Homework 1

Zhehao Wang 404380075 (Dis 1D)

Apr 4, 2016

1 Binary addition algorithm correctness proof

The input number n's binary representation can be denoted as $n = a_k...a_0$, where $a_i = 0$ or $a_i = 1$ ($0 \leq i \leq k$, k is the most significant bit). The flipped output number n' can be denoted as $n' = a'_k...a'_0$ in binary. We have

$$n = \sum_{j=0}^{k} a_j \cdot 2^j \qquad \text{and} \qquad n' = \sum_{j=0}^{k} a'_j \cdot 2^j$$

Denote the position of the first 0 in n to i, we have

$$n = \sum_{j=0}^{i-1} 1 \cdot 2^j + 0 \cdot 2^i + \sum_{j=i+1}^{k} a_j \cdot 2^j = \frac{2^i - 1}{2 - 1} + \sum_{j=i+1}^{k} a_j \cdot 2^j = 2^i - 1 + \sum_{j=i+1}^{k} a_j \cdot 2^j$$

and

$$n + 1 = 2^i + \sum_{j=i+1}^{k} a_j \cdot 2^j$$

After the flip in question, resulting number n' can be denoted as

$$n' = \sum_{j=0}^{k} a'_j \cdot 2^j = \sum_{j=0}^{i-1} 0 \cdot 2^j + 1 \cdot 2^i + \sum_{j=i+1}^{k} a'_j \cdot 2^j = 2^i + \sum_{j=i+1}^{k} a_j \cdot 2^j$$

Thus we have $n' = n + 1$, and the binary addition algorithm is correct.

2 Binary tree depth algorithm

Algorithm 1 Binary tree depth recursive

```
function MAXDEPTH(node)
    if node = nil then
        return 0
    leftDepth ← maxDepth(node.leftChild)
    rightDepth ← maxDepth(node.rightChild)
    if leftDepth > rightDepth then
        return leftDepth + 1
    else
        return rightDepth + 1
```

Alg. 1 (recursive)

Correctness: by definition, the depth of the tree is the longest path from root to leaves. Thus the depth at each node is $1 + max(depth(leftChild), depth(rightChild))$. The algorithm is correct as it recursively traverses both the left and right subtree of each node, and returns the larger depth of the two.

Time complexity: this algorithm is $O(n)$, where n is the number of nodes in the tree. Reason being that each node (including *null* leaves) of the tree will be visited exactly once, and the number of instructions executed during each visit is constant. The number of *null* nodes will be $2n$ at maximum, so the algorithm visits at most $3n$ nodes, which makes it $O(n)$ by asymptotic notation.

Algorithm 2 Binary tree depth iterative

```
function getDepth(root)
    queue ← [root]
    height ← 0
    while True do
        nodeCount ← queue.size()
        if nodeCount = 0 then
            return height
        height ← height + 1
        while nodeCount > 0 do
            r ← queue.dequeue()
            if r.hasLeft() then
                queue.enqueue(r.leftChild)
            if r.hasRight() then
                queue.enqueue(r.rightChild)
            nodeCount ← nodeCount − 1
```

Alg. 2 (iterative)

Correctness: the algorithm does a level-by-level traversal of the binary tree using a queue, not unlike a BFS (difference being that this has a nested loop, so that each inner loop only traverses the nodes on the same level; while BFS, having one loop with an empty queue being the exit condition, traverses the entire tree in that loop). If a level is not empty, all the nodes on the level are visited, all nodes on the next level are enqueued, and the tree's height increases by 1. Otherwise the algorithm returns the height, which is the current number of levels.

Time complexity: this algorithm is $O(n)$, where n is the number of nodes in the tree. Reason being that each node of the tree will be visited exactly once.

3 Elementary-school-division algorithm

Alg. 3 (recursive)

Time complexity: each call on *division* on line 5 will cause $n - m$ recursive calls, where n is the number of digits in the decimal representation of a, and m is the number of digits in the decimal representation of b. The number of calls on *division* reduces by 1 each time we calculate the result of the division of the remainder on line 6. Thus, the number of calls is $\frac{(n-m+1)(n-m)}{2}$, which makes the algorithm $O((n-m)^2)$. Given this particular implementation, each call on *division* results in 5 or 2 operations, depending on if $a > 10 \cdot b$. The total number of operations is $5 \cdot \frac{(n-m-1)(n-m)}{2} + 2 \cdot (n-m)$

Algorithm 3 Elementary-school-division recursive

function DIVISION_R(a, b)
  if $a < 10 \cdot b$ **then**
    return a/b
  else
    $temp \leftarrow division\_r(a, 10 \cdot b)$
    return $temp \cdot 10 + division\_r(a - temp \cdot 10 \cdot b, b)$

Algorithm 4 Elementary-school-division iterative

function DIVISION_I(a, b)
  $temp \leftarrow b$
  $stack \leftarrow []$
  while $a > 10 \cdot temp$ **do**
    $stack.push(temp)$
    $temp = 10 \cdot temp$
  $res \leftarrow a/temp$
  $result \leftarrow res$
  while $!stack.isEmpty()$ **do**
    $res \leftarrow a - temp \cdot result$
    $temp \leftarrow stack.pop()$
    $res \leftarrow res/temp$
    $result \leftarrow 10 \cdot result + res$
  return $result$

Alg. 4 (iterative)

Time complexity: both loops will run $n - m$ times, where n is the number of digits in the decimal representation of a, and m is the number of digits in the decimal representation of b. Asymptotically, this algorithm is $O(n - m)$. The total number of operations is $c \cdot (n - m)$. Given this particular implementation, the value of c is 9. (Assuming the cost of assignment is 0, and the costs of stack operations, like *push*, *pop* and *isEmpty*, are 1.)

Note: this algorithm stores $b, b \cdot 10, b \cdot 10^2 ... b \cdot 10^n$ until it's large enough to divide a. The algorithm then gets the most significant bit from the division, calculates the remainder, and pops out $b \cdot 10^{n-1}$ to divide the remainder. The algorithm iteratively repeat this process until we get the result. This algorithm has better time complexity than Alg. 3, because the stack essentially plays the role of a cache in dynamic programming here, so that we don't have to recalculate $b \cdot 10^i$ in each round.

4 NIM game

(a.1)

Theorem: for any favorable table, there exists a move that makes the table unfavorable.

Proof: denote the columns that have odd number of ones as $c_0 ... c_k$, where c_k is the most signifcant column. There exist at least one row r whose number of matches m_r's binary representation has 1 in column c_k.

Flip m_r's bit in column $c_0 ... c_k$, we have a new number of matches m'_r. $m'_r < m_r$ since m_r has 1 in the most significant flipped bit (in column c_k), while m'_r has 0. By taking out $m_r - m'_r$ matches from row r, the number of ones in column $c_0 ... c_k$ increases or decreases by 1, and the number of ones in other columns remains unchanged. Thus after this removal, all columns will have even number of ones, making the table unfavorable.

(a.2)

Theorem: for any unfavorable table, any move makes the table favorable for one's opponent.

Lemma: for any non-empty row r with m_r matches, removing $i = 1..m_r$ matches from row r will at least flip one column of m_r.

This lemma is trivial to prove: denote the number of matches after the removal as m'_r. If after removing i matches, none of the bits of m_r changed, then $m_r = m'_r$, and $i = m'_r - m_r = 0$, which contradicts with $i = 1..m_r$.

Proof: by the above lemma, for any non-empty row r, any move will flip at least one column of m_r. Denote the flipped columns of m_r as $c_0..c_k$, the number of ones in columns $c_0...c_k$ will increase or decrease by 1 because of the removal, thus column $c_0..c_k$ will have odd number of ones, making the table favorable for the opponent.

(b)

Winning strategy: if the board's favorable with only one row left, claim all the matches of that row. Otherwise, starting from a favorable board, pick a move that makes the board unfavorable for the opponent. Such a move always exists according to Theorem (a.1). When the board's unfavorable, whatever the opponent's move is, the board will become favorable again according to Theorem (a.2). The winning strategy is to repeat the steps above until claiming the last match.