Analyzing Algorithmic Patterns Based on Real Coding Interview Questions

Ian Dempsey

I. Introduction

A. Motivation

Nowadays interviews for jobs in the I.T industry are becoming more complex and demanding on the applicant. As this section of the working industry is usually quite technical, it is quite common for the applicants to have to perform some form of technical test. These tests normally involve a question or two that are based on a wide variety of topics in programming. These questions in particular are used to test potential employees on whether they have strong analytical and critical assessment skills. It is these programming questions that are focused on in this paper. The questions themselves have a wide range of difficulty, from easier ones which focus on in basics of programming- e.g., array indexing, sorting algorithms, up to harder questions which really push candidates- e.g., dynamic programming, graphs. This area of interviewing is known as *whiteboarding*, and has become quite common, it has led to some books being published which compile together numerous questions which are popular for interviewers to ask[1].The focus in this paper is not the ability to solve as many quetions as possible to give to the applicants to study, but to try and discover patterns in problems which are similar, and then classify these patterns based off of pre-established patterns that candidates would be used to. This classification will then be able to help applicants and fellow programmers to learn how to see the patterns in problems quicker and be able to apply them to multiple problems, where they need only alter specific areas of code to solve the target problem.

B. Problem Statement

The main problem that I will be solving in this project is to try and solve upwards of fifty questions that have been asked before in interviews for various companies. I will be getting these questions off of an online repository of programming questions, www.leetcode.com[2]. This site has over six hundred questions based on algorithms alone. I will solve questions in the areas of Palindromes,Merge Sort, Graphs (which includes Trees, Breadth-First Traversal, Depth-First Traversal and Graphs) and finally Dynamic Programming. The online resource LeetCode provides its own editor, which allows for a multitude of languages. I will be using Java for all my solutions. This online editor also allows for code submission for the problems, whereby the code is tested against multiple testcases. If the code fails a testcase the particular testcase and error is displayed. If the solution passes all testcases it will be accepted, and also display the runtime of the solution to complete the testcases. Also when the solution is accepted there is the ability to see how fast the accepted solution compares to other accpeted solutions by other users.

II. General Patterns for Analysis

A. Palindrome

Palindromes are extremely useful for searching algorithms, as Palindromes are meant to be the same in both directions, one can easily discover if the input is an actual palindrome. This is helpful for searching because one can search and find the odd character out, or the unique piece of data in some text. This style of algorithm is also space efficient, they normally have a space analysis of O(n/2) as the algorithm works over two elements of the input at a time. The basic approach of a Palindrome algorithm is to work inwards with both pointers starting at either end of the input and constantly moving towards one another and comparing if the elements are the same.
The following description is a general algorithm for solving the palindrome problem which is a common problem in Java and other languages. This approach can be used to solve numerous other problems by altering the inside of the loop. Pseudocode:

```
   Data: Given input of characters, S
   Result: Boolean
 1 initialization;
 2 leftIndex ⟵S[0];
 3 rightIndex ⟵S.length-1;
 4 while leftIndex < rightIndex do
 5 |   compare leftIndex with rightIndex;
 6 |   if leftIndex !=rightIndex then
 7 |   |   return false;
 8 |   leftIndex++;
 9 |   rightIndex++;
10 return true;
```

Algorithm 1: The Palindrome Algorithm

Whilst studying and working on this project, I have answered a number of questions from Leetcode.com which I was able to solve using an altered version of the above Palindrome pattern. Leetcode question 1 TwoSum is one example. This question is described as: *Given an array of integers, return indices of the two numbers such that they add up to a specific target. You may assume that each input would have exactly one solution, and you may not use the same element twice.* The following pseudocode is my answer to this question. In it I have taken the basic idea of the Palindrome algorithm of

having two pointers used for looking through the input data, but I have altered the way these two pointers behave.In this pseudocode I have highlighted any differences in blue.

```
Data: Array of integers nums, target S
Result: Indices i,j
leftIndex ⟵S[0];
rightIndex ⟵S.length-1;
while leftIndex < rightIndex do
    Sort the input array nums
    if nums[leftIndex]+nums[rightIndex] > S then
        rightIndex−−
    else if nums[leftIndex]+nums[rightIndex]<S
     then
        leftIndex++
    else
        return leftIndex, rightIndex
return 0;
```

Algorithm 2: LeetCode Q1 TwoSum

As can be seen from the above answer, the main differences are the conditionals inside the while loop. Instead of the normal approach of a Palindrome where one checks if the two elemts are equal, and if so moves the two pointers are moved at the same time towards eachother, I have changed the condisitionals such that instead of moving both of these pointers together at the same time, I only ever move one pointer at a time. Depending on the result of adding the two current elements at each pointer together. If the total from adding the two integers together was greater than the target sum, then I knew that I had to move the right-pointer left once, as this would allow me to have a smaller sum and potentially the correct sum. If on the other hand the sum was smaller than the total, I would only move the left-pointer right once and add the two elements and get a new result. This process was repeated until the target sum was found, or until the two pointers crossed which meant the target was not found.

B. Merge Sort

Merge Sort is a very powerful algorithm. It is more efficient than most styles of insertion, with a time analysis of $O(n * log_n)$, whereas insertion is $O(n^2)$. The idea of merge sort is to divide an array or some input in half and then sort each half before joining it back together. They do not have to be the same size which is useful.
Merge Sort uses the idea of divide and conquer, this means the list to be sorted should be divided up into equal parts first, then these new smaller parts should be sorted individually first before recreating the full list. Pseudocode:

```
Data: List of unsorted data
Result: Sorted List
if length of A is 1 then
    return 1
else
    Split A into two halves , L and R. Repeat until
     size of part =1
    Sort each part individually
    Merge with another subdivided section into B,
     the sorted list
    Return B, the sorted structure
```

Algorithm 3: The Merge Sort Algorithm through Recursion

C. Graphs

Graphs are common in our lives. News media use them to help us visualize certain statistics. Though these are not the graphs that are studied by Computer Scientists. Graphs studied by Computer Scientists are usually based on the tree structure, and the relationships among data elements. A tree is just one of the special types of graphs that can be studied, where the parent-child relationship is used to organise data. In this section I have focused on the Tree Abstract Data Type, Breadth-First Traversal, Depth-First Traversal and Graphs in general.

1) Trees: Trees are one of the most powerful styles of data structures for processing data, this is because they allow rapid searching and fast insetion/deletion of a node. Trees are made up of nodes, these are Objects which hold some data and have a key. This key allows one to determine where this node should be in the tree. The important distinction here with these nodes in comparison to other nodes used in various data structures, is that these nodes contain references to children instead of just the next Link. Each node has exactly one parent, but can have many children.A Binary Tree is a special type of tree, this is a tree which has between 0 and 2 children. The first node in a tree is the Root, and it is possible to traverse to any node in the tree from this node. With trees the main function is traversal. There are three basic styles of traversal: inorder, preorder and postorder. Inorder visits every Node in ascending order based on their key values. Preorder is where the root is visited first, followed by it's left subtree and then it's right subtree. Finally, postorder is where the left subtree is followed by the right subtree and then the root.
The following is an example of how one might search for a particular Node in a Tree.
Pseudocode:

Leetcode question 104, *Maximum Depth of Binary Tree*, is an example of a problem where I used the basic tree traversal styles to solve the question.The problem description is given as: *Given a binary tree, find its maximum depth.The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.* In this algorithm I used postorder traversal. Pseudocode can be found in Algorithm 6 of this paper.

As highlighted above, the key diference is inside the if statement. I have changed the conditional itself as it is the base case to stop the recursive execution. I have created a variable *ldepth* which is the total depth of the left subtree. This is a recursive call on the leftChild of the current node.

```
   Data: Given a key to search for
   Result: The desired Node, or null
 1 initialization;
 2 Nodecurrent ⟵root;
 3 while current.data is not key do
 4 |   if current is null then
 5 |   |   return null;
 6 |   if current.data > key then
 7 |   |   move left on the tree;
 8 |   else
 9 |   |   move right on the tree;
10 return current;
```

Algorithm 4: Finding a specific Node in a tree based on the key

```
1 postOrder(NodelocalRoot)
2 if localRoot ! = null then
3 |  postOrder (localRoot leftChild)
4 |  postOrder (localRoot rightChild)
5 |  Print(localRoot data);
```

Algorithm 5: Basic Tree Traversal using PostOrder Traversal

Line 4 will execute until it can't go left anymore, if it reaches null it will try to go right once, and then return going left. This is repeted until both calls return null. It then will go back all the way up until we reach the first call to go left, and set this summation of steps to ldepth. It then performs this same process but on the right subtree of the current node(which is the original node given), and set it to the variable *rdepth*. Finally there is a conditional checking if ldepth is greater than rdepth. If so it returns ldepth+1,or it returns rdepth+1. This+1 is required as I need to take into account the root node's level, if I didn't take this into account my answer would always be one less than the actual depth of the tree. The major similarities between the postorder algorithm and this solution is the way I went down the left subtree first, then the right subtree and finally to take into account the root I added one to the height. This particular way of traversing the tree, left-right-root, is exactly how postorder traversal is performed.

```
1  maxDepth(TreeNodelocalRoot)
2  if localRoot == null then
3  |  return 0
   int ldepth =maxDepth (localRoot leftChild);
   int rdepth =maxDepth (localRoot rightChild);
6  if ldepth > rdepth then
7  |  return ldepth+1
   else
9  |  return rdepth+1
```

Algorithm 6: Leetcode Q104 Max. Depth of Binary Tree

2) Breadth-First Traversal: This is a special way to visit the nodes in a tree, the ordering in this traversal pattern is to visit the root node, then move onto the children of the root node, printing each child in turn. It then will repeat this for each child of these nodes.Figure 1 shows the nodes in numerical order of visitation when using Breadth-First Traversal.

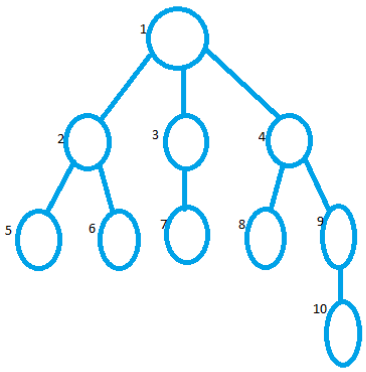

Fig. 1. Breadth-First Traversal

3) Depth-First Traversal: This is a second way of visiting nodes in a tree. This pattern involves starting at the root node, then going to the left most child, then repeating this movement until the traversal reaches a leaf node (a node which has no children), then it will move back up one node and try to visit the next child node of this current node. It repeats this until the traversal finds no univisited node. Figure 2 shows the nodes in numerical order of visitation when using Depth-First Traversal.

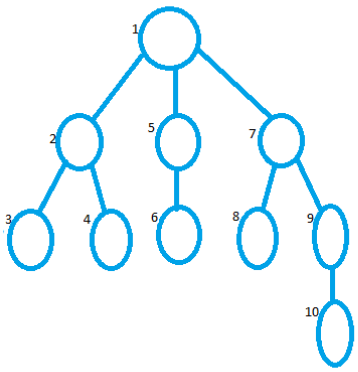

Fig. 2. Depth-First Traversal

III. The Questions Chosen

I based all of my questions off of the six-hundred or so that were available on www.leetcode.com[2]. To shrink the

question pool that I was choosing from I would try to pick the quesitons which were under the filter of Top Interview Questions. I did pick questions which were not listed under this filter, as I felt that solving some questions from outside of this restriction allowed me to get a more diverse seletion of questions. Also the questions were tagged by the user base of the site. Meaning that some of the questions which were not listed as being interview related, could have been asked in one, and vice versa. This does lead to the issue of having faith in the honesty of the community on the site.

IV. Problems with the questions

Whilst working on the numerous questions that I have solved on leetcode.com, I did come across issues which related to the questions themselves. These problems stemmed from the questions being worded strangely, or the example given for the solution was not explicitly clear in the way the question worked. These were just some of the issues which occurred whilst attempting to solve the problems. An example of such an issue is leetcode question 654. Maximum Binary Tree. This question's problem was easy to understand:
Given an integer array with no duplicates. A maximum tree building on this array is defined as follow:

1) *The root is the maximum number in the array.*
2) *The left subtree is the maximum tree constructed from left part subarray divided by the maximum number.*
3) *The right subtree is the maximum tree constructed from right part subarray divided by the maximum number.*

Construct the maximum tree by the given array and output the root node of this tree.
The issue which I personally came across was when I tried to understand the way the left and right subtree should be structured. The description for them was not clear to me, as I was unsure what was meant by ″left part subarray divided by the maximum number″. I then took a look at the provided example.

Example 1:

```
Input: [3,2,1,6,0,5]
Output: return the tree root node representing the following tree:

      6
    /   \
   3     5
    \    /
     2  0
       \
        1
```

Fig. 3. Q654 Given Example

For me personally this did not make it any clearer as to which way I should structure the subtrees. I was able to understand that the biggest number of each subarray should be the next number and work down in a descending order from the root. I was unable to understand why the 2 was to the right of 3, and yet 0 was to the left of 5. This confusion did cause me to take longer to solve this question than anticipated.

References

[1] G. McDowell, *Cracking the Coding Interview: 189 Programming Questions and Solutions*. CareerCup, LLC, 2015.
[2] LeetCode, "Leetcode."