

Analyzing Algorithmic Patterns Based on Real Coding Interview Questions

Ian Dempsey,
Computer Science Department,
Maynooth University,
Email: ian.dempsey.2013@mumail.ie

I. ABSTRACT

This is my abstract. It is empty for now. Fill in later

II. INTRODUCTION

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. The technical tests also allow the employer to see how many standard algorithms the candidate knows, and is able to use. However, it is difficult to master all of the different types of algorithms and their intricacies.

The purpose of this paper is to study the relationship between the classical and well known algorithms, and the interview questions which are commonly asked. This paper aims to give guidance to people who want to study the interview questions, but also learn a core set of common ideas which will help them solve multiple problems. This would therefore allow them to save time as they prepare for the interview.

This paper presents a comprehensive report summarising the similarities between interview questions and standard algorithms taught in any programming course. This distinguishes our work from interview preparation books which focus on the problem solving skills, simply showing the reader a solution to a single problem, and not reinforcing material they would already be familiar with. This paper's goal is to show people that understanding the fundamentals of several common algorithms will allow them to be able to solve a wide array of problems. It is also the purpose of this paper that a beginner of programming or a fresh graduate can use this paper as guidance to show them how much time is needed for preparatory work before they have the interview.

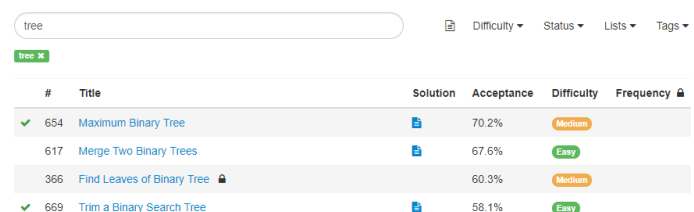
III. TECHNICAL BACKGROUND

Nowadays, in general, there are a lot of online repositories for interview questions and interview related material. Websites such as hackerrank.com, glassdoor.ie, geeksforgeeks.org, and leetcode.com all offer information on what questions are common during interviews for different companies. They also

offer ways to test potential solutions to some of the given questions.

For this paper we chose to use leetcode.com[1], as this website comes with a huge amount of online material that is generally taken from previously asked interview questions. Companies also use sites similar to leetcode when they are building up a bank of questions to pose to candidates. This was important in our decision for the repository, as it meant we were using a platform which was regularly updated. LeetCode itself also has some built-in features which greatly appealed to us. The website has an online discussion forum for each question, allowing the community to discuss solutions, issues, and the questions themselves. LeetCode also host their own weekly coding competitions which allow users to gain more experience and confidence in coding problems. One of the unique main features this online judge website has in comparison to others is a section for a user to perform a mock interview. This mock interview is under the time constraint of a normal real world interview, this was a pre-eminent for us in choosing our online judge. The online editor that is used by leetcode also allows a user to select from a multitude of programming languages, as shown in figure 3 below. For the purpose of this paper the language that was chosen was java. The reason for this is because it is a language most people learn first. However, any choice of programming language would be just as acceptable, as we present pseudocode throughout the paper which can be used by any major programming language.

The user interface of LeetCode is very simple and accessible to use. The main page which lists all of the problems, offers its users the option to filter questions by difficulty, which company has asked it before, what area the questions focuses on and much more. As seen in figure 1, we have chosen to focus on questions which are tagged as being related to trees.



#	Title	Solution	Acceptance	Difficulty	Frequency
654	Maximum Binary Tree		70.2%	Medium	
617	Merge Two Binary Trees		67.6%	Easy	
366	Find Leaves of Binary Tree		60.3%	Medium	
669	Trim a Binary Search Tree		58.1%	Easy	

Fig. 1. Homepage

A user simply chooses the question which they wish to attempt, and is promptly brought to the problem's specific screen, as seen in figure 2. On this screen there is a general description of the problem and a few examples for more clarification. This page also contains the online editor that the user will operate to attempt the problem. The online editor is intuitive to understand, and as stated has a variety of options for the user to choose from.

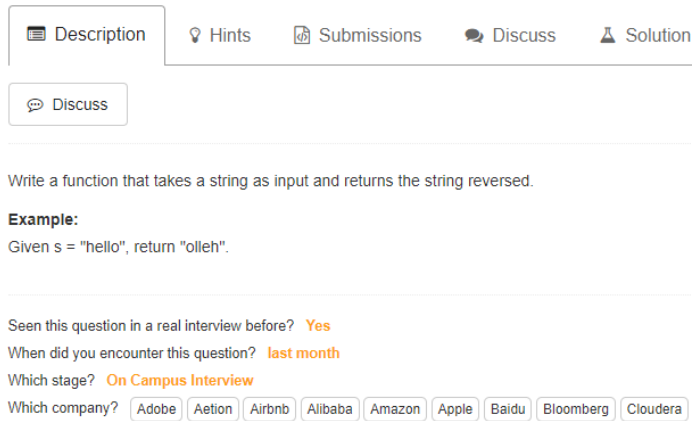


Fig. 2. Problem Specific Page

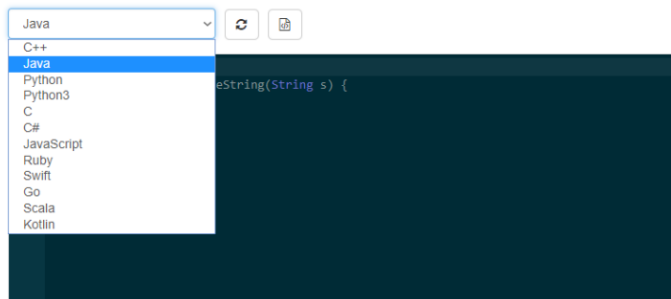


Fig. 3. Online editor with options displayed

The user will attempt to solve the problem they have selected, and they will then want to submit their answer and see if it was correct. They perform this by clicking the submit solution button underneath the editor. Leetcode allows a user to submit their code and be informed almost instantly if they are correct or not. There are two possible outcomes once the user clicks the submit button. Either it is accepted and leetcode returns an accepted result with suggestions for the next question a user could attempt, shown in figure 4, or their answer failed a certain testcase or even their code failed to compile and run, shown in figure 5. In the case of failure, leetcode will report where an error is in the code by referencing the line.



Fig. 4. User solution passes all testcases

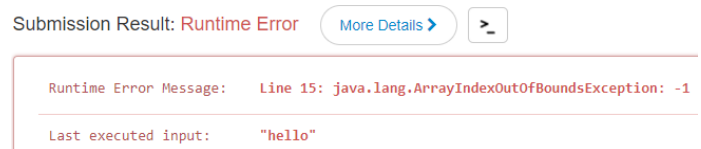


Fig. 5. User solution failing submission

IV. PROBLEM STATEMENT

In order to fulfill the goal of this paper, we will answer the following research questions:

- 1) What algorithms to choose, and what categories are these algorithms associated with.
- 2) How do these algorithms relate to the classic algorithms and how similar are they.
- 3) How easy is it for a person to identify the pattern of the algorithm.

V. THE SOLUTION

To answer question one above, we had to perform some information gathering. Many companies choose similarly styled questions for their technical tests, so we gathered information from forums, such as glassdoor.ie [2], and websites which discussed experiences people had at interviews. We also read through the main categories listed in books which prepare people for interviews, such as Cracking the Coding Interview [3]. These discussions listed the questions they were asked, and by what companies. The website geeksforgeeks.org has a list of the top interview questions that a user should study prior to an interview [4]. Therefore this paper focuses on the following areas:

- Sorting Algorithms
- Searching Algorithms
- Graphs
- Trees
- Dynamic Programming

In each of these areas we categorised them into some key patterns. These are patterns or types of the above listed algorithms which commonly appeared. These include, but are not limited to:

- Palindrome
- Merge Sort
- Binary Search
- Tree Traversal
- Breadth-First Search
- Depth-First Search

par Leetcode has over six hundred questions, which is impossible for us to answer in such a short period of time. So we chose a representative amount of this total. We selected them based on reputation, if they were asked in interviews, and the solve rate of the question. Relating back to figure 1, it can be seen that we are able to filter out the questions we chose.

With regards to question two above, in order to find the most similar algorithm, we first compared the ideas of the classical algorithm and the solution to the question we were solving. It is possible to have multiple algorithms apply to a problem.

We implemented the easiest solution which is the first which came to our minds. This is because one normally has limited time in an interview, and this means that a person would build on an idea they initially discover after they understand the question being asked. The candidate would not have enough time to explore multiple solutions to a problem under the interview conditions. We also compared a control flow graph of our solution, in comparison to the standard algorithm. We have listed the aforementioned patterns in the following section, along with sample questions which are similar to the classical algorithms. We picked only a few to represent all of the questions due to the page limit. Please see table II for more detail.

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:

Input : Given input of characters, S

Output: Boolean

```

1 leftIndex ← S[0];
2 rightIndex ← S.length-1;
3 while leftIndex < rightIndex do
4   compare leftIndex with rightIndex;
5   if leftIndex != rightIndex then
6     return false;
7   leftIndex++;
8   rightIndex++;
9 return true;
```

Algorithm 1: The Palindrome Algorithm

Whilst studying and working on this project, we have answered a number of questions from Leetcode.com which we were able to solve using an altered version of the above Palindrome pattern. **ID 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 our answer to this question. In it we have taken the basic idea of the Palindrome algorithm of having two pointers used for looking through the input data,

but we have altered the way these two pointers behave. In this pseudocode we have highlighted any differences in blue.

Input : Array of integers nums, target S

Output: Indices i,j

```

1 leftIndex ← S[0];
2 rightIndex ← S.length-1;
3 while leftIndex < rightIndex do
4   Sort the input array nums
5   if nums[leftIndex]+nums[rightIndex] > S then
6     rightIndex--
7   else if nums[leftIndex]+nums[rightIndex]<S
8     then
9     leftIndex++
10  else
11    return leftIndex, rightIndex
12 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 elements are equal, and if so moves the two pointers are moved at the same time towards each other, we have changed the conditionals such that instead of moving both of these pointers together at the same time, we 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 we knew that we had to move the right-pointer left once, as this would allow us to have a smaller sum and potentially the correct sum. If on the other hand the sum was smaller than the total, we 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:

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

Input : List of unsorted data

Output: Sorted List

```

1 if length of A is 1 then
2   return 1
3 else
4   Split A into two halves , L and R. Repeat until
    size of part =1
5   Sort each part individually
6   Merge with another subdivided section into B,
    the sorted list
7   Return B, the sorted structure

```

Algorithm 3: The Merge Sort Algorithm through Recursion

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 insertion/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 its left subtree and then its 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:

Input : Given a key to search for

Output: The desired Node, or null

```

1 Nodecurrent ← root;
2 while current.data is not key do
3   if current is null then
4     return null;
5   if current.data > key then
6     move left on the tree;
7   else
8     move right on the tree;
9 return current;

```

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

```

postOrder(Node localRoot)
if localRoot != null then
3 postOrder(localRoot leftChild)
4 postOrder(localRoot rightChild)
5 Print(localRoot data);

```

Algorithm 5: Basic Tree Traversal using PostOrder Traversal

ID 104, *Maximum Depth of Binary Tree*, is an example of a problem where we 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 we used postorder traversal. Pseudocode can be found in Algorithm 6 of this paper.

```

maxDepth(TreeNode localRoot)
if localRoot == null then
3 return 0
int ldepth = maxDepth(localRoot leftChild);
int rdepth = maxDepth(localRoot rightChild);
if ldepth > rdepth then
7 return ldepth+1
else
9 return rdepth+1

```

Algorithm 6: Leetcode Q104 Max. Depth of Binary Tree

As highlighted above, the key difference is inside the if statement. We changed the conditional itself as it is the base case to stop the recursive execution. We 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. 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 repeated until both calls return null. It will then 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 we need to take into account the root node's level, if we 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 we went down the left subtree first, then the right subtree and finally to take the root into account we added one to the height. This particular way of traversing the tree, left-right-root, is exactly how postorder traversal is performed.

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.

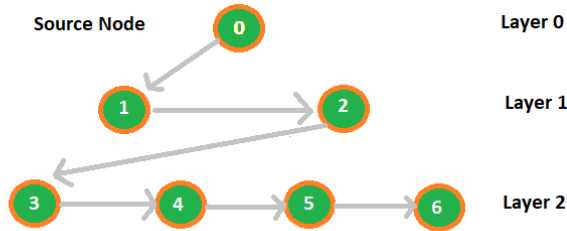


Fig. 6. Breadth-First Traversal

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 unvisited node. Figure 2 shows the nodes in numerical order of visitation when using Depth-First Traversal.

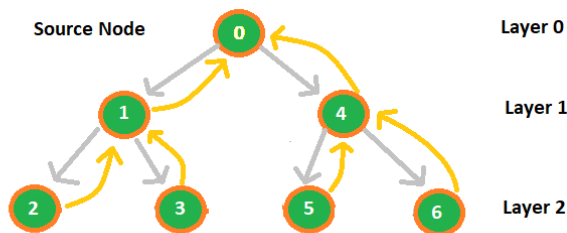


Fig. 7. Depth-First Traversal

D. Dynamic Programming

Dynamic Programming is often referred to as ‘smart recursion’. This is because it takes the idea of recursion and improves the inefficiencies that come along with it. These inefficiencies mainly being that recursive algorithms are generally space inefficient. Each recursive call adds a new layer to the stack. This is why dynamic programming is such a strong style of programming. It is normally just a matter of taking a recursive algorithm and finding the overlapping subproblems. A user then caches these results for future recursive calls. This style is known as *Top-Down*. There is a different approach, known as *Bottom-Up*. We will give a quick demonstration of the two by using the classic recursive algorithm to solve Fibonacci numbers.

Input : Integer n

Output: Fibonacci Numbers

```
1 if n < 3 then
2   return 1;
3 return fib(n-2) + fib(n-1);
```

Algorithm 7: Fibonacci Numbers through normal recursion

1) *Top-Down*: This method of dynamic programming is known as top-down as we would follow the normal structure of a simple recursion solution, we start from n and compute smaller results as we need them in order to solve the original problem. We then store these results in some form of data structure, this is known as memoization. We can then access these results quickly and efficiently in comparison to having to recompute the same values for n over and over again. This leads to us saving space and time when working with recursively styled problems. Though it is not the most efficient still.

Input : Integer n

Output: Fibonacci Numbers

```
1 computed ← HashMap;
2 if n < 3 then
3   return 1;
4 return fib(n-2) + fib(n-1);
5 computed.put(1,1);
6 computed.put(2,1);
7 return fib2(n, computed);
```

Algorithm 8: fib; Fibonacci Numbers through Top-Down and Memoization

Input : Integer n, HashMap computed

Output: Fibonacci Number

```
1 if computed.containsKey(n) then
2   return computed.get(n);
3 computed.put(n-1, fib2(n-1, computed));
4 computed.put(n-2, fib2(n-2, computed));
5 newVal ← computed.get(n-1) + computed.get(n-2);
6 return newVal;
```

Algorithm 9: fib2; Fibonacci Numbers through Top-Down and Memoization

2) *Bottom-Up*: The issue with this memoized version is that even when we are saving the results to reuse them later, we still have to go all the way down to the base cases with the recursion the first time. This is when we have yet to compute any values, so we have nothing stored. Bottom-Up solves this by going in reverse to the normal recursive way. Instead of going down to the base case from the value of n, it works its way up to the value of n.

It must be noted though that Dynamic Programming is one of the toughest sections of programming to learn. This only makes it more challenging for candidates to answer questions in interviews in such a limited window. The reason it is so tough is because a candidate must think of the problem in

Input : Integer n

Output: Fibonacci Numbers

```

1 results[] ←int[n+1];
2 results[1] = 1;
3 results[2] = 2;
4 for i ← 3 to n do
5   results[i] = results[i-1]+results[i-2];
6 return results[n];

```

Algorithm 10: fibDP; Fibonacci Numbers through Bottom-Up

	Algorithm
1	Palindrome
2	Merge Sort
3	Binary Search
4	Tree Traversal
5	Breadth-First Search
6	Depth-First Search
7	Dynamic Programming
8	Not Applicable

TABLE I

THE CLASSICAL ALGORITHMS

multiple ways, and make the decision to tackle it in a certain way. If they choose the wrong option, then they will end up with a very inefficient solution, choose correctly and they will have shown a very strong understanding of this area.

VI. EVALUATION

In the previous section of this paper, we only gave specific examples of the questions that we answered. Here we give a detailed summary of all solved questions in table II. All of these were accepted by leetcode's online judging system. We also have created a table which details a numeric value for the algorithms previously listed in V.

In table II there are a number of headings. Question ID is the specific ID of each question solved. PassRate is the percentage of solutions accepted by leetcode. These were taken at the time of writing. Difficulty is the level that was awarded to each question by leetcode themselves. Time is the total time in minutes it took us to complete each question. Evaluation is our opinion on the difficulty of each question we solved. Algorithm, this is the algorithm we feel that each question was closest to. The number in this column refers to table I, which states the number assigned to each algorithm.

As can be seen in table II, some of the questions which were marked as having difficulty easy, took longer than some which were marked as medium. This is because it took longer for us to see the pattern which was useful to solve the problem.

It is not easy to judge how fast a user can see the underlying pattern of a problem. This is because each user will have a different background. We have categorised the different users into four distinct types.

- 1) These users are people which have a background in coding competitions. They are highly experienced and are unlikely to find the questions hard.
- 2) Users with some computer science background, who understand the internal workings of data structures and

Question ID	PassRate	Difficulty	Time(mins)	Evaluation	Algorithm
1	36.0%	Easy	9	Easy	1
2	28.1%	Medium	20	Medium	–
20	33.7%	Easy	12	Easy	1?
21	39.3%	Easy	25	Easy	Graphs(?)
23	27.7%	Hard	27	Medium	2
35	39.8%	Easy	10	Easy	Linear Search
53	39.9%	Easy	20	Medium	Array
58	32.0%	Easy	15	Easy	String/Array
69	28.2%	Easy	19	Easy	3
70	40.5%	Easy	18	Easy	7
72	32.1%	Hard	29	Hard	7
88	32.1%	Easy	12	Medium	Array
102	40.9%	Medium	30	Medium	5
104	53.5%	Easy	28	Medium	4
111	33.3%	Easy	17	Easy	5
114	35.7%	Medium	33	Medium	4
136	54.9%	Easy	10	Easy	Binary Search(?)
147	33.4%	Medium	26	Medium	Insertion Sort
148	29.2%	Medium	35	Hard	Insertion Sort
152	26.2%	Medium	31	Hard	7
167	47.1%	Easy	16	Easy	1
205	34.2%	Easy	18	Easy	HashMaps(?)
290	33.2%	Easy	28	Medium	–
326	40.4%	Easy	25	Easy	–
336	26.5%	Hard	18	Medium	1
344	59.5%	Easy	8	Easy	1
387	47.1%	Easy	19	Medium	Linear Search(?)
389	50.9%	Easy	10	Easy	Linear Search
404	47.3%	Easy	16	Medium	4
654	70.1%	Medium	46	Hard	2
669	58.1%	Easy	18	Easy	4
690	52.9%	Easy	45	Medium	5

TABLE II

ALL SOLUTIONS SOLVED BY US

Background Type	Expected Difficulty
1	Easy
2	Medium
3	Medium
4	Hard

TABLE III

THE EXPECTED DIFFICULTY EACH TYPE OF CATEGORISED USER WOULD HAVE WITH THE QUESTIONS ON LEETCODE.COM.

other concepts. These people would generally find most problems approachable.

- 3) There exists people which only have some self study and learning completed. These people would find some of the concepts difficult to grasp and require some study.
- 4) Finally there would be people who are interested in learning to code who have no experience. They will have to put a lot of study in to grasp even basic concepts.

This leads us to create a table, table III, in which we have given the expected difficulty that each type of user listed above would be expected to have with the questions on leetcode.com.

A. Question Evaluation

We must also evaluate the questions themselves and give a description of some issues we had with them. Whilst working on the numerous questions that we have solved on leetcode.com, we 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 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 we personally came across was when we tried to understand the way the left and right subtree should be structured. The description for them was not clear to us, as we were unsure what was meant by "left part subarray divided by the maximum number". We then took a look at the provided example.

Example 1:

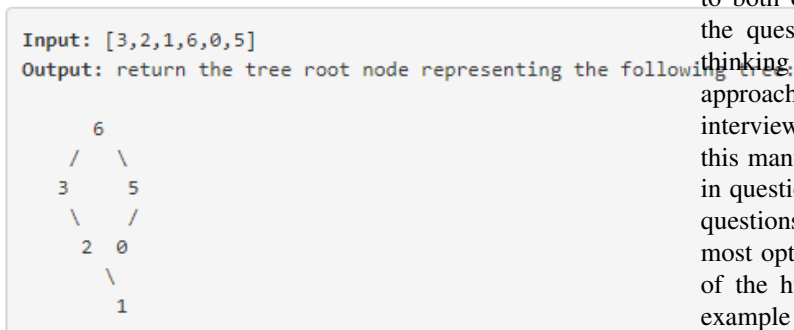


Fig. 8. ID 654 Given Example

For us this did not make it any clearer as to which way we should structure the subtrees. We were 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. We were 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 us to take longer to solve this question than anticipated.

VII. CONCLUSION

In this paper we have created a guide for beginners in programming, fresh graduates and readers who wish to refresh their knowledge. This paper enables them to learn and understand some of the fundamental algorithms of programming quickly. Through learning these patterns, the reader should be able to identify solutions to a wide array of coding problems.

The results of this paper have a potentially generalisable impact on the coding world, yet they also have a specific impact on the key area of algorithms. The potentially generalisable case is impacted as this paper can apply to multiple different programming languages. Java was used to solve all of the questions in this paper, but in reality any language would have been applicable for the questions. Therefore, this paper has

the potential to be useful for people who come from different language backgrounds. The specific case is due to this paper focusing in on the algorithms and approach to solving the problems. By showing how the algorithms which are learned by every coder can be applied to multiple problems by simply changing some area of these important patterns. The paper clearly shows the reader that learning the main algorithms will allow them to spend more time thinking of the solution based off of previously learned algorithms. This will give the reader a good starting point when they are attempting to solve a question. This should allow the reader to be more confident in their coding ability, as they will have a strong basis to start from.

The approach taken in this paper was to solve each question manually by a human. The approach was stipulated as follows: allocate a certain amount of time to solving the question and then attempt it. Whilst solving the question, the approach was to document the ideas which were used to try and solve the question. This process had both positives and negatives to it. The positives were that the solutions were all attempted and completed by a human, meaning that this process of solving the questions was similar to the interview process. This is due to both of these scenarios, the interview and human solving the questions for this paper, having the human element of thinking of multiple ways to tackle the issues. This allowed our approach to be as valid and authentic to the real scenario of the interview as possible. The negatives of this approach were that this manual style required a lot of human effort. This resulted in questions taking longer than expected to be completed, and questions being solved in ways which might not have been the most optimal. This method is dependent on the coding ability of the human, which depending on the topic can vary. For example the questions relating to the advanced topic Dynamic Programming took longer to learn and be able to complete. In this paper there was no automatic approach to solving the questions developed. This would have been helpful for validating the solutions produced by the manual methodology. This automatic process was unable to be produced because of the time constraints of this paper. This paper does have its limitations. As mentioned previously the approach to a solve the questions can be improved. Also, the topics which were covered, which are important and fundamental to any programmer wishing to further their knowledge in this field, did not address all of the areas which the questions answered. For example the questions focusing on Strings. Whilst String manipulation was used throughout multiple answers, it was never formally addressed in this paper. This applies to other key patterns such as insertion sort, red-black trees etc. The reason this paper does not cover more topic areas is because of the time constraint on this paper. It also would have been extremely useful to have a numerical representation of how similar a solution was to the original pattern it is based off of. This was explored by us, but we were unable to produce this numerical value. This was because we could not justify a way to classify similarity. This was because justifying similarity is subjective. One might want to base the similarity on the length of the code, as pieces of code which have similar length might be closely related, another person might base it

purely off of the logic involved, but not take into account the length of it, meaning the solution could be correct and employ the algorithm to some extent but it would be too far removed from the original algorithm then. A basic approach to judging similarity that we discovered was to count the additions and deletions required to turn one piece of source code A, into source code B. The issue was that most difference calculators like diffchecker.com [5], or the function diff(), would take variable names into account and count them as being different. This was not good enough as variable names should not be taken into account. The similarity should be calculated based on the logic of each source. We also looked into using tools for plagiarism checking such as Turnitin and Moss [6]. Moss is a powerful tool for detecting plagiarism. Moss is described as an automatic system for determining the similarity of programs [7]. The tool itself is heavily focused on finding plagiarised pieces of code from sources given. Even though we were able to see the benefits of using Moss from other users experiences [8], we decided not to use it. This was for a few reasons, mainly that the process was quite laborious, as we needed to send off each file to compare with a base file.

REFERENCES

- [1] LeetCode, "Leetcode."
- [2] GlassDoor, Sept. 2017.
- [3] G. McDowell, *Cracking the Coding Interview: 189 Programming Questions and Solutions*. CareerCup, LLC, 2015.
- [4] GeeksForGeeks, "Top 10 algorithms in interview questions."
- [5] DiffChecker. Online, Nov. 2017.
- [6] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, (New York, NY, USA), pp. 76–85, ACM, 2003.
- [7] A. Aiken, "Moss." Online.
- [8] K. W. Bowyer and L. O. Hall, "Experience using "moss" to detect cheating on programming assignments," in *Frontiers in Education Conference, 1999. FIE '99. 29th Annual*, vol. 3, pp. 13B3/18–13B3/22 vol.3, Nov 1999.