CS180 Solution of Homework 1

1. Prove the correctness of the following algorithm of binary addition.

```
Input: $n$ (in binary representation)
Output: $n+1$ (in binary representation)
   find the first digit that is 0 starting from right to left. Denote the position of first "0" to $i$.
   flip the $i^{th}$ digit to 1 and flip $1, 2, \dots (i-1)^{th}$ digit to 0
   return the flipped number $n'$
```

Solution:

We first define the representation of the nubmer. Let $n = x_k x_{k-1} \dots x_1$, where $x_t = \{0,1\}, t = 1, ..., k$. Assume the first "0" is at position i, we have $x_1 = x_2 = ... = x_{i-1} = 1, x_i = 0$. According to the algorithm, $n' = x_k x_{k-1} ... x_{i+1} x'_i x'_{i-1} \dots x'_1$, where $x'_1 = x'_2 = ... = x'_{i-1} = 0, x'_i = 1$.

So we can just show $n' - n = 1$. Consider in $n' - n$ in decimal representation:

$$\begin{aligned}
(n+1) - n &= 2^{k-1}x_k + 2^{k-2}x_{k-1} + ... + 2^i \cdot x_{i+1} + 2^{i-1} \cdot 1 - (2^{k-1}x_k + 2^{k-2}x_{k-1} + ... + 2^0 x_1) \\
&= 2^{i-1} - (2^{i-1} \times x_i + ... + 2^0 \times x_1) \\
&= 2^{i-1} - (2^{i-2} + ... + 2^0) \\
&= 2^{i-1} - \frac{2^{i-1} - 1}{2 - 1} \\
&= 1
\end{aligned}$$

2. A binary tree is a tree that every node has at most 2 children. The depth of a tree is the longest path that starts from the root and ends at a leaf. Give a recursive and an iterative algorithm that compute the depth of a binary tree. Justify the correctness and the complexity of your algorithms.

Solution:

Recursive algorithm: Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1.

maxDepth() //returns the maximum depth of the binary tree

```
If tree is empty
    return 0
Else
    maxDepthLeft = maxDepth(tree->left) // max depth of left subtree
    maxDepthRight = maxDepth(tree->right) // max depth of right subtree
    maxDepth = max(maxDepthLeft, maxDepthRight)+1 // max depth
    return maxDepth
```

Time Complexity: O(n)
The above algorithm is correct because: each time the recursive call reduces the problem of max depth of the of the tree of the current root to 2 problems (one of the max depth of the left subtree, and the other of the max depth of the right subtree) plus the comparison of the depth between the left and right subtrees, as well as the trivial operation of incrementing the depth count by 1, only if the current root has any child. So for a binary tree with n nodes, we need n recursive calls (one call for each node), and for each call it involves at most 2 operations specified above. Therefor the complexity is O(n).

Iterative algorithm:
We can use level order traversal to find height without recursion. The idea is to traverse level by level. Whenever

move down to a level, increment height by 1 (height is initialized as 0). Count number of nodes at each level, stop traversing when count of nodes at next level is 0.

maxDepth() //returns the maximum depth of the binary tree

```
Create a queue
Push root into the queue
Initialize height = 0
Loop
    nodeCount = size.queue()
    If nodeCount == 0
        return height
    Else
        increment height by 1
    While (nodeCount > 0)
        pop node from front
        push its children to the queue
        decrement nodeCount by 1
```

Time Complexity: O(n)
The above algorithm is correct because: with the tool of a queue, we can do level order traversal and count the exact number of nodes in each level, so the problem of max depth of a binary tree reduces to counting the number levels for the tree. In order to count the number of nodes in each level, we need to enqueue each node for each level and dequeue each node before its children could enqueue. We find that for each node, it enqueues once and dequeues once. Therefore for a binary tree with n nodes the time complexity is O(n).

3. You are given integers n and d in decimal representation. Describe a recursive and iterative version of elementary-school long division to divide n by d. Your primitives are addition, subtraction, multiplication of integers a and b, and integer division of a by b if $a < 10b$. Analyze the complexity (number of operations) of your algorithm. An operation is using any of the primitives between a and b.

Solution:

In the following we define a recursive function DIVIDE(a,b) that uses a basic procedure DIV(a,b). The problem statement says that DIV(a,b) works only if $a < 10b$. Also we can call DIV(a,c) where c is a constant. DIV(a,b) returns both the quotient q and remainder r where $a = bq + r$, $0 \le r < b$. Since DIV(a,b) returns a pair, we use DIV(a,b)[0] to refer to the quotient and DIV(a,b)[1] to refer to the remainder.

Here is the algorithm:

DIVIDE(a,b) //returns a pair (q, r) where q is the quotient and r is the remainder

```
while (DIV($a, 10^k$)[0] >= $b$)
    $k = k + 1$
$k = k - 1$ // we ensure that k is such that $\frac{a}{10^{k+1}} < b \le \frac{a}{10^k}$
let $(q_1, r_1) = $ DIV$(a, 10^k)$ // $a = 10^k q_1 + r_1$
call $(q_2, r_2) = $ DIV$(q_1, b)$ // $q_1 = q_2 b + r_2$, so $a = 10^k(q_2 b + r_2) + r_1 = (10^k q_2)b + 10^k r_2 + r_1$
$(q, r) = $ DIVIDE$(10^k r_2 + r_1, b)$ // $10^k r_2 + r_1 = qb + r$
return $(10^k q_2 + q, r)$//From the above two lines $a = (10^k q_2)b + 10^k r_2 + r_1 = (10^k q_2 + q)b + r$
```

The above algorithm is correct because: $a = (10^k q_2 + q)b + r$, so the quotient of DIVIDE(a,b) is $(10^k)q_2 + q$ and the remainder is r. We can see that the algorithm reduces DIVIDE(a, b) to DIVIDE($10^k r_2 + r_2$, b), but $10^k r_2 + r_1 = a - (10^k q_2)b \le a - 10^k b < a - a/10 = 9a/10$. Note that the first inequality holds because $b \ge 1$ and the second inequality follows from $\frac{a}{10^{k+1}} < b$. So the recursive call every time reduces a by factor of $9/10$ so the algorithm needs $(\log_{10} a)$ recursive calls and the total amount of work in each recursive call (including inside the while loop) is $\log_{10} a$, therefore the total running time is $O(\log_{10}^2 a)$.

4. Suppose you are playing the game of NIM. This game begins with a placement of $n$rows of matches on a table. Each row i has m_i matches. Players take turns selecting a row of matches and removing any or all of the matches in that row. Whoever claims the final match from the table wins the game. This game has a winning strategy based on writing the count for each row in binary and lining up the binary numbers (by place value) in columns. We note that a table is *favorable* if there is a column with an odd number of ones in it, and the table is *unfavorable* if all columns have an even number of ones.

   Example: Suppose we start off with three rows of matches of 7, 8 and 9. The binary representations of number of matches are $7 = 0111, 8 = 0111, 9 = 1001$. Therefore, the board will look like this

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

   Since the third row from the right and the second row from the right, each have odd numbers of ones, the table is favorable.

   (a) Prove that, for any favorable table, there exists a move that makes the table unfavorable. Prove also that, for any unfavorable table, any move makes the table favorable for one's opponent. Write the algorithm that on an input of favorable table outputs the row and the number of matches to remove from that row, to make the table unfavorable.

   Solution:

   The first thing to prove is that, for any favorable table, there exists a move that makes the table unfavorable for one's opponent. The easiest way to prove that something exists is by construction: that is, show how to take a favorable table and select a move to produce an unfavorable table.

   If the table is favorable, this means that there is at least one column that has an odd sum. Inspect the most significant column with an odd count. Select an arbitrary row with a one in that column; this is the pile of matches from which we will remove some. Change the 1 in that column to a 0 (note that this change alone is sufficient to show that the resulting number of matches is smaller than the original count, and thus a valid move, no matter how many other columns we change). Change any other columns, if necessary, to produce even parity. This will produce an unfavorable table for the opponent.

   We also need to prove that, given an unfavorable table, any move produces a favorable table for one's opponent. By definition of unfavorable table, there is even parity for each column. However, we may only select one row, and must remove at least one match from that row. As a result, at least one bit will change, and will change in only one row. Any bit change will modify the parity of the column, and as such, this will produce odd parity in at least one column, and thus a favorable table for the opponent.

   (b) Give a winning strategy for this game.

   Solution:

   The loser of the game will see no match on the table, which is a unfavorable table. So, a wining strategy is always leave a unfavorable table to the other.