# CSCE 586 HW 2

Marvin Newlin

18 October 2018

Collaborators: Zach Madison, Sean Mochocki, Kyril Sarantsev

1. **Chapter 3 Problem 5:** A binary tree is a rooted tree in which each node has at most two children. Show by induction that in any binary tree the number of nodes with two children is exactly one less than the number of leaves.

   <u>**Solution:**</u>
   Note: In this solution, when we refer to a $k$-node binary tree we are saying that the number of nodes with two children is $k$.

   **Base Case:** The base case is that we have a single node with two children. In this case we have the number of nodes, $k = 1$. With a 1-node binary tree we have 1 node and 2 children and $1 = 2 - 1$ so we have exactly one less node than leaves. Thus, the statement holds for $k = 1$.

   **Inductive Hypothesis:** Now, assume that for some $k \in \mathbf{N}$, a $k$-node binary tree has $k + 1$ leaf nodes.

   **Inductive Hypothesis:** Given a $k + 1$-node binary tree, we can view this tree as a construction of a $k$-node binary tree and a 1-node binary tree as follows: Replace any leaf node of the $k$-node tree with the 1-node tree. The result is clearly still a binary tree, and we now have $k + 1$-node binary tree. Originally, our $k$-node tree had $k + 1$ children and our 1-node tree had 2 children. Replacing the one leaf node removed one leaf node but added the two from our 1-node tree so our total number of leaves is now $k + 1 - 1 + 2 = k + 0 + 2 = k + 1 + 1 = (k + 1) + 1$.
   Thus, we have exactly one less node than leaves in our $k + 1$-node tree.
   Therefore, by the Principle of Mathematical Induction, in a binary tree, the number of nodes with two children is exactly one less than the number of leaves. ■

OK 25/25

2. **Chapter 4 Problem 4:** Some of your friends have gotten into the burgeoning field of *time-series data mining*, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges—what's being bought— are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain "patterns" in them—for example, they may want to know if the four events

   buy Yahoo, buy eBay, buy Yahoo, buy Oracle

   occur in this sequence $S$, in order but not necessarily consecutively.
   They begin with a collection of possible events (e.g., the possible transactions) and a sequence $S$ of $n$ of these events. A given event may occur multiple times in $S$ (e.g., Yahoo stock may be bought many times in a single sequence $S$). We will say that a sequence $S'$ is a subsequence of $S$ if there is a way to delete certain of the events from $S$ so that the remaining events, in order, are equal to the sequence $S'$. So, for example, the sequence of four events above is a subsequence of the sequence

   buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle

   Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of $S$.
   So this is the problem they pose to you: Give an algorithm that takes two sequences of events—$S'$ of length $m$ and $S$ of length $n$, each possibly containing an event more than once—and decides in time $O(m + n)$ whether $S'$ is a subsequence of $S$.

   **Solution:** Since we need a solution in $O(n+m)$ time, we can see that we need to basically only search through each list one time. Our algorithm does that simultaneously as demonstrated in the description.

   **Description:**
   Assumption: $n \geq m$ (if it is not then the subsequence cannot be contained in the sequence)

```
NOTE: The size of sequence is n and the size of subsequence is m

Algorithm: FindPattern (sequence, subsequence)
    subCounter <- 0
    seqCounter <- 0
    WHILE (seqCounter < n AND subCounter < m)
        IF (sequence[seqCounter] == subsequence[subCounter])
            subCounter <- subCounter + 1
        End IF
        seqCounter <- seqCounter + 1
    END WHILE
    IF (subCounter == m)
        return TRUE
    ELSE
        return FALSE
```

On

**Proof of Correctness:**

Termination: The loop will always terminate because at every iteration of the while loop, the counter that maintains the position we are at in the sequence is incremented so after $n$ iterations (seqCounter not less than $n$), the condition of the while loop will be false and the loop will terminate.

25/25

Output Correctness: Suppose that the output of our algorithm was wrong. There are two cases here.

Case 1: The algorithm output true but the subsequence was not actually contained in the sequence.
Case 2: The algorithm output false but the subsequence was actually contained in the sequence.

Proof: Assume that we have case 1 and the subsequence was not contained in the sequence. Then, since our algorithm returned true, we matched every element in the subsequence to an element of the original sequence. Therefore, the subsequence was actually contained in the sequence which is a contradiction. Therefore, case 1 cannot occur.
Now, assume that case 2 has occurred and our algorithm returned false even though the sequence does contain the subsequence. The only way for the algorithm to return false is to not have completely traversed the subsequence. But since we returned false then the loop terminated while subCounter $< m$. Thus, seqCounter $= n$ and so we have that $n < m$, meaning that the sequence is shorter than the subsequence which is a violation of our initial assumption since a subsequence cannot be longer than its containing sequence. Therfore, we have a contradiction so case 2 cannot occur.
Since neither wrong output can occur, then our algorithm outputs correctly whether a subsequence is contained in a given sequence.

**Runtime Analysis:**

Worst/Average Case: At worst, the last element of the subsequence is also the last element of the subsequence and so we have to iterate through the entirety of both the sequence and subsequence. In this case, the runtime is $O(n + m)$.
On average, let's assume that that the subsequence is contained in the sequence so then our runtime will be $\Theta(m + cn), c \in \mathbf{R}[0, 1]$, where $cn$ is the amount of the sequence we had to iterate through to find all of the subsequence.

Best Case: The best case is where the subsequence is one element and then we are only searching the original sequence for one item so our runtime is $O(n)$.

3. **Chapter 4 Problem 19:** A group of network designers at the communications company CluNet find themselves facing the following problem. They have a connected graph $G = (V, E)$, in which the nodes represent sites that want to communicate.Each edge $e$ is a communication link, with a given available bandwidth $b_e$.

   For each pair of nodes $u, v \in V$, they want to select a single $u - v$ path $P$ on which this pair will communicate. The bottleneck rate $b(P)$ of this path $P$ is the minimum bandwidth of any edge it contains; that is, $b(P) = \min_{e \in P} be$. The best achievable bottleneck rate for the pair $u, v$ in $G$ is simply the maximum, over all $u - v$ paths $P$ in $G$, of the value $b(P)$.

   **Solution:**
   Given that we need to find a spanning tree in which we maximize the bandwidth for each path leads us to believe that this is a Minimum Spanning Tree (MST) problem. Even though we are trying to maximize a quantity, the point is still the same. We picked Prim's algorithm to modify in order to provide a Minimum (maximum bandwidth) Spanning Tree. Additionally, we can assume that all nodes on the network are connected.

   **Description:**
   The algorithm we choose here is exactly Prim's algorithm described on page 143 of the Kleinberg and Tardos textbook, with the following modification.

   Instead of selecting the edge with the minimum bandwidth, we select the edge with the maximum bandwidth that connects the two subsets of vertices (Kleingberg, Tardos 143).

   **Proof of Correctness:**
   We know based on the proof of Prim's algorithm that it outputs a MST

on

25/25

(Kleinberg, Tardos 4.19). Since we have modified it now, we need to verify that we output a Maximum Spanning Tree. In this case, maximum means the greatest amount of bandwidth.
To prove that this is a maximum spanning tree, we can just run the original Prim's algorithm on the graph $G' = (V, E')$ where $E'$ has the reciprocal bandwidths of each edge in $E$. Since Prim's will select the minimum cost edge $e' \in E'$, the result is a minimum spanning tree $T'$ and taking the reciprocal of each of the edge weights in $T'$ will yield the maximum spanning tree since reciprocating the minimum value yields the maximum value at each step.

**Runtime Analysis:**
Since this is basically Prim's algorithm, we know that it will have the same run times. Here, we assume that the algorithm is implemented with a heap and adjacency list priority queue. Additionally, we assume the graph has $m$ edges and $n$ vertices.

Best/Average/Worst Case: The runtime for Prim's algorithm is $O(m \log n)$. Obviously, our modification can take place in constant time so this is indeed the all case runtime for our algorithm. This is both the average and worst case because in general we have to iterate through all the vertices and edges.

4. **References**
Kleinberg, Jon, and Tardos Eva. Algorithm Design. 1st ed., Pearson, 2006.