}$.
true   false

(i) Least-significant first radix sort is correct no matter which sorting algorithm we use on each digit.
true   false

(j) There is an algorithm which can find the $n^{1/10}$-smallest number in an unsorted array of $n$ numbers in $O(n)$ time.
true   false
Problem 2 (25 points): Searching / Sorting

(a) (13 points) Let $A$ be an array of $n$ distinct integers (not necessarily positive) where $A$ is already sorted in ascending order. Our problem is to find an index $i, 1 \leq i \leq n$, such that $A[i] = i$ or determine that no such $i$ exists. Prove that any comparison-based algorithm for this problem must have $\Omega(\log n)$ worst-case running time.

In this context, “comparison-based” means that the algorithm is only allowed to access $A$ through queries that compare an element of $A$ to an arbitrary integer (since $A$ is already sorted there is no need to compare two elements of $A$). These queries are either $<$, $=$, or $>$. So, for example, queries such as “is $A[i] < k$?” or “is $A[i] = j$?” are allowed.

Solution: First, note that if there is a valid index, it can be in any of the $n$ locations of $A$. Each comparison we do rules out some locations while others are still possible, so we can think of any comparison based algorithm as a decision tree with at least $n$ leaves where each internal node is labeled with a particular query. Since each comparison is binary there are two possible outcomes of any comparison, and so the decision tree is a binary tree and thus has height at least $\log n$. Hence any comparison-based algorithm has worst-case running time of at least $\Omega(\log n)$. 
(b) (12 points) Suppose that you are given an unsorted array of $n$ elements (not necessarily distinct) which are comparable. Let’s assume that $n$ is odd. You want to know whether there is a majority element, i.e., an element that appears in the array at least $\lceil n/2 \rceil$ times. Design an $O(n)$ time algorithm for this problem. Argue correctness and running time, but formal proofs are not necessary.

**Solution:** Run the deterministic linear-time median-finding algorithm from class (BPFRT). If this returns element $e$, then scan through the array to count how many times $e$ appears. If $e$ appears at least $\lceil n/2 \rceil$ times, then return yes, otherwise return no.

The running time is obvious: we make one call to BPFRT, which takes $O(n)$ time, and then do a single scan through the array while keeping track of a counter (also $O(n)$ time). Thus the total time is $O(n)$.

For correctness, suppose that there is a majority element. Then clearly the median element of the array will be that element, and hence BPFRT will return it, and our scan will confirm that it is a majority element and so we will return yes. On the other hand, if there is no majority element, then whatever BPFRT returns, when we do the scan we will find out that it is not a majority element, and so will return no.
3 Problem 3 (25 points): Amortized Analysis

Suppose that we want to implement a stack as an array, for example if we want to also have read-access to elements in the middle rather than just being able to pop out the element on top. We saw in class that if we double the size of the array when it is full, then the amortized cost of an insert is still only $O(1)$. In other words, $n$ inserts only take time $O(n)$ even though a single insert might take $\Omega(n)$ time (to create a new array of twice the size and copy over all of the elements).

What if we add in the ability to pop from the stack, though? Pops can be implemented quickly, but we might end up in the undesirable situation of having an array that is much, much bigger than the size of the stack. For example, if we do $n$ pushes and $n - 1$ pops, then the size of the array will be $\Theta(n)$ even though there is only one element in the stack!

To fix this, consider the following algorithm. Let $D$ be the current array, and suppose that $\alpha$ positions of the array have a stack element. So if $\alpha = |D|$ then the array is full, and if $\alpha = 0$ then the array is empty. As before, if the array is full then when we push a new element we double its size. Let’s say that this takes time exactly $|D| + 1 = \alpha + 1$, since it takes $|D| = \alpha$ time to copy the elements and 1 to push the new element. On the other hand, if after a pop the array is less than $1/3$ full then we contract the array by making a new array of $2/3$ the size and copying all of the elements. This also takes time $\alpha + 1$ (to pop a single element and then copy the rest), but note that in this case $\alpha = |D|/3$ rather than $|D|$.

Consider the potential function $\Phi = |2\alpha - |D||$.

(a) (9 points) Prove that on any push, the amortized cost (when using the above potential function) is at most $O(1)$.

Solution: If we do a push and do not expand the array, the amortized cost is $1 + \Delta\Phi \leq 3$. (the change in potential will be 2 if $\alpha$ was initially at least $|D|/2$, and will be $-2$ if $\alpha$ was initially less than $|D|/2$). If we do a push that does cause the array to expand, then it must be the case that before the push the array was full, so $\alpha = |D|$ and thus $\Phi = \alpha$. Then since the expansion double the array, after the expansion we have that $\alpha = |D|/2$ and thus $\Phi = 0$. So the amortized cost is $\alpha + 1 + \Delta\Phi = 1$. 

7
(b) (9 points) Prove that on any pop, the amortized cost (when using the above potential function) is at most $O(1)$.

**Solution:** If we do a pop but do not contract the array then the amortized cost will be $1 + \Delta \Phi \leq 3$. If we do a pop that causes the array to contract then initially $\alpha = |D|/3$ and thus $\Phi = \alpha$, and after the contraction $\alpha = |D|/2$ so $\Phi = 0$. Thus the amortized cost is $\alpha + 1 + \Delta \Phi = 1$.

(c) (7 points) Using the previous parts, prove that for every sequence of $n$ operations (pushes and pops) the total running time is $O(n)$. You may assume initially $|D| = 2$ and there is one element in the array.

**Solution:** We know by definition that the total running time is equal to the total amortized time plus $\Phi_{\text{init}} - \Phi_{\text{final}}$. Since $\Phi_{\text{init}} = 0$ and $\Phi_{\text{final}} \geq 0$, this implies that total time of $n$ operations is at most the total amortized time. By the previous two parts, the total amortized time is $O(n)$, and hence the total time is $O(n)$. 


4 Problem 4 (25 points): Dynamic Programming

Let \( T = (V, E) \) be a binary tree rooted at a node \( r \in V \). Let \( w : V \to \mathbb{R}^+ \) be an assignment of nonnegative weights to the nodes. Assume that every node \( v \) has pointers \( v.l \) and \( v.r \) to its children. A vertex cover of \( T \) is a set \( A \subseteq V \) such that for every edge, at least one of its two endpoints is in \( A \). In other words, for every (parent, child) pair, either the parent or child or both is in \( A \). In this problem you will design an algorithm to find the minimum-weight vertex cover, i.e., the vertex cover \( A \) which minimizes \( \sum_{v \in A} w(v) \).

(a) (13 points) For each vertex \( v \in V \), let \( OPT(v) \) be the weight of the minimum-weight vertex cover for the subtree rooted at \( v \) (i.e., the subtree consisting of \( v \) and all of its descendants). So, for example, \( OPT(v) = 0 \) if \( v \) is a leaf since there are no edges in the subtree rooted at \( v \).

Give a formula for \( OPT(v) \) in terms of \( OPT(v.l), OPT(v.r), OPT(v.l.l), OPT(v.l.r), OPT(v.r.l), \) and \( OPT(v.r.r) \) (and weights of nodes). To make notation easier, you can assume that dereferencing a NULL pointer just returns another NULL pointer.

You do not need a formal proof, but you should justify your answer.

Hint: what is \( OPT(v) \) if \( v \) is included in the solution of cost \( OPT(v) \), and what is \( OPT(v) \) if \( v \) is not included in the solution of cost \( OPT(v) \)?

Solution: Clearly \( OPT(v) = 0 \) if \( v \) is a leaf or is NULL. Otherwise, if \( v \) is included in the optimal solution for subproblem \( v \), then \( OPT(v) = w(v) + OPT(v.l) + OPT(v.r) \), since \( v \) covers the edges to its two children. On the other hand, if \( v \) is not included in the optimal solution for subproblem \( v \), then both of the children of \( v \) must be included in order to cover their edges to \( v \). Thus in this case we get that \( OPT(v) = w(v.l) + w(v.r) + OPT(v.l.l) + OPT(v.l.r) + OPT(v.r.l) + OPT(v.r.r) \). So

\[
OPT(v) = \min(w(v) + OPT(v.l) + OPT(v.r),
\quad w(v.l) + w(v.r) + OPT(v.l.l) + OPT(v.l.r) + OPT(v.r.l) + OPT(v.r.r))
\]
(b) (12 points) Design an $O(n)$ time algorithm to compute the weight of the minimum-weight vertex cover, given a pointer to the root $r$ of the tree. Argue correctness and running time (you do not need a fully formal proof, but do need to provide justification). Hint: use the previous part.

**Solution:** First, set $M[v] = -1$ for all vertices $v$. Now consider the following algorithm, and initially call $\text{Alg}(r)$.

```c
Alg(v) {
    if(v == NULL) return 0;
    if(M[v] != -1) return M[v];
    Set M[v] = min(w(v) + Alg(v.l) + Alg(v.r),
                  w(v.l) + w(v.r) + Alg(v.l.l) + Alg(v.l.r) + Alg(v.r.l) + Alg(v.r.r))
    return M[v];
}
```

This is just a top-down dynamic programming algorithm. To see correctness, note that it is just computing $OPT(v)$ for all $v$ using the formula from part a. Formally, we could prove this by induction on the height of the node – for the base case it is true at the leaves, and for the inductive step if it true at the children and grandchildren of a node $v$ then it is also true at $v$ because of part a. To see the running time, note that the running time of $\text{Alg}(v)$ excluding recursive calls is clearly $O(1)$. So the running time is just $O$(number of recursive calls). But we only make a recursive call when filling in a table entry (we in fact make six recursive calls), and each table entry only gets filled in once. Hence the total number of recursive calls is at most $O(n)$, and thus the running time is $O(n)$. 