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</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>13</td>
</tr>
<tr>
<td>b</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>9</td>
</tr>
<tr>
<td>b</td>
<td>9</td>
</tr>
<tr>
<td>c</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>13</td>
</tr>
<tr>
<td>b</td>
<td>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
2 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.
(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.
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 $O(1)$. 


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

(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.
4  Problem 4 (25 points): Dynamic Programming

Let $T = (V, E)$ be a binary tree rooted at a node $r \in V$. Let $w : V \rightarrow \mathbb{R}^+$ 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 $OPT(v)$?
(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.