Simple algorithm questions, loosely inspired by my experience with fileutil and gfsdu

a) Given an array of queues, where pop() is a very expensive operation (remote network request), find the shortest queue.

b) Given an Array of Queue<int>, where pop() is a very expensive operation, find the queue with smallest sum.

Part 1: Given a function randInt(a,b) that returns a random integer between a and b inclusively, write a function generateBingoCard() that randomly generates a bingo card.

For further clarification: a bingo card is a 5x5 grid of squares, each containing a distinct number (except for the center space, which has no number). The numbers for the first column can be from the range 1-15, with subsequent columns having ranges 16-30, 31-45, 46-60 and 61-75. I ask for output as a column-first 2D array (e.g. [[1,2,3,4,5],[16,17,18,19,20],[31,32,33,34],[46,47,48,49,50],[61, 62,63,64,65]]).

Part 2: Building on the first function, create a function generateBingoCards(N) that returns N bingo cards, with the stipulation that every column on every bingo card is distinct from every other column on every other bingo card (they cannot have the same numbers in the same order). If candidates ask about the size of N, I tell them it is potentially large, but far smaller than the possibility space of a single column.

I ask this series of questions as a warm-up. It identifies unqualified candidates quickly, and it gives good candidates a confidence-builder before diving into something more challenging.

For each question, I ask the candidate to describe the algorithm they would use and to give the running time.

1. Find a particular element in an unsorted array.
2. Find a particular element in a sorted array.
3. Find the intersection of two unsorted arrays.
4. Find the intersection of two sorted arrays.

5. Given a sorted array with duplicate values, count the number of occurrences of a given value.

For most candidates, these questions are easy and we quickly move on. If these turn out to be just the right level of difficulty, I can ask them to write code for #4.

Sometimes I will use #5 as a coding question. The most common solution is to do two binary searches to find the lower and upper bounds, and then subtract the indices. But I don't want to ask candidates to code binary search on the whiteboard, so I guide them towards a different solution by hinting "divide and conquer".

BT is a binary tree (assume more or less balanced) where every node also has a string tag. This is not a binary search tree, (no assumptions on string values) and the same string can appear in many unrelated nodes. Define an IsSimilar method that takes two trees and returns true/false. Identical trees are similar. If two trees are similar, swapping the left_child and right_child pointers of any number of nodes still result in similar trees.
Notes: I think this is a relatively easy question, suitable on occasions such as:
- Phone interviews,
- Interviewing less experienced candidates,
- As a second question (that shouldn't take more than half the interview time) on on site interviews.

There are (at least) three canonical solutions. Coming up with the naive solution is easy. Finding an improvement (one of the other canonical answers) takes some thought. Analyzing the run time is not trivial. The candidate will not need to write much code but will need to be careful about null values.
To get a good score a candidate will need to have good understanding of binary trees, ability to write recursive functions and ability to do run time efficiency analysis.
1. Naive Solution:
(Augment the data structure with NULL nodes)

```
bool IsSimilar(BT left, BT right) {
  if (left == NULL || right == NULL)
    return left == NULL && right == NULL;
  if (left.value() != right.value()) return false;
  if (IsSimilar(left.left_child(), right.left_child())
```

```
    return IsSimilar(left.right_child(), right.right_child();
  return
    IsSimilar(left.left_child(), right.right_child()) &&
    IsSimilar(left.right_child(), right.left_child();
}
```

Follow up: What is the run-time complexity?
The answer is O(n^(log 3)) (i.e. it takes 3^n time on an input of size 2^n) but it will take a above average candidate to see that.
Follow up: Find a solution with better run time complexity.
Candidate Question: Can I 'destroy' the given input?
Answer: Yes.
2. Better and Longer Solution:

```
enum Comparison { LESS, EQUAL, GREATER }

// Define a total order on all binary trees.
// For example: NULL is less than every other node.
// For non-null nodes compare string values, left childs,
// and right childs in 'dictionary order'.
Comparison Compare(BT left, BT right) {
  if(left == NULL) return right == NULL ? EQUAL : LESS;
  if(right== NULL) return GREATER;
  if(left.value() != right.value()) {
    return left.value() < right.value() ? LESS : GREATER;
  }
  Comparison left_comp =
    CompareWith(left.left_child(), right.left_child());
  if (left_comp != EQUAL) return left_comp;
  return Compare(left.right_child(), right.right_child());
}

// A normal form BT is one which for all nodes n, left_child of n
// is less than or equal to the right child of n.
void Normalize(BT tree) {
  // Recursively normalize left and right child.
  // If left child is greater than the right child, swap them.
}

//
bool IsIdentical(BT left, BT right) {
  // return true iff strings of the roots are equal,
  // and left childs are equal, and right childs are equal.
}

bool IsSimilar(BT left, BT right) {
  return IsIdentical(Normalize(left), Normalize(right));
}
```

Follow up: What is the run time complexity?
Compare is O(n), for Normalize we have: O(n) = 2 O(n/2) + n. So it is O(n log(n)) but it'll take a good candidate to see that.
Follow up: Can we have something faster, that'll return the correct

answer with very high probability?
Hint: Can you use hashing for imperfect but faster comparison?
3. Fast & Probabilistic Solution

```
int64 GetHash(BT tree) {
  if (tree == NULL) return some_constant;
  return constant*tree.value().some_hash() +
      (GetHash(tree.left) xor GetHash(tree.right)));
}

bool IsSimilar(BT left, BT right) {
  return GetHash(left) == GetHash(right);
}
```

Pre-question: have you played Go before?

OK, doesn't matter, I'm going to ask a question set on the Go game board.

Go is played on a 19x19 grid of intersections. I'll draw a few (I draw the board corner with 7-8 vertical and 7-8 horizontal lines). Two players take turns putting down white stones and black stones. The notation is... [I draw a little legend beside the picture], black stones are represented with a solid dot like this, and white stones with a hollow dot like this. It's a war game, so there are notions of "alive" and "dead" where dying is about being surrounded by enemy stones.

But only the cardinal directions count! Not diagonals.

I draw a black stone, say it's alive; draw three white stones around it, it's still alive; but one more white stone to surround it, now it's dead.

But there's one complexity, which is that aliveness depends on the aliveness of the whole contiguous group a stone might be part of. So if THIS stone were black instead of white... [I color in a hollow dot]... the first stone is alive again. Does that make sense? [I draw more white stones] Still alive... still alive... dead. And to hammer in the point, if THIS stone were black, the original stone would be alive

again.

So [I write this on the board]:

GIVEN
 * a 19 by 19 array of something (TBD by candidate) and
 * the coordinates of a black stone,

DETERMINE
 * Is it alive?

You are given a function F, which maps non-negative integers to non-negative integers. You know that F is monotonically increasing: if x > y, then F(x) > F(y).

Write code which given such a function F, and in integer y, quickly finds the closest inverse x mapping from y - so that | F(x) - y | is minimized. For example, if F is a function that squares its input, than your function should return 5 when asked to invert 25 (an exact inverse), and 4 when asked to invert 20 (5*5 is further from 20 than 4*4 is).

Why I ask it

passing a function - particularly for Java programmers, it's nice to see if they can define an Interface. For python programmers, you can see if they know lambda notation. For Javascript progammers, you can see if they know the 'call' method. For C/C++ programmers, see if they remember the ghastly syntax for a parameter which is a pointer to a function that maps ints to ints: I think you would say Invert(int(*f)(int), int y) but I'm probably wrong.

O() analysis - easy and concrete way to see if they know the basics: there is an easy O(n) solution, a less-easy O(lg n) which they usually get.

makes good candidates feel good - this is the kind of problem that I've found good candidates are challenged by, but quickly surmount,

which leaves them feeling good, which is nice.

A Variation

I sometimes ask the following variation, as it was based on an actual problem I had in a prior life - you may prefer this formulation:

The API for Microsoft Outlook lets you pass in an integer i, and tells you what your i'th upcoming appointment is. They are ordered chronologically, so that the 0'th appointment is the next one you have coming up. Because of repeating appointments, the sequence is infinite - a birthday, for example, will show up over and over in the output sequence.

How would you quickly find which appointments (if any) you have scheduled for February 28th, 2016?

There are three main ways that candidates trip up:
1) They can't figure out how to pass a function as a parameter. Move on.
2) They try to over-solve it, using Newton-Raphson approximation. This is OK as it gives you good insight, but you might yank them back if they are taking too long.
3) They see the O(n) solution, but don't see the O(lg n). Ask them to go ahead and code the O(n) solution, often they get the O(lg n) one on the way. If they don't, well, then, judge accordingly.

Given an n*n grid, each cell is 0/1. 0 means you can work on it, and 1 means obstacle. You start from the top left cell. Try reaching the bottom right cell with a shortest path.

Warm up: What is the length of the shortest path and how will you find it?
Only need to talk about how to find it. An Breadth-First-Search of O(n^2) is ok.

Main question: If you have a magic that allows you to change one

cell from 1 to 0 in the grid, what is the length of shortest path?
Needs Coding.
Naive solution: try turning each 1 into 0 and run the warm up solution O(n^4).
Expected solution is still O(n^2). Solution 1: recording whether the magic has been used or not during BFS. Solution 2: doing BFS from both start/end cell, and test each cell of 1 with its neighbor cells' shortest paths.

Follow up: What if the magic could be used for at most m times?
Just need to talk about solutions. No coding.

Tricks:
How to move in the grid? (4 direction or 8 direction?)
What if the start/end cells is 1?
In the main question, what if you can reach the shortest path witout the magic?

The game is played on a rectangular board.
Each cell can contain a positive number or a hole.

The game starts at the upper left corner of the board (which contains a positive number).
When standing on a cell with a number X, you can jump EXACTLY X steps in any direction.
You can't land outside of the board or in a hole.

Write a method that computes the maximum number of moves you can make on a given board.

| 2 | - | 8 | 4 |

| - | 1 | 1 | - |

| 3 | - | - | 5 |

In this example, we have two paths - 2 -> 8 and 2 -> 3 -> 5. So the maximum is 2 jumps.

Question: Given a list of strings that represent files, output the Directory structure represented by these files.

Ex:
Input files = [
  “/app/components/header”,
  “/app/services”,
  “/app/tests/components/header”,
  “/images/image.png”,
  "tsconfig.json",
  "index.html",
];

Output:
-- app
    -- components
        -- header
    -- services
    -- tests
        -- components
            --header
-- images
    Image.png
-- tsconfig.json
-- index.html

[How do you elaborate the question to candidates?]

SWE/Coding focussed

I usually start with a warm up question.

Given a string that represents a file, write a function that prints its directory structure. This is just to have them get comfortable writing code and is simple enough. If the candidates start processing the

string char-by-char, I tell them they can use any in-built functions of the language (split, repeat etc) or assume that such functions exist.

I follow this up with the main question and give them an example as written above. I would tell them to assume that we are building an online editor and we need to process this list for the UI.

Let's assume that each computational job at Google requires a constant number of CPUs for a given constant duration. Each scheduled job can thus be represented as follows:

```
class Job {
  public int startTime;
  public int duration;
  public int cpus;
}
```

Google's data centers have a limited amount of CPUs. Therefore, we must ensure that scheduled jobs don't use more CPUs than what is available at any point in time. We'd like you to design an algorithm to verify if a given schedule is valid or not. Inputs of the problem are:

a positive integer maxCpus that represents the total amount of available CPUs
an array of jobs scheduled at a given time.

Your solution should return false if the schedule exceeds the CPUs limit at some point in time; it returns true otherwise.

Details

- Jobs are already scheduled, i.e., the startTime is fixed.
- Jobs are basically rectangles of length duration and height cpu.
- The end time (start + duration) is exclusive meaning that a job

stops consuming cpus just before its end time.
cpu and maxCpus are non-negative numbers (they can be 0).
startTime might be negative but duration is a non-negative number (it can be 0).

Naive solution

- Time complexity: O(N*T)
- Space complexity: O(1)

```
boolean checkSchedule(int maxCpu, Job[] jobs) {
  // Compute time horizon.
  int minStartTime = Integer.MAX_VALUE;
  int maxEndTime = Integer.MIN_VALUE;
  for (Job job: jobs) {
    minStartTime = Math.min(minStartTime, job.startTime);
    maxEndTime = Math.max(maxEndTime, job.startTime +
duration);
  }
  // Check schedule.
  for (int t = minStartTime; t < maxEndTime; t++) {
    if (!checkScheduleAtTime(maxCpu, jobs, t)) {
      return false;
    }
  }
  return true;
}

boolean checkScheduleAtTime(int maxCpu, Job[] jobs, int time) {
  int currentCpu = 0;
  for (Job job: jobs) {
    if (job.startTime <= t && t < job.startTime + duration) {
      currentCpu += job.cpu;
      if (currentCpu > maxCpu) {
        return false;
      }
    }
  }
```

```
  return true;
}
```

Many candidates simply iterate over each time point, computing the quantity of CPU used at that time point. This solution has the advantage of being optimal in terms of space complexity but can be terrible if the time horizon is large (e.g. millisecond granularity on a full day).

OK solution

- Time complexity: O(N*N)
- Space complexity: O(1)

```
boolean checkSchedule(int maxCpu, Job[] jobs) {
  for (Job job: jobs) {
    if (!checkScheduleAtTime(maxCpu, jobs, job.startTime)) {
      return false;
    }
  }
  return true;
}

boolean checkScheduleAtTime(int maxCpu, Job[] jobs, int time) {
  int currentCpu = 0;
  for (Job job: jobs) {
    if (job.startTime <= t && t < job.startTime + duration) {
      currentCpu += job.cpu;
      if (currentCpu > maxCpu) {
        return false;
      }
    }
  }
  return true;
}
```

This solution is an improvement of the previous one. It is based on the observation that a "CPU overload" can only happen at the start time of a job. There are thus only N time points to check.

Optimal solution

- Time complexity: O(N log N)
- Space complexity: Theta(N)

```
class Event {
  public int time;
  public int cpuChange;
  public Event(int time, int cpuChange) {
    this.time = time;
    this.cpuChange = cpuChange;
  }
}

class EventByTime implements Comparator<Event> {
  public int compare(Event a, Event b) {
    int tmp = a.time - b.time;
    return tmp != 0 ? tmp : a.cpuChange - b.cpuChange;
  }
}

boolean checkSchedule(int maxCpu, Job[] jobs) {
  Event[] events = new Event[jobs.length * 2];
  for (int i = 0; i < jobs.length; i++) {
    events[i*2] = new Event(jobs[i].startTime, jobs[i].cpu);
    events[i*2 + 1] = new Event(jobs[i].startTime + duration, -
jobs[i].cpu);
  }

  Arrays.sort(events, new EventByTime());

  currentCpu = 0;
  for (Event event: events) {
    currentCpu += event.cpuChange;
```

```
    if (currentCpu > maxCpu) {
      return false;
    }
  }
  return true;
}
```

Note that this solution has a best case time complexity of Omega(N) if the events are already sorted.

Follow-up questions: - How would you implement this if the array of jobs doesn't fit in memory?
- Ask about testing and corner cases.
- Ask about the inplace solution if the candidate didn't find the solution during the interview.
