SIG Algorithm Challenges

Week 2: Recursion

Dane's Contact Email: dziema2@uic.edu

Join our Slack channel, #sig_algorithm_chal on the UIC ACM slack

Recursion Principles

Recursion is when a function calls itself to complete a function

- This is accomplished by identifying a base case that is easy to solve, and then repeatedly calling the function on smaller and smaller sub cases until a base case is reached

Recursion is tricky to get a handle on, but understanding the principles of recursion can help you solve other difficult types of problems

- Tree problems
- Dynamic Programming Problems
- General divide and conquer problems

Make sure to understand how your language handles recursion/parameter passing

- Java: Pass by reference only
- C++: Can do pass by reference or value
- Some additional features, like tail recursion support are nice to know

Don’t Overuse Extra Parameters

Some times, creating a recursive helper function with extra parameters is necessary to solve a problem

However, sometime people who are not familiar with recursion will write a recursive function like…

```
public class RealRecursion {
    public int helper(Collection problem, int depth) {
        if (depth == 0) {
            return 1;
        }
        /* do stuff */
        return helper(problem, depth - 1);
    }
    public int mainProblem(Collection problem) {
        int notAForLoop = 10;
        return helper(problem, 10);
    }
}
```

Extra parameters should contain details about the problem, not the recursion itself

Identifying Recursion Problems

Recursion problems can be difficult to spot

- One clue is the problem itself being defined recursively
  - Example: “Consider class X where the first instance is Y, and each subsequent member of class X is determined by rule Z

Not often explicitly tested for, but often times problems such as tree problems or linked list problems will be best solved recursively

- Demonstrating comfortability with recursion is a big plus
- Understanding the principles of recursion will help solve other problems

A lot of problems that involve recursion can be solved without recursion, but the recursive solution is much cleaner and more elegant.

General Tips for Recursion

- Think through how to divide this problem into two or more smaller instances of the exact same problem.
  - If this can be done, think of a way to combine the solutions to the smaller instances into a solution to the larger one (e.g. mergesort)
  - Make sure your sub problems are always decreasing in size
  - Once you have identified the sub problems, try stepping through some instances of them in your head

- Be smart with extra parameters
  - Review an implementation of recursive binary search. The original problem is on an Array, but the recursive helper will use two indexes as extra parameters.
  - These extra parameters are used to indicate subproblem divisions, so they are useful

- Keep it simple
  - Make the recursion do most of the work
- Brush up on how to analyze runtimes of recursive functions

Sample Recursion Problem

Given an array of size *n*, find the majority element. The majority element is the element that appears more than ⌊ n/2 ⌋ times.

Example 1:

```
Input: [3,2,3]
Output: 3
```

Example 2:

```
Input: [2,2,1,1,1,2,2]
Output: 2
```

Source: https://leetcode.com/problems/majority-element/

Problem Approach

Ideas for how to finish this problem

1: If an element is the majority element, it must be the majority of either the left or the right half of the array

2: If we know the majority element of the left and the right halves of the array, we can determine the majority element of the whole array

3: Scan the array and count the instances of the majority element from the right, and do the same for the left

4: If either of those elements appear over half the time, they are the majority element, otherwise there is none.

Pseudocode for this problem

```
majorityElement(Array):

        If (array.size == 1): return array[0]

        majRight = majorityElement(array.rightHalf)

        majLeft = majorityElement(array.leftHalf)

        If (majRight == magLeft) return majLeft

        lCount = 0

        rCount = 0

        For a in array

                If (a == majRigh)t: rCount++

                Else if (a == majLeft) lCount++

        Return lCount if lCount > array.size/2, rCount if rCount > array.size/2, NULL otherwise
```

Java Code for this problem (part 1)

Driver Code

```
public int majorityElement(int[] nums) {
    int l = 0;
    int r = 0;
    return majorityHelper(nums,0,nums.length-1);
}
```

The “divide” part of the recursive function

```
public int majorityHelper(int[] nums, int start, int end) {
    //if there is only one element then that element is the majority element
    if (start == end) {
        return nums[start];
    }
    //divide the problem into two parts
    int middle = (end - start) / 2 + start;
    int majLeft = majorityHelper(nums,start,middle);
    int majRight = majorityHelper(nums,middle+1, end);
    //if the two majorities are equal, then that must be the
    //gloabl majority
    if (majLeft == majRight) {
        return majLeft;
    }
```

Java Code for this problem (part 2)

```
else {
    //count the instances of the left majority in the full array
    int count = 0;
    for (int i=start; i<=end; i++) {
        if (nums[i] == majLeft) {
            count++;
        }
    }
    //if this is the majority element the return it
    if (count > (end-start)/2) {
        return majLeft;
    }
    int count = 0;
    //otherwise count the right majority element
    for (int i=start; i<=end; i++) {
        if (nums[i] == majRight) {
            count++;
        }
    }
    //if this is the majority element return it
    if (count > (end-start)/2) {
        return majRight;
    }
    //if neither were the majority element then there is no majority element
    return -1;
}
```

Recursion Problems to work on

Links to level 2-4 can be found at https://github.com/dane8373/SIG_Algorithm_Challenges/tree/master/Week2

Level 1) Implement MergeSort recursively

Level 2) Given an array A, an inversion is defined as an instance where there are two indexes, i and j such that i < j and A[i] > A[j] Count the number of inversions in an array

Level 3) On the first row, we write a `0`. Now in every subsequent row, we look at the previous row and replace each occurrence of `0` with `01`, and each occurrence of `1` with `10`. Given row `N` and index `K`, return the `K`-th indexed symbol in row `N`. (The values of `K` are 1-indexed.) (1 indexed).

Level 4) Given a mathematical expresssion with +, - and * such as “8 * 5 + 2 - 7 * 4”, return all possible results from computing all the different possible ways to group numbers and operators

Problem Approach for Level 1 problem

You guys probably know how to do Mergesort already

1: If the array is size 1, then it is sorted

2: Otherwise, split it in half and sort each half using Mergesort

3: Combine the two sorted halves into a single list in linear time by repeatedly taking the smaller element from each list

Code will not be shown for this problem as problem 2's code is very similar (spoilers)

Problem Approach for Level 2 problem

Observation 1: A list of size one has no inversions

Observation 2: We can get the total number of inversions by counting the incersions in each half, then by counting how many inversions there are between elements of the left half and elements of the right half

Mergesort provides a great way to count both of these

1: If the list is size 1, then there are no inversions

2: otherwise, sort the right and left half and count the number of inversions in the left and right half Every time you choose the element from the right half, that means all the remaining

3: Re-insert the two sorted arrays into the original array by selecting the minimum of the two sorted arrays.

4: Every time you choose the element from the right half, that means all the remaining elements on the left are inverted with that element, so increase the inversion count by the number of elements remaining in the left side

Java Code for level 2 problem (Part 1)

Base Case code

```
public class Solution {
    static long countInversions(int[] arr) {
        /* base cases for lengths of 0,1,2 */
        if (arr.length <= 1) {
            return 0;
        }
        else if (arr.length == 2) {
            if (arr[0] > arr[1]) {
                int temp = arr[1];
                arr[1] = arr[0];
                arr[0] = temp;
                return 1;
            }
            return 0;
        }
```

Array Splitting Code

```
/* split the array in two */
int start = 0;
int end = arr.length-1;
int middle = (end - start)/2 + start;
int [] left = new int[middle-start+1];
int j = 0;
for (int i=start; i<=middle; i++) {
    left[j++] = arr[i];
}
int [] right = new int[end-(middle+1)+1];
j = 0;
for (int i=middle+1; i<=end; i++) {
    right[j++] = arr[i];
}
/* count the inversions in both halves */
long count = countInversions(left);
count += countInversions(right);
```

Java Code for level 2 problem (Part 2)

Case 1: right side element is smaller

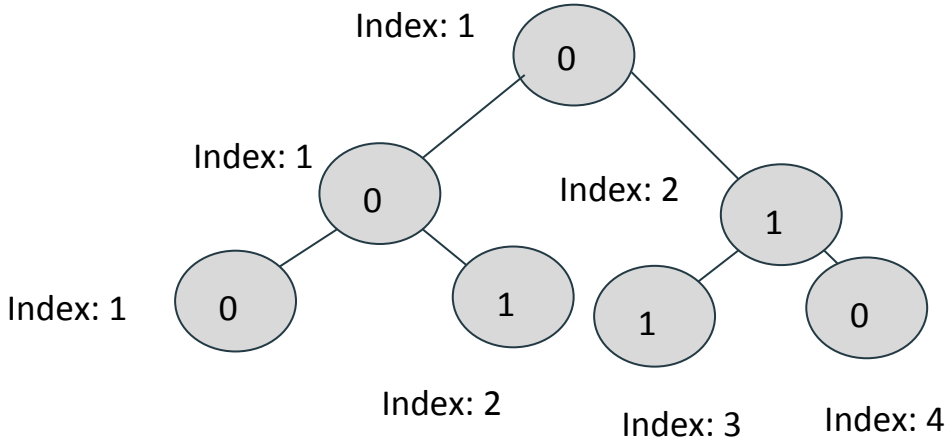
```
int mainIndex = 0;
int leftIndex = 0;
int rightIndex = 0;
/* while both of the arrays are not finished */
while(rightIndex < right.length && leftIndex < left.length) {
    //if the smallest thing on the right is
    //smaller than the smallest on the left
    if (right[rightIndex] < left[leftIndex]) {
        //this element on the right was after
        //left.length - leftIndex items that are greater than it
        count += left.length - leftIndex;
        arr[mainIndex] = right[rightIndex];
        mainIndex++;
        rightIndex++;
        //if the right is fully processed put
        // all the left elements in
        if (rightIndex >= right.length) {
            while (leftIndex < left.length) {
                arr[mainIndex] = left[leftIndex];
                leftIndex++;
                mainIndex++;
            }
        }
    }
```

Case 2: left side element is smaller

```
    //if the smallest thing on the left is the
    //smallest then we don't have an inversion for this element
    else {
        arr[mainIndex] = left[leftIndex];
        mainIndex++;
        leftIndex++;
        //if the left is fully processed empty the right
        if (leftIndex >= left.length) {
            while (rightIndex < right.length) {
                arr[mainIndex] = right[rightIndex];
                rightIndex++;
                mainIndex++;
            }
        }
    }
}
return count;
```

Problem Approach for Level 3 problem

Observation 1: It really helps to visualize this as a tree
(You don't ever have to use a tree class/representation, just visualize it as a tree)



Observation 2: If you know a node's parent, and you know if the node is the right or left child, then you know its value

Observation 3: All right children are even numbers, all left children are odd numbers

Observation 4: Given a Node at index N in row K, the index of its parent is N/2 if N is even, or N/2 + 1 if N is odd
(Equivalent to Parent(N) = N/2 + N%2)

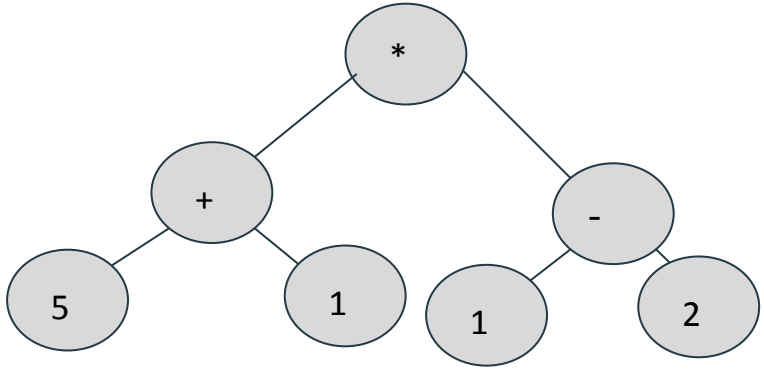
Java Code for level 3 problem

```
class Solution {
    public int kthGrammar(int N, int K) {
        //the 1st row is only zero
        if (N==1) {
            return 0;
        }
        //otherwise, find the node that led to this item
        int predescessor = kthGrammar(N-1,K/2+K%2);;
        if (predescessor == 0) {
            //if pred is 0, and K is even, then return 1 otherwise 0
            return K%2 == 0 ? 1 : 0;
        }
        else {
            //if pred is 0, and K is even, then return 0 otherwise 1
            return K%2 == 0 ? 0 : 1;
        }
    }
};
```

Problem Approach for Level 4 problem

Observation 1: Visualize operator precedence as a tree

- Deeper in parenthesis = lower on tree
- Root evaluated last
- Example shown for (5+1) * (1-2)



Observation 2: You can get every possible permutation by switching which operator gets to be the root, and then setting the subtrees to be all the possible evaluations of the operators to the left and right (recursively)

1: If there are no operators and only numbers in an equation string, then all possible values is just the single number contained in string

2: Otherwise, loop through all operators in string, setting each one as root and for each root, calculate all possible values for the all values to the left of the operator, and all possible values for the values to the right of the operator

3: Perform the operator at the root for all possible combinations of the left and right subtree values, and add this to the list of possible values

Java Code for level 4 problem (Part 1)

Need some helper functions to make this problem more compact

```java
class Solution {
    //helper function to determine if something is an operation
    public boolean isOp(Character c) {
        return c=='-' || c =='+' || c=='*';
    }

    //helper function to convert a character into an operation
    public int doOp(int a, int b, Character c) {
        if (c == '+') {
            return a+b;
        }
        else if (c == '-') {
            return a-b;
        }
        else {
            return a*b;
        }
    }
```

Driver code for recursive function

```java
if (ops.size() == 0) {
    return numbers;
}
//recursively solve
List<Integer> ret = evalTree(numbers, ops, 0, ops.size()-1);
Collections.sort(ret);
return ret;
```

Java Code for level 4 problem (Part 2)

It is easier for me to do this by converting my string to a list of operators and numbers

```
public List<Integer> diffWaysToCompute(String input) {
    //parse the input into numbers and operations
    if (input.length() == 0) {
        return new ArrayList<Integer>();
    }
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    ArrayList<Character> ops = new ArrayList<Character>();
    int start = 0;
    int end = 0;
    //for the whole input
    while (end < input.length()) {
        //if we hit an op, then all the stuff before was a number
        if (isOp(input.charAt(end))) {
            //parse the number and ad it
            numbers.add(Integer.parseInt(input.substring(start,end)));
            //put the operator in
            ops.add(input.charAt(end));
            end++;
            start = end;
        }
        else {
            end++;
        }
    }
    //add the last number
    numbers.add(Integer.parseInt(input.substring(start)));
```

Java Code for level 4 problem (Part 3)

```java
//recursive helper function to evaluate all possible configureations
public List<Integer> evalTree(List<Integer> numbers, List<Character> ops, int startOp, int endOp) {
    List<Integer> retList = new ArrayList<Integer>();
    //startOp > endOp, there are no operations left, just return the number after the last op we did
    if (startOp > endOp) {
        retList.add(numbers.get(endOp+1));
        return retList;
    }
    //otherwise, for every remaining operation, calulate an operation tree with that operation as the root
    for (int i=startOp; i<=endOp; i++) {
        //evaluate all possible values of the left subtree of this operation
        List<Integer> left = evalTree(numbers,ops,startOp,i-1);
        //evaluate all possible values of the right subtree of this operation
        List<Integer> right = evalTree(numbers,ops,i+1,endOp);
        //perform all possible combinations of left (op) right
        for (Integer l: left) {
            for (Integer r: right) {
                retList.add(doOp(l,r,ops.get(i)));
            }
        }
    }
    return retList;
}
```

Next Week: Hash Tables