Recursion Applications

Arnab Ganguly, Assistant Professor
Department of Computer Science, University of Wisconsin – Whitewater
Data Structures (CS 223)

1 Fast Exponentiation

The objective is to compute n^k, where k is an integer. However, instead of a for-loop, we want a fast algorithm, one which achieves this in $O(\log k)$ time.

Base-case: fastExp(n, 0) = 1 **and** fastExp(n, 1) = n
Recursive Rule: Let $x = fastExp(n, k/2)$. Then,

$$fastExp(n,k) = \begin{cases} x * x & \textbf{if } k \textbf{ is even} \\ n * x * x & \textbf{if } k \textbf{ is odd} \end{cases}$$

Fast Exponentiation

```
double fastExp(double n, int k) {
    if (0 == k)
        return 1;
    else if (1 == k)
        return n;
    else {
        double x = fastExp(n, k/2);
        if (k % 2 == 0)
            return x*x;
        else
            return n*x*x;
    }
}
```

Note: We can compute the n^{th} fibonacci number in $O(\log n)$ time, using Binet's formula and fast exponentiation. So, we see that recursion can be extremely fast as well as extremely slow, depending on how we use it.

2 Printing a Binary Search Tree in Sorted Order

The objective is to print a binary search tree in sorted order. Recall that in a BST, all nodes in the left subtree of a node are smaller than the node's value, whereas the right subtree has higher values.

Base-case: Node is null

Recursive Rule: Print the left subtree recursively, then the current node, and finally print the right subtree recursively.

Binary Search Tree in sorted order

```
void bstSorted(BSTNode node) { //void bstSorted(BSTNode *node) in C++
    if (null != node) // if (NULL != node)
        bstSorted(node.left);    // bstSorted(node -> left);
        System.out.print(node.value + " ");    //cout << node->value << " ";
        bstSorted(node.right);    // bstSorted(node -> right);
    }
}

void bstSorted() {
    bstSorted(root);
}
```

3 Printing a Linked List in Reverse Order

The objective is to print a linked list in reverse order, i.e., tail is printed first, then the previous node, and so on until the head is printed.

Base-case: Printing the tail of the linked list
Recursive Rule: Print the next node before printing the current node. Hence, traverse to the next node, print it recursively, and then print the current node.

Printing a Linked List in reverse order

```
void linkedListRev(ListNode node) { //void linkedListRev(ListNode *node) in C++
    if (tail == node)
        System.out.print(node.value + " ");    //cout << node->value << " ";
    else {
        linkedListRev(node.next);          //linkedListRev(node->next);
        System.out.print(node.value + " ");    //cout << node->value << " ";
    }
}

void linkedListRev() {
    linkedListRev(head);
}
```

4 Height of a Binary Tree

The objective is to compute the height of a binary tree. Recall that the height of a binary tree is the number of nodes on the longest path from root to a leaf in the tree.

Base-case: Height of a leaf is 1. In other words, height of a null node is 0.

Recursive Rule: Height of a subtree rooted at a node is one more than the maximum of the height of the left subtree and height of the right subtree.

Height of a binary tree

```
int height(BSTNode node) { //int height(BSTNode *node) in C++
    if (null == node) // if (NULL == node)
        return 0;
    else {
        int leftH = height(node.left);      //int leftH = height(node->left); in C++
        int rightH = height(node.right);    //int rightH = height(node->right); in C++
        if (leftH > rightH)
            return 1 + leftH;
        else
            return 1 + rightH;
    }
}

int height() {
    return height(root);
}
```