COMPSCI 250: Fall 2023
Homework 4

David A. Mix Barrington , Ghazaleh Parvini

Due Date : Friday, November 3

(12 points) **Problem 4.4.7**

A polygon is called **convex** if every line segment from one vertex to another lies entirely within the polygon. To **triangulate** a polygon, we take some of these line segments, which don't cross one another, and use them to divide the polygon into triangles. Prove, by strong induction for all naturals n with $n \geq 3$, that every convex polygon with n sides has a triangulation, and that every triangulation contains exactly $n-2$ triangles. (**Hint:** When you divide an n-gon with a single line segment, you create an i-gon and a j-gon for some naturals i and j. What does your strong inductive hypothesis tell you about triangulations of these polygons?)

Solution:
Base: when $n = 3$, there is only one triangle which is the polygon itself.
IH: Let $Q(n)$ mean that there are $n-2$ triangles in the triangulation of convex polygon with n or less sides. Assume $Q(n)$ is true.
IS: For a convex polygon with $n+1$ sides, we can use a line segment to connect two vertices of the polygon and divide the polygon into a triangle and a convex polygon with n sides. Thus, when triangulating the convex with n sides, there are $n-2$ triangles, plus the triangle from the $n+1$ sides polygon, there are $n-2+1$, i.e.$n+1-2$ triangles.

Solution (Strong Induction):
Base: when $n = 3$, there is only one triangle which is the polygon itself.
IH: Let $Q(k)$ mean that there are $k-2$ triangles in the triangulation of convex polygon with k sides. Assume $Q(k)$ is true for all $3 \leq k \leq n$.
IS: For a convex polygon with $k+1$ sides, we can use a line segment to connect two vertices of the polygon and divide the polygon into two smaller convex polygons with i and j sides. Here, $i + j = k + 3$ sides, since there were $k+1$ original sides, plus the two created by the dividing line segment. Since each i-gon and j-gon has at least 3 and no more than k sides, each can be triangulated by the IH into a triangulation with $i-2$ and $j-2$ triangles, respectively. Reconnecting the i and j-gons gives us a triangulated polygon with $k+1$ sides and $(i-2)+(j-2) = k+3-4 = (k+1)-2$ triangles.

(12 points) **Problem 4.7.6**

(uses Java) We can define the **balanced parenthesis language** using recursion. This is the set of sequences of left and right parentheses that are balanced, in that every left paren has a matching right paren and the pairs are nested properly. We'll use "L" and "R" instead of "(" and ")" for readability.

We define the language Paren by the following four rules:

(a) λ is in Paren.

(b) If u is in Paren, then so is LuR.

(c) If u and v are in Paren, then so is uv.

(d) No other strings are in Paren

Write a real-Java static method (or a Python function) `isBalanced` that takes a `String` argument and returns a `boolean` telling whether the input string is in Paren. A non-recursive method is simpler.

Solution:

Java:

```
public static boolean isBalanced(String str) {
    int paren = 0;
    for(int i = 0; i < str.length(); i++) {
        if(str.charAt(i) == '(') {
            paren++;
        }
        else if (str.charAt(i) == ')') {
            paren--;
        }
        else {
            return false;
        }
        if(paren < 0) {
            return false;
        }
    }
    return (paren == 0);
}
```

Python:

```
def isBalanced(str):
    paren = 0
    for i in range(0, len(str)):
        if str[i] == '(':
```

```
            paren++
        elif str[i] == ')':
            paren--
        else:
            return false
        if paren < 0:
            return false
    return (paren == 0)
```

Solution (Recursive):

Java:

```
public static boolean isBalanced(String str, int count, int pos) {
    if (count < 0) {
        return false;
    }
    if (pos >= str.length()) {
        return (count == 0);
    }
    if (str.charAt(pos) == '(') {
        return isBalanced(str, count + 1, pos + 1);
    }
    else if (str.charAt(pos) == ')') {
        return isBalanced(str, count - 1, pos + 1);
    }
    else {
        return false;
    }
}
```

Python:

```
def isBalanced(str, count, pos):
    if count < 0:
        return false
    if pos >= len(str):
        return count == 0;
    if str[pos] == '(':
        return isBalanced(str, count + 1, pos + 1)
    elif str[pos] == ')':
        return isBalanced(str, count - 1, pos + 1)
    else:
        return false
```

(10 points) **Problem 4.9.2**

Prove that any directed cycle in the graph of a partial order must only involve one node. (**Hint:** If the cycle were to contain two distinct nodes x and y, what does transitivity tell you about arcs between x and y?)

Solution: The edge relation E is a partial order, so it is reflexive, antisymmetric and transitive. Let u and v be different nodes that are involved in the same cycle. We can say that there is a path $P(u, v)$ and a path $P(v, u)$. Because E is transitive, and $P(u, v)$ and $P(v, u)$, we can conclude that $E(u, v)$ and $E(v, u)$. Contradiction because E is antisymmetric but $u \neq v$.

(10 points) **Problem 4.10.5**

Prove that if T is any rooted directed binary tree (where every internal node has out-degree exactly two), then the number of leaves in T is one greater than the number of internal nodes. (**Hint:** Use induction on the definition of such trees.)

Solution:

Base Case: In the case that there is only one node, we know that the number of leaves in T is one greater than the number of internal nodes because there is one leaf and zero internal nodes.

Inductive Step: Assume that a rooted directed binary tree with n internal nodes have n + 1 leaves. We want to prove that a rooted directed binary tree with n + 1 internal nodes have n + 2 leaves.

Proof: We can prove this by choosing one leaf in a rooted directed binary tree with n internal nodes and turning that into an internal node by adding two nodes that are its children. This would increase the number of internal nodes by one since we're just adding an internal node and not modifying any other nodes. This would also increase the number of leaves by one since the number of leaves after we turn one into an internal node is $n+1-1+2 = n+2$. We know that this can form any rooted directed binary tree with n + 1 internal nodes because out of the n + 1 internal nodes, we can choose all nodes except for one that is connected to leaves to make into internal nodes for a binary tree with n internal nodes. This finishes the inductive step and proves that the number of leaves in a rooted directed binary tree is one greater than the number of internal nodes.

(12 points) **Problem 4.11.1**

Show that a $3 \times n$ rectangle can be covered exactly with L-shaped tiles if and only if n is even. (**Hint:** For the negative result, use induction on all odd numbers and an indirect proof in the inductive step.)

Solution:

First, we show proof that if n is even ($n = 2k$) the board can be tiled.

Base Case: In the case that $n = 2 * 0 = 0$. The board is trivially tiled and if $n = 2$

we can tile the board with 2 L-shaped tiles.

Inductive Step: Assume $P(2k)$ as our IH, that is we can tile a board of the size 3 * 2k. We want to prove that we can tile a board of size $3 * 2(k+1)$. We can break the board up into a $3*2k$ board and a 3*2 board. Both of which we can tile separately and join the tilings.

Next, we prove that if n is odd ($n = 2k+1$), the board cannot be tiled.

Base Case: In the case that $k = 0$, $n + 2 * 0 + 1 = 1$. The board cannot be tiled because our tiles have a width of 2.

Inductive Step: Assume that $\neg P(2(k-1)+1)$. We want to prove that $\neg P(2k+1)$ We will use a proof by contradiction and three cases. We can denote each square as the tuple (i, j), which i is in $i \in 0, 1, 2$ and $j \in 0, 1, 2, , ...2k$. Assume $P(2k+1)$, then Square(0, 0) is covered by some L-shaped tile.

Case 1: It is the middle square of a tile that covers $(0, 1)$ and $(1, 0)$. Then $(2, 0)$ can only be covered by a tile that includes $(1, 1)$ and $(2, 1)$. The remaining tiles then cover the $3 * 2k - 1$ rectangle $0, 1, 2$ times $2, ..., 2k$, which contradicts the IH.

Case 2: $(0, 0)$ is covered by a tile that also covers $(0, 1)$ and $(1, 1)$. Then $(1, 0)$ can only be covered by a tile that covers $(2, 0)$ and $(2, 1)$, and the remaining tiles violate the IH as in Case 1

Case 3: : $(0, 0)$ is covered by a tile that also covers $(1, 0)$ and $(1, 1)$. Then $(2, 0)$ cannot be covered by an L-shaped tile at all, since such a tile would have to cover either $(1, 0)$ or $(1, 1)$. So this case is impossible. Therefore $\neg P(2k+1)$.

(12 points) **Problem 4.11.4**

Prove the claim at the end of the section about the Euclidean Algorithm and Fibonacci numbers. Specifically, prove that if positive naturals a and b are each at most $F(n)$, then the Euclidean Algorithm performs at most $n-2$ divisions. (You may assume that $n > 2$.) (It follows from this result that Fibonacci numbers are the worst case, but you may not use that fact to solve this problem!)

Solution: $P(n)$ is "if a and b are each at most $F(n)$, $EA(a, b)$ takes $\leq n-2$ divisions". We use strong induction to prove $P(n)$ for all n with $n > 2$.
Base cases: First base case is $P(3)$, where a and b are each $\leq F(3) = 2$, and there is at most one division because the remainder is guaranteed to be 0. Second base is $P(4)$, where a and b are each $\leq F(4) = 3$. The only case where the first division does not have remainder 0 is $a = 3$, $b = 2$, where on the second division we divide 2 by 1 and get remainder 0. So there are at most 2 divisions.
Inductive hypothesis: Assume $P(i)$ is true whenever $3 \leq i \leq n$, and that $n \geq 4$.
Inductive step: Let a and b each be at most $F(n+1)$.
Case 1: $b \leq F(n)$. In this case $a\%b$ is also $\leq F(n)$, so after the first division, by the IH, we

have $n-2$ more, for $n-1$ total.
Case 2: $b > F(n)$. In this case $a\%b$ is at most $F(n+1) - F(n) = F(n-1)$, so after two divisions both numbers will be at most $F(n-1)$. By the IH, we have $n-3$ more, for $n-1$ total.

(10 points) **Problem 9.1.5**

State and prove a theorem giving the maximum and minimum possible number of leaves in a rooted tree of depth d and degree k. Repeat for the maximum and minimum total number of nodes.

Solution: *Theorem 1.1* The maximum possible number of leaves in a rooted tree of depth d and degree k is k^d.
Proof: Let's have induction on d to prove that the maximum number of leaves in a rooted tree of depth d and degree k is k^d. Let's assume every internal node has k children. Base: when $d = 0$, k^d=k^0=1, which indicates the root node which is the only leaf. IH: Let's assume it is true for d. IS: If we add another level and make d=d+1, then there will be k^{d+1} leaves. This is true because the number of nodes before the last level is k^d (according to IH) and each of these k^d nodes will have k child nodes (as k is the degree); resulting in $k^d * k$=k^{d+1} number of leaves.

Theorem 1.2 The minimum possible number of leaves in a rooted tree of depth d and degree k is k.
Proof:

Theorem 2.1 The maximum possible number of nodes in a rooted tree of depth d and degree k is $\frac{k^{d+1}-1}{(k-1)}$.
Proof: If every internal node has k children, then in each level l, there will be k^l nodes (proof if Theorem 1.1). So, this results in total number of nodes as: k^0+k^1+k^2+...+k^d=$\frac{k^{d+1}}{(k-1)}$

Theorem 1.2 The minimum possible number of nodes in a rooted tree of depth d and degree k is $k + d$.
Proof:

(10 points) **Problem 9.3.5**

Let T be a parse tree for an expression with only unary and binary operators, and let m be the number of primitive elements in the expression. Prove that if $m > 1$, then there exists a node x of T such that the subtree rooted at x contains at least $m/3$ of the primitive elements and at most $2m/3$ of the primitive elements.

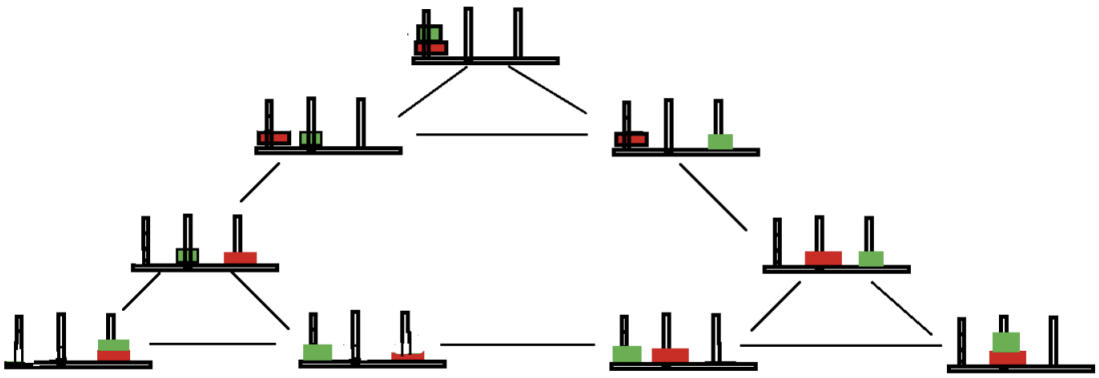
Solution: Notes: T_x stands for subtree rooted on node x, and l_x is the number of leaves in T_x. If we want to find node x such that the T_x contains at least $m/3$ primitive elements, or leaves, and at most $2m/3$ leaves, i.e., $m/3 \leq l_x \leq 2m/3$, we can proceed like this:
The root is not the node we are looking for, because the entire tree has m leaves. As we just

have unary and binary operators there are two possibilities. If the root has just one child x, T_x still with the same number of nodes and then we just make x the new root and try to find the node on T_x. The other possible case is when the root has two children x and y. We test if $m/3 \leq l_x \leq 2m/3$ is true. If it is then we found the node. If not, we can say that $l_x < m/3$ or $l_x > 2m/3$. Let's assume the first case w.l.o.g. In this case x is not our node and we need to test y. But, it must be the case that $m > ly > 2m/3$ so we need to look for our node on subtree T_y. Now suppose that it is possible that none node has the property. This implies that, at some point of the procedure we have the following situation: $l_x < m/3$ and $l_y < m/3$. But, if this happens, it means that the previous root had $m/3 \geq l_x + l_y < 2m/3$ and then the algorithm should have stopped at that moment.

(12 points) **Problem 9.4.7**

Following Exercise 9.4.9, describe the state graph of the Towers of Hanoi puzzle for general n. Prove that the puzzle is always solvable, and find the number of moves in the shortest possible solution. (It will be useful to have a recursive definition of the state space.)

Solution: For the Tower of Hanoi, we will be defining a move to be legal if the movement does not result in a larger disk to be on top of a smaller disk. The state space of The Tower of Hanoi for n = 2:



From the above diagram we can see that the the state space is an undirected graph. The state transitions are represented by straight lines. The start and the end transitions are represented by the topmost and the extreme vertices of the triangle. At every state(except the start and end) we have an option to go back to the parent state or two new states. Therefore, every time we add a new disk we triple the size of the state diagram. Thus, for a Tower with N disks we have 3N states. The state space is complete.As it is possible to move from any legal position to another it is solvable. In order to find the shortest path we can define a recursive definition : T(n + 1) = 2 * T(n) + 1 where T(n) represents the solution of Tower with N tiles.
Claim : The shortest possible solution : $2^n - 1$
Proof:

1. Inductive Hypothesis: The shortest possible solution for n tiles : $2^n - 1$
2. Inductive Goal: The shortest possible solution for n + 1 tiles : $2^{n+1} - 1$
3. Base Case: n = 1 : $2^1 - 1 = 1$. This holds as we just need one move to reach the final state.
4. Inductive Step: $T(n+1) = 2 * T(n) + 1 T(n+1) = 2 * (2^{n-1}) + 1 T(n+1) = 2^{n+1} - 2 + 1 = T(n+1) = 2^{n+1} - 1$

Extra Credit

(10 points) **Problem 9.5.8**

In the knight's tour problem, we are looking for a path of knight's moves from a square of the $n \times n$ chessboard to itself, such that the path visits all n^2 squares of the board. How can we modify BFS or DFS to solve this problem? (The solution may not be efficient, but it is correct.)

Solution: A modified BFS or DFS that solves the knight's tour problem would first start with a square on the board whose location is (i, j) and then process point (i, j) by pushing all points $(i \pm 2, j \pm 1)$ and $(i \pm 1, j \pm 2)$ that are located on the $n \times n$ board. For BFS we would process the board coordinates which were pushed earliest and for DFS we would process the board coordinates which were pushed latest. The path from our starting point and back covering all n^2 squares will be the order of the processed squares by our modified BFS or DFS algorithm.