COMP 250: Introduction to Computer Science
More sample problems

1. **Graph coloring**

   Determining if an undirected graph can be colored with only three colors is an NP-complete problem. However, determining if it can be colored with only two colors is much easier. Write the pseudocode of an algorithm that determines if the vertices of a given connected undirected graph can be colored with only two colors (named 0 and 1) so that no two adjacent vertices have the same color. Assume the graph has n vertices numbered 0,1, ... $n-1$. Use the following graph ADT methods:

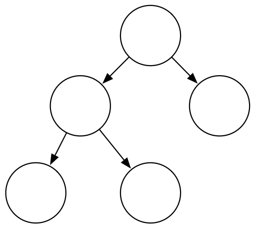
   - getNeighbors(int i) returns the set of neighbors of vertex i. It is fine for you to write something like: for each vertex v in getNeighbors(17) do ...
   - boolean getVisited(int i) returns TRUE if and only if vertex i has been marked as visited.
   - setVisited(int i, boolean b) sets the visited status of vertex i to b.
   - getColor(int i) returns the color of vertex i, either 0 or 1. If the color of vertex i has not been set previously, getColor(i) is undefined.
   - setColor(int i, color c) sets the color of vertex i to c.

   Solution: The idea is to start at some node (does not matter which one, since the graph is undirected and has only one connected component). Since it is easy, we will start with node 0. We will color the node in one color, then we color its neighbors the opposite color, and repeat. This is a breadth-first search. If we encounter a node that has already been colored in the wrong color, the graph cannot be colored as required. If we manage to color all nodes, we return true. The pseudocode is as follows:

```
Algorithm: TwoColor (Graph g, int $n$)
Input: A graph g with $n$ vertices
Output: Return true if g can be colored in two colors as required, false otherwise
int $i \leftarrow 0$;
we will use this to index the nodes in the graph
while $i < n$ do
        g.setVisited($i$,false);
        $i \leftarrow i + 1$
boolean color = true;//current color
Queue q=new Queue;// this will hold the nodes for the breadth-first search expansion
g.setColor(0,color);
q.enqueue(0);
while (!q.empty()) do
       $i \leftarrow$ q.dequeue();
       while (!g.getVisited($i$)) do
              g.setVisited($i$,true);
              color← !g.getColor(i);// we flip the color
        for each $j \in$ g.getNeighbors($i$) do
            if (g.getVisited($j$) and !(g.getColor(j)=color)) then return false;
            if (!g.getVisited($j$)) then
                g.setColor($j$,color);
                q.enqueue($j$);
return true;
```

2. **Binary trees**

   Suppose that you have a binary tree such that any internal node has exactly two children. An example of such a tree is depicted in Figure 1.



   Show by induction that a tree of this type with n leaves has exactly $2n - 2$ edges.

   Solution:

   Base case: If the tree has only 1 node, this is 1 leaf and 2*1-2 edges.

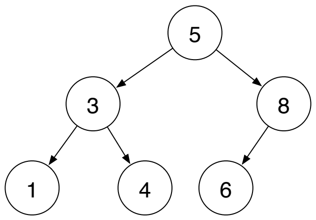
   Induction step: As always with binary trees, we will *cut out the root* and *use the induction hypothesis on the tow subtrees*. When we cut out the root, we get two subtrees, with l and $n - l$ leaves respectively. Assuming the inducton hypothesis is true, they have $2l - 2$ and $2(n - l) - 2$ edged. The whole tree has:

$$(2l - 2) + (2(n - l) - 2) + 2 = 2n - 2 \text{ edges}$$

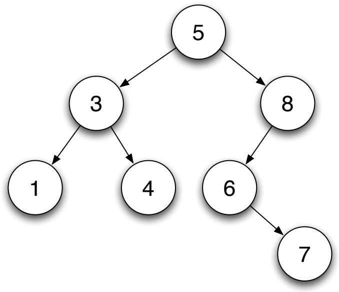
   which proves the statement.

3. **Binary search trees and heaps**

   (a) For the binary tree in the figure, show the result of inserting 7 into the tree.

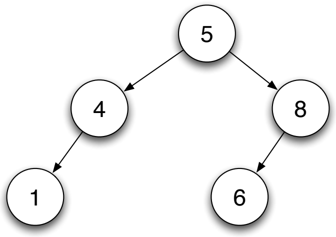


   Solution:

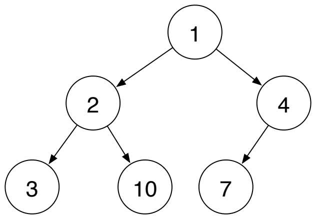


(b) For the same binary tree (without the previous insertion), show the result of removing 3 from the tree

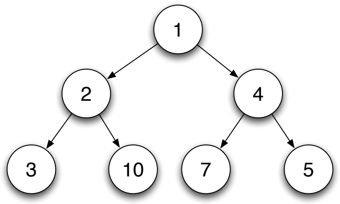
Solution:



(c) For the heap in the figure, show the result of inserting 5 in the heap

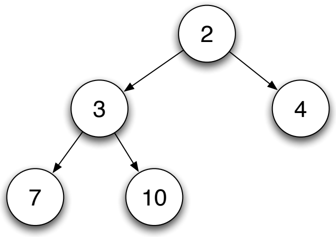


Solution:



(d) For the same heap (without the previous insertion) show the result of removing the minimum element.

Solution:



4. **General trees**

The **fringe** of a tree is the sequence of leaves, listed from left to right. Write an algorithm for a binary tree class that prints out the fringe of the tree. What is the O() of your method?

Solution: We give here just pseudocode for the fringe method.

```
**Algorithm** Fringe(BinaryTreeNode root)
**Input:** The root of a binary tree
**Output:** Prints the fringe of the tree
**if** root.left=**null and** root.right=**null then**
        **print** root.content;
        **return;**
**if** root.left$\neq$**null then** Frindge(root.left);
**if** root.right$\neq$**null then** Frindge(root.right);
```

For the running time, we have the following recurrence:

$$T(n) = c + T(k) + T(n - k - 1)$$

where c is a constant and k is the number of nodes on the left. Note that this means $O(n)$ where n is the number of nodes in the tree (because each node gets touched once).

5. **Proofs by induction for trees**

   (a) Suppose you have a full binary tree of depth k. Prove by induction that there are 2^k paths from root to leaves

   Solution:

   Base case: For 1 depth, we have a root with 2 leaves, so we have 2 paths from the root to leaves.

   Induction step: A binary tree of depth k consists of a root with 2 children, which are roots of sub-trees of height/depth $k - 1$. Hence, the number of paths from the root to the leaves is:

   $$P(k) = 2P(k - 1) = 2 * 2^{k-1} = 2^k$$

   where in the second equality we used the induction hypothesis.

   (b) Suppose you have a binary tree in which each node has either 0 or 2 successors. Show by induction that the total number of nodes has to be odd.

   Solution: Base case: For a tree with 1 node, this is trivially true.

   Induction step: Suppose that any tree up to depth k has an odd number of nodes. Hence, its number of nodes can be written as $2m+1$ for some natural number m. Now consider a tree of depth k. Its root will have two subtrees. The number of nodes, using the tree structure and the induction hypothesis, is:

   $$1 + (2l + 1) + (2r + 1) = 1 + 2(l + r + 1)$$

   which is odd. This concludes the proof.

6. **Counting elements with particular features**

   Consider the problem of counting how many times integers divisible by a given number k occur in an array a.

(a) What is the lower bound on the running time for any algorithm that attempts to solve this problem correctly?

Solution: The argument is similar to the one used in lecture 9 for the max algorithm. The lower bound on the algorithm is $O(n)$ where n is the length of the array. Suppose you do not touch some element of index i. An enemy can hide a number divisible by k in $a[i]$, so your algorithm will be incorrect.

(b) Give a **recursive** algorithm (in pseudocode) for solving this problem. Make your algorithm as efficient as possible.

Solution: We will give an algorithm very similar to max, so it will be a tail recursion

```
**Algorithm:** CountDivK ($a$, $n$, $k$)
**Input:** An array $a$ of natural number of length $n$ and a natural number $k$
**Output:** The number of elements of $a$ that are divisible by $k$
if ($n = 0$) then return 0
if ($a[n]$ mod $k = 0$) return 1+CountDivK($a$, $n - 1$, $k$)
return CountDivK($a$, $n - 1$, $k$)
```

(c) Prove by induction that your algorithm is correct

Solution: Again, we will follow the pattern from the max recursive algorithm. For the base case, $n = 0$ and the answer returned by the algorithm is trivially correct. For the induction step, we assume that the recursive call worked and returned correctly the number of elements divisible by k in the array up to index $n - 1$. Our algorithm in this case adds 1 if $a[n]$ is divisible by k, otherwise it adds a 0. So the answer will be correct, and the proof is concluded.

(d) Write a recurrence for the running time of your algorithm, and use it to determine $O()$ for your algorithm.

Solution: The recursive call happens regardless of the value of $a[n]$ and the rest of the algorithm contains constant-time steps, so we have:

$$T(n) = c + T(n - 1)$$

where c is a constant. As shown in lecture 9, this yields that $T(n) \in O(n)$.

7. **Proofs by induction**

(a) Prove by induction that $n^3 + (n + 1)^3 + (n + 2)^3$ is divisible by 3 for all n.

Solution:

Base case: $n = 0 \rightarrow 0 + 1^3 + 2^3 = 1 + 8 = 9$ which is divisible by 3

Induction step: Suppose the statement is true for n, so we have:

$$n^3 + (n + 1)^3 + (n + 2)^3 = 3k$$

for some natural number k. Now we examine the sum for $n + 1$:

$$\begin{aligned}(n + 1)^3 + (n + 2)^3 + (n + 3)^3 &= n^3 + (n + 1)^3 + (n + 2)^3 - n^3 + (n + 3)(n^2 + 6n + 9) \\ &= 3k - n^3 + n^3 + 3(2n^2 + 3) + 3(n^2 + 6n + 9) = 3k'\end{aligned}$$

Hence, we proved that the result holds for $n = 1$ which concludes the proof by induction.

(b) The Fibonacci numbers are a famous sequence defined as follows:

$$\begin{aligned} f_1 &= 1, f_2 = 1 \\ f_n &= f_{n-1} + f_{n-2}, \forall n \geq 3 \end{aligned}$$

Prove by induction that:

$$\sum_{i=1}^{n} f_i^2 = f_n f_{n+1}$$

Solution:

Base case: For $n = 1$, we have:

$$\sum_{i=1}^{1} f_i = f_1 = 1 = 1 * 1 = f_1 f_2$$

Induction step: Suppose the result hold for an arbitrary n, we now show it holds for $n + 1$:

$$\begin{aligned} \sum_{i=1}^{n+1} f_i^2 &= \sum_{i=1}^{n} f_i^2 + f_{n+1}^2 \\ &= f_n f_{n+1} + f_{n+1}^2 \text{ (using the induction hypothesis)} \\ &= f_{n+1}(f_n + f_{n+1}) \\ &= f_{n+1} f_{n+1} \text{ (using the definition of Fibonacci numbers)} \end{aligned}$$

This concludes the proof.