Section 09: Interview Prep

1. TEBOW-IT Walkthrough

When answering a question in a technical interview, your interviewer isn't just looking to see if you get the right answer – they also want to **see your thought process**. It is useful to have a plan to organize your thinking. Various other resources have similar processes. We will work through the TEBOW IT approach.

1.1. Question

Your interviewer will start by giving you a question. Here's the one we'll be working through:

Given a list of numbers and a target number x, find all pairs in the list that sum to x.

1.2. T: Talk (/Listen)

The prompt you have at this point is vague. Intentionally so! Your interviewer wants to see how you approach formalizing this problem – how you take vague English text and turn it into a precise programming problem.

How do you do that? By asking questions and listening to the answers. Brainstorm some questions you can ask to narrow down the prompt.

Make sure you listen to the answers, and process what your interviewer is saying. If they are feeling nice (or lazy...) they may say "whichever is easier for you" to some of these questions, or volunteer a simplifying assumption you can make. In any case, once we've asked our questions, we have a much more specific prompt.

Given an **array** of `Integers` and a target `Integer` x, **print** all pairs in the list that sum to x.
You may assume there are no duplicates in the list.
Consider (a, b) **and** (b, a) **to be the same pair – print only one of them.**

1.3. E: Examples

Now that we've narrowed down our prompt a little bit, now is the time to really make sure you and your interviewer are on the same page. Come up with a few example inputs and their desired outputs. You'll also use these later to check the algorithm you come up with.

1.4. B: Brute Force

Hopefully, you've been talking for quite a bit of time now, and the back of your brain has come up with how you want to actually solve the problem.

Whether you have or haven't SLOW DOWN. You don't want to just start writing code.

Before you do, give a description of a plan for a correct algorithm. **Any** correct algorithm, even if it's not particularly fast.

How would you describe one possible algorithm?

1.5. O: Optimize

Now that you have some algorithm, it's time to optimize it – try to turn it into the best one you can think of.

Unfortunately, designing algorithms is as much art as science. The best way to get better is to practice. If you're hitting a block, asking yourself questions like these might guide your thinking:

(a) Is this a graph problem?

(b) Is there a data structure we can use to make this easier? A dictionary or a priority queue?

(c) If we made some assumption could we make the problem simpler? Maybe getting rid of duplicate values, or making a graph unweighted, or sorting the list

- Break the problem into smaller pieces – instead of solving the whole thing, can you do one step?
- Sometimes you can turn an algorithm for a special case into an algorithm for the original problem; if you can you should!
- If you can't, it's still worth mentioning. Maybe your interviewer will let you add that assumption. even if they won't, it will show your thought process, which is more important than the actual "answer."

Try designing a faster algorithm for this problem.

You should practice actually saying your thought process out loud. If it has been a few minutes and you feel like you haven't gotten anywhere, your interviewer might give you a hint. There's a possible hint at the top of the solution for this part.

1.6. W: Walkthrough

At this point, you should have a clear idea of what you're going to do. Explain your planned algorithm to your interviewer at a high level. Think comments you would write before writing a method.

1.7. I: Implement

Now, it's time to write actual code. Writing code on a whiteboard is **hard**. You won't have an IDE reminding you of syntax, and you've probably never done it before. Your interviewer shouldn't care about esoteric details of Java code. But the closer you can get to code that would actually compile the better.

Actually practice this part on a piece of paper, or better yet a real whiteboard. Typing it up just isn't good practice.

1.8. T: Test/Debug

This last part is the easiest to forget, but it's really important – make sure your code works!

You have test inputs and the correct answers from way back in the "E" of TEBOW. Use them!

Once you feel confident your code is correct, analyze it for how fast it is (you probably did this in your head back in "O", but you should do it carefully and out loud if you haven't yet)

2. Now TEBOW-IT Yourself!

There are two questions below that have been chosen by your TAs. Pair up with another student (your TA will probably handle this) and take turns. For 10 minutes one of you will attempt the TEBOW-IT method on the given problem while one of you plays interviewer. Then you'll switch roles for another ten minutes!

Note: The solution slides to each problem are **on the website** along with this worksheet. If you're acting as the interviewer, you should have the solution slides open as your partner tries to TEBOW-IT: if they get stuck, give them a hint! Often this is what actual interviewers will do.

2.1. Validate BST

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

A valid BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

You can make the following assumptions:

- The number of nodes in the tree is in the range [1, 10^4].
- -2^{31} <= Node.val <= 2^{31} - 1.
- Assume the tree is made of the TreeNode class below.

```java
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}
```

2.2. Course Schedule

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [a_i, b_i] indicates that you **must** take course b_i first if you want to take course a_i. **For example**, the pair [0, 1] indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.

Constraints + valid assumptions:

- 1 <= numCourses <= 10^5
- 0 <= prerequisites.length <= 5000
- prerequisites[i].length == 2
- 0 <= a_i, b_i < numCourses
- All the pairs prerequisites[i] are **unique**.

3.   Challenge Question: Merge Intervals

Try out the TEBOW-IT method on the following question!

Given an array of intervals where `intervals[i]` = `[startIndex, endIndex]` (so this is a 2D array), merge all over-lapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

You can assume:

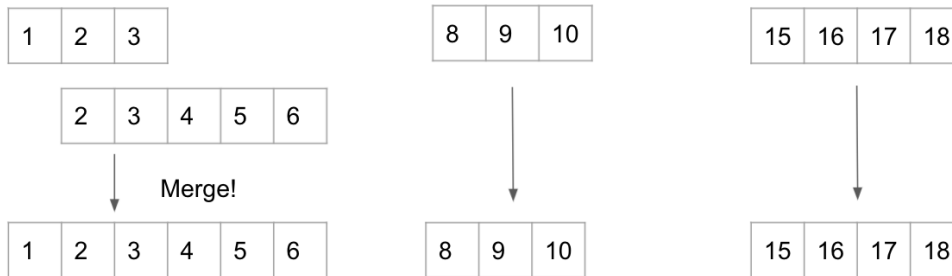`1 <= intervals.length <= 104`

`intervals[i].length == 2`

`0 <= startIndex<= endIndex <= 104`

Note: This ones's pretty tough, so we'll give an example input/output:

Input: `[[1,3],[2,6],[8,10],[15,18]]`

Output: `[[1,6],[8,10],[15,18]]`

Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

4. Bonus Interview Questions

Note: These are bonus questions from a past quarter- we don't have solutions written up quite yet, but they're definitely useful to noodle on :) Post on the Ed board if you think you've got a solution!

4.1. Nodes and Graphs

(a) Write an iterator for a binary tree that returns the elements in level-order (i.e. it first returns the root, then the children of the root at level 1, etc.).

(b) You have two very large binary trees containing many nodes. The first tree A contains millions of nodes; the second tree B contains hundreds to thousands. Create an algorithm to decide if B is a subtree of A.

Note: the tree B would be a subtree of A if there exists some node n in A where n and all its children are exactly equivalent to B.

(c) What if we have many different trees and want to check each one to see if they're a subtree of A? How could we do this efficiently? Assume each of these trees contain only a few hundred to thousand elements.

(d) Suppose we have a graph containing both directed and undirected edges. If we ignore the undirected edges, we know for certain that the directed edges taken together do *not* introduce acycle.

Your job is to implement an algorithm that assigns each of the undirected edges a direction such the graph remains acycle once there are no more undirected edges.

4.2. General

(a) Implement a method named `fizzBuzz` that prints out the numbers from 1 to 100. If the number is a multiple of 3, print out "Fizz" instead of the number. If the number is a multiple of 5, print out "Buzz" instead. If the number is a multiple of 15, print out "Fizzbuzz' instead.

(b) Imagine you have a chess knight on a old-school phone dialpad. Given some starting position and a length, how many phone numbers can the knight dial?

For example, if the knight starts on 1 and dials numbers of length 3, it can dial '161', '167', '160', '181', and '183' for a total of 5 numbers. Describe the time and space complexity of your solution.

(c) Given some stream of data (let's say ints), we would like to return a (true) random subset of size K of that stream. Assume that the stream is very large – we are not able to store every single int we receive in memory.

(d) Suppose we have some steadily incoming stream of data. We want to be able to find the median of all the data we've seen so far as quickly as possible.

How would you do this? Can you figure out a solution to do so in $\mathcal{O}(1)$ time?

(e) Suppose we want a map that keeps track of not only key-value pairs, but also keeps track of an *expiry time* per each pair. If the expiry time for a pair is past, that pair should no longer be considered a part of the map.

(f) Suppose we want to implement an algorithm that accepts an integer and returns the equivalent Roman numerals. For example, the number '597' would be DXCVII in Roman numeral.

For a review of how Roman numerals work, see https://tinyurl.com/yb736cb5