CRACKING
the
CODING INTERVIEW

6TH EDITION

DISCLAIMER:

ReadMe website is intended for academic and demonstration purposes only. We’re only showing a preview of the book to respect the author’s copyright. Thank you for your understanding!

- Group 4: The Classified

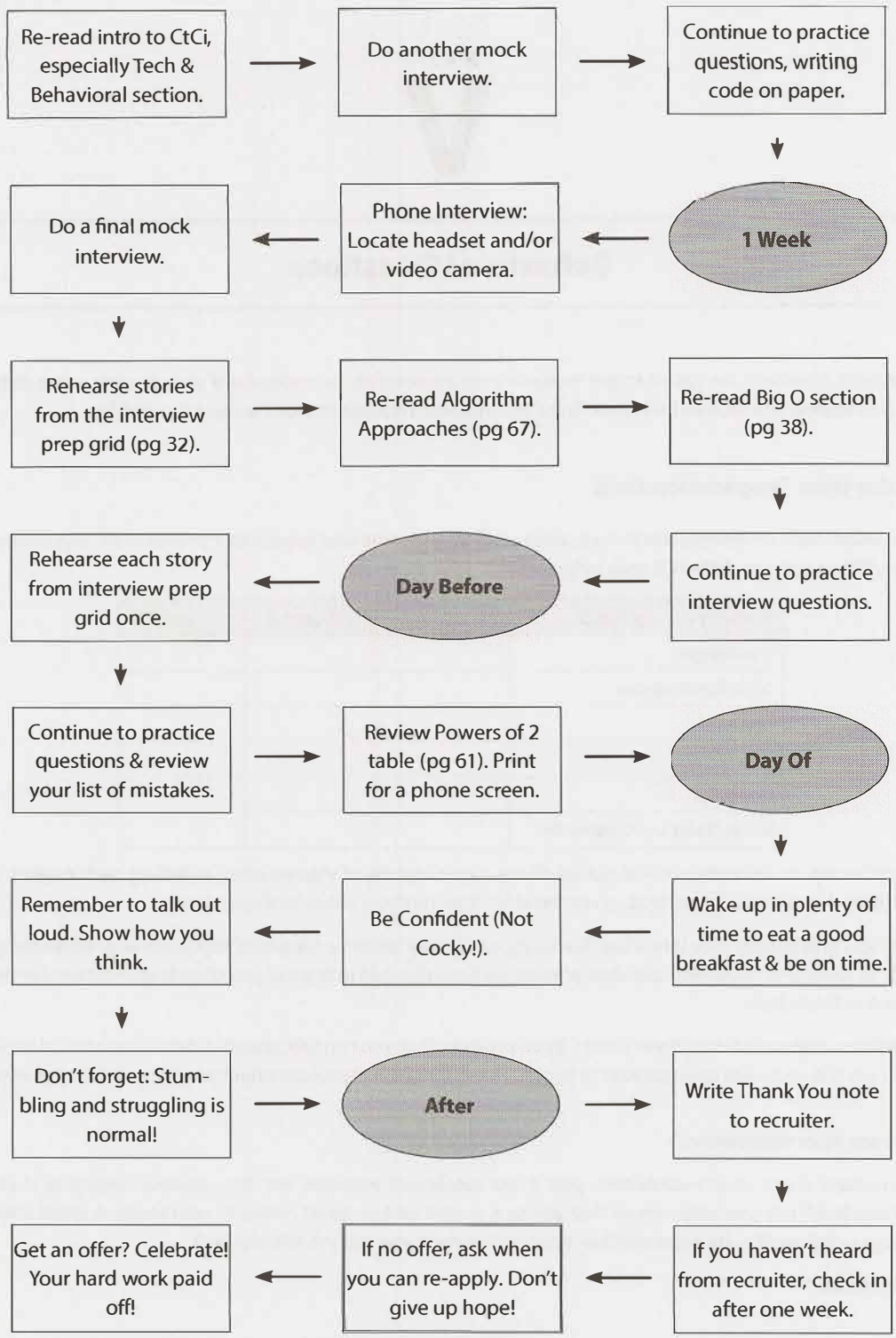
Also by Gayle Laakmann McDowell

Cracking the PM Interview

How to Land a Product Manager Job in Technology

Cracking the Tech Career

Insider Advice on Landing a Job at Google, Microsoft, Apple, or Any Top Tech Company


Re-read intro to CtCi, especially Tech & Behavioral section.
Do another mock interview.
Continue to practice questions, writing code on paper.
1 Week
Phone Interview: Locate headset and/or video camera.
Do a final mock interview.
Rehearse stories from the interview prep grid (pg 32).
Re-read Algorithm Approaches (pg 67).
Re-read Big O section (pg 38).
Continue to practice interview questions.
Day Before
Rehearse each story from interview prep grid once.
Continue to practice questions & review your list of mistakes.
Review Powers of 2 table (pg 61). Print for a phone screen.
Day Of
Wake up in plenty of time to eat a good breakfast & be on time.
Be Confident (Not Cocky!).
Remember to talk out loud. Show how you think.
Don't forget: Stumbling and struggling is normal!
After
Write Thank You note to recruiter.
If you haven't heard from recruiter, check in after one week.
If no offer, ask when you can re-apply. Don't give up hope!
Get an offer? Celebrate! Your hard work paid off!

Example: Given a smaller string s and a bigger string b, design an algorithm to find all permutations of the shorter string within the longer one. Print the location of each permutation.

Think for a moment about how you'd solve this problem. Note permutations are rearrangements of the string, so the characters in s can appear in any order in b. They must be contiguous though (not split by other characters).

If you're like most candidates, you probably thought of something like: Generate all permutations of s and then look for each in b. Since there are S! permutations, this will take O(S! * B) time, where S is the length of s and B is the length of b.

This works, but it's an extraordinarily slow algorithm. It's actually *worse* than an exponential algorithm. If s has 14 characters, that's over 87 billion permutations. Add one more character into s and we have 15 times more permutations. Ouch!

Approached a different way, you could develop a decent algorithm fairly easily. Give yourself a big example, like this one:

```
s: abbc
b: cbabadcbbabbcbabaabccbabc
```

Where are the permutations of s within b? Don't worry about how you're doing it. Just find them. Even a 12 year old could do this!

(No, really, go find them. I'll wait!)

I've underlined below each permutation.

```
s: abbc
b: cbabadcbbabbcbabaabccbabc
     ____    ____      ____
       ____  ____      ____
             ____
```

Did you find these? How?

Few people—even those who earlier came up with the O(S! * B) algorithm—actually generate all the permutations of abbc to locate those permutations in b. Almost everyone takes one of two (very similar) approaches:

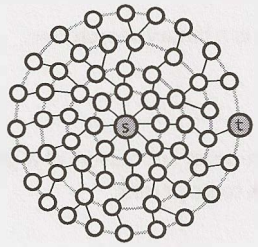
1. Walk through b and look at sliding windows of 4 characters (since s has length 4). Check if each window is a permutation of s.
2. Walk through b. Every time you see a character in s, check if the next four (the length of s) characters are a permutation of s.

Depending on the exact implementation of the "is this a permutation" part, you'll probably get a runtime of either O(B * S), O(B * S log S), or $O(B * S^2)$. None of these are the most optimal algorithm (there is an O(B) algorithm), but it's a lot better than what we had before.

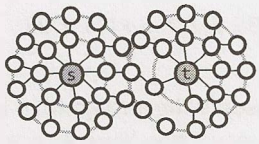
Try this approach when you're solving questions. Use a nice, big example and intuitively—manually, that is—solve it for the specific example. Then, afterwards, think hard about how you solved it. Reverse engineer your own approach.

Be particularly aware of any "optimizations" you intuitively or automatically made. For example, when you were doing this problem, you might have just skipped right over the sliding window with "d" in it, since "d" isn't in abbc. That's an optimization your brain made, and it's something you should at least be aware of in your algorithm.

Breadth-First Search
Single search from s to t that collides after four levels.



Bidirectional Search
Two searches (one from s and one from t) that collide after four levels total (two levels each).



To see why this is faster, consider a graph where every node has at most k adjacent nodes and the shortest path from node s to node t has length d.

- In traditional breadth-first search, we would search up to k nodes in the first "level" of the search. In the second level, we would search up to k nodes for each of those first k nodes, so k^2 nodes total (thus far). We would do this d times, so that's $O(k^d)$ nodes.
- In bidirectional search, we have two searches that collide after approximately $d/2$ levels (the midpoint of the path). The search from s visits approximately $k^{d/2}$, as does the search from t. That's approximately $2\,k^{d/2}$, or $O(k^{d/2})$, nodes total.

This might seem like a minor difference, but it's not. It's huge. Recall that $(k^{d/2})*(k^{d/2}) = k^d$. The bidirectional search is actually faster by a factor of $k^{d/2}$.

Put another way: if our system could only support searching "friend of friend" paths in breadth-first search, it could now likely support "friend of friend of friend of friend" paths. We can support paths that are twice as long.

Additional Reading: Topological Sort (pg 632), Dijkstra's Algorithm (pg 633), AVL Trees (pg 637), Red-Black Trees (pg 639).

Interview Questions

4.1 **Route Between Nodes:** Given a directed graph, design an algorithm to find out whether there is a route between two nodes.

Hints: #127

pg 241

4.2 **Minimal Tree:** Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height.

Hints: #19, #73, #116

pg 242

4.3 **List of Depths:** Given a binary tree, design an algorithm which creates a linked list of all the nodes at each depth (e.g., if you have a tree with depth D, you'll have D linked lists).

Hints: #107, #123, #135

pg 243

The space complexity of merge sort is $O(n)$ due to the auxiliary space used to merge parts of the array.

Quick Sort | Runtime: $O(n \log(n))$ **average,** $O(n^2)$ **worst case. Memory:** $O(\log(n))$.

In quick sort, we pick a random element and partition the array, such that all numbers that are less than the partitioning element come before all elements that are greater than it. The partitioning can be performed efficiently through a series of swaps (see below).

If we repeatedly partition the array (and its sub-arrays) around an element, the array will eventually become sorted. However, as the partitioned element is not guaranteed to be the median (or anywhere near the median), our sorting could be very slow. This is the reason for the $O(n^2)$ worst case runtime.

```
1  void quickSort(int[] arr, int left, int right) {
2     int index = partition(arr, left, right);
3     if (left < index - 1) { // Sort left half
4        quickSort(arr, left, index - 1);
5     }
6     if (index < right) { // Sort right half
7        quickSort(arr, index, right);
8     }
9  }
10
11 int partition(int[] arr, int left, int right) {
12    int pivot = arr[(left + right) / 2]; // Pick pivot point
13    while (left <= right) {
14       // Find element on left that should be on right
15       while (arr[left] < pivot) left++;
16
17       // Find element on right that should be on left
18       while (arr[right] > pivot) right--;
19
20       // Swap elements, and move left and right indices
21       if (left <= right) {
22          swap(arr, left, right); // swaps elements
23          left++;
24          right--;
25       }
26    }
27    return left;
28 }
```

Radix Sort | Runtime: $O(kn)$ **(see below)**

Radix sort is a sorting algorithm for integers (and some other data types) that takes advantage of the fact that integers have a finite number of bits. In radix sort, we iterate through each digit of the number, grouping numbers by each digit. For example, if we have an array of integers, we might first sort by the first digit, so that the 0s are grouped together. Then, we sort each of these groupings by the next digit. We repeat this process sorting by each subsequent digit, until finally the whole array is sorted.

Unlike comparison sorting algorithms, which cannot perform better than $O(n \log(n))$ in the average case, radix sort has a runtime of $O(kn)$, where n is the number of elements and k is the number of passes of the sorting algorithm.

17.7 Baby Names: Each year, the government releases a list of the 10000 most common baby names and their frequencies (the number of babies with that name). The only problem with this is that some names have multiple spellings. For example, "John" and "Jon" are essentially the same name but would be listed separately in the list. Given two lists, one of names/frequencies and the other of pairs of equivalent names, write an algorithm to print a new list of the true frequency of each name. Note that if John and Jon are synonyms, and Jon and Johnny are synonyms, then John and Johnny are synonyms. (It is both transitive and symmetric.) In the final list, any name can be used as the "real" name.

EXAMPLE

Input:

Names: John (15), Jon (12), Chris (13), Kris (4), Christopher (19)

Synonyms: (Jon, John), (John, Johnny), (Chris, Kris), (Chris, Christopher)

Output: John (27), Kris (36)

Hints: #478, #493, #512, #537, #586, #605, #655, #675, #704

pg 541

17.8 Circus Tower: A circus is designing a tower routine consisting of people standing atop one another's shoulders. For practical and aesthetic reasons, each person must be both shorter and lighter than the person below him or her. Given the heights and weights of each person in the circus, write a method to compute the largest possible number of people in such a tower.

EXAMPLE

Input (ht, wt): `(65, 100) (70, 150) (56, 90) (75, 190) (60, 95) (68, 110)`

Output: The longest tower is length 6 and includes from top to bottom:

`(56, 90) (60,95) (65,100) (68,110) (70,150) (75,190)`

Hints: #638, #657, #666, #682, #699

pg 546

17.9 Kth Multiple: Design an algorithm to find the kth number such that the only prime factors are 3, 5, and 7. Note that 3, 5, and 7 do not have to be factors, but it should not have any other prime factors. For example, the first several multiples would be (in order) 1, 3, 5, 7, 9, 15, 21.

Hints: #488, #508, #550, #591, #622, #660, #686

pg 549

17.10 Majority Element: A majority element is an element that makes up more than half of the items in an array. Given a positive integers array, find the majority element. If there is no majority element, return -1. Do this in `O(N)` time and `O(1)` space.

EXAMPLE

Input: `1 2 5 9 5 9 5 5 5`

Output: 5

Hints: #522, #566, #604, #620, #650

pg 554

17.11 Word Distance: You have a large text file containing words. Given any two words, find the shortest distance (in terms of number of words) between them in the file. If the operation will be repeated many times for the same file (but different pairs of words), can you optimize your solution?

Hints: #486, #501, #538, #558, #633

pg 557

Look at `weaveLists`. It has a specific job: to weave two lists together and return a list of all possible weaves. The existence of `allSequences` is irrelevant. Focus on the task that `weaveLists` has to do and design this algorithm.

As you're implementing `allSequences` (whether you do this before or after `weaveLists`), trust that `weaveLists` will do the right thing. Don't concern yourself with the particulars of how `weaveLists` operates while implementing something that is essentially independent. Focus on what you're doing while you're doing it.

In fact, this is good advice in general when you're confused during whiteboard coding. Have a good understanding of what a particular function should do ("okay, this function is going to return a list of ___"). You should verify that it's really doing what you think. But when you're not dealing with that function, focus on the one you are dealing with and trust that the others do the right thing. It's often too much to keep the implementations of multiple algorithms straight in your head.

4.10 Check Subtree: T1 and T2 are two very large binary trees, with T1 much bigger than T2. Create an algorithm to determine if T2 is a subtree of T1.

A tree *T2* is a subtree of *T1* if there exists a node *n* in *T1* such that the subtree of *n* is identical to *T2*. That is, if you cut off the tree at node *n*, the two trees would be identical.

pg 111

SOLUTION

In problems like this, it's useful to attempt to solve the problem assuming that there is just a small amount of data. This will give us a basic idea of an approach that might work.

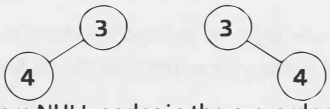
The Simple Approach

In this smaller, simpler problem, we could consider comparing string representations of traversals of each tree. If T2 is a subtree of T1, then T2's traversal should be a substring of T1. Is the reverse true? If so, should we use an in-order traversal or a pre-order traversal?

An in-order traversal will definitely not work. After all, consider a scenario in which we were using binary search trees. A binary search tree's in-order traversal always prints out the values in sorted order. Therefore, two binary search trees with the same values will always have the same in-order traversals, even if their structure is different.

What about a pre-order traversal? This is a bit more promising. At least in this case we know certain things, like the first element in the pre-order traversal is the root node. The left and right elements will follow.

Unfortunately, trees with different structures could still have the same pre-order traversal.



There's a simple fix though. We can store NULL nodes in the pre-order traversal string as a special character, like an 'X'. (We'll assume that the binary trees contain only integers.) The left tree would have the traversal `{3, 4, X}` and the right tree will have the traversal `{3, X, 4}`.

Observe that, as long as we represent the NULL nodes, the pre-order traversal of a tree is unique. That is, if two trees have the same pre-order traversal, then we know they are identical trees in values and structure.

Notice what each test strip really means. It's a binary indicator for poisoned or unpoisoned. Is it possible to map 1000 keys to 10 binary values such that each key is mapped to a unique configuration of values? Yes, of course. This is what a binary number is.

We can take each bottle number and look at its binary representation. If there's a 1 in the `i`th digit, then we will add a drop of this bottle's contents to test strip `i`. Observe that 2^{10} is 1024, so 10 test strips will be enough to handle up to 1024 bottles.

We wait seven days, and then read the results. If test strip `i` is positive, then set bit `i` of the result value. Reading all the test strips will give us the ID of the poisoned bottle.

```
1  int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
2    runTests(bottles, strips);
3    ArrayList<Integer> positive = getPositiveOnDay(strips, 7);
4    return setBits(positive);
5  }
6
7  /* Add bottle contents to test strips */
8  void runTests(ArrayList<Bottle> bottles, ArrayList<TestStrip> testStrips) {
9    for (Bottle bottle : bottles) {
10     int id = bottle.getId();
11     int bitIndex = 0;
12     while (id > 0) {
13       if ((id & 1) == 1) {
14         testStrips.get(bitIndex).addDropOnDay(0, bottle);
15       }
16       bitIndex++;
17       id >>= 1;
18     }
19   }
20 }
21
22 /* Get test strips that are positive on a particular day. */
23 ArrayList<Integer> getPositiveOnDay(ArrayList<TestStrip> testStrips, int day) {
24   ArrayList<Integer> positive = new ArrayList<Integer>();
25   for (TestStrip testStrip : testStrips) {
26     int id = testStrip.getId();
27     if (testStrip.isPositiveOnDay(day)) {
28       positive.add(id);
29     }
30   }
31   return positive;
32 }
33
34 /* Create number by setting bits with indices specified in positive. */
35 int setBits(ArrayList<Integer> positive) {
36   int id = 0;
37   for (Integer bitIndex : positive) {
38     id |= 1 << bitIndex;
39   }
40   return id;
41 }
```

This approach will work as long as $2^T >= B$, where T is the number of test strips and B is the number of bottles.

A simple implementation of this code is below.

```
1  int countWays(int n) {
2    if (n < 0) {
3      return 0;
4    } else if (n == 0) {
5      return 1;
6    } else {
7      return countWays(n-1) + countWays(n-2) + countWays(n-3);
8    }
9  }
```

Like the Fibonacci problem, the runtime of this algorithm is exponential (roughly $O(3^n)$), since each call branches out to three more calls.

Memoization Solution

The previous solution for `countWays` is called many times for the same values, which is unnecessary. We can fix this through memoization.

Essentially, if we've seen this value of n before, return the cached value. Each time we compute a fresh value, add it to the cache.

Typically we use a `HashMap<Integer, Integer>` for a cache. In this case, the keys will be exactly 1 through n. It's more compact to use an integer array.

```
1  int countWays(int n) {
2    int[] memo = new int[n + 1];
3    Arrays.fill(memo, -1);
4    return countWays(n, memo);
5  }
6
7  int countWays(int n, int[] memo) {
8    if (n < 0) {
9      return 0;
10   } else if (n == 0) {
11     return 1;
12   } else if (memo[n] > -1) {
13     return memo[n];
14   } else {
15     memo[n] = countWays(n - 1, memo) + countWays(n - 2, memo) +
16               countWays(n - 3, memo);
17     return memo[n];
18   }
19 }
```

Regardless of whether or not you use memoization, note that the number of ways will quickly overflow the bounds of an integer. By the time you get to just `n = 37`, the result has already overflowed. Using a `long` will delay, but not completely solve, this issue.

It is great to communicate this issue to your interviewer. He probably won't ask you to work around it (although you could, with a `BigInteger` class), but it's nice to demonstrate that you think about these issues.

- A hash table allows efficient lookups of data, but it wouldn't ordinarily allow easy data purging.

How can we get the best of both worlds? By merging the two data structures. Here's how this works:

Just as before, we create a linked list where a node is moved to the front every time it's accessed. This way, the end of the linked list will always contain the stalest information.

In addition, we have a hash table that maps from a query to the corresponding node in the linked list. This allows us to not only efficiently return the cached results, but also to move the appropriate node to the front of the list, thereby updating its "freshness."

For illustrative purposes, abbreviated code for the cache is below. The code attachment provides the full code for this part. Note that in your interview, it is unlikely that you would be asked to write the full code for this as well as perform the design for the larger system.

```
1   public class Cache {
2     public static int MAX_SIZE = 10;
3     public Node head, tail;
4     public HashMap<String, Node> map;
5     public int size = 0;
6
7     public Cache() {
8       map = new HashMap<String, Node>();
9     }
10
11    /* Moves node to front of linked list */
12    public void moveToFront(Node node) { ... }
13    public void moveToFront(String query) { ... }
14
15    /* Removes node from linked list */
16    public void removeFromLinkedList(Node node) { ... }
17
18    /* Gets results from cache, and updates linked list */
19    public String[] getResults(String query) {
20      if (!map.containsKey(query)) return null;
21
22      Node node = map.get(query);
23      moveToFront(node); // update freshness
24      return node.results;
25    }
26
27    /* Inserts results into linked list and hash */
28    public void insertResults(String query, String[] results) {
29      if (map.containsKey(query)) { // update values
30        Node node = map.get(query);
31        node.results = results;
32        moveToFront(node); // update freshness
33        return;
34      }
35
36      Node node = new Node(query, results);
37      moveToFront(node);
38      map.put(query, node);
39
40      if (size > MAX_SIZE) {
41        map.remove(tail.query);
42        removeFromLinkedList(tail);
43      }
```

You'll need to make sure that the pen holds up under these conditions.

Remember that in any testing question, you need to test both the intended and unintended scenarios. People don't always use the product the way you want them to.

11.6 Test an ATM: How would you test an ATM in a distributed banking system?

pg 157

SOLUTION

The first thing to do on this question is to clarify assumptions. Ask the following questions:

- Who is going to use the ATM? Answers might be "anyone," or it might be "blind people," or any number of other answers.
- What are they going to use it for? Answers might be "withdrawing money," "transferring money," "checking their balance," or many other answers.
- What tools do we have to test? Do we have access to the code, or just to the ATM?

Remember: a good tester makes sure she knows what she's testing!

Once we understand what the system looks like, we'll want to break down the problem into different testable components. These components include:

- Logging in
- Withdrawing money
- Depositing money
- Checking balance
- Transferring money

We would probably want to use a mix of manual and automated testing.

Manual testing would involve going through the steps above, making sure to check for all the error cases (low balance, new account, nonexistent account, and so on).

Automated testing is a bit more complex. We'll want to automate all the standard scenarios, as shown above, and we also want to look for some very specific issues, such as race conditions. Ideally, we would be able to set up a closed system with fake accounts and ensure that, even if someone withdraws and deposits money rapidly from different locations, he never gets money or loses money that he shouldn't.

Above all, we need to prioritize security and reliability. People's accounts must always be protected, and we must make sure that money is always properly accounted for. No one wants to unexpectedly lose money! A good tester understands the system priorities.

	FizzBuzz	Fizz	Buzz	Number
current % 3 == 0	true	true	false	false
current % 5 == 0	true	false	true	false
to print	FizzBuzz	Fizz	Buzz	current

For the most part, this can be handled by taking in "target" parameters and the value to print. The output for the `Number` thread needs to be overwritten, though, as it's not a simple, fixed string.

We can implement a `FizzBuzzThread` class which handles most of this. A `NumberThread` class can extend `FizzBuzzThread` and override the `print` method.

```
1  Thread[] threads = {new FizzBuzzThread(true, true, n, "FizzBuzz"),
2                      new FizzBuzzThread(true, false, n, "Fizz"),
3                      new FizzBuzzThread(false, true, n, "Buzz"),
4                      new NumberThread(false, false, n)};
5  for (Thread thread : threads) {
6    thread.start();
7  }
8
9  public class FizzBuzzThread extends Thread {
10   private static Object lock = new Object();
11   protected static int current = 1;
12   private int max;
13   private boolean div3, div5;
14   private String toPrint;
15
16   public FizzBuzzThread(boolean div3, boolean div5, int max, String toPrint) {
17     this.div3 = div3;
18     this.div5 = div5;
19     this.max = max;
20     this.toPrint = toPrint;
21   }
22
23   public void print() {
24     System.out.println(toPrint);
25   }
26
27   public void run() {
28     while (true) {
29       synchronized (lock) {
30         if (current > max) {
31           return;
32         }
33
34         if ((current % 3 == 0) == div3 &&
35             (current % 5 == 0) == div5) {
36           print();
37           current++;
38         }
39       }
40     }
41   }
42 }
43
44 public class NumberThread extends FizzBuzzThread {
```

Finally, we look at 3. Adding 3 to sum (4) gives us 7, so we update maxSum. The max subsequence is therefore the sequence {6, -2, 3}.

When we look at this in the fully expanded array, our logic is identical. The code below implements this algorithm.

```
1   int getMaxSum(int[] a) {
2     int maxsum = 0;
3     int sum = 0;
4     for (int i = 0; i < a.length; i++) {
5       sum += a[i];
6       if (maxsum < sum) {
7         maxsum = sum;
8       } else if (sum < 0) {
9         sum = 0;
10      }
11    }
12    return maxsum;
13  }
```

If the array is all negative numbers, what is the correct behavior? Consider this simple array: {-3, -10, -5}. You could make a good argument that the maximum sum is either:

1. -3 (if you assume the subsequence can't be empty)
2. 0 (the subsequence has length 0)
3. MINIMUM_INT (essentially, the error case).

We went with option #2 (maxSum = 0), but there's no "correct" answer. This is a great thing to discuss with your interviewer; it will show how detail-oriented you are.

16.18 Pattern Matching: You are given two strings, pattern and value. The pattern string consists of just the letters a and b, describing a pattern within a string. For example, the string catcatgocatgo matches the pattern aabab (where cat is a and go is b). It also matches patterns like a, ab, and b. Write a method to determine if value matches pattern.

pg 183

SOLUTION

As always, we can start with a simple brute force approach.

Brute Force

A brute force algorithm is to just try all possible values for a and b and then check if this works.

We could do this by iterating through all substrings for a and all possible substrings for b. There are $O(n^2)$ substrings in a string of length n, so this will actually take $O(n^4)$ time. But then, for each value of a and b, we need to build the new string of this length and compare it for equality. This building/comparison step takes $O(n)$ time, giving an overall runtime of $O(n^5)$.

```
1   for each possible substring a
2     for each possible substring b
3       candidate = buildFromPattern(pattern, a, b)
4       if candidate equals value
5         return true
```

Ouch.

```
/* Compute the difference between the number of letters and numbers between the
 * beginning of the array and each index. */
int[] computeDeltaArray(char[] array) {
  int[] deltas = new int[array.length];
  int delta = 0;
  for (int i = 0; i < array.length; i++) {
    if (Character.isLetter(array[i])) {
      delta++;
    } else if (Character.isDigit(array[i])) {
      delta--;
    }
    deltas[i] = delta;
  }
  return deltas;
}

/* Find the matching pair of values in the deltas array with the largest
 * difference in indices. */
int[] findLongestMatch(int[] deltas) {
  HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
  map.put(0,  -1);
  int[] max = new int[2];
  for (int i = 0; i < deltas.length; i++) {
    if (!map.containsKey(deltas[i])) {
      map.put(deltas[i],  i);
    } else {
      int match = map.get(deltas[i]);
      int distance = i - match;
      int longest = max[1] - max[0];
      if (distance > longest) {
        max[1] = i;
        max[0] = match;
      }
    }
  }
  return max;
}

char[] extract(char[] array, int start, int end) { /* same */ }
```

This solution takes O(N) time, where N is size of the array.

17.6 Count of 2s: Write a method to count the number of 2s between 0 and n.

pg 186

SOLUTION

Our first approach to this problem can be—and probably should be—a brute force solution. Remember that interviewers want to see how you're approaching a problem. Offering a brute force solution is a great way to start.

```
/* Counts the number of '2' digits between 0 and n */
int numberOf2sInRange(int n) {
  int count = 0;
  for (int i = 2; i <= n; i++) { // Might as well start at 2
    count += numberOf2s(i);
```

- » take `best(4) = 90`.

The first gives us 135 minutes, `best(3) = 135`.

- `best(2)`: What's the best option for $\{r_2 = 60, \ldots\}$? We can either:
  - » take $r_2 = 60$ and merge it with `best(4) = 90`, or:
  - » take `best(3) = 135`.

  The first gives us 150 minutes, `best(2) = 150`.
- `best(1)`: What's the best option for $\{r_1 = 15, \ldots\}$? We can either:
  - » take $r_1 = 15$ and merge it with `best(3) = 135`, or:
  - » take `best(2) = 150`.

  Either way, `best(1) = 150`.
- `best(0)`: What's the best option for $\{r_0 = 30, \ldots\}$? We can either:
  - » take $r_0 = 30$ and merge it with `best(2) = 150`, or:
  - » take `best(1) = 150`.

  The first gives us 180 minutes, `best(0) = 180`.

Therefore, we return 180 minutes.

The code below implements this algorithm.

```
1  int maxMinutes(int[] massages) {
2    /* Allocating two extra slots in the array so we don't have to do bounds
3     * checking on lines 7 and 8. */
4    int[] memo = new int[massages.length + 2];
5    memo[massages.length] = 0;
6    memo[massages.length + 1] = 0;
7    for (int i = massages.length - 1; i >= 0; i--) {
8      int bestWith = massages[i] + memo[i + 2];
9      int bestWithout = memo[i + 1];
10     memo[i] = Math.max(bestWith, bestWithout);
11   }
12   return memo[0];
13 }
```

The runtime of this solution is `O(n)` and the space complexity is also `O(n)`.

It's nice in some ways that it's iterative, but we haven't actually "won" anything here. The recursive solution had the same time and space complexity.

Solution #4: Iterative with Optimal Time and Space

In reviewing the last solution, we can recognize that we only use the values in the memo table for a short amount of time. Once we are several elements past an index, we never use that element's index again.

In fact, at any given index `i`, we only need to know the best value from `i + 1` and `i + 2`. Therefore, we can get rid of the memo table and just use two integers.

```
1  int maxMinutes(int[] massages) {
2    int oneAway = 0;
3    int twoAway = 0;
4    for (int i = massages.length - 1; i >= 0; i--) {
5      int bestWith = massages[i] + twoAway;
6      int bestWithout = oneAway;
```

Well, if we're going to create a rectangle of words, we know that each row must be the same length and each column must be the same length. So let's group the words of the dictionary based on their sizes. Let's call this grouping D, where `D[i]` contains the list of words of length `i`.

Next, observe that we're looking for the largest rectangle. What is the largest rectangle that could be formed? It's `length(largest word)`^{2}.

```
1  int maxRectangle = longestWord * longestWord;
2  for z = maxRectangle to 1 {
3    for each pair of numbers (i, j) where i*j = z {
4      /* attempt to make rectangle. return if successful. */
5    }
6  }
```

By iterating from the biggest possible rectangle to the smallest, we ensure that the first valid rectangle we find will be the largest possible one.

Now, for the hard part: `makeRectangle(int l, int h)`. This method attempts to build a rectangle of words which has length `l` and height h.

One way to do this is to iterate through all (ordered) sets of h words and then check if the columns are also valid words. This will work, but it's rather inefficient.

Imagine that we are trying to build a 6x5 rectangle and the first few rows are:

```
there
queen
pizza
.....
```

At this point, we know that the first column starts with `tqp`. We know—or *should* know—that no dictionary word starts with `tqp`. Why do we bother continuing to build a rectangle when we know we'll fail to create a valid one in the end?

This leads us to a more optimal solution. We can build a trie to easily look up if a substring is a prefix of a word in the dictionary. Then, when we build our rectangle, row by row, we check to see if the columns are all valid prefixes. If not, we fail immediately, rather than continue to try to build this rectangle.

The code below implements this algorithm. It is long and complex, so we will go through it step by step.

First, we do some pre-processing to group words by their lengths. We create an array of tries (one for each word length), but hold off on building the tries until we need them.

```
1  WordGroup[] groupList = WordGroup.createWordGroups(list);
2  int maxWordLength = groupList.length;
3  Trie trieList[] = new Trie[maxWordLength];
```

The `maxRectangle` method is the "main" part of our code. It starts with the biggest possible rectangle area (which is `maxWordLength`^{2}) and tries to build a rectangle of that size. If it fails, it subtracts one from the area and attempts this new, smaller size. The first rectangle that can be successfully built is guaranteed to be the biggest.

```
1  Rectangle maxRectangle() {
2    int maxSize = maxWordLength * maxWordLength;
3    for (int z = maxSize; z > 0; z--) { // start from biggest area
4      for (int i = 1; i <= maxWordLength; i ++ ) {
5        if (z % i == 0) {
6          int j = z / i;
7          if (j <= maxWordLength) {
8            /* Create rectangle of length i and height j. Note that i * j = z. */
9            Rectangle rectangle = makeRectangle(i, j);
```

#31. 4.10 Although the problem seems like it stems from duplicate values, it's really deeper than that. The issue is that the pre-order traversal is the same only because there are null nodes that we skipped over (because they're null). Consider inserting a placeholder value into the pre-order traversal string whenever you reach a null node. Register the null node as a "real" node so that you can distinguish between the different structures.

#32. 3.5 Imagine your secondary stack is sorted. Can you insert elements into it in sorted order? You might need some extra storage. What could you use for extra storage?

#33. 4.4 If you've developed a brute force solution, be careful about its runtime. If you are computing the height of the subtrees for each node, you could have a pretty inefficient algorithm.

#34. 1.9 If a string is a rotation of another, then it's a rotation at a particular point. For example, a rotation of `waterbottle` at character 3 means cutting `waterbottle` at character 3 and putting the right half (`erbottle`) before the left half (`wat`).

#35. 4.5 If you traversed the tree using an in-order traversal and the elements were truly in the right order, does this indicate that the tree is actually in order? What happens for duplicate elements? If duplicate elements are allowed, they must be on a specific side (usually the left).

#36. 4.8 Start with the root. Can you identify if root is the first common ancestor? If it is not, can you identify which side of root the first common ancestor is on?

#37. 4.10 Alternatively, we can handle this problem recursively. Given a specific node within T1, can we check to see if its subtree matches T2?

#38. 3.1 If you want to allow for flexible divisions, you can shift stacks around. Can you ensure that all available capacity is used?

#39. 4.9 What is the very first value that must be in each array?

#40. 2.1 Without extra space, you'll need $O(N^2)$ time. Try using two pointers, where the second one searches ahead of the first one.

#41. 2.2 Try implementing it recursively. If you could find the (K-1)th to last element, can you find the Kth element?

#42. 4.11 Be very careful in this problem to ensure that each node is equally likely and that your solution doesn't slow down the speed of standard binary search tree algorithms (like `insert`, `find`, and `delete`). Also, remember that even if you assume that it's a balanced binary search tree, this doesn't mean that the tree is full/complete/perfect.

#43. 3.5 Keep the secondary stack in sorted order, with the biggest elements on the top. Use the primary stack for additional storage.

#44. 1.1 Try a hash table.

#45. 2.7 Examples will help you. Draw a picture of intersecting linked lists and two equivalent linked lists (by value) that do not intersect.

#46. 4.8 Try a recursive approach. Check if p and q are descendants of the left subtree and the right subtree. If they are descendants of different subtrees, then the current node is the first common ancestor. If they are descendants of the same subtree, then that subtree holds the first common ancestor. Now, how do you implement this efficiently?