*EE360C: Algorithms*
The University of Texas at Austin — Exam #2 Practice Questions
Dr. Pedro Santacruz & Dr. Evdokia Nikolova — October 30, 2014

**Name:** **EID:**

# Exam #2 Practice Questions

**Instructions.** No calculators, laptops, or other devices are allowed. This exam is **closed book**, but you are allowed to use a **one-page** cheat sheet. You must submit your cheat sheet with the exam. Write your answers on the test pages. If you need scratch paper, use the back of the test pages, but indicate where your answers are. Write down your process for solving questions and intermediate answers that **may** earn you partial credit.

If you are unsure of the meaning of a specific test question, write down your assumptions and proceed to answer the question on that basis. **Questions about the meaning of an exam question will not be answered during the test.**

You have **75 minutes** to complete the exam. The maximum possible score is 100.

Some useful information:

Logarithms and Factorial:

$$\log(n!) = \Theta(n \log n)$$

Arithmetic Series:

$$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1)$$

Sum of Squares:

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

Sum of Cubes:

$$\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4}$$

Geometric Series:

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

Infinite Geometric Series:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

## Practice Problem 1: Minimum Spanning Trees

Suppose you are given a connected graph $G = (V, E)$, with edge costs that you may assume are all distinct. Given a particular edge $e \in E$, give an algorithm with $O(|V| + |E|)$ running time to decide whether $e$ is contained in a minimum spanning tree of $G$.

**Solution**

Lets say the edge $e$ connects two nodes $u$ and $v$. Form a graph $G'$ by deleting all of the edges from $G$ that have a weight greater than $e$s weight. Delete $e$, too. This takes $O(2|E|)$ time. Build a breadth first search tree rooted at $u$. This takes $O(|V| + |E|)$ time. If this generates a path from $u$ to $v$, then $e$ does not belong to any minimum spanning tree. If no path exists, then $e$ does belong to some minimum spanning tree.

Heres the proof that makes this work:

*Edge $e = (v, w)$ does not belong to a minimum spanning tree of $G$ if and only if $v$ and $w$ can be joined by a path consisting entirely of edges that are cheaper than $e$.*

First, suppose that $P$ is a $v - w$ path consisting entirely of edges cheaper than $e$. If we add $e$ to $P$, we get a cycle on which $e$ is the most expensive edge (dangerous edge). One of the other edges $P$ must also cross the same cut that $e$ does, and it has a lower cost than $e$, so if we had a spanning tree that contained $e$, we could replace $e$ with this other edge and get a lower cost tree. So the tree containing $e$ is not minimum. On the other hand, suppose that $v$ and $w$ cannot be joined by a path consisting entirely of edges cheaper than $e$. We identify a set $S$ for which $e$ is the cheapest edge with one end in $S$ and the other in $V - S$, making $e$ a sure bet for every minimum spanning tree. $S$ contains all of the nodes that are reachable from $V$ using a path consisting only of edges that are cheaper than $e$. Our assumption implies $w \in V - S$. By the definition of $S$, there cannot be an edge $f = (x, y)$ that is cheaper than $e$ and for which $x \in S$ and $y \in V - S$. If there were such an $f$, then since $x$ is reachable from $v$ using only edges cheaper than $e$ (by definition of $S$), $y$ would be reachable as well. Therefore, $e$ is the cheapest edge with one end in $S$ and the other end in $V - S$ and is the choice for crossing the cut.

## Practice Problem 2: Binary Search Trees

Consider the problem of building a binary search tree $T$. In this tree, the root is identified as $T.root$ and each node $x$ has a key value $x.key$, a right child $x.right$, and a left child $x.left$. Nodes with equal keys pose a problem for the implementation of binary search trees. Consider the following algorithm that adds a new node $z$ to an existing binary search tree $T$:

```
TREE-INSERT($T, z$)
 1  $y \leftarrow$ NIL
 2  $x \leftarrow T.root$
 3  while $x \neq$ NIL
 4        do $y \leftarrow x$
 5           if $z.key < x.key$
 6              then $x \leftarrow x.left$
 7              else $x \leftarrow x.right$
 8  $z.p \leftarrow y$
 9  if $y =$ NIL
10     then $T.root \leftarrow z$
11     else if $z.key < y.key$
12              then $y.left \leftarrow z$
13              else $y.right \leftarrow z$
```

**a)** What is the asymptotic performance of TREE-INSERT when used to insert $n$ items with identical keys into an initially empty binary search tree?

**Solution**

$O(n^2)$

**b)** We propose to improve TREE-INSERT by testing before line 5 whether or not $z.key = x.key$ and by testing before line 11 whether or not $z.key = x.key$. If equality holds, we implement one of the three following strategies. For each strategy, find the asymptotic performance of inserting $n$ items with identical keys into an initially empty binary search tree. (The psuedocode for TREE-INSERT is given above for reference; the strategies are described for line 5, in which we compare the keys for $z$ and $x$; substitute $y$ for $x$ to arrive at the strategies for line 11.)

**Strategy 1:** Keep a boolean flag $x.b$ at node $x$ and set $x$ to either $x.left$ or $x.right$ based on the value of $x.b$, which alternates between FALSE and TRUE each time $x$ is visited during insertion of a node with the same key as $x$.

**Solution**

$O(n \lg n)$

**Strategy 2:** Keep a list of nodes with equal keys at $x$, and insert $z$ into the list.

**Solution**

$O(n)$

**Strategy 3:** Randomly set $x$ to either $x.left$ or $x.right$. (For this strategy, give the worst-case performance and informally derive the average-case performance.)

**Solution**

Worst case is unluckily picking the same every time: $O(n^2)$. Average case is an equal mix of the two: $O(n \lg n)$.

## Practice Problem 3: Breadth-First Search and Depth-First Search

We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$ . Suppose we compute a depth-first search tree rooted at $u$ and obtain a tree $T$ that includes all nodes of $G$. Suppose we then compute a breadth-first search tree rooted at $u$ and obtain the same tree $T$. Prove that $G = T$. (In other words, if $T$ is both a depth-first search tree and a breadth-first search tree rooted at $u$, then $G$ cannot contain any edges that do not belong to $T$.)

**Solution**

Suppose that $G$ has an edge $e = (a, b)$ that does not belong to $T$. Since $T$ is a depth first search tree, one of the two ends must be an ancestor of the othersay $a$ is an ancestor of $b$. Since $T$ is a breadth-first search tree, the distance of the two nodes from $u$ in $T$ can differ by at most one. But if $a$ is an ancestor of $b$ and the distance from $u$ to $b$ in $T$ is at most one greater than the distance from $u$ to $a$, then $a$ must in fact be the direct parent of $b$ in $T$. From this it follows that $(a, b)$ is an edge of $T$, contradicting our initial assumption that $e = (a, b)$ did not belong to $T$.

## Practice Problem 4: Topological Sort

The following algorithm relies on the fact that the *first* node in the order of a topological sort must have no incoming edges. This algorithm outputs nodes in a topologically sorted order.

```
TOPOLOGICAL-SORT($G$)
1  select $v \in G.V$ with no incoming edges
2  output $v$
3  $G.V = G.V - v$
4  TOPOLOGICAL-SORT($G$)
```

In general, TOPOLOGICAL-SORT assumes that $G$ is a directed acyclic graph (a DAG). Suppose were given an arbitrary directed graph that may or may not be a DAG. Extend the topological sort algorithm so that, given an input directed graph $G$, it outputs one of two things: (a) a topological ordering, thus establishing that $G$ is a DAG; or (b) a cycle in $G$, thus establishing that $G$ is not a DAG. The running time of your algorithm should be $O(|V| + |E|)$ for a directed graph $G = (V, E)$. (Hint: If youre stuck, try running a couple of instances of this topological sort algorithm to see how its working.)

**Solution**

We use the above algorithm with a small modification. If in every iteration, we find a node with no incoming edges, then we will succeed in producing a topological ordering. If in some iteration, it transpires that every node has at least one incoming edge, then the graph must contain a cycle. So our algorithm to find the cycle is this: repeatedly follow an edge into the node were currently at (choosing the first one on the adjacency list of incoming edges to reduce the running time). Since every node has an incoming edge, we can do this repeatedly until we revisit a node $v$ for the first time. The set of nodes encountered between these two successive visits is a cycle (traversed in the reverse direction).

Scratch Page