

CONTENTS

		Page
I	Introduction	1
II	Background	1
III	Problem Statement	2
IV	The Solution	2
V	Evaluation	8
VI	Limitations	9
VII	Related Work	10
VIII	Conclusion and Future Works	11
	References	11

LIST OF FIGURES

1	The homepage of LeetCode allows users to search for specific types of questions.	2
2	The question page displays a description of the question and gives a concise example.	2
3	Online editor with options displayed allowing candidate to use different programming languages.	2
4	Acceptance message with suggested questions for a user after passing all test cases for a question.	2
5	Error message after a user submitted a solution for testing.	2
6	Control Flow Graph for Palindrome	4
7	Control Flow Graph for TwoSum	4
8	Control Flow Graph for Binary Search	6
9	Control Flow Graph for Arranging Coins	6
10	Control Flow Graph for postorder	7
11	Control Flow Graph for Max Depth	7
12	Breadth-First Search. The grey arrows indicate the traversal path.	7
13	Unclear example for ID 654	9

LIST OF TABLES

I	Algorithms chosen for this project	2
II	The standard algorithms and data structures	9
IV	The expected difficulty each type of categorised user would have with the questions	9
III	Total of 36 solutions we analysed for this project	10

Analysing Algorithmic Patterns Based on Real Coding Interview Questions

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

Supervisor: Hao Wu
Computer Science Department,
Maynooth University,
Email: haowu@cs.nuim.ie

Abstract—This thesis aims to analyse algorithmic patterns based on real coding interview questions. This is because programming questions have become staples in any interview with technology companies. To perform detailed analysis, we first select a small set of highly used interview questions, simulate and complete each question within restrictive time allowances. This allows the solutions created to be as authentic as possible. We then plot control flow graphs for each selected question and identify the pattern by highlighting the key differences in different algorithms. Further, this thesis shows that mastering a small set of questions allows interviewees to prepare a wide range of key topics to be interviewed. Thus, the identified patterns are important and can be used to save a significant amount of time on preparing long and exhausting coding practice before the interview.

I. 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 perform some form of technical tests. These tests normally involve several questions that are based on a wide variety of topics in algorithms and data structures. These types of interview questions typically require a candidate to understand and analyse questions, then write a piece of code that is bug-free. This is quite challenging and demanding for an even experienced programmer. To prepare this kind of interview, a candidate usually goes through as many algorithmic questions as possible. This process is very time consuming and not easy to manage as a question bank usually has more than 500 questions and many of them are very challenging. They require a candidate to go over solutions multiple times to truly master them.

The purpose of this thesis is to study the relationship between the standard (fundamental) algorithms and the interview questions that are commonly asked. This thesis aims to give a guidance to people who are willing to spend much less time on preparing the interview questions and in the meanwhile cover a wide range of frequently interviewed topics.

This thesis presents a study that summarises the similarities between interview questions and standard algorithms taught in a standard 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.

The contribution of this thesis can be summarised as the following:

1. We view mastering the fundamental algorithms as a key to cover a wide range of real world interview questions.
2. We identify patterns that are closely related to these standard algorithms.
3. We believe that this thesis can be used as a guidance for fresh graduates which can prepare them for real world interview questions.

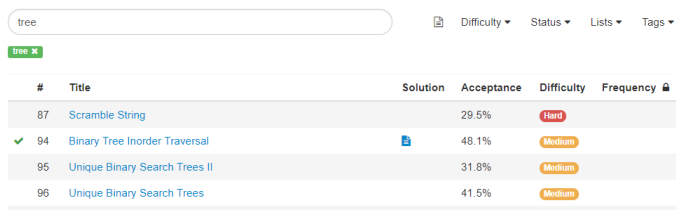
II. BACKGROUND

In general, there are quite a lot of online repositories showing interview questions and interview related material. For example, websites like hackerrank.com, glassdoor.ie, geeksforgeeks.org, and leetcode.com offer information on what questions are common during interviews for different companies. These websites also provide an online judge system to run potential solutions to some of the given questions.

For this thesis we choose to use leetcode.com[1], as this website comes with a huge amount of online material that is 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 is important in our decision for the repository, as it means that we are using a platform that is regularly updated. LeetCode itself also has built-in features that greatly appeal to us. The website has an online discussion forum for each question, allowing the community to discuss solutions, issues, and the interview questions themselves. LeetCode host their own weekly coding competitions that 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 is 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.

We choose Java for this project. The reason for this is because Java is widely used. However, our approach is applicable in other programming languages.

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



#	Title	Solution	Acceptance	Difficulty	Frequency
87	Scramble String		29.5%	Hard	
94	Binary Tree Inorder Traversal		48.1%	Medium	
95	Unique Binary Search Trees II		31.8%	Medium	
96	Unique Binary Search Trees		41.5%	Medium	

Fig. 1: The homepage of LeetCode allows users to search for specific types of questions.

A user simply chooses the question that 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.

Given a binary tree, return the *inorder* traversal of its nodes' values.

For example:

Given binary tree `[1,null,2,3]`,

```

  1
   \
    2
   /
  3

```

return `[1,3,2]`.

Fig. 2: The question page displays a description of the question and gives a concise example.

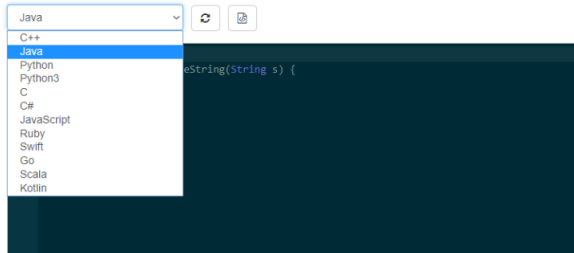


Fig. 3: Online editor with options displayed allowing candidate to use different programming languages.

The users attempt to solve the problem they have selected, and they submit their answer and test if it can pass all test cases. 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 test case or even their code failed to compile and run, shown in Figure 5. In the case of failure, LeetCode reports where an error is in the code by referencing the line.

Submission Result: **Accepted** [More Details](#)

Next challenges: [Reverse Vowels of a String](#) [Reverse String II](#)

Fig. 4: Acceptance message with suggested questions for a user after passing all test cases for a question.

Submission Result: **Runtime Error** [More Details](#)

Runtime Error Message: `Line 15: java.lang.ArrayIndexOutOfBoundsException: -1`

Last executed input: `"hello"`

Fig. 5: Error message after a user submitted a solution for testing.

III. PROBLEM STATEMENT

We formulate the problem statement of this thesis into the following three research questions:

- **RQ1:** What algorithms to choose, and what categories are these algorithms associated with.
- **RQ2:** How do these algorithms relate to the fundamental algorithms and how similar are they.
- **RQ3:** How easy is it for a person to identify the pattern of the algorithm.

IV. THE SOLUTION

To answer RQ1, we first collect useful information. For example, many companies take their questions from online repositories such as LeetCode, and websites that discuss experiences people had at interviews, such as GlassDoor [2]. We also read through interview preparatory books. These books typically list a range of topics covered in interviews, such as Cracking the Coding Interview covers Binary Search, Dynamic Programming and Graph Traversal [3]. Further many websites publish the top real-world interview questions. This is based on the user interview experience. For example, GeeksforGeeks has a list of the top interview questions that a user should study prior to an interview, for example Linked Lists, Binary Search Trees and Sorting [4]. For these reasons, this thesis focuses on the following algorithms:

Algorithms
Sorting Algorithms
Searching Algorithms
Graphs
Trees
Dynamic Programming

TABLE I: Algorithms chosen for this project

In each of these areas we categorise them into seven key patterns (see Table II). These are patterns of the listed algorithms that commonly appeared.

Among the many websites and interview books, we discovered Leetcode is possibly the best resource for preparing for interviews. Therefore, we select thirty-six questions. These thirty-six questions are representative of the five algorithms in Table I. The questions that we choose are based on online voting and the tags associated with the question. LeetCode's system allows users to vote on questions, indicating that the questions appeared frequently in interviews. We also consider the pass rate and reputation of the question. Relating back to Figure 1, it can be seen that questions can be filtered. Leetcode has over six hundred questions, which is impossible to answer in such a short period of time.

With regards to RQ2, in order to find the most similar algorithm, we first compare the ideas of the standard algorithm and the solution to the question we were solving. It is possible to have multiple algorithms apply to a problem. We implement the easiest solution, this being the first that comes to our minds. This is because one normally has limited time in an interview, and a person builds on an idea they initially discover.

To try and obtain a numerical representation of similarity between the standard algorithm and the solution we create, we decide to count how many edit operations we require to transform one piece of code into another. These edits are categorised into two operations, adding and deleting code.

As this would be a considerable task to perform by hand, we initially consider an online text comparator called DiffChecker [5]. DiffChecker returns the number of edit operations required to transform one piece of code into the other. However, we found that DiffChecker fails on a logical level. For example, the same logic structure but with different variable names will be returned as different by DiffChecker. Therefore, these tools can only compare the algorithm on a text level.

Secondly, we consider tools used for plagiarism detection, for example Moss. These tools can be used to detect differences on a logical level. However, using these tools is not easy, as there is a lack of documentation for users and it is laborious to use [6].

Therefore we decide to combine the text comparison and logical comparison approach to form our solution to RQ2. Our approach first judge's similarity by using our own intuition. Secondly, we use control flow graphs (CFG) to manually verify the logic [7]. Given the timeframe this is possibly the most suitable approach to make sure this project meets the deadline. Our approach can be seen in the sections that follow. We first display the standard algorithm, then a solution to a question we have judged to be related. Finally, we list the two CFGs for the standard algorithm and the question we have chosen.

We choose control flow graphs because they are simple to create and understand. They show the natural flow of the program and this allows for easy evaluation of similarity to a standard pattern. Adding or deleting a node or edge was counted as one operation. We total the edit operations and if

the number is below ten, these two pieces of code were similar. Anything above ten is regarded as being dissimilar.

We now present the identified algorithms in Table II in the following section. We display the pseudocode for the algorithms and the control flow graphs for each. We display only a few to represent all of the questions due to the page limit. Please see Table III for more detail. All the control flow graphs can be found in Appendix B of the Supporting folder of this thesis.

A. Palindrome

Palindromes are extremely useful for searching. As Palindromes are the same in both directions, one can easily discern if the input is a palindrome. This is helpful for searching as one can search and find any odd characters, or the unique piece of data in some text. This style of algorithm is space efficient. It has 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 two pointers starting at either end of the input, moving towards one another and comparing if the elements are the same.

Algorithm 1 is a general algorithm for solving the palindrome problem. This approach can be used to solve numerous other problems by altering the inside of the loop.

<p>Input : Given input of characters, S</p> <p>Output: Boolean</p> <pre> 1 leftIndex ← S[0] ; rightIndex ← S.length - 1 2 while leftIndex < rightIndex do 3 compare leftIndex with rightIndex 4 if leftIndex != rightIndex then 5 return false 6 end 7 leftIndex++ 8 rightIndex++ 9 end 10 return true </pre>
--

Algorithm 1: The Palindrome Algorithm

ID 1 TwoSum is one example where the palindrome algorithm was used to solve the question. 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." Algorithm 2 is the solution accepted by LeetCode. In this pseudocode we have highlighted any differences in blue, this is the same for all the following answers.

Algorithm 2 shows the main difference is inside the while loop. The loop guard is still identical, and the core idea found inside the loop is based off the same requirements. The key difference is that the moving of the pointers is not performed at the same time, but rather when certain conditions are met. It can be seen that understanding the need to check the values at the positions, and performing a particular action as a result,

Input : Array of integers nums, target S

Output: Indices i,j

```

1 leftIndex ← S[0] ; rightIndex ← S.length - 1
2 while leftIndex < rightIndex do
3   Sort the input array nums
4   if nums[leftIndex]+nums[rightIndex] > S then
5     rightIndex--
6   else if nums[leftIndex]+nums[rightIndex]<S
7     then
8     leftIndex++
9   else
10    return leftIndex, rightIndex
11 end
12 return 0;

```

Algorithm 2: LeetCode Q1 TwoSum

is the fundamental idea of both the standard algorithm and the specific solution for Algorithm 2.

Each node in the graphs represent one or more statements in the source code. The lines of code that they represent are next to each node. With regards to the two graphs, Figure 6 has six nodes and six edges, a total of twelve. In comparison, Figure 7 has eight nodes and ten edges, a total of eighteen. Therefore, six edit operations are required transform one into the other; two additional nodes and four edges. This result only proved to back up the decision that the answer to TwoSum is analogous to the palindrome algorithm.

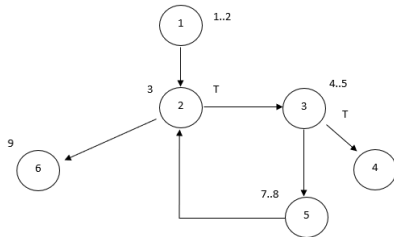


Fig. 6: Control Flow Graph for Palindrome

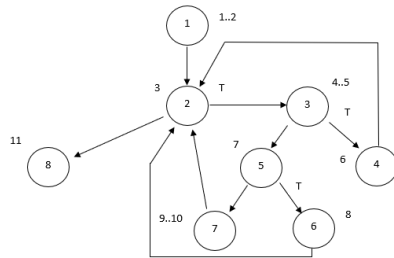


Fig. 7: Control Flow Graph for TwoSum

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's style of algorithm is known as divide and conquer. It is a simple idea, but in practice contains a lot of parts. We see this algorithm as one of the most useful to learn. The merge aspect of it is used in many other areas. Algorithms 3 and 4 show this.

Input : List of unsorted data A, int left, int right

Output: Sorted List

```

1 if left == right then
2   return A[left]
3 else
4   mid ← (left + right)/2
5   recMergeSort(A, left, mid)
6   recMergeSort(A, mid + 1, right)
7   merge(A, left, mid + 1, right)
8 end

```

Algorithm 3: recMergeSort(). The Merge Sort Algorithm through Recursion

Input : List of unsorted data A, int left, int mid, int right

Output: Sorted List A

```

1 n1 ← mid - left + 1 ; n2 ← right - mid;
2 temp1[] ← [n1] ; temp2[] ← [n2];
3 /* Copy data into each temparray */
4 for i ← 1 to n1 do
5   copy
6 end
7 for i ← 1 to n2 do
8   copy
9 end
10 i ← 0 ; j ← 0 ; k ← left
11 while i < n1 && j < n2 do
12   if temp1[i] <= temp2[j] then
13     A[k] = temp1[i]; i ++
14   else
15     A[k] = temp2[j]; j ++
16   end
17   k ++
18 end
19 while i < n1 do
20   A[k] = temp1[i]; i ++; k ++
21 end
22 while j < n2 do
23   A[k] = temp2[j]; j ++; k ++
24 end
25 return A

```

Algorithm 4: merge(). The merge method for Merge Sort

As can be seen, Merge Sort is quite a long algorithm. It contains many intricate parts which must be correct in order for the algorithm to work correctly. **ID 23 Merge K Sorted Lists** is an example of Merge Sort being used to solve a problem. The question is defined as: "Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity." Algorithms 5 and 6 are the solution to this question.

```

Input : ListNode[] A, int left, int right
Output: Sorted List[] A
1 if left == right then
2   | return A[left];
3 else
4   | mid ← (left + right)/2
5   | l1 ← recMergeSort(A, left, mid)
   | l2 ← recMergeSort(A, mid + 1, right)
   | merge(l1, l2) /* See Algorithm 6 */;
6 end

```

Algorithm 5: recMergeSort() function Merge K Sorted Lists

```

Input : ListNode l1, ListNode l2
Output: Sorted List
1 if l1 == null then
2   | return l2
3 end
4 if l2 == null then
5   | return l1
6 end
7 if l1.val < l2.val then
8   | l1.next = merge(l1.next, l2)
9   | return l1
10 end
11 l2.next = merge(l1, l2.next)
12 return l2

```

Algorithm 6: merge() function for **ID 23**

Comparing this solution which is based off of Merge Sort, it can be seen that the main difference is inside the merge method, Algorithm 6. On the other hand, Algorithm 5 is very similar to the standard algorithm (Algorithm 3). Therefore, understanding how to split the data into equal parts is fundamental to both algorithms. It is part of the underlying pattern. The second area for both algorithms is their respective merge methods. Both do differ, this is mainly due to them having different data structures. The key idea that the value of each element of one sub list, should be compared to the other sub list's elements value exists in both Algorithm 4 and 6.

C. Binary Search

Binary Search is a rapid algorithm for locating data. It has a time complexity of $O(\log n)$ and is used when working with sorted arrays. The algorithm is also used when using

binary search trees. In this section the examples are shown with arrays. This algorithm compares the target value to the middle value of the array. If the target is less than this middle value, then the upper half is eliminated as the target cannot lie in this section. The search continues on the remaining half until the search is successful or not. Algorithm 7 shows the standard Binary Search algorithm.

```

Input : Sorted array S, target x
Output: index of target or -1
1 left ← 0 ; right ← S.length - 1
2 while left <= right do
3   | mid ← left + (right - left)/2
4   | if S[mid] == x then
5     | return mid
6   | end
7   | if S[mid] < x then
8     | left = mid + 1
9   | end
10  | else
11    | right = mid - 1
12  | end
13 end
14 return -1

```

Algorithm 7: Binary Search

ID 69 Sqrt(x), is just one example of Binary Search being employed to solve a question. This question is described as: "Implement int sqrt(int x). Compute and return the square root of x. x is guaranteed to be a non-negative integer. Algorithm 8 is the solution to this problem.

```

Input : Int x
Output: Int y
1 if x == 0 then
2   | return 0
3 end
4 left ← 1 ; right ← x/2
5 while true do
6   | mid ← left + (right - left)/2
7   | if mid > x/mid then
8     | right = mid - 1
9   | end
10  | else
11    | if mid + 1 > x/(mid + 1) then
12      | return mid
13    | end
14    | left = mid + 1
15  | end
16 end

```

Algorithm 8: Sqrt(x)

A second example of a question from leetcode.com which is based on Binary Search is **ID 441 Arranging Coins**. The description for this questions is: "You have a total of n coins

that you want to form in a staircase shape, where every k -th row must have exactly k coins. Given n , find the total number of full staircase rows that can be formed. n is a non-negative integer and fits within the range of a 32-bit signed integer.” Algorithm 9 is the solution to this problem.

```

Input : Int n
Output: Number of full steps completed
1  $low \leftarrow 1$  ;  $high \leftarrow n$ 
2 while  $low \leq high$  do
3    $mid \leftarrow low + (high - low)/2$ 
4   if  $mid * (mid+1)L \leq 2L*n$  then
5      $low = mid + 1$ 
6   end
7   else
8      $high = mid - 1$ 
9   end
10 end
11 return high;

```

Algorithm 9: Arranging Coins

Both of the solutions we present in Algorithm 8 and 9 are extremely close to the original pattern of Binary Search. Both have a calculation to obtain the middle index and both compare the middle index value against the given value using a conditional. Both use a loop structure to cycle through the given data and/or value too. This shows that understanding the need to calculate the middle index and use this for comparisons against the given information will allow a user to solve multiple problems.

The total number of nodes in the original pattern is eight and it contains nine edges. In the specific version for Arranging Coins there are a total of six nodes and seven edges. The total is therefore four edit operations required to transform between the two pieces of source code.

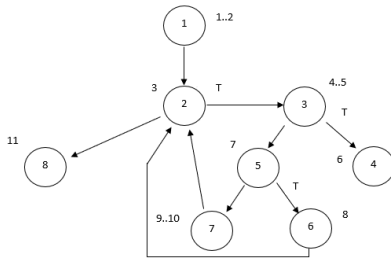


Fig. 8: Control Flow Graph for Binary Search

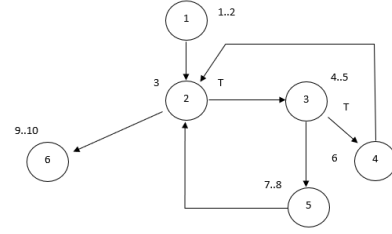


Fig. 9: Control Flow Graph for Arranging Coins

D. Graphs

Graphs are common in our lives. 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 we focus on the Tree Abstract Data Type, Breadth-First Traversal and Graphs in general.

1) *Trees*: Trees are one of the most powerful styles of data structures for processing data. They allow rapid searching and fast insertion/deletion of a node. Trees are made up of nodes, which are objects which hold some data and have a key. This key allows one to determine where this node should be in the tree. 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 Search Tree is a special type of tree. It has strictly between zero and two children, and keeps their keys in sorted order. It has $O(n)$ space, search, insertion and deletion operations.

With trees the main function is traversal. There are three basic styles of traversal: inorder, preorder and postorder. Inorder visits every node in the left subtree, then the root and finally the right subtree. Preorder visits the root first, followed by the left subtree and then the right subtree. Finally, postorder is where the left subtree is followed by the right subtree and then the root.

Algorithm 10 is an example of how one might search for a particular node in a Tree. Algorithm 11 is the basic postorder traversal of a tree. The key area to note is the order of calling the different children of a node. Altering this will result in preorder and inorder traversal.

ID 104, Maximum Depth of Binary Tree, is an example of a question where the basic tree traversal styles were used 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." Postorder traversal was used. Algorithm 12 details the solution.

The similarities between Algorithms 12 and 11 are that in each case we recursively traverse down the left branch of the tree, followed by the right branch and finally we consider the root node. The key idea to understand from the original

Input : Given a key to search for
Output: The desired Node, or null

```

1 Node current ← root;
2 while current.data ≠ key do
3   if current == null then
4     return null
5   end
6   if current.data > key then
7     move left on the tree
8   else
9     move right on the tree
10  end
11 end
12 return current;

```

Algorithm 10: Finding a specific Node in a tree based on the key

```

postOrder(Node localRoot)
if localRoot ≠ null
then
  postOrder(localRoot leftChild)
  postOrder(localRoot rightChild)
  Print(localRoot data)
end

```

Algorithm 11: Basic Tree Traversal using PostOrder Traversal

```

maxDepth(TreeNode localRoot)
if localRoot == null then
  return 0
end
else
  ldepth = maxDepth(localRoot.leftChild)
  rdepth = maxDepth(localRoot.rightChild)
  if ldepth > rdepth then
    return ldepth + 1
  end
  else
    return rdepth + 1
  end
end

```

Algorithm 12: Leetcode Q104 Max. Depth of Binary Tree

algorithm is that by traversing down one branch at a time, the user can obtain the longest path in the tree. By initially going down the left and only the left, the user can obtain the value for this branch. When this left branch of the tree has been completely searched and the path value stored, then and only then will the right branch be traversed in the same manner. Therefore, having the knowledge that traversing a tree in a specific way can give a user information is vital and is what the user should aim to understand with tree traversal patterns.

This similarity is only reinforced when the CFGs are considered. Figures 10 has four nodes and 6 edges, Figure 11 has five nodes and five edges. Therefore, they have an identical total of ten. This shows the two algorithms are closely related, as detailed in the previous paragraph.

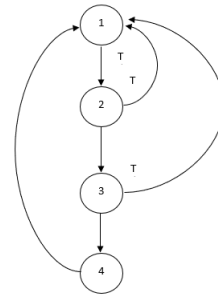


Fig. 10: Control Flow Graph for postorder

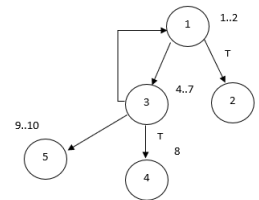


Fig. 11: Control Flow Graph for Max Depth

2) **Breadth-First Search:** This is a special traversal pattern. The ordering visits 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 12 shows the nodes in numerical order of visitation.

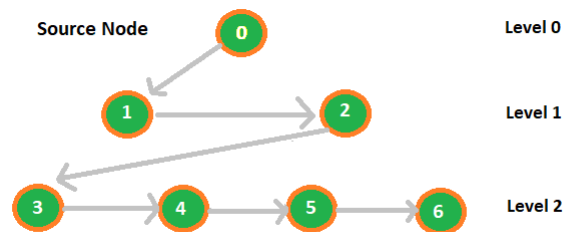


Fig. 12: Breadth-First Search. The grey arrows indicate the traversal path.

3) *Depth-First Search*: This pattern involves starting at the root, then going to the left most child, repeating this movement until the traversal reaches a leaf node, then it will move back up one node and try to visit the next child node. It repeats this until the traversal finds no unvisited node.

E. Dynamic Programming

Dynamic Programming is outlined as ‘smart recursion’. This is because it takes the idea of recursion and improves on the inefficiencies of recursion. These inefficiencies mainly being that recursive algorithms are generally space inefficient. Each recursive call adds a new layer to the stack. Therefore, dynamic programming is such a strong style of programming as it uses the simplicity of recursion but does not have the problem of causing a stack overflow. Dynamic programming is described as 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 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 end
4 return  $fib(n - 2) + fib(n - 1)$ 

```

Algorithm 13: Fibonacci Numbers through normal recursion

1) *Top-Down*: This method of dynamic programming follows the normal structure of a simple recursive solution. Start from n and compute smaller results as they are needed to solve the original problem. Results are stored in some form of data structure, this is known as memoization. These stored results are accessed efficiently in comparison to having to recompute the same values for n repeatedly. See Algorithms 14 and 15.

```

Input : Integer n
Output: Fibonacci Numbers
1  $computed \leftarrow \text{HashMap}$ ;
2 if  $n < 3$  then
3   | return 1
4 end
5 return  $fib(n - 2) + fib(n - 1)$ 
6  $computed.put(1, 1)$ 
7  $computed.put(2, 1)$ 
8 return  $fib2(n, computed)$ 

```

Algorithm 14: fib; Fibonacci Numbers through Top-Down and Memoization

2) *Bottom-Up*: The issue with the Top Down version is that even though the results are being saved to reuse later, they must be calculated from n to the base cases through recursion the first time. This is performed only once, when

```

Input : Integer n, HashMap computed
Output: Fibonacci Number
1 if  $computed.containsKey(n)$  then
2   | return  $computed.get(n)$ 
3 end
4  $computed.put(n - 1, fib2(n - 1, computed))$ 
5  $computed.put(n - 2, fib2(n - 2, computed))$ 
6  $newVal \leftarrow$ 
    $computed.get(n - 1) + computed.get(n - 2)$ 
7 return  $newVal$ 

```

Algorithm 15: fib2; Fibonacci Numbers through Top-Down and Memoization

no values have been computed. Bottom-Up solves this by starting at the base cases and computing until it reaches the value of n . See Algorithm 16.

```

Input : Integer n
Output: Fibonacci Numbers
1  $results[] \leftarrow \text{int}[n + 1]$  ;  $results[1] = 1$ 
2  $results[2] = 2$ 
3 for  $i \leftarrow 3$  to  $n$  do
4   |  $results[i] = results[i - 1] + results[i - 2]$ 
5 end
6 return  $results[n]$ 

```

Algorithm 16: fibDP; Fibonacci Numbers through Bottom-Up

It must be noted 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 such a limited window. The reason it is tough is because a candidate must think of the problem in multiple ways, and make the decision to tackle it in a certain manner. If they choose the wrong option, then they will end up with an inefficient solution. Choose correctly and they will have shown a very strong understanding of this area.

The important aspect of this pattern is storing the calculated information in a data structure. It therefore allows for rapid lookup and less calculations to be on the stack at any given time.

V. EVALUATION

In the previous section of this thesis, only specific examples of the solutions were shown. Here we give a detailed summary of all solved questions in Table III. The solutions to these questions passed all test cases and were accepted by LeetCode’s online judging system. We also create a table for the algorithms previously listed in the solution section, and a table for the data structures used in their solutions, Table II.

In Table III, *Question ID* indicates a specific ID of LeetCode question we solved. *PassRate* is the percentage of solutions accepted by LeetCode. These were taken at the time of writing.

	Algorithms
1	Palindrome
2	Merge Sort
3	Binary Search
4	Tree Traversal
5	Breadth-First Search
6	Depth-First Search
7	Dynamic Programming
-	None required

	Data Structures
1	List
2	Trees
3	Stacks
4	Queues
5	Arrays
6	Maps
7	Strings
-	None required

TABLE II: The standard algorithms and data structures

Difficulty is the level that was awarded to each question by LeetCode. *Time* is the total time in minutes it took to complete each question. *Evaluation* is our opinion on the difficulty of each question. *Algorithm Pattern* implies the algorithms and data structures pattern we identify and are similar to the standard algorithms in Table II. Each pattern is enclosed in a list. The first component in this list is a number that refers to a similar standard algorithm. The second component is a list that refers to the relevant data structures (in Table II) used. For example, [4, [2]] means it uses *Tree Traversal* (from algorithm 4 in Table II) and *tree* data structure from Table II. When a "-" is used inside the algorithm part of our notation, this means that the algorithm used to solve the question was not one which was named as part of the five standard algorithms. When it is used in the data structure part our notation, it means that no data structure was explicitly needed. For the source code of each solution listed in Table III, please view our online repository:

<https://github.com/Demostroyer/interview-project/>

As can be seen in Table III, some of the questions that are marked as having difficulty *easy*, took longer than some that are marked as *medium*. This is because it takes longer for the pattern that is useful to solve the problem to be discovered.

To answer RQ3 we describe the different types of users that potentially will prepare interview questions. It is not easy to judge how fast a user can detect 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) People which have a background in coding competitions. They are highly experienced and unlikely to find the questions hard.
- 2) Users with some computer science background, who understand the internal workings of data structures and other concepts. These people would find most problems approachable.
- 3) People who understand basic concepts. They will struggle with the advanced topics.
- 4) People who are interested in learning to code who have no experience. They will have to put a lot of study in to grasp basic concepts.

This leads to the creation of Table IV. Here each type of user is

paired with the expected difficulty they would face answering the questions on LeetCode.

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

TABLE IV: The expected difficulty each type of categorised user would have with the questions

A. Question Evaluation

Whilst working on the numerous questions, some issues did arise that related to the questions themselves. These problems stem from the questions being worded abnormally, or the example given for the solution not being explicitly clear in the way the question operated. These were just some of the issues that occurred whilst attempting to solve the problems. An example of such an issue is **ID 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 that we personally came across was trying to understand the way the left and right subtree should be structured. The description for them was not clear, as we were unsure what was meant by "left part subarray divided by the maximum number". The provided example is as follows.

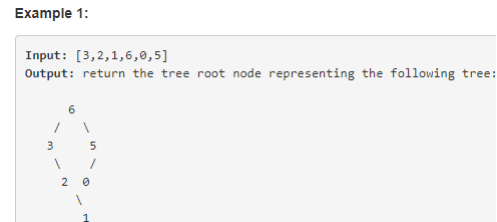


Fig. 13: Unclear example for ID 654

This did not make it any clearer. It was simple to understand that the biggest number in the subarray should be the next number in the tree and to work in a descending manner. What is hard to understand is the two being the right child of the three, and yet zero was to the left of five, where it was to be expected. This confusion caused a delay in the solution to this question being completed.

VI. LIMITATIONS

There are two limitations in this thesis.

Question ID	PassRate	Difficulty	Time(in minutes)	Evaluation	Algorithm Pattern
1	36.0%	Easy	9	Easy	[1,[5]]
2	28.1%	Medium	20	Medium	[-,[1]]
20	33.7%	Easy	12	Easy	[1,[3]]
21	39.3%	Easy	25	Easy	[-,[1]]
23	27.7%	Hard	27	Medium	[2,[1,5]]
35	39.8%	Easy	10	Easy	[-,[5]]
53	39.9%	Easy	20	Medium	[-,[5]]
58	32.0%	Easy	15	Easy	[-,[5,7]]
69	28.2%	Easy	19	Easy	[3,[-]]
70	40.5%	Easy	18	Easy	[7,[5]]
72	32.1%	Hard	29	Hard	[7,[5]]
88	32.1%	Easy	12	Medium	[-,[5]]
94	47.8%	Medium	27	Medium	[4,[1,2,3]]
102	40.9%	Medium	30	Medium	[5,[1]]
104	53.5%	Easy	28	Medium	[4,[2]]
111	33.3%	Easy	17	Easy	[5,[2,4]]
114	35.7%	Medium	33	Medium	[4,[2]]
136	54.9%	Easy	10	Easy	[-,[5]]
147	33.4%	Medium	26	Medium	[-,[1]]
152	26.2%	Medium	31	Hard	[7,[5]]
167	47.1%	Easy	16	Easy	[1,[5]]
169	47.3%	Easy	20	Easy	[-,[5]]
205	34.2%	Easy	18	Easy	[-,[6]]
230	44.1%	Medium	19	Medium	[4,[2,3]]
290	33.2%	Easy	28	Medium	[-[5,6]]
326	40.4%	Easy	25	Easy	[-[-]]
336	26.5%	Hard	18	Medium	[1,[1,7]]
344	59.5%	Easy	8	Easy	[1,[5,7]]
387	47.1%	Easy	19	Medium	[-,[5]]
389	50.9%	Easy	10	Easy	[-,[5]]
404	47.3%	Easy	16	Medium	[4,[2]]
441	36.3%	Easy	6	Easy	[3,[-]]
654	70.1%	Medium	46	Hard	[2,[2,5]]
669	58.1%	Easy	18	Easy	[4,[2]]
687	33.5%	Easy	29	Medium	[-,[2]]
690	52.9%	Easy	45	Medium	[5,[3,6]]

TABLE III: Total of 36 solutions we analysed for this project

Firstly, the topics that are covered are only a selection of algorithms, there are many more to be studied. For example, the questions focusing on *Insertion Sort* were never formally addressed in this thesis. This can be improved by solving more algorithms from more topics.

Secondly, our approach has its limitations. It was manual, meaning it was dependent on the ability of us as programmers. This meant that our ability to solve some questions was stronger on certain topics than others. For example, the questions relating to the advanced topic Dynamic Programming took longer to learn and therefore, took longer to complete. This is due to this section being an advanced area. As this approach was manual, there was a lot of human effort required. This results in questions taking longer than expected to be

completed, and questions being solved in ways that might not be the most optimal. This can be solved by developing some form of automatic validator.

VII. RELATED WORK

As was mentioned earlier in this thesis, there are some pieces of literature that focus on analogous topics. These include books that aim to give solutions to common programming questions [3] and the fundamentals of Java [8][9]. There are also some websites that have some mock interview questions and stage the mock interview to be like the normal interview process by having limited time and selected questions [10].

With regards to our approach to finding similarity between the algorithm and solution, there are a number of related works that we have previously mentioned. These being DiffChecker for text comparison and Moss for plagiarism checking.

DiffChecker works by comparing each letter inside the text. It returns any discrepancies found whilst scanning through the text. The issue is that this will return pieces of the text that are logically the same but textually different. In comparison to our approach of using control flow graphs, this is not accurate enough.

Moss is a powerful tool for detecting plagiarism. Moss is described as an automatic system for determining the similarity of programs [6]. The tool itself is heavily focused on finding plagiarised pieces of code from given sources. Moss uses a technique known as robust winnowing. Winnowing is a technique normally found in machine learning and document fingerprinting [11]. This approach is strong, but slow in practice. In terms of the time frame given for this project, we decided it is better to perform the logical check manually. This is achieved by using control flow graphs.

VIII. CONCLUSION AND FUTURE WORKS

In this thesis we have created a guide for beginners in programming, fresh graduates and readers who wish to refresh their knowledge. This 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 approach taken in this thesis was to solve each question manually. This means the solutions were all attempted and completed by a human. Therefore, this process was similar to the interview environment. This is due to both the interview and approach taken for this thesis, having the human element of thinking of multiple ways to tackle the issues. This allowed the approach to be as valid and authentic to the real scenario of the interview as possible. In this thesis 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.

This thesis serves as a pilot study for future improvement. In the future, we plan to use machine learning techniques to predicate a possible algorithmic pattern and data structures to be used to solve a given interview question. This could allow interviewers and candidates to further improve the quality of interview preparation.

REFERENCES

- [1] LeetCode, "Leetcode." <https://leetcode.com>.
- [2] GlassDoor. <https://glassdoor.ie>, 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." <http://www.geeksforgeeks.org/top-10-algorithms-in-interview-questions/>.
- [5] DiffChecker. www.diffchecker.com, Nov. 2017.
- [6] A. Aiken, "Moss." <https://theory.stanford.edu/~aiken/moss/>.
- [7] D. B. D. T. D. P. T. L. D. Ye, *Software Testing Principles and Practice*. China Machine Press, 2013.
- [8] N. Markham, *Java Programming Interviews Exposed*. Wrox guides, Wiley, 2014.
- [9] A. Aziz, T. Lee, and A. Prakash, *Elements of Programming Interviews: The Insiders' Guide*. ElementsOfProgrammingInterviews.com, 2012.
- [10] I. Pramp, "Pramp." <https://www.pramp.com>.
- [11] 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.
- [12] M. Goodrich, R. Tamassia, and M. Goldwasser, *Data Structures and Algorithms in Java*. Wiley, 2014.
- [13] F. Carrano and W. Savitch, *Data Structures and Abstractions with Java*. Prentice Hall, 2003.
- [14] 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.
- [15] M. V. Janičić, M. Nikolić, D. Tošić, and V. Kuncak, "Software Verification and Graph Similarity for Automated Evaluation of Students' Assignments," *Inf. Softw. Technol.*, vol. 55, pp. 1004–1016, June 2013.
- [16] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev.*, vol. 31, pp. 249–260, Mar. 1987.
- [17] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Software Engineering*, Oct 2017.
- [18] M. Chilowicz, E. Duris, and G. Roussel, "Syntax tree fingerprinting for source code similarity detection," in *2009 IEEE 17th International Conference on Program Comprehension*, pp. 243–247, May 2009.
- [19] I. Dempsey, "Analyzing algorithmic patterns based on real coding interview questions," 2017. <https://github.com/Demostroyer/interview-project>.