X Solutions

*Join us at **www.CrackingTheCodingInterview.com** to download the complete solutions, contribute or view solutions in other programming languages, discuss problems from this book with other readers, ask questions, report issues, view this book's errata, and seek additional advice.*

1

Solutions to Arrays and Strings

1.1 Is Unique: Implement an algorithm to determine if a string has all unique characters. What if you cannot use additional data structures?

pg 90

SOLUTION

You should first ask your interviewer if the string is an ASCII string or a Unicode string. Asking this question will show an eye for detail and a solid foundation in computer science. We'll assume for simplicity the character set is ASCII. If this assumption is not valid, we would need to increase the storage size.

One solution is to create an array of boolean values, where the flag at index `i` indicates whether character `i` in the alphabet is contained in the string. The second time you see this character you can immediately return `false`.

We can also immediately return `false` if the string length exceeds the number of unique characters in the alphabet. After all, you can't form a string of 280 unique characters out of a 128-character alphabet.

> It's also okay to assume 256 characters. This would be the case in extended ASCII. You should clarify your assumptions with your interviewer.

The code below implements this algorithm.

```
boolean isUniqueChars(String str) {
   if (str.length() > 128) return false;

   boolean[] char_set = new boolean[128];
   for (int i = 0; i < str.length(); i++) {
      int val = str.charAt(i);
      if (char_set[val]) { // Already found this char in string
         return false;
      }
      char_set[val] = true;
   }
   return true;
}
```

The time complexity for this code is `O(n)`, where n is the length of the string. The space complexity is `O(1)`. (You could also argue the time complexity is `O(1)`, since the for loop will never iterate through more than 128 characters.) If you didn't want to assume the character set is fixed, you could express the complexity as `O(c)` space and `O(min(c, n))` or `O(c)` time, where c is the size of the character set.

We can reduce our space usage by a factor of eight by using a bit vector. We will assume, in the below code, that the string only uses the lowercase letters a through z. This will allow us to use just a single `int`.

```
boolean isUniqueChars(String str) {
  int checker = 0;
  for (int i = 0; i < str.length(); i++) {
    int val = str.charAt(i) - 'a';
    if ((checker & (1 << val)) > 0) {
      return false;
    }
    checker |= (1 << val);
  }
  return true;
}
```

If we can't use additional data structures, we can do the following:

1. Compare every character of the string to every other character of the string. This will take $O(n^2)$ time and $O(1)$ space.
2. If we are allowed to modify the input string, we could sort the string in $O(n \log(n))$ time and then linearly check the string for neighboring characters that are identical. Careful, though: many sorting algorithms take up extra space.

These solutions are not as optimal in some respects, but might be better depending on the constraints of the problem.

1.2 **Check Permutation:** Given two strings, write a method to decide if one is a permutation of the other.

pg 90

SOLUTION

Like in many questions, we should confirm some details with our interviewer. We should understand if the permutation comparison is case sensitive. That is: is God a permutation of dog? Additionally, we should ask if whitespace is significant. We will assume for this problem that the comparison is case sensitive and whitespace is significant. So, "god   " is different from "dog".

Observe first that strings of different lengths cannot be permutations of each other. There are two easy ways to solve this problem, both of which use this optimization.

Solution #1: Sort the strings.

If two strings are permutations, then we know they have the same characters, but in different orders. Therefore, sorting the strings will put the characters from two permutations in the same order. We just need to compare the sorted versions of the strings.

```
String sort(String s) {
  char[] content = s.toCharArray();
  java.util.Arrays.sort(content);
  return new String(content);
}

boolean permutation(String s, String t) {
  if (s.length() != t.length()) {
    return false;
  }
```

```
    return sort(s).equals(sort(t));
}
```

Though this algorithm is not as optimal in some senses, it may be preferable in one sense: It's clean, simple and easy to understand. In a practical sense, this may very well be a superior way to implement the problem.

However, if efficiency is very important, we can implement it a different way.

Solution #2: Check if the two strings have identical character counts.

We can also use the definition of a permutation—two words with the same character counts—to implement this algorithm. We simply iterate through this code, counting how many times each character appears. Then, afterwards, we compare the two arrays.

```
1  boolean permutation(String s, String t) {
2     if (s.length() != t.length()) {
3        return false;
4     }
5
6     int[] letters = new int[128]; // Assumption
7
8     char[] s_array = s.toCharArray();
9     for (char c : s_array) { // count number of each char in s.
10       letters[c]++;
11    }
12
13    for (int i = 0; i < t.length(); i++) {
14       int c = (int) t.charAt(i);
15       letters[c]--;
16       if (letters[c] < 0) {
17          return false;
18       }
19    }
20
21    return true;
22 }
```

Note the assumption on line 6. In your interview, you should always check with your interviewer about the size of the character set. We assumed that the character set was ASCII.

1.3 **URLify:** Write a method to replace all spaces in a string with '%20'. You may assume that the string has sufficient space at the end to hold the additional characters, and that you are given the "true" length of the string. (Note: if implementing in Java, please use a character array so that you can perform this operation in place.)

EXAMPLE

Input: `"Mr John Smith    ", 13`

Output: `"Mr%20John%20Smith"`

pg 90

SOLUTION

A common approach in string manipulation problems is to edit the string starting from the end and working backwards. This is useful because we have an extra buffer at the end, which allows us to change characters without worrying about what we're overwriting.

We will use this approach in this problem. The algorithm employs a two-scan approach. In the first scan, we count the number of spaces. By tripling this number, we can compute how many extra characters we will have in the final string. In the second pass, which is done in reverse order, we actually edit the string. When we see a space, we replace it with `%20`. If there is no space, then we copy the original character.

The code below implements this algorithm.

```
void replaceSpaces(char[] str, int trueLength) {
  int spaceCount = 0, index, i = 0;
  for (i = 0; i < trueLength; i++) {
    if (str[i] == ' ') {
      spaceCount++;
    }
  }
  index = trueLength + spaceCount * 2;
  if (trueLength < str.length) str[trueLength] = '\0'; // End array
  for (i = trueLength - 1; i >= 0; i--) {
    if (str[i] == ' ') {
      str[index - 1] = '0';
      str[index - 2] = '2';
      str[index - 3] = '%';
      index = index - 3;
    } else {
      str[index - 1] = str[i];
      index--;
    }
  }
}
```

We have implemented this problem using character arrays, because Java strings are immutable. If we used strings directly, the function would have to return a new copy of the string, but it would allow us to implement this in just one pass.

1.4 Palindrome Permutation: Given a string, write a function to check if it is a permutation of a palindrome. A palindrome is a word or phrase that is the same forwards and backwards. A permutation is a rearrangement of letters. The palindrome does not need to be limited to just dictionary words.

EXAMPLE

Input: `Tact Coa`

Output: `True` (permutations: "`taco cat`", "`atco cta`", etc.)

pg 91

SOLUTION

This is a question where it helps to figure out what it means for a string to be a permutation of a palindrome. This is like asking what the "defining features" of such a string would be.

A palindrome is a string that is the same forwards and backwards. Therefore, to decide if a string is a permutation of a palindrome, we need to know if it can be written such that it's the same forwards and backwards.

What does it take to be able to write a set of characters the same way forwards and backwards? We need to have an even number of almost all characters, so that half can be on one side and half can be on the other side. At most one character (the middle character) can have an odd count.

For example, we know `tactcoapapa` is a permutation of a palindrome because it has two Ts, four As, two

Cs, two Ps, and one O. That O would be the center of all possible palindromes.

> To be more precise, strings with even length (after removing all non-letter characters) must have all even counts of characters. Strings of an odd length must have exactly one character with an odd count. Of course, an "even" string can't have an odd number of exactly one character, otherwise it wouldn't be an even-length string (an odd number + many even numbers = an odd number). Likewise, a string with odd length can't have all characters with even counts (sum of evens is even). It's therefore sufficient to say that, to be a permutation of a palindrome, a string can have no more than one character that is odd. This will cover both the odd and the even cases.

This leads us to our first algorithm.

Solution #1

Implementing this algorithm is fairly straightforward. We use a hash table to count how many times each character appears. Then, we iterate through the hash table and ensure that no more than one character has an odd count.

```
boolean isPermutationOfPalindrome(String phrase) {
  int[] table = buildCharFrequencyTable(phrase);
  return checkMaxOneOdd(table);
}

/* Check that no more than one character has an odd count. */
boolean checkMaxOneOdd(int[] table) {
  boolean foundOdd = false;
  for (int count : table) {
    if (count % 2 == 1) {
      if (foundOdd) {
        return false;
      }
      foundOdd = true;
    }
  }
  return true;
}

/* Map each character to a number. a -> 0, b -> 1, c -> 2, etc.
 * This is case insensitive. Non-letter characters map to -1. */
int getCharNumber(Character c) {
  int a = Character.getNumericValue('a');
  int z = Character.getNumericValue('z');
  int val = Character.getNumericValue(c);
  if (a <= val && val <= z) {
    return val - a;
  }
  return -1;
}

/* Count how many times each character appears. */
int[] buildCharFrequencyTable(String phrase) {
  int[] table = new int[Character.getNumericValue('z') -
                        Character.getNumericValue('a') + 1];
  for (char c : phrase.toCharArray()) {
    int x = getCharNumber(c);
```

```
      if (x != -1) {
         table[x]++;
      }
   }
   return table;
}
```

This algorithm takes `O(N)` time, where N is the length of the string.

Solution #2

We can't optimize the big O time here since any algorithm will always have to look through the entire string. However, we can make some smaller incremental improvements. Because this is a relatively simple problem, it can be worthwhile to discuss some small optimizations or at least some tweaks.

Instead of checking the number of odd counts at the end, we can check as we go along. Then, as soon as we get to the end, we have our answer.

```
boolean isPermutationOfPalindrome(String phrase) {
   int countOdd = 0;
   int[] table = new int[Character.getNumericValue('z') -
                         Character.getNumericValue('a') + 1];
   for (char c : phrase.toCharArray()) {
      int x = getCharNumber(c);
      if (x != -1) {
         table[x]++;
         if (table[x] % 2 == 1) {
            countOdd++;
         } else {
            countOdd--;
         }
      }
   }
   return countOdd <= 1;
}
```

It's important to be very clear here that this is not necessarily more optimal. It has the same big O time and might even be slightly slower. We have eliminated a final iteration through the hash table, but now we have to run a few extra lines of code for each character in the string.

You should discuss this with your interviewer as an alternate, but not necessarily more optimal, solution.

Solution #3

If you think more deeply about this problem, you might notice that we don't actually need to know the counts. We just need to know if the count is even or odd. Think about flipping a light on/off (that is initially off). If the light winds up in the `off` state, we don't know how many times we flipped it, but we do know it was an even count.

Given this, we can use a single integer (as a bit vector). When we see a letter, we map it to an integer between 0 and 26 (assuming an English alphabet). Then we toggle the bit at that value. At the end of the iteration, we check that at most one bit in the integer is set to 1.

We can easily check that no bits in the integer are 1: just compare the integer to 0. There is actually a very elegant way to check that an integer has exactly one bit set to 1.

Picture an integer like `00010000`. We could of course shift the integer repeatedly to check that there's only a single 1. Alternatively, if we subtract 1 from the number, we'll get `00001111`. What's notable about this is that there is no overlap between the numbers (as opposed to say `00101000`, which, when we subtract 1 from, we get `00100111`.) So, we can check to see that a number has exactly one 1 because if we subtract 1 from it and then AND it with the new number, we should get `0`.

```
00010000 - 1 = 00001111
00010000 & 00001111 = 0
```

This leads us to our final implementation.

```
boolean isPermutationOfPalindrome(String phrase) {
  int bitVector = createBitVector(phrase);
  return bitVector == 0 || checkExactlyOneBitSet(bitVector);
}

/* Create a bit vector for the string. For each letter with value i, toggle the
 * ith bit. */
int createBitVector(String phrase) {
  int bitVector = 0;
  for (char c : phrase.toCharArray()) {
    int x = getCharNumber(c);
    bitVector = toggle(bitVector, x);
  }
  return bitVector;
}

/* Toggle the ith bit in the integer. */
int toggle(int bitVector, int index) {
  if (index < 0) return bitVector;

  int mask = 1 << index;
  if ((bitVector & mask) == 0) {
    bitVector |= mask;
  } else {
    bitVector &= ~mask;
  }
  return bitVector;
}

/* Check that exactly one bit is set by subtracting one from the integer and
 * ANDing it with the original integer. */
boolean checkExactlyOneBitSet(int bitVector) {
  return (bitVector & (bitVector - 1)) == 0;
}
```

Like the other solutions, this is `O(N)`.

It's interesting to note a solution that we did not explore. We avoided solutions along the lines of "create all possible permutations and check if they are palindromes." While such a solution would work, it's entirely infeasible in the real world. Generating all permutations requires factorial time (which is actually worse than exponential time), and it is essentially infeasible to perform on strings longer than about 10-15 characters.

I mention this (impractical) solution because a lot of candidates hear a problem like this and say, "In order to check if A is in group B, I must know everything that is in B and then check if one of the items equals A." That's not always the case, and this problem is a simple demonstration of it. You don't need to generate all permutations in order to check if one is a palindrome.

1.5 **One Away:** There are three types of edits that can be performed on strings: insert a character, remove a character, or replace a character. Given two strings, write a function to check if they are one edit (or zero edits) away.

EXAMPLE

```
pale,  ple  -> true
pales, pale -> true
pale,  bale -> true
pale,  bae  -> false
```

pg 91

SOLUTION

There is a "brute force" algorithm to do this. We could check all possible strings that are one edit away by testing the removal of each character (and comparing), testing the replacement of each character (and comparing), and then testing the insertion of each possible character (and comparing).

That would be too slow, so let's not bother with implementing it.

This is one of those problems where it's helpful to think about the "meaning" of each of these operations. What does it mean for two strings to be one insertion, replacement, or removal away from each other?

- **Replacement:** Consider two strings, such as `bale` and `pale`, that are one replacement away. Yes, that does mean that you could replace a character in `bale` to make `pale`. But more precisely, it means that they are different only in one place.
- **Insertion:** The strings `apple` and `aple` are one insertion away. This means that if you compared the strings, they would be identical—except for a shift at some point in the strings.
- **Removal:** The strings `apple` and `aple` are also one removal away, since removal is just the inverse of insertion.

We can go ahead and implement this algorithm now. We'll merge the insertion and removal check into one step, and check the replacement step separately.

Observe that you don't need to check the strings for insertion, removal, and replacement edits. The lengths of the strings will indicate which of these you need to check.

```java
boolean oneEditAway(String first, String second) {
  if (first.length() == second.length()) {
    return oneEditReplace(first, second);
  } else if (first.length() + 1 == second.length()) {
    return oneEditInsert(first, second);
  } else if (first.length() - 1 == second.length()) {
    return oneEditInsert(second, first);
  }
  return false;
}

boolean oneEditReplace(String s1, String s2) {
  boolean foundDifference = false;
  for (int i = 0; i < s1.length(); i++) {
    if (s1.charAt(i) != s2.charAt(i)) {
      if (foundDifference) {
        return false;
      }

```

```
        foundDifference = true;
      }
    }
    return true;
}

/* Check if you can insert a character into s1 to make s2. */
boolean oneEditInsert(String s1, String s2) {
    int index1 = 0;
    int index2 = 0;
    while (index2 < s2.length() && index1 < s1.length()) {
      if (s1.charAt(index1) != s2.charAt(index2)) {
        if (index1 != index2) {
          return false;
        }
        index2++;
      } else {
        index1++;
        index2++;
      }
    }
    return true;
}
```

This algorithm (and almost any reasonable algorithm) takes O(n) time, where n is the length of the shorter string.

> Why is the runtime dictated by the shorter string instead of the longer string? If the strings are the same length (plus or minus one character), then it doesn't matter whether we use the longer string or the shorter string to define the runtime. If the strings are very different lengths, then the algorithm will terminate in O(1) time. One really, really long string therefore won't significantly extend the runtime. It increases the runtime only if both strings are long.

We might notice that the code for `oneEditReplace` is very similar to that for `oneEditInsert`. We can merge them into one method.

To do this, observe that both methods follow similar logic: compare each character and ensure that the strings are only different by one. The methods vary in how they handle that difference. The method `oneEditReplace` does nothing other than flag the difference, whereas `oneEditInsert` increments the pointer to the longer string. We can handle both of these in the same method.

```
boolean oneEditAway(String first, String second) {
    /* Length checks. */
    if (Math.abs(first.length() - second.length()) > 1) {
      return false;
    }

    /* Get shorter and longer string.*/
    String s1 = first.length() < second.length() ? first : second;
    String s2 = first.length() < second.length() ? second : first;

    int index1 = 0;
    int index2 = 0;
    boolean foundDifference = false;
    while (index2 < s2.length() && index1 < s1.length()) {
      if (s1.charAt(index1) != s2.charAt(index2)) {
```

```
         /* Ensure that this is the first difference found.*/
         if (foundDifference) return false;
         foundDifference = true;

         if (s1.length() == s2.length()) { // On replace, move shorter pointer
            index1++;
         }
      } else {
         index1++; // If matching, move shorter pointer
      }
      index2++; // Always move pointer for longer string
   }
   return true;
}
```

Some people might argue the first approach is better, as it is clearer and easier to follow. Others, however, will argue that the second approach is better, since it's more compact and doesn't duplicate code (which can facilitate maintainability).

You don't necessarily need to "pick a side." You can discuss the tradeoffs with your interviewer.

1.6 **String Compression:** Implement a method to perform basic string compression using the counts of repeated characters. For example, the string `aabcccccaaa` would become `a2b1c5a3`. If the "compressed" string would not become smaller than the original string, your method should return the original string. You can assume the string has only uppercase and lowercase letters (a - z).

pg 91

SOLUTION

At first glance, implementing this method seems fairly straightforward, but perhaps a bit tedious. We iterate through the string, copying characters to a new string and counting the repeats. At each iteration, check if the current character is the same as the next character. If not, add its compressed version to the result.

How hard could it be?

```
String compressBad(String str) {
   String compressedString = "";
   int countConsecutive = 0;
   for (int i = 0; i < str.length(); i++) {
      countConsecutive++;

      /* If next character is different than current, append this char to result.*/
      if (i + 1 >= str.length() || str.charAt(i) != str.charAt(i + 1)) {
         compressedString += "" + str.charAt(i) + countConsecutive;
         countConsecutive = 0;
      }
   }
   return compressedString.length() < str.length() ? compressedString : str;
}
```

This works. Is it efficient, though? Take a look at the runtime of this code.

The runtime is $O(p + k^2)$, where p is the size of the original string and k is the number of character sequences. For example, if the string is `aabccdeeaa`, then there are six character sequences. It's slow because string concatenation operates in $O(n^2)$ time (see `StringBuilder` on pg 89).

We can fix this by using a `StringBuilder`.

```java
String compress(String str) {
  StringBuilder compressed = new StringBuilder();
  int countConsecutive = 0;
  for (int i = 0; i < str.length(); i++) {
    countConsecutive++;

    /* If next character is different than current, append this char to result.*/
    if (i + 1 >= str.length() || str.charAt(i) != str.charAt(i + 1)) {
      compressed.append(str.charAt(i));
      compressed.append(countConsecutive);
      countConsecutive = 0;
    }
  }
  return compressed.length() < str.length() ? compressed.toString() : str;
}
```

Both of these solutions create the compressed string first and then return the shorter of the input string and the compressed string.

Instead, we can check in advance. This will be more optimal in cases where we don't have a large number of repeating characters. It will avoid us having to create a string that we never use. The downside of this is that it causes a second loop through the characters and also adds nearly duplicated code.

```java
String compress(String str) {
  /* Check final length and return input string if it would be longer. */
  int finalLength = countCompression(str);
  if (finalLength >= str.length()) return str;

  StringBuilder compressed = new StringBuilder(finalLength); // initial capacity
  int countConsecutive = 0;
  for (int i = 0; i < str.length(); i++) {
    countConsecutive++;

    /* If next character is different than current, append this char to result.*/
    if (i + 1 >= str.length() || str.charAt(i) != str.charAt(i + 1)) {
      compressed.append(str.charAt(i));
      compressed.append(countConsecutive);
      countConsecutive = 0;
    }
  }
  return compressed.toString();
}

int countCompression(String str) {
  int compressedLength = 0;
  int countConsecutive = 0;
  for (int i = 0; i < str.length(); i++) {
    countConsecutive++;

    /* If next character is different than current, increase the length.*/
    if (i + 1 >= str.length() || str.charAt(i) != str.charAt(i + 1)) {
      compressedLength += 1 + String.valueOf(countConsecutive).length();
      countConsecutive = 0;
    }
  }
  return compressedLength;
}
```

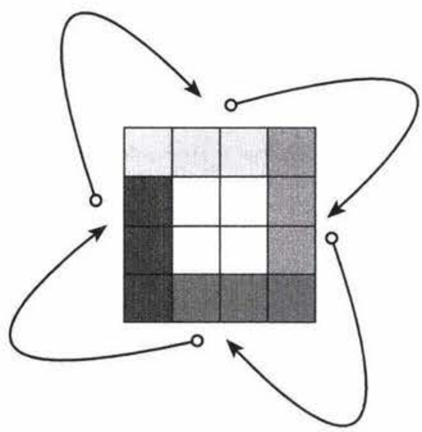
One other benefit of this approach is that we can initialize `StringBuilder` to its necessary capacity up-front. Without this, `StringBuilder` will (behind the scenes) need to double its capacity every time it hits capacity. The capacity could be double what we ultimately need.

1.7 **Rotate Matrix:** Given an image represented by an NxN matrix, where each pixel in the image is 4 bytes, write a method to rotate the image by 90 degrees. Can you do this in place?

pg 91

SOLUTION

Because we're rotating the matrix by 90 degrees, the easiest way to do this is to implement the rotation in layers. We perform a circular rotation on each layer, moving the top edge to the right edge, the right edge to the bottom edge, the bottom edge to the left edge, and the left edge to the top edge.



How do we perform this four-way edge swap? One option is to copy the top edge to an array, and then move the left to the top, the bottom to the left, and so on. This requires `O(N)` memory, which is actually unnecessary.

A better way to do this is to implement the swap index by index. In this case, we do the following:

```
for i = 0 to n
   temp = top[i];
   top[i] = left[i]
   left[i] = bottom[i]
   bottom[i] = right[i]
   right[i] = temp
```

We perform such a swap on each layer, starting from the outermost layer and working our way inwards. (Alternatively, we could start from the inner layer and work outwards.)

The code for this algorithm is below.

```
boolean rotate(int[][] matrix) {
   if (matrix.length == 0 || matrix.length != matrix[0].length) return false;
   int n = matrix.length;
   for (int layer = 0; layer < n / 2; layer++) {
      int first = layer;
      int last = n - 1 - layer;
      for(int i = first; i < last; i++) {
         int offset = i - first;
```

```
            int top = matrix[first][i]; // save top

            // left -> top
            matrix[first][i] = matrix[last-offset][first];

            // bottom -> left
            matrix[last-offset][first] = matrix[last][last - offset];

            // right -> bottom
            matrix[last][last - offset] = matrix[i][last];

            // top -> right
            matrix[i][last] = top; // right <- saved top
        }
    }
    return true;
}
```

This algorithm is $O(N^2)$, which is the best we can do since any algorithm must touch all N^2 elements.

1.8 Zero Matrix: Write an algorithm such that if an element in an MxN matrix is 0, its entire row and column are set to 0.

pg 91

SOLUTION

At first glance, this problem seems easy: just iterate through the matrix and every time we see a cell with value zero, set its row and column to 0. There's one problem with that solution though: when we come across other cells in that row or column, we'll see the zeros and change their row and column to zero. Pretty soon, our entire matrix will be set to zeros.

One way around this is to keep a second matrix which flags the zero locations. We would then do a second pass through the matrix to set the zeros. This would take O(MN) space.

Do we really need O(MN) space? No. Since we're going to set the entire row and column to zero, we don't need to track that it was exactly `cell[2][4]` (row 2, column 4). We only need to know that row 2 has a zero somewhere, and column 4 has a zero somewhere. We'll set the entire row and column to zero anyway, so why would we care to keep track of the exact location of the zero?

The code below implements this algorithm. We use two arrays to keep track of all the rows with zeros and all the columns with zeros. We then nullify rows and columns based on the values in these arrays.

```
void setZeros(int[][] matrix) {
   boolean[] row = new boolean[matrix.length];
   boolean[] column = new boolean[matrix[0].length];

   // Store the row and column index with value 0
   for (int i = 0; i < matrix.length; i++) {
      for (int j = 0; j < matrix[0].length;j++) {
         if (matrix[i][j] == 0) {
            row[i] = true;
            column[j] = true;
         }
      }
   }

```

```
    // Nullify rows
    for (int i = 0; i < row.length; i++) {
      if (row[i]) nullifyRow(matrix, i);
    }

    // Nullify columns
    for (int j = 0; j < column.length; j++) {
      if (column[j]) nullifyColumn(matrix, j);
    }
  }

  void nullifyRow(int[][] matrix, int row) {
    for (int j = 0; j < matrix[0].length; j++) {
      matrix[row][j] = 0;
    }
  }

  void nullifyColumn(int[][] matrix, int col) {
    for (int i = 0; i < matrix.length; i++) {
      matrix[i][col] = 0;
    }
  }
```

To make this somewhat more space efficient, we could use a bit vector instead of a boolean array. It would still be `O(N)` space.

We can reduce the space to `O(1)` by using the first row as a replacement for the `row` array and the first column as a replacement for the `column` array. This works as follows:

1. Check if the first row and first column have any zeros, and set variables `rowHasZero` and `columnHasZero`. (We'll nullify the first row and first column later, if necessary.)
2. Iterate through the rest of the matrix, setting `matrix[i][0]` and `matrix[0][j]` to zero whenever there's a zero in `matrix[i][j]`.
3. Iterate through rest of matrix, nullifying row `i` if there's a zero in `matrix[i][0]`.
4. Iterate through rest of matrix, nullifying column `j` if there's a zero in `matrix[0][j]`.
5. Nullify the first row and first column, if necessary (based on values from Step 1).

This code is below:

```
void setZeros(int[][] matrix) {
  boolean rowHasZero = false;
  boolean colHasZero = false;

  // Check if first row has a zero
  for (int j = 0; j < matrix[0].length; j++) {
    if (matrix[0][j] == 0) {
      rowHasZero = true;
      break;
    }
  }

  // Check if first column has a zero
  for (int i = 0; i < matrix.length; i++) {
    if (matrix[i][0] == 0) {
      colHasZero = true;
      break;
```

```
      }
   }

   // Check for zeros in the rest of the array
   for (int i = 1; i < matrix.length; i++) {
      for (int j = 1; j < matrix[0].length;j++) {
         if (matrix[i][j] == 0) {
            matrix[i][0] = 0;
            matrix[0][j] = 0;
         }
      }
   }

   // Nullify rows based on values in first column
   for (int i = 1; i < matrix.length; i++) {
      if (matrix[i][0] == 0) {
         nullifyRow(matrix, i);
      }
   }

   // Nullify columns based on values in first row
   for (int j = 1; j < matrix[0].length; j++) {
      if (matrix[0][j] == 0) {
         nullifyColumn(matrix, j);
      }
   }

   // Nullify first row
   if (rowHasZero) {
      nullifyRow(matrix, 0);
   }

   // Nullify first column
   if (colHasZero) {
      nullifyColumn(matrix, 0);
   }
}
```

This code has a lot of "do this for the rows, then the equivalent action for the column." In an interview, you could abbreviate this code by adding comments and TODOs that explain that the next chunk of code looks the same as the earlier code, but using rows. This would allow you to focus on the most important parts of the algorithm.

1.9 **String Rotation:** Assume you have a method `isSubstring` which checks if one word is a substring of another. Given two strings, `s1` and `s2`, write code to check if `s2` is a rotation of `s1` using only one call to `isSubstring` (e.g., "`waterbottle`" is a rotation of "`erbottlewat`").

pg 91

SOLUTION

If we imagine that `s2` is a rotation of `s1`, then we can ask what the rotation point is. For example, if you rotate `waterbottle` after `wat`, you get `erbottlewat`. In a rotation, we cut `s1` into two parts, `x` and `y`, and rearrange them to get `s2`.

```
s1 = xy = waterbottle
x = wat
```

```
y = erbottle
s2 = yx = erbottlewat
```

So, we need to check if there's a way to split `s1` into x and y such that xy = `s1` and yx = `s2`. Regardless of where the division between x and y is, we can see that yx will always be a substring of `xyxy`. That is, `s2` will always be a substring of `s1s1`.

And this is precisely how we solve the problem: simply do `isSubstring(s1s1, s2)`.

The code below implements this algorithm.

```
boolean isRotation(String s1, String s2) {
  int len = s1.length();
  /* Check that s1 and s2 are equal length and not empty */
  if (len == s2.length() && len > 0) {
    /* Concatenate s1 and s1 within new buffer */
    String s1s1 = s1 + s1;
    return isSubstring(s1s1, s2);
  }
  return false;
}
```

The runtime of this varies based on the runtime of `isSubstring`. But if you assume that `isSubstring` runs in `O(A+B)` time (on strings of length A and B), then the runtime of `isRotation` is `O(N)`.

2

Solutions to Linked Lists

2.1 Remove Dups: Write code to remove duplicates from an unsorted linked list.

FOLLOW UP

How would you solve this problem if a temporary buffer is not allowed?

pg 94

SOLUTION

In order to remove duplicates from a linked list, we need to be able to track duplicates. A simple hash table will work well here.

In the below solution, we simply iterate through the linked list, adding each element to a hash table. When we discover a duplicate element, we remove the element and continue iterating. We can do this all in one pass since we are using a linked list.

```
void deleteDups(LinkedListNode n) {
  HashSet<Integer> set = new HashSet<Integer>();
  LinkedListNode previous = null;
  while (n != null) {
    if (set.contains(n.data)) {
      previous.next = n.next;
    } else {
      set.add(n.data);
      previous = n;
    }
    n = n.next;
  }
}
```

The above solution takes `O(N)` time, where N is the number of elements in the linked list.

Follow Up: No Buffer Allowed

If we don't have a buffer, we can iterate with two pointers: `current` which iterates through the linked list, and `runner` which checks all subsequent nodes for duplicates.

```
void deleteDups(LinkedListNode head) {
  LinkedListNode current = head;
  while (current != null) {
    /* Remove all future nodes that have the same value */
    LinkedListNode runner = current;
    while (runner.next != null) {
      if (runner.next.data == current.data) {
```

```
            runner.next = runner.next.next;
         } else {
            runner = runner.next;
         }
      }
      current = current.next;
   }
}
```

This code runs in O(1) space, but $O(N^2)$ time.

2.2 Return Kth to Last: Implement an algorithm to find the kth to last element of a singly linked list.

pg 223

SOLUTION

We will approach this problem both recursively and non-recursively. Remember that recursive solutions are often cleaner but less optimal. For example, in this problem, the recursive implementation is about half the length of the iterative solution but also takes O(n) space, where n is the number of elements in the linked list.

Note that for this solution, we have defined k such that passing in k = 1 would return the last element, k = 2 would return to the second to last element, and so on. It is equally acceptable to define k such that k = 0 would return the last element.

Solution #1: If linked list size is known

If the size of the linked list is known, then the kth to last element is the `(length - k)`th element. We can just iterate through the linked list to find this element. Because this solution is so trivial, we can almost be sure that this is not what the interviewer intended.

Solution #2: Recursive

This algorithm recurses through the linked list. When it hits the end, the method passes back a counter set to 0. Each parent call adds 1 to this counter. When the counter equals k, we know we have reached the kth to last element of the linked list.

Implementing this is short and sweet—provided we have a way of "passing back" an integer value through the stack. Unfortunately, we can't pass back a node and a counter using normal return statements. So how do we handle this?

Approach A: Don't Return the Element.

One way to do this is to change the problem to simply printing the kth to last element. Then, we can pass back the value of the counter simply through return values.

```
int printKthToLast(LinkedListNode head, int k) {
   if (head == null) {
      return 0;
   }
   int index = printKthToLast(head.next, k) + 1;
   if (index == k) {
      System.out.println(k + "th to last node is " + head.data);
   }
   return index;
}
```

Of course, this is only a valid solution if the interviewer says it is valid.

Approach B: Use C++.

A second way to solve this is to use C++ and to pass values by reference. This allows us to return the node value, but also update the counter by passing a pointer to it.

```
node* nthToLast(node* head, int k, int& i) {
  if (head == NULL) {
    return NULL;
  }
  node* nd = nthToLast(head->next, k, i);
  i = i + 1;
  if (i == k) {
    return head;
  }
  return nd;
}

node* nthToLast(node* head, int k) {
  int i = 0;
  return nthToLast(head, k, i);
}
```

Approach C: Create a Wrapper Class.

We described earlier that the issue was that we couldn't simultaneously return a counter and an index. If we wrap the `counter` value with simple class (or even a single element array), we can mimic passing by reference.

```
class Index {
  public int value = 0;
}

LinkedListNode kthToLast(LinkedListNode head, int k) {
  Index idx = new Index();
  return kthToLast(head, k, idx);
}

LinkedListNode kthToLast(LinkedListNode head, int k, Index idx) {
  if (head == null) {
    return null;
  }
  LinkedListNode node = kthToLast(head.next, k, idx);
  idx.value = idx.value + 1;
  if (idx.value == k) {
    return head;
  }
  return node;
}
```

Each of these recursive solutions takes `O(n)` space due to the recursive calls.

There are a number of other solutions that we haven't addressed. We could store the counter in a static variable. Or, we could create a class that stores both the node and the counter, and return an instance of that class. Regardless of which solution we pick, we need a way to update both the node and the counter in a way that all levels of the recursive stack will see.

Solution #3: Iterative

A more optimal, but less straightforward, solution is to implement this iteratively. We can use two pointers, p1 and p2. We place them k nodes apart in the linked list by putting p2 at the beginning and moving p1 k nodes into the list. Then, when we move them at the same pace, p1 will hit the end of the linked list after `LENGTH - k` steps. At that point, p2 will be `LENGTH - k` nodes into the list, or k nodes from the end.

The code below implements this algorithm.

```
LinkedListNode nthToLast(LinkedListNode head, int k) {
  LinkedListNode p1 = head;
  LinkedListNode p2 = head;

  /* Move p1 k nodes into the list.*/
  for (int i = 0; i < k; i++) {
    if (p1 == null) return null; // Out of bounds
    p1 = p1.next;
  }

  /* Move them at the same pace. When p1 hits the end, p2 will be at the right
   * element. */
  while (p1 != null) {
    p1 = p1.next;
    p2 = p2.next;
  }
  return p2;
}
```

This algorithm takes `O(n)` time and `O(1)` space.

2.3 **Delete Middle Node:** Implement an algorithm to delete a node in the middle (i.e., any node but the first and last node, not necessarily the exact middle) of a singly linked list, given only access to that node.

EXAMPLE

Input: the node c from the linked list `a->b->c->d->e->f`

Result: nothing is returned, but the new linked list looks like `a->b->d->e->f`

pg 94

SOLUTION

In this problem, you are not given access to the head of the linked list. You only have access to that node. The solution is simply to copy the data from the next node over to the current node, and then to delete the next node.

The code below implements this algorithm.

```
boolean deleteNode(LinkedListNode n) {
  if (n == null || n.next == null) {
    return false; // Failure
  }
  LinkedListNode next = n.next;
  n.data = next.data;
  n.next = next.next;
  return true;
}
```

Note that this problem cannot be solved if the node to be deleted is the last node in the linked list. That's okay—your interviewer wants you to point that out, and to discuss how to handle this case. You could, for example, consider marking the node as dummy.

2.4 **Partition:** Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x. If x is contained within the list, the values of x only need to be after the elements less than x (see below). The partition element x can appear anywhere in the "right partition"; it does not need to appear between the left and right partitions.

EXAMPLE

Input: `3 -> 5 -> 8 -> 5 -> 10 -> 2 -> 1` [partition = 5]

Output: `3 -> 1 -> 2 -> 10 -> 5 -> 5 -> 8`

pg 94

SOLUTION

If this were an array, we would need to be careful about how we shifted elements. Array shifts are very expensive.

However, in a linked list, the situation is much easier. Rather than shifting and swapping elements, we can actually create two different linked lists: one for elements less than x, and one for elements greater than or equal to x.

We iterate through the linked list, inserting elements into our `before` list or our `after` list. Once we reach the end of the linked list and have completed this splitting, we merge the two lists.

This approach is mostly "stable" in that elements stay in their original order, other than the necessary movement around the partition. The code below implements this approach.

```
/* Pass in the head of the linked list and the value to partition around */
LinkedListNode partition(LinkedListNode node, int x) {
   LinkedListNode beforeStart = null;
   LinkedListNode beforeEnd = null;
   LinkedListNode afterStart = null;
   LinkedListNode afterEnd = null;

   /* Partition list */
   while (node != null) {
      LinkedListNode next = node.next;
      node.next = null;
      if (node.data < x) {
         /* Insert node into end of before list */
         if (beforeStart == null) {
            beforeStart = node;
            beforeEnd = beforeStart;
         } else {
            beforeEnd.next = node;
            beforeEnd = node;
         }
      } else {
         /* Insert node into end of after list */
         if (afterStart == null) {
            afterStart = node;
            afterEnd = afterStart;
         } else {
```

```
                afterEnd.next = node;
                afterEnd = node;
            }
        }
        node = next;
    }

    if (beforeStart == null) {
        return afterStart;
    }

    /* Merge before list and after list */
    beforeEnd.next = afterStart;
    return beforeStart;
}
```

If it bugs you to keep around four different variables for tracking two linked lists, you're not alone. We can make this code a bit shorter.

If we don't care about making the elements of the list "stable" (which there's no obligation to, since the interviewer hasn't specified that), then we can instead rearrange the elements by growing the list at the head and tail.

In this approach, we start a "new" list (using the existing nodes). Elements bigger than the pivot element are put at the tail and elements smaller are put at the head. Each time we insert an element, we update either the head or tail.

```
LinkedListNode partition(LinkedListNode node, int x) {
    LinkedListNode head = node;
    LinkedListNode tail = node;

    while (node != null) {
        LinkedListNode next = node.next;
        if (node.data < x) {
            /* Insert node at head. */
            node.next = head;
            head = node;
        } else {
            /* Insert node at tail. */
            tail.next = node;
            tail = node;
        }
        node = next;
    }
    tail.next = null;

    // The head has changed, so we need to return it to the user.
    return head;
}
```

There are many equally optimal solutions to this problem. If you came up with a different one, that's okay!

2.5 Sum Lists: You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in reverse order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.

EXAMPLE

Input: `(7-> 1 -> 6) + (5 -> 9 -> 2)`. That is, `617 + 295`.

Output: `2 -> 1 -> 9`. That is, `912`.

FOLLOW UP

Suppose the digits are stored in forward order. Repeat the above problem.

Input: `(6 -> 1 -> 7) + (2 -> 9 -> 5)`. That is, `617 + 295`.

Output: `9 -> 1 -> 2`. That is, `912`.

pg 95

SOLUTION

It's useful to remember in this problem how exactly addition works. Imagine the problem:

```
  6 1 7
+ 2 9 5
```

First, we add 7 and 5 to get 12. The digit 2 becomes the last digit of the number, and 1 gets carried over to the next step. Second, we add 1, 1, and 9 to get 11. The 1 becomes the second digit, and the other 1 gets carried over the final step. Third and finally, we add 1, 6 and 2 to get 9. So, our value becomes 912.

We can mimic this process recursively by adding node by node, carrying over any "excess" data to the next node. Let's walk through this for the below linked list:

```
    7 -> 1 -> 6
+   5 -> 9 -> 2
```

We do the following:

1. We add 7 and 5 first, getting a result of 12. 2 becomes the first node in our linked list, and we "carry" the 1 to the next sum.

   `List: 2 -> ?`

2. We then add 1 and 9, as well as the "carry," getting a result of 11. 1 becomes the second element of our linked list, and we carry the 1 to the next sum.

   `List: 2 -> 1 -> ?`

3. Finally, we add 6, 2 and our "carry," to get 9. This becomes the final element of our linked list.

   `List: 2 -> 1 -> 9.`

The code below implements this algorithm.

```
LinkedListNode addLists(LinkedListNode l1, LinkedListNode l2, int carry) {
   if (l1 == null && l2 == null && carry == 0) {
      return null;
   }

   LinkedListNode result = new LinkedListNode();
   int value = carry;
   if (l1 != null) {
      value += l1.data;
   }
   if (l2 != null) {
```

```
      value += l2.data;
   }

   result.data = value % 10; /* Second digit of number */

   /* Recurse */
   if (l1 != null || l2 != null) {
      LinkedListNode more = addLists(l1 == null ? null : l1.next,
                                     l2 == null ? null : l2.next,
                                     value >= 10 ? 1 : 0);
      result.setNext(more);
   }
   return result;
}
```

In implementing this code, we must be careful to handle the condition when one linked list is shorter than another. We don't want to get a null pointer exception.

Follow Up

Part B is conceptually the same (recurse, carry the excess), but has some additional complications when it comes to implementation:

1. One list may be shorter than the other, and we cannot handle this "on the fly." For example, suppose we were adding (1 -> 2 -> 3 -> 4) and (5 -> 6 -> 7). We need to know that the 5 should be "matched" with the 2, not the 1. We can accomplish this by comparing the lengths of the lists in the beginning and padding the shorter list with zeros.
2. In the first part, successive results were added to the tail (i.e., passed forward). This meant that the recursive call would be *passed* the carry, and would return the result (which is then appended to the tail). In this case, however, results are added to the head (i.e., passed backward). The recursive call must return the result, as before, as well as the carry. This is not terribly challenging to implement, but it is more cumbersome. We can solve this issue by creating a wrapper class called Partial Sum.

The code below implements this algorithm.

```
class PartialSum {
   public LinkedListNode sum = null;
   public int carry = 0;
}

LinkedListNode addLists(LinkedListNode l1, LinkedListNode l2) {
   int len1 = length(l1);
   int len2 = length(l2);

   /* Pad the shorter list with zeros - see note (1) */
   if (len1 < len2) {
      l1 = padList(l1, len2 - len1);
   } else {
      l2 = padList(l2, len1 - len2);
   }

   /* Add lists */
   PartialSum sum = addListsHelper(l1, l2);

   /* If there was a carry value left over, insert this at the front of the list.
    * Otherwise, just return the linked list. */
   if (sum.carry == 0) {
```

```
        return sum.sum;
    } else {
        LinkedListNode result = insertBefore(sum.sum, sum.carry);
        return result;
    }
}

PartialSum addListsHelper(LinkedListNode l1, LinkedListNode l2) {
    if (l1 == null && l2 == null) {
        PartialSum sum = new PartialSum();
        return sum;
    }
    /* Add smaller digits recursively */
    PartialSum sum = addListsHelper(l1.next, l2.next);

    /* Add carry to current data */
    int val = sum.carry + l1.data + l2.data;

    /* Insert sum of current digits */
    LinkedListNode full_result = insertBefore(sum.sum, val % 10);

    /* Return sum so far, and the carry value */
    sum.sum = full_result;
    sum.carry = val / 10;
    return sum;
}

/* Pad the list with zeros */
LinkedListNode padList(LinkedListNode l, int padding) {
    LinkedListNode head = l;
    for (int i = 0; i < padding; i++) {
        head = insertBefore(head, 0);
    }
    return head;
}

/* Helper function to insert node in the front of a linked list */
LinkedListNode insertBefore(LinkedListNode list, int data) {
    LinkedListNode node = new LinkedListNode(data);
    if (list != null) {
        node.next = list;
    }
    return node;
}
```

Note how we have pulled `insertBefore()`, `padList()`, and `length()` (not listed) into their own methods. This makes the code cleaner and easier to read—a wise thing to do in your interviews!

2.6 Palindrome: Implement a function to check if a linked list is a palindrome.

pg 95

SOLUTION

To approach this problem, we can picture a palindrome like `0 -> 1 -> 2 -> 1 -> 0`. We know that, since it's a palindrome, the list must be the same backwards and forwards. This leads us to our first solution.

Solution #1: Reverse and Compare

Our first solution is to reverse the linked list and compare the reversed list to the original list. If they're the same, the lists are identical.

Note that when we compare the linked list to the reversed list, we only actually need to compare the first half of the list. If the first half of the normal list matches the first half of the reversed list, then the second half of the normal list must match the second half of the reversed list.

```
boolean isPalindrome(LinkedListNode head) {
  LinkedListNode reversed = reverseAndClone(head);
  return isEqual(head, reversed);
}

LinkedListNode reverseAndClone(LinkedListNode node) {
  LinkedListNode head = null;
  while (node != null) {
    LinkedListNode n = new LinkedListNode(node.data); // Clone
    n.next = head;
    head = n;
    node = node.next;
  }
  return head;
}

boolean isEqual(LinkedListNode one, LinkedListNode two) {
  while (one != null && two != null) {
    if (one.data != two.data) {
      return false;
    }
    one = one.next;
    two = two.next;
  }
  return one == null && two == null;
}
```

Observe that we've modularized this code into `reverse` and `isEqual` functions. We've also created a new class so that we can return both the head and the tail of this method. We could have also returned a two-element array, but that approach is less maintainable.

Solution #2: Iterative Approach

We want to detect linked lists where the front half of the list is the reverse of the second half. How would we do that? By reversing the front half of the list. A stack can accomplish this.

We need to push the first half of the elements onto a stack. We can do this in two different ways, depending on whether or not we know the size of the linked list.

If we know the size of the linked list, we can iterate through the first half of the elements in a standard for loop, pushing each element onto a stack. We must be careful, of course, to handle the case where the length of the linked list is odd.

If we don't know the size of the linked list, we can iterate through the linked list, using the fast runner / slow runner technique described in the beginning of the chapter. At each step in the loop, we push the data from the slow runner onto a stack. When the fast runner hits the end of the list, the slow runner will have reached the middle of the linked list. By this point, the stack will have all the elements from the front of the linked list, but in reverse order.

Now, we simply iterate through the rest of the linked list. At each iteration, we compare the node to the top of the stack. If we complete the iteration without finding a difference, then the linked list is a palindrome.

```
boolean isPalindrome(LinkedListNode head) {
   LinkedListNode fast = head;
   LinkedListNode slow = head;

   Stack<Integer> stack = new Stack<Integer>();

   /* Push elements from first half of linked list onto stack. When fast runner
    * (which is moving at 2x speed) reaches the end of the linked list, then we
    * know we're at the middle */
   while (fast != null && fast.next != null) {
      stack.push(slow.data);
      slow = slow.next;
      fast = fast.next.next;
   }

   /* Has odd number of elements, so skip the middle element */
   if (fast != null) {
      slow = slow.next;
   }

   while (slow != null) {
      int top = stack.pop().intValue();

      /* If values are different, then it's not a palindrome */
      if (top != slow.data) {
         return false;
      }
      slow = slow.next;
   }
   return true;
}
```

Solution #3: Recursive Approach

First, a word on notation: in this solution, when we use the notation `node Kx`, the variable K indicates the value of the node data, and x (which is either `f` or b) indicates whether we are referring to the front node with that value or the back node. For example, in the below linked list, `node 2b` would refer to the second (back) node with value 2.

Now, like many linked list problems, you can approach this problem recursively. We may have some intuitive idea that we want to compare element `0` and element `n - 1`, element 1 and element `n-2`, element 2 and element `n-3`, and so on, until the middle element(s). For example:

```
0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0
```

In order to apply this approach, we first need to know when we've reached the middle element, as this will form our base case. We can do this by passing in `length - 2` for the `length` each time. When the length equals `0` or 1, we're at the center of the linked list. This is because the length is reduced by 2 each time. Once we've recursed $\frac{N}{2}$ times, `length` will be down to 0.

```
recurse(Node n, int length) {
   if (length == 0 || length == 1) {
      return [something]; // At middle
   }
   recurse(n.next, length - 2);
```

```
  ...
}
```

This method will form the outline of the `isPalindrome` method. The "meat" of the algorithm though is comparing node `i` to node `n - i` to check if the linked list is a palindrome. How do we do that?

Let's examine what the call stack looks like:

```
1  v1 = isPalindrome: list = 0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 7
2    v2 = isPalindrome: list = 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 5
3      v3 = isPalindrome: list = 2 ( 3 ) 2 ) 1 ) 0. length = 3
4        v4 = isPalindrome: list = 3 ) 2 ) 1 ) 0. length = 1
5        returns v3
6      returns v2
7    returns v1
8  returns ?
```

In the above call stack, each call wants to check if the list is a palindrome by comparing its head node with the corresponding node from the back of the list. That is:

- Line 1 needs to compare node `0f` with node `0b`
- Line 2 needs to compare node `1f` with node `1b`
- Line 3 needs to compare node `2f` with node `2b`
- Line 4 needs to compare node `3f` with node `3b`.

If we rewind the stack, passing nodes back as described below, we can do just that:

- Line 4 sees that it is the middle node (since `length = 1`), and passes back `head.next`. The value `head` equals node 3, so `head.next` is node 2b.
- Line 3 compares its head, node `2f`, to `returned_node` (the value from the previous recursive call), which is node 2b. If the values match, it passes a reference to node `1b` (`returned_node.next`) up to line 2.
- Line 2 compares its head (node `1f`) to `returned_node` (node 1b). If the values match, it passes a reference to node `0b` (or, `returned_node.next`) up to line 1.
- Line 1 compares its head, node `0f`, to `returned_node`, which is node `0b`. If the values match, it returns true.

To generalize, each call compares its head to `returned_node`, and then passes `returned_node.next` up the stack. In this way, every node `i` gets compared to node `n - i`. If at any point the values do not match, we return `false`, and every call up the stack checks for that value.

But wait, you might ask, sometimes we said we'll return a `boolean` value, and sometimes we're returning a node. Which is it?

It's both. We create a simple class with two members, a `boolean` and a node, and return an instance of that class.

```
class Result {
  public LinkedListNode node;
  public boolean result;
}
```

The example below illustrates the parameters and return values from this sample list.

```
isPalindrome: list = 0 ( 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 9
  isPalindrome: list = 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 7
    isPalindrome: list = 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 5
```

```
      isPalindrome: list = 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 3
        isPalindrome: list = 4 ) 3 ) 2 ) 1 ) 0. len = 1
        returns node 3b, true
      returns node 2b, true
    returns node 1b, true
  returns node 0b, true
returns null, true
```

Implementing this code is now just a matter of filling in the details.

```
boolean isPalindrome(LinkedListNode head) {
  int length = lengthOfList(head);
  Result p = isPalindromeRecurse(head, length);
  return p.result;
}

Result isPalindromeRecurse(LinkedListNode head, int length) {
  if (head == null || length <= 0) { // Even number of nodes
    return new Result(head, true);
  } else if (length == 1) { // Odd number of nodes
    return new Result(head.next, true);
  }

  /* Recurse on sublist. */
  Result res = isPalindromeRecurse(head.next, length - 2);

  /* If child calls are not a palindrome, pass back up
   * a failure. */
  if (!res.result || res.node == null) {
    return res;
  }

  /* Check if matches corresponding node on other side. */
  res.result = (head.data == res.node.data);

  /* Return corresponding node. */
  res.node = res.node.next;

  return res;
}

int lengthOfList(LinkedListNode n) {
  int size = 0;
  while (n != null) {
    size++;
    n = n.next;
  }
  return size;
}
```

Some of you might be wondering why we went through all this effort to create a special `Result` class. Isn't there a better way? Not really—at least not in Java.

However, if we were implementing this in C or C++, we could have passed in a double pointer.

```
bool isPalindromeRecurse(Node head, int length, Node** next) {
  ...
}
```

It's ugly, but it works.

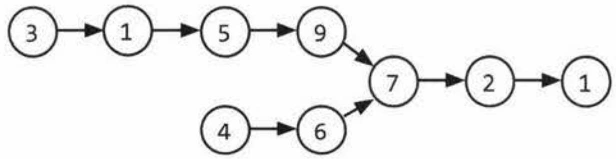
2.7 **Intersection**: Given two (singly) linked lists, determine if the two lists intersect. Return the intersecting node. Note that the intersection is defined based on reference, not value. That is, if the `k`th node of the first linked list is the exact same node (by reference) as the `j`th node of the second linked list, then they are intersecting.

pg 95

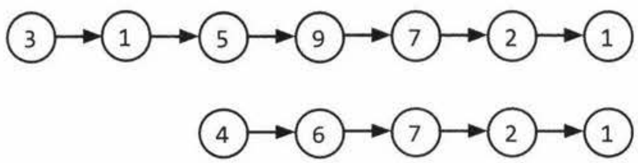
SOLUTION

Let's draw a picture of intersecting linked lists to get a better feel for what is going on.

Here is a picture of intersecting linked lists:



And here is a picture of non-intersecting linked lists:



We should be careful here to not inadvertently draw a special case by making the linked lists the same length.

Let's first ask how we would determine if two linked lists intersect.

Determining if there's an intersection.

How would we detect if two linked lists intersect? One approach would be to use a hash table and just throw all the linked lists nodes into there. We would need to be careful to reference the linked lists by their memory location, not by their value.

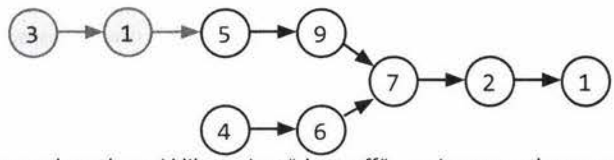
There's an easier way though. Observe that two intersecting linked lists will always have the same last node. Therefore, we can just traverse to the end of each linked list and compare the last nodes.

How do we find where the intersection is, though?

Finding the intersecting node.

One thought is that we could traverse backwards through each linked list. When the linked lists "split", that's the intersection. Of course, you can't really traverse backwards through a singly linked list.

If the linked lists were the same length, you could just traverse through them at the same time. When they collide, that's your intersection.



When they're not the same length, we'd like to just "chop off"—or ignore—those excess (gray) nodes.

How can we do this? Well, if we know the lengths of the two linked lists, then the difference between those two linked lists will tell us how much to chop off.

We can get the lengths at the same time as we get the tails of the linked lists (which we used in the first step to determine if there's an intersection).

Putting it all together.

We now have a multistep process.

1. Run through each linked list to get the lengths and the tails.
2. Compare the tails. If they are different (by reference, not by value), return immediately. There is no intersection.
3. Set two pointers to the start of each linked list.
4. On the longer linked list, advance its pointer by the difference in lengths.
5. Now, traverse on each linked list until the pointers are the same.

The implementation for this is below.

```
LinkedListNode findIntersection(LinkedListNode list1, LinkedListNode list2) {
   if (list1 == null || list2 == null) return null;

   /* Get tail and sizes. */
   Result result1 = getTailAndSize(list1);
   Result result2 = getTailAndSize(list2);

   /* If different tail nodes, then there's no intersection. */
   if (result1.tail != result2.tail) {
      return null;
   }

   /* Set pointers to the start of each linked list. */
   LinkedListNode shorter = result1.size < result2.size ? list1 : list2;
   LinkedListNode longer = result1.size < result2.size ? list2 : list1;

   /* Advance the pointer for the longer linked list by difference in lengths. */
   longer = getKthNode(longer, Math.abs(result1.size - result2.size));

   /* Move both pointers until you have a collision. */
   while (shorter != longer) {
      shorter = shorter.next;
      longer = longer.next;
   }

   /* Return either one. */
   return longer;
}

```

```
class Result {
  public LinkedListNode tail;
  public int size;
  public Result(LinkedListNode tail, int size) {
    this.tail = tail;
    this.size = size;
  }
}

Result getTailAndSize(LinkedListNode list) {
  if (list == null) return null;

  int size = 1;
  LinkedListNode current = list;
  while (current.next != null) {
    size++;
    current = current.next;
  }
  return new Result(current, size);
}

LinkedListNode getKthNode(LinkedListNode head, int k) {
  LinkedListNode current = head;
  while (k > 0 && current != null) {
    current = current.next;
    k--;
  }
  return current;
}
```

This algorithm takes `O(A + B)` time, where A and B are the lengths of the two linked lists. It takes `O(1)` additional space.

2.8 Loop Detection: Given a circular linked list, implement an algorithm that returns the node at the beginning of the loop.

DEFINITION

Circular linked list: A (corrupt) linked list in which a node's next pointer points to an earlier node, so as to make a loop in the linked list.

EXAMPLE

Input: `A -> B -> C -> D -> E -> C` [the same C as earlier]

Output: `C`

pg 95

SOLUTION

This is a modification of a classic interview problem: detect if a linked list has a loop. Let's apply the Pattern Matching approach.

Part 1: Detect If Linked List Has A Loop

An easy way to detect if a linked list has a loop is through the `FastRunner` / `SlowRunner` approach. `FastRunner` moves two steps at a time, while `SlowRunner` moves one step. Much like two cars racing around a track at different steps, they must eventually meet.

An astute reader may wonder if `FastRunner` might "hop over" `SlowRunner` completely, without ever colliding. That's not possible. Suppose that `FastRunner` *did* hop over `SlowRunner`, such that `SlowRunner` is at spot `i` and `FastRunner` is at spot `i + 1`. In the previous step, `SlowRunner` would be at spot `i - 1` and `FastRunner` would at spot `((i + 1) - 2)`, or spot `i - 1`. That is, they would have collided.

Part 2: When Do They Collide?

Let's assume that the linked list has a "non-looped" part of size k.

If we apply our algorithm from part 1, when will `FastRunner` and `SlowRunner` collide?

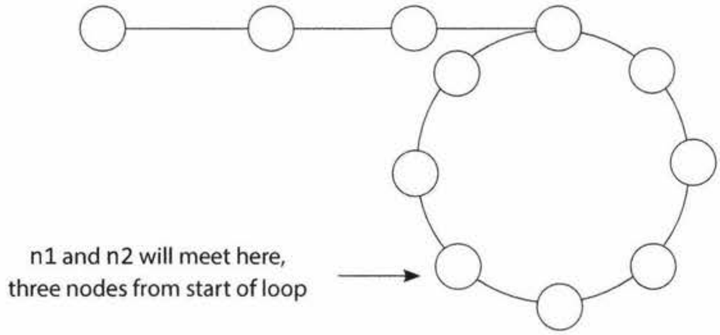
We know that for every p steps that `SlowRunner` takes, `FastRunner` has taken 2p steps. Therefore, when `SlowRunner` enters the looped portion after k steps, `FastRunner` has taken 2k steps total and must be `2k - k` steps, or k steps, into the looped portion. Since k might be much larger than the loop length, we should actually write this as `mod(k, LOOP_SIZE)` steps, which we will denote as K.

At each subsequent step, `FastRunner` and `SlowRunner` get either one step farther away or one step closer, depending on your perspective. That is, because we are in a circle, when A moves q steps away from B, it is also moving q steps closer to B.

So now we know the following facts:

1. `SlowRunner` is 0 steps into the loop.
2. `FastRunner` is K steps into the loop.
3. `SlowRunner` is K steps behind `FastRunner`.
4. `FastRunner` is `LOOP_SIZE - K` steps behind `SlowRunner`.
5. `FastRunner` catches up to `SlowRunner` at a rate of 1 step per unit of time.

So, when do they meet? Well, if `FastRunner` is `LOOP_SIZE - K` steps behind `SlowRunner`, and `FastRunner` catches up at a rate of 1 step per unit of time, then they meet after `LOOP_SIZE - K` steps. At this point, they will be K steps before the head of the loop. Let's call this point `CollisionSpot`.



Part 3: How Do You Find The Start of the Loop?

We now know that `CollisionSpot` is K nodes before the start of the loop. Because `K = mod(k, LOOP_SIZE)` (or, in other words, `k = K + M * LOOP_SIZE`, for any integer M), it is also correct to say that it is k nodes from the loop start. For example, if node N is 2 nodes into a 5 node loop, it is also correct to say that it is 7, 12, or even 397 nodes into the loop.

Therefore, both `CollisionSpot` and `LinkedListHead` are k nodes from the start of the loop.

Now, if we keep one pointer at CollisionSpot and move the other one to LinkedListHead, they will each be k nodes from LoopStart. Moving the two pointers at the same speed will cause them to collide again—this time after k steps, at which point they will both be at LoopStart. All we have to do is return this node.

Part 4: Putting It All Together

To summarize, we move FastPointer twice as fast as SlowPointer. When SlowPointer enters the loop, after k nodes, FastPointer is k nodes into the loop. This means that FastPointer and SlowPointer are LOOP_SIZE - k nodes away from each other.

Next, if FastPointer moves two nodes for each node that SlowPointer moves, they move one node closer to each other on each turn. Therefore, they will meet after LOOP_SIZE - k turns. Both will be k nodes from the front of the loop.

The head of the linked list is also k nodes from the front of the loop. So, if we keep one pointer where it is, and move the other pointer to the head of the linked list, then they will meet at the front of the loop.

Our algorithm is derived directly from parts 1, 2 and 3.

1. Create two pointers, FastPointer and SlowPointer.
2. Move FastPointer at a rate of 2 steps and SlowPointer at a rate of 1 step.
3. When they collide, move SlowPointer to LinkedListHead. Keep FastPointer where it is.
4. Move SlowPointer and FastPointer at a rate of one step. Return the new collision point.

The code below implements this algorithm.

```
LinkedListNode FindBeginning(LinkedListNode head) {
   LinkedListNode slow = head;
   LinkedListNode fast = head;

   /* Find meeting point. This will be LOOP_SIZE - k steps into the linked list. */
   while (fast != null && fast.next != null) {
      slow = slow.next;
      fast = fast.next.next;
      if (slow == fast) { // Collision
         break;
      }
   }

   /* Error check - no meeting point, and therefore no loop */
   if (fast == null || fast.next == null) {
      return null;
   }

   /* Move slow to Head. Keep fast at Meeting Point. Each are k steps from the
    * Loop Start. If they move at the same pace, they must meet at Loop Start. */
   slow = head;
   while (slow != fast) {
      slow = slow.next;
      fast = fast.next;
   }

   /* Both now point to the start of the loop. */
   return fast;
}
```

3

Solutions to Stacks and Queues

3.1 Three in One: Describe how you could use a single array to implement three stacks.

pg 98

SOLUTION

Like many problems, this one somewhat depends on how well we'd like to support these stacks. If we're okay with simply allocating a fixed amount of space for each stack, we can do that. This may mean though that one stack runs out of space, while the others are nearly empty.

Alternatively, we can be flexible in our space allocation, but this significantly increases the complexity of the problem.

Approach 1: Fixed Division

We can divide the array in three equal parts and allow the individual stack to grow in that limited space. Note: We will use the notation "[" to mean inclusive of an end point and "(" to mean exclusive of an end point.

- For stack 1, we will use [0, $\frac{n}{3}$).
- For stack 2, we will use [$\frac{n}{3}$, $\frac{2n}{3}$).
- For stack 3, we will use [$\frac{2n}{3}$, n).

The code for this solution is below.

```
class FixedMultiStack {
   private int numberOfStacks = 3;
   private int stackCapacity;
   private int[] values;
   private int[] sizes;

   public FixedMultiStack(int stackSize) {
      stackCapacity = stackSize;
      values = new int[stackSize * numberOfStacks];
      sizes = new int[numberOfStacks];
   }

   /* Push value onto stack. */
   public void push(int stackNum, int value) throws FullStackException {
      /* Check that we have space for the next element */
      if (isFull(stackNum)) {
         throw new FullStackException();
```

```
        }

        /* Increment stack pointer and then update top value. */
        sizes[stackNum]++;
        values[indexOfTop(stackNum)] = value;
    }

    /* Pop item from top stack. */
    public int pop(int stackNum) {
        if (isEmpty(stackNum)) {
            throw new EmptyStackException();
        }

        int topIndex = indexOfTop(stackNum);
        int value = values[topIndex]; // Get top
        values[topIndex] = 0; // Clear
        sizes[stackNum]--; // Shrink
        return value;
    }

    /* Return top element. */
    public int peek(int stackNum) {
        if (isEmpty(stackNum)) {
            throw new EmptyStackException();
        }
        return values[indexOfTop(stackNum)];
    }

    /* Return if stack is empty. */
    public boolean isEmpty(int stackNum) {
        return sizes[stackNum] == 0;
    }

    /* Return if stack is full. */
    public boolean isFull(int stackNum) {
        return sizes[stackNum] == stackCapacity;
    }

    /* Returns index of the top of the stack. */
    private int indexOfTop(int stackNum) {
        int offset = stackNum * stackCapacity;
        int size = sizes[stackNum];
        return offset + size - 1;
    }
}
```

If we had additional information about the expected usages of the stacks, then we could modify this algorithm accordingly. For example, if we expected Stack 1 to have many more elements than Stack 2, we could allocate more space to Stack 1 and less space to Stack 2.

Approach 2: Flexible Divisions

A second approach is to allow the stack blocks to be flexible in size. When one stack exceeds its initial capacity, we grow the allowable capacity and shift elements as necessary.

We will also design our array to be circular, such that the final stack may start at the end of the array and wrap around to the beginning.

Please note that the code for this solution is far more complex than would be appropriate for an interview. You could be responsible for pseudocode, or perhaps the code of individual components, but the entire implementation would be far too much work.

```
public class MultiStack {
  /* StackInfo is a simple class that holds a set of data about each stack. It
   * does not hold the actual items in the stack. We could have done this with
   * just a bunch of individual variables, but that's messy and doesn't gain us
   * much. */
  private class StackInfo {
    public int start, size, capacity;
    public StackInfo(int start, int capacity) {
      this.start = start;
      this.capacity = capacity;
    }

    /* Check if an index on the full array is within the stack boundaries. The
     * stack can wrap around to the start of the array. */
    public boolean isWithinStackCapacity(int index) {
      /* If outside of bounds of array, return false. */
      if (index < 0 || index >= values.length) {
        return false;
      }

      /* If index wraps around, adjust it. */
      int contiguousIndex = index < start ? index + values.length : index;
      int end = start + capacity;
      return start <= contiguousIndex && contiguousIndex < end;
    }

    public int lastCapacityIndex() {
      return adjustIndex(start + capacity - 1);
    }

    public int lastElementIndex() {
      return adjustIndex(start + size - 1);
    }

    public boolean isFull() { return size == capacity; }
    public boolean isEmpty() { return size == 0; }
  }

  private StackInfo[] info;
  private int[] values;

  public MultiStack(int numberOfStacks, int defaultSize) {
    /* Create metadata for all the stacks. */
    info = new StackInfo[numberOfStacks];
    for (int i = 0; i < numberOfStacks; i++) {
      info[i] = new StackInfo(defaultSize * i, defaultSize);
    }
    values = new int[numberOfStacks * defaultSize];
  }

  /* Push value onto stack num, shifting/expanding stacks as necessary. Throws
   * exception if all stacks are full. */
  public void push(int stackNum, int value) throws FullStackException {
```

```
      if (allStacksAreFull()) {
         throw new FullStackException();
      }

      /* If this stack is full, expand it. */
      StackInfo stack = info[stackNum];
      if (stack.isFull()) {
         expand(stackNum);
      }

      /* Find the index of the top element in the array + 1, and increment the
       * stack pointer */
      stack.size++;
      values[stack.lastElementIndex()] = value;
   }

   /* Remove value from stack. */
   public int pop(int stackNum) throws Exception {
      StackInfo stack = info[stackNum];
      if (stack.isEmpty()) {
         throw new EmptyStackException();
      }

      /* Remove last element. */
      int value = values[stack.lastElementIndex()];
      values[stack.lastElementIndex()] = 0; // Clear item
      stack.size--; // Shrink size
      return value;
   }

   /* Get top element of stack.*/
   public int peek(int stackNum) {
      StackInfo stack = info[stackNum];
      return values[stack.lastElementIndex()];
   }
   /* Shift items in stack over by one element. If we have available capacity, then
    * we'll end up shrinking the stack by one element. If we don't have available
    * capacity, then we'll need to shift the next stack over too. */
   private void shift(int stackNum) {
      System.out.println("/// Shifting " + stackNum);
      StackInfo stack = info[stackNum];

      /* If this stack is at its full capacity, then you need to move the next
       * stack over by one element. This stack can now claim the freed index. */
      if (stack.size >= stack.capacity) {
         int nextStack = (stackNum + 1) % info.length;
         shift(nextStack);
         stack.capacity++; // claim index that next stack lost
      }

      /* Shift all elements in stack over by one. */
      int index = stack.lastCapacityIndex();
      while (stack.isWithinStackCapacity(index)) {
         values[index] = values[previousIndex(index)];
         index = previousIndex(index);
      }
```

```

    /* Adjust stack data. */
    values[stack.start] = 0; // Clear item
    stack.start = nextIndex(stack.start); // move start
    stack.capacity--; // Shrink capacity
  }

  /* Expand stack by shifting over other stacks */
  private void expand(int stackNum) {
    shift((stackNum + 1) % info.length);
    info[stackNum].capacity++;
  }

  /* Returns the number of items actually present in stack. */
  public int numberOfElements() {
    int size = 0;
    for (StackInfo sd : info) {
      size += sd.size;
    }
    return size;
  }

  /* Returns true is all the stacks are full. */
  public boolean allStacksAreFull() {
    return numberOfElements() == values.length;
  }

  /* Adjust index to be within the range of 0 -> length - 1. */
  private int adjustIndex(int index) {
    /* Java's mod operator can return neg values. For example, (-11 % 5) will
     * return -1, not 4. We actually want the value to be 4 (since we're wrapping
     * around the index). */
    int max = values.length;
    return ((index % max) + max) % max;
  }

  /* Get index after this index, adjusted for wrap around. */
  private int nextIndex(int index) {
    return adjustIndex(index + 1);
  }

  /* Get index before this index, adjusted for wrap around. */
  private int previousIndex(int index) {
    return adjustIndex(index - 1);
  }
}
```

In problems like this, it's important to focus on writing clean, maintainable code. You should use additional classes, as we did with `StackInfo`, and pull chunks of code into separate methods. Of course, this advice applies to the "real world" as well.

3.2 Stack Min: How would you design a stack which, in addition to `push` and pop, has a function `min` which returns the minimum element? `Push`, pop and `min` should all operate in `O(1)` time.

pg 98

SOLUTION

The thing with minimums is that they don't change very often. They only change when a smaller element is added.

One solution is to have just a single `int` value, `minValue`, that's a member of the `Stack` class. When `minValue` is popped from the stack, we search through the stack to find the new minimum. Unfortunately, this would break the constraint that `push` and pop operate in `O(1)` time.

To further understand this question, let's walk through it with a short example:

```
push(5); // stack is {5}, min is 5
push(6); // stack is {6, 5}, min is 5
push(3); // stack is {3, 6, 5}, min is 3
push(7); // stack is {7, 3, 6, 5}, min is 3
pop(); // pops 7. stack is {3, 6, 5}, min is 3
pop(); // pops 3. stack is {6, 5}. min is 5.
```

Observe how once the stack goes back to a prior state (`{6, 5}`), the minimum also goes back to its prior state (5). This leads us to our second solution.

If we kept track of the minimum at each state, we would be able to easily know the minimum. We can do this by having each node record what the minimum beneath itself is. Then, to find the `min`, you just look at what the top element thinks is the min.

When you push an element onto the stack, the element is given the current minimum. It sets its "`local min`" to be the `min`.

```
public class StackWithMin extends Stack<NodeWithMin> {
  public void push(int value) {
    int newMin = Math.min(value, min());
    super.push(new NodeWithMin(value, newMin));
  }

  public int min() {
    if (this.isEmpty()) {
      return Integer.MAX_VALUE; // Error value
    } else {
      return peek().min;
    }
  }
}

class NodeWithMin {
  public int value;
  public int min;
  public NodeWithMin(int v, int min){
    value = v;
    this.min = min;
  }
}
```

There's just one issue with this: if we have a large stack, we waste a lot of space by keeping track of the `min` for every single element. Can we do better?

We can (maybe) do a bit better than this by using an additional stack which keeps track of the `mins`.

```
public class StackWithMin2 extends Stack<Integer> {
  Stack<Integer> s2;
  public StackWithMin2() {
    s2 = new Stack<Integer>();
  }

  public void push(int value){
    if (value <= min()) {
      s2.push(value);
    }
    super.push(value);
  }

  public Integer pop() {
    int value = super.pop();
    if (value == min()) {
      s2.pop();
    }
    return value;
  }

  public int min() {
    if (s2.isEmpty()) {
      return Integer.MAX_VALUE;
    } else {
      return s2.peek();
    }
  }
}
```

Why might this be more space efficient? Suppose we had a very large stack and the first element inserted happened to be the minimum. In the first solution, we would be keeping n integers, where n is the size of the stack. In the second solution though, we store just a few pieces of data: a second stack with one element and the members within this stack.

3.3 **Stack of Plates:** Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure `SetOfStacks` that mimics this. `SetOfStacks` should be composed of several stacks and should create a new stack once the previous one exceeds capacity. `SetOfStacks.push()` and `SetOfStacks.pop()` should behave identically to a single stack (that is, `pop()` should return the same values as it would if there were just a single stack).

FOLLOW UP

Implement a function `popAt(int index)` which performs a pop operation on a specific sub-stack.

pg 99

SOLUTION

In this problem, we've been told what our data structure should look like:

```
class SetOfStacks {
  ArrayList<Stack> stacks = new ArrayList<Stack>();
  public void push(int v) { ... }
```

```
    public int pop() { ... }
}
```

We know that push() should behave identically to a single stack, which means that we need push() to call push() on the last stack in the array of stacks. We have to be a bit careful here though: if the last stack is at capacity, we need to create a new stack. Our code should look something like this:

```
void push(int v) {
    Stack last = getLastStack();
    if (last != null && !last.isFull()) { // add to last stack
        last.push(v);
    } else { // must create new stack
        Stack stack = new Stack(capacity);
        stack.push(v);
        stacks.add(stack);
    }
}
```

What should pop() do? It should behave similarly to push() in that it should operate on the last stack. If the last stack is empty (after popping), then we should remove the stack from the list of stacks.

```
int pop() {
    Stack last = getLastStack();
    if (last == null) throw new EmptyStackException();
    int v = last.pop();
    if (last.size == 0) stacks.remove(stacks.size() - 1);
    return v;
}
```

Follow Up: Implement popAt(int index)

This is a bit trickier to implement, but we can imagine a "rollover" system. If we pop an element from stack 1, we need to remove the *bottom* of stack 2 and push it onto stack 1. We then need to rollover from stack 3 to stack 2, stack 4 to stack 3, etc.

You could make an argument that, rather than "rolling over," we should be okay with some stacks not being at full capacity. This would improve the time complexity (by a fair amount, with a large number of elements), but it might get us into tricky situations later on if someone assumes that all stacks (other than the last) operate at full capacity. There's no "right answer" here; you should discuss this trade-off with your interviewer.

```
public class SetOfStacks {
    ArrayList<Stack> stacks = new ArrayList<Stack>();
    public int capacity;
    public SetOfStacks(int capacity) {
        this.capacity = capacity;
    }

    public Stack getLastStack() {
        if (stacks.size() == 0) return null;
        return stacks.get(stacks.size() - 1);
    }

    public void push(int v) { /* see earlier code */ }
    public int pop() { /* see earlier code */ }
    public boolean isEmpty() {
        Stack last = getLastStack();
        return last == null || last.isEmpty();
    }
```

```java

  public int popAt(int index) {
    return leftShift(index, true);
  }

  public int leftShift(int index, boolean removeTop) {
    Stack stack = stacks.get(index);
    int removed_item;
    if (removeTop) removed_item = stack.pop();
    else removed_item = stack.removeBottom();
    if (stack.isEmpty()) {
      stacks.remove(index);
    } else if (stacks.size() > index + 1) {
      int v = leftShift(index + 1, false);
      stack.push(v);
    }
    return removed_item;
  }
}

public class Stack {
  private int capacity;
  public Node top, bottom;
  public int size = 0;

  public Stack(int capacity) { this.capacity = capacity; }
  public boolean isFull() { return capacity == size; }

  public void join(Node above, Node below) {
    if (below != null) below.above = above;
    if (above != null) above.below = below;
  }

  public boolean push(int v) {
    if (size >= capacity) return false;
    size++;
    Node n = new Node(v);
    if (size == 1) bottom = n;
    join(n, top);
    top = n;
    return true;
  }

  public int pop() {
    Node t = top;
    top = top.below;
    size--;
    return t.value;
  }

  public boolean isEmpty() {
    return size == 0;
  }

  public int removeBottom() {
    Node b = bottom;
```

```
    bottom = bottom.above;
    if (bottom != null) bottom.below = null;
    size--;
    return b.value;
  }
}
```

This problem is not conceptually that tough, but it requires a lot of code to implement it fully. Your interviewer would not ask you to implement the entire code.

A good strategy on problems like this is to separate code into other methods, like a `leftShift` method that `popAt` can call. This will make your code cleaner and give you the opportunity to lay down the skeleton of the code before dealing with some of the details.

3.4 Queue via Stacks: Implement a `MyQueue` class which implements a queue using two stacks.

pg 99

SOLUTION

Since the major difference between a queue and a stack is the order (first-in first-out vs. last-in first-out), we know that we need to modify `peek()` and `pop()` to go in reverse order. We can use our second stack to reverse the order of the elements (by popping `s1` and pushing the elements on to `s2`). In such an implementation, on each `peek()` and `pop()` operation, we would pop everything from `s1` onto `s2`, perform the `peek` / pop operation, and then `push` everything back.

This will work, but if two `pop` / `peeks` are performed back-to-back, we're needlessly moving elements. We can implement a "lazy" approach where we let the elements sit in `s2` until we absolutely must reverse the elements.

In this approach, `stackNewest` has the newest elements on top and `stackOldest` has the oldest elements on top. When we dequeue an element, we want to remove the oldest element first, and so we dequeue from `stackOldest`. If `stackOldest` is empty, then we want to transfer all elements from `stackNewest` into this stack in reverse order. To insert an element, we push onto `stackNewest`, since it has the newest elements on top.

The code below implements this algorithm.

```
public class MyQueue<T> {
  Stack<T> stackNewest, stackOldest;

  public MyQueue() {
    stackNewest = new Stack<T>();
    stackOldest = new Stack<T>();
  }

  public int size() {
    return stackNewest.size() + stackOldest.size();
  }

  public void add(T value) {
    /* Push onto stackNewest, which always has the newest elements on top */
    stackNewest.push(value);
  }

  /* Move elements from stackNewest into stackOldest. This is usually done so that
   * we can do operations on stackOldest. */
```

```
    private void shiftStacks() {
       if (stackOldest.isEmpty()) {
          while (!stackNewest.isEmpty()) {
             stackOldest.push(stackNewest.pop());
          }
       }
    }

    public T peek() {
       shiftStacks(); // Ensure stackOldest has the current elements
       return stackOldest.peek(); // retrieve the oldest item.
    }

    public T remove() {
       shiftStacks(); // Ensure stackOldest has the current elements
       return stackOldest.pop(); // pop the oldest item.
    }
 }
```

During your actual interview, you may find that you forget the exact API calls. Don't stress too much if that happens to you. Most interviewers are okay with your asking for them to refresh your memory on little details. They're much more concerned with your big picture understanding.

3.5 **Sort Stack:** Write a program to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure (such as an array). The stack supports the following operations: `push`, `pop`, `peek`, and `isEmpty`.

pg 99

SOLUTION

One approach is to implement a rudimentary sorting algorithm. We search through the entire stack to find the minimum element and then push that onto a new stack. Then, we find the new minimum element and push that. This will actually require a total of three stacks: `s1` is the original stack, `s2` is the final sorted stack, and `s3` acts as a buffer during our searching of `s1`. To search `s1` for each minimum, we need to pop elements from `s1` and push them onto the buffer, `s3`.

Unfortunately, this requires two additional stacks, and we can only use one. Can we do better? Yes.

Rather than searching for the minimum repeatedly, we can sort `s1` by inserting each element from `s1` in order into `s2`. How would this work?

Imagine we have the following stacks, where `s2` is "sorted" and `s1` is not:

s1	s2
	12
5	8
10	3
7	1

When we pop 5 from `s1`, we need to find the right place in `s2` to insert this number. In this case, the correct place is on `s2` just above 3. How do we get it there? We can do this by popping 5 from `s1` and holding it in a temporary variable. Then, we move 12 and 8 over to `s1` (by popping them from `s2` and pushing them onto `s1`) and then push 5 onto `s2`.

Step 1

s1	s2
	12
	8
10	3
7	1

tmp = 5

->

Step 2

s1	s2
8	
12	
10	3
7	1

tmp = 5

->

Step 3

s1	s2
8	
12	5
10	3
7	1

tmp = --

Note that 8 and 12 are still in s1—and that's okay! We just repeat the same steps for those two numbers as we did for 5, each time popping off the top of s1 and putting it into the "right place" on s2. (Of course, since 8 and 12 were moved from s2 to s1 precisely *because* they were larger than 5, the "right place" for these elements will be right on top of 5. We won't need to muck around with s2's other elements, and the inside of the below `while` loop will not be run when `tmp` is 8 or 12.)

```
void sort(Stack<Integer> s) {
   Stack<Integer> r = new Stack<Integer>();
   while(!s.isEmpty()) {
      /* Insert each element in s in sorted order into r. */
      int tmp = s.pop();
      while(!r.isEmpty() && r.peek() > tmp) {
         s.push(r.pop());
      }
      r.push(tmp);
   }

   /* Copy the elements from r back into s. */
   while (!r.isEmpty()) {
      s.push(r.pop());
   }
}
```

This algorithm is $O(N^2)$ time and $O(N)$ space.

If we were allowed to use unlimited stacks, we could implement a modified quicksort or mergesort.

With the mergesort solution, we would create two extra stacks and divide the stack into two parts. We would recursively sort each stack, and then merge them back together in sorted order into the original stack. Note that this would require the creation of two additional stacks per level of recursion.

With the quicksort solution, we would create two additional stacks and divide the stack into the two stacks based on a pivot element. The two stacks would be recursively sorted, and then merged back together into the original stack. Like the earlier solution, this one involves creating two additional stacks per level of recursion.

3.6 Animal Shelter: An animal shelter, which holds only dogs and cats, operates on a strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as `enqueue`, `dequeueAny`, `dequeueDog`, and `dequeueCat`. You may use the built-in `LinkedList` data structure.

pg 99

SOLUTION

We could explore a variety of solutions to this problem. For instance, we could maintain a single queue. This would make `dequeueAny` easy, but `dequeueDog` and `dequeueCat` would require iteration through the queue to find the first dog or cat. This would increase the complexity of the solution and decrease the efficiency.

An alternative approach that is simple, clean and efficient is to simply use separate queues for dogs and cats, and to place them within a wrapper class called `AnimalQueue`. We then store some sort of timestamp to mark when each animal was enqueued. When we call `dequeueAny`, we peek at the heads of both the `dog` and `cat` queue and return the oldest.

```
abstract class Animal {
  private int order;
  protected String name;
  public Animal(String n) { name = n; }
  public void setOrder(int ord) { order = ord; }
  public int getOrder() { return order; }

  /* Compare orders of animals to return the older item. */
  public boolean isOlderThan(Animal a) {
    return this.order < a.getOrder();
  }
}

class AnimalQueue {
  LinkedList<Dog> dogs = new LinkedList<Dog>();
  LinkedList<Cat> cats = new LinkedList<Cat>();
  private int order = 0; // acts as timestamp

  public void enqueue(Animal a) {
    /* Order is used as a sort of timestamp, so that we can compare the insertion
     * order of a dog to a cat. */
    a.setOrder(order);
    order++;

    if (a instanceof Dog) dogs.addLast((Dog) a);
    else if (a instanceof Cat) cats.addLast((Cat)a);
  }

  public Animal dequeueAny() {
    /* Look at tops of dog and cat queues, and pop the queue with the oldest
     * value. */
    if (dogs.size() == 0) {
      return dequeueCats();
    } else if (cats.size() == 0) {
      return dequeueDogs();
    }
```

```java

        Dog dog = dogs.peek();
        Cat cat = cats.peek();
        if (dog.isOlderThan(cat)) {
            return dequeueDogs();
        } else {
            return dequeueCats();
        }
    }

    public Dog dequeueDogs() {
        return dogs.poll();
    }

    public Cat dequeueCats() {
        return cats.poll();
    }
}

public class Dog extends Animal {
    public Dog(String n) { super(n); }
}

public class Cat extends Animal {
    public Cat(String n) { super(n); }
}
```

It is important that Dog and `Cat` both inherit from an `Animal` class since `dequeueAny()` needs to be able to support returning both Dog and `Cat` objects.

If we wanted, `order` could be a true timestamp with the actual date and time. The advantage of this is that we wouldn't have to set and maintain the numerical order. If we somehow wound up with two animals with the same timestamp, then (by definition) we don't have an older animal and we could return either one.

4

Solutions to Trees and Graphs

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

pg 109

SOLUTION

This problem can be solved by just simple graph traversal, such as depth-first search or breadth-first search. We start with one of the two nodes and, during traversal, check if the other node is found. We should mark any node found in the course of the algorithm as "already visited" to avoid cycles and repetition of the nodes.

The code below provides an iterative implementation of breadth-first search.

```
enum State { Unvisited, Visited, Visiting; }

boolean search(Graph g, Node start, Node end) {
  if (start == end) return true;

  // operates as Queue
  LinkedList<Node> q = new LinkedList<Node>();

  for (Node u : g.getNodes()) {
    u.state = State.Unvisited;
  }
  start.state = State.Visiting;
  q.add(start);
  Node u;
  while (!q.isEmpty()) {
    u = q.removeFirst(); // i.e., dequeue()
    if (u != null) {
      for (Node v : u.getAdjacent()) {
        if (v.state == State.Unvisited) {
          if (v == end) {
            return true;
          } else {
            v.state = State.Visiting;
            q.add(v);
          }
        }
      }
      u.state = State.Visited;
    }
```

```
    }
    return false;
}
```

It may be worth discussing with your interviewer the tradeoffs between breadth-first search and depth-first search for this and other problems. For example, depth-first search is a bit simpler to implement since it can be done with simple recursion. Breadth-first search can also be useful to find the shortest path, whereas depth-first search may traverse one adjacent node very deeply before ever going onto the immediate neighbors.

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.

pg 109

SOLUTION

To create a tree of minimal height, we need to match the number of nodes in the left subtree to the number of nodes in the right subtree as much as possible. This means that we want the root to be the middle of the array, since this would mean that half the elements would be less than the root and half would be greater than it.

We proceed with constructing our tree in a similar fashion. The middle of each subsection of the array becomes the root of the node. The left half of the array will become our left subtree, and the right half of the array will become the right subtree.

One way to implement this is to use a simple `root.insertNode(int v)` method which inserts the value v through a recursive process that starts with the root node. This will indeed construct a tree with minimal height but it will not do so very efficiently. Each insertion will require traversing the tree, giving a total cost of `O(N log N)` to the tree.

Alternatively, we can cut out the extra traversals by recursively using the `createMinimalBST` method. This method is passed just a subsection of the array and returns the root of a minimal tree for that array.

The algorithm is as follows:

1. Insert into the tree the middle element of the array.
2. Insert (into the left subtree) the left subarray elements.
3. Insert (into the right subtree) the right subarray elements.
4. Recurse.

The code below implements this algorithm.

```
TreeNode createMinimalBST(int array[]) {
  return createMinimalBST(array, 0, array.length - 1);
}

TreeNode createMinimalBST(int arr[], int start, int end) {
  if (end < start) {
    return null;
  }
  int mid = (start + end) / 2;
  TreeNode n = new TreeNode(arr[mid]);
  n.left = createMinimalBST(arr, start, mid - 1);
  n.right = createMinimalBST(arr, mid + 1, end);
  return n;
```

```
}
```

Although this code does not seem especially complex, it can be very easy to make little off-by-one errors. Be sure to test these parts of the code very thoroughly.

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).

pg 109

SOLUTION

Though we might think at first glance that this problem requires a level-by-level traversal, this isn't actually necessary. We can traverse the graph any way that we'd like, provided we know which level we're on as we do so.

We can implement a simple modification of the pre-order traversal algorithm, where we pass in `level + 1` to the next recursive call. The code below provides an implementation using depth-first search.

```
void createLevelLinkedList(TreeNode root, ArrayList<LinkedList<TreeNode>> lists,
                           int level) {
  if (root == null) return; // base case

  LinkedList<TreeNode> list = null;
  if (lists.size() == level) { // Level not contained in list
    list = new LinkedList<TreeNode>();
    /* Levels are always traversed in order. So, if this is the first time we've
     * visited level i, we must have seen levels 0 through i - 1. We can
     * therefore safely add the level at the end. */
    lists.add(list);
  } else {
    list = lists.get(level);
  }
  list.add(root);
  createLevelLinkedList(root.left, lists, level + 1);
  createLevelLinkedList(root.right, lists, level + 1);
}

ArrayList<LinkedList<TreeNode>> createLevelLinkedList(TreeNode root) {
  ArrayList<LinkedList<TreeNode>> lists = new ArrayList<LinkedList<TreeNode>>();
  createLevelLinkedList(root, lists, 0);
  return lists;
}
```

Alternatively, we can also implement a modification of breadth-first search. With this implementation, we want to iterate through the root first, then level 2, then level 3, and so on.

With each level `i`, we will have already fully visited all nodes on level `i - 1`. This means that to get which nodes are on level `i`, we can simply look at all children of the nodes of level `i - 1`.

The code below implements this algorithm.

```
ArrayList<LinkedList<TreeNode>> createLevelLinkedList(TreeNode root) {
  ArrayList<LinkedList<TreeNode>> result = new ArrayList<LinkedList<TreeNode>>();
  /* "Visit" the root */
  LinkedList<TreeNode> current = new LinkedList<TreeNode>();
  if (root != null) {
    current.add(root);
  }
```

```
   while (current.size() > 0) {
      result.add(current); // Add previous level
      LinkedList<TreeNode> parents = current; // Go to next level
      current = new LinkedList<TreeNode>();
      for (TreeNode parent : parents) {
         /* Visit the children */
         if (parent.left != null) {
            current.add(parent.left);
         }
         if (parent.right != null) {
            current.add(parent.right);
         }
      }
   }
   return result;
}
```

One might ask which of these solutions is more efficient. Both run in O(N) time, but what about the space efficiency? At first, we might want to claim that the second solution is more space efficient.

In a sense, that's correct. The first solution uses O(log N) recursive calls (in a balanced tree), each of which adds a new level to the stack. The second solution, which is iterative, does not require this extra space.

However, both solutions require returning O(N) data. The extra O(log N) space usage from the recursive implementation is dwarfed by the O(N) data that must be returned. So while the first solution may actually use more data, they are equally efficient when it comes to "big O."

4.4 Check Balanced: Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.

pg 110

SOLUTION

In this question, we've been fortunate enough to be told exactly what balanced means: that for each node, the two subtrees differ in height by no more than one. We can implement a solution based on this definition. We can simply recurse through the entire tree, and for each node, compute the heights of each subtree.

```
int getHeight(TreeNode root) {
   if (root == null) return -1; // Base case
   return Math.max(getHeight(root.left), getHeight(root.right)) + 1;
}

boolean isBalanced(TreeNode root) {
   if (root == null) return true; // Base case

   int heightDiff = getHeight(root.left) - getHeight(root.right);
   if (Math.abs(heightDiff) > 1) {
      return false;
   } else { // Recurse
      return isBalanced(root.left) && isBalanced(root.right);
   }
}
```

Although this works, it's not very efficient. On each node, we recurse through its entire subtree. This means that `getHeight` is called repeatedly on the same nodes. The algorithm is `O(N log N)` since each node is "touched" once per node above it.

We need to cut out some of the calls to `getHeight`.

If we inspect this method, we may notice that `getHeight` could actually check if the tree is balanced at the same time as it's checking heights. What do we do when we discover that the subtree isn't balanced? Just return an error code.

This improved algorithm works by checking the height of each subtree as we recurse down from the root. On each node, we recursively get the heights of the left and right subtrees through the `checkHeight` method. If the subtree is balanced, then `checkHeight` will return the actual height of the subtree. If the subtree is not balanced, then `checkHeight` will return an error code. We will immediately break and return an error code from the current call.

> What do we use for an error code? The height of a null tree is generally defined to be -1, so that's not a great idea for an error code. Instead, we'll use `Integer.MIN_VALUE`.

The code below implements this algorithm.

```
int checkHeight(TreeNode root) {
  if (root == null) return -1;

  int leftHeight = checkHeight(root.left);
  if (leftHeight == Integer.MIN_VALUE) return Integer.MIN_VALUE; // Pass error up

  int rightHeight = checkHeight(root.right);
  if (rightHeight == Integer.MIN_VALUE) return Integer.MIN_VALUE; // Pass error up

  int heightDiff = leftHeight - rightHeight;
  if (Math.abs(heightDiff) > 1) {
    return Integer.MIN_VALUE; // Found error -> pass it back
  } else {
    return Math.max(leftHeight, rightHeight) + 1;
  }
}

boolean isBalanced(TreeNode root) {
  return checkHeight(root) != Integer.MIN_VALUE;
}
```

This code runs in `O(N)` time and `O(H)` space, where H is the height of the tree.

4.5 Validate BST: Implement a function to check if a binary tree is a binary search tree.

pg 110

SOLUTION

We can implement this solution in two different ways. The first leverages the in-order traversal, and the second builds off the property that `left <= current < right`.

Solution #1: In-Order Traversal

Our first thought might be to do an in-order traversal, copy the elements to an array, and then check to see if the array is sorted. This solution takes up a bit of extra memory, but it works—mostly.

The only problem is that it can't handle duplicate values in the tree properly. For example, the algorithm cannot distinguish between the two trees below (one of which is invalid) since they have the same in-order traversal.

Valid BST **Invalid BST**

However, if we assume that the tree cannot have duplicate values, then this approach works. The pseudocode for this method looks something like:

```
int index = 0;
void copyBST(TreeNode root, int[] array) {
  if (root == null) return;
  copyBST(root.left, array);
  array[index] = root.data;
  index++;
  copyBST(root.right, array);
}

boolean checkBST(TreeNode root) {
  int[] array = new int[root.size];
  copyBST(root, array);
  for (int i = 1; i < array.length; i++) {
    if (array[i] <= array[i - 1]) return false;
  }
  return true;
}
```

Note that it is necessary to keep track of the logical "end" of the array, since it would be allocated to hold all the elements.

When we examine this solution, we find that the array is not actually necessary. We never use it other than to compare an element to the previous element. So why not just track the last element we saw and compare it as we go?

The code below implements this algorithm.

```
Integer last_printed = null;
boolean checkBST(TreeNode n) {
  if (n == null) return true;

  // Check / recurse left
  if (!checkBST(n.left)) return false;

  // Check current
  if (last_printed != null && n.data <= last_printed) {
    return false;
  }
  last_printed = n.data;

  // Check / recurse right
```

```
    if (!checkBST(n.right)) return false;

    return true; // All good!
}
```

We've used an `Integer` instead of `int` so that we can know when `last_printed` has been set to a value.

If you don't like the use of static variables, then you can tweak this code to use a wrapper class for the integer, as shown below.

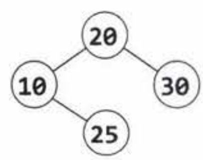
```
class WrapInt {
  public int value;
}
```

Or, if you're implementing this in C++ or another language that supports passing integers by reference, then you can simply do that.

Solution #2: The Min / Max Solution

In the second solution, we leverage the definition of the binary search tree.

What does it mean for a tree to be a binary search tree? We know that it must, of course, satisfy the condition `left.data <= current.data < right.data` for each node, but this isn't quite sufficient. Consider the following small tree:

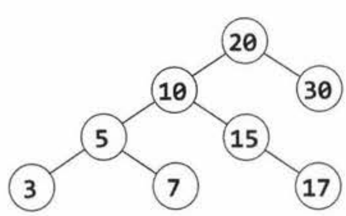


Although each node is bigger than its left node and smaller than its right node, this is clearly not a binary search tree since 25 is in the wrong place.

More precisely, the condition is that *all* left nodes must be less than or equal to the current node, which must be less than all the right nodes.

Using this thought, we can approach the problem by passing down the min and max values. As we iterate through the tree, we verify against progressively narrower ranges.

Consider the following sample tree:



We start with a range of `(min = NULL, max = NULL)`, which the root obviously meets. (`NULL` indicates that there is no min or max.) We then branch left, checking that these nodes are within the range `(min = NULL, max = 20)`. Then, we branch right, checking that the nodes are within the range `(min = 20, max = NULL)`.

We proceed through the tree with this approach. When we branch left, the max gets updated. When we branch right, the min gets updated. If anything fails these checks, we stop and return false.

The time complexity for this solution is O(N), where N is the number of nodes in the tree. We can prove that this is the best we can do, since any algorithm must touch all N nodes.

Due to the use of recursion, the space complexity is O(log N) on a balanced tree. There are up to O(log N) recursive calls on the stack since we may recurse up to the depth of the tree.

The recursive code for this is as follows:

```
boolean checkBST(TreeNode n) {
  return checkBST(n, null, null);
}

boolean checkBST(TreeNode n, Integer min, Integer max) {
  if (n == null) {
    return true;
  }
  if ((min != null && n.data <= min) || (max != null && n.data > max)) {
    return false;
  }

  if (!checkBST(n.left, min, n.data) || !checkBST(n.right, n.data, max)) {
    return false;
  }
  return true;
}
```

Remember that in recursive algorithms, you should always make sure that your base cases, as well as your null cases, are well handled.

4.6 **Successor:** Write an algorithm to find the "next" node (i.e., in-order successor) of a given node in a binary search tree. You may assume that each node has a link to its parent.

pg 110

SOLUTION

Recall that an in-order traversal traverses the left subtree, then the current node, then the right subtree. To approach this problem, we need to think very, very carefully about what happens.

Let's suppose we have a hypothetical node. We know that the order goes left subtree, then current side, then right subtree. So, the next node we visit should be on the right side.

But which node on the right subtree? It should be the first node we'd visit if we were doing an in-order traversal of that subtree. This means that it should be the leftmost node on the right subtree. Easy enough!

But what if the node doesn't have a right subtree? This is where it gets a bit trickier.

If a node n doesn't have a right subtree, then we are done traversing n's subtree. We need to pick up where we left off with n's parent, which we'll call q.

If n was to the left of q, then the next node we should traverse should be q (again, since `left -> current -> right`).

If n were to the right of q, then we have fully traversed q's subtree as well. We need to traverse upwards from q until we find a node x that we have *not* fully traversed. How do we know that we have not fully traversed a node x? We know we have hit this case when we move from a left node to its parent. The left node is fully traversed, but its parent is not.

The pseudocode looks like this:

```
Node inorderSucc(Node n) {
   if (n has a right subtree) {
      return leftmost child of right subtree
   } else {
      while (n is a right child of n.parent) {
         n = n.parent; // Go up
      }
      return n.parent; // Parent has not been traversed
   }
}
```

But wait—what if we traverse all the way up the tree before finding a left child? This will happen only when we hit the very end of the in-order traversal. That is, if we're *already* on the far right of the tree, then there is no in-order successor. We should return `null`.

The code below implements this algorithm (and properly handles the null case).

```
TreeNode inorderSucc(TreeNode n) {
   if (n == null) return null;

   /* Found right children -> return leftmost node of right subtree. */
   if (n.right != null) {
      return leftMostChild(n.right);
   } else {
      TreeNode q = n;
      TreeNode x = q.parent;
      // Go up until we're on left instead of right
      while (x != null && x.left != q) {
         q = x;
         x = x.parent;
      }
      return x;
   }
}

TreeNode leftMostChild(TreeNode n) {
   if (n == null) {
      return null;
   }
   while (n.left != null) {
      n = n.left;
   }
   return n;
}
```

This is not the most algorithmically complex problem in the world, but it can be tricky to code perfectly. In a problem like this, it's useful to sketch out pseudocode to carefully outline the different cases.

4.7 Build Order: You are given a list of projects and a list of dependencies (which is a list of pairs of projects, where the second project is dependent on the first project). All of a project's dependencies must be built before the project is. Find a build order that will allow the projects to be built. If there is no valid build order, return an error.

EXAMPLE

Input:

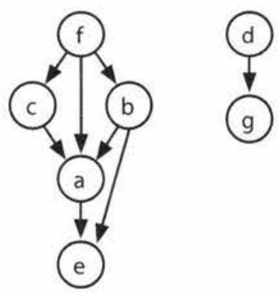
```
projects: a, b, c, d, e, f
dependencies: (a, d), (f, b), (b, d), (f, a), (d, c)
```

Output: `f, e, a, b, d, c`

pg 110

SOLUTION

Visualizing the information as a graph probably works best. Be careful with the direction of the arrows. In the graph below, an arrow from d to g means that d must be compiled before g. You can also draw them in the opposite direction, but you need to consistent and clear about what you mean. Let's draw a fresh example.



In drawing this example (which is *not* the example from the problem description), I looked for a few things.

- I wanted the nodes labeled somewhat randomly. If I had instead put a at the top, with b and c as children, then d and e, it could be misleading. The alphabetical order would match the compile order.
- I wanted a graph with multiple parts/components, since a connected graph is a bit of a special case.
- I wanted a graph where a node links to a node that cannot immediately follow it. For example, `f` links to a but a cannot immediately follow it (since b and c must come before a and after `f`).
- I wanted a larger graph since I need to figure out the pattern.
- I wanted nodes with multiple dependencies.

Now that we have a good example, let's get started with an algorithm.

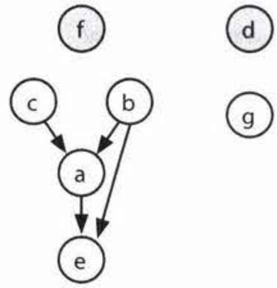
Solution #1

Where do we start? Are there any nodes that we can definitely compile immediately?

Yes. Nodes with no incoming edges can be built immediately since they don't depend on anything. Let's add all such nodes to the build order. In the earlier example, this means we have an order of `f`, d (or d, `f`).

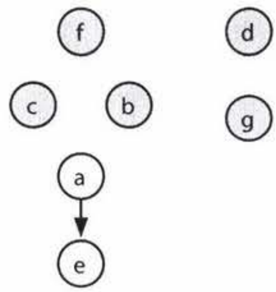
Once we've done that, it's irrelevant that some nodes are dependent on d and `f` since d and `f` have already been built. We can reflect this new state by removing d and `f`'s outgoing edges.

```
build order: f, d
```



Next, we know that c, b, and g are free to build since they have no incoming edges. Let's build those and then remove their outgoing edges.

```
build order: f, d, c, b, g
```



Project a can be built next, so let's do that and remove its outgoing edges. This leaves just e. We build that next, giving us a complete build order.

```
build order: f, d, c, b, g, a, e
```

Did this algorithm work, or did we just get lucky? Let's think about the logic.

1. We first added the nodes with no incoming edges. If the set of projects can be built, there must be some "first" project, and that project can't have any dependencies. If a project has no dependencies (incoming edges), then we certainly can't break anything by building it first.
2. We removed all outgoing edges from these roots. This is reasonable. Once those root projects were built, it doesn't matter if another project depends on them.
3. After that, we found the nodes that *now* have no incoming edges. Using the same logic from steps 1 and 2, it's okay if we build these. Now we just repeat the same steps: find the nodes with no dependencies, add them to the build order, remove their outgoing edges, and repeat.
4. What if there are nodes remaining, but all have dependencies (incoming edges)? This means there's no way to build the system. We should return an error.

The implementation follows this approach very closely.

Initialization and setup:

1. Build a graph where each project is a node and its outgoing edges represent the projects that depend on it. That is, if A has an edge to B (A -> B), it means B has a dependency on A and therefore A must be built before B. Each node also tracks the number of *incoming* edges.
2. Initialize a `buildOrder` array. Once we determine a project's build order, we add it to the array. We also continue to iterate through the array, using a `toBeProcessed` pointer to point to the next node to be fully processed.

3. Find all the nodes with zero incoming edges and add those to a `buildOrder` array. Set a `toBeProcessed` pointer to the beginning of the array.

Repeat until `toBeProcessed` is at the end of the `buildOrder`:

1. Read node at `toBeProcessed`.
   - If node is `null`, then all remaining nodes have a dependency and we have detected a cycle.
2. For each child of node:
   - Decrement `child.dependencies` (the number of incoming edges).
   - If `child.dependencies` is zero, add `child` to end of `buildOrder`.
3. Increment `toBeProcessed`.

The code below implements this algorithm.

```
/* Find a correct build order. */
Project[] findBuildOrder(String[] projects, String[][] dependencies) {
  Graph graph = buildGraph(projects, dependencies);
  return orderProjects(graph.getNodes());
}

/* Build the graph, adding the edge (a, b) if b is dependent on a. Assumes a pair
 * is listed in "build order". The pair (a, b) in dependencies indicates that b
 * depends on a and a must be built before b. */
Graph buildGraph(String[] projects, String[][] dependencies) {
  Graph graph = new Graph();
  for (String project : projects) {
    graph.createNode(project);
  }

  for (String[] dependency : dependencies) {
    String first = dependency[0];
    String second = dependency[1];
    graph.addEdge(first, second);
  }

  return graph;
}

/* Return a list of the projects a correct build order.*/
Project[] orderProjects(ArrayList<Project> projects) {
  Project[] order = new Project[projects.size()];

  /* Add "roots" to the build order first.*/
  int endOfList = addNonDependent(order, projects, 0);

  int toBeProcessed = 0;
  while (toBeProcessed < order.length) {
    Project current = order[toBeProcessed];

    /* We have a circular dependency since there are no remaining projects with
     * zero dependencies. */
    if (current == null) {
      return null;
    }

```

```
      /* Remove myself as a dependency. */
      ArrayList<Project> children = current.getChildren();
      for (Project child : children) {
        child.decrementDependencies();
      }

      /* Add children that have no one depending on them. */
      endOfList = addNonDependent(order, children, endOfList);
      toBeProcessed++;
    }

    return order;
}

/* A helper function to insert projects with zero dependencies into the order
 * array, starting at index offset. */
int addNonDependent(Project[] order, ArrayList<Project> projects, int offset) {
    for (Project project : projects) {
      if (project.getNumberDependencies() == 0) {
        order[offset] = project;
        offset++;
      }
    }
    return offset;
}

public class Graph {
    private ArrayList<Project> nodes = new ArrayList<Project>();
    private HashMap<String, Project> map = new HashMap<String, Project>();

    public Project getOrCreateNode(String name) {
      if (!map.containsKey(name)) {
        Project node = new Project(name);
        nodes.add(node);
        map.put(name, node);
      }

      return map.get(name);
    }

    public void addEdge(String startName, String endName) {
      Project start = getOrCreateNode(startName);
      Project end = getOrCreateNode(endName);
      start.addNeighbor(end);
    }

    public ArrayList<Project> getNodes() { return nodes; }
}

public class Project {
    private ArrayList<Project> children = new ArrayList<Project>();
    private HashMap<String, Project> map = new HashMap<String, Project>();
    private String name;
    private int dependencies = 0;

    public Project(String n) { name = n; }
```

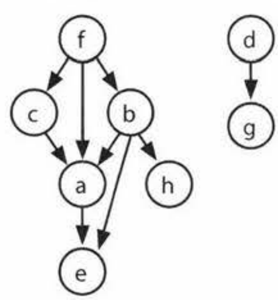
```
  public void addNeighbor(Project node) {
     if (!map.containsKey(node.getName())) {
        children.add(node);
        map.put(node.getName(), node);
        node.incrementDependencies();
     }
  }

  public void incrementDependencies() { dependencies++; }
  public void decrementDependencies() { dependencies--; }

  public String getName() { return name; }
  public ArrayList<Project> getChildren() { return children; }
  public int getNumberDependencies() { return dependencies; }
}
```

This solution takes O(P + D) time, where P is the number of projects and D is the number of dependency pairs.

> **Note:** You might recognize this as the topological sort algorithm on page 632. We've rederived this from scratch. Most people won't know this algorithm and it's reasonable for an interviewer to expect you to be able to derive it.

Solution #2

Alternatively, we can use depth-first search (DFS) to find the build path.



Suppose we picked an arbitrary node (say b) and performed a depth-first search on it. When we get to the end of a path and can't go any further (which will happen at h and e), we know that those terminating nodes can be the last projects to be built. No projects depend on them.

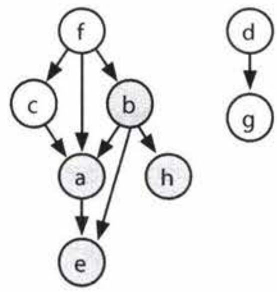
```
DFS(b)                                // Step 1
    DFS(h)                            // Step 2
        build order = ..., h          // Step 3
    DFS(a)                            // Step 4
        DFS(e)                        // Step 5
            build order = ..., e, h   // Step 6
        ...                           // Step 7+
    ...
```

Now, consider what happens at node a when we return from the DFS of e. We know a's children need to appear after a in the build order. So, once we return from searching a's children (and therefore they have been added), we can choose to add a to the front of the build order.

Once we return from a, and complete the DFS of b's other children, then everything that must appear after b is in the list. Add b to the front.

```
DFS(b)                              // Step 1
    DFS(h)                          // Step 2
        build order = ..., h        // Step 3
    DFS(a)                          // Step 4
        DFS(e)                      // Step 5
            build order = ..., e, h // Step 6
        build order = ..., a, e, h  // Step 7
    DFS(e) -> return                // Step 8
    build order = ..., b, a, e, h   // Step 9
```

Let's mark these nodes as having been built too, just in case someone else needs to build them.



Now what? We can start with any old node again, doing a DFS on it and then adding the node to the front of the build queue when the DFS is completed.

```
DFS(d)
    DFS(g)
        build order = ..., g, b, a, e, h
    build order = ..., d, g, b, a, e, h

DFS(f)
    DFS(c)
        build order = ..., c, d, g, b, a, e, h
    build order = f, c, d, g, b, a, e, h
```

In an algorithm like this, we should think about the issue of cycles. There is no possible build order if there is a cycle. But still, we don't want to get stuck in an infinite loop just because there's no possible solution.

A cycle will happen if, while doing a DFS on a node, we run back into the same path. What we need therefore is a signal that indicates "I'm still processing this node, so if you see the node again, we have a problem."

What we can do for this is to mark each node as a "`partial`" (or "`is visiting`") state just before we start the DFS on it. If we see any node whose state is `partial`, then we know we have a problem. When we're done with this node's DFS, we need to update the state.

We also need a state to indicate "I've already processed/built this node" so we don't re-build the node. Our state therefore can have three options: `COMPLETED`, `PARTIAL`, and `BLANK`.

The code below implements this algorithm.

```
Stack<Project> findBuildOrder(String[] projects, String[][] dependencies) {
  Graph graph = buildGraph(projects, dependencies);
  return orderProjects(graph.getNodes());
}
```

```java

Stack<Project> orderProjects(ArrayList<Project> projects) {
  Stack<Project> stack = new Stack<Project>();
  for (Project project : projects) {
    if (project.getState() == Project.State.BLANK) {
      if (!doDFS(project, stack)) {
        return null;
      }
    }
  }
  return stack;
}

boolean doDFS(Project project, Stack<Project> stack) {
  if (project.getState() == Project.State.PARTIAL) {
    return false; // Cycle
  }

  if (project.getState() == Project.State.BLANK) {
    project.setState(Project.State.PARTIAL);
    ArrayList<Project> children = project.getChildren();
    for (Project child : children) {
      if (!doDFS(child, stack)) {
        return false;
      }
    }
    project.setState(Project.State.COMPLETE);
    stack.push(project);
  }
  return true;
}

/* Same as before */
Graph buildGraph(String[] projects, String[][] dependencies) {...}
public class Graph {}

/* Essentially equivalent to earlier solution, with state info added and
 * dependency count removed. */
public class Project {
  public enum State {COMPLETE, PARTIAL, BLANK};
  private State state = State.BLANK;
  public State getState() { return state; }
  public void setState(State st) { state = st; }
  /* Duplicate code removed for brevity */
}
```

Like the earlier algorithm, this solution is O(P+D) time, where P is the number of projects and D is the number of dependency pairs.

By the way, this problem is called **topological sort**: linearly ordering the vertices in a graph such that for every edge (a, b), a appears before b in the linear order.

4.8 **First Common Ancestor:** Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.

pg 110

SOLUTION

If this were a binary search tree, we could modify the `find` operation for the two nodes and see where the paths diverge. Unfortunately, this is not a binary search tree, so we must try other approaches.

Let's assume we're looking for the common ancestor of nodes p and q. One question to ask here is if each node in our tree has a link to its parents.

Solution #1: With Links to Parents

If each node has a link to its parent, we could trace p and q's paths up until they intersect. This is essentially the same problem as question 2.7 which find the intersection of two linked lists. The "linked list" in this case is the path from each node up to the root. (Review this solution on page 221.)

```
TreeNode commonAncestor(TreeNode p, TreeNode q) {
  int delta = depth(p) - depth(q); // get difference in depths
  TreeNode first = delta > 0 ? q : p; // get shallower node
  TreeNode second = delta > 0 ? p : q; // get deeper node
  second = goUpBy(second, Math.abs(delta)); // move deeper node up

  /* Find where paths intersect. */
  while (first != second && first != null && second != null) {
    first = first.parent;
    second = second.parent;
  }
  return first == null || second == null ? null : first;
}

TreeNode goUpBy(TreeNode node, int delta) {
  while (delta > 0 && node != null) {
    node = node.parent;
    delta--;
  }
  return node;
}

int depth(TreeNode node) {
  int depth = 0;
  while (node != null) {
    node = node.parent;
    depth++;
  }
  return depth;
}
```

This approach will take `O(d)` time, where d is the depth of the deeper node.

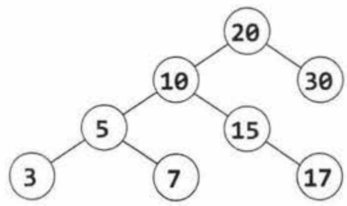
Solution #2: With Links to Parents (Better Worst-Case Runtime)

Similar to the earlier approach, we could trace p's path upwards and check if any of the nodes cover q. The first node that covers q (we already know that every node on this path will cover p) must be the first common ancestor.

Observe that we don't need to re-check the entire subtree. As we move from a node x to its parent y, all the nodes under x have already been checked for q. Therefore, we only need to check the new nodes "uncovered", which will be the nodes under x's sibling.

For example, suppose we're looking for the first common ancestor of node `p = 7` and node `q = 17`. When we go to `p.parent` (5), we uncover the subtree rooted at 3. We therefore need to search this subtree for q.

Next, we go to node 10, uncovering the subtree rooted at 15. We check this subtree for node 17 and—voila—there it is.



To implement this, we can just traverse upwards from p, storing the `parent` and the `sibling` node in a variable. (The `sibling` node is always a child of `parent` and refers to the newly uncovered subtree.) At each iteration, `sibling` gets set to the old `parent`'s sibling node and `parent` gets set to `parent.parent`.

```
TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
  /* Check if either node is not in the tree, or if one covers the other. */
  if (!covers(root, p) || !covers(root, q)) {
    return null;
  } else if (covers(p, q)) {
    return p;
  } else if (covers(q, p)) {
    return q;
  }

  /* Traverse upwards until you find a node that covers q. */
  TreeNode sibling = getSibling(p);
  TreeNode parent = p.parent;
  while (!covers(sibling, q)) {
    sibling = getSibling(parent);
    parent = parent.parent;
  }
  return parent;
}

boolean covers(TreeNode root, TreeNode p) {
  if (root == null) return false;
  if (root == p) return true;
  return covers(root.left, p) || covers(root.right, p);
}

TreeNode getSibling(TreeNode node) {
  if (node == null || node.parent == null) {
    return null;
  }

  TreeNode parent = node.parent;
```

```
  return parent.left == node ? parent.right : parent.left;
}
```

This algorithm takes `O(t)` time, where `t` is the size of the subtree for the first common ancestor. In the worst case, this will be `O(n)`, where n is the number of nodes in the tree. We can derive this runtime by noticing that each node in that subtree is searched once.

Solution #3: Without Links to Parents

Alternatively, you could follow a chain in which p and q are on the same side. That is, if p and q are both on the left of the node, branch left to look for the common ancestor. If they are both on the right, branch right to look for the common ancestor. When p and q are no longer on the same side, you must have found the first common ancestor.

The code below implements this approach.

```
TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
  /* Error check - one node is not in the tree. */
  if (!covers(root, p) || !covers(root, q)) {
    return null;
  }
  return ancestorHelper(root, p, q);
}

TreeNode ancestorHelper(TreeNode root, TreeNode p, TreeNode q) {
  if (root == null || root == p || root == q) {
    return root;
  }

  boolean pIsOnLeft = covers(root.left, p);
  boolean qIsOnLeft = covers(root.left, q);
  if (pIsOnLeft != qIsOnLeft) { // Nodes are on different side
    return root;
  }
  TreeNode childSide = pIsOnLeft ? root.left : root.right;
  return ancestorHelper(childSide, p, q);
}

boolean covers(TreeNode root, TreeNode p) {
  if (root == null) return false;
  if (root == p) return true;
  return covers(root.left, p) || covers(root.right, p);
}
```

This algorithm runs in `O(n)` time on a balanced tree. This is because `covers` is called on 2n nodes in the first call (n nodes for the left side, and n nodes for the right side). After that, the algorithm branches left or right, at which point `covers` will be called on $\frac{2n}{2}$ nodes, then $\frac{2n}{4}$, and so on. This results in a runtime of `O(n)`.

We know at this point that we cannot do better than that in terms of the asymptotic runtime since we need to potentially look at every node in the tree. However, we may be able to improve it by a constant multiple.

Solution #4: Optimized

Although Solution #3 is optimal in its runtime, we may recognize that there is still some inefficiency in how it operates. Specifically, `covers` searches all nodes under `root` for p and q, including the nodes in each subtree (`root.left` and `root.right`). Then, it picks one of those subtrees and searches all of its nodes. Each subtree is searched over and over again.

We may recognize that we should only need to search the entire tree once to find p and q. We should then be able to "bubble up" the findings to earlier nodes in the stack. The basic logic is the same as the earlier solution.

We recurse through the entire tree with a function called `commonAncestor(TreeNode root, TreeNode p, TreeNode q)`. This function returns values as follows:

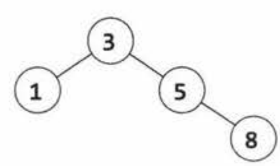
- Returns p, if `root`'s subtree includes p (and not q).
- Returns q, if `root`'s subtree includes q (and not p).
- Returns `null`, if neither p nor q are in `root`'s subtree.
- Else, returns the common ancestor of p and q.

Finding the common ancestor of p and q in the final case is easy. When `commonAncestor(n.left, p, q)` and `commonAncestor(n.right, p, q)` both return non-`null` values (indicating that p and q were found in different subtrees), then n will be the common ancestor.

The code below offers an initial solution, but it has a bug. Can you find it?

```
/* The below code has a bug. */
TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
   if (root == null) return null;
   if (root == p && root == q) return root;

   TreeNode x = commonAncestor(root.left, p, q);
   if (x != null && x != p && x != q) { // Already found ancestor
      return x;
   }

   TreeNode y = commonAncestor(root.right, p, q);
   if (y != null && y != p && y != q) { // Already found ancestor
      return y;
   }

   if (x != null && y != null) { // p and q found in diff. subtrees
      return root; // This is the common ancestor
   } else if (root == p || root == q) {
      return root;
   } else {
      return x == null ? y : x; /* return the non-null value */
   }
}
```

The problem with this code occurs in the case where a node is not contained in the tree. For example, look at the following tree:



Suppose we call `commonAncestor(node 3, node 5, node 7)`. Of course, node 7 does not exist—and that's where the issue will come in. The calling order looks like:

```
commonAnc(node 3, node 5, node 7)              // --> 5
   calls commonAnc(node 1, node 5, node 7)     // --> null
```

```
  calls commonAnc(node 5, node 5, node 7)      // --> 5
    calls commonAnc(node 8, node 5, node 7)  // --> null
```

In other words, when we call `commonAncestor` on the right subtree, the code will return `node 5`, just as it should. The problem is that, in finding the common ancestor of p and q, the calling function can't distinguish between the two cases:

- Case 1: p is a child of q (or, q is a child of p)
- Case 2: p is in the tree and q is not (or, q is in the tree and p is not)

In either of these cases, `commonAncestor` will return p. In the first case, this is the correct return value, but in the second case, the return value should be `null`.

We somehow need to distinguish between these two cases, and this is what the code below does. This code solves the problem by returning two values: the node itself and a flag indicating whether this node is actually the common ancestor.

```
class Result {
  public TreeNode node;
  public boolean isAncestor;
  public Result(TreeNode n, boolean isAnc) {
    node = n;
    isAncestor = isAnc;
  }
}

TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
  Result r = commonAncestorHelper(root, p, q);
  if (r.isAncestor) {
    return r.node;
  }
  return null;
}

Result commonAncHelper(TreeNode root, TreeNode p, TreeNode q) {
  if (root == null) return new Result(null, false);

  if (root == p && root == q) {
    return new Result(root, true);
  }

  Result rx = commonAncHelper(root.left, p, q);
  if (rx.isAncestor) { // Found common ancestor
    return rx;
  }

  Result ry = commonAncHelper(root.right, p, q);
  if (ry.isAncestor) { // Found common ancestor
    return ry;
  }

  if (rx.node != null && ry.node != null) {
    return new Result(root, true); // This is the common ancestor
  } else if (root == p || root == q) {
    /* If we're currently at p or q, and we also found one of those nodes in a
     * subtree, then this is truly an ancestor and the flag should be true. */
    boolean isAncestor = rx.node != null || ry.node != null;
```

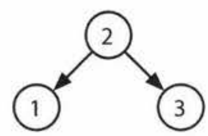
```
        return new Result(root, isAncestor);
    } else {
        return new Result(rx.node!=null ? rx.node : ry.node, false);
    }
}
```

Of course, as this issue only comes up when p or q is not actually in the tree, an alternative solution would be to first search through the entire tree to make sure that both nodes exist.

4.9 **BST Sequences:** A binary search tree was created by traversing through an array from left to right and inserting each element. Given a binary search tree with distinct elements, print all possible arrays that could have led to this tree.

EXAMPLE

Input:

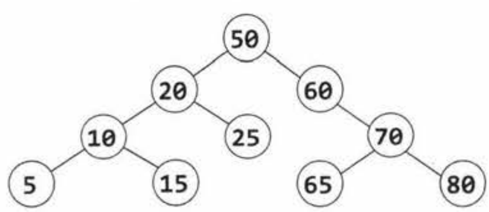


Output: `{2, 1, 3}`, `{2, 3, 1}`

pg 110

SOLUTION

It's useful to kick off this question with a good example.



We should also think about the ordering of items in a binary search tree. Given a node, all nodes on its left must be less than all nodes on its right. Once we reach a place without a node, we insert the new value there.

What this means is that the very first element in our array must have been a 50 in order to create the above tree. If it were anything else, then that value would have been the root instead.

What else can we say? Some people jump to the conclusion that everything on the left must have been inserted before elements on the right, but that's not actually true. In fact, the reverse is true: the order of the left or right items doesn't matter.

Once the 50 is inserted, all items less than 50 will be routed to the left and all items greater than 50 will be routed to the right. The 60 or the 20 could be inserted first, and it wouldn't matter.

Let's think about this problem recursively. If we had all arrays that could have created the subtree rooted at 20 (call this `arraySet20`), and all arrays that could have created the subtree rooted at 60 (call this `arraySet60`), how would that give us the full answer? We could just "weave" each array from `arraySet20` with each array from `arraySet60`—and then prepend each array with a 50.

Here's what we mean by weaving. We are merging two arrays in all possible ways, while keeping the elements within each array in the same relative order.

```
array1: {1, 2}
array2: {3, 4}
weaved: {1, 2, 3, 4}, {1, 3, 2, 4}, {1, 3, 4, 2},
        {3, 1, 2, 4}, {3, 1, 4, 2}, {3, 4, 1, 2}
```

Note that, as long as there aren't any duplicates in the original array sets, we won't have to worry that weaving will create duplicates.

The last piece to talk about here is how the weaving works. Let's think recursively about how to weave `{1, 2, 3}` and `{4, 5, 6}`. What are the subproblems?

- Prepend a 1 to all weaves of `{2, 3}` and `{4, 5, 6}`.
- Prepend a 4 to all weaves of `{1, 2, 3}` and `{5, 6}`.

To implement this, we'll store each as linked lists. This will make it easy to add and remove elements. When we recurse, we'll push the prefixed elements down the recursion. When `first` or `second` are empty, we add the remainder to `prefix` and store the result.

It works something like this:

```
weave(first, second, prefix):
    weave({1, 2}, {3, 4}, {})
        weave({2}, {3, 4}, {1})
            weave({}, {3, 4}, {1, 2})
                {1, 2, 3, 4}
            weave({2}, {4}, {1, 3})
                weave({}, {4}, {1, 3, 2})
                    {1, 3, 2, 4}
                weave({2}, {}, {1, 3, 4})
                    {1, 3, 4, 2}
        weave({1, 2}, {4}, {3})
            weave({2}, {4}, {3, 1})
                weave({}, {4}, {3, 1, 2})
                    {3, 1, 2, 4}
                weave({2}, {}, {3, 1, 4})
                    {3, 1, 4, 2}
            weave({1, 2}, {}, {3, 4})
                {3, 4, 1, 2}
```

Now, let's think through the implementation of removing, say, 1 from `{1, 2}` and recursing. We need to be careful about modifying this list, since a later recursive call (e.g., `weave({1, 2}, {4}, {3})`) might need the 1 still in `{1, 2}`.

We could clone the list when we recurse, so that we only modify the recursive calls. Or, we could modify the list, but then "revert" the changes after we're done with recursing.

We've chosen to implement it the latter way. Since we're keeping the same reference to `first`, `second`, and `prefix` the entire way down the recursive call stack, then we'll need to clone `prefix` just before we store the complete result.

```
ArrayList<LinkedList<Integer>> allSequences(TreeNode node) {
  ArrayList<LinkedList<Integer>> result = new ArrayList<LinkedList<Integer>>();

  if (node == null) {
    result.add(new LinkedList<Integer>());
    return result;
```

```java
  }

  LinkedList<Integer> prefix = new LinkedList<Integer>();
  prefix.add(node.data);

  /* Recurse on left and right subtrees. */
  ArrayList<LinkedList<Integer>> leftSeq = allSequences(node.left);
  ArrayList<LinkedList<Integer>> rightSeq = allSequences(node.right);

  /* Weave together each list from the left and right sides. */
  for (LinkedList<Integer> left : leftSeq) {
    for (LinkedList<Integer> right : rightSeq) {
      ArrayList<LinkedList<Integer>> weaved =
        new ArrayList<LinkedList<Integer>>();
      weaveLists(left, right, weaved, prefix);
      result.addAll(weaved);
    }
  }
  return result;
}

/* Weave lists together in all possible ways. This algorithm works by removing the
 * head from one list, recursing, and then doing the same thing with the other
 * list. */
void weaveLists(LinkedList<Integer> first, LinkedList<Integer> second,
      ArrayList<LinkedList<Integer>> results, LinkedList<Integer> prefix) {
  /* One list is empty. Add remainder to [a cloned] prefix and store result. */
  if (first.size() == 0 || second.size() == 0) {
    LinkedList<Integer> result = (LinkedList<Integer>) prefix.clone();
    result.addAll(first);
    result.addAll(second);
    results.add(result);
    return;
  }

  /* Recurse with head of first added to the prefix. Removing the head will damage
   * first, so we'll need to put it back where we found it afterwards. */
  int headFirst = first.removeFirst();
  prefix.addLast(headFirst);
  weaveLists(first, second, results, prefix);
  prefix.removeLast();
  first.addFirst(headFirst);

  /* Do the same thing with second, damaging and then restoring the list.*/
  int headSecond = second.removeFirst();
  prefix.addLast(headSecond);
  weaveLists(first, second, results, prefix);
  prefix.removeLast();
  second.addFirst(headSecond);
}
```

Some people struggle with this problem because there are two different recursive algorithms that must be designed and implemented. They get confused with how the algorithms should interact with each other and they try to juggle both in their heads.

If this sounds like you, try this: *trust and focus*. *Trust* that one method does the right thing when implementing an independent method, and *focus* on the one thing that this independent method needs to do.

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.

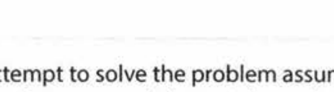
The Simple Approach

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.

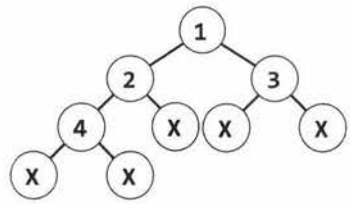


There's a simple fix though. We can store NULL nodes in the pre-order traversal string as a special character, like an 'X'. (We'll assume that the binary trees contain only integers.) The left tree would have the traversal `{3, 4, X}` and the right tree will have the traversal `{3, X, 4}`.

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.

To see this, consider reconstructing a tree from its pre-order traversal (with NULL nodes indicated). For example: 1, 2, 4, X, X, X, 3, X, X.

The root is 1, and its left node, 2, follows it. 2.left must be 4. 4 must have two NULL nodes (since it is followed by two Xs). 4 is complete, so we move back up to its parent, 2. 2.right is another X (NULL). 1's left subtree is now complete, so we move to 1's right child. We place a 3 with two NULL children there. The tree is now complete.



This whole process was deterministic, as it will be on any other tree. A pre-order traversal always starts at the root and, from there, the path we take is entirely defined by the traversal. Therefore, two trees are identical if they have the same pre-order traversal.

Now consider the subtree problem. If T2's pre-order traversal is a substring of T1's pre-order traversal, then T2's root element must be found in T1. If we do a pre-order traversal from this element in T1, we will follow an identical path to T2's traversal. Therefore, T2 is a subtree of T1.

Implementing this is quite straightforward. We just need to construct and compare the pre-order traversals.

```
boolean containsTree(TreeNode t1, TreeNode t2) {
  StringBuilder string1 = new StringBuilder();
  StringBuilder string2 = new StringBuilder();

  getOrderString(t1, string1);
  getOrderString(t2, string2);

  return string1.indexOf(string2.toString()) != -1;
}

void getOrderString(TreeNode node, StringBuilder sb) {
  if (node == null) {
    sb.append("X");             // Add null indicator
    return;
  }
  sb.append(node.data + " ");     // Add root
  getOrderString(node.left, sb);  // Add left
  getOrderString(node.right, sb); // Add right
}
```

This approach takes `O(n + m)` time and `O(n + m)` space, where n and `m` are the number of nodes in T1 and T2, respectively. Given millions of nodes, we might want to reduce the space complexity.

The Alternative Approach

An alternative approach is to search through the larger tree, T1. Each time a node in `T1` matches the root of T2, call `matchTree`. The `matchTree` method will compare the two subtrees to see if they are identical.

Analyzing the runtime is somewhat complex. A naive answer would be to say that it is `O(nm)` time, where n is the number of nodes in `T1` and `m` is the number of nodes in `T2`. While this is technically correct, a little more thought can produce a tighter bound.

We do not actually call `matchTree` on every node in T1. Rather, we call it k times, where k is the number of occurrences of T2's root in T1. The runtime is closer to `O(n + km)`.

In fact, even that overstates the runtime. Even if the root were identical, we exit `matchTree` when we find a difference between T1 and T2. We therefore probably do not actually look at m nodes on each call of `matchTree`.

The code below implements this algorithm.

```
boolean containsTree(TreeNode t1, TreeNode t2) {
  if (t2 == null) return true; // The empty tree is always a subtree
  return subTree(t1, t2);
}

boolean subTree(TreeNode r1, TreeNode r2) {
  if (r1 == null) {
     return false; // big tree empty & subtree still not found.
  } else if (r1.data == r2.data && matchTree(r1, r2)) {
     return true;
  }
  return subTree(r1.left, r2) || subTree(r1.right, r2);
}

boolean matchTree(TreeNode r1, TreeNode r2) {
  if (r1 == null && r2 == null) {
     return true; // nothing left in the subtree
  } else if (r1 == null || r2 == null) {
     return false; // exactly tree is empty, therefore trees don't match
  } else if (r1.data != r2.data) {
     return false;  // data doesn't match
  } else {
     return matchTree(r1.left, r2.left) && matchTree(r1.right, r2.right);
  }
}
```

When might the simple solution be better, and when might the alternative approach be better? This is a great conversation to have with your interviewer. Here are a few thoughts on that matter:

1. The simple solution takes `O(n + m)` memory. The alternative solution takes `O(log(n) + log(m))` memory. Remember: memory usage can be a very big deal when it comes to scalability.
2. The simple solution is `O(n + m)` time and the alternative solution has a worst case time of `O(nm)`. However, the worst case time can be deceiving; we need to look deeper than that.
3. A slightly tighter bound on the runtime, as explained earlier, is `O(n + km)`, where k is the number of occurrences of T2's root in T1. Let's suppose the node data for T1 and T2 were random numbers picked between 0 and p. The value of k would be approximately $\frac{n}{p}$. Why? Because each of n nodes in T1 has a $\frac{1}{p}$ chance of equaling the root, so approximately $\frac{n}{p}$ nodes in T1 should equal `T2.root`. So, let's say `p = 1000`, `n = 1000000` and `m = 100`. We would do somewhere around 1,100,000 node checks ($1100000 = 1000000 + \frac{100 * 1000000}{1000}$).
4. More complex mathematics and assumptions could get us an even tighter bound. We assumed in #3 above that if we call `matchTree`, we would end up traversing all m nodes of T2. It's far more likely, though, that we will find a difference very early on in the tree and will then exit early.

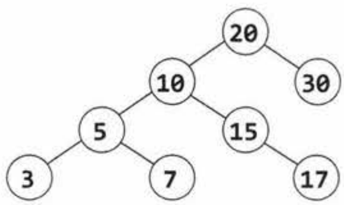
In summary, the alternative approach is certainly more optimal in terms of space and is likely more optimal in terms of time as well. It all depends on what assumptions you make and whether you prioritize reducing the average case runtime at the expense of the worst case runtime. This is an excellent point to make to your interviewer.

4.11 Random Node: You are implementing a binary search tree class from scratch, which, in addition to `insert`, `find`, and `delete`, has a method `getRandomNode()` which returns a random node from the tree. All nodes should be equally likely to be chosen. Design and implement an algorithm for `getRandomNode`, and explain how you would implement the rest of the methods.

pg 111

SOLUTION

Let's draw an example.



We're going to explore many solutions until we get to an optimal one that works.

One thing we should realize here is that the question was phrased in a very interesting way. The interviewer did not simply say, "Design an algorithm to return a random node from a binary tree." We were told that this is a class that we're building from scratch. There is a reason the question was phrased that way. We probably need access to some part of the internals of the data structure.

Option #1 [Slow & Working]

One solution is to copy all the nodes to an array and return a random element in the array. This solution will take `O(N)` time and `O(N)` space, where `N` is the number of nodes in the tree.

We can guess our interviewer is probably looking for something more optimal, since this is a little too straightforward (and should make us wonder why the interviewer gave us a binary tree, since we don't need that information).

We should keep in mind as we develop this solution that we probably need to know something about the internals of the tree. Otherwise, the question probably wouldn't specify that we're developing the tree class from scratch.

Option #2 [Slow & Working]

Returning to our original solution of copying the nodes to an array, we can explore a solution where we maintain an array at all times that lists all the nodes in the tree. The problem is that we'll need to remove nodes from this array as we delete them from the tree, and that will take `O(N)` time.

Option #3 [Slow & Working]

We could label all the nodes with an index from 1 to `N` and label them in binary search tree order (that is, according to its inorder traversal). Then, when we call `getRandomNode`, we generate a random index between 1 and `N`. If we apply the label correctly, we can use a binary search tree search to find this index.

However, this leads to a similar issue as earlier solutions. When we insert a node or a delete a node, all of the indices might need to be updated. This can take O(N) time.

Option #4 [Fast & Not Working]

What if we knew the depth of the tree? (Since we're building our own class, we can ensure that we know this. It's an easy enough piece of data to track.)

We could pick a random depth, and then traverse left/right randomly until we go to that depth. This wouldn't actually ensure that all nodes are equally likely to be chosen though.

First, the tree doesn't necessarily have an equal number of nodes at each level. This means that nodes on levels with fewer nodes might be more likely to be chosen than nodes on a level with more nodes.

Second, the random path we take might end up terminating before we get to the desired level. Then what? We could just return the last node we find, but that would mean unequal probabilities at each node.

Option #5 [Fast & Not Working]

We could try just a simple approach: traverse randomly down the tree. At each node:

- With $1/3$ odds, we return the current node.
- With $1/3$ odds, we traverse left.
- With $1/3$ odds, we traverse right.

This solution, like some of the others, does not distribute the probabilities evenly across the nodes. The root has a $1/3$ probability of being selected—the same as all the nodes in the left put together.

Option #6 [Fast & Working]

Rather than just continuing to brainstorm new solutions, let's see if we can fix some of the issues in the previous solutions. To do so, we must diagnose—deeply—the root problem in a solution.

Let's look at Option #5. It fails because the probabilities aren't evenly distributed across the options. Can we fix that while keeping the basic algorithm the same?

We can start with the root. With what probability should we return the root? Since we have N nodes, we must return the root node with $1/N$ probability. (In fact, we must return each node with $1/N$ probability. After all, we have N nodes and each must have equal probability. The total must be 1 (100%), therefore each must have $1/N$ probability.)

We've resolved the issue with the root. Now what about the rest of the problem? With what probability should we traverse left versus right? It's not 50/50. Even in a balanced tree, the number of nodes on each side might not be equal. If we have more nodes on the left than the right, then we need to go left more often.

One way to think about it is that the odds of picking something—anything—from the left must be the sum of each individual probability. Since each node must have probability $1/N$, the odds of picking something from the left must have probability `LEFT_SIZE * ` $1/N$. This should therefore be the odds of going left.

Likewise, the odds of going right should be `RIGHT_SIZE * ` $1/N$.

This means that each node must know the size of the nodes on the left and the size of the nodes on the right. Fortunately, our interviewer has told us that we're building this tree class from scratch. It's easy to keep track of this size information on inserts and deletes. We can just store a `size` variable in each node. Increment `size` on inserts and decrement it on deletes.

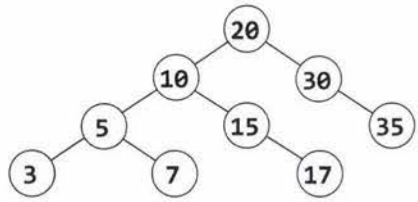
```
class TreeNode {
  private int data;
  public TreeNode left;
  public TreeNode right;
  private int size = 0;

  public TreeNode(int d) {
    data = d;
    size = 1;
  }

  public TreeNode getRandomNode() {
    int leftSize = left == null ? 0 : left.size();
    Random random = new Random();
    int index = random.nextInt(size);
    if (index < leftSize) {
      return left.getRandomNode();
    } else if (index == leftSize) {
      return this;
    } else {
      return right.getRandomNode();
    }
  }

  public void insertInOrder(int d) {
    if (d <= data) {
      if (left == null) {
        left = new TreeNode(d);
      } else {
        left.insertInOrder(d);
      }
    } else {
      if (right == null) {
        right = new TreeNode(d);
      } else {
        right.insertInOrder(d);
      }
    }
    size++;
  }

  public int size() { return size; }
  public int data() { return data; }

  public TreeNode find(int d) {
    if (d == data) {
      return this;
    } else if (d <= data) {
      return left != null ? left.find(d) : null;
    } else if (d > data) {
      return right != null ? right.find(d) : null;
    }
    return null;
  }
}
```

In a balanced tree, this algorithm will be `O(log N)`, where N is the number of nodes.

Option #7 [Fast & Working]

Random number calls can be expensive. If we'd like, we can reduce the number of random number calls substantially.

Imagine we called `getRandomNode` on the tree below, and then traversed left.



We traversed left because we picked a number between 0 and 5 (inclusive). When we traverse left, we again pick a random number between 0 and 5. Why re-pick? The first number will work just fine.

But what if we went right instead? We have a number between 7 and 8 (inclusive) but we would need a number between 0 and 1 (inclusive). That's easy to fix: just subtract out `LEFT_SIZE + 1`.

Another way to think about what we're doing is that the initial random number call indicates which node (`i`) to return, and then we're locating the `i`th node in an in-order traversal. Subtracting `LEFT_SIZE + 1` from `i` reflects that, when we go right, we skip over `LEFT_SIZE + 1` nodes in the in-order traversal.

```
class Tree {
  TreeNode root = null;

  public int size() { return root == null ? 0 : root.size(); }

  public TreeNode getRandomNode() {
    if (root == null) return null;

    Random random = new Random();
    int i = random.nextInt(size());
    return root.getIthNode(i);
  }

  public void insertInOrder(int value) {
    if (root == null) {
      root = new TreeNode(value);
    } else {
      root.insertInOrder(value);
    }
  }
}

class TreeNode {
  /* constructor and variables are the same. */

  public TreeNode getIthNode(int i) {
    int leftSize = left == null ? 0 : left.size();
    if (i < leftSize) {
      return left.getIthNode(i);
    } else if (i == leftSize) {
      return this;
    } else {
```

```
            /* Skipping over leftSize + 1 nodes, so subtract them. */
            return right.getIthNode(i - (leftSize + 1));
        }
    }

    public void insertInOrder(int d) { /* same */ }
    public int size() { return size; }
    public TreeNode find(int d) { /* same */ }
}
```

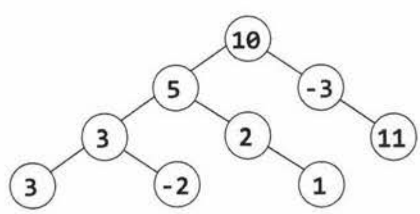
Like the previous algorithm, this algorithm takes `O(log N)` time in a balanced tree. We can also describe the runtime as `O(D)`, where D is the max depth of the tree. Note that `O(D)` is an accurate description of the runtime whether the tree is balanced or not.

4.12 **Paths with Sum:** You are given a binary tree in which each node contains an integer value (which might be positive or negative). Design an algorithm to count the number of paths that sum to a given value. The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).

pg 111

SOLUTION

Let's pick a potential sum—say, 8—and then draw a binary tree based on this. This tree intentionally has a number of paths with this sum.



One option is the brute force approach.

Solution #1: Brute Force

In the brute force approach, we just look at all possible paths. To do this, we traverse to each node. At each node, we recursively try all paths downwards, tracking the sum as we go. As soon as we hit our target sum, we increment the total.

```
int countPathsWithSum(TreeNode root, int targetSum) {
    if (root == null) return 0;

    /* Count paths with sum starting from the root. */
    int pathsFromRoot = countPathsWithSumFromNode(root, targetSum, 0);

    /* Try the nodes on the left and right. */
    int pathsOnLeft = countPathsWithSum(root.left, targetSum);
    int pathsOnRight = countPathsWithSum(root.right, targetSum);

    return pathsFromRoot + pathsOnLeft + pathsOnRight;
}

/* Returns the number of paths with this sum starting from this node. */
```

```
int countPathsWithSumFromNode(TreeNode node, int targetSum, int currentSum) {
   if (node == null) return 0;

   currentSum += node.data;

   int totalPaths = 0;
   if (currentSum == targetSum) { // Found a path from the root
      totalPaths++;
   }

   totalPaths += countPathsWithSumFromNode(node.left, targetSum, currentSum);
   totalPaths += countPathsWithSumFromNode(node.right, targetSum, currentSum);
   return totalPaths;
}
```

What is the time complexity of this algorithm?

Consider that node at depth d will be "touched" (via `countPathsWithSumFromNode`) by d nodes above it.

In a balanced binary tree, d will be no more than approximately `log N`. Therefore, we know that with N nodes in the tree, `countPathsWithSumFromNode` will be called `O(N log N)` times. The runtime is O(N log N).

We can also approach this from the other direction. At the root node, we traverse to all `N - 1` nodes beneath it (via `countPathsWithSumFromNode`). At the second level (where there are two nodes), we traverse to `N - 3` nodes. At the third level (where there are four nodes, plus three above those), we traverse to `N - 7` nodes. Following this pattern, the total work is roughly:

```
(N - 1) + (N - 3) + (N - 7) + (N - 15) + (N - 31) + ... + (N - N)
```

To simplify this, notice that the left side of each term is always N and the right side is one less than a power of two. The number of terms is the depth of the tree, which is `O(log N)`. For the right side, we can ignore the fact that it's one less than a power of two. Therefore, we really have this:

```
O(N * [number of terms] - [sum of powers of two from 1 through N])
O(N log N - N)
O(N log N)
```

If the value of the sum of powers of two from 1 through N isn't obvious to you, think about what the powers of two look like in binary:

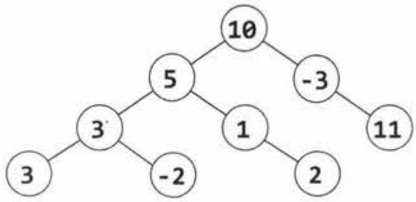
```
  0001
+ 0010
+ 0100
+ 1000
= 1111
```

Therefore, the runtime is `O(N log N)` in a balanced tree.

In an unbalanced tree, the runtime could be much worse. Consider a tree that is just a straight line down. At the root, we traverse to `N - 1` nodes. At the next level (with just a single node), we traverse to `N - 2` nodes. At the third level, we traverse to `N - 3` nodes, and so on. This leads us to the sum of numbers between 1 and N, which is $O(N^2)$.

Solution #2: Optimized

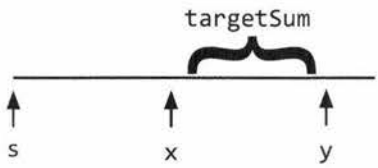
In analyzing the last solution, we may realize that we repeat some work. For a path such as `10 -> 5 -> 3 -> -2`, we traverse this path (or parts of it) repeatedly. We do it when we start with node 10, then when we go to node 5 (looking at 5, then 3, then -2), then when we go to node 3, and then finally when we go to node -2. Ideally, we'd like to reuse this work.



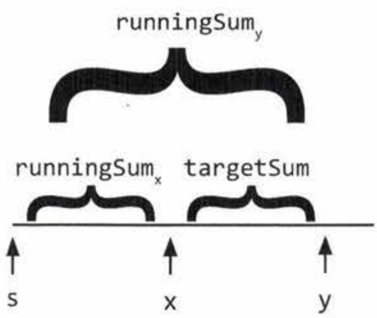
Let's isolate a given path and treat it as just an array. Consider a (hypothetical, extended) path like:

```
10 -> 5 -> 1 -> 2 -> -1 -> -1 -> 7 -> 1 -> 2
```

What we're really saying then is: How many contiguous subsequences in this array sum to a target sum such as 8? In other words, for each y, we're trying to find the x values below. (Or, more accurately, the number of x values below.)



If each value knows its running sum (the sum of values from s through itself), then we can find this pretty easily. We just need to leverage this simple equation: $runningSum_x = runningSum_y - targetSum$. We then look for the values of x where this is true.



Since we're just looking for the number of paths, we can use a hash table. As we iterate through the array, build a hash table that maps from a `runningSum` to the number of times we've seen that sum. Then, for each y, look up $runningSum_y$ - `targetSum` in the hash table. The value in the hash table will tell you the number of paths with sum `targetSum` that end at y.

For example:

```
index:  0     1     2     3      4      5     6     7     8
value:  10 -> 5 -> 1 -> 2 -> -1 -> -1 -> 7 -> 1 -> 2
sum:    10    15    16    18     17     16    23    24    26
```

The value of $runningSum_7$ is 24. If `targetSum` is 8, then we'd look up 16 in the hash table. This would have a value of 2 (originating from index 2 and index 5). As we can see above, indexes 3 through 7 and indexes 6 through 7 have sums of 8.

Now that we've settled the algorithm for an array, let's review this on a tree. We take a similar approach.

We traverse through the tree using depth-first search. As we visit each node:

1. Track its `runningSum`. We'll take this in as a parameter and immediately increment it by `node.value`.
2. Look up `runningSum - targetSum` in the hash table. The value there indicates the total number. Set `totalPaths` to this value.
3. If `runningSum == targetSum`, then there's one additional path that starts at the root. Increment `totalPaths`.
4. Add `runningSum` to the hash table (incrementing the value if it's already there).
5. Recurse left and right, counting the number of paths with sum `targetSum`.
6. After we're done recursing left and right, decrement the value of `runningSum` in the hash table. This is essentially backing out of our work; it reverses the changes to the hash table so that other nodes don't use it (since we're now done with `node`).

Despite the complexity of deriving this algorithm, the code to implement this is relatively simple.

```
int countPathsWithSum(TreeNode root, int targetSum) {
  return countPathsWithSum(root, targetSum, 0, new HashMap<Integer, Integer>());
}

int countPathsWithSum(TreeNode node, int targetSum, int runningSum,
                      HashMap<Integer, Integer> pathCount) {
  if (node == null) return 0; // Base case

  /* Count paths with sum ending at the current node. */
  runningSum += node.data;
  int sum = runningSum - targetSum;
  int totalPaths = pathCount.getOrDefault(sum, 0);

  /* If runningSum equals targetSum, then one additional path starts at root.
   * Add in this path.*/
  if (runningSum == targetSum) {
    totalPaths++;
  }

  /* Increment pathCount, recurse, then decrement pathCount. */
  incrementHashTable(pathCount, runningSum, 1); // Increment pathCount
  totalPaths += countPathsWithSum(node.left, targetSum, runningSum, pathCount);
  totalPaths += countPathsWithSum(node.right, targetSum, runningSum, pathCount);
  incrementHashTable(pathCount, runningSum, -1); // Decrement pathCount

  return totalPaths;
}

void incrementHashTable(HashMap<Integer, Integer> hashTable, int key, int delta) {
  int newCount = hashTable.getOrDefault(key, 0) + delta;
  if (newCount == 0) { // Remove when zero to reduce space usage
    hashTable.remove(key);
  } else {
    hashTable.put(key, newCount);
  }
}
```

The runtime for this algorithm is `O(N)`, where `N` is the number of nodes in the tree. We know it is `O(N)` because we travel to each node just once, doing `O(1)` work each time. In a balanced tree, the space complexity is `O(log N)` due to the hash table. The space complexity can grow to `O(n)` in an unbalanced tree.

5

Solutions to Bit Manipulation

5.1 **Insertion:** You are given two 32-bit numbers, N and M, and two bit positions, `i` and `j`. Write a method to insert M into N such that M starts at bit `j` and ends at bit `i`. You can assume that the bits `j` through `i` have enough space to fit all of M. That is, if M = `10011`, you can assume that there are at least 5 bits between `j` and `i`. You would not, for example, have `j` = 3 and `i` = 2, because M could not fully fit between bit 3 and bit 2.

EXAMPLE

Input: `N = 10000000000, M = 10011, i = 2, j = 6`

Output: `N = 10001001100`

pg 115

SOLUTION

This problem can be approached in three key steps:

1. Clear the bits `j` through `i` in N
2. Shift M so that it lines up with bits `j` through `i`
3. Merge M and N.

The trickiest part is Step 1. How do we clear the bits in N? We can do this with a mask. This mask will have all 1s, except for 0s in the bits `j` through `i`. We create this mask by creating the left half of the mask first, and then the right half.

```
int updateBits(int n, int m, int i, int j) {
   /* Create a mask to clear bits i through j in n. EXAMPLE: i = 2, j = 4. Result
    * should be 11100011. For simplicity, we'll use just 8 bits for the example. */
   int allOnes = ~0; // will equal sequence of all 1s

   // 1s before position j, then 0s. left = 11100000
   int left = allOnes << (j + 1);

   // 1's after position i. right = 00000011
   int right = ((1 << i) - 1);

   // All 1s, except for 0s between i and j. mask = 11100011
   int mask = left | right;

   /* Clear bits j through i then put m in there */
   int n_cleared = n & mask; // Clear bits j through i.
   int m_shifted = m << i; // Move m into correct position.
```

```

   return n_cleared | m_shifted; // OR them, and we're done!
}
```

In a problem like this (and many bit manipulation problems), you should make sure to thoroughly test your code. It's extremely easy to wind up with off-by-one errors.

5.2 **Binary to String:** Given a real number between 0 and 1 (e.g., 0.72) that is passed in as a double, print the binary representation. If the number cannot be represented accurately in binary with at most 32 characters, print "ERROR."

pg 116

SOLUTION

NOTE: When otherwise ambiguous, we'll use the subscripts x_2 and x_{10} to indicate whether x is in base 2 or base 10.

First, let's start off by asking ourselves what a non-integer number in binary looks like. By analogy to a decimal number, the binary number 0.101_2 would look like:

$$0.101_2 = 1 * \frac{1}{2^1} + 0 * \frac{1}{2^2} + 1 * \frac{1}{2^3}.$$

To print the decimal part, we can multiply by 2 and check if 2n is greater than or equal to 1. This is essentially "shifting" the fractional sum. That is:

$$\begin{aligned} r &= 2_{10} * n \\ &= 2_{10} * 0.101_2 \\ &= 1 * \frac{1}{2^0} + 0 * \frac{1}{2^1} + 1 * \frac{1}{2^2} \\ &= 1.01_2 \end{aligned}$$

If `r >= 1`, then we know that n had a 1 right after the decimal point. By doing this continuously, we can check every digit.

```
String printBinary(double num) {
   if (num >= 1 || num <= 0) {
      return "ERROR";
   }

   StringBuilder binary = new StringBuilder();
   binary.append(".");
   while (num > 0) {
      /* Setting a limit on length: 32 characters */
      if (binary.length() >= 32) {
         return "ERROR";
      }

      double r = num * 2;
      if (r >= 1) {
         binary.append(1);
         num = r - 1;
      } else {
         binary.append(0);
         num = r;
      }
   }
   return binary.toString();
}
```

Alternatively, rather than multiplying the number by two and comparing it to 1, we can compare the number to .5, then .25, and so on. The code below demonstrates this approach.

```
String printBinary2(double num) {
   if (num >= 1 || num <= 0) {
      return “ERROR”;
   }

   StringBuilder binary = new StringBuilder();
   double frac = 0.5;
   binary.append(“.”);
   while (num > 0) {
      /* Setting a limit on length: 32 characters */
      if (binary.length() > 32) {
         return “ERROR”;
      }
      if (num >= frac) {
         binary.append(1);
         num -= frac;
      } else {
         binary.append(0);
      }
      frac /= 2;
   }
   return binary.toString();
}
```

Both approaches are equally good; choose the one you feel most comfortable with.

Either way, you should make sure to prepare thorough test cases for this problem—and to actually run through them in your interview.

5.3 **Flip Bit to Win:** You have an integer and you can flip exactly one bit from a 0 to a 1. Write code to find the length of the longest sequence of 1s you could create.

EXAMPLE

Input: 1775 (or: 11011101111)

Output: 8

pg 116

SOLUTION

We can think about each integer as being an alternating sequence of 0s and 1s. Whenever a 0s sequence has length one, we can potentially merge the adjacent 1s sequences.

Brute Force

One approach is to convert an integer into an array that reflects the lengths of the 0s and 1s sequences. For example, 11011101111 would be (reading from right to left) $[0_0, 4_1, 1_0, 3_1, 1_0, 2_1, 21_0]$. The subscript reflects whether the integer corresponds to a 0s sequence or a 1s sequence, but the actual solution doesn't need this. It's a strictly alternating sequence, always starting with the 0s sequence.

Once we have this, we just walk through the array. At each 0s sequence, then we consider merging the adjacent 1s sequences if the 0s sequence has length 1.

```
int longestSequence(int n) {
```

```
    if (n == -1) return Integer.BYTES * 8;
    ArrayList<Integer> sequences = getAlternatingSequences(n);
    return findLongestSequence(sequences);
}

/* Return a list of the sizes of the sequences. The sequence starts off with the
   number of 0s (which might be 0) and then alternates with the counts of each
   value.*/
ArrayList<Integer> getAlternatingSequences(int n) {
    ArrayList<Integer> sequences = new ArrayList<Integer>();

    int searchingFor = 0;
    int counter = 0;

    for (int i = 0; i < Integer.BYTES * 8; i++) {
        if ((n & 1) != searchingFor) {
            sequences.add(counter);
            searchingFor = n & 1; // Flip 1 to 0 or 0 to 1
            counter = 0;
        }
        counter++;
        n >>>= 1;
    }
    sequences.add(counter);

    return sequences;
}

/* Given the lengths of alternating sequences of 0s and 1s, find the longest one
 * we can build. */
int findLongestSequence(ArrayList<Integer> seq) {
    int maxSeq = 1;

    for (int i = 0; i < seq.size(); i += 2) {
        int zerosSeq = seq.get(i);
        int onesSeqRight = i - 1 >= 0 ? seq.get(i - 1) : 0;
        int onesSeqLeft = i + 1 < seq.size() ? seq.get(i + 1) : 0;

        int thisSeq = 0;
        if (zerosSeq == 1) { // Can merge
            thisSeq = onesSeqLeft + 1 + onesSeqRight;
        } if (zerosSeq > 1) { // Just add a zero to either side
            thisSeq = 1 + Math.max(onesSeqRight, onesSeqLeft);
        } else if (zerosSeq == 0) { // No zero, but take either side
            thisSeq = Math.max(onesSeqRight, onesSeqLeft);
        }
        maxSeq = Math.max(thisSeq, maxSeq);
    }

    return maxSeq;
}
```

This is pretty good. It's O(b) time and O(b) memory, where b is the length of the sequence.

> Be careful with how you express the runtime. For example, if you say the runtime is `O(n)`, what is n? It is not correct to say that this algorithm is O(value of the integer). This algorithm is O(number of bits). For this reason, when you have potential ambiguity in what n might mean, it's best just to not use n. Then neither you nor your interviewer will be confused. Pick a different variable name. We used "b", for the number of bits. Something logical works well.

Can we do better? Recall the concept of Best Conceivable Runtime. The B.C.R. for this algorithm is `O(b)` (since we'll always have to read through the sequence), so we know we can't optimize the time. We can, however, reduce the memory usage.

Optimal Algorithm

To reduce the space usage, note that we don't need to hang on to the length of each sequence the entire time. We only need it long enough to compare each 1s sequence to the immediately preceding 1s sequence.

Therefore, we can just walk through the integer doing this, tracking the current 1s sequence length and the previous 1s sequence length. When we see a zero, update `previousLength`:

- If the next bit is a 1, `previousLength` should be set to `currentLength`.
- If the next bit is a 0, then we can't merge these sequences together. So, set `previousLength` to 0.

Update `maxLength` as we go.

```
int flipBit(int a) {
  /* If all 1s, this is already the longest sequence. */
  if (~a == 0) return Integer.BYTES * 8;

  int currentLength = 0;
  int previousLength = 0;
  int maxLength = 1; // We can always have a sequence of at least one 1
  while (a != 0) {
    if ((a & 1) == 1) { // Current bit is a 1
      currentLength++;
    } else if ((a & 1) == 0) { // Current bit is a 0
      /* Update to 0 (if next bit is 0) or currentLength (if next bit is 1). */
      previousLength = (a & 2) == 0 ? 0 : currentLength;
      currentLength = 0;
    }
    maxLength = Math.max(previousLength + currentLength + 1, maxLength);
    a >>>= 1;
  }
  return maxLength;
}
```

The runtime of this algorithm is still `O(b)`, but we use only `O(1)` additional memory.

5.4 **Next Number:** Given a positive integer, print the next smallest and the next largest number that have the same number of 1 bits in their binary representation.

pg 116

SOLUTION

There are a number of ways to approach this problem, including using brute force, using bit manipulation, and using clever arithmetic. Note that the arithmetic approach builds on the bit manipulation approach. You'll want to understand the bit manipulation approach before going on to the arithmetic one.

> The terminology can be confusing for this problem. We'll call `getNext` the bigger number and `getPrev` the smaller number.

The Brute Force Approach

An easy approach is simply brute force: count the number of 1s in n, and then increment (or decrement) until you find a number with the same number of 1s. Easy—but not terribly interesting. Can we do something a bit more optimal? Yes!

Let's start with the code for `getNext`, and then move on to `getPrev`.

Bit Manipulation Approach for Get Next Number

If we think about what the next number *should* be, we can observe the following. Given the number 13948, the binary representation looks like:

1	1	0	1	1	0	0	1	1	1	1	1	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

We want to make this number bigger (but not *too* big). We also need to keep the same number of ones.

Observation: Given a number n and two bit locations `i` and `j`, suppose we flip bit `i` from a `1` to a `0`, and bit `j` from a `0` to a `1`. If `i > j`, then n will have decreased. If `i < j`, then n will have increased.

We know the following:

1. If we flip a zero to a one, we must flip a one to a zero.
2. When we do that, the number will be bigger if and only if the zero-to-one bit was to the left of the one-to-zero bit.
3. We want to make the number bigger, but not unnecessarily bigger. Therefore, we need to flip the rightmost zero which has ones on the right of it.

To put this in a different way, we are flipping the rightmost non-trailing zero. That is, using the above example, the trailing zeros are in the `0`th and `1`st spot. The rightmost non-trailing zero is at bit 7. Let's call this position p.

Step 1: Flip rightmost non-trailing zero

1	1	0	1	1	0	1	1	1	1	1	1	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

With this change, we have increased the size of n. But, we also have one too many ones, and one too few zeros. We'll need to shrink the size of our number as much as possible while keeping that in mind.

We can shrink the number by rearranging all the bits to the right of bit p such that the `0`s are on the left and the `1`s are on the right. As we do this, we want to replace one of the 1s with a `0`.

A relatively easy way of doing this is to count how many ones are to the right of p, clear all the bits from 0 until p, and then add back in `c1-1` ones. Let `c1` be the number of ones to the right of p and `c0` be the number of zeros to the right of p.

Let's walk through this with an example.

Step 2: Clear bits to the right of p. From before, `c0 = 2`. `c1 = 5`. `p = 7`.

1	1	0	1	1	0	1	0	0	0	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

To clear these bits, we need to create a mask that is a sequence of ones, followed by p zeros. We can do this as follows:

```
a = 1 << p; // all zeros except for a 1 at position p.
b = a - 1;   // all zeros, followed by p ones.
mask = ~b;   // all ones, followed by p zeros.
n = n & mask; // clears rightmost p bits.
```

Or, more concisely, we do:

```
n &= ~((1 << p) - 1).
```

Step 3: Add in `c1 - 1` *ones.*

1	1	0	1	1	0	1	0	0	0	1	1	1	1
13	12	11	10	9	8	7	6	5	4	3	2	1	0

To insert `c1 - 1` ones on the right, we do the following:

```
a = 1 << (c1 - 1); // 0s with a 1 at position c1 - 1
b = a - 1;         // 0s with 1s at positions 0 through c1 - 1
n = n | b;         // inserts 1s at positions 0 through c1 - 1
```

Or, more concisely:

```
n |= (1 << (c1 - 1)) - 1;
```

We have now arrived at the smallest number bigger than n with the same number of ones.

The code for `getNext` is below.

```
int getNext(int n) {
  /* Compute c0 and c1 */
  int c = n;
  int c0 = 0;
  int c1 = 0;
  while (((c & 1) == 0) && (c != 0)) {
    c0++;
    c >>= 1;
  }

  while ((c & 1) == 1) {
    c1++;
    c >>= 1;
  }

  /* Error: if n == 11..1100...00, then there is no bigger number with the same
   * number of 1s. */
  if (c0 + c1 == 31 || c0 + c1 == 0) {
    return -1;
  }

  int p = c0 + c1; // position of rightmost non-trailing zero

  n |= (1 << p); // Flip rightmost non-trailing zero
  n &= ~((1 << p) - 1); // Clear all bits to the right of p
  n |= (1 << (c1 - 1)) - 1; // Insert (c1-1) ones on the right.
```

```
  return n;
}
```

Bit Manipulation Approach for Get Previous Number

To implement `getPrev`, we follow a very similar approach.

1. Compute `c0` and `c1`. Note that `c1` is the number of trailing ones, and `c0` is the size of the block of zeros immediately to the left of the trailing ones.
2. Flip the rightmost non-trailing one to a zero. This will be at position `p = c1 + c0`.
3. Clear all bits to the right of bit p.
4. Insert `c1 + 1` ones immediately to the right of position p.

Note that Step 2 sets bit p to a zero and Step 3 sets bits `0` through `p-1` to a zero. We can merge these steps.

Let's walk through this with an example.

Step 1: Initial Number. `p = 7. c1 = 2. c0 = 5.`

1	0	0	1	1	1	1	0	0	0	0	0	1	1
13	12	11	10	9	8	7	6	5	4	3	2	1	0

Steps 2 & 3: Clear bits `0` through p.

1	0	0	1	1	1	0	0	0	0	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

We can do this as follows:

```
int a = ~0;               // Sequence of 1s
int b = a << (p + 1);     // Sequence of 1s followed by p + 1 zeros.
n &= b;                   // Clears bits 0 through p.
```

Steps 4: Insert `c1 + 1` *ones immediately to the right of position* p.

1	0	0	1	1	1	0	1	1	1	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note that since `p = c1 + c0`, the `(c1 + 1)` ones will be followed by `(c0 - 1)` zeros.

We can do this as follows:

```
int a = 1 << (c1 + 1); // 0s with 1 at position (c1 + 1)
int b = a - 1;         // 0s followed by c1 + 1 ones
int c = b << (c0 - 1); // c1+1 ones followed by c0-1 zeros.
n |= c;
```

The code to implement this is below.

```
int getPrev(int n) {
  int temp = n;
  int c0 = 0;
  int c1 = 0;
  while (temp & 1 == 1) {
    c1++;
    temp >>= 1;
  }

```

```
    if (temp == 0) return -1;

    while (((temp & 1) == 0) && (temp != 0)) {
      c0++;
      temp >>= 1;
    }

    int p = c0 + c1; // position of rightmost non-trailing one
    n &= ((~0) << (p + 1)); // clears from bit p onwards

    int mask = (1 << (c1 + 1)) - 1; // Sequence of (c1+1) ones
    n |= mask << (c0 - 1);

    return n;
}
```

Arithmetic Approach to Get Next Number

If `c0` is the number of trailing zeros, `c1` is the size of the one block immediately following, and `p = c0 + c1`, we can word our solution from earlier as follows:

1. Set the pth bit to 1.
2. Set all bits following p to `0`.
3. Set bits `0` through `c1 - 2` to 1. This will be `c1 - 1` total bits.

A quick and dirty way to perform steps 1 and 2 is to set the trailing zeros to 1 (giving us p trailing ones), and then add 1. Adding one will flip all trailing ones, so we wind up with a 1 at bit p followed by p zeros. We can perform this arithmetically.

```
n += 2^c0 - 1;     // Sets trailing 0s to 1, giving us p trailing 1s
n += 1;            // Flips first p 1s to 0s, and puts a 1 at bit p.
```

Now, to perform Step 3 arithmetically, we just do:

```
n += 2^(c1 - 1) - 1; // Sets trailing c1 - 1 zeros to ones.
```

This math reduces to:

```
next = n + (2^c0 - 1) + 1 + (2^(c1 - 1) - 1)
     = n + 2^c0 + 2^(c1 - 1) - 1
```

The best part is that, using a little bit manipulation, it's simple to code.

```
int getNextArith(int n) {
  /* ... same calculation for c0 and c1 as before ... */
  return n + (1 << c0) + (1 << (c1 - 1)) - 1;
}
```

Arithmetic Approach to Get Previous Number

If c_1 is the number of trailing ones, c_0 is the size of the zero block immediately following, and $p = c_0 + c_1$, we can word the initial `getPrev` solution as follows:

1. Set the pth bit to `0`
2. Set all bits following p to `1`
3. Set bits 0 through $c_0 - 1$ to `0`.

We can implement this arithmetically as follows. For clarity in the example, we will assume `n = 10000011`. This makes $c_1 = 2$ and $c_0 = 5$.

```
n -= 2^c1 - 1;        // Removes trailing 1s. n is now 10000000.
n -= 1;               // Flips trailing 0s. n is now 01111111.
n -= 2^(c0 - 1) - 1;  // Flips last (c0-1) 0s. n is now 01110000.
```

This reduces mathematically to:

```
next   = n - (2^c1 - 1) - 1 - (2^(c0 - 1) - 1).
       = n - 2^c1 - 2^(c0 - 1) + 1
```

Again, this is very easy to implement.

```
int getPrevArith(int n) {
  /* ... same calculation for c0 and c1 as before ... */
  return n - (1 << c1) - (1 << (c0 - 1)) + 1;
}
```

Whew! Don't worry, you wouldn't be expected to get all this in an interview—at least not without a lot of help from the interviewer.

5.5 **Debugger:** Explain what the following code does: ((n & (n-1)) == 0).

pg 116

SOLUTION

We can work backwards to solve this question.

What does it mean if A & B == 0?

It means that A and B never have a 1 bit in the same place. So if `n & (n-1) == 0`, then n and `n-1` never share a 1.

What does n-1 look like (as compared with n)?

Try doing subtraction by hand (in base 2 or 10). What happens?

```
  1101011000 [base 2]           593100 [base 10]
-          1                  -      1
= 1101010111 [base 2]         = 593099 [base 10]
```

When you subtract 1 from a number, you look at the least significant bit. If it's a 1 you change it to 0, and you are done. If it's a zero, you must "borrow" from a larger bit. So, you go to increasingly larger bits, changing each bit from a 0 to a 1, until you find a 1. You flip that 1 to a 0 and you are done.

Thus, `n-1` will look like n, except that n's initial 0s will be 1s in `n-1`, and n's least significant 1 will be a 0 in `n-1`. That is:

```
if     n = abcde1000
then n-1 = abcde0111
```

So what does n & (n-1) == 0 indicate?

n and `n-1` must have no 1s in common. Given that they look like this:

```
if     n = abcde1000
then n-1 = abcde0111
```

`abcde` must be all `0`s, which means that n must look like this: `00001000`. The value n is therefore a power of two.

So, we have our answer: `((n & (n-1)) == 0)` checks if n is a power of 2 (or if n is 0).

5.6 **Conversion:** Write a function to determine the number of bits you would need to flip to convert integer A to integer B.

EXAMPLE

Input: `29 (or: 11101), 15 (or: 01111)`

Output: 2

pg 116

SOLUTION

This seemingly complex problem is actually rather straightforward. To approach this, ask yourself how you would figure out which bits in two numbers are different. Simple: with an XOR.

Each 1 in the XOR represents a bit that is different between A and B. Therefore, to check the number of bits that are different between A and B, we simply need to count the number of bits in A^B that are 1.

```
int bitSwapRequired(int a, int b) {
  int count = 0;
  for (int c = a ^ b; c != 0; c = c >> 1) {
    count += c & 1;
  }
  return count;
}
```

This code is good, but we can make it a bit better. Rather than simply shifting `c` repeatedly while checking the least significant bit, we can continuously flip the least significant bit and count how long it takes `c` to reach 0. The operation `c = c & (c - 1)` will clear the least significant bit in `c`.

The code below utilizes this approach.

```
int bitSwapRequired(int a, int b) {
  int count = 0;
  for (int c = a ^ b; c != 0; c = c & (c-1)) {
    count++;
  }
  return count;
}
```

The above code is one of those bit manipulation problems that comes up sometimes in interviews. Though it'd be hard to come up with it on the spot if you've never seen it before, it is useful to remember the trick for your interviews.

5.7 **Pairwise Swap:** Write a program to swap odd and even bits in an integer with as few instructions as possible (e.g., bit 0 and bit 1 are swapped, bit 2 and bit 3 are swapped, and so on).

pg 116

SOLUTION

Like many of the previous problems, it's useful to think about this problem in a different way. Operating on individual pairs of bits would be difficult, and probably not that efficient either. So how else can we think about this problem?

We can approach this as operating on the odds bits first, and then the even bits. Can we take a number n and move the odd bits over by 1? Sure. We can mask all odd bits with `10101010` in binary (which is `0xAA`), then shift them right by 1 to put them in the even spots. For the even bits, we do an equivalent operation. Finally, we merge these two values.

This takes a total of five instructions. The code below implements this approach.

```
int swapOddEvenBits(int x) {
  return ( ((x & 0xaaaaaaaa) >>> 1) | ((x & 0x55555555) << 1) );
}
```

Note that we use the logical right shift, instead of the arithmetic right shift. This is because we want the sign bit to be filled with a zero.

We've implemented the code above for 32-bit integers in Java. If you were working with 64-bit integers, you would need to change the mask. The logic, however, would remain the same.

5.8 **Draw Line:** A monochrome screen is stored as a single array of bytes, allowing eight consecutive pixels to be stored in one byte. The screen has width `w`, where `w` is divisible by 8 (that is, no byte will be split across rows). The height of the screen, of course, can be derived from the length of the array and the width. Implement a function that draws a horizontal line from (x1, y) to (x2, y).

The method signature should look something like:

`drawLine(byte[] screen, int width, int x1, int x2, int y)`

pg 116

SOLUTION

A naive solution to the problem is straightforward: iterate in a for loop from `x1` to `x2`, setting each pixel along the way. But that's hardly any fun, is it? (Nor is it very efficient.)

A better solution is to recognize that if `x1` and `x2` are far away from each other, several full bytes will be contained between them. These full bytes can be set one at a time by doing `screen[byte_pos] = 0xFF`. The residual start and end of the line can be set using masks.

```
void drawLine(byte[] screen, int width, int x1, int x2, int y) {
  int start_offset = x1 % 8;
  int first_full_byte = x1 / 8;
  if (start_offset != 0) {
    first_full_byte++;
  }

  int end_offset = x2 % 8;
  int last_full_byte = x2 / 8;
  if (end_offset != 7) {
    last_full_byte--;
  }

  // Set full bytes
  for (int b = first_full_byte; b <= last_full_byte; b++) {
    screen[(width / 8) * y + b] = (byte) 0xFF;
  }

  // Create masks for start and end of line
  byte start_mask = (byte) (0xFF >> start_offset);
  byte end_mask = (byte) ~(0xFF >> (end_offset + 1));

  // Set start and end of line
  if ((x1 / 8) == (x2 / 8)) { // x1 and x2 are in the same byte
```

```
      byte mask = (byte) (start_mask & end_mask);
      screen[(width / 8) * y + (x1 / 8)] |= mask;
   } else {
      if (start_offset != 0) {
         int byte_number = (width / 8) * y + first_full_byte - 1;
         screen[byte_number] |= start_mask;
      }
      if (end_offset != 7) {
         int byte_number = (width / 8) * y + last_full_byte + 1;
         screen[byte_number] |= end_mask;
      }
   }
}
```

Be careful on this problem; there are a lot of "gotchas" and special cases. For example, you need to consider the case where x1 and x2 are in the same byte. Only the most careful candidates can implement this code bug-free.

6

Solutions to Math and Logic Puzzles

6.1 **The Heavy Pill:** You have 20 bottles of pills. 19 bottles have 1.0 gram pills, but one has pills of weight 1.1 grams. Given a scale that provides an exact measurement, how would you find the heavy bottle? You can only use the scale once.

pg 122

SOLUTION

Sometimes, tricky constraints can be a clue. This is the case with the constraint that we can only use the scale once.

Because we can only use the scale once, we know something interesting: we must weigh multiple pills at the same time. In fact, we know we must weigh pills from at least 19 bottles at the same time. Otherwise, if we skipped two or more bottles entirely, how could we distinguish between those missed bottles? Remember that we only have *one* chance to use the scale.

So how can we weigh pills from more than one bottle and discover which bottle has the heavy pills? Let's suppose there were just two bottles, one of which had heavier pills. If we took one pill from each bottle, we would get a weight of 2.1 grams, but we wouldn't know which bottle contributed the extra 0.1 grams. We know we must treat the bottles differently somehow.

If we took one pill from Bottle #1 and two pills from Bottle #2, what would the scale show? It depends. If Bottle #1 were the heavy bottle, we would get 3.1 grams. If Bottle #2 were the heavy bottle, we would get 3.2 grams. And that is the trick to this problem.

We know the "expected" weight of a bunch of pills. The difference between the expected weight and the actual weight will indicate which bottle contributed the heavier pills, *provided* we select a different number of pills from each bottle.

We can generalize this to the full solution: take one pill from Bottle #1, two pills from Bottle #2, three pills from Bottle #3, and so on. Weigh this mix of pills. If all pills were one gram each, the scale would read 210 grams (`1 + 2 + ... + 20 = 20 * 21 / 2 = 210`). Any "overage" must come from the extra 0.1 gram pills.

This formula will tell you the bottle number:

$$\frac{\texttt{weight-210grams}}{\texttt{0.1grams}}$$

So, if the set of pills weighed 211.3 grams, then Bottle #13 would have the heavy pills.

6.2 **Basketball:** You have a basketball hoop and someone says that you can play one of two games.

Game 1: You get one shot to make the hoop.

Game 2: You get three shots and you have to make two of three shots.

If p is the probability of making a particular shot, for which values of p should you pick one game or the other?

pg 123

SOLUTION

To solve this problem, we can apply straightforward probability laws by comparing the probabilities of winning each game.

Probability of winning Game 1:

The probability of winning Game 1 is p, by definition.

Probability of winning Game 2:

Let `s(k,n)` be the probability of making exactly k shots out of n. The probability of winning Game 2 is the probability of making exactly two shots out of three OR making all three shots. In other words:

```
P(winning) = s(2,3) + s(3,3)
```

The probability of making all three shots is:

$$s(3,3) = p^3$$

The probability of making exactly two shots is:

```
P(making 1 and 2, and missing 3)
       + P(making 1 and 3, and missing 2)
       + P(missing 1, and making 2 and 3)
    = p * p * (1 - p) + p * (1 - p) * p + (1 - p) * p * p
```

$$= 3 (1 - p) p^2$$

Adding these together, we get:

$$= p^3 + 3 (1 - p) p^2$$
$$= p^3 + 3p^2 - 3p^3$$
$$= 3p^2 - 2p^3$$

Which game should you play?

You should play Game 1 if `P(Game 1) > P(Game 2)`:

$$p > 3p^2 - 2p^3.$$
$$1 > 3p - 2p^2$$
$$2p^2 - 3p + 1 > 0$$
$$(2p - 1)(p - 1) > 0$$

Both terms must be positive, or both must be negative. But we know $p < 1$, so $p - 1 < 0$. This means both terms must be negative.

$$2p - 1 < 0$$
$$2p < 1$$
$$p < .5$$

So, we should play Game 1 if $0 < p < .5$ and Game 2 if $.5 < p < 1$.

If $p = 0, 0.5$, or 1, then `P(Game 1) = P(Game 2)`, so it doesn't matter which game we play.

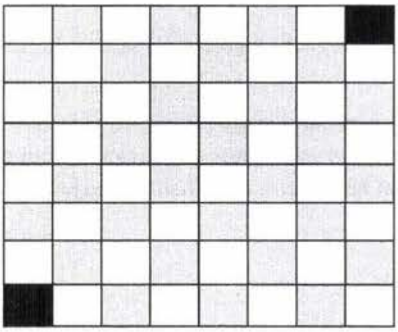
6.3 **Dominos:** There is an 8x8 chessboard in which two diagonally opposite corners have been cut off. You are given 31 dominos, and a single domino can cover exactly two squares. Can you use the 31 dominos to cover the entire board? Prove your answer (by providing an example or showing why it's impossible).

pg 123

SOLUTION

At first, it seems like this should be possible. It's an 8 x 8 board, which has 64 squares, but two have been cut off, so we're down to 62 squares. A set of 31 dominoes should be able to fit there, right?

When we try to lay down dominoes on row 1, which only has 7 squares, we may notice that one domino must stretch into the row 2. Then, when we try to lay down dominoes onto row 2, again we need to stretch a domino into row 3.



For each row we place, we'll always have one domino that needs to poke into the next row. No matter how many times and ways we try to solve this issue, we won't be able to successfully lay down all the dominoes.

There's a cleaner, more solid proof for why it won't work. The chessboard initially has 32 black and 32 white squares. By removing opposite corners (which must be the same color), we're left with 30 of one color and 32 of the other color. Let's say, for the sake of argument, that we have 30 black and 32 white squares.

Each domino we set on the board will always take up one white and one black square. Therefore, 31 dominos will take up 31 white squares and 31 black squares exactly. On this board, however, we must have 30 black squares and 32 white squares. Hence, it is impossible.

6.4 **Ants on a Triangle:** There are three ants on different vertices of a triangle. What is the probability of collision (between any two or all of them) if they start walking on the sides of the triangle? Assume that each ant randomly picks a direction, with either direction being equally likely to be chosen, and that they walk at the same speed.

Similarly, find the probability of collision with n ants on an n-vertex polygon.

pg 123

SOLUTION

The ants will collide if any of them are moving towards each other. So, the only way that they won't collide is if they are all moving in the same direction (clockwise or counterclockwise). We can compute this probability and work backwards from there.

Since each ant can move in two directions, and there are three ants, the probability is:

```
P(clockwise) = (1/2)^3
P(counter clockwise) = (1/2)^3
P(same direction) = (1/2)^3 + (1/2)^3 = 1/4
```

The probability of collision is therefore the probability of the ants *not* moving in the same direction:

```
P(collision) = 1 - P(same direction) = 1 - 1/4 = 3/4
```

To generalize this to an n-vertex polygon: there are still only two ways in which the ants can move to avoid a collision, but there are 2^n ways they can move in total. Therefore, in general, probability of collision is:

```
P(clockwise) = (1/2)^n
P(counter) = (1/2)^n
P(same direction) = 2(1/2)^n = (1/2)^(n-1)
P(collision) = 1 - P(same direction) = 1 - (1/2)^(n-1)
```

6.5 **Jugs of Water:** You have a five-quart jug, a three-quart jug, and an unlimited supply of water (but no measuring cups). How would you come up with exactly four quarts of water? Note that the jugs are oddly shaped, such that filling up exactly "half" of the jug would be impossible.

pg 123

SOLUTION

If we just play with the jugs, we'll find that we can pour water back and forth between them as follows:

5 Quart	3 Quart	Action
5	0	Filled 5-quart jug.
2	3	Filled 3-quart with 5-quart's contents.
2	0	Dumped 3-quart.
0	2	Fill 3-quart with 5-quart's contents.
5	2	Filled 5-quart.
4	3	Fill remainder of 3-quart with 5-quart.
4		Done! We have 4 quarts.

This question, like many puzzle questions, has a math/computer science root. If the two jug sizes are relatively prime, you can measure any value between one and the sum of the jug sizes.

6.6 Blue-Eyed Island: A bunch of people are living on an island, when a visitor comes with a strange order: all blue-eyed people must leave the island as soon as possible. There will be a flight out at 8:00pm every evening. Each person can see everyone else's eye color, but they do not know their own (nor is anyone allowed to tell them). Additionally, they do not know how many people have blue eyes, although they do know that at least one person does. How many days will it take the blue-eyed people to leave?

pg 123

SOLUTION

Let's apply the Base Case and Build approach. Assume that there are n people on the island and c of them have blue eyes. We are explicitly told that `c > 0`.

Case c = 1: Exactly one person has blue eyes.

Assuming all the people are intelligent, the blue-eyed person should look around and realize that no one else has blue eyes. Since he knows that at least one person has blue eyes, he must conclude that it is he who has blue eyes. Therefore, he would take the flight that evening.

Case c = 2: Exactly two people have blue eyes.

The two blue-eyed people see each other, but are unsure whether c is 1 or 2. They know, from the previous case, that if c = 1, the blue-eyed person would leave on the first night. Therefore, if the other blue-eyed person is still there, he must deduce that c = 2, which means that he himself has blue eyes. Both men would then leave on the second night.

Case c > 2: The General Case.

As we increase c, we can see that this logic continues to apply. If c = 3, then those three people will immediately know that there are either 2 or 3 people with blue eyes. If there were two people, then those two people would have left on the second night. So, when the others are still around after that night, each person would conclude that c = 3 and that they, therefore, have blue eyes too. They would leave that night.

This same pattern extends up through any value of c. Therefore, if c men have blue eyes, it will take c nights for the blue-eyed men to leave. All will leave on the same night.

6.7 The Apocalypse: In the new post-apocalyptic world, the world queen is desperately concerned about the birth rate. Therefore, she decrees that all families should ensure that they have one girl or else they face massive fines. If all families abide by this policy—that is, they have continue to have children until they have one girl, at which point they immediately stop—what will the gender ratio of the new generation be? (Assume that the odds of someone having a boy or a girl on any given pregnancy is equal.) Solve this out logically and then write a computer simulation of it.

pg 123

SOLUTION

If each family abides by this policy, then each family will have a sequence of zero or more boys followed by a single girl. That is, if "G" indicates a girl and "B" indicates a boy, the sequence of children will look like one of: G; BG; BBG; BBBG; BBBBG; and so on.

We can solve this problem multiple ways.

Mathematically

We can work out the probability for each gender sequence.

- $P(G) = 1/2$. That is, 50% of families will have a girl first. The others will go on to have more children.
- $P(BG) = 1/4$. Of those who have a second child (which is 50%), 50% of them will have a girl the next time.
- $P(BBG) = 1/8$. Of those who have a third child (which is 25%), 50% of them will have a girl the next time.

And so on.

We know that every family has exactly one girl. How many boys does each family have, on average? To compute this, we can look at the expected value of the number of boys. The expected value of the number of boys is the probability of each sequence multiplied by the number of boys in that sequence.

Sequence	Number of Boys	Probability	Number of Boys * Probability
G	0	$1/2$	0
BG	1	$1/4$	$1/4$
BBG	2	$1/8$	$2/8$
BBBG	3	$1/16$	$3/16$
BBBBG	4	$1/32$	$4/32$
BBBBBG	5	$1/64$	$5/64$
BBBBBBG	6	$1/128$	$6/128$

Or in other words, this is the sum of i to infinity of i divided by 2^i.

$$\sum_{i=0}^{\infty} \frac{i}{2^i}$$

You probably won't know this off the top of your head, but we can try to estimate it. Let's try converting the above values to a common denominator of 128 (2^6).

$1/4 = 32/128$ $\qquad$ $4/32 = 16/128$

$2/8 = 32/128$ $\qquad$ $5/64 = 10/128$

$3/16 = 24/128$ $\qquad$ $6/128 = 6/128$

$$\frac{32 + 32 + 24 + 16 + 10 + 6}{128} = \frac{120}{128}$$

This looks like it's going to inch closer to $128/128$ (which is of course 1). This "looks like" intuition is valuable, but it's not exactly a mathematical concept. It's a clue though and we can turn to logic here. Should it be 1?

Logically

If the earlier sum is 1, this would mean that the gender ratio is even. Families contribute exactly one girl and on average one boy. The birth policy is therefore ineffective. Does this make sense?

At first glance, this seems wrong. The policy is designed to favor girls as it ensures that all families have a girl.

On the other hand, the families that keep having children contribute (potentially) multiple boys to the population. This could offset the impact of the "one girl" policy.

One way to think about this is to imagine that we put all the gender sequence of each family into one giant string. So if family 1 has BG, family 2 has BBG, and family 3 has G, we would write BGBBGG.

In fact, we don't really care about the groupings of families because we're concerned about the population as a whole. As soon as a child is born, we can just append its gender (B or G) to the string.

What are the odds of the next character being a G? Well, if the odds of having a boy and girl is the same, then the odds of the next character being a G is 50%. Therefore, roughly half of the string should be Gs and half should be Bs, giving an even gender ratio.

This actually makes a lot of sense. Biology hasn't been changed. Half of newborn babies are girls and half are boys. Abiding by some rule about when to stop having children doesn't change this fact.

Therefore, the gender ratio is 50% girls and 50% boys.

Simulation

We'll write this in a simple way that directly corresponds to the problem.

```
double runNFamilies(int n) {
  int boys = 0;
  int girls = 0;
  for (int i = 0; i < n; i++) {
    int[] genders = runOneFamily();
    girls += genders[0];
    boys += genders[1];
  }
  return girls / (double) (boys + girls);
}

int[] runOneFamily() {
  Random random = new Random();
  int boys = 0;
  int girls = 0;
  while (girls == 0) { // until we have a girl
    if (random.nextBoolean()) { // girl
      girls += 1;
    } else { // boy
      boys += 1;
    }
  }
  int[] genders = {girls, boys};
  return genders;
}
```

Sure enough, if you run this on large values of n, you should get something very close to 0.5.

6.8 The Egg Drop Problem: There is a building of 100 floors. If an egg drops from the Nth floor or above, it will break. If it's dropped from any floor below, it will not break. You're given two eggs. Find N, while minimizing the number of drops for the worst case.

pg 124

SOLUTION

We may observe that, regardless of how we drop Egg 1, Egg 2 must do a linear search (from lowest to highest) between the "breaking floor" and the next highest non-breaking floor. For example, if Egg 1 is dropped from floors 5 and 10 without breaking, but it breaks when it's dropped from floor 15, then Egg 2 must be dropped, in the worst case, from floors 11, 12, 13, and 14.

The Approach

As a first try, suppose we drop an egg from the 10th floor, then the 20th, ...

- If Egg 1 breaks on the first drop (floor 10), then we have at most 10 drops total.
- If Egg 1 breaks on the last drop (floor 100), then we have at most 19 drops total (floors 10, 20, ...,90, 100, then 91 through 99).

That's pretty good, but all we've considered is the absolute worst case. We should do some "load balancing" to make those two cases more even.

Our goal is to create a system for dropping Egg 1 such that the number of drops is as consistent as possible, whether Egg 1 breaks on the first drop or the last drop.

1. A perfectly load-balanced system would be one in which `Drops(Egg 1) + Drops(Egg 2)` is always the same, regardless of where Egg 1 breaks.
2. For that to be the case, since each drop of Egg 1 takes one more step, Egg 2 is allowed one fewer step.
3. We must, therefore, reduce the number of steps potentially required by Egg 2 by one drop each time. For example, if Egg 1 is dropped on floor 20 and then floor 30, Egg 2 is potentially required to take 9 steps. When we drop Egg 1 again, we must reduce potential Egg 2 steps to only 8. That is, we must drop Egg 1 at floor 39.
4. Therefore, Egg 1 must start at floor X, then go up by X-1 floors, then X-2, ..., until it gets to 100.
5. Solve for X.

$$X + (X - 1) + (X - 2) + \ldots + 1 = 100$$
$$X(X+1)/2 = 100$$
$$X \approx 13.65$$

X clearly needs to be an integer. Should we round X up or down?

- If we round X up to 14, then we would go up by 14, then 13, then 12, and so on. The last increment would be 4, and it would happen on floor 99. If Egg 1 broke on any of the prior floors, we know we've balanced the eggs such that the number of drops of Egg 1 and Egg 2 always sum to the same thing: 14. If Egg 1 hasn't broken by floor 99, then we just need one more drop to determine if it will break at floor 100. Either way, the number of drops is no more than 14.
- If we round X down to 13, then we would go up by 13, then 12, then 11, and so on. The last increment will be 1 and it will happen at floor 91. This is after 13 drops. Floors 92 through 100 have not been covered yet. We can't cover those floors in just one drop (which would be necessary to merely tie the

"round up" case).

Therefore, we should round X up to 14. That is, we go to floor 14, then 27, then 39, ... This takes 14 steps in the worse case.

As in many other maximizing / minimizing problems, the key in this problem is "worst case balancing."

The following code simulates this approach.

```
int breakingPoint = ...;
int countDrops = 0;

boolean drop(int floor) {
  countDrops++;
  return floor >= breakingPoint;
}

int findBreakingPoint(int floors) {
  int interval = 14;
  int previousFloor = 0;
  int egg1 = interval;

  /* Drop egg1 at decreasing intervals. */
  while (!drop(egg1) && egg1 <= floors) {
    interval -= 1;
    previousFloor = egg1;
    egg1 += interval;
  }

  /* Drop egg2 at 1 unit increments. */
  int egg2 = previousFloor + 1;
  while (egg2 < egg1 && egg2 <= floors && !drop(egg2)) {
    egg2 += 1;
  }

  /* If it didn't break, return -1. */
  return egg2 > floors ? -1 : egg2;
}
```

If we want to generalize this code for more building sizes, then we can solve for x in:

$$x(x+1)/2 = \text{number of floors}$$

This will involve the quadratic formula.

6.9 **100 Lockers:** There are 100 closed lockers in a hallway. A man begins by opening all 100 lockers. Next, he closes every second locker. Then, on his third pass, he toggles every third locker (closes it if it is open or opens it if it is closed). This process continues for 100 passes, such that on each pass `i`, the man toggles every `i`th locker. After his 100th pass in the hallway, in which he toggles only locker #100, how many lockers are open?

pg 124

SOLUTION

We can tackle this problem by thinking through what it means for a door to be toggled. This will help us deduce which doors at the very end will be left opened.

Question: For which rounds is a door toggled (open or closed)?

A door n is toggled once for each factor of n, including itself and 1. That is, door 15 is toggled on rounds 1, 3, 5, and 15.

Question: When would a door be left open?

A door is left open if the number of factors (which we will call x) is odd. You can think about this by pairing factors off as an open and a close. If there's one remaining, the door will be open.

Question: When would x be odd?

The value x is odd if n is a perfect square. Here's why: pair n's factors by their complements. For example, if n is 36, the factors are (1, 36), (2, 18), (3, 12), (4, 9), (6, 6). Note that (6, 6) only contributes one factor, thus giving n an odd number of factors.

Question: How many perfect squares are there?

There are 10 perfect squares. You could count them (1, 4, 9, 16, 25, 36, 49, 64, 81, 100), or you could simply realize that you can take the numbers 1 through 10 and square them:

```
1*1, 2*2, 3*3, ..., 10*10
```

Therefore, there are 10 lockers open at the end of this process.

6.10 **Poison**: You have 1000 bottles of soda, and exactly one is poisoned. You have 10 test strips which can be used to detect poison. A single drop of poison will turn the test strip positive permanently. You can put any number of drops on a test strip at once and you can reuse a test strip as many times as you'd like (as long as the results are negative). However, you can only run tests once per day and it takes seven days to return a result. How would you figure out the poisoned bottle in as few days as possible?

Follow up: Write code to simulate your approach.

pg 124

SOLUTION

Observe the wording of the problem. Why seven days? Why not have the results just return immediately?

The fact that there's such a lag between starting a test and reading the results likely means that we'll be doing something else in the meantime (running additional tests). Let's hold on to that thought, but start off with a simple approach just to wrap our heads around the problem.

Naive Approach (28 days)

A simple approach is to divide the bottles across the 10 test strips, first in groups of 100. Then, we wait seven days. When the results come back, we look for a positive result across the test strips. We select the bottles associated with the positive test strip, "toss" (i.e., ignore) all the other bottles, and repeat the process. We perform this operation until there is only one bottle left in the test set.

1. Divide bottles across available test strips, one drop per test strip.
2. After seven days, check the test strips for results.
3. On the positive test strip: select the bottles associated with it into a new set of bottles. If this set size is 1,

we have located the poisoned bottle. If it's greater than one, go to step 1.

To simulate this, we'll build classes for `Bottle` and `TestStrip` that mirror the problem's functionality.

```
class Bottle {
  private boolean poisoned = false;
  private int id;

  public Bottle(int id) { this.id = id; }
  public int getId() { return id; }
  public void setAsPoisoned() { poisoned = true; }
  public boolean isPoisoned() { return poisoned; }
}

class TestStrip {
  public static int DAYS_FOR_RESULT = 7;
  private ArrayList<ArrayList<Bottle>> dropsByDay =
    new ArrayList<ArrayList<Bottle>>();
  private int id;

  public TestStrip(int id) { this.id = id; }
  public int getId() { return id; }

  /* Resize list of days/drops to be large enough. */
  private void sizeDropsForDay(int day) {
    while (dropsByDay.size() <= day) {
      dropsByDay.add(new ArrayList<Bottle>());
    }
  }

  /* Add drop from bottle on specific day. */
  public void addDropOnDay(int day, Bottle bottle) {
    sizeDropsForDay(day);
    ArrayList<Bottle> drops = dropsByDay.get(day);
    drops.add(bottle);
  }

  /* Checks if any of the bottles in the set are poisoned. */
  private boolean hasPoison(ArrayList<Bottle> bottles) {
    for (Bottle b : bottles) {
      if (b.isPoisoned()) {
        return true;
      }
    }
    return false;
  }

  /* Gets bottles used in the test DAYS_FOR_RESULT days ago. */
  public ArrayList<Bottle> getLastWeeksBottles(int day) {
    if (day < DAYS_FOR_RESULT) {
      return null;
    }
    return dropsByDay.get(day - DAYS_FOR_RESULT);
  }

  /* Checks for poisoned bottles since before DAYS_FOR_RESULT */
  public boolean isPositiveOnDay(int day) {
```

```
      int testDay = day - DAYS_FOR_RESULT;
      if (testDay < 0 || testDay >= dropsByDay.size()) {
         return false;
      }
      for (int d = 0; d <= testDay; d++) {
         ArrayList<Bottle> bottles = dropsByDay.get(d);
         if (hasPoison(bottles)) {
            return true;
         }
      }
      return false;
   }
}
```

This is just one way of simulating the behavior of the bottles and test strips, and each has its pros and cons.

With this infrastructure built, we can now implement code to test our approach.

```
int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
   int today = 0;

   while (bottles.size() > 1 && strips.size() > 0) {
      /* Run tests. */
      runTestSet(bottles, strips, today);

      /* Wait for results. */
      today += TestStrip.DAYS_FOR_RESULT;

      /* Check results. */
      for (TestStrip strip : strips) {
         if (strip.isPositiveOnDay(today)) {
            bottles = strip.getLastWeeksBottles(today);
            strips.remove(strip);
            break;
         }
      }
   }

   if (bottles.size() == 1) {
      return bottles.get(0).getId();
   }
   return -1;
}

/* Distribute bottles across test strips evenly. */
void runTestSet(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips, int day) {
   int index = 0;
   for (Bottle bottle : bottles) {
      TestStrip strip = strips.get(index);
      strip.addDropOnDay(day, bottle);
      index = (index + 1) % strips.size();
   }
}

/* The complete code can be found in the downloadable code attachment. */
```

Note that this approach makes the assumption that there will always be multiple test strips at each round. This assumption is valid for 1000 bottles and 10 test strips.

If we can't assume this, we can implement a fail-safe. If we have just one test strip remaining, we start doing one bottle at a time: test a bottle, wait a week, test another bottle. This approach will take at most 28 days.

Optimized Approach (10 days)

As noted in the beginning of the solution, it might be more optimal to run multiple tests at once.

If we divide the bottles up into 10 groups (with bottles 0 - 99 going to strip 0, bottles 100 - 199 going to strip 1, bottles 200 - 299 going to strip 2, and so on), then day 7 will reveal the first digit of the bottle number. A positive result on strip `i` at day 7 shows that the first digit (100's digit) of the bottle number is `i`.

Dividing the bottles in a different way can reveal the second or third digit. We just need to run these tests on different days so that we don't confuse the results.

	Day 0 -> 7	Day 1 -> 8	Day 2 -> 9
Strip 0	0xx	x0x	xx0
Strip 1	1xx	x1x	xx1
Strip 2	2xx	x2x	xx2
Strip 3	3xx	x3x	xx3
Strip 4	4xx	x4x	xx4
Strip 5	5xx	x5x	xx5
Strip 6	6xx	x6x	xx6
Strip 7	7xx	x7x	xx7
Strip 8	8xx	x8x	xx8
Strip 9	9xx	x9x	xx9

For example, if day 7 showed a positive result on strip 4, day 8 showed a positive result on strip 3, and day 9 showed a positive result on strip 8, then this would map to bottle #438.

This mostly works, except for one edge case: what happens if the poisoned bottle has a duplicate digit? For example, bottle #882 or bottle #383.

In fact, these cases are quite different. If day 8 doesn't have any "new" positive results, then we can conclude that digit 2 equals digit 1.

The bigger issue is what happens if day 9 doesn't have any new positive results. In this case, all we know is that digit 3 equals either digit 1 or digit 2. We could not distinguish between bottle #383 and bottle #388. They will both have the same pattern of test results.

We will need to run one additional test. We could run this at the end to clear up ambiguity, but we can also run it at day 3, just in case there's any ambiguity. All we need to do is shift the final digit so that it winds up in a different place than day 2's results.

	Day 0 -> 7	Day 1 -> 8	Day 2 -> 9	Day 3 -> 10
Strip 0	0xx	x0x	xx0	xx9
Strip 1	1xx	x1x	xx1	xx0
Strip 2	2xx	x2x	xx2	xx1
Strip 3	3xx	x3x	xx3	xx2
Strip 4	4xx	x4x	xx4	xx3
Strip 5	5xx	x5x	xx5	xx4

	Day 0 -> 7	Day 1 -> 8	Day 2 -> 9	Day 3 -> 10
Strip 6	6xx	x6x	xx6	xx5
Strip 7	7xx	x7x	xx7	xx6
Strip 8	8xx	x8x	xx8	xx7
Strip 9	9xx	x9x	xx9	xx8

Now, bottle #383 will see (Day 7 = #3, Day 8 -> #8, Day 9 -> [NONE], Day 10 -> #4), while bottle #388 will see (Day 7 = #3, Day 8 -> #8, Day 9 -> [NONE], Day 10 -> #9). We can distinguish between these by "reversing" the shifting on day 10's results.

What happens, though, if day 10 still doesn't see any new results? Could this happen?

Actually, yes. Bottle #898 would see (Day 7 = #8, Day 8 -> #9, Day 9 -> [NONE], Day 10 -> [NONE]). That's okay, though. We just need to distinguish bottle #898 from #899. Bottle #899 will see (Day 7 = #8, Day 8 -> #9, Day 9 -> [NONE], Day 10 -> #0).

The "ambiguous" bottles from day 9 will always map to different values on day 10. The logic is:

- If Day 3->10's test reveals a new test result, "unshift" this value to derive the third digit.
- Otherwise, we know that the third digit equals either the first digit or the second digit *and* that the third digit, when shifted, still equals either the first digit or the second digit. Therefore, we just need to figure out whether the first digit "shifts" into the second digit or the other way around. In the former case, the third digit equals the first digit. In the latter case, the third digit equals the second digit.

Implementing this requires some careful work to prevent bugs.

```
int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
  if (bottles.size() > 1000 || strips.size() < 10) return -1;

  int tests = 4; // three digits, plus one extra
  int nTestStrips = strips.size();

  /* Run tests. */
  for (int day = 0; day < tests; day++) {
    runTestSet(bottles, strips, day);
  }

  /* Get results. */
  HashSet<Integer> previousResults = new HashSet<Integer>();
  int[] digits = new int[tests];
  for (int day = 0; day < tests; day++) {
    int resultDay = day + TestStrip.DAYS_FOR_RESULT;
    digits[day] = getPositiveOnDay(strips, resultDay, previousResults);
    previousResults.add(digits[day]);
  }

  /* If day 1's results matched day 0's, update the digit. */
  if (digits[1] == -1) {
    digits[1] = digits[0];
  }

  /* If day 2 matched day 0 or day 1, check day 3. Day 3 is the same as day 2, but
   * incremented by 1. */
  if (digits[2] == -1) {
```

```
      if (digits[3] == -1) { /* Day 3 didn't give new result */
         /* Digit 2 equals digit 0 or digit 1. But, digit 2, when incremented also
          * matches digit 0 or digit 1. This means that digit 0 incremented matches
          * digit 1, or the other way around. */
         digits[2] = ((digits[0] + 1) % nTestStrips) == digits[1] ?
                      digits[0] : digits[1];
      } else {
          digits[2] = (digits[3] - 1 + nTestStrips) % nTestStrips;
      }
   }

   return digits[0] * 100 + digits[1] * 10 + digits[2];
}

/* Run set of tests for this day. */
void runTestSet(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips, int day) {
   if (day > 3) return; // only works for 3 days (digits) + one extra

   for (Bottle bottle : bottles) {
      int index = getTestStripIndexForDay(bottle, day, strips.size());
      TestStrip testStrip = strips.get(index);
      testStrip.addDropOnDay(day, bottle);
   }
}

/* Get strip that should be used on this bottle on this day. */
int getTestStripIndexForDay(Bottle bottle, int day, int nTestStrips) {
   int id = bottle.getId();
   switch (day) {
      case 0: return id /100;
      case 1: return (id % 100) / 10;
      case 2: return id % 10;
      case 3: return (id % 10 + 1) % nTestStrips;
      default: return -1;
   }
}

/* Get results that are positive for a particular day, excluding prior results. */
int getPositiveOnDay(ArrayList<TestStrip> testStrips, int day,
                     HashSet<Integer> previousResults) {
   for (TestStrip testStrip : testStrips) {
      int id = testStrip.getId();
      if (testStrip.isPositiveOnDay(day) && !previousResults.contains(id)) {
         return testStrip.getId();
      }
   }
   return -1;
}
```

It will take 10 days in the worst case to get a result with this approach.

Optimal Approach (7 days)

We can actually optimize this slightly more, to return a result in just seven days. This is of course the minimum number of days possible.

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.

```
int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
  runTests(bottles, strips);
  ArrayList<Integer> positive = getPositiveOnDay(strips, 7);
  return setBits(positive);
}

/* Add bottle contents to test strips */
void runTests(ArrayList<Bottle> bottles, ArrayList<TestStrip> testStrips) {
  for (Bottle bottle : bottles) {
    int id = bottle.getId();
    int bitIndex = 0;
    while (id > 0) {
      if ((id & 1) == 1) {
        testStrips.get(bitIndex).addDropOnDay(0, bottle);
      }
      bitIndex++;
      id >>= 1;
    }
  }
}

/* Get test strips that are positive on a particular day. */
ArrayList<Integer> getPositiveOnDay(ArrayList<TestStrip> testStrips, int day) {
  ArrayList<Integer> positive = new ArrayList<Integer>();
  for (TestStrip testStrip : testStrips) {
    int id = testStrip.getId();
    if (testStrip.isPositiveOnDay(day)) {
      positive.add(id);
    }
  }
  return positive;
}

/* Create number by setting bits with indices specified in positive. */
int setBits(ArrayList<Integer> positive) {
  int id = 0;
  for (Integer bitIndex : positive) {
    id |= 1 << bitIndex;
  }
  return id;
}
```

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.

7

Solutions to Object-Oriented Design

7.1 Deck of Cards: Design the data structures for a generic deck of cards. Explain how you would subclass the data structures to implement blackjack.

pg 127

SOLUTION

First, we need to recognize that a "generic" deck of cards can mean many things. Generic could mean a standard deck of cards that can play a poker-like game, or it could even stretch to Uno or Baseball cards. It is important to ask your interviewer what she means by generic.

Let's assume that your interviewer clarifies that the deck is a standard 52-card set, like you might see used in a blackjack or poker game. If so, the design might look like this:

```
public enum Suit {
  Club (0), Diamond (1), Heart (2), Spade (3);
  private int value;
  private Suit(int v) { value = v; }
  public int getValue() { return value; }
  public static Suit getSuitFromValue(int value) { ... }
}

public class Deck <T extends Card> {
  private ArrayList<T> cards; // all cards, dealt or not
  private int dealtIndex = 0; // marks first undealt card

  public void setDeckOfCards(ArrayList<T> deckOfCards) { ... }

  public void shuffle() { ... }
  public int remainingCards() {
    return cards.size() - dealtIndex;
  }
  public T[] dealHand(int number) { ... }
  public T dealCard() { ... }
}

public abstract class Card {
  private boolean available = true;

  /* number or face that's on card - a number 2 through 10, or 11 for Jack, 12 for
   * Queen, 13 for King, or 1 for Ace */
  protected int faceValue;
  protected Suit suit;
```

```

   public Card(int c, Suit s) {
      faceValue = c;
      suit = s;
   }

   public abstract int value();
   public Suit suit() { return suit; }

   /* Checks if the card is available to be given out to someone */
   public boolean isAvailable() { return available; }
   public void markUnavailable() { available = false; }
   public void markAvailable() { available = true; }
}

public class Hand <T extends Card> {
   protected ArrayList<T> cards = new ArrayList<T>();

   public int score() {
      int score = 0;
      for (T card : cards) {
         score += card.value();
      }
      return score;
   }

   public void addCard(T card) {
      cards.add(card);
   }
}
```

In the above code, we have implemented `Deck` with generics but restricted the type of T to `Card`. We have also implemented `Card` as an abstract class, since methods like `value()` don't make much sense without a specific game attached to them. (You could make a compelling argument that they should be implemented anyway, by defaulting to standard poker rules.)

Now, let's say we're building a blackjack game, so we need to know the value of the cards. Face cards are 10 and an ace is 11 (most of the time, but that's the job of the Hand class, not the following class).

```
public class BlackJackHand extends Hand<BlackJackCard> {
   /* There are multiple possible scores for a blackjack hand, since aces have
    * multiple values. Return the highest possible score that's under 21, or the
    * lowest score that's over. */
   public int score() {
      ArrayList<Integer> scores = possibleScores();
      int maxUnder = Integer.MIN_VALUE;
      int minOver = Integer.MAX_VALUE;
      for (int score : scores) {
         if (score > 21 && score < minOver) {
            minOver = score;
         } else if (score <= 21 && score > maxUnder) {
            maxUnder = score;
         }
      }
      return maxUnder == Integer.MIN_VALUE ? minOver : maxUnder;
   }

```

```
    /* return a list of all possible scores this hand could have (evaluating each
     * ace as both 1 and 11 */
    private ArrayList<Integer> possibleScores() { ... }

    public boolean busted() { return score() > 21; }
    public boolean is21() { return score() == 21; }
    public boolean isBlackJack() { ... }
}

public class BlackJackCard extends Card {
    public BlackJackCard(int c, Suit s) { super(c, s); }
    public int value() {
       if (isAce()) return 1;
       else if (faceValue >= 11 && faceValue <= 13) return 10;
       else return faceValue;
    }

    public int minValue() {
       if (isAce()) return 1;
       else return value();
    }

    public int maxValue() {
       if (isAce()) return 11;
       else return value();
    }

    public boolean isAce() {
       return faceValue == 1;
    }

    public boolean isFaceCard() {
       return faceValue >= 11 && faceValue <= 13;
    }
}
```

This is just one way of handling aces. We could, alternatively, create a class of type Ace that extends BlackJackCard.

An executable, fully automated version of blackjack is provided in the downloadable code attachment.

7.2 **Call Center:** Imagine you have a call center with three levels of employees: respondent, manager, and director. An incoming telephone call must be first allocated to a respondent who is free. If the respondent can't handle the call, he or she must escalate the call to a manager. If the manager is not free or not able to handle it, then the call should be escalated to a director. Design the classes and data structures for this problem. Implement a method dispatchCall() which assigns a call to the first available employee.

pg 127

SOLUTION

All three ranks of employees have different work to be done, so those specific functions are profile specific. We should keep these things within their respective class.

There are a few things which are common to them, like address, name, job title, and age. These things can be kept in one class and can be extended or inherited by others.

Finally, there should be one `CallHandler` class which would route the calls to the correct person.

Note that on any object-oriented design question, there are many ways to design the objects. Discuss the trade-offs of different solutions with your interviewer. You should usually design for long-term code flexibility and maintenance.

We'll go through each of the classes below in detail.

`CallHandler` represents the body of the program, and all calls are funneled first through it.

```
public class CallHandler {
  /* 3 levels of employees: respondents, managers, directors. */
  private final int LEVELS = 3;

  /* Initialize 10 respondents, 4 managers, and 2 directors. */
  private final int NUM_RESPONDENTS = 10;
  private final int NUM_MANAGERS = 4;
  private final int NUM_DIRECTORS = 2;

  /* List of employees, by level.
   * employeeLevels[0] = respondents
   * employeeLevels[1] = managers
   * employeeLevels[2] = directors
   */
  List<List<Employee>> employeeLevels;

  /* queues for each call's rank */
  List<List<Call>> callQueues;

  public CallHandler() { ... }

  /* Gets the first available employee who can handle this call.*/
  public Employee getHandlerForCall(Call call) { ... }

  /* Routes the call to an available employee, or saves in a queue if no employee
   * is available. */
  public void dispatchCall(Caller caller) {
    Call call = new Call(caller);
    dispatchCall(call);
  }

  /* Routes the call to an available employee, or saves in a queue if no employee
   * is available. */
  public void dispatchCall(Call call) {
    /* Try to route the call to an employee with minimal rank. */
    Employee emp = getHandlerForCall(call);
    if (emp != null) {
      emp.receiveCall(call);
      call.setHandler(emp);
    } else {
      /* Place the call into corresponding call queue according to its rank. */
      call.reply("Please wait for free employee to reply");
      callQueues.get(call.getRank().getValue()).add(call);
    }
  }
```

```

    /* An employee got free. Look for a waiting call that employee can serve. Return
     * true if we assigned a call, false otherwise. */
    public boolean assignCall(Employee emp) { ... }
}
```

`Call` represents a call from a user. A call has a minimum rank and is assigned to the first employee who can handle it.

```
public class Call {
    /* Minimal rank of employee who can handle this call. */
    private Rank rank;

    /* Person who is calling. */
    private Caller caller;

    /* Employee who is handling call. */
    private Employee handler;

    public Call(Caller c) {
        rank = Rank.Responder;
        caller = c;
    }

    /* Set employee who is handling call. */
    public void setHandler(Employee e) { handler = e; }

    public void reply(String message) { ... }
    public Rank getRank() { return rank; }
    public void setRank(Rank r) { rank = r; }
    public Rank incrementRank() { ... }
    public void disconnect() { ... }
}
```

`Employee` is a super class for the `Director`, `Manager`, and `Respondent` classes. It is implemented as an abstract class since there should be no reason to instantiate an `Employee` type directly.

```
abstract class Employee {
    private Call currentCall = null;
    protected Rank rank;

    public Employee(CallHandler handler) { ... }

    /* Start the conversation */
    public void receiveCall(Call call) { ... }

    /* the issue is resolved, finish the call */
    public void callCompleted() { ... }

    /* The issue has not been resolved. Escalate the call, and assign a new call to
     * the employee. */
    public void escalateAndReassign() { ... }

    /* Assign a new call to an employee, if the employee is free. */
    public boolean assignNewCall() { ... }

    /* Returns whether or not the employee is free. */
    public boolean isFree() { return currentCall == null; }

```

```
    public Rank getRank() { return rank; }
}

```

The Respondent, Director, and Manager classes are now just simple extensions of the Employee class.

```
class Director extends Employee {
   public Director() {
      rank = Rank.Director;
   }
}

class Manager extends Employee {
   public Manager() {
      rank = Rank.Manager;
   }
}

class Respondent extends Employee {
   public Respondent() {
      rank = Rank.Responder;
   }
}
```

This is just one way of designing this problem. Note that there are many other ways that are equally good.

This may seem like an awful lot of code to write in an interview, and it is. We've been much more thorough here than you would need. In a real interview, you would likely be much lighter on some of the details until you have time to fill them in.

7.3 **Jukebox:** Design a musical jukebox using object-oriented principles.

pg 127

SOLUTION

In any object-oriented design question, you first want to start off with asking your interviewer some questions to clarify design constraints. Is this jukebox playing CDs? Records? MP3s? Is it a simulation on a computer, or is it supposed to represent a physical jukebox? Does it take money, or is it free? And if it takes money, which currency? And does it deliver change?

Unfortunately, we don't have an interviewer here that we can have this dialogue with. Instead, we'll make some assumptions. We'll assume that the jukebox is a computer simulation that closely mirrors physical jukeboxes, and we'll assume that it's free.

Now that we have that out of the way, we'll outline the basic system components:

- Jukebox
- CD
- Song
- Artist
- Playlist
- Display (displays details on the screen)

Now, let's break this down further and think about the possible actions.

- Playlist creation (includes add, delete, and shuffle)
- CD selector
- Song selector
- Queuing up a song
- Get next song from playlist

A user also can be introduced:

- Adding
- Deleting
- Credit information

Each of the main system components translates roughly to an object, and each action translates to a method. Let's walk through one potential design.

The `Jukebox` class represents the body of the problem. Many of the interactions between the components of the system, or between the system and the user, are channeled through here.

```
public class Jukebox {
  private CDPlayer cdPlayer;
  private User user;
  private Set<CD> cdCollection;
  private SongSelector ts;

  public Jukebox(CDPlayer cdPlayer, User user, Set<CD> cdCollection,
                 SongSelector ts) { ... }

  public Song getCurrentSong() { return ts.getCurrentSong(); }
  public void setUser(User u) { this.user = u;}
}
```

Like a real CD player, the `CDPlayer` class supports storing just one CD at a time. The CDs that are not in play are stored in the jukebox.

```
public class CDPlayer {
  private Playlist p;
  private CD c;

  /* Constructors. */
  public CDPlayer(CD c, Playlist p) { ... }
  public CDPlayer(Playlist p) { this.p = p; }
  public CDPlayer(CD c) { this.c = c; }

  /* Play song */
  public void playSong(Song s) { ... }

  /* Getters and setters */
  public Playlist getPlaylist() { return p; }
  public void setPlaylist(Playlist p) { this.p = p; }

  public CD getCD() { return c; }
  public void setCD(CD c) { this.c = c; }
}
```

The `Playlist` manages the current and next songs to play. It is essentially a wrapper class for a queue and offers some additional methods for convenience.

```
public class Playlist {
  private Song song;
  private Queue<Song> queue;
  public Playlist(Song song, Queue<Song> queue) {
    ...
  }
  public Song getNextSToPlay() {
    return queue.peek();
  }
  public void queueUpSong(Song s) {
    queue.add(s);
  }
}
```

The classes for CD, Song, and User are all fairly straightforward. They consist mainly of member variables and getters and setters.

```
public class CD { /* data for id, artist, songs, etc */ }

public class Song { /* data for id, CD (could be null), title, length, etc */ }

public class User {
  private String name;
  public String getName() { return name; }
  public void setName(String name) {  this.name = name; }
  public long getID() { return ID; }
  public void setID(long iD) { ID = iD; }
  private long ID;
  public User(String name, long iD) { ... }
  public User getUser() { return this; }
  public static User addUser(String name, long iD) { ... }
}
```

This is by no means the only "correct" implementation. The interviewer's responses to initial questions, as well as other constraints, will shape the design of the jukebox classes.

7.4 Parking Lot: Design a parking lot using object-oriented principles.

pg 127

SOLUTION

The wording of this question is vague, just as it would be in an actual interview. This requires you to have a conversation with your interviewer about what types of vehicles it can support, whether the parking lot has multiple levels, and so on.

For our purposes right now, we'll make the following assumptions. We made these specific assumptions to add a bit of complexity to the problem without adding too much. If you made different assumptions, that's totally fine.

- The parking lot has multiple levels. Each level has multiple rows of spots.
- The parking lot can park motorcycles, cars, and buses.
- The parking lot has motorcycle spots, compact spots, and large spots.
- A motorcycle can park in any spot.
- A car can park in either a single compact spot or a single large spot.

- A bus can park in five large spots that are consecutive and within the same row. It cannot park in small spots.

In the below implementation, we have created an abstract class `Vehicle`, from which `Car`, `Bus`, and `Motorcycle` inherit. To handle the different parking spot sizes, we have just one class `ParkingSpot` which has a member variable indicating the size.

```
public enum VehicleSize { Motorcycle, Compact,    Large }

public abstract class Vehicle {
  protected ArrayList<ParkingSpot> parkingSpots = new ArrayList<ParkingSpot>();
  protected String licensePlate;
  protected int spotsNeeded;
  protected VehicleSize size;

  public int getSpotsNeeded() { return spotsNeeded; }
  public VehicleSize getSize() { return size; }

  /* Park vehicle in this spot (among others, potentially) */
  public void parkInSpot(ParkingSpot s) { parkingSpots.add(s); }

  /* Remove car from spot, and notify spot that it's gone */
  public void clearSpots() { ... }

  /* Checks if the spot is big enough for the vehicle (and is available). This
   * compares the SIZE only. It does not check if it has enough spots. */
  public abstract boolean canFitInSpot(ParkingSpot spot);
}

public class Bus extends Vehicle {
  public Bus() {
    spotsNeeded = 5;
    size = VehicleSize.Large;
  }

  /* Checks if the spot is a Large. Doesn't check num of spots */
  public boolean canFitInSpot(ParkingSpot spot) { ... }
}

public class Car extends Vehicle {
  public Car() {
    spotsNeeded = 1;
    size = VehicleSize.Compact;
  }

  /* Checks if the spot is a Compact or a Large. */
  public boolean canFitInSpot(ParkingSpot spot) { ... }
}

public class Motorcycle extends Vehicle {
  public Motorcycle() {
    spotsNeeded = 1;
    size = VehicleSize.Motorcycle;
  }

  public boolean canFitInSpot(ParkingSpot spot) { ... }
}
```

The `ParkingLot` class is essentially a wrapper class for an array of `Levels`. By implementing it this way, we are able to separate out logic that deals with actually finding free spots and parking cars out from the broader actions of the `ParkingLot`. If we didn't do it this way, we would need to hold parking spots in some sort of double array (or hash table which maps from a level number to the list of spots). It's cleaner to just separate `ParkingLot` from `Level`.

```
public class ParkingLot {
  private Level[] levels;
  private final int NUM_LEVELS = 5;

  public ParkingLot() { ... }

  /* Park the vehicle in a spot (or multiple spots). Return false if failed. */
  public boolean parkVehicle(Vehicle vehicle) { ... }
}

/* Represents a level in a parking garage */
public class Level {
  private int floor;
  private ParkingSpot[] spots;
  private int availableSpots = 0; // number of free spots
  private static final int SPOTS_PER_ROW = 10;

  public Level(int flr, int numberSpots) { ... }

  public int availableSpots() { return availableSpots; }

  /* Find a place to park this vehicle. Return false if failed. */
  public boolean parkVehicle(Vehicle vehicle) { ... }

  /* Park a vehicle starting at the spot spotNumber, and continuing until
   * vehicle.spotsNeeded. */
  private boolean parkStartingAtSpot(int num, Vehicle v) { ... }

  /* Find a spot to park this vehicle. Return index of spot, or -1 on failure. */
  private int findAvailableSpots(Vehicle vehicle) { ... }

  /* When a car was removed from the spot, increment availableSpots */
  public void spotFreed() { availableSpots++; }
}
```

The `ParkingSpot` is implemented by having just a variable which represents the size of the spot. We could have implemented this by having classes for `LargeSpot`, `CompactSpot`, and `MotorcycleSpot` which inherit from `ParkingSpot`, but this is probably overkill. The spots probably do not have different behaviors, other than their sizes.

```
public class ParkingSpot {
  private Vehicle vehicle;
  private VehicleSize spotSize;
  private int row;
  private int spotNumber;
  private Level level;

  public ParkingSpot(Level lvl, int r, int n, VehicleSize s) {...}

  public boolean isAvailable() { return vehicle == null; }

```

```
    /* Check if the spot is big enough and is available */
    public boolean canFitVehicle(Vehicle vehicle) { ... }

    /* Park vehicle in this spot. */
    public boolean park(Vehicle v) { ... }

    public int getRow() { return row; }
    public int getSpotNumber() { return spotNumber; }

    /* Remove vehicle from spot, and notify level that a new spot is available */
    public void removeVehicle() { ... }
}
```

A full implementation of this code, including executable test code, is provided in the downloadable code attachment.

7.5 Online Book Reader: Design the data structures for an online book reader system.

pg 127

SOLUTION

Since the problem doesn't describe much about the functionality, let's assume we want to design a basic online reading system which provides the following functionality:

- User membership creation and extension.
- Searching the database of books.
- Reading a book.
- Only one active user at a time
- Only one active book by this user.

To implement these operations we may require many other functions, like `get`, `set`, `update`, and so on. The objects required would likely include `User`, `Book`, and `Library`.

The class `OnlineReaderSystem` represents the body of our program. We could implement the class such that it stores information about all the books, deals with user management, and refreshes the display, but that would make this class rather hefty. Instead, we've chosen to tear off these components into `Library`, `UserManager`, and `Display` classes.

```
public class OnlineReaderSystem {
    private Library library;
    private UserManager userManager;
    private Display display;

    private Book activeBook;
    private User activeUser;

    public OnlineReaderSystem() {
        userManager = new UserManager();
        library = new Library();
        display = new Display();
    }

    public Library getLibrary() { return library;  }
    public UserManager getUserManager() { return userManager; }
```

```
  public Display getDisplay() { return display; }

  public Book getActiveBook() { return activeBook; }
  public void setActiveBook(Book book) {
    activeBook = book;
    display.displayBook(book);
  }

  public User getActiveUser() { return activeUser; }
  public void setActiveUser(User user) {
    activeUser = user;
    display.displayUser(user);
  }
}
```

We then implement separate classes to handle the user manager, the library, and the display components.

```
public class Library {
  private HashMap<Integer, Book> books;

  public Book addBook(int id, String details) {
    if (books.containsKey(id)) {
      return null;
    }
    Book book = new Book(id, details);
    books.put(id, book);
    return book;
  }

  public boolean remove(Book b) { return remove(b.getID()); }
  public boolean remove(int id) {
    if (!books.containsKey(id)) {
      return false;
    }
    books.remove(id);
    return true;
  }

  public Book find(int id) {
    return books.get(id);
  }
}

public class UserManager {
  private HashMap<Integer, User> users;

  public User addUser(int id, String details, int accountType) {
    if (users.containsKey(id)) {
      return null;
    }
    User user = new User(id, details, accountType);
    users.put(id, user);
    return user;
  }

  public User find(int id) { return users.get(id); }
  public boolean remove(User u) { return remove(u.getID()); }
  public boolean remove(int id) {
```

```
    if (!users.containsKey(id)) {
      return false;
    }
    users.remove(id);
    return true;
  }
}

public class Display {
  private Book activeBook;
  private User activeUser;
  private int pageNumber = 0;

  public void displayUser(User user) {
    activeUser = user;
    refreshUsername();
  }

  public void displayBook(Book book) {
    pageNumber = 0;
    activeBook = book;

    refreshTitle();
    refreshDetails();
    refreshPage();
  }

  public void turnPageForward() {
    pageNumber++;
    refreshPage();
  }

  public void turnPageBackward() {
    pageNumber--;
    refreshPage();
  }

  public void refreshUsername() { /* updates username display */ }
  public void refreshTitle() { /* updates title display */ }
  public void refreshDetails() { /* updates details display */   }
  public void refreshPage() { /* updated page display */ }
}
```

The classes for `User` and Book simply hold data and provide little true functionality.

```
public class Book {
  private int bookId;
  private String details;

  public Book(int id, String det) {
    bookId = id;
    details = det;
  }

  public int getID() { return bookId;  }
  public void setID(int id) { bookId = id; }
  public String getDetails() { return details; }
  public void setDetails(String d) { details = d; }
```

```
}

public class User {
   private int userId;
   private String details;
   private int accountType;

   public void renewMembership() {  }

   public User(int id, String details, int accountType) {
      userId = id;
      this.details = details;
      this.accountType = accountType;
   }

   /* Getters and setters */
   public int getID() { return userId; }
   public void setID(int id) { userId = id; }
   public String getDetails() {
      return details;
   }

   public void setDetails(String details) {
      this.details = details;
   }
   public int getAccountType() { return accountType; }
   public void setAccountType(int t) { accountType = t; }
}
```

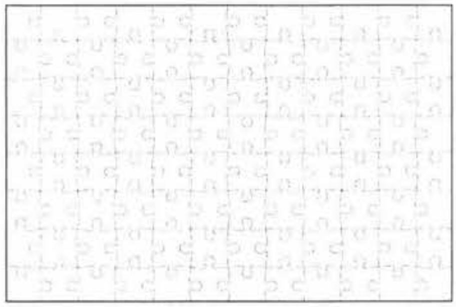
The decision to tear off user management, library, and display into their own classes, when this functionality could have been in the general `OnlineReaderSystem` class, is an interesting one. On a very small system, making this decision could make the system overly complex. However, as the system grows, and more and more functionality gets added to `OnlineReaderSystem`, breaking off such components prevents this main class from getting overwhelmingly lengthy.

7.6 **Jigsaw:** Implement an NxN jigsaw puzzle. Design the data structures and explain an algorithm to solve the puzzle. You can assume that you have a `fitsWith` method which, when passed two puzzle edges, returns true if the two edges belong together.

pg 128

SOLUTION

We have a traditional jigsaw puzzle. The puzzle is grid-like, with rows and columns. Each piece is located in a single row and column and has four edges. Each edge comes in one of three types: inner, outer, and flat. A corner piece, for example, will have two flat edges and two other edges, which could be inner or outer.



As we solve the jigsaw puzzle (manually or algorithmically), we'll need to store the position of each piece. We could think about the position as absolute or relative:

- *Absolute Position:* "This piece is located at position (12, 23)."
- *Relative Position:* "I don't know where this piece is actually located, but I know it is next to this other piece."

For our solution, we will use the absolute position.

We'll need classes to represent `Puzzle`, `Piece`, and `Edge`. Additionally, we'll want enums for the different shapes (`inner`, `outer`, `flat`) and the orientations of the edges (`left`, `top`, `right`, `bottom`).

`Puzzle` will start off with a list of the pieces. When we solve the puzzle, we'll fill in an NxN `solution` matrix of pieces.

`Piece` will have a hash table that maps from an orientation to the appropriate edge. Note that we might rotate the piece at some point, so the hash table could change. The orientation of the edges will be arbitrarily assigned at first.

`Edge` will have just its shape and a pointer back to its parent piece. It will not keep its orientation.

A potential object-oriented design looks like the following:

```
public enum Orientation {
  LEFT, TOP, RIGHT, BOTTOM; // Should stay in this order

  public Orientation getOpposite() {
    switch (this) {
      case LEFT: return RIGHT;
      case RIGHT: return LEFT;
      case TOP: return BOTTOM;
      case BOTTOM: return TOP;
      default: return null;
    }
  }
}

public enum Shape {
  INNER, OUTER, FLAT;

  public Shape getOpposite() {
    switch (this) {
      case INNER: return OUTER;
      case OUTER: return INNER;
      default: return null;
```

```
    }
  }
}

public class Puzzle {
  private LinkedList<Piece> pieces; /* Remaining pieces to put away. */
  private Piece[][] solution;
  private int size;

  public Puzzle(int size, LinkedList<Piece> pieces) { ... }


  /* Put piece into the solution, turn it appropriately, and remove from list. */
  private void setEdgeInSolution(LinkedList<Piece> pieces, Edge edge, int row,
                                 int column, Orientation orientation) {
    Piece piece = edge.getParentPiece();
    piece.setEdgeAsOrientation(edge, orientation);
    pieces.remove(piece);
    solution[row][column] = piece;
  }

  /* Find the matching piece in piecesToSearch and insert it at row, column. */
  private boolean fitNextEdge(LinkedList<Piece> piecesToSearch, int row, int col);

  /* Solve puzzle. */
  public boolean solve() { ... }
}

public class Piece {
  private HashMap<Orientation, Edge> edges = new HashMap<Orientation, Edge>();

  public Piece(Edge[] edgeList) { ... }

  /* Rotate edges by "numberRotations". */
  public void rotateEdgesBy(int numberRotations) { ... }

  public boolean isCorner() { ... }
  public boolean isBorder() { ... }
}

public class Edge {
  private Shape shape;
  private Piece parentPiece;
  public Edge(Shape shape) { ... }
  public boolean fitsWith(Edge edge) { ... }
}
```

Algorithm to Solve the Puzzle

Just as a kid might in solving a puzzle, we'll start with grouping the pieces into corner pieces, border pieces, and inside pieces.

Once we've done that, we'll pick an arbitrary corner piece and put it in the top left corner. We will then walk through the puzzle in order, filling in piece by piece. At each location, we search through the correct group of pieces to find the matching piece. When we insert the piece into the puzzle, we need to rotate the piece to fit correctly.

The code below outlines this algorithm.

```
/* Find the matching piece within piecesToSearch and insert it at row, column. */
boolean fitNextEdge(LinkedList<Piece> piecesToSearch, int row, int column) {
  if (row == 0 && column == 0) { // On top left corner, just put in a piece
    Piece p = piecesToSearch.remove();
    orientTopLeftCorner(p);
    solution[0][0] = p;
  } else {
    /* Get the right edge and list to match. */
    Piece pieceToMatch = column == 0 ? solution[row - 1][0] :
                                       solution[row][column - 1];
    Orientation orientationToMatch = column == 0 ? Orientation.BOTTOM :
                                                   Orientation.RIGHT;
    Edge edgeToMatch = pieceToMatch.getEdgeWithOrientation(orientationToMatch);

    /* Get matching edge. */
    Edge edge = getMatchingEdge(edgeToMatch, piecesToSearch);
    if (edge == null) return false; // Can't solve

    /* Insert piece and edge. */
    Orientation orientation = orientationToMatch.getOpposite();
    setEdgeInSolution(piecesToSearch, edge, row, column, orientation);
  }
  return true;
}

boolean solve() {
  /* Group pieces. */
  LinkedList<Piece> cornerPieces = new LinkedList<Piece>();
  LinkedList<Piece> borderPieces = new LinkedList<Piece>();
  LinkedList<Piece> insidePieces = new LinkedList<Piece>();
  groupPieces(cornerPieces, borderPieces, insidePieces);

  /* Walk through puzzle, finding the piece that joins the previous one. */
  solution = new Piece[size][size];
  for (int row = 0; row < size; row++) {
    for (int column = 0; column < size; column++) {
      LinkedList<Piece> piecesToSearch = getPieceListToSearch(cornerPieces,
        borderPieces, insidePieces, row, column);
      if (!fitNextEdge(piecesToSearch, row, column)) {
        return false;
      }
    }
  }

  return true;
}
```

The full code for this solution can be found in the downloadable code attachment.

7.7 **Chat Server:** Explain how you would design a chat server. In particular, provide details about the various backend components, classes, and methods. What would be the hardest problems to solve?

pg 128

SOLUTION

Designing a chat server is a huge project, and it is certainly far beyond the scope of what could be completed in an interview. After all, teams of many people spend months or years creating a chat server. Part of your job, as a candidate, is to focus on an aspect of the problem that is reasonably broad, but focused enough that you could accomplish it during an interview. It need not match real life exactly, but it should be a fair representation of an actual implementation.

For our purposes, we'll focus on the core user management and conversation aspects: adding a user, creating a conversation, updating one's status, and so on. In the interest of time and space, we will not go into the networking aspects of the problem, or how the data actually gets pushed out to the clients.

We will assume that "friending" is mutual; I am only your contact if you are mine. Our chat system will support both group chat and one-on-one (private) chats. We will not worry about voice chat, video chat, or file transfer.

What specific actions does it need to support?

This is also something to discuss with your interviewer, but here are some ideas:

- Signing online and offline.
- Add requests (sending, accepting, and rejecting).
- Updating a status message.
- Creating private and group chats.
- Adding new messages to private and group chats.

This is just a partial list. If you have more time, you can add more actions.

What can we learn about these requirements?

We must have a concept of users, add request status, online status, and messages.

What are the core components of the system?

The system would likely consist of a database, a set of clients, and a set of servers. We won't include these parts in our object-oriented design, but we can discuss the overall view of the system.

The database will be used for more permanent storage, such as the user list or chat archives. A SQL database is a good bet, or, if we need more scalability, we could potentially use BigTable or a similar system.

For communication between the client and servers, using XML will work well. Although it's not the most compressed format (and you should point this out to your interviewer), it's nice because it's easy for both computers and humans to read. Using XML will make your debugging efforts easier—and that matters a lot.

The server will consist of a set of machines. Data will be split across machines, requiring us to potentially hop from machine to machine. When possible, we will try to replicate some data across machines to minimize the lookups. One major design constraint here is to prevent having a single point of failure. For instance, if one machine controlled all the user sign-ins, then we'd cut off millions of users potentially if a single machine lost network connectivity.

What are the key objects and methods?

The key objects of the system will be a concept of users, conversations, and status messages. We've implemented a `UserManager` class. If we were looking more at the networking aspects of the problem, or a different component, we might have instead dived into those objects.

```
/* UserManager serves as a central place for core user actions. */
public class UserManager {
  private static UserManager instance;
  /* maps from a user id to a user */
  private HashMap<Integer, User> usersById;

  /* maps from an account name to a user */
  private HashMap<String, User> usersByAccountName;

  /* maps from the user id to an online user */
  private HashMap<Integer, User> onlineUsers;

  public static UserManager getInstance() {
    if (instance == null) instance = new UserManager();
    return instance;
  }

  public void addUser(User fromUser, String toAccountName) { ... }
  public void approveAddRequest(AddRequest req) { ... }
  public void rejectAddRequest(AddRequest req) { ... }
  public void userSignedOn(String accountName) { ... }
  public void userSignedOff(String accountName) { ... }
}
```

The method `receivedAddRequest`, in the `User` class, notifies User B that User A has requested to add him. User B approves or rejects the request (via `UserManager.approveAddRequest` or `rejectAddRequest`), and the `UserManager` takes care of adding the users to each other's contact lists.

The method `sentAddRequest` in the `User` class is called by `UserManager` to add an `AddRequest` to User A's list of requests. So the flow is:

1. User A clicks "add user" on the client, and it gets sent to the server.
2. User A calls `requestAddUser(User B)`.
3. This method calls `UserManager.addUser`.
4. `UserManager` calls both `User A.sentAddRequest` and `User B.receivedAddRequest`.

Again, this is just *one* way of designing these interactions. It is not the only way, or even the only "good" way.

```
public class User {
  private int id;
  private UserStatus status = null;

  /* maps from the other participant's user id to the chat */
  private HashMap<Integer, PrivateChat> privateChats;

  /* list of group chats */
```

```
    private ArrayList<GroupChat> groupChats;

    /* maps from the other person's user id to the add request */
    private HashMap<Integer, AddRequest> receivedAddRequests;

    /* maps from the other person's user id to the add request */
    private HashMap<Integer, AddRequest> sentAddRequests;

    /* maps from the user id to user object */
    private HashMap<Integer, User> contacts;

    private String accountName;
    private String fullName;

    public User(int id, String accountName, String fullName) { ... }
    public boolean sendMessageToUser(User to, String content){ ... }
    public boolean sendMessageToGroupChat(int id, String cnt){...}
    public void setStatus(UserStatus status) { ... }
    public UserStatus getStatus() { ... }
    public boolean addContact(User user) { ... }
    public void receivedAddRequest(AddRequest req) { ...}
    public void sentAddRequest(AddRequest req) { ... }
    public void removeAddRequest(AddRequest req) { ... }
    public void requestAddUser(String accountName) { ... }
    public void addConversation(PrivateChat conversation) { ... }
    public void addConversation(GroupChat conversation) { ... }
    public int getId() { ... }
    public String getAccountName() { ... }
    public String getFullName() { ... }
}
```

The `Conversation` class is implemented as an abstract class, since all `Conversations` must be either a `GroupChat` or a `PrivateChat`, and since these two classes each have their own functionality.

```
public abstract class Conversation {
    protected ArrayList<User> participants;
    protected int id;
    protected ArrayList<Message> messages;

    public ArrayList<Message> getMessages() { ... }
    public boolean addMessage(Message m) { ... }
    public int getId() { ... }
}

public class GroupChat extends Conversation {
    public void removeParticipant(User user) { ... }
    public void addParticipant(User user) { ... }
}

public class PrivateChat extends Conversation {
    public PrivateChat(User user1, User user2) { ...
    public User getOtherParticipant(User primary) { ... }
}

public class Message {
    private String content;
    private Date date;
    public Message(String content, Date date) { ... }
```

```
    public String getContent() { ... }
    public Date getDate() { ... }
}
```

`AddRequest` and `UserStatus` are simple classes with little functionality. Their main purpose is to group data that other classes will act upon.

```
public class AddRequest {
    private User fromUser;
    private User toUser;
    private Date date;
    RequestStatus status;

    public AddRequest(User from, User to, Date date) { ... }
    public RequestStatus getStatus() { ... }
    public User getFromUser() { ... }
    public User getToUser() { ... }
    public Date getDate() { ... }
}

public class UserStatus {
     private String message;
     private UserStatusType type;
     public UserStatus(UserStatusType type, String message) { ... }
     public UserStatusType getStatusType() { ... }
     public String getMessage() { ... }
}

public enum UserStatusType {
    Offline, Away, Idle, Available, Busy
}

public enum RequestStatus {
    Unread, Read, Accepted, Rejected
}
```

The downloadable code attachment provides a more detailed look at these methods, including implementations for the methods shown above.

What problems would be the hardest to solve (or the most interesting)?

The following questions may be interesting to discuss with your interviewer further.

Q1: How do we know if someone is online—I mean, really, really know?

While we would like users to tell us when they sign off, we can't know for sure. A user's connection might have died, for example. To make sure that we know when a user has signed off, we might try regularly pinging the client to make sure it's still there.

Q2: How do we deal with conflicting information?

We have some information stored in the computer's memory and some in the database. What happens if they get out of sync? Which one is "right"?

Q3: How do we make our server scale?

While we designed out chat server without worrying—too much– about scalability, in real life this would be a concern. We'd need to split our data across many servers, which would increase our concern about out-of-sync data.

Q4: How we do prevent denial of service attacks?

Clients can push data to us—what if they try to DOS (denial of service) us? How do we prevent that?

7.8 **Othello:** Othello is played as follows: Each Othello piece is white on one side and black on the other. When a piece is surrounded by its opponents on both the left and right sides, or both the top and bottom, it is said to be captured and its color is flipped. On your turn, you must capture at least one of your opponent's pieces. The game ends when either user has no more valid moves. The win is assigned to the person with the most pieces. Implement the object-oriented design for Othello.

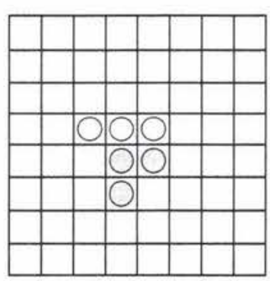
pg 128

SOLUTION

Let's start with an example. Suppose we have the following moves in an Othello game:

1. Initialize the board with two black and two white pieces in the center. The black pieces are placed at the upper left hand and lower right hand corners.
2. Play a black piece at (row 6, column 4). This flips the piece at (row 5, column 4) from white to black.
3. Play a white piece at (row 4, column 3). This flips the piece at (row 4, column 4) from black to white.

This sequence of moves leads to the board below.



The core objects in Othello are probably the game, the board, the pieces (black or white), and the players. How do we represent these with elegant object-oriented design?

Should BlackPiece and WhitePiece be classes?

At first, we might think we want to have a `BlackPiece` class and a `WhitePiece` class, which inherit from an abstract `Piece`. However, this is probably not a great idea. Each piece may flip back and forth between colors frequently, so continuously destroying and creating what is really the same object is probably not wise. It may be better to just have a `Piece` class, with a flag in it representing the current color.

Do we need separate Board and Game classes?

Strictly speaking, it may not be necessary to have both a `Game` object and a `Board` object. Keeping the objects separate allows us to have a logical separation between the board (which contains just logic involving placing pieces) and the game (which involves times, game flow, etc.). However, the drawback is that we are adding extra layers to our program. A function may call out to a method in Game, only to have it immediately call Board. We have made the choice below to keep Game and Board separate, but you should discuss this with your interviewer.

Who keeps score?

We know we should probably have some sort of score keeping for the number of black and white pieces. But who should maintain this information? One could make a strong argument for either Game or Board maintaining this information, and possibly even for Piece (in static methods). We have implemented this with Board holding this information, since it can be logically grouped with the board. It is updated by Piece or Board calling the colorChanged and colorAdded methods within Board.

Should Game be a Singleton class?

Implementing Game as a singleton class has the advantage of making it easy for anyone to call a method within Game, without having to pass around references to the Game object.

However, making Game a singleton means it can only be instantiated once. Can we make this assumption? You should discuss this with your interviewer.

One possible design for Othello is below.

```
public enum Direction {
  left, right, up, down
}

public enum Color {
  White, Black
}

public class Game {
  private Player[] players;
  private static Game instance;
  private Board board;
  private final int ROWS = 10;
  private final int COLUMNS = 10;

  private Game() {
    board = new Board(ROWS, COLUMNS);
    players = new Player[2];
    players[0] = new Player(Color.Black);
    players[1] = new Player(Color.White);
  }

  public static Game getInstance() {
    if (instance == null) instance = new Game();
    return instance;
  }

  public Board getBoard() {
    return board;
  }
}
```

The Board class manages the actual pieces themselves. It does not handle much of the game play, leaving that up to the Game class.

```
public class Board {
  private int blackCount = 0;
  private int whiteCount = 0;
  private Piece[][] board;

  public Board(int rows, int columns) {
    board = new Piece[rows][columns];
  }

  public void initialize() {
    /* initialize center black and white pieces */
  }

  /* Attempt to place a piece of color color at (row, column). Return true if we
   * were successful. */
  public boolean placeColor(int row, int column, Color color) {
    ...
  }

  /* Flips pieces starting at (row, column) and proceeding in direction d. */
  private int flipSection(int row, int column, Color color, Direction d) { ... }

  public int getScoreForColor(Color c) {
    if (c == Color.Black) return blackCount;
    else return whiteCount;
  }

  /* Update board with additional newPieces pieces of color newColor. Decrease
   * score of opposite color. */
  public void updateScore(Color newColor, int newPieces) { ... }
}
```

As described earlier, we implement the black and white pieces with the `Piece` class, which has a simple `Color` variable representing whether it is a black or white piece.

```
public class Piece {
  private Color color;
  public Piece(Color c) { color = c; }

  public void flip() {
    if (color == Color.Black) color = Color.White;
    else color = Color.Black;
  }

  public Color getColor() { return color; }
}
```

The `Player` holds only a very limited amount of information. It does not even hold its own score, but it does have a method one can call to get the score. `Player.getScore()` will call out to the Game object to retrieve this value.

```
public class Player {
  private Color color;
  public Player(Color c) { color = c; }

  public int getScore() { ...  }
```

```

    public boolean playPiece(int r, int c) {
        return Game.getInstance().getBoard().placeColor(r, c, color);
    }

    public Color getColor() { return color; }
}
```

A fully functioning (automated) version of this code can be found in the downloadable code attachment.

Remember that in many problems, what you did is less important than *why* you did it. Your interviewer probably doesn't care much whether you chose to implement Game as a singleton or not, but she probably does care that you took the time to think about it and discuss the trade-offs.

7.9 Circular Array: Implement a `CircularArray` class that supports an array-like data structure which can be efficiently rotated. If possible, the class should use a generic type (also called a template), and should support iteration via the standard `for (Obj o : circularArray)` notation.

pg 128

SOLUTION

This problem really has two parts to it. First, we need to implement the `CircularArray` class. Second, we need to support iteration. We will address these parts separately.

Implementing the CircularArray class

One way to implement the `CircularArray` class is to actually shift the elements each time we call `rotate(int shiftRight)`. Doing this is, of course, not very efficient.

Instead, we can just create a member variable `head` which points to what should be *conceptually* viewed as the start of the circular array. Rather than shifting around the elements in the array, we just increment `head` by `shiftRight`.

The code below implements this approach.

```
public class CircularArray<T> {
    private T[] items;
    private int head = 0;

    public CircularArray(int size) {
        items = (T[]) new Object[size];
    }

    private int convert(int index) {
        if (index < 0) {
            index += items.length;
        }
        return (head + index) % items.length;
    }

    public void rotate(int shiftRight) {
        head = convert(shiftRight);
    }

    public T get(int i) {
        if (i < 0 || i >= items.length) {
```

```
            throw new java.lang.IndexOutOfBoundsException("...");
        }
        return items[convert(i)];
    }

    public void set(int i, T item) {
        items[convert(i)] = item;
    }
}
```

There are a number of things here which are easy to make mistakes on, such as:

- In Java, we cannot create an array of the generic type. Instead, we must either cast the array or define `items` to be of type `List<T>`. For simplicity, we have done the former.
- The % operator will return a negative value when we do `negValue % posVal`. For example, `-8 % 3` is `-2`. This is different from how mathematicians would define the modulus function. We must add `items.length` to a negative index to get the correct positive result.
- We need to be sure to consistently convert the raw index to the rotated index. For this reason, we have implemented a `convert` function that is used by other methods. Even the `rotate` function uses `convert`. This is a good example of code reuse.

Now that we have the basic code for `CircularArray` out of the way, we can focus on implementing an iterator.

Implementing the Iterator Interface

The second part of this question asks us to implement the `CircularArray` class such that we can do the following:

```
CircularArray<String> array = ...
for (String s : array) { ... }
```

Implementing this requires implementing the `Iterator` interface. The details of this implementation apply to Java, but similar things can be implemented in other languages.

To implement the `Iterator` interface, we need to do the following:

- Modify the `CircularArray<T>` definition to add `implements Iterable<T>`. This will also require us to add an `iterator()` method to `CircularArray<T>`.
- Create a `CircularArrayIterator<T>` which implements `Iterator<T>`. This will also require us to implement, in the `CircularArrayIterator`, the methods `hasNext()`, `next()`, and `remove()`.

Once we've done the above items, the for loop will "magically" work.

In the code below, we have removed the aspects of `CircularArray` which were identical to the earlier implementation.

```
public class CircularArray<T> implements Iterable<T> {
    ...
    public Iterator<T> iterator() {
        return new CircularArrayIterator<T>(this);
    }

    private class CircularArrayIterator<TI> implements Iterator<TI>{
        /* current reflects the offset from the rotated head, not from the actual
         * start of the raw array. */
        private int _current = -1;
```

```
        private TI[] _items;

        public CircularArrayIterator(CircularArray<TI> array){
            _items = array.items;
        }

        @Override
        public boolean hasNext() {
            return _current < items.length - 1;
        }

        @Override
        public TI next() {
            _current++;
            TI item = (TI) _items[convert(_current)];
            return item;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException("...");
        }
    }
}
```

In the above code, note that the first iteration of the for loop will call `hasNext()` and then `next()`. Be very sure that your implementation will return the correct values here.

When you get a problem like this one in an interview, there's a good chance you don't remember exactly what the various methods and interfaces are called. In this case, work through the problem as well as you can. If you can reason out what sorts of methods one might need, that alone will show a good degree of competency.

7.10 **Minesweeper:** Design and implement a text-based Minesweeper game. Minesweeper is the classic single-player computer game where an NxN grid has B mines (or bombs) hidden across the grid. The remaining cells are either blank or have a number behind them. The numbers reflect the number of bombs in the surrounding eight cells. The user then uncovers a cell. If it is a bomb, the player loses. If it is a number, the number is exposed. If it is a blank cell, this cell and all adjacent blank cells (up to and including the surrounding numeric cells) are exposed. The player wins when all non-bomb cells are exposed. The player can also flag certain places as potential bombs. This doesn't affect game play, other than to block the user from accidentally clicking a cell that is thought to have a bomb. (Tip for the reader: if you're not familiar with this game, please play a few rounds online first.)

This is a fully exposed board with 3 bombs. This is not shown to the user.

	1	1	1			
	1	*	1			
	2	2	2			
	1	*	1			
	1	1	1			
			1	1	1	
			1	*	1	

The player initially sees a board with nothing exposed.

?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

Clicking on cell (row = 1, col = 0) would expose this:

	1	?	?	?	?	?
	1	?	?	?	?	?
	2	?	?	?	?	?
	1	?	?	?	?	?
	1	1	1	?	?	?
			1	?	?	?
			1	?	?	?

The user wins when everything other than bombs has been exposed.

	1	1	1			
	1	?	1			
	2	2	2			
	1	?	1			
	1	1	1			
			1	1	1	
			1	?	1	

pg 129

SOLUTION

Writing an entire game—even a text-based one—would take far longer than the allotted time you have in an interview. This doesn't mean that it's not fair game as a question. It just means that your interviewer's expectation will not be that you actually write all of this in an interview. It also means that you need to focus on getting the key ideas—or structure—out.

Let's start with what the classes are. We certainly want a `Cell` class as well as a `Board` class. We also probably want to have a `Game` class.

> We could potentially merge `Board` and `Game` together, but it's probably best to keep them separate. Err towards more organization, not less. `Board` can hold the list of `Cell` objects and do some basic moves with flipping over cells. `Game` will hold the game state and handle user input.

Design: Cell

Cell will need to have knowledge of whether it's a bomb, a number, or a blank. We could potentially subclass Cell to hold this data, but I'm not sure that offers us much benefit.

We could also have an enum TYPE {BOMB, NUMBER, BLANK} to describe the type of cell. We've chosen not to do this because BLANK is really a type of NUMBER cell, where the number is 0. It's sufficient to just have an isBomb flag.

It's okay to have made different choices here. These aren't the only good choices. Explain the choices you make and their tradeoffs with your interviewer.

We also need to store state for whether the cell is exposed or not. We probably do not want to subclass Cell for ExposedCell and UnexposedCell. This is a bad idea because Board holds a reference to the cells, and we'd have to change the reference when we flip a cell. And then what if other objects reference the instance of Cell?

It's better to just have a boolean flag for isExposed. We'll do a similar thing for isGuess.

```
public class Cell {
  private int row;
  private int column;
  private boolean isBomb;
  private int number;
  private boolean isExposed = false;
  private boolean isGuess = false;

  public Cell(int r, int c) { ... }

  /* Getters and setters for above variables. */
  ...

  public boolean flip() {
    isExposed = true;
    return !isBomb;
  }

  public boolean toggleGuess() {
    if (!isExposed) {
      isGuess = !isGuess;
    }
    return isGuess;
  }

  /* Full code can be found in downloadable code solutions. */
}
```

Design: Board

Board will need to have an array of all the Cell objects. A two-dimension array will work just fine.

We'll probably want Board to keep state of how many unexposed cells there are. We'll track this as we go, so we don't have to continuously count it.

Board will also handle some of the basic algorithms:

- Initializing the board and laying out the bombs.
- Flipping a cell.

- Expanding blank areas.

It will receive the game plays from the Game object and carry them out. It will then need to return the result of the play, which could be any of {clicked a bomb and lost, clicked out of bounds, clicked an already exposed area, clicked a blank area and still playing, clicked a blank area and won, clicked a number and won}. This is really two different items that need to be returned: successful (whether or not the play was successfully made) and a game state (won, lost, playing). We'll use an additional `GamePlayResult` to return this data.

We'll also use a `GamePlay` class to hold the move that the player plays. We need to use a row, column, and then a flag to indicate whether this was an actual flip or the user was just marking this as a "guess" at a possible bomb.

The basic skeleton of this class might look something like this:

```
public class Board {
  private int nRows;
  private int nColumns;
  private int nBombs = 0;
  private Cell[][] cells;
  private Cell[] bombs;
  private int numUnexposedRemaining;

  public Board(int r, int c, int b) { ... }

  private void initializeBoard() { ... }
  private boolean flipCell(Cell cell) { ... }
  public void expandBlank(Cell cell) { ... }
  public UserPlayResult playFlip(UserPlay play) { ... }
  public int getNumRemaining() { return numUnexposedRemaining; }
}

public class UserPlay {
  private int row;
  private int column;
  private boolean isGuess;
  /* constructor, getters, setters. */
}

public class UserPlayResult {
  private boolean successful;
  private Game.GameState resultingState;
  /* constructor, getters, setters. */
}
```

Design: Game

The Game class will store references to the board and hold the game state. It also takes the user input and sends it off to `Board`.

```
public class Game {
  public enum GameState { WON, LOST, RUNNING }

  private Board board;
  private int rows;
  private int columns;
  private int bombs;
  private GameState state;
```

```

    public Game(int r, int c, int b) { ... }

    public boolean initialize() { ... }
    public boolean start() { ... }
    private boolean playGame() { ... } // Loops until game is over.
}
```

Algorithms

This is the basic object-oriented design in our code. Our interviewer might ask us now to implement a few of the most interesting algorithms.

In this case, the three interesting algorithms is the initialization (placing the bombs randomly), setting the values of the numbered cells, and expanding the blank region.

Placing the Bombs

To place the bombs, we could randomly pick a cell and then place a bomb if it's still available, and otherwise pick a different location for it. The problem with this is that if there are a lot of bombs, it could get very slow. We could end up in a situation where we repeatedly pick cells with bombs.

To get around this, we could take an approach similar to the card deck shuffling problem (pg 531). We could place the K bombs in the first K cells and then shuffle all the cells around.

Shuffling an array operates by iterating through the array from `i = 0` through `N-1`. For each `i`, we pick a random index between `i` and `N-1` and swap it with that index.

To shuffle a grid, we do a very similar thing, just converting the index into a row and column location.

```
void shuffleBoard() {
   int nCells = nRows * nColumns;
   Random random = new Random();
   for (int index1 = 0; index1 < nCells; index1++) {
      int index2 = index1 + random.nextInt(nCells - index1);
      if (index1 != index2) {
         /* Get cell at index1. */
         int row1 = index1 / nColumns;
         int column1 = (index1 - row1 * nColumns) % nColumns;
         Cell cell1 = cells[row1][column1];

         /* Get cell at index2. */
         int row2 = index2 / nColumns;
         int column2 = (index2 - row2 * nColumns) % nColumns;
         Cell cell2 = cells[row2][column2];

         /* Swap. */
         cells[row1][column1] = cell2;
         cell2.setRowAndColumn(row1, column1);
         cells[row2][column2] = cell1;
         cell1.setRowAndColumn(row2, column2);
      }
   }
}
```

Setting the Numbered Cells

Once the bombs have been placed, we need to set the values of the numbered cells. We could go through each cell and check how many bombs are around it. This would work, but it's actually a bit slower than is necessary.

Instead, we can go to each bomb and increment each cell around it. For example, cells with 3 bombs will get `incrementNumber` called three times on them and will wind up with a number of 3.

```
/* Set the cells around the bombs to the right number. Although the bombs have
 * been shuffled, the reference in the bombs array is still to same object. */
void setNumberedCells() {
  int[][] deltas = { // Offsets of 8 surrounding cells
      {-1, -1}, {-1, 0}, {-1, 1},
      { 0, -1},          { 0, 1},
      { 1, -1}, { 1, 0}, { 1, 1}
  };
  for (Cell bomb : bombs) {
    int row = bomb.getRow();
    int col = bomb.getColumn();
    for (int[] delta : deltas) {
      int r = row + delta[0];
      int c = col + delta[1];
      if (inBounds(r, c)) {
        cells[r][c].incrementNumber();
      }
    }
  }
}
```

Expanding a Blank Region

Expanding the blank region could be done either iteratively or recursively. We implemented it iteratively.

You can think about this algorithm like this: each blank cell is surrounded by either blank cells or numbered cells (never a bomb). All need to be flipped. But, if you're flipping a blank cell, you also need to add the blank cells to a queue, to flip their neighboring cells.

```
void expandBlank(Cell cell) {
  int[][] deltas = {
      {-1, -1}, {-1, 0}, {-1, 1},
      { 0, -1},          { 0, 1},
      { 1, -1}, { 1, 0}, { 1, 1}
  };

  Queue<Cell> toExplore = new LinkedList<Cell>();
  toExplore.add(cell);

  while (!toExplore.isEmpty()) {
    Cell current = toExplore.remove();

    for (int[] delta : deltas) {
      int r = current.getRow() + delta[0];
      int c = current.getColumn() + delta[1];

      if (inBounds(r, c)) {
        Cell neighbor = cells[r][c];
        if (flipCell(neighbor) && neighbor.isBlank()) {
          toExplore.add(neighbor);
```

```
            }
         }
      }
   }
}
```

You could instead implement this algorithm recursively. In this algorithm, rather than adding the cell to a queue, you would make a recursive call.

Your implementation of these algorithms could vary substantially depending on your class design.

7.11 **File System:** Explain the data structures and algorithms that you would use to design an in-memory file system. Illustrate with an example in code where possible.

pg 129

SOLUTION

Many candidates may see this problem and instantly panic. A file system seems so low level!

However, there's no need to panic. If we think through the components of a file system, we can tackle this problem just like any other object-oriented design question.

A file system, in its most simplistic version, consists of `Files` and `Directories`. Each `Directory` contains a set of `Files` and `Directories`. Since `Files` and `Directories` share so many characteristics, we've implemented them such that they inherit from the same class, `Entry`.

```
public abstract class Entry {
   protected Directory parent;
   protected long created;
   protected long lastUpdated;
   protected long lastAccessed;
   protected String name;

   public Entry(String n, Directory p) {
      name = n;
      parent = p;
      created = System.currentTimeMillis();
      lastUpdated = System.currentTimeMillis();
      lastAccessed = System.currentTimeMillis();
   }

   public boolean delete() {
      if (parent == null) return false;
      return parent.deleteEntry(this);
   }

   public abstract int size();

   public String getFullPath() {
      if (parent == null) return name;
      else return parent.getFullPath() + "/" + name;
   }

   /* Getters and setters. */
   public long getCreationTime() { return created; }
   public long getLastUpdatedTime() { return lastUpdated; }
   public long getLastAccessedTime() { return lastAccessed; }
```

```java
  public void changeName(String n) { name = n; }
  public String getName() { return name; }
}

public class File extends Entry {
  private String content;
  private int size;

  public File(String n, Directory p, int sz) {
    super(n, p);
    size = sz;
  }

  public int size() { return size; }
  public String getContents() { return content; }
  public void setContents(String c) { content = c; }
}

public class Directory extends Entry {
  protected ArrayList<Entry> contents;

  public Directory(String n, Directory p) {
    super(n, p);
    contents = new ArrayList<Entry>();
  }

  public int size() {
    int size = 0;
    for (Entry e : contents) {
      size += e.size();
    }
    return size;
  }

  public int numberOfFiles() {
    int count = 0;
    for (Entry e : contents) {
      if (e instanceof Directory) {
        count++; // Directory counts as a file
        Directory d = (Directory) e;
        count += d.numberOfFiles();
      } else if (e instanceof File) {
        count++;
      }
    }
    return count;
  }

  public boolean deleteEntry(Entry entry) {
    return contents.remove(entry);
  }

  public void addEntry(Entry entry) {
    contents.add(entry);
  }

```

```
  protected ArrayList<Entry> getContents() { return contents; }
}
```

Alternatively, we could have implemented `Directory` such that it contains separate lists for files and subdirectories. This makes the `numberOfFiles()` method a bit cleaner, since it doesn't need to use the `instanceof` operator, but it does prohibit us from cleanly sorting files and directories by dates or names.

7.12 Hash Table: Design and implement a hash table which uses chaining (linked lists) to handle collisions.

pg 129

SOLUTION

Suppose we are implementing a hash table that looks like `Hash<K, V>`. That is, the hash table maps from objects of type K to objects of type V.

At first, we might think our data structure would look something like this:

```
class Hash<K, V> {
  LinkedList<V>[] items;
  public void put(K key, V value) { ... }
  public V get(K key) { ... }
}
```

Note that `items` is an array of linked lists, where `items[i]` is a linked list of all objects with keys that map to index `i` (that is, all the objects that collided at `i`).

This would seem to work until we think more deeply about collisions.

Suppose we have a very simple hash function that uses the string length.

```
int hashCodeOfKey(K key) {
  return key.toString().length() % items.length;
}
```

The keys `jim` and `bob` will map to the same index in the array, even though they are different keys. We need to search through the linked list to find the actual object that corresponds to these keys. But how would we do that? All we've stored in the linked list is the value, not the original key.

This is why we need to store both the value and the original key.

One way to do that is to create another object called `Cell` which pairs keys and values. With this implementation, our linked list is of type `Cell`.

The code below uses this implementation.

```
public class Hasher<K, V> {
  /* Linked list node class. Used only within hash table. No one else should get
   * access to this. Implemented as doubly linked list. */
  private static class LinkedListNode<K, V> {
    public LinkedListNode<K, V> next;
    public LinkedListNode<K, V> prev;
    public K key;
    public V value;
    public LinkedListNode(K k, V v) {
      key = k;
      value = v;
    }
  }

```

```
    private ArrayList<LinkedListNode<K, V>> arr;
    public Hasher(int capacity) {
       /* Create list of linked lists at a particular size. Fill list with null
        * values, as it's the only way to make the array the desired size. */
       arr = new ArrayList<LinkedListNode<K, V>>();
       arr.ensureCapacity(capacity); // Optional optimization
       for (int i = 0; i < capacity; i++) {
          arr.add(null);
       }
    }

    /* Insert key and value into hash table. */
    public void put(K key, V value) {
       LinkedListNode<K, V> node = getNodeForKey(key);
       if (node != null) { // Already there
          node.value = value; // just update the value.
          return;
       }

       node = new LinkedListNode<K, V>(key, value);
       int index = getIndexForKey(key);
       if (arr.get(index) != null) {
          node.next = arr.get(index);
          node.next.prev = node;
       }
       arr.set(index, node);
    }

    /* Remove node for key. */
    public void remove(K key) {
       LinkedListNode<K, V> node = getNodeForKey(key);
       if (node.prev != null) {
          node.prev.next = node.next;
       } else {
          /* Removing head - update. */
          int hashKey = getIndexForKey(key);
          arr.set(hashKey, node.next);
       }

       if (node.next != null) {
          node.next.prev = node.prev;
       }
    }

    /* Get value for key. */
    public V get(K key) {
       LinkedListNode<K, V> node = getNodeForKey(key);
       return node == null ? null : node.value;
    }

    /* Get linked list node associated with a given key. */
    private LinkedListNode<K, V> getNodeForKey(K key) {
       int index = getIndexForKey(key);
       LinkedListNode<K, V> current = arr.get(index);
       while (current != null) {
          if (current.key == key) {
```

```
                return current;
            }
            current = current.next;
        }
        return null;
    }

    /* Really naive function to map a key to an index. */
    public int getIndexForKey(K key) {
        return Math.abs(key.hashCode() % arr.size());
    }
}

```

Alternatively, we could implement a similar data structure (a key->value lookup) with a binary search tree as the underlying data structure. Retrieving an element will no longer be O(1) (although, technically, this implementation is not O(1) if there are many collisions), but it prevents us from creating an unnecessarily large array to hold items.

8

Solutions to Recursion and Dynamic Programming

8.1 Triple Step: A child is running up a staircase with n steps and can hop either 1 step, 2 steps, or 3 steps at a time. Implement a method to count how many possible ways the child can run up the stairs.

pg 134

SOLUTION

Let's think about this with the following question: What is the very last step that is done?

The very last hop the child makes—the one that lands her on the nth step—was either a 3-step hop, a 2-step hop, or a 1-step hop.

How many ways then are there to get up to the nth step? We don't know yet, but we can relate it to some subproblems.

If we thought about all of the paths to the nth step, we could just build them off the paths to the three previous steps. We can get up to the nth step by any of the following:

- Going to the (n-1)st step and hopping 1 step.
- Going to the (n-2)nd step and hopping 2 steps.
- Going to the (n-3)rd step and hopping 3 steps.

Therefore, we just need to add the number of these paths together.

Be very careful here. A lot of people want to multiply them. Multiplying one path with another would signify taking one path and then taking the other. That's not what's happening here.

Brute Force Solution

This is a fairly straightforward algorithm to implement recursively. We just need to follow logic like this:

```
countWays(n-1) + countWays(n-2) + countWays(n-3)
```

The one tricky bit is defining the base case. If we have 0 steps to go (we're currently standing on the step), are there zero paths to that step or one path?

That is, what is `countWays(0)`? Is it 1 or 0?

You could define it either way. There is no "right" answer here.

However, it's a lot easier to define it as 1. If you defined it as 0, then you would need some additional base cases (or else you'd just wind up with a series of 0s getting added).

A simple implementation of this code is below.

```
int countWays(int n) {
  if (n < 0) {
    return 0;
  } else if (n == 0) {
    return 1;
  } else {
    return countWays(n-1) + countWays(n-2) + countWays(n-3);
  }
}
```

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.

```
int countWays(int n) {
  int[] memo = new int[n + 1];
  Arrays.fill(memo, -1);
  return countWays(n, memo);
}

int countWays(int n, int[] memo) {
  if (n < 0) {
    return 0;
  } else if (n == 0) {
    return 1;
  } else if (memo[n] > -1) {
    return memo[n];
  } else {
    memo[n] = countWays(n - 1, memo) + countWays(n - 2, memo) +
              countWays(n - 3, memo);
    return memo[n];
  }
}
```

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.

8.2 Robot in a Grid: Imagine a robot sitting on the upper left corner of grid with r rows and c columns. The robot can only move in two directions, right and down, but certain cells are "off limits" such that the robot cannot step on them. Design an algorithm to find a path for the robot from the top left to the bottom right.

pg 135

SOLUTION

If we picture this grid, the only way to move to spot (r,c) is by moving to one of the adjacent spots: (r-1,c) or (r,c-1). So, we need to find a path to either (r-1,c) or (r,c-1).

How do we find a path to those spots? To find a path to (r-1,c) or (r,c-1), we need to move to one of its adjacent cells. So, we need to find a path to a spot adjacent to (r-1,c), which are coordinates (r-2,c) and (r-1,c-1), or a spot adjacent to (r,c-1), which are spots (r-1,c-1) and (r,c-2). Observe that we list the point (r-1,c-1) twice; we'll discuss that issue later.

> Tip: A lot of people use the variable names x and y when dealing with two-dimensional arrays. This can actually cause some bugs. People tend to think about x as the first coordinate in the matrix and y as the second coordinate (e.g., matrix[x][y]). But, this isn't really correct. The first coordinate is usually thought of as the row number, which is in fact the y value (it goes vertically!). You should write matrix[y][x]. Or, just make your life easier by using r (row) and c (column) instead.

So then, to find a path from the origin, we just work backwards like this. Starting from the last cell, we try to find a path to each of its adjacent cells. The recursive code below implements this algorithm.

```
ArrayList<Point> getPath(boolean[][] maze) {
   if (maze == null || maze.length == 0) return null;
   ArrayList<Point> path = new ArrayList<Point>();
   if (getPath(maze, maze.length - 1, maze[0].length - 1, path)) {
      return path;
   }
   return null;
}

boolean getPath(boolean[][] maze, int row, int col, ArrayList<Point> path) {
   /* If out of bounds or not available, return.*/
   if (col < 0 || row < 0 || !maze[row][col]) {
      return false;
   }

   boolean isAtOrigin = (row == 0) && (col == 0);

   /* If there's a path from the start to here, add my location. */
   if (isAtOrigin || getPath(maze, row, col - 1, path) ||
       getPath(maze, row - 1, col, path)) {
      Point p = new Point(row, col);
      path.add(p);
      return true;
   }

   return false;
}
```

This solution is $O(2^{r+c})$, since each path has r+c steps and there are two choices we can make at each step.

We should look for a faster way.

Often, we can optimize exponential algorithms by finding duplicate work. What work are we repeating?

If we walk through the algorithm, we'll see that we are visiting squares multiple times. In fact, we visit each square many, many times. After all, we have rc squares but we're doing $O(2^{r+c})$ work. If we were only visiting each square once, we would probably have an algorithm that was O(rc) (unless we were somehow doing a lot of work during each visit).

How does our current algorithm work? To find a path to (r,c), we look for a path to an adjacent coordinate: (r-1,c) or (r,c-1). Of course, if one of those squares is off limits, we ignore it. Then, we look at their adjacent coordinates: (r-2,c), (r-1,c-1), (r-1,c-1), and (r,c-2). The spot (r-1,c-1) appears twice, which means that we're duplicating effort. Ideally, we should remember that we already visited (r-1,c-1) so that we don't waste our time.

This is what the dynamic programming algorithm below does.

```
ArrayList<Point> getPath(boolean[][] maze) {
  if (maze == null || maze.length == 0) return null;
  ArrayList<Point> path = new ArrayList<Point>();
  HashSet<Point> failedPoints = new HashSet<Point>();
  if (getPath(maze, maze.length - 1, maze[0].length - 1, path, failedPoints)) {
    return path;
  }
  return null;
}

boolean getPath(boolean[][] maze, int row, int col, ArrayList<Point> path,
                HashSet<Point> failedPoints) {
  /* If out of bounds or not available, return.*/
  if (col < 0 || row < 0 || !maze[row][col]) {
    return false;
  }

  Point p = new Point(row, col);

  /* If we've already visited this cell, return. */
  if (failedPoints.contains(p)) {
    return false;
  }

  boolean isAtOrigin = (row == 0) && (col == 0);

  /* If there's a path from start to my current location, add my location.*/
  if (isAtOrigin || getPath(maze, row, col - 1, path, failedPoints) ||
      getPath(maze, row - 1, col, path, failedPoints)) {
    path.add(p);
    return true;
  }

  failedPoints.add(p); // Cache result
  return false;
}
```

This simple change will make our code run substantially faster. The algorithm will now take O(XY) time because we hit each cell just once.

8.3 **Magic Index:** A magic index in an array `A[1...n-1]` is defined to be an index such that `A[i] = i`. Given a sorted array of distinct integers, write a method to find a magic index, if one exists, in array A.

FOLLOW UP

What if the values are not distinct?

pg 135

SOLUTION

Immediately, the brute force solution should jump to mind—and there's no shame in mentioning it. We simply iterate through the array, looking for an element which matches this condition.

```
int magicSlow(int[] array) {
   for (int i = 0; i < array.length; i++) {
      if (array[i] == i) {
         return i;
      }
   }
   return -1;
}
```

Given that the array is sorted, though, it's very likely that we're supposed to use this condition.

We may recognize that this problem sounds a lot like the classic binary search problem. Leveraging the Pattern Matching approach for generating algorithms, how might we apply binary search here?

In binary search, we find an element k by comparing it to the middle element, x, and determining if k would land on the left or the right side of x.

Building off this approach, is there a way that we can look at the middle element to determine where a magic index might be? Let's look at a sample array:

-40	-20	-1	1	2	3	5	7	9	12	13
0	1	2	3	4	5	6	7	8	9	10

When we look at the middle element `A[5] = 3`, we know that the magic index must be on the right side, since `A[mid] < mid`.

Why couldn't the magic index be on the left side? Observe that when we move from `i` to `i-1`, the value at this index must decrease by at least 1, if not more (since the array is sorted and all the elements are distinct). So, if the middle element is already too small to be a magic index, then when we move to the left, subtracting k indexes and (at least) k values, all subsequent elements will also be too small.

We continue to apply this recursive algorithm, developing code that looks very much like binary search.

```
int magicFast(int[] array) {
   return magicFast(array, 0, array.length - 1);
}

int magicFast(int[] array, int start, int end) {
   if (end < start) {
      return -1;
   }
   int mid = (start + end) / 2;
   if (array[mid] == mid) {
      return mid;
   } else if (array[mid] > mid){
```

```
      return magicFast(array, start, mid - 1);
    } else {
      return magicFast(array, mid + 1, end);
    }
  }
```

Follow Up: What if the elements are not distinct?

If the elements are not distinct, then this algorithm fails. Consider the following array:

-10	-5	2	2	2	3	4	7	9	12	13
0	1	2	3	4	5	6	7	8	9	10

When we see that `A[mid] < mid`, we cannot conclude which side the magic index is on. It could be on the right side, as before. Or, it could be on the left side (as it, in fact, is).

Could it be *anywhere* on the left side? Not exactly. Since `A[5] = 3`, we know that `A[4]` couldn't be a magic index. `A[4]` would need to be 4 to be the magic index, but `A[4]` must be less than or equal to `A[5]`.

In fact, when we see that `A[5] = 3`, we'll need to recursively search the right side as before. But, to search the left side, we can skip a bunch of elements and only recursively search elements `A[0]` through `A[3]`. `A[3]` is the first element that could be a magic index.

The general pattern is that we compare `midIndex` and `midValue` for equality first. Then, if they are not equal, we recursively search the left and right sides as follows:

- Left side: search indices `start` through `Math.min(midIndex - 1, midValue)`.
- Right side: search indices `Math.max(midIndex + 1, midValue)` through end.

The code below implements this algorithm.

```
int magicFast(int[] array) {
  return magicFast(array, 0, array.length - 1);
}

int magicFast(int[] array, int start, int end) {
  if (end < start) return -1;

  int midIndex = (start + end) / 2;
  int midValue = array[midIndex];
  if (midValue == midIndex) {
    return midIndex;
  }

  /* Search left */
  int leftIndex = Math.min(midIndex - 1, midValue);
  int left = magicFast(array, start, leftIndex);
  if (left >= 0) {
    return left;
  }

  /* Search right */
  int rightIndex = Math.max(midIndex + 1, midValue);
  int right = magicFast(array, rightIndex, end);

  return right;
}
```

Note that in the above code, if the elements are all distinct, the method operates almost identically to the first solution.

8.4 Power Set: Write a method to return all subsets of a set.

pg 135

SOLUTION

We should first have some reasonable expectations of our time and space complexity.

How many subsets of a set are there? When we generate a subset, each element has the "choice" of either being in there or not. That is, for the first element, there are two choices: it is either in the set, or it is not. For the second, there are two, etc. So, doing `{2 * 2 * ... }` n times gives us 2^n subsets.

Assuming that we're going to be returning a list of subsets, then our best case time is actually the total number of elements across all of those subsets. There are 2^n subsets and each of the n elements will be contained in half of the subsets (which 2^{n-1} subsets). Therefore, the total number of elements across all of those subsets is $n * 2^{n-1}$.

We will not be able to beat $O(n2^n)$ in space or time complexity.

The subsets of $\{a_1, a_2, \ldots, a_n\}$ are also called the powerset, $P(\{a_1, a_2, \ldots, a_n\})$, or just $P(n)$.

Solution #1: Recursion

This problem is a good candidate for the Base Case and Build approach. Imagine that we are trying to find all subsets of a set like $S = \{a_1, a_2, \ldots, a_n\}$. We can start with the Base Case.

Base Case: n = 0.

There is just one subset of the empty set: {}.

Case: n = 1.

There are two subsets of the set $\{a_1\}$: $\{\}$, $\{a_1\}$.

Case: n = 2.

There are four subsets of the set $\{a_1, a_2\}$: $\{\}$, $\{a_1\}$, $\{a_2\}$, $\{a_1, a_2\}$.

Case: n = 3.

Now here's where things get interesting. We want to find a way of generating the solution for n = 3 based on the prior solutions.

What is the difference between the solution for n = 3 and the solution for n = 2? Let's look at this more deeply:

$$P(2) = \{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$$
$$P(3) = \{\}, \{a_1\}, \{a_2\}, \{a_3\}, \{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

The difference between these solutions is that $P(2)$ is missing all the subsets containing a_3.

$$P(3) - P(2) = \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

How can we use $P(2)$ to create $P(3)$? We can simply clone the subsets in $P(2)$ and add a_3 to them:

$$P(2) = \{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$$
$$P(2) + a_3 = \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

When merged together, the lines above make P(3).

Case: n > 0

Generating P(n) for the general case is just a simple generalization of the above steps. We compute P(n-1), clone the results, and then add a_n to each of these cloned sets.

The following code implements this algorithm:

```
ArrayList<ArrayList<Integer>> getSubsets(ArrayList<Integer> set, int index) {
  ArrayList<ArrayList<Integer>> allsubsets;
  if (set.size() == index) { // Base case - add empty set
    allsubsets = new ArrayList<ArrayList<Integer>>();
    allsubsets.add(new ArrayList<Integer>()); // Empty set
  } else {
    allsubsets = getSubsets(set, index + 1);
    int item = set.get(index);
    ArrayList<ArrayList<Integer>> moresubsets =
      new ArrayList<ArrayList<Integer>>();
    for (ArrayList<Integer> subset : allsubsets) {
      ArrayList<Integer> newsubset = new ArrayList<Integer>();
      newsubset.addAll(subset); //
      newsubset.add(item);
      moresubsets.add(newsubset);
    }
    allsubsets.addAll(moresubsets);
  }
  return allsubsets;
}
```

This solution will be $O(n2^n)$ in time and space, which is the best we can do. For a slight optimization, we could also implement this algorithm iteratively.

Solution #2: Combinatorics

While there's nothing wrong with the above solution, there's another way to approach it.

Recall that when we're generating a set, we have two choices for each element: (1) the element is in the set (the "yes" state) or (2) the element is not in the set (the "no" state). This means that each subset is a sequence of yeses / nos—e.g., "yes, yes, no, no, yes, no"

This gives us 2^n possible subsets. How can we iterate through all possible sequences of "yes" / "no" states for all elements? If each "yes" can be treated as a 1 and each "no" can be treated as a 0, then each subset can be represented as a binary string.

Generating all subsets, then, really just comes down to generating all binary numbers (that is, all integers). We iterate through all numbers from 0 to 2^n (exclusive) and translate the binary representation of the numbers into a set. Easy!

```
ArrayList<ArrayList<Integer>> getSubsets2(ArrayList<Integer> set) {
  ArrayList<ArrayList<Integer>> allsubsets = new ArrayList<ArrayList<Integer>>();
  int max = 1 << set.size(); /* Compute 2^n */
  for (int k = 0; k < max; k++) {
    ArrayList<Integer> subset = convertIntToSet(k, set);
    allsubsets.add(subset);
  }
  return allsubsets;
}
```

```

ArrayList<Integer> convertIntToSet(int x, ArrayList<Integer> set) {
  ArrayList<Integer> subset = new ArrayList<Integer>();
  int index = 0;
  for (int k = x; k > 0; k >>= 1) {
    if ((k & 1) == 1) {
      subset.add(set.get(index));
    }
    index++;
  }
  return subset;
}
```

There's nothing substantially better or worse about this solution compared to the first one.

8.5 **Recursive Multiply:** Write a recursive function to multiply two positive integers without using the * operator (or / operator). You can use addition, subtraction, and bit shifting, but you should minimize the number of those operations.

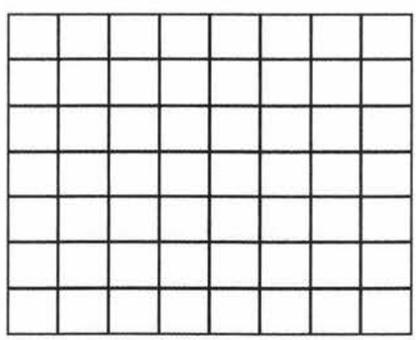
pg 135

SOLUTION

Let's pause for a moment and think about what it means to do multiplication.

> This is a good approach for a lot of interview questions. It's often useful to think about what it really means to do something, even when it's pretty obvious.

We can think about multiplying 8x7 as doing 8+8+8+8+8+8+8 (or adding 7 eight times). We can also think about it as the number of squares in an 8x7 grid.



Solution #1

How would we count the number of squares in this grid? We could just count each cell. That's pretty slow, though.

Alternatively, we could count half the squares and then double it (by adding this count to itself). To count half the squares, we repeat the same process.

Of course, this "doubling" only works if the number is in fact even. When it's not even, we need to do the counting/summing from scratch.

```
int minProduct(int a, int b) {
  int bigger = a < b ? b : a;
```

```
  int smaller = a < b ? a : b;
  return minProductHelper(smaller, bigger);
}

int minProductHelper(int smaller, int bigger) {
  if (smaller == 0) { // 0 x bigger = 0
    return 0;
  } else if (smaller == 1) { // 1 x bigger = bigger
    return bigger;
  }

  /* Compute half. If uneven, compute other half. If even, double it. */
  int s = smaller >> 1; // Divide by 2
  int side1 = minProduct(s, bigger);
  int side2 = side1;
  if (smaller % 2 == 1) {
    side2 = minProductHelper(smaller - s, bigger);
  }

  return side1 + side2;
}
```

Can we do better? Yes.

Solution #2

If we observe how the recursion operates, we'll notice that we have duplicated work. Consider this example:

```
minProduct(17, 23)
      minProduct(8, 23)
          minProduct(4, 23) * 2
              ...
    + minProduct(9, 23)
          minProduct(4, 23)
              ...
        + minProduct(5, 23)
              ...
```

The second call to `minProduct(4, 23)` is unaware of the prior call, and so it repeats the same work. We should cache these results.

```
int minProduct(int a, int b) {
  int bigger = a < b ? b : a;
  int smaller = a < b ? a : b;

  int memo[] = new int[smaller + 1];
  return minProduct(smaller, bigger, memo);
}

int minProduct(int smaller, int bigger, int[] memo) {
  if (smaller == 0) {
    return 0;
  } else if (smaller == 1) {
    return bigger;
  } else if (memo[smaller] > 0) {
    return memo[smaller];
  }

  /* Compute half. If uneven, compute other half. If even, double it. */
```

```
    int s = smaller >> 1; // Divide by 2
    int side1 = minProduct(s, bigger, memo); // Compute half
    int side2 = side1;
    if (smaller % 2 == 1) {
       side2 = minProduct(smaller - s, bigger, memo);
    }

    /* Sum and cache.*/
    memo[smaller] = side1 + side2;
    return memo[smaller];
 }
```

We can still make this a bit faster.

Solution #3

One thing we might notice when we look at this code is that a call to `minProduct` on an even number is much faster than one on an odd number. For example, if we call `minProduct(30, 35)`, then we'll just do `minProduct(15, 35)` and double the result. However, if we do `minProduct(31, 35)`, then we'll need to call `minProduct(15, 35)` and `minProduct(16, 35)`.

This is unnecessary. Instead, we can do:

```
minProduct(31, 35) = 2 * minProduct(15, 35) + 35
```

After all, since `31 = 2*15+1`, then `31x35 = 2*15*35+35`.

The logic in this final solution is that, on even numbers, we just divide `smaller` by 2 and double the result of the recursive call. On odd numbers, we do the same, but then we also add `bigger` to this result.

In doing so, we have an unexpected "win." Our `minProduct` function just recurses straight downwards, with increasingly small numbers each time. It will never repeat the same call, so there's no need to cache any information.

```
int minProduct(int a, int b) {
   int bigger = a < b ? b : a;
   int smaller = a < b ? a : b;
   return minProductHelper(smaller, bigger);
}

int minProductHelper(int smaller, int bigger) {
   if (smaller == 0) return 0;
   else if (smaller == 1) return bigger;

   int s = smaller >> 1; // Divide by 2
   int halfProd = minProductHelper(s, bigger);

   if (smaller % 2 == 0) {
      return halfProd + halfProd;
   } else {
      return halfProd + halfProd + bigger;
   }
}
```

This algorithm will run in `O(log s)` time, where `s` is the smaller of the two numbers.

8.6 **Towers of Hanoi:** In the classic problem of the Towers of Hanoi, you have 3 towers and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e., each disk sits on top of an even larger one). You have the following constraints:

(1) Only one disk can be moved at a time.

(2) A disk is slid off the top of one tower onto another tower.

(3) A disk cannot be placed on top of a smaller disk.

Write a program to move the disks from the first tower to the last using Stacks.

pg 135

SOLUTION

This problem sounds like a good candidate for the Base Case and Build approach.

Let's start with the smallest possible example: `n = 1`.

Case `n = 1`. Can we move Disk 1 from Tower 1 to Tower 3? Yes.

1. We simply move Disk 1 from Tower 1 to Tower 3.

Case `n = 2`. Can we move Disk 1 and Disk 2 from Tower 1 to Tower 3? Yes.

1. Move Disk 1 from Tower 1 to Tower 2
2. Move Disk 2 from Tower 1 to Tower 3
3. Move Disk 1 from Tower 2 to Tower 3

Note how in the above steps, Tower 2 acts as a buffer, holding a disk while we move other disks to Tower 3.

Case `n = 3`. Can we move Disk 1, 2, and 3 from Tower 1 to Tower 3? Yes.

1. We know we can move the top two disks from one tower to another (as shown earlier), so let's assume we've already done that. But instead, let's move them to Tower 2.
2. Move Disk 3 to Tower 3.
3. Move Disk 1 and Disk 2 to Tower 3. We already know how to do this—just repeat what we did in Step 1.

Case `n = 4`. Can we move Disk 1, 2, 3 and 4 from Tower 1 to Tower 3? Yes.

1. Move Disks 1, 2, and 3 to Tower 2. We know how to do that from the earlier examples.
2. Move Disk 4 to Tower 3.
3. Move Disks 1, 2 and 3 back to Tower 3.

Remember that the labels of Tower 2 and Tower 3 aren't important. They're equivalent towers. So, moving disks to Tower 3 with Tower 2 serving as a buffer is equivalent to moving disks to Tower 2 with Tower 3 serving as a buffer.

This approach leads to a natural recursive algorithm. In each part, we are doing the following steps, outlined below with pseudocode:

```
moveDisks(int n, Tower origin, Tower destination, Tower buffer) {
  /* Base case */
  if (n <= 0) return;

  /* move top n - 1 disks from origin to buffer, using destination as a buffer. */
  moveDisks(n - 1, origin, buffer, destination);

  /* move top from origin to destination
  moveTop(origin, destination);

  /* move top n - 1 disks from buffer to destination, using origin as a buffer. */
  moveDisks(n - 1, buffer, destination, origin);
}
```

The following code provides a more detailed implementation of this algorithm, using concepts of object-oriented design.

```
void main(String[] args) {
  int n = 3;
  Tower[] towers = new Tower[n];
  for (int i = 0; i < 3; i++) {
    towers[i] = new Tower(i);
  }

  for (int i = n - 1; i >= 0; i--) {
    towers[0].add(i);
  }
  towers[0].moveDisks(n, towers[2], towers[1]);
}

class Tower {
  private Stack<Integer> disks;
  private int index;
  public Tower(int i) {
    disks = new Stack<Integer>();
    index = i;
  }

  public int index() {
    return index;
  }

  public void add(int d) {
    if (!disks.isEmpty() && disks.peek() <= d) {
      System.out.println("Error placing disk " + d);
    } else {
      disks.push(d);
    }
  }

  public void moveTopTo(Tower t) {
    int top = disks.pop();
    t.add(top);
  }

```

```
  public void moveDisks(int n, Tower destination, Tower buffer) {
    if (n > 0) {
      moveDisks(n - 1, buffer, destination);
      moveTopTo(destination);
      buffer.moveDisks(n - 1, destination, this);
    }
  }
}
```

Implementing the towers as their own objects is not strictly necessary, but it does help to make the code cleaner in some respects.

8.7 Permutations without Dups: Write a method to compute all permutations of a string of unique characters.

pg 135

SOLUTION

Like in many recursive problems, the Base Case and Build approach will be useful. Assume we have a string S represented by the characters $a_1a_2...a_n$.

Approach 1: Building from permutations of first n-1 characters.

Base Case: permutations of first character substring

The only permutation of a_1 is the string a_1. So:

$P(a_1) = a_1$

Case: permutations of a_1a_2

$P(a_1a_2) = a_1a_2$ and a_2a_1

Case: permutations of $a_1a_2a_3$

$P(a_1a_2a_3) = a_1a_2a_3,\ a_1a_3a_2,\ a_2a_1a_3,\ a_2a_3a_1,\ a_3a_1a_2,\ a_3a_2a_1,$

Case: permutations of $a_1a_2a_3a_4$

This is the first interesting case. How can we generate permutations of $a_1a_2a_3a_4$ from $a_1a_2a_3$?

Each permutation of $a_1a_2a_3a_4$ represents an ordering of $a_1a_2a_3$. For example, $a_2a_4a_1a_3$ represents the order $a_2a_1a_3$.

Therefore, if we took all the permutations of $a_1a_2a_3$ and added a_4 into all possible locations, we would get all permutations of $a_1a_2a_3a_4$.

$a_1a_2a_3 \text{ -> } a_4a_1a_2a_3,\ a_1a_4a_2a_3,\ a_1a_2a_4a_3,\ a_1a_2a_3a_4$
$a_1a_3a_2 \text{ -> } a_4a_1a_3a_2,\ a_1a_4a_3a_2,\ a_1a_3a_4a_2,\ a_1a_3a_2a_4$
$a_3a_1a_2 \text{ -> } a_4a_3a_1a_2,\ a_3a_4a_1a_2,\ a_3a_1a_4a_2,\ a_3a_1a_2a_4$
$a_2a_1a_3 \text{ -> } a_4a_2a_1a_3,\ a_2a_4a_1a_3,\ a_2a_1a_4a_3,\ a_2a_1a_3a_4$
$a_2a_3a_1 \text{ -> } a_4a_2a_3a_1,\ a_2a_4a_3a_1,\ a_2a_3a_4a_1,\ a_2a_3a_1a_4$
$a_3a_2a_1 \text{ -> } a_4a_3a_2a_1,\ a_3a_4a_2a_1,\ a_3a_2a_4a_1,\ a_3a_2a_1a_4$

We can now implement this algorithm recursively.

```
ArrayList<String> getPerms(String str) {
  if (str == null) return null;

  ArrayList<String> permutations = new ArrayList<String>();
  if (str.length() == 0) { // base case
    permutations.add("");
```

```
        return permutations;
    }

    char first = str.charAt(0); // get the first char
    String remainder = str.substring(1); // remove the first char
    ArrayList<String> words = getPerms(remainder);
    for (String word : words) {
        for (int j = 0; j <= word.length(); j++) {
            String s = insertCharAt(word, first, j);
            permutations.add(s);
        }
    }
    return permutations;
}

/* Insert char c at index i in word. */
String insertCharAt(String word, char c, int i) {
    String start = word.substring(0, i);
    String end = word.substring(i);
    return start + c + end;
}
```

Approach 2: Building from permutations of all n-1 character substrings.

Base Case: single-character strings

The only permutation of a_1 is the string a_1. So:

$$P(a_1) = a_1$$

Case: two-character strings

$$P(a_1a_2) = a_1a_2 \text{ and } a_2a_1.$$
$$P(a_2a_3) = a_2a_3 \text{ and } a_3a_2.$$
$$P(a_1a_3) = a_1a_3 \text{ and } a_3a_1.$$

Case: three-character strings

Here is where the cases get more interesting. How can we generate all permutations of three-character strings, such as $a_1a_2a_3$, given the permutations of two-character strings?

Well, in essence, we just need to "try" each character as the first character and then append the permutations.

$$P(a_1a_2a_3) = \{a_1 + P(a_2a_3)\} + a_2 + P(a_1a_3)\} + \{a_3 + P(a_1a_2)\}$$
$$\{a_1 + P(a_2a_3)\} \text{ -> } a_1a_2a_3,\ a_1a_3a_2$$
$$\{a_2 + P(a_1a_3)\} \text{ -> } a_2a_1a_3,\ a_2a_3a_1$$
$$\{a_3 + P(a_1a_2)\} \text{ -> } a_3a_1a_2,\ a_3a_2a_1$$

Now that we can generate all permutations of three-character strings, we can use this to generate permutations of four-character strings.

$$P(a_1a_2a_3a_4) = \{a_1 + P(a_2a_3a_4)\} + \{a_2 + P(a_1a_3a_4)\} + \{a_3 + P(a_1a_2a_4)\} + \{a_4 + P(a_1a_2a_3)\}$$

This is now a fairly straightforward algorithm to implement.

```
ArrayList<String> getPerms(String remainder) {
    int len = remainder.length();
    ArrayList<String> result = new ArrayList<String>();

    /* Base case. */
```

```
    if (len == 0) {
      result.add(""); // Be sure to return empty string!
      return result;
    }


    for (int i = 0; i < len; i++) {
      /* Remove char i and find permutations of remaining chars.*/
      String before = remainder.substring(0, i);
      String after = remainder.substring(i + 1, len);
      ArrayList<String> partials = getPerms(before + after);

      /* Prepend char i to each permutation.*/
      for (String s : partials) {
        result.add(remainder.charAt(i) + s);
      }
    }

    return result;
  }
```

Alternatively, instead of passing the permutations back up the stack, we can push the prefix down the stack. When we get to the bottom (base case), `prefix` holds a full permutation.

```
  ArrayList<String> getPerms(String str) {
    ArrayList<String> result = new ArrayList<String>();
    getPerms("", str, result);
    return result;
  }

  void getPerms(String prefix, String remainder, ArrayList<String> result) {
    if (remainder.length() == 0) result.add(prefix);

    int len = remainder.length();
    for (int i = 0; i < len; i++) {
      String before = remainder.substring(0, i);
      String after = remainder.substring(i + 1, len);
      char c = remainder.charAt(i);
      getPerms(prefix + c, before + after, result);
    }
  }
```

For a discussion of the runtime of this algorithm, see Example 12 on page 51.

8.8 Permutations with Duplicates: Write a method to compute all permutations of a string whose characters are not necessarily unique. The list of permutations should not have duplicates.

pg 135

SOLUTION

This is very similar to the previous problem, except that now we could potentially have duplicate characters in the word.

One simple way of handling this problem is to do the same work to check if a permutation has been created before and then, if not, add it to the list. A simple hash table will do the trick here. This solution will take `O(n!)` time in the worst case (and, in fact, in all cases).

While it's true that we can't beat this worst case time, we should be able to design an algorithm to beat this in many cases. Consider a string with all duplicate characters, like aaaaaaaaaaaaa. This will take an extremely long time (since there are over 6 billion permutations of a 13-character string), even though there is only one unique permutation.

Ideally, we would like to only create the unique permutations, rather than creating every permutation and then ruling out the duplicates.

We can start with computing the count of each letter (easy enough to get this—just use a hash table). For a string such as aabbbbc, this would be:

```
a->2 | b->4 | c->1
```

Let's imagine generating a permutation of this string (now represented as a hash table). The first choice we make is whether to use an a, b, or c as the first character. After that, we have a subproblem to solve: find all permutations of the remaining characters, and append those to the already picked "prefix."

```
P(a->2 | b->4 | c->1) = {a + P(a->1 | b->4 | c->1)} +
                        {b + P(a->2 | b->3 | c->1)} +
                        {c + P(a->2 | b->4 | c->0)}
    P(a->1 | b->4 | c->1) = {a + P(a->0 | b->4 | c->1)} +
                            {b + P(a->1 | b->3 | c->1)} +
                            {c + P(a->1 | b->4 | c->0)}
    P(a->2 | b->3 | c->1) = {a + P(a->1 | b->3 | c->1)} +
                            {b + P(a->2 | b->2 | c->1)} +
                            {c + P(a->2 | b->3 | c->0)}
    P(a->2 | b->4 | c->0) = {a + P(a->1 | b->4 | c->0)} +
                            {b + P(a->2 | b->3 | c->0)}
```

Eventually, we'll get down to no more characters remaining.

The code below implements this algorithm.

```
ArrayList<String> printPerms(String s) {
  ArrayList<String> result = new ArrayList<String>();
  HashMap<Character, Integer> map = buildFreqTable(s);
  printPerms(map, “”, s.length(), result);
  return result;
}

HashMap<Character, Integer> buildFreqTable(String s) {
  HashMap<Character, Integer> map = new HashMap<Character, Integer>();
  for (char c : s.toCharArray()) {
    if (!map.containsKey(c)) {
      map.put(c, 0);
    }
    map.put(c, map.get(c) + 1);
  }
  return map;
}

void printPerms(HashMap<Character, Integer> map, String prefix, int remaining,
                ArrayList<String> result) {
  /* Base case. Permutation has been completed. */
  if (remaining == 0) {
    result.add(prefix);
    return;
  }

  /* Try remaining letters for next char, and generate remaining permutations. */
```

```
    for (Character c : map.keySet()) {
      int count = map.get(c);
      if (count > 0) {
        map.put(c,  count - 1);
        printPerms(map, prefix + c, remaining - 1, result);
        map.put(c,  count);
      }
    }
}
```

In situations where the string has many duplicates, this algorithm will run a lot faster than the earlier algorithm.

8.9 Parens: Implement an algorithm to print all valid (i.e., properly opened and closed) combinations of n pairs of parentheses.

EXAMPLE

Input: 3

Output: `((()))`, `(()())`, `(())()`, `()(())`, `()()()`

pg 136

SOLUTION

Our first thought here might be to apply a recursive approach where we build the solution for `f(n)` by adding pairs of parentheses to `f(n-1)`. That's certainly a good instinct.

Let's consider the solution for `n = 3`:

```
(()())    ((()))    ()(())    (())()    ()()()
```

How might we build this from `n = 2`?

```
(())    ()()
```

We can do this by inserting a pair of parentheses inside every existing pair of parentheses, as well as one at the beginning of the string. Any other places that we could insert parentheses, such as at the end of the string, would reduce to the earlier cases.

So, we have the following:

```
(()) -> (()()) /* inserted pair after 1st left paren */
     -> ((())) /* inserted pair after 2nd left paren */
     -> ()(()) /* inserted pair at beginning of string */
()() -> (())() /* inserted pair after 1st left paren */
     -> ()(()) /* inserted pair after 2nd left paren */
     -> ()()() /* inserted pair at beginning of string */
```

But wait—we have some duplicate pairs listed. The string `()(())` is listed twice.

If we're going to apply this approach, we'll need to check for duplicate values before adding a string to our list.

```
Set<String> generateParens(int remaining) {
  Set<String> set = new HashSet<String>();
  if (remaining == 0) {
    set.add("");
  } else {
    Set<String> prev = generateParens(remaining - 1);
    for (String str : prev) {
      for (int i = 0; i < str.length(); i++) {
```

```
            if (str.charAt(i) == '(') {
               String s = insertInside(str, i);
               /* Add s to set if it's not already in there. Note:  HashSet
                * automatically checks for duplicates before adding, so an explicit
                * check is not necessary. */
               set.add(s);
            }
         }
         set.add("()" + str);
      }
   }
   return set;
}

String insertInside(String str, int leftIndex) {
   String left = str.substring(0, leftIndex + 1);
   String right = str.substring(leftIndex + 1, str.length());
   return left + "()" + right;
}
```

This works, but it's not very efficient. We waste a lot of time coming up with the duplicate strings.

We can avoid this duplicate string issue by building the string from scratch. Under this approach, we add left and right parens, as long as our expression stays valid.

On each recursive call, we have the index for a particular character in the string. We need to select either a left or a right paren. When can we use a left paren, and when can we use a right paren?

1. *Left Paren:* As long as we haven't used up all the left parentheses, we can always insert a left paren.
2. *Right Paren:* We can insert a right paren as long as it won't lead to a syntax error. When will we get a syntax error? We will get a syntax error if there are more right parentheses than left.

So, we simply keep track of the number of left and right parentheses allowed. If there are left parens remaining, we'll insert a left paren and recurse. If there are more right parens remaining than left (i.e., if there are more left parens in use than right parens), then we'll insert a right paren and recurse.

```
void addParen(ArrayList<String> list, int leftRem, int rightRem, char[] str,
              int index) {
   if (leftRem < 0 || rightRem < leftRem) return; // invalid state

   if (leftRem == 0 && rightRem == 0) { /* Out of left and right parentheses */
      list.add(String.copyValueOf(str));
   } else {
      str[index] = '('; // Add left and recurse
      addParen(list, leftRem - 1, rightRem, str, index + 1);

      str[index] = ')'; // Add right and recurse
      addParen(list, leftRem, rightRem - 1, str, index + 1);
   }
}

ArrayList<String> generateParens(int count) {
   char[] str = new char[count*2];
   ArrayList<String> list = new ArrayList<String>();
   addParen(list, count, count, str, 0);
   return list;
}
```

Because we insert left and right parentheses at each index in the string, and we never repeat an index, each string is guaranteed to be unique.

8.10 **Paint Fill:** Implement the "paint fill" function that one might see on many image editing programs. That is, given a screen (represented by a two-dimensional array of colors), a point, and a new color, fill in the surrounding area until the color changes from the original color.

pg 136

SOLUTION

First, let's visualize how this method works. When we call `paintFill` (i.e., "click" paint fill in the image editing application) on, say, a green pixel, we want to "bleed" outwards. Pixel by pixel, we expand outwards by calling `paintFill` on the surrounding pixel. When we hit a pixel that is not green, we stop.

We can implement this algorithm recursively:

```
enum Color { Black, White, Red, Yellow, Green }

boolean PaintFill(Color[][] screen, int r, int c, Color ncolor) {
  if (screen[r][c] == ncolor) return false;
  return PaintFill(screen, r, c, screen[r][c], ncolor);
}

boolean PaintFill(Color[][] screen, int r, int c, Color ocolor, Color ncolor) {
  if (r < 0 || r >= screen.length || c < 0 || c >= screen[0].length) {
    return false;
  }

  if (screen[r][c] == ocolor) {
    screen[r][c] = ncolor;
    PaintFill(screen, r - 1, c, ocolor, ncolor); // up
    PaintFill(screen, r + 1, c, ocolor, ncolor); // down
    PaintFill(screen, r, c - 1, ocolor, ncolor); // left
    PaintFill(screen, r, c + 1, ocolor, ncolor); // right
  }
  return true;
}
```

If you used the variable names x and y to implement this, be careful about the ordering of the variables in `screen[y][x]`. Because x represents the *horizontal* axis (that is, it's left to right), it actually corresponds to the column number, not the row number. The value of y equals the number of rows. This is a very easy place to make a mistake in an interview, as well as in your daily coding. It's typically clearer to use `row` and `column` instead, as we've done here.

Does this algorithm seem familiar? It should! This is essentially depth-first search on a graph. At each pixel, we are searching outwards to each surrounding pixel. We stop once we've fully traversed all the surrounding pixels of this color.

We could alternatively implement this using breadth-first search.

8.11 Coins: Given an infinite number of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent), write code to calculate the number of ways of representing n cents.

pg 136

SOLUTION

This is a recursive problem, so let's figure out how to compute `makeChange(n)` using prior solutions (i.e., subproblems).

Let's say `n = 100`. We want to compute the number of ways of making change for 100 cents. What is the relationship between this problem and its subproblems?

We know that making change for 100 cents will involve either 0, 1, 2, 3, or 4 quarters. So:

```
makeChange(100) = makeChange(100 using 0 quarters) +
                  makeChange(100 using 1 quarter)  +
                  makeChange(100 using 2 quarters) +
                  makeChange(100 using 3 quarters) +
                  makeChange(100 using 4 quarters)
```

Inspecting this further, we can see that some of these problems reduce. For example, `makeChange(100 using 1 quarter)` will equal `makeChange(75 using 0 quarters)`. This is because, if we must use exactly one quarter to make change for 100 cents, then our only remaining choices involve making change for the remaining 75 cents.

We can apply the same logic to `makeChange(100 using 2 quarters)`, `makeChange(100 using 3 quarters)` and `makeChange(100 using 4 quarters)`. We have thus reduced the above statement to the following.

```
makeChange(100) = makeChange(100 using 0 quarters) +
                  makeChange(75 using 0 quarters) +
                  makeChange(50 using 0 quarters) +
                  makeChange(25 using 0 quarters) +
                  1
```

Note that the final statement from above, `makeChange(100 using 4 quarters)`, equals 1. We call this "fully reduced."

Now what? We've used up all our quarters, so now we can start applying our next biggest denomination: dimes.

Our approach for quarters applies to dimes as well, but we apply this for *each* of the four of five parts of the above statement. So, for the first part, we get the following statements:

```
makeChange(100 using 0 quarters) = makeChange(100 using 0 quarters, 0 dimes) +
                                   makeChange(100 using 0 quarters, 1 dime)  +
                                   makeChange(100 using 0 quarters, 2 dimes) +
                                   ...
                                   makeChange(100 using 0 quarters, 10 dimes)

makeChange(75 using 0 quarters) = makeChange(75 using 0 quarters, 0 dimes) +
                                  makeChange(75 using 0 quarters, 1 dime)  +
                                  makeChange(75 using 0 quarters, 2 dimes) +
                                  ...
                                  makeChange(75 using 0 quarters, 7 dimes)

makeChange(50 using 0 quarters) = makeChange(50 using 0 quarters, 0 dimes) +
                                  makeChange(50 using 0 quarters, 1 dime)  +
                                  makeChange(50 using 0 quarters, 2 dimes) +
```

```
                                         ...
                                         makeChange(50 using 0 quarters, 5 dimes)

makeChange(25 using 0 quarters) = makeChange(25 using 0 quarters, 0 dimes) +
                                  makeChange(25 using 0 quarters, 1 dime)  +
                                  makeChange(25 using 0 quarters, 2 dimes)
```

Each one of these, in turn, expands out once we start applying nickels. We end up with a tree-like recursive structure where each call expands out to four or more calls.

The base case of our recursion is the fully reduced statement. For example, `makeChange(50 using 0 quarters, 5 dimes)` is fully reduced to 1, since 5 dimes equals 50 cents.

This leads to a recursive algorithm that looks like this:

```
int makeChange(int amount, int[] denoms, int index) {
  if (index >= denoms.length - 1) return 1; // last denom
  int denomAmount = denoms[index];
  int ways = 0;
  for (int i = 0; i * denomAmount <= amount; i++) {
    int amountRemaining = amount - i * denomAmount;
    ways += makeChange(amountRemaining, denoms, index + 1);
  }
  return ways;
}

int makeChange(int n) {
  int[] denoms = {25, 10, 5, 1};
  return makeChange(n, denoms, 0);
}
```

This works, but it's not as optimal as it could be. The issue is that we will be recursively calling `makeChange` several times for the same values of `amount` and `index`.

We can resolve this issue by storing the previously computed values. We'll need to store a mapping from each pair `(amount, index)` to the precomputed result.

```
int makeChange(int n) {
  int[] denoms = {25, 10, 5, 1};
  int[][] map = new int[n + 1][denoms.length]; // precomputed vals
  return makeChange(n, denoms, 0, map);
}

int makeChange(int amount, int[] denoms, int index, int[][] map) {
  if (map[amount][index] > 0) { // retrieve value
    return map[amount][index];
  }
  if (index >= denoms.length - 1) return 1; // one denom remaining
  int denomAmount = denoms[index];
  int ways = 0;
  for (int i = 0; i * denomAmount <= amount; i++) {
    // go to next denom, assuming i coins of denomAmount
    int amountRemaining = amount - i * denomAmount;
    ways += makeChange(amountRemaining, denoms, index + 1, map);
  }
  map[amount][index] = ways;
  return ways;
}
```

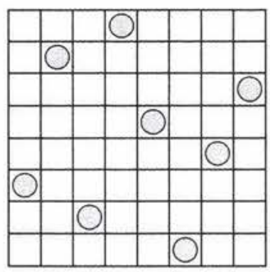
Note that we've used a two-dimensional array of integers to store the previously computed values. This is simpler, but takes up a little extra space. Alternatively, we could use an actual hash table that maps from `amount` to a new hash table, which then maps from `denom` to the precomputed value. There are other alternative data structures as well.

8.12 Eight Queens: Write an algorithm to print all ways of arranging eight queens on an 8x8 chess board so that none of them share the same row, column, or diagonal. In this case, "diagonal" means all diagonals, not just the two that bisect the board.

pg 136

SOLUTION

We have eight queens which must be lined up on an 8x8 chess board such that none share the same row, column or diagonal. So, we know that each row and column (and diagonal) must be used exactly once.



A "Solved" Board with 8 Queens

Picture the queen that is placed last, which we'll assume is on row 8. (This is an okay assumption to make since the ordering of placing the queens is irrelevant.) On which cell in row 8 is this queen? There are eight possibilities, one for each column.

So if we want to know all the valid ways of arranging 8 queens on an 8x8 chess board, it would be:

```
ways to arrange 8 queens on an 8x8 board =
   ways to arrange 8 queens on an 8x8 board with queen at (7, 0) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 1) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 2) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 3) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 4) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 5) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 6) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 7)
```

We can compute each one of these using a very similar approach:

```
ways to arrange 8 queens on an 8x8 board with queen at (7, 3) =
   ways to ... with queens at (7, 3) and (6, 0) +
   ways to ... with queens at (7, 3) and (6, 1) +
   ways to ... with queens at (7, 3) and (6, 2) +
   ways to ... with queens at (7, 3) and (6, 4) +
   ways to ... with queens at (7, 3) and (6, 5) +
   ways to ... with queens at (7, 3) and (6, 6) +
   ways to ... with queens at (7, 3) and (6, 7)
```

Note that we don't need to consider combinations with queens at `(7, 3)` and `(6, 3)`, since this is a violation of the requirement that every queen is in its own row, column and diagonal.

Implementing this is now reasonably straightforward.

```
int GRID_SIZE = 8;

void placeQueens(int row, Integer[] columns, ArrayList<Integer[]> results) {
  if (row == GRID_SIZE) { // Found valid placement
    results.add(columns.clone());
  } else {
    for (int col = 0; col < GRID_SIZE; col++) {
      if (checkValid(columns, row, col)) {
        columns[row] = col;  // Place queen
        placeQueens(row + 1, columns, results);
      }
    }
  }
}

/* Check if (row1, column1) is a valid spot for a queen by checking if there is a
 * queen in the same column or diagonal. We don't need to check it for queens in
 * the same row because the calling placeQueen only attempts to place one queen at
 * a time. We know this row is empty. */
boolean checkValid(Integer[] columns, int row1, int column1) {
  for (int row2 = 0; row2 < row1; row2++) {
    int column2 = columns[row2];
    /* Check if (row2, column2) invalidates (row1, column1) as a
     * queen spot. */

    /* Check if rows have a queen in the same column */
    if (column1 == column2) {
      return false;
    }

    /* Check diagonals: if the distance between the columns equals the distance
     * between the rows, then they're in the same diagonal. */
    int columnDistance = Math.abs(column2 - column1);

    /* row1 > row2, so no need for abs */
    int rowDistance = row1 - row2;
    if (columnDistance == rowDistance) {
      return false;
    }
  }
  return true;
}
```

Observe that since each row can only have one queen, we don't need to store our board as a full 8x8 matrix. We only need a single array where `column[r] = c` indicates that row r has a queen at column c.

8.13 Stack of Boxes: You have a stack of n boxes, with widths w_1, heights h_1, and depths d_1. The boxes cannot be rotated and can only be stacked on top of one another if each box in the stack is strictly larger than the box above it in width, height, and depth. Implement a method to compute the height of the tallest possible stack. The height of a stack is the sum of the heights of each box.

pg 136

SOLUTION

To tackle this problem, we need to recognize the relationship between the different subproblems.

Solution #1

Imagine we had the following boxes: b_1, b_2, ..., b_n. The biggest stack that we can build with all the boxes equals the max of (biggest stack with bottom b_1, biggest stack with bottom b_2, ..., biggest stack with bottom b_n). That is, if we experimented with each box as a bottom and built the biggest stack possible with each, we would find the biggest stack possible.

But, how would we find the biggest stack with a particular bottom? Essentially the same way. We experiment with different boxes for the second level, and so on for each level.

Of course, we only experiment with valid boxes. If b_5 is bigger than b_1, then there's no point in trying to build a stack that looks like $\{b_1, b_5, \ldots\}$. We already know b_1 can't be below b_5.

We can perform a small optimization here. The requirements of this problem stipulate that the lower boxes must be strictly greater than the higher boxes in all dimensions. Therefore, if we sort (descending order) the boxes on a dimension—any dimension—then we know we don't have to look backwards in the list. The box b_1 cannot be on top of box b_5, since its height (or whatever dimension we sorted on) is greater than b_5's height.

The code below implements this algorithm recursively.

```
int createStack(ArrayList<Box> boxes) {
  /* Sort in decending order by height. */
  Collections.sort(boxes, new BoxComparator());
  int maxHeight = 0;
  for (int i = 0; i < boxes.size(); i++) {
    int height = createStack(boxes, i);
    maxHeight = Math.max(maxHeight, height);
  }
  return maxHeight;
}

int createStack(ArrayList<Box> boxes, int bottomIndex) {
  Box bottom = boxes.get(bottomIndex);
  int maxHeight = 0;
  for (int i = bottomIndex + 1; i < boxes.size(); i++) {
    if (boxes.get(i).canBeAbove(bottom)) {
      int height = createStack(boxes, i);
      maxHeight = Math.max(height, maxHeight);
    }
  }
  maxHeight += bottom.height;
  return maxHeight;
}

class BoxComparator implements Comparator<Box> {
```

```
    @Override
    public int compare(Box x, Box y){
        return y.height - x.height;
    }
}
```

The problem in this code is that it gets very inefficient. We try to find the best solution that looks like {b_3, b_4, ...} even though we may have already found the best solution with b_4 at the bottom. Instead of generating these solutions from scratch, we can cache these results using memoization.

```
int createStack(ArrayList<Box> boxes) {
    Collections.sort(boxes, new BoxComparator());
    int maxHeight = 0;
    int[] stackMap = new int[boxes.size()];
    for (int i = 0; i < boxes.size(); i++) {
        int height = createStack(boxes, i, stackMap);
        maxHeight = Math.max(maxHeight, height);
    }
    return maxHeight;
}

int createStack(ArrayList<Box> boxes, int bottomIndex, int[] stackMap) {
    if (bottomIndex < boxes.size() && stackMap[bottomIndex] > 0) {
        return stackMap[bottomIndex];
    }

    Box bottom = boxes.get(bottomIndex);
    int maxHeight = 0;
    for (int i = bottomIndex + 1; i < boxes.size(); i++) {
        if (boxes.get(i).canBeAbove(bottom)) {
            int height = createStack(boxes, i, stackMap);
            maxHeight = Math.max(height, maxHeight);
        }
    }
    maxHeight += bottom.height;
    stackMap[bottomIndex] = maxHeight;
    return maxHeight;
}
```

Because we're only mapping from an index to a height, we can just use an integer array for our "hash table."

Be very careful here with what each spot in the hash table represents. In this code, `stackMap[i]` represents the tallest stack with box `i` at the bottom. Before pulling the value from the hash table, you have to ensure that box `i` can be placed on top of the current bottom.

It helps to keep the line that recalls from the hash table symmetric with the one that inserts. For example, in this code, we recall from the hash table with `bottomIndex` at the start of the method. We insert into the hash table with `bottomIndex` at the end.

Solution #2

Alternatively, we can think about the recursive algorithm as making a choice, at each step, whether to put a particular box in the stack. (We will again sort our boxes in descending order by a dimension, such as height.)

First, we choose whether or not to put box 0 in the stack. Take one recursive path with box 0 at the bottom and one recursive path without box 0. Return the better of the two options.

Then, we choose whether or not to put box 1 in the stack. Take one recursive path with box 1 at the bottom and one path without box 1. Return the better of the two options.

We will again use memoization to cache the height of the tallest stack with a particular bottom.

```
1  int createStack(ArrayList<Box> boxes) {
2    Collections.sort(boxes, new BoxComparator());
3    int[] stackMap = new int[boxes.size()];
4    return createStack(boxes, null, 0, stackMap);
5  }
6
7  int createStack(ArrayList<Box> boxes, Box bottom, int offset, int[] stackMap) {
8    if (offset >= boxes.size()) return 0; // Base case
9
10   /* height with this bottom */
11   Box newBottom = boxes.get(offset);
12   int heightWithBottom = 0;
13   if (bottom == null || newBottom.canBeAbove(bottom)) {
14     if (stackMap[offset] == 0) {
15       stackMap[offset] = createStack(boxes, newBottom, offset + 1, stackMap);
16       stackMap[offset] += newBottom.height;
17     }
18     heightWithBottom = stackMap[offset];
19   }
20
21   /* without this bottom */
22   int heightWithoutBottom = createStack(boxes, bottom, offset + 1, stackMap);
23
24   /* Return better of two options. */
25   return Math.max(heightWithBottom, heightWithoutBottom);
26 }
```

Again, pay close attention to when you recall and insert values into the hash table. It's typically best if these are symmetric, as they are in lines 15 and 16-18.

8.14 Boolean Evaluation: Given a boolean expression consisting of the symbols `0` (false), `1` (true), `&` (AND), `|` (OR), and `^` (XOR), and a desired boolean result value `result`, implement a function to count the number of ways of parenthesizing the expression such that it evaluates to `result`. The expression should be fully parenthesized (e.g., `(0)^(1)`) but not extraneously (e.g., `(((0))^(1))`).

EXAMPLE

```
countEval("1^0|0|1", false) -> 2
countEval("0&0&0&1^1|0", true) -> 10
```

pg 136

SOLUTION

As in other recursive problems, the key to this problem is to figure out the relationship between a problem and its subproblems.

Brute Force

Consider an expression like `0^0&0^1|1` and the target result `true`. How can we break down `countEval(0^0&0^1|1, true)` into smaller problems?

We could just essentially iterate through each possible place to put a parenthesis.

```
countEval(0^0&0^1|1, true) =
    countEval(0^0&0^1|1 where paren around char 1, true)
  + countEval(0^0&0^1|1 where paren around char 3, true)
  + countEval(0^0&0^1|1 where paren around char 5, true)
  + countEval(0^0&0^1|1 where paren around char 7, true)
```

Now what? Let's look at just one of those expressions—the paren around char 3. This gives us `(0^0)&(0^1)`.

In order to make that expression true, both the left and right sides must be true. So:

```
left = "0^0"
right = "0^1|1"
countEval(left & right, true) = countEval(left, true) * countEval(right, true)
```

The reason we multiply the results of the left and right sides is that each result from the two sides can be paired up with each other to form a unique combination.

Each of those terms can now be decomposed into smaller problems in a similar process.

What happens when we have an "|" (OR)? Or an "^" (XOR)?

If it's an OR, then either the left or the right side must be true—or both.

```
countEval(left | right, true) = countEval(left, true)  * countEval(right, false)
                              + countEval(left, false) * countEval(right, true)
                              + countEval(left, true)  * countEval(right, true)
```

If it's an XOR, then the left or the right side can be true, but not both.

```
countEval(left ^ right, true) = countEval(left, true) * countEval(right, false)
                              + countEval(left, false) * countEval(right, true)
```

What if we were trying to make the result `false` instead? We can switch up the logic from above:

```
countEval(left & right, false) = countEval(left, true)  * countEval(right, false)
                               + countEval(left, false) * countEval(right, true)
                               + countEval(left, false) * countEval(right, false)
countEval(left | right, false) = countEval(left, false) * countEval(right, false)
countEval(left ^ right, false) = countEval(left, false) * countEval(right, false)
                               + countEval(left, true)  * countEval(right, true)
```

Alternatively, we can just use the same logic from above and subtract it out from the total number of ways of evaluating the expression.

```
totalEval(left) = countEval(left, true) + countEval(left, false)
totalEval(right) = countEval(right, true) + countEval(right, false)
totalEval(expression) = totalEval(left) * totalEval(right)
countEval(expression, false) = totalEval(expression) - countEval(expression, true)
```

This makes the code a bit more concise.

```
int countEval(String s, boolean result) {
  if (s.length() == 0) return 0;
  if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;

  int ways = 0;
  for (int i = 1; i < s.length(); i += 2) {
    char c = s.charAt(i);
    String left = s.substring(0, i);
    String right = s.substring(i + 1, s.length());

    /* Evaluate each side for each result. */
    int leftTrue = countEval(left, true);
    int leftFalse = countEval(left, false);
    int rightTrue = countEval(right, true);
```

```
      int rightFalse = countEval(right, false);
      int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);

      int totalTrue = 0;
      if (c == ‘^’) { // required: one true and one false
         totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
      } else if (c == ‘&’) { // required: both true
         totalTrue = leftTrue * rightTrue;
      } else if (c == ‘|’) { // required: anything but both false
         totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
                     leftTrue * rightFalse;
      }

      int subWays = result ? totalTrue : total - totalTrue;
      ways += subWays;
   }

   return ways;
}

boolean stringToBool(String c) {
   return c.equals(“1”) ? true : false;
}
```

Note that the tradeoff of computing the `false` results from the `true` ones, and of computing the `{leftTrue, rightTrue, leftFalse, and rightFalse}` values upfront, is a small amount of extra work in some cases. For example, if we're looking for the ways that an AND (&) can result in `true`, we never would have needed the `leftFalse` and `rightFalse` results. Likewise, if we're looking for the ways that an OR (|) can result in `false`, we never would have needed the `leftTrue` and `rightTrue` results.

Our current code is blind to what we do and don't actually need to do and instead just computes all of the values. This is probably a reasonable tradeoff to make (especially given the constraints of whiteboard coding) as it makes our code substantially shorter and less tedious to write. Whichever approach you make, you should discuss the tradeoffs with your interviewer.

That said, there are more important optimizations we can make.

Optimized Solutions

If we follow the recursive path, we'll note that we end up doing the same computation repeatedly.

Consider the expression `0^0&0^1|1` and these recursion paths:

- Add parens around char 1. `(0)^(0&0^1|1)`
  - Add parens around char 3. `(0)^((0)&(0^1|1))`
- Add parens around char 3. `(0^0)&(0^1|1)`
  - Add parens around char 1. `((0)^(0))&(0^1|1)`

Although these two expressions are different, they have a similar component: `(0^1|1)`. We should reuse our effort on this.

We can do this by using memoization, or a hash table. We just need to store the result of `countEval(expression, result)` for each expression and result. If we see an expression that we've calculated before, we just return it from the cache.

```
int countEval(String s, boolean result, HashMap<String, Integer> memo) {
   if (s.length() == 0) return 0;
   if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;
```

```
    if (memo.containsKey(result + s)) return memo.get(result + s);

    int ways = 0;

    for (int i = 1; i < s.length(); i += 2) {
       char c = s.charAt(i);
       String left = s.substring(0, i);
       String right = s.substring(i + 1, s.length());
       int leftTrue = countEval(left, true, memo);
       int leftFalse = countEval(left, false, memo);
       int rightTrue = countEval(right, true, memo);
       int rightFalse = countEval(right, false, memo);
       int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);

       int totalTrue = 0;
       if (c == ‘^’) {
          totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
       } else if (c == ‘&’) {
          totalTrue = leftTrue * rightTrue;
       } else if (c == ‘|’) {
          totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
                      leftTrue * rightFalse;
       }

       int subWays = result ? totalTrue : total - totalTrue;
       ways += subWays;
    }

    memo.put(result + s, ways);
    return ways;
 }
```

The added benefit of this is that we could actually end up with the same substring in multiple parts of the expression. For example, an expression like `0^1^0&0^1^0` has two instances of `0^1^0`. By caching the result of the substring value in a memoization table, we'll get to reuse the result for the right part of the expression after computing it for the left.

There is one further optimization we can make, but it's far beyond the scope of the interview. There *is* a closed form expression for the number of ways of parenthesizing an expression, but you wouldn't be expected to know it. It is given by the Catalan numbers, where n is the number of operators:

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

We could use this to compute the total ways of evaluating the expression. Then, rather than computing `leftTrue` and `leftFalse`, we just compute one of those and calculate the other using the Catalan numbers. We would do the same thing for the right side.