CRACKING *the* CODING INTERVIEW

189 Programming Questions & Solutions

GAYLE LAAKMANN MCDOWELL

Author of Cracking the PM Interview and Cracking the Tech Career

6TH EDITION

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.

```
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`.

```
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.

```
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;
}
```

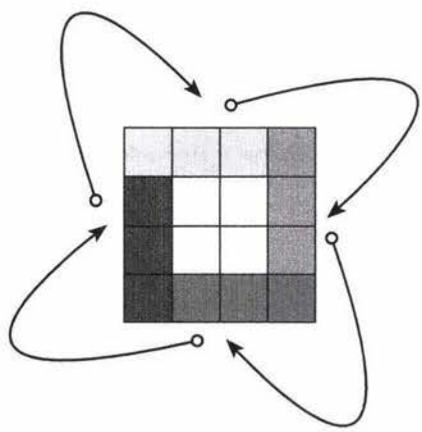
One other benefit of this approach is that we can initialize `StringBuilder` to its necessary capacity up-front. Without this, `StringBuilder` will (behind the scenes) need to double its capacity every time it hits capacity. The capacity could be double what we ultimately need.

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(N²)` 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 $N/2$ 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.

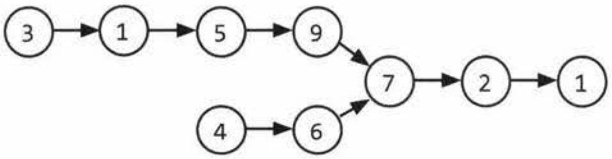
2.7 **Intersection**: Given two (singly) linked lists, determine if the two lists intersect. Return the intersecting node. Note that the intersection is defined based on reference, not value. That is, if the `k`th node of the first linked list is the exact same node (by reference) as the `j`th node of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

pg 95

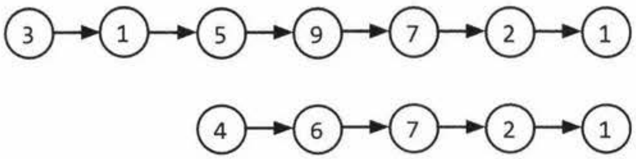
SOLUTION

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.

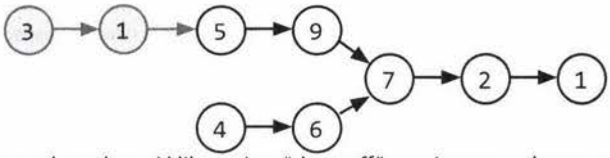
There's an easier way though. Observe that two intersecting linked lists will always have the same last node. Therefore, we can just traverse to the end of each linked list and compare the last nodes.

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?

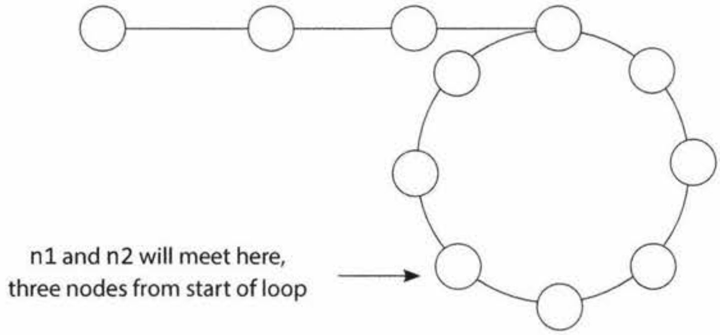
We know that for every p steps that `SlowRunner` takes, `FastRunner` has taken 2p steps. Therefore, when `SlowRunner` enters the looped portion after k steps, `FastRunner` has taken 2k steps total and must be `2k - k` steps, or k steps, into the looped portion. Since k might be much larger than the loop 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();
  }
```

```
   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();
    }
```

```

      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.

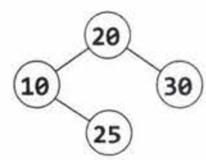
```
class WrapInt {
    public int value;
}
```

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:

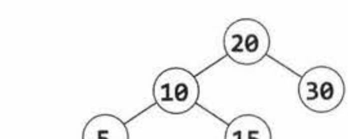


Although each node is bigger than its left node and smaller than its right node, this is clearly not a binary search tree since 25 is in the wrong place.

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:

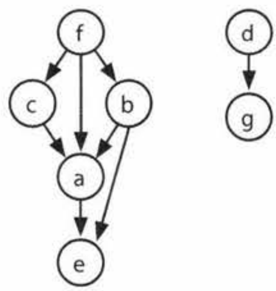
```
projects: a, b, c, d, e, f
dependencies: (a, d), (f, b), (b, d), (f, a), (d, c)
```

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.

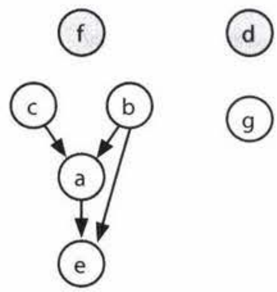
Solution #1

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`).

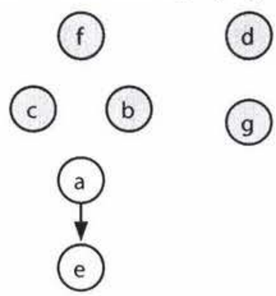
Once we've done that, it's irrelevant that some nodes are dependent on d and `f` since d and `f` have already been built. We can reflect this new state by removing d and `f`'s outgoing edges.

```
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;
    }

```

```java
      /* 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; }
```

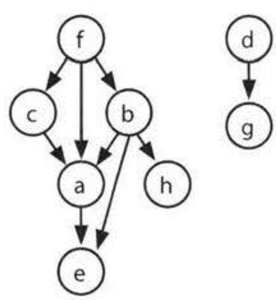
```

  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.

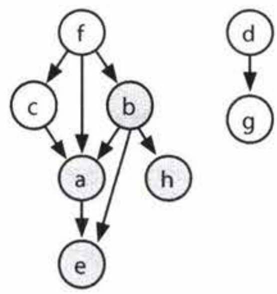
```
DFS(b)                              // Step 1
   DFS(h)                           // Step 2
      build order = ..., h          // Step 3
   DFS(a)                           // Step 4
      DFS(e)                        // Step 5
         build order = ..., e, h    // Step 6
      ...           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

```

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.

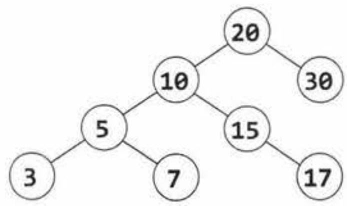
Solution #2: With Links to Parents (Better Worst-Case Runtime)

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 $2n/2$ nodes, then $2n/4$, 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:

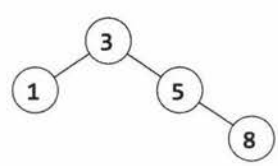
- Returns p, if `root`'s subtree includes p (and not q).
- Returns q, if `root`'s subtree includes q (and not p).
- Returns `null`, if neither p nor q are in `root`'s subtree.
- Else, returns the common ancestor of p and q.

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;
```

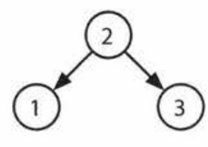
```
        return new Result(root, isAncestor);
    } else {
        return new Result(rx.node!=null ? rx.node : ry.node, false);
    }
}
```

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:

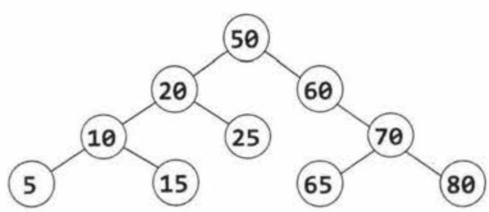


Output: `{2, 1, 3}`, `{2, 3, 1}`

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.

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

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

```
    }

    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.

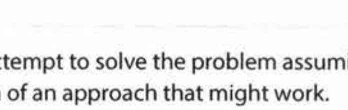
The Simple Approach

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.

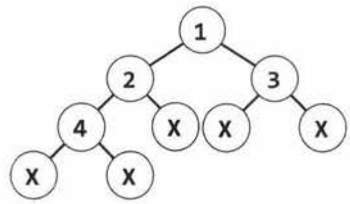


There's a simple fix though. We can store NULL nodes in the pre-order traversal string as a special character, like an 'X'. (We'll assume that the binary trees contain only integers.) The left tree would have the traversal `{3, 4, X}` and the right tree will have the traversal `{3, X, 4}`.

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 n/p. Why? Because each of n nodes in `T1` has a $1/p$ chance of equaling the root, so approximately 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.

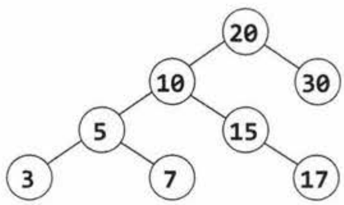
In summary, the alternative approach is certainly more optimal in terms of space and is likely more optimal in terms of time as well. It all depends on what assumptions you make and whether you prioritize reducing the average case runtime at the expense of the worst case runtime. This is an excellent 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 $\frac{1}{3}$ odds, we return the current node.
- With $\frac{1}{3}$ odds, we traverse left.
- With $\frac{1}{3}$ odds, we traverse right.

This solution, like some of the others, does not distribute the probabilities evenly across the nodes. The root has a $\frac{1}{3}$ 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 $\frac{1}{N}$ probability. (In fact, we must return each node with $\frac{1}{N}$ probability. After all, we have N nodes and each must have equal probability. The total must be 1 (100%), therefore each must have $\frac{1}{N}$ 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 $\frac{1}{N}$, the odds of picking something from the left must have probability `LEFT_SIZE *` $\frac{1}{N}$. This should therefore be the odds of going left.

Likewise, the odds of going right should be `RIGHT_SIZE *` $\frac{1}{N}$.

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.

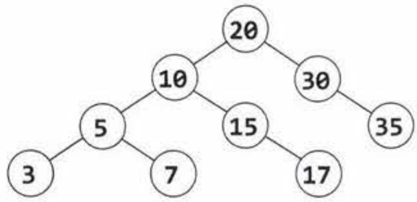
```
class TreeNode {
  private int data;
  public TreeNode left;
  public TreeNode right;
  private int size = 0;

  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 */ }
}
```

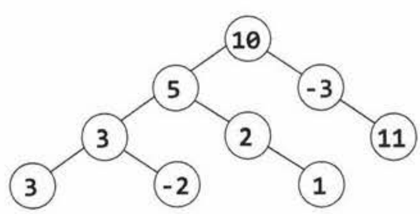
Like the previous algorithm, this algorithm takes `O(log N)` time in a balanced tree. We can also describe the runtime as `O(D)`, where D is the max depth of the tree. Note that `O(D)` is an accurate description of the runtime whether the tree is balanced or not.

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:

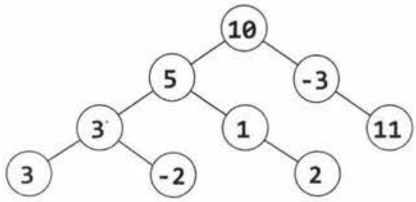
```
  0001
+ 0010
+ 0100
+ 1000
= 1111
```

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

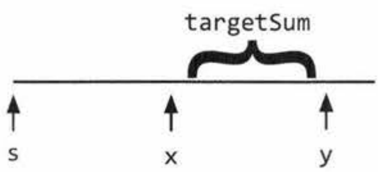
In analyzing the last solution, we may realize that we repeat some work. For a path such as `10 -> 5 -> 3 -> -2`, we traverse this path (or parts of it) repeatedly. We do it when we start with node 10, then when we go to node 5 (looking at 5, then 3, then -2), then when we go to node 3, and then finally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



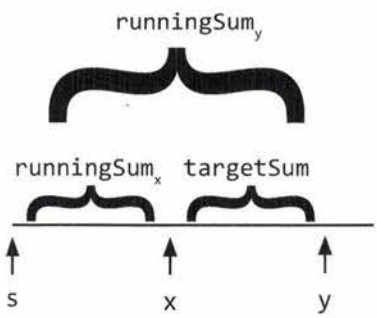
Let's isolate a given path and treat it as just an array. Consider a (hypothetical, extended) path like:

```
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: $\texttt{runningSum}_x = \texttt{runningSum}_y - \texttt{targetSum}$. 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 $\texttt{runningSum}_y - \texttt{targetSum}$ 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 $\texttt{runningSum}_7$ 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.

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.

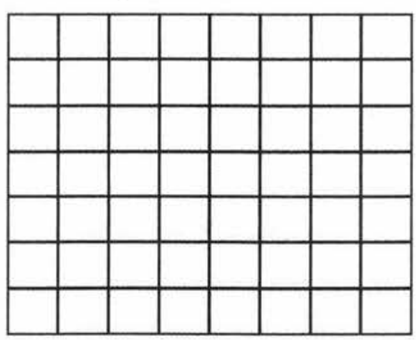
pg 135

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 $a_1a_2 \ldots a_n$.

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 $a_1a_2a_3a_4$.

$a_1a_2a_3$ -> $a_4a_1a_2a_3,\ a_1a_4a_2a_3,\ a_1a_2a_4a_3,\ a_1a_2a_3a_4$
$a_1a_3a_2$ -> $a_4a_1a_3a_2,\ a_1a_4a_3a_2,\ a_1a_3a_4a_2,\ a_1a_3a_2a_4$
$a_3a_1a_2$ -> $a_4a_3a_1a_2,\ a_3a_4a_1a_2,\ a_3a_1a_4a_2,\ a_3a_1a_2a_4$
$a_2a_1a_3$ -> $a_4a_2a_1a_3,\ a_2a_4a_1a_3,\ a_2a_1a_4a_3,\ a_2a_1a_3a_4$
$a_2a_3a_1$ -> $a_4a_2a_3a_1,\ a_2a_4a_3a_1,\ a_2a_3a_4a_1,\ a_2a_3a_1a_4$
$a_3a_2a_1$ -> $a_4a_3a_2a_1,\ a_3a_4a_2a_1,\ a_3a_2a_4a_1,\ a_3a_2a_1a_4$

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.

```java
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;
}
```

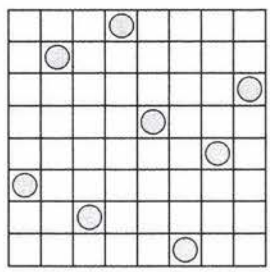
Note that we've used a two-dimensional array of integers to store the previously computed values. This is simpler, but takes up a little extra space. Alternatively, we could use an actual hash table that maps from `amount` to a new hash table, which then maps from `denom` to the precomputed value. There 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 $\{b_1, b_5, ...\}$. 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.

```java
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.

16

Solutions to Moderate

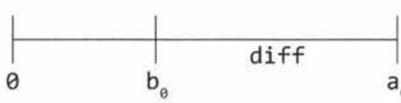
16.1 Number Swapper: Write a function to swap a number in place (that is, without temporary variables).

pg 181

SOLUTION

This is a classic interview problem, and it's a reasonably straightforward one. We'll walk through this using a_0 to indicate the original value of a and b_0 to indicate the original value of b. We'll also use `diff` to indicate the value of a_0 - b_0.

Let's picture these on a number line for the case where a > b.



First, we briefly set a to `diff`, which is the right side of the above number line. Then, when we add b and `diff` (and store that value in b), we get a_0. We now have b = a_0 and a = `diff`. All that's left to do is to set a equal to a_0 - `diff`, which is just b - a.

The code below implements this.

```
// Example for a = 9, b = 4
a = a - b; // a = 9 - 4 = 5
b = a + b; // b = 5 + 4 = 9
a = b - a; // a = 9 - 5
```

We can implement a similar solution with bit manipulation. The benefit of this solution is that it works for more data types than just integers.

```
// Example for a = 101 (in binary) and b = 110
a = a^b; // a = 101^110 = 011
b = a^b; // b = 011^110 = 101
a = a^b; // a = 011^101 = 110
```

This code works by using XORs. The easiest way to see how this works is by focusing on a specific bit. If we can correctly swap two bits, then we know the entire operation works correctly.

Let's take two bits, x and y, and walk through this line by line.

1. x = x ^ y

   This line essentially checks if x and y have different values. It will result in 1 if and only if x != y.

2. y = x ^ y

Or: `y = {0 if originally same, 1 if different} ^ {original y}`

Observe that XORing a bit with 1 always flips the bit, whereas XORing with 0 will never change it.

Therefore, if we do `y = 1 ^ {original y}` when `x != y`, then y will be flipped and therefore have x's original value.

Otherwise, if `x == y`, then we do `y = 0 ^ {original y}` and the value of `y` does not change.

Either way, y will be equal to the original value of x.

3. `x = x ^ y`

   Or: `x = {0 if originally same, 1 if different} ^ {original x}`

   At this point, y is equal to the original value of x. This line is essentially equivalent to the line above it, but for different variables.

   If we do `x = 1 ^ {original x}` when the values are different, x will be flipped.

   If we do `x = 0 ^ {original x}` when the values are the same, x will not be changed.

This operation happens for each bit. Since it correctly swaps each bit, it will correctly swap the entire number.

16.2 Word Frequencies: Design a method to find the frequency of occurrences of any given word in a book. What if we were running this algorithm multiple times?

pg 181

SOLUTION

Let's start with the simple case.

Solution: Single Query

In this case, we simply go through the book, word by word, and count the number of times that a word appears. This will take O(n) time. We know we can't do better than that since we must look at every word in the book.

```
int getFrequency(String[] book, String word) {
  word = word.trim().toLowerCase();
  int count = 0;
  for (String w : book) {
    if (w.trim().toLowerCase().equals(word)) {
      count++;
    }
  }
  return count;
}
```

We have also converted the string to lowercase and trimmed it. You can discuss with your interviewer if this is necessary (or even desired).

Solution: Repetitive Queries

If we're doing the operation repeatedly, then we can probably afford to take some time and extra memory to do pre-processing on the book. We can create a hash table which maps from a word to its frequency. The frequency of any word can be easily looked up in O(1) time. The code for this is below.

```
HashMap<String, Integer> setupDictionary(String[] book) {
  HashMap<String, Integer> table =
```

```
      new HashMap<String, Integer>();
    for (String word : book) {
      word = word.toLowerCase();
      if (word.trim() != "") {
        if (!table.containsKey(word)) {
          table.put(word, 0);
        }
        table.put(word, table.get(word) + 1);
      }
    }
    return table;
}

int getFrequency(HashMap<String, Integer> table, String word) {
    if (table == null || word == null) return -1;
    word = word.toLowerCase();
    if (table.containsKey(word)) {
      return table.get(word);
    }
    return 0;
}
```

Note that a problem like this is actually relatively easy. Thus, the interviewer is going to be looking heavily at how careful you are. Did you check for error conditions?

16.3 Intersection: Given two straight line segments (represented as a start point and an end point), compute the point of intersection, if any.

pg 181

SOLUTION

We first need to think about what it means for two line segments to intersect.

For two infinite lines to intersect, they only have to have different slopes. If they have the same slope, then they must be the exact same line (same y-intercept). That is:

```
slope 1 != slope 2
OR
slope 1 == slope 2 AND intersect 1 == intersect 2
```

For two straight lines to intersect, the condition above must be true, *plus* the point of intersection must be within the ranges of each line segment.

```
extended infinite segments intersect
AND
intersection is within line segment 1 (x and y coordinates)
AND
intersection is within line segment 2 (x and y coordinates)
```

What if the two segments represent the same infinite line? In this case, we have to ensure that some portion of their segments overlap. If we order the line segments by their x locations (start is before end, point 1 is before point 2), then an intersection occurs only if:

```
Assume:
    start1.x < start2.x && start1.x < end1.x && start2.x < end2.x
Then intersection occurs if:
    start2 is between start1 and end1
```

We can now go ahead and implement this algorithm.

```
Point intersection(Point start1, Point end1, Point start2, Point end2) {
  /* Rearranging these so that, in order of x values: start is before end and
   * point 1 is before point 2. This will make some of the later logic simpler. */
  if (start1.x > end1.x) swap(start1, end1);
  if (start2.x > end2.x) swap(start2, end2);
  if (start1.x > start2.x) {
    swap(start1, start2);
    swap(end1, end2);
  }

  /* Compute lines (including slope and y-intercept). */
  Line line1 = new Line(start1, end1);
  Line line2 = new Line(start2, end2);

  /* If the lines are parallel, they intercept only if they have the same y
   * intercept and start 2 is on line 1. */
  if (line1.slope == line2.slope) {
    if (line1.yintercept == line2.yintercept &&
        isBetween(start1, start2, end1)) {
      return start2;
    }
    return null;
  }

  /* Get intersection coordinate. */
  double x = (line2.yintercept - line1.yintercept) / (line1.slope - line2.slope);
  double y = x * line1.slope + line1.yintercept;
  Point intersection = new Point(x, y);

  /* Check if within line segment range. */
  if (isBetween(start1, intersection, end1) &&
      isBetween(start2, intersection, end2)) {
    return intersection;
  }
  return null;
}

/* Checks if middle is between start and end. */
boolean isBetween(double start, double middle, double end) {
  if (start > end) {
    return end <= middle && middle <= start;
  } else {
    return start <= middle && middle <= end;
  }
}

/* Checks if middle is between start and end. */
boolean isBetween(Point start, Point middle, Point end) {
  return isBetween(start.x, middle.x, end.x) &&
         isBetween(start.y, middle.y, end.y);
}

/* Swap coordinates of point one and two. */
void swap(Point one, Point two) {
  double x = one.x;
  double y = one.y;
```

```
    one.setLocation(two.x, two.y);
    two.setLocation(x, y);
}

public class Line {
    public double slope, yintercept;

    public Line(Point start, Point end) {
        double deltaY = end.y - start.y;
        double deltaX = end.x - start.x;
        slope = deltaY / deltaX; // Will be Infinity (not exception) when deltaX = 0
        yintercept = end.y - slope * end.x;
    }

public class Point {
    public double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public void setLocation(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

For simplicity and compactness (it really makes the code easier to read), we've chosen to make the variables within `Point` and `Line` `public`. You can discuss with your interviewer the advantages and disadvantages of this choice.

16.4 Tic Tac Win: Design an algorithm to figure out if someone has won a game of tic-tac-toe.

pg 181

SOLUTION

At first glance, this problem seems really straightforward. We're just checking a tic-tac-toe board; how hard could it be? It turns out that the problem is a bit more complex, and there is no single "perfect" answer. The optimal solution depends on your preferences.

There are a few major design decisions to consider:

1. Will `hasWon` be called just once or many times (for instance, as part of a tic-tac-toe website)? If the latter is the case, we may want to add pre-processing time to optimize the runtime of `hasWon`.
2. Do we know the last move that was made?
3. Tic-tac-toe is usually on a 3x3 board. Do we want to design for just that, or do we want to implement it as an NxN solution?
4. In general, how much do we prioritize compactness of code versus speed of execution vs. clarity of code? Remember: The most efficient code may not always be the best. Your ability to understand and maintain the code matters, too.

Solution #1: If hasWon is called many times

There are only 3^9, or about 20,000, tic-tac-toe boards (assuming a 3x3 board). Therefore, we can represent our tic-tac-toe board as an `int`, with each digit representing a piece (0 means Empty, 1 means Red, 2 means Blue). We set up a hash table or array in advance with all possible boards as keys and the value indicating who has won. Our function then is simply this:

```
Piece hasWon(int board) {
  return winnerHashtable[board];
}
```

To convert a board (represented by a char array) to an `int`, we can use what is essentially a "base 3" representation. Each board is represented as $3^0v_0 + 3^1v_1 + 3^2v_2 + \ldots + 3^8v_8$, where v_i is a 0 if the space is empty, a 1 if it's a "blue spot" and a 2 if it's a "red spot."

```
enum Piece { Empty, Red, Blue };

int convertBoardToInt(Piece[][] board) {
  int sum = 0;
  for (int i = 0; i < board.length; i++) {
    for (int j = 0; j < board[i].length; j++) {
      /* Each value in enum has an integer associated with it. We
       * can just use that. */
      int value = board[i][j].ordinal();
      sum = sum * 3 + value;
    }
  }
  return sum;
}
```

Now looking up the winner of a board is just a matter of looking it up in a hash table.

Of course, if we need to convert a board into this format every time we want to check for a winner, we haven't saved ourselves any time compared with the other solutions. But, if we can store the board this way from the very beginning, then the lookup process will be very efficient.

Solution #2: If we know the last move

If we know the very last move that was made (and we've been checking for a winner up until now), then we only need to check the row, column, and diagonal that overlaps with this position.

```
Piece hasWon(Piece[][] board, int row, int column) {
  if (board.length != board[0].length) return Piece.Empty;

  Piece piece = board[row][column];

  if (piece == Piece.Empty) return Piece.Empty;

  if (hasWonRow(board, row) || hasWonColumn(board, column)) {
    return piece;
  }

  if (row == column && hasWonDiagonal(board, 1)) {
    return piece;
  }

  if (row == (board.length - column - 1) && hasWonDiagonal(board, -1)) {
    return piece;
  }
```

```

  return Piece.Empty;
}

boolean hasWonRow(Piece[][] board, int row) {
  for (int c = 1; c < board[row].length; c++) {
    if (board[row][c] != board[row][0]) {
      return false;
    }
  }
  return true;
}

boolean hasWonColumn(Piece[][] board, int column) {
  for (int r = 1; r < board.length; r++) {
    if (board[r][column] != board[0][column]) {
      return false;
    }
  }
  return true;
}

boolean hasWonDiagonal(Piece[][] board, int direction) {
  int row = 0;
  int column = direction == 1 ? 0 : board.length - 1;
  Piece first = board[0][column];
  for (int i = 0; i < board.length; i++) {
    if (board[row][column] != first) {
      return false;
    }
    row += 1;
    column += direction;
  }
  return true;
}
```

There is actually a way to clean up this code to remove some of the duplicated code. We'll see this approach in a later function.

Solution #3: Designing for just a 3x3 board

If we really only want to implement a solution for a 3x3 board, the code is relatively short and simple. The only complex part is trying to be clean and organized, without writing too much duplicated code.

The code below checks each row, column, and diagonal to see if there is a winner.

```
Piece hasWon(Piece[][] board) {
  for (int i = 0; i < board.length; i++) {
    /* Check Rows */
    if (hasWinner(board[i][0], board[i][1], board[i][2])) {
      return board[i][0];
    }

    /* Check Columns */
    if (hasWinner(board[0][i], board[1][i], board[2][i])) {
      return board[0][i];
    }
```

```
    }

    /* Check Diagonal */
    if (hasWinner(board[0][0], board[1][1], board[2][2])) {
       return board[0][0];
    }

    if (hasWinner(board[0][2], board[1][1], board[2][0])) {
       return board[0][2];
    }

    return Piece.Empty;
 }

 boolean hasWinner(Piece p1, Piece p2, Piece p3) {
    if (p1 == Piece.Empty) {
       return false;
    }
    return p1 == p2 && p2 == p3;
 }
```

This is an okay solution in that it's relatively easy to understand what is going on. The problem is that the values are hard coded. It's easy to accidentally type the wrong indices.

Additionally, it won't be easy to scale this to an NxN board.

Solution #4: Designing for an NxN board

There are a number of ways to implement this on an NxN board.

Nested For-Loops

The most obvious way is through a series of nested for-loops.

```
Piece hasWon(Piece[][] board) {
   int size = board.length;
   if (board[0].length != size) return Piece.Empty;
   Piece first;

   /* Check rows. */
   for (int i = 0; i < size; i++) {
      first = board[i][0];
      if (first == Piece.Empty) continue;
      for (int j = 1; j < size; j++) {
         if (board[i][j] != first) {
            break;
         } else if (j == size - 1) { // Last element
            return first;
         }
      }
   }

   /* Check columns. */
   for (int i = 0; i < size; i++) {
      first = board[0][i];
      if (first == Piece.Empty) continue;
      for (int j = 1; j < size; j++) {
         if (board[j][i] != first) {
```

```
                break;
            } else if (j == size - 1) { // Last element
                return first;
            }
        }
    }

    /* Check diagonals. */
    first = board[0][0];
    if (first != Piece.Empty) {
        for (int i = 1; i < size; i++) {
            if (board[i][i] != first) {
                break;
            } else if (i == size - 1) { // Last element
                return first;
            }
        }
    }

    first = board[0][size - 1];
    if (first != Piece.Empty) {
        for (int i = 1; i < size; i++) {
            if (board[i][size - i - 1] != first) {
                break;
            } else if (i == size - 1) { // Last element
                return first;
            }
        }
    }

    return Piece.Empty;
}
```

This is, to the say the least, pretty ugly. We're doing nearly the same work each time. We should look for a way of reusing the code.

Increment and Decrement Function

One way that we can reuse the code better is to just pass in the values to another function that increments/ decrements the rows and columns. The hasWon function now just needs the starting position and the amount to increment the row and column by.

```
class Check {
    public int row, column;
    private int rowIncrement, columnIncrement;
    public Check(int row, int column, int rowI, int colI) {
        this.row = row;
        this.column = column;
        this.rowIncrement = rowI;
        this.columnIncrement = colI;
    }

    public void increment() {
        row += rowIncrement;
        column += columnIncrement;
    }

    public boolean inBounds(int size) {
```

```
      return row >= 0 && column >= 0 && row < size && column < size;
    }
  }

  Piece hasWon(Piece[][] board) {
    if (board.length != board[0].length) return Piece.Empty;
    int size = board.length;

    /* Create list of things to check. */
    ArrayList<Check> instructions = new ArrayList<Check>();
    for (int i = 0; i < board.length; i++) {
      instructions.add(new Check(0, i, 1, 0));
      instructions.add(new Check(i, 0, 0, 1));
    }
    instructions.add(new Check(0, 0, 1, 1));
    instructions.add(new Check(0, size - 1, 1, -1));

    /* Check them. */
    for (Check instr : instructions) {
      Piece winner = hasWon(board, instr);
      if (winner != Piece.Empty) {
          return winner;
      }
    }
    return Piece.Empty;
  }

  Piece hasWon(Piece[][] board, Check instr) {
    Piece first = board[instr.row][instr.column];
    while (instr.inBounds(board.length)) {
      if (board[instr.row][instr.column] != first) {
        return Piece.Empty;
      }
      instr.increment();
    }
    return first;
  }
```

The Check function is essentially operating as an iterator.

Iterator

Another way of doing it is, of course, to actually build an iterator.

```
  Piece hasWon(Piece[][] board) {
    if (board.length != board[0].length) return Piece.Empty;
    int size = board.length;

    ArrayList<PositionIterator> instructions = new ArrayList<PositionIterator>();
    for (int i = 0; i < board.length; i++) {
      instructions.add(new PositionIterator(new Position(0, i), 1, 0, size));
      instructions.add(new PositionIterator(new Position(i, 0), 0, 1, size));
    }
    instructions.add(new PositionIterator(new Position(0, 0), 1, 1, size));
    instructions.add(new PositionIterator(new Position(0, size - 1), 1, -1, size));

    for (PositionIterator iterator : instructions) {
      Piece winner = hasWon(board, iterator);
```

```
    if (winner != Piece.Empty) {
      return winner;
    }
  }
  return Piece.Empty;
}

Piece hasWon(Piece[][] board, PositionIterator iterator) {
  Position firstPosition = iterator.next();
  Piece first = board[firstPosition.row][firstPosition.column];
  while (iterator.hasNext()) {
    Position position = iterator.next();
    if (board[position.row][position.column] != first) {
      return Piece.Empty;
    }
  }
  return first;
}

class PositionIterator implements Iterator<Position> {
  private int rowIncrement, colIncrement, size;
  private Position current;

  public PositionIterator(Position p, int rowIncrement,
                          int colIncrement, int size) {
    this.rowIncrement = rowIncrement;
    this.colIncrement = colIncrement;
    this.size = size;
    current = new Position(p.row - rowIncrement, p.column - colIncrement);
  }

  @Override
  public boolean hasNext() {
    return current.row + rowIncrement < size &&
           current.column + colIncrement < size;
  }

  @Override
  public Position next() {
    current = new Position(current.row + rowIncrement,
                           current.column + colIncrement);
    return current;
  }
}

public class Position {
  public int row, column;
  public Position(int row, int column) {
    this.row = row;
    this.column = column;
  }
}
```

All of this is potentially overkill, but it's worth discussing the options with your interviewer. The point of this problem is to assess your understanding of how to code in a clean and maintainable way.

16.5 Factorial Zeros: Write an algorithm which computes the number of trailing zeros in n factorial.

pg 181

SOLUTION

A simple approach is to compute the factorial, and then count the number of trailing zeros by continuously dividing by ten. The problem with this though is that the bounds of an `int` would be exceeded very quickly. To avoid this issue, we can look at this problem mathematically.

Consider a factorial like `19!`:

```
19! = 1*2*3*4*5*6*7*8*9*10*11*12*13*14*15*16*17*18*19
```

A trailing zero is created with multiples of 10, and multiples of 10 are created with pairs of 5-multiples and 2-multiples.

For example, in 19!, the following terms create the trailing zeros:

```
19! = 2 * ... * 5 * ... * 10 * ... * 15 * 16 * ...
```

Therefore, to count the number of zeros, we only need to count the pairs of multiples of 5 and 2. There will always be more multiples of 2 than 5, though, so simply counting the number of multiples of 5 is sufficient.

One "gotcha" here is 15 contributes a multiple of 5 (and therefore one trailing zero), while 25 contributes two (because `25 = 5 * 5`).

There are two different ways to write this code.

The first way is to iterate through all the numbers from 2 through n, counting the number of times that 5 goes into each number.

```
/* If the number is a 5 of five, return which power of 5. For example: 5 -> 1,
 * 25-> 2, etc. */
int factorsOf5(int i) {
  int count = 0;
  while (i % 5 == 0) {
    count++;
    i /= 5;
  }
  return count;
}

int countFactZeros(int num) {
  int count = 0;
  for (int i = 2; i <= num; i++) {
    count += factorsOf5(i);
  }
  return count;
}
```

This isn't bad, but we can make it a little more efficient by directly counting the factors of 5. Using this approach, we would first count the number of multiples of 5 between 1 and n (which is $^n/_5$), then the number of multiples of 25 ($^n/_{25}$), then 125, and so on.

To count how many multiples of `m` are in n, we can just divide n by `m`.

```
int countFactZeros(int num) {
  int count = 0;
  if (num < 0) {
    return -1;
  }
```

```
    for (int i = 5; num / i > 0; i *= 5) {
      count += num / i;
    }
    return count;
}
```

This problem is a bit of a brainteaser, but it can be approached logically (as shown above). By thinking through what exactly will contribute a zero, you can come up with a solution. You should be very clear in your rules upfront so that you can implement it correctly.

16.6 Smallest Difference: Given two arrays of integers, compute the pair of values (one value in each array) with the smallest (non-negative) difference. Return the difference.

EXAMPLE

Input: {1, 3, 15, 11, 2}, {23, 127, 235, 19, 8}

Output: 3. That is, the pair (11, 8).

pg 181

SOLUTION

Let's start first with a brute force solution.

Brute Force

The simple brute force way is to just iterate through all pairs, compute the difference, and compare it to the current minimum difference.

```
int findSmallestDifference(int[] array1, int[] array2) {
  if (array1.length == 0 || array2.length == 0) return -1;

  int min = Integer.MAX_VALUE;
  for (int i = 0; i < array1.length; i++) {
    for (int j = 0; j < array2.length; j++) {
      if (Math.abs(array1[i] - array2[j]) < min) {
        min = Math.abs(array1[i] - array2[j]);
      }
    }
  }
  return min;
}
```

One minor optimization we could perform from here is to return immediately if we find a difference of zero, since this is the smallest difference possible. However, depending on the input, this might actually be slower.

This will only be faster if there's a pair with difference zero early in the list of pairs. But to add this optimization, we need to execute an additional line of code each time. There's a tradeoff here; it's faster for some inputs and slower for others. Given that it adds complexity in reading the code, it may be best to leave it out.

With or without this "optimization," the algorithm will take O(AB) time.

Optimal

A more optimal approach is to sort the arrays. Once the arrays are sorted, we can find the minimum difference by iterating through the array.

Consider the following two arrays:

```
A: {1, 2, 11, 15}
B: {4, 12, 19, 23, 127, 235}
```

Try the following approach:

1. Suppose a pointer a points to the beginning of A and a pointer b points to the beginning of B. The current difference between a and b is 3. Store this as the `min`.
2. How can we (potentially) make this difference smaller? Well, the value at b is bigger than the value at a, so moving b will only make the difference larger. Therefore, we want to move a.
3. Now a points to 2 and b (still) points to 4. This difference is 2, so we should update `min`. Move a, since it is smaller.
4. Now a points to 11 and b points to 4. Move b.
5. Now a points to 11 and b points to 12. Update `min` to 1. Move b.

And so on.

```
int findSmallestDifference(int[] array1, int[] array2) {
  Arrays.sort(array1);
  Arrays.sort(array2);
  int a = 0;
  int b = 0;
  int difference = Integer.MAX_VALUE;
  while (a < array1.length && b < array2.length) {
    if (Math.abs(array1[a] - array2[b]) < difference) {
      difference = Math.abs(array1[a] - array2[b]);
    }

    /* Move smaller value. */
    if (array1[a] < array2[b]) {
      a++;
    } else {
      b++;
    }
  }
  return difference;
}
```

This algorithm takes `O(A log A + B log B)` time to sort and `O(A + B)` time to find the minimum difference. Therefore, the overall runtime is `O(A log A + B log B)`.

16.7 Number Max: Write a method that finds the maximum of two numbers. You should not use if-else or any other comparison operator.

pg 181

SOLUTION

A common way of implementing a `max` function is to look at the sign of `a - b`. In this case, we can't use a comparison operator on this sign, but we *can* use multiplication.

Let k equal the sign of `a - b` such that if `a - b >= 0`, then k is 1. Else, `k = 0`. Let q be the inverse of k.

We can then implement the code as follows:

```
/* Flips a 1 to a 0 and a 0 to a 1 */
int flip(int bit) {
```

```
    return 1^bit;
}

/* Returns 1 if a is positive, and 0 if a is negative */
int sign(int a) {
    return flip((a >> 31) & 0x1);
}

int getMaxNaive(int a, int b) {
    int k = sign(a - b);
    int q = flip(k);
    return a * k + b * q;
}
```

This code almost works. It fails, unfortunately, when a - b overflows. Suppose, for example, that a is INT_MAX - 2 and b is -15. In this case, a - b will be greater than INT_MAX and will overflow, resulting in a negative value.

We can implement a solution to this problem by using the same approach. Our goal is to maintain the condition where k is 1 when a > b. We will need to use more complex logic to accomplish this.

When does a - b overflow? It will overflow only when a is positive and b is negative, or the other way around. It may be difficult to specially detect the overflow condition, but we *can* detect when a and b have different signs. Note that if a and b have different signs, then we want k to equal `sign(a)`.

The logic looks like:

```
if a and b have different signs:
    // if a > 0, then b < 0, and k = 1.
    // if a < 0, then b > 0, and k = 0.
    // so either way, k = sign(a)
    let k = sign(a)
else
    let k = sign(a - b) // overflow is impossible
```

The code below implements this, using multiplication instead of if-statements.

```
int getMax(int a, int b) {
    int c = a - b;

    int sa = sign(a); // if a >= 0, then 1 else 0
    int sb = sign(b); // if b >= 0, then 1 else 0
    int sc = sign(c); // depends on whether or not a - b overflows

    /* Goal: define a value k which is 1 if a > b and 0 if a < b.
     * (if a = b, it doesn't matter what value k is) */

    // If a and b have different signs, then k = sign(a)
    int use_sign_of_a = sa ^ sb;

    // If a and b have the same sign, then k = sign(a - b)
    int use_sign_of_c = flip(sa ^ sb);

    int k = use_sign_of_a * sa + use_sign_of_c * sc;
    int q = flip(k); // opposite of k

    return a * k + b * q;
}
```

Note that for clarity, we split up the code into many different methods and variables. This is certainly not the most compact or efficient way to write it, but it does make what we're doing much cleaner.

16.8 English Int: Given any integer, print an English phrase that describes the integer (e.g., "One Thousand, Two Hundred Thirty Four").

pg 182

SOLUTION

This is not an especially challenging problem, but it is a somewhat tedious one. The key is to be organized in how you approach the problem—and to make sure you have good test cases.

We can think about converting a number like 19,323,984 as converting each of three 3-digit segments of the number, and inserting "thousands" and "millions" in between as appropriate. That is,

```
convert(19,323,984) = convert(19) + " million " + convert(323) + " thousand " +
                      convert(984)
```

The code below implements this algorithm.

```
String[] smalls = {"Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven",
   "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen",
   "Sixteen", "Seventeen", "Eighteen", "Nineteen"};
String[] tens = {"", "", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy",
   "Eighty", "Ninety"};
String[] bigs = {"", "Thousand", "Million", "Billion"};
String hundred = "Hundred";
String negative = "Negative";

String convert(int num) {
   if (num == 0) {
      return smalls[0];
   } else if (num < 0) {
      return negative + " " + convert(-1 * num);
   }

   LinkedList<String> parts = new LinkedList<String>();
   int chunkCount = 0;

   while (num > 0) {
      if (num % 1000 != 0) {
         String chunk = convertChunk(num % 1000) + " " + bigs[chunkCount];
         parts.addFirst(chunk);
      }
      num /= 1000; // shift chunk
      chunkCount++;
   }

   return listToString(parts);
}

String convertChunk(int number) {
   LinkedList<String> parts = new LinkedList<String>();

   /* Convert hundreds place */
   if (number >= 100) {
      parts.addLast(smalls[number / 100]);
```

```
      parts.addLast(hundred);
      number %= 100;
   }

   /* Convert tens place */
   if (number >= 10 && number <= 19) {
      parts.addLast(smalls[number]);
   } else if (number >= 20) {
      parts.addLast(tens[number / 10]);
      number %= 10;
   }

   /* Convert ones place */
   if (number >= 1 && number <= 9) {
      parts.addLast(smalls[number]);
   }

   return listToString(parts);
}
/* Convert a linked list of strings to a string, dividing it up with spaces. */
String listToString(LinkedList<String> parts) {
   StringBuilder sb = new StringBuilder();
   while (parts.size() > 1) {
      sb.append(parts.pop());
      sb.append(" ");
   }
   sb.append(parts.pop());
   return sb.toString();
}
```

The key in a problem like this is to make sure you consider all the special cases. There are a lot of them.

16.9 Operations: Write methods to implement the multiply, subtract, and divide operations for integers. The results of all of these are integers. Use only the add operator.

pg 182

SOLUTION

The only operation we have to work with is the add operator. In each of these problems, it's useful to think in depth about what these operations really do or how to phrase them in terms of other operations (either add or operations we've already completed).

Subtraction

How can we phrase subtraction in terms of addition? This one is pretty straightforward. The operation `a - b` is the same thing as `a + (-1) * b`. However, because we are not allowed to use the * (multiply) operator, we must implement a `negate` function.

```
/* Flip a positive sign to negative or negative sign to pos. */
int negate(int a) {
   int neg = 0;
   int newSign = a < 0 ? 1 : -1;
   while (a != 0) {
      neg += newSign;
      a += newSign;
   }
```

```
    return neg;
}

/* Subtract two numbers by negating b and adding them */
int minus(int a, int b) {
    return a + negate(b);
}
```

The negation of the value k is implemented by adding -1 k times. Observe that this will take `O(k)` time.

If optimizing is something we value here, we can try to get a to zero faster. (For this explanation, we'll assume that a is positive.) To do this, we can first reduce a by 1, then 2, then 4, then 8, and so on. We'll call this value `delta`. We want a to reach exactly zero. When reducing a by the next `delta` would change the sign of a, we reset `delta` back to 1 and repeat the process.

For example:

```
a:       29   28   26   22   14   13   11    7    6    4    0
delta:     -1   -2   -4   -8   -1   -2   -4   -1   -2   -4
```

The code below implements this algorithm.

```
int negate(int a) {
    int neg = 0;
    int newSign = a < 0 ? 1 : -1;
    int delta = newSign;
    while (a != 0) {
        boolean differentSigns = (a + delta > 0) != (a > 0);
        if (a + delta != 0 && differentSigns) { // If delta is too big, reset it.
            delta = newSign;
        }
        neg += delta;
        a += delta;
        delta += delta; // Double the delta
    }
    return neg;
}
```

Figuring out the runtime here takes a bit of calculation.

Observe that reducing a by half takes `O(log a)` work. Why? For each round of "reduce a by half", the absolute values of a and `delta` always add up to the same number. The values of `delta` and a will converge at $\frac{a}{2}$. Since `delta` is being doubled each time, it will take `O(log a)` steps to reach half of a.

We do O(log a) rounds.

1. Reducing a to $\frac{a}{2}$ takes `O(log a)` time.
2. Reducing $\frac{a}{2}$ to $\frac{a}{4}$ takes $O(\log \frac{a}{2})$ time.
3. Reducing $\frac{a}{4}$ to $\frac{a}{8}$ takes $O(\log \frac{a}{4})$ time.

... As so on, for `O(log a)` rounds.

The runtime therefore is $O(\log a + \log(\frac{a}{2}) + \log(\frac{a}{4}) + \ldots)$, with `O(log a)` terms in the expression.

Recall two rules of logs:

- $\log(xy) = \log x + \log y$
- $\log(\frac{x}{y}) = \log x - \log y$.

If we apply this to the above expression, we get:

1. $O(\log a + \log(\frac{a}{2}) + \log(\frac{a}{4}) + \ldots)$
2. $O(\log a + (\log a - \log 2) + (\log a - \log 4) + (\log a - \log 8) + \ldots$
3. $O((\log a)*(\log a) - (\log 2 + \log 4 + \log 8 + \ldots + \log a))$ // $O(\log a)$ terms
4. $O((\log a)*(\log a) - (1 + 2 + 3 + \ldots + \log a))$ // computing the values of logs
5. $O((\log a)*(\log a) - \frac{(\log a)(1 + \log a)}{2})$ // apply equation for sum of 1 through k
6. $O((\log a)^2)$ // drop second term from step 5

Therefore, the runtime is $O((\log a)^2)$.

This math is considerably more complicated than most people would be able to do (or expected to do) in an interview. You could make a simplification: You do $O(\log a)$ rounds and the longest round takes $O(\log a)$ work. Therefore, as an upper bound, `negate` takes $O((\log a)^2)$ time. In this case, the upper bound happens to be the true time.

There are some faster solutions too. For example, rather than resetting `delta` to 1 at each round, we could change `delta` to its previous value. This would have the effect of `delta` "counting up" by multiples of two, and then "counting down" by multiples of two. The runtime of this approach would be $O(\log a)$. However, this implementation would require a stack, division, or bit shifting—any of which might violate the spirit of the problem. You could certainly discuss those implementations with your interviewer though.

Multiplication

The connection between addition and multiplication is equally straightforward. To multiply a by b, we just add a to itself b times.

```
/* Multiply a by b by adding a to itself b times */
int multiply(int a, int b) {
  if (a < b) {
    return multiply(b, a); // algorithm is faster if b < a
  }
  int sum = 0;
  for (int i = abs(b); i > 0; i = minus(i, 1)) {
    sum += a;
  }
  if (b < 0) {
    sum = negate(sum);
  }
  return sum;
}

/* Return absolute value */
int abs(int a) {
  if (a < 0) {
    return negate(a);
  } else {
    return a;
  }
}
```

The one thing we need to be careful of in the above code is to properly handle multiplication of negative numbers. If b is negative, we need to flip the value of `sum`. So, what this code really does is:

```
multiply(a, b) <-- abs(b) * a * (-1 if b < 0).
```

We also implemented a simple `abs` function to help.

Division

Of the three operations, division is certainly the hardest. The good thing is that we can use the `multiply`, `subtract`, and `negate` methods now to implement `divide`.

We are trying to compute x where $X = {}^{a}/_{b}$. Or, to put this another way, find x where a = bx. We've now changed the problem into one that can be stated with something we know how to do: multiplication.

We could implement this by multiplying b by progressively higher values, until we reach a. That would be fairly inefficient, particularly given that our implementation of `multiply` involves a lot of adding.

Alternatively, we can look at the equation a = xb to see that we can compute x by adding b to itself repeatedly until we reach a. The number of times we need to do that will equal x.

Of course, a might not be evenly divisible by b, and that's okay. Integer division, which is what we've been asked to implement, is supposed to truncate the result.

The code below implements this algorithm.

```
int divide(int a, int b) throws java.lang.ArithmeticException {
  if (b == 0) {
    throw new java.lang.ArithmeticException("ERROR");
  }
  int absa = abs(a);
  int absb = abs(b);

  int product = 0;
  int x = 0;
  while (product + absb <= absa) { /* don't go past a */
    product += absb;
    x++;
  }

  if ((a < 0 && b < 0) || (a > 0 && b > 0)) {
    return x;
  } else {
    return negate(x);
  }
}
```

In tackling this problem, you should be aware of the following:

- A logical approach of going back to what exactly multiplication and division do comes in handy. Remember that. All (good) interview problems can be approached in a logical, methodical way!
- The interviewer is looking for this sort of logical work-your-way-through-it approach.
- This is a great problem to demonstrate your ability to write clean code—specifically, to show your ability to reuse code. For example, if you were writing this solution and didn't put `negate` in its own method, you should move it into its own method once you see that you'll use it multiple times.
- Be careful about making assumptions while coding. Don't assume that the numbers are all positive or that a is bigger than b.

16.10 Living People: Given a list of people with their birth and death years, implement a method to compute the year with the most number of people alive. You may assume that all people were born between 1900 and 2000 (inclusive). If a person was alive during any portion of that year, they should be included in that year's count. For example, Person (birth = 1908, death = 1909) is included in the counts for both 1908 and 1909.

pg 182

SOLUTION

The first thing we should do is outline what this solution will look like. The interview question hasn't specified the exact form of input. In a real interview, we could ask the interviewer how the input is structured. Alternatively, you can explicitly state your (reasonable) assumptions.

Here, we'll need to make our own assumptions. We will assume that we have an array of simple `Person` objects:

```
public class Person {
   public int birth;
   public int death;
   public Person(int birthYear, int deathYear) {
      birth = birthYear;
      death = deathYear;
   }
}
```

We could have also given `Person` a `getBirthYear()` and `getDeathYear()` objects. Some would argue that's better style, but for compactness and clarity, we'll just keep the variables public.

The important thing here is to actually use a `Person` object. This shows better style than, say, having an integer array for birth years and an integer array for death years (with an implicit association of `births[i]` and `deaths[i]` being associated with the same person). You don't get a lot of chances to demonstrate great coding style, so it's valuable to take the ones you get.

With that in mind, let's start with a brute force algorithm.

Brute Force

The brute force algorithm falls directly out from the wording of the problem. We need to find the year with the most number of people alive. Therefore, we go through each year and check how many people are alive in that year.

```
int maxAliveYear(Person[] people, int min, int max) {
   int maxAlive = 0;
   int maxAliveYear = min;

   for (int year = min; year <= max; year++) {
      int alive = 0;
      for (Person person : people) {
         if (person.birth <= year && year <= person.death) {
            alive++;
         }
      }
      if (alive > maxAlive) {
         maxAlive = alive;
         maxAliveYear = year;
      }
   }
```

```

  return maxAliveYear;
}
```

Note that we have passed in the values for the min year (1900) and max year (2000). We shouldn't hard code these values.

The runtime of this is O(RP), where R is the range of years (100 in this case) and P is the number of people.

Slightly Better Brute Force

A slightly better way of doing this is to create an array where we track the number of people born in each year. Then, we iterate through the list of people and increment the array for each year they are alive.

```
1  int maxAliveYear(Person[] people, int min, int max) {
2    int[] years = createYearMap(people, min, max);
3    int best = getMaxIndex(years);
4    return best + min;
5  }
6
7  /* Add each person's years to a year map. */
8  int[] createYearMap(Person[] people, int min, int max) {
9    int[] years = new int[max - min + 1];
10   for (Person person : people) {
11     incrementRange(years, person.birth - min, person.death - min);
12   }
13   return years;
14 }
15
16 /* Increment array for each value between left and right. */
17 void incrementRange(int[] values, int left, int right) {
18   for (int i = left; i <= right; i++) {
19     values[i]++;
20   }
21 }
22
23 /* Get index of largest element in array. */
24 int getMaxIndex(int[] values) {
25   int max = 0;
26   for (int i = 1; i < values.length; i++) {
27     if (values[i] > values[max]) {
28       max = i;
29     }
30   }
31   return max;
32 }
```

Be careful on the size of the array in line 9. If the range of years is 1900 to 2000 inclusive, then that's 101 years, not 100. That is why the array has size `max - min + 1`.

Let's think about the runtime by breaking this into parts.

- We create an R-sized array, where R is the min and max years.
- Then, for P people, we iterate through the years (Y) that the person is alive.
- Then, we iterate through the R-sized array again.

The total runtime is O(PY + R). In the worst case, Y is R and we have done no better than we did in the first algorithm.

More Optimal

Let's create an example. (In fact, an example is really helpful in almost all problems. Ideally, you've already done this.) Each column below is matched, so that the items correspond to the same person. For compactness, we'll just write the last two digits of the year.

```
birth: 12  20  10  01  10  23  13  90  83  75
death: 15  90  98  72  98  82  98  98  99  94
```

It's worth noting that it doesn't really matter whether these years are matched up. Every birth adds a person and every death removes a person.

Since we don't actually need to match up the births and deaths, let's sort both. A sorted version of the years might help us solve the problem.

```
birth: 01  10  10  12  13  20  23  75  83  90
death: 15  72  82  90  94  98  98  98  98  99
```

We can try walking through the years.

- At year 0, no one is alive.
- At year 1, we see one birth.
- At years 2 through 9, nothing happens.
- Let's skip ahead until year 10, when we have two births. We now have three people alive.
- At year 15, one person dies. We are now down to two people alive.
- And so on.

If we walk through the two arrays like this, we can track the number of people alive at each point.

```
int maxAliveYear(Person[] people, int min, int max) {
  int[] births = getSortedYears(people, true);
  int[] deaths = getSortedYears(people, false);

  int birthIndex = 0;
  int deathIndex = 0;
  int currentlyAlive = 0;
  int maxAlive = 0;
  int maxAliveYear = min;

  /* Walk through arrays. */
  while (birthIndex < births.length) {
    if (births[birthIndex] <= deaths[deathIndex]) {
      currentlyAlive++; // include birth
      if (currentlyAlive > maxAlive) {
        maxAlive = currentlyAlive;
        maxAliveYear = births[birthIndex];
      }
      birthIndex++; // move birth index
    } else if (births[birthIndex] > deaths[deathIndex]) {
      currentlyAlive--; // include death
      deathIndex++; // move death index
    }
  }

  return maxAliveYear;
}

/* Copy birth years or death years (depending on the value of copyBirthYear into
```

```
30   * integer array, then sort array. */
31  int[] getSortedYears(Person[] people, boolean copyBirthYear) {
32    int[] years = new int[people.length];
33    for (int i = 0; i < people.length; i++) {
34      years[i] = copyBirthYear ? people[i].birth : people[i].death;
35    }
36    Arrays.sort(years);
37    return years;
38  }
```

There are some very easy things to mess up here.

On line 13, we need to think carefully about whether this should be a less than (<) or a less than or equals (<=). The scenario we need to worry about is that you see a birth and death in the same year. (It doesn't matter whether the birth and death is from the same person.)

When we see a birth and death from the same year, we want to include the birth *before* we include the death, so that we count this person as alive for that year. That is why we use a <= on line 13.

We also need to be careful about where we put the updating of `maxAlive` and `maxAliveYear`. It needs to be after the `currentAlive++`, so that it takes into account the updated total. But it needs to be before `birthIndex++`, or we won't have the right year.

This algorithm will take `O(P log P)` time, where P is the number of people.

More Optimal (Maybe)

Can we optimize this further? To optimize this, we'd need to get rid of the sorting step. We're back to dealing with unsorted values:

```
birth: 12  20  10  01  10  23  13  90  83  75
death: 15  90  98  72  98  82  98  98  99  94
```

Earlier, we had logic that said that a birth is just adding a person and a death is just subtracting a person. Therefore, let's represent the data using the logic:

```
01: +1      10: +1      10: +1      12: +1      13: +1
15: -1      20: +1      23: +1      72: -1      75: +1
82: -1      83: +1      90: +1      90: -1      94: -1
98: -1      98: -1      98: -1      98: -1      99: -1
```

We can create an array of the years, where the value at `array[year]` indicates how the population changed in that year. To create this array, we walk through the list of people and increment when they're born and decrement when they die.

Once we have this array, we can walk through each of the years, tracking the current population as we go (adding the value at `array[year]` each time).

This logic is reasonably good, but we should think about it more. Does it really work?

One edge case we should consider is when a person dies the same year that they're born. The increment and decrement operations will cancel out to give 0 population change. According to the wording of the problem, this person should be counted as living in that year.

In fact, the "bug" in our algorithm is broader than that. This same issue applies to all people. People who die in 1908 shouldn't be removed from the population count until 1909.

There's a simple fix: instead of decrementing `array[deathYear]`, we should decrement `array[deathYear + 1]`.

```
int maxAliveYear(Person[] people, int min, int max) {
```

```
    /* Build population delta array. */
    int[] populationDeltas = getPopulationDeltas(people, min, max);
    int maxAliveYear = getMaxAliveYear(populationDeltas);
    return maxAliveYear + min;
  }

  /* Add birth and death years to deltas array. */
  int[] getPopulationDeltas(Person[] people, int min, int max) {
    int[] populationDeltas = new int[max - min + 2];
    for (Person person : people) {
      int birth = person.birth - min;
      populationDeltas[birth]++;

      int death = person.death - min;
      populationDeltas[death + 1]--;
    }
    return populationDeltas;
  }

  /* Compute running sums and return index with max. */
  int getMaxAliveYear(int[] deltas) {
    int maxAliveYear = 0;
    int maxAlive = 0;
    int currentlyAlive = 0;
    for (int year = 0; year < deltas.length; year++) {
      currentlyAlive += deltas[year];
      if (currentlyAlive > maxAlive) {
        maxAliveYear = year;
        maxAlive = currentlyAlive;
      }
    }

    return maxAliveYear;
  }
```

This algorithm takes `O(R + P)` time, where R is the range of years and P is the number of people. Although `O(R + P)` might be faster than `O(P log P)` for many expected inputs, you cannot directly compare the speeds to say that one is faster than the other.

16.11 Diving Board: You are building a diving board by placing a bunch of planks of wood end-to-end. There are two types of planks, one of length `shorter` and one of length `longer`. You must use exactly K planks of wood. Write a method to generate all possible lengths for the diving board.

pg 182

SOLUTION

One way to approach this is to think about the choices we make as we're building a diving board. This leads us to a recursive algorithm.

Recursive Solution

For a recursive solution, we can imagine ourselves building a diving board. We make K decisions, each time choosing which plank we will put on next. Once we've put on K planks, we have a complete diving board and we can add this to the list (assuming we haven't seen this length before).

We can follow this logic to write recursive code. Note that we don't need to track the sequence of planks. All we need to know is the current length and the number of planks remaining.

```
HashSet<Integer> allLengths(int k, int shorter, int longer) {
  HashSet<Integer> lengths = new HashSet<Integer>();
  getAllLengths(k, 0, shorter, longer, lengths);
  return lengths;
}

void getAllLengths(int k, int total, int shorter, int longer,
                   HashSet<Integer> lengths) {
  if (k == 0) {
    lengths.add(total);
    return;
  }
  getAllLengths(k - 1, total + shorter, shorter, longer, lengths);
  getAllLengths(k - 1, total + longer, shorter, longer, lengths);
}
```

We've added each length to a hash set. This will automatically prevent adding duplicates.

This algorithm takes $O(2^K)$ time, since there are two choices at each recursive call and we recurse to a depth of K.

Memoization Solution

As in many recursive algorithms (especially those with exponential runtimes), we can optimize this through memorization (a form of dynamic programming).

Observe that some of the recursive calls will be essentially equivalent. For example, picking plank 1 and then plank 2 is equivalent to picking plank 2 and then plank 1.

Therefore, if we've seen this `(total, plank count)` pair before then we stop this recursive path. We can do this using a `HashSet` with a key of `(total, plank count)`.

> Many candidates will make a mistake here. Rather than stopping only when they've seen `(total, plank count)`, they'll stop whenever they've seen just `total` before. This is incorrect. Seeing two planks of length 1 is not the same thing as one plank of length 2, because there are different numbers of planks remaining. In memoization problems, be very careful about what you choose for your key.

The code for this approach is very similar to the earlier approach.

```
HashSet<Integer> allLengths(int k, int shorter, int longer) {
  HashSet<Integer> lengths = new HashSet<Integer>();
  HashSet<String> visited = new HashSet<String>();
  getAllLengths(k, 0, shorter, longer, lengths, visited);
  return lengths;
}

void getAllLengths(int k, int total, int shorter, int longer,
                   HashSet<Integer> lengths, HashSet<String> visited) {
  if (k == 0) {
    lengths.add(total);
    return;
  }
  String key = k + " " + total;
```

```
    if (visited.contains(key)) {
      return;
    }
    getAllLengths(k - 1, total + shorter, shorter, longer, lengths, visited);
    getAllLengths(k - 1, total + longer, shorter, longer, lengths, visited);
    visited.add(key);
  }
```

For simplicity, we've set the key to be a string representation of `total` and the current plank count. Some people may argue it's better to use a data structure to represent this pair. There are benefits to this, but there are drawbacks as well. It's worth discussing this tradeoff with your interviewer.

The runtime of this algorithm is a bit tricky to figure out.

One way we can think about the runtime is by understanding that we're basically filling in a table of `SUMS x PLANK COUNTS`. The biggest possible sum is `K * LONGER` and the biggest possible plank count is K. Therefore, the runtime will be no worse than $O(K^2 * \text{LONGER})$.

Of course, a bunch of those sums will never actually be reached. How many unique sums can we get? Observe that any path with the same number of each type of planks will have the same sum. Since we can have at most K planks of each type, there are only K different sums we can make. Therefore, the table is really KxK, and the runtime is $O(K^2)$.

Optimal Solution

If you re-read the prior paragraph, you might notice something interesting. There are only K distinct sums we can get. Isn't that the whole point of the problem—to find all possible sums?

We don't actually need to go through all arrangements of planks. We just need to go through all unique sets of K planks (sets, not orders!). There are only K ways of picking K planks if we only have two possible types: {0 of type A, K of type B}, {1 of type A, K-1 of type B}, {2 of type A, K-2 of type B}, ...

This can be done in just a simple for loop. At each "sequence", we just compute the sum.

```
HashSet<Integer> allLengths(int k, int shorter, int longer) {
  HashSet<Integer> lengths = new HashSet<Integer>();
  for (int nShorter = 0; nShorter <= k; nShorter++) {
    int nLonger = k - nShorter;
    int length = nShorter * shorter + nLonger * longer;
    lengths.add(length);
  }
  return lengths;
}
```

We've used a `HashSet` here for consistency with the prior solutions. This isn't really necessary though, since we shouldn't get any duplicates. We could instead use an `ArrayList`. If we do this, though, we just need to handle an edge case where the two types of planks are the same length. In this case, we would just return an `ArrayList` of size 1.

16.12 XML Encoding: Since XML is very verbose, you are given a way of encoding it where each tag gets mapped to a pre-defined integer value. The language/grammar is as follows:

```
Element   --> Tag Attributes END Children END
Attribute --> Tag Value
END       --> 0
Tag       --> some predefined mapping to int
Value     --> string value
```

For example, the following XML might be converted into the compressed string below (assuming a mapping of `family -> 1, person ->2, firstName -> 3, lastName -> 4, state -> 5`).

```
<family lastName="McDowell" state="CA">
  <person firstName="Gayle">Some Message</person>
</family>
```

Becomes:

```
1 4 McDowell 5 CA 0 2 3 Gayle 0 Some Message 0 0
```

Write code to print the encoded version of an XML element (passed in `Element` and `Attribute` objects).

pg 182

SOLUTION

Since we know the element will be passed in as an `Element` and `Attribute`, our code is reasonably simple. We can implement this by applying a tree-like approach.

We repeatedly call `encode()` on parts of the XML structure, handling the code in slightly different ways depending on the type of the XML element.

```
void encode(Element root, StringBuilder sb) {
  encode(root.getNameCode(), sb);
  for (Attribute a : root.attributes) {
    encode(a, sb);
  }
  encode("0", sb);
  if (root.value != null && root.value != "") {
    encode(root.value, sb);
  } else {
    for (Element e : root.children) {
      encode(e, sb);
    }
  }
  encode("0", sb);
}

void encode(String v, StringBuilder sb) {
  sb.append(v);
  sb.append(" ");
}

void encode(Attribute attr, StringBuilder sb) {
  encode(attr.getTagCode(), sb);
  encode(attr.value, sb);
}

```

```
27 String encodeToString(Element root) {
28    StringBuilder sb = new StringBuilder();
29    encode(root, sb);
30    return sb.toString();
31 }
```

Observe in line 17, the use of the very simple `encode` method for a string. This is somewhat unnecessary; all it does is insert the string and a space following it. However, using this method is a nice touch as it ensures that every element will be inserted with a space surrounding it. Otherwise, it might be easy to break the encoding by forgetting to append the empty string.

16.13 Bisect Squares: Given two squares on a two-dimensional plane, find a line that would cut these two squares in half. Assume that the top and the bottom sides of the square run parallel to the x-axis.

pg 182

SOLUTION

Before we start, we should think about what exactly this problem means by a "line." Is a line defined by a slope and a y-intercept? Or by any two points on the line? Or, should the line be really a line segment, which starts and ends at the edges of the squares?

We will assume, since it makes the problem a bit more interesting, that we mean the third option: that the line should end at the edges of the squares. In an interview situation, you should discuss this with your interviewer.

This line that cuts two squares in half must connect the two middles. We can easily calculate the slope, knowing that `slope` $= \frac{y1 - y2}{x1 - x2}$. Once we calculate the slope using the two middles, we can use the same equation to calculate the start and end points of the line segment.

In the below code, we will assume the origin `(0, 0)` is in the upper left-hand corner.

```
public class Square {
   ...
   public Point middle() {
      return new Point((this.left + this.right) / 2.0,
                       (this.top + this.bottom) / 2.0);
   }

   /* Return the point where the line segment connecting mid1 and mid2 intercepts
    * the edge of square 1. That is, draw a line from mid2 to mid1, and continue it
    * out until the edge of the square. */
   public Point extend(Point mid1, Point mid2, double size) {
      /* Find what direction the line mid2 -> mid1 goes. */
      double xdir = mid1.x < mid2.x ? -1 : 1;
      double ydir = mid1.y < mid2.y ? -1 : 1;

      /* If mid1 and mid2 have the same x value, then the slope calculation will
       * throw a divide by 0 exception. So, we compute this specially. */
      if (mid1.x == mid2.x) {
         return new Point(mid1.x, mid1.y + ydir * size / 2.0);
      }

      double slope = (mid1.y - mid2.y) / (mid1.x - mid2.x);
      double x1 = 0;
      double y1 = 0;

```

```
        /* Calculate slope using the equation (y1 - y2) / (x1 - x2).
         * Note: if the slope is "steep" (>1) then the end of the line segment will
         * hit size / 2 units away from the middle on the y axis. If the slope is
         * "shallow" (<1) the end of the line segment will hit size / 2 units away
         * from the middle on the x axis. */
        if (Math.abs(slope) == 1) {
            x1 = mid1.x + xdir * size / 2.0;
            y1 = mid1.y + ydir * size / 2.0;
        } else if (Math.abs(slope) < 1) { // shallow slope
            x1 = mid1.x + xdir * size / 2.0;
            y1 = slope * (x1 - mid1.x) + mid1.y;
        } else { // steep slope
            y1 = mid1.y + ydir * size / 2.0;
            x1 = (y1 - mid1.y) / slope + mid1.x;
        }
        return new Point(x1, y1);
    }

    public Line cut(Square other) {
        /* Calculate where a line between each middle would collide with the edges of
         * the squares */
        Point p1 = extend(this.middle(), other.middle(), this.size);
        Point p2 = extend(this.middle(), other.middle(), -1 * this.size);
        Point p3 = extend(other.middle(), this.middle(), other.size);
        Point p4 = extend(other.middle(), this.middle(), -1 * other.size);

        /* Of above points, find start and end of lines. Start is farthest left (with
         * top most as a tie breaker) and end is farthest right (with bottom most as
         * a tie breaker. */
        Point start = p1;
        Point end = p1;
        Point[] points = {p2, p3, p4};
        for (int i = 0; i < points.length; i++) {
            if (points[i].x < start.x ||
                (points[i].x == start.x && points[i].y < start.y)) {
                start = points[i];
            } else if (points[i].x > end.x ||
                        (points[i].x == end.x && points[i].y > end.y)) {
                end = points[i];
            }
        }

        return new Line(start, end);
    }
```

The main goal of this problem is to see how careful you are about coding. It's easy to glance over the special cases (e.g., the two squares having the same middle). You should make a list of these special cases before you start the problem and make sure to handle them appropriately. This is a question that requires careful and thorough testing.

16.14 Best Line: Given a two-dimensional graph with points on it, find a line which passes the most number of points.

pg 183

SOLUTION

This solution seems quite straightforward at first. And it is—sort of.

We just "draw" an infinite line (that is, not a line segment) between every two points and, using a hash table, track which line is the most common. This will take $O(N^2)$ time, since there are N^2 line segments.

We will represent a line as a slope and y-intercept (as opposed to a pair of points), which allows us to easily check to see if the line from `(x1, y1)` to `(x2, y2)` is equivalent to the line from `(x3, y3)` to `(x4, y4)`.

To find the most common line then, we just iterate through all lines segments, using a hash table to count the number of times we've seen each line. Easy enough!

However, there's one little complication. We're defining two lines to be equal if the lines have the same slope and y-intercept. We are then, furthermore, hashing the lines based on these values (specifically, based on the slope). The problem is that floating point numbers cannot always be represented accurately in binary. We resolve this by checking if two floating point numbers are within an `epsilon` value of each other.

What does this mean for our hash table? It means that two lines with "equal" slopes may not be hashed to the same value. To solve this, we will round the slope down to the next epsilon and use this `flooredSlope` as the hash key. Then, to retrieve all lines that are *potentially* equal, we will search the hash table at three spots: `flooredSlope`, `flooredSlope - epsilon`, and `flooredSlope + epsilon`. This will ensure that we've checked out all lines that might be equal.

```
/* Find line that goes through most number of points. */
Line findBestLine(GraphPoint[] points) {
  HashMapList<Double, Line> linesBySlope = getListOfLines(points);
  return getBestLine(linesBySlope);
}

/* Add each pair of points as a line to the list. */
HashMapList<Double, Line> getListOfLines(GraphPoint[] points) {
  HashMapList<Double, Line> linesBySlope = new HashMapList<Double, Line>();
  for (int i = 0; i < points.length; i++) {
    for (int j = i + 1; j < points.length; j++) {
      Line line = new Line(points[i], points[j]);
      double key = Line.floorToNearestEpsilon(line.slope);
      linesBySlope.put(key, line);
    }
  }
  return linesBySlope;
}

/* Return the line with the most equivalent other lines. */
Line getBestLine(HashMapList<Double, Line> linesBySlope) {
  Line bestLine = null;
  int bestCount = 0;

  Set<Double> slopes = linesBySlope.keySet();

  for (double slope : slopes) {
```

```
      ArrayList<Line> lines = linesBySlope.get(slope);
      for (Line line : lines) {
         /* count lines that are equivalent to current line */
         int count = countEquivalentLines(linesBySlope, line);

         /* if better than current line, replace it */
         if (count > bestCount) {
            bestLine = line;
            bestCount = count;
            bestLine.Print();
            System.out.println(bestCount);
         }
      }
   }
   return bestLine;
}

/* Check hashmap for lines that are equivalent. Note that we need to check one
 * epsilon above and below the actual slope since we're defining two lines as
 * equivalent if they're within an epsilon of each other. */
int countEquivalentLines(HashMapList<Double, Line> linesBySlope, Line line) {
   double key = Line.floorToNearestEpsilon(line.slope);
   int count = countEquivalentLines(linesBySlope.get(key), line);
   count += countEquivalentLines(linesBySlope.get(key - Line.epsilon), line);
   count += countEquivalentLines(linesBySlope.get(key + Line.epsilon), line);
   return count;
}

/* Count lines within an array of lines which are "equivalent" (slope and
 * y-intercept are within an epsilon value) to a given line */
int countEquivalentLines(ArrayList<Line> lines, Line line) {
   if (lines == null) return 0;

   int count = 0;
   for (Line parallelLine : lines) {
      if (parallelLine.isEquivalent(line)) {
         count++;
      }
   }
   return count;
}

public class Line {
   public static double epsilon = .0001;
   public double slope, intercept;
   private boolean infinite_slope = false;

   public Line(GraphPoint p, GraphPoint q) {
      if (Math.abs(p.x - q.x) > epsilon) { // if x's are different
         slope = (p.y - q.y) / (p.x - q.x); // compute slope
         intercept = p.y - slope * p.x; // y intercept from y=mx+b
      } else {
         infinite_slope = true;
         intercept = p.x; // x-intercept, since slope is infinite
      }
   }
```

```
    public static double floorToNearestEpsilon(double d) {
       int r = (int) (d / epsilon);
       return ((double) r) * epsilon;
    }

    public boolean isEquivalent(double a, double b) {
       return (Math.abs(a - b) < epsilon);
    }

    public boolean isEquivalent(Object o) {
       Line l = (Line) o;
       if (isEquivalent(l.slope, slope) && isEquivalent(l.intercept, intercept) &&
           (infinite_slope == l.infinite_slope)) {
         return true;
       }
       return false;
    }
}

/* HashMapList<String, Integer> is a HashMap that maps from Strings to
 * ArrayList<Integer>. See appendix for implementation. */
```

We need to be careful about the calculation of the slope of a line. The line might be completely vertical, which means that it doesn't have a y-intercept and its slope is infinite. We can keep track of this in a separate flag (`infinite_slope`). We need to check this condition in the equals method.

16.15 Master Mind: The Game of Master Mind is played as follows:

The computer has four slots, and each slot will contain a ball that is red (R), yellow (Y), green (G) or blue (B). For example, the computer might have RGGB (Slot #1 is red, Slots #2 and #3 are green, Slot #4 is blue).

You, the user, are trying to guess the solution. You might, for example, guess YRGB.

When you guess the correct color for the correct slot, you get a "hit." If you guess a color that exists but is in the wrong slot, you get a "pseudo-hit." Note that a slot that is a hit can never count as a pseudo-hit.

For example, if the actual solution is RGBY and you guess GGRR, you have one hit and one pseudo-hit.

Write a method that, given a guess and a solution, returns the number of hits and pseudo-hits.

pg 183

SOLUTION

This problem is straightforward, but it's surprisingly easy to make little mistakes. You should check your code *extremely* thoroughly, on a variety of test cases.

We'll implement this code by first creating a frequency array which stores how many times each character occurs in `solution`, excluding times when the slot is a "hit." Then, we iterate through `guess` to count the number of pseudo-hits.

The code below implements this algorithm.

```
class Result {
   public int hits = 0;
```

```
    public int pseudoHits = 0;

    public String toString() {
      return "(" + hits + ", " + pseudoHits + ")";
    }
  }

int code(char c) {
  switch (c) {
  case 'B':
    return 0;
  case 'G':
    return 1;
  case 'R':
    return 2;
  case 'Y':
    return 3;
  default:
    return -1;
  }
}

int MAX_COLORS = 4;

Result estimate(String guess, String solution) {
  if (guess.length() != solution.length()) return null;

  Result res = new Result();
  int[] frequencies = new int[MAX_COLORS];

  /* Compute hits and build frequency table */
  for (int i = 0; i < guess.length(); i++) {
    if (guess.charAt(i) == solution.charAt(i)) {
      res.hits++;
    } else {
      /* Only increment the frequency table (which will be used for pseudo-hits)
       * if it's not a hit. If it's a hit, the slot has already been "used." */
      int code = code(solution.charAt(i));
      frequencies[code]++;
    }
  }

  /* Compute pseudo-hits */
  for (int i = 0; i < guess.length(); i++) {
    int code = code(guess.charAt(i));
    if (code >= 0 && frequencies[code] > 0 &&
        guess.charAt(i) != solution.charAt(i)) {
      res.pseudoHits++;
      frequencies[code]--;
    }
  }
  return res;
}
```

Note that the easier the algorithm for a problem is, the more important it is to write clean and correct code. In this case, we've pulled `code(char c)` into its own method, and we've created a `Result` class to hold the result, rather than just printing it.

16.16 Sub Sort: Given an array of integers, write a method to find indices `m` and `n` such that if you sorted elements `m` through `n`, the entire array would be sorted. Minimize `n - m` (that is, find the smallest such sequence).

EXAMPLE

Input: `1, 2, 4, 7, 10, 11, 7, 12, 6, 7, 16, 18, 19`

Output: `(3, 9)`

pg 183

SOLUTION

Before we begin, let's make sure we understand what our answer will look like. If we're looking for just two indices, this indicates that some middle section of the array will be sorted, with the start and end of the array already being in order.

Now, let's approach this problem by looking at an example.

```
1, 2, 4, 7, 10, 11, 8, 12, 5, 6, 16, 18, 19
```

Our first thought might be to just find the longest increasing subsequence at the beginning and the longest increasing subsequence at the end.

```
left:   1, 2, 4, 7, 10, 11
middle: 8, 12
right:  5, 6, 16, 18, 19
```

These subsequences are easy to generate. We just start from the left and the right sides, and work our way inward. When an element is out of order, then we have found the end of our increasing/decreasing subsequence.

In order to solve our problem, though, we would need to be able to sort the middle part of the array and, by doing just that, get all the elements in the array in order. Specifically, the following would have to be true:

```
/* all items on left are smaller than all items in middle */
min(middle) > end(left)

/* all items in middle are smaller than all items in right */
max(middle) < start(right)
```

Or, in other words, for all elements:

```
left < middle < right
```

In fact, this condition will *never* be met. The middle section is, by definition, the elements that were out of order. That is, it is *always* the case that `left.end > middle.start` and `middle.end > right.start`. Thus, you cannot sort the middle to make the entire array sorted.

But, what we can do is *shrink* the left and right subsequences until the earlier conditions are met. We need the left part to be smaller than all the elements in the middle and right side, and the right part to be bigger than all the elements on the left and right side.

Let `min` equal `min(middle and right side)` and `max` equal `max(middle and left side)`. Observe that since the right and left sides are already in sorted order, we only actually need to check their start or end point.

On the left side, we start with the end of the subsequence (value 11, at element 5) and move to the left. The value `min` equals 5. Once we find an element `i` such that `array[i] < min`, we know that we could sort the middle and have that part of the array appear in order.

Then, we do a similar thing on the right side. The value `max` equals 12. So, we begin with the start of the right subsequence (value 6) and move to the right. We compare the max of 12 to 6, then 7, then 16. When reach 16, we know that no elements smaller than 12 could be after it (since it's an increasing subsequence). Thus, the middle of the array could now be sorted to make the entire array sorted.

The following code implements this algorithm.

```
void findUnsortedSequence(int[] array) {
  // find left subsequence
  int end_left = findEndOfLeftSubsequence(array);
  if (end_left >= array.length - 1) return; // Already sorted

  // find right subsequence
  int start_right = findStartOfRightSubsequence(array);

  // get min and max
  int max_index = end_left; // max of left side
  int min_index = start_right; // min of right side
  for (int i = end_left + 1; i < start_right; i++) {
    if (array[i] < array[min_index]) min_index = i;
    if (array[i] > array[max_index]) max_index = i;
  }

  // slide left until less than array[min_index]
  int left_index = shrinkLeft(array, min_index, end_left);

  // slide right until greater than array[max_index]
  int right_index = shrinkRight(array, max_index, start_right);

  System.out.println(left_index + “ “ + right_index);
}

int findEndOfLeftSubsequence(int[] array) {
  for (int i = 1; i < array.length; i++) {
    if (array[i] < array[i - 1]) return i - 1;
  }
  return array.length - 1;
}

int findStartOfRightSubsequence(int[] array) {
  for (int i = array.length - 2; i >= 0; i--) {
    if (array[i] > array[i + 1]) return i + 1;
  }
  return 0;
}

int shrinkLeft(int[] array, int min_index, int start) {
  int comp = array[min_index];
  for (int i = start - 1; i >= 0; i--) {
    if (array[i] <= comp) return i + 1;
  }
  return 0;
}

int shrinkRight(int[] array, int max_index, int start) {
  int comp = array[max_index];
  for (int i = start; i < array.length; i++) {
```

```
        if (array[i] >= comp) return i - 1;
    }
    return array.length - 1;
}
```

Note the use of other methods in this solution. Although we could have jammed it all into one method, it would have made the code a lot harder to understand, maintain, and test. In your interview coding, you should prioritize these aspects.

16.17 Contiguous Sequence: You are given an array of integers (both positive and negative). Find the contiguous sequence with the largest sum. Return the sum.

EXAMPLE

Input: `2, -8, 3, -2, 4, -10`

Output: `5 (i.e., {3, -2, 4})`

pg 183

SOLUTION

This is a challenging problem, but an extremely common one. Let's approach this by looking at an example:

```
2   3   -8   -1   2   4   -2   3
```

If we think about our array as having alternating sequences of positive and negative numbers, we can observe that we would never include only part of a negative subsequence or part of a positive sequence. Why would we? Including part of a negative subsequence would make things unnecessarily negative, and we should just instead not include that negative sequence at all. Likewise, including only part of a positive subsequence would be strange, since the sum would be even bigger if we included the whole thing.

For the purposes of coming up with our algorithm, we can think about our array as being a sequence of alternating negative and positive numbers. Each number corresponds to the sum of a subsequence of positive numbers of a subsequence of negative numbers. For the array above, our new reduced array would be:

```
5   -9   6   -2   3
```

This doesn't give away a great algorithm immediately, but it does help us to better understand what we're working with.

Consider the array above. Would it ever make sense to have `{5, -9}` in a subsequence? No. These numbers sum to `-4`, so we're better off not including either number, or possibly just having the sequence be just `{5}`).

When would we want negative numbers included in a subsequence? Only if it allows us to join two positive subsequences, each of which have a sum greater than the negative value.

We can approach this in a step-wise manner, starting with the first element in the array.

When we look at 5, this is the biggest sum we've seen so far. We set `maxSum` to 5, and `sum` to 5. Then, we consider -9. If we added it to `sum`, we'd get a negative value. There's no sense in extending the subsequence from 5 to -9 (which "reduces" to a sequence of just -4), so we just reset the value of `sum`.

Now, we consider 6. This subsequence is greater than 5, so we update both `maxSum` and `sum`.

Next, we look at -2. Adding this to 6 will set `sum` to 4. Since this is still a "value add" (when adjoined to another, bigger sequence), we *might* want `{6, -2}` in our max subsequence. We'll update `sum`, but not `maxSum`.

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

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

```
int getMaxSum(int[] a) {
   int maxsum = 0;
   int sum = 0;
   for (int i = 0; i < a.length; i++) {
      sum += a[i];
      if (maxsum < sum) {
         maxsum = sum;
      } else if (sum < 0) {
         sum = 0;
      }
   }
   return maxsum;
}
```

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

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

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

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

pg 183

SOLUTION

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

Brute Force

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

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

```
for each possible substring a
   for each possible substring b
      candidate = buildFromPattern(pattern, a, b)
      if candidate equals value
         return true
```

Ouch.

One easy optimization is to notice that if the pattern starts with 'a', then the a string must start at the beginning of `value`. (Otherwise, the b string must start at the beginning of `value`.) Therefore, there aren't $O(n^2)$ possible values for a; there are $O(n)$.

The algorithm then is to check if the pattern starts with a or b. If it starts with b, we can "invert" it (flipping each 'a' to a 'b' and each 'b' to an 'a') so that it starts with 'a'. Then, iterate through all possible substrings for a (each of which must begin at index 0) and all possible substrings for b (each of which must begin at some character after the end of a). As before, we then compare the string for this pattern with the original string.

This algorithm now takes $O(n^4)$ time.

There's one more minor (optional) optimization we can make. We don't actually need to do this "inversion" if the string starts with 'b' instead of 'a'. The `buildFromPattern` method can take care of this. We can think about the first character in the pattern as the "main" item and the other character as the alternate character. The `buildFromPattern` method can build the appropriate string based on whether 'a' is the main character or alternate character.

```
boolean doesMatch(String pattern, String value) {
  if (pattern.length() == 0) return value.length() == 0;

  int size = value.length();
  for (int mainSize = 0; mainSize < size; mainSize++) {
    String main = value.substring(0, mainSize);
    for (int altStart = mainSize; altStart <= size; altStart++) {
      for (int altEnd = altStart; altEnd <= size; altEnd++) {
        String alt = value.substring(altStart, altEnd);
        String cand = buildFromPattern(pattern, main, alt);
        if (cand.equals(value)) {
          return true;
        }
      }
    }
  }
  return false;
}

String buildFromPattern(String pattern, String main, String alt) {
  StringBuffer sb = new StringBuffer();
  char first = pattern.charAt(0);
  for (char c : pattern.toCharArray()) {
    if (c == first) {
      sb.append(main);
    } else {
      sb.append(alt);
    }
  }
  return sb.toString();
}
```

We should look for a more optimal algorithm.

Optimized

Let's think through our current algorithm. Searching through all values for the main string is fairly fast (it takes $O(n)$ time). It's the alternate string that is so slow: $O(n^2)$ time. We should study how to optimize that.

Suppose we have a pattern like aabab and we're comparing it to the string catcatgocatgo. Once we've picked "cat" as the value for a to try, then the a strings are going to take up nine characters (three a strings with length three each). Therefore, the b strings must take up the remaining four characters, with each having length two. Moreover, we actually know exactly where they must occur, too. If a is cat, and the pattern is aabab, then b must be go.

In other words, once we've picked a, we've picked b too. There's no need to iterate. Gathering some basic stats on pattern (number of as, number of bs, first occurrence of each) and iterating through values for a (or whichever the main string is) will be sufficient.

```
boolean doesMatch(String pattern, String value) {
  if (pattern.length() == 0) return value.length() == 0;

  char mainChar = pattern.charAt(0);
  char altChar = mainChar == 'a' ? 'b' : 'a';
  int size = value.length();

  int countOfMain = countOf(pattern, mainChar);
  int countOfAlt = pattern.length() - countOfMain;
  int firstAlt = pattern.indexOf(altChar);
  int maxMainSize = size / countOfMain;

  for (int mainSize = 0; mainSize <= maxMainSize; mainSize++) {
    int remainingLength = size - mainSize * countOfMain;
    String first = value.substring(0, mainSize);
    if (countOfAlt == 0 || remainingLength % countOfAlt == 0) {
      int altIndex = firstAlt * mainSize;
      int altSize = countOfAlt == 0 ? 0 : remainingLength / countOfAlt;
      String second = countOfAlt == 0 ? "" :
                      value.substring(altIndex, altSize + altIndex);

      String cand = buildFromPattern(pattern, first, second);
      if (cand.equals(value)) {
        return true;
      }
    }
  }
  return false;
}

int countOf(String pattern, char c) {
  int count = 0;
  for (int i = 0; i < pattern.length(); i++) {
    if (pattern.charAt(i) == c) {
      count++;
    }
  }
  return count;
}

String buildFromPattern(...) {  /* same as before */ }
```

This algorithm takes $O(n^2)$, since we iterate through O(n) possibilities for the main string and do $O(n)$ work to build and compare the strings.

Observe that we've also cut down the possibilities for the main string that we try. If there are three instances of the main string, then its length cannot be any more than one third of value.

Optimized (Alternate)

If you don't like the work of building a string only to compare it (and then destroy it), we can eliminate this.

Instead, we can iterate through the values for a and b as before. But this time, to check if the string matches the pattern (given those values for a and b), we walk through `value`, comparing each substring to the first instance of the a and b strings.

```
boolean doesMatch(String pattern, String value) {
  if (pattern.length() == 0) return value.length() == 0;

  char mainChar = pattern.charAt(0);
  char altChar = mainChar == 'a' ? 'b' : 'a';
  int size = value.length();

  int countOfMain = countOf(pattern, mainChar);
  int countOfAlt = pattern.length() - countOfMain;
  int firstAlt = pattern.indexOf(altChar);
  int maxMainSize = size / countOfMain;

  for (int mainSize = 0; mainSize <= maxMainSize; mainSize++) {
    int remainingLength = size - mainSize * countOfMain;
    if (countOfAlt == 0 || remainingLength % countOfAlt == 0) {
      int altIndex = firstAlt * mainSize;
      int altSize = countOfAlt == 0 ? 0 : remainingLength / countOfAlt;
      if (matches(pattern, value, mainSize, altSize, altIndex)) {
        return true;
      }
    }
  }
  return false;
}

/* Iterates through pattern and value. At each character within pattern, checks if
 * this is the main string or the alternate string. Then checks if the next set of
 * characters in value match the original set of those characters (either the main
 * or the alternate. */
boolean matches(String pattern, String value, int mainSize, int altSize,
                int firstAlt) {
  int stringIndex = mainSize;
  for (int i = 1; i < pattern.length(); i++) {
    int size = pattern.charAt(i) == pattern.charAt(0) ? mainSize : altSize;
    int offset = pattern.charAt(i) == pattern.charAt(0) ? 0 : firstAlt;
    if (!isEqual(value, offset, stringIndex, size)) {
      return false;
    }
    stringIndex += size;
  }
  return true;
}

/* Checks if two substrings are equal, starting at given offsets and continuing to
 * size. */
boolean isEqual(String s1, int offset1, int offset2, int size) {
  for (int i = 0; i < size; i++) {
    if (s1.charAt(offset1 + i) != s1.charAt(offset2 + i)) {
      return false;
```

```
        }
    }
    return true;
}
```

This algorithm will still take $O(n^2)$ time, but the benefit is that it can short circuit when matches fail early (which they usually will). The previous algorithm must go through all the work to build the string before it can learn that it has failed.

16.19 Pond Sizes: You have an integer matrix representing a plot of land, where the value at that location represents the height above sea level. A value of zero indicates water. A pond is a region of water connected vertically, horizontally, or diagonally. The size of the pond is the total number of connected water cells. Write a method to compute the sizes of all ponds in the matrix.

EXAMPLE

Input:

```
0 2 1 0
0 1 0 1
1 1 0 1
0 1 0 1
```

Output: 2, 4, 1 (in any order)

pg 184

SOLUTION

The first thing we can try is just walking through the array. It's easy enough to find water: when it's a zero, that's water.

Given a water cell, how can we compute the amount of water nearby? If the cell is not adjacent to any zero cells, then the size of this pond is 1. If it is, then we need to add in the adjacent cells, plus any water cells adjacent to those cells. We need to, of course, be careful to not recount any cells. We can do this with a modified breadth-first or depth-first search. Once we visit a cell, we permanently mark it as visited.

For each cell, we need to check eight adjacent cells. We could do this by writing in lines to check up, down, left, right, and each of the four diagonal cells. It's even easier, though, to do this with a loop.

```
ArrayList<Integer> computePondSizes(int[][] land) {
    ArrayList<Integer> pondSizes = new ArrayList<Integer>();
    for (int r = 0; r < land.length; r++) {
        for (int c = 0; c < land[r].length; c++) {
            if (land[r][c] == 0) { // Optional. Would return anyway.
                int size = computeSize(land, r, c);
                pondSizes.add(size);
            }
        }
    }
    return pondSizes;
}

int computeSize(int[][] land, int row, int col) {
    /* If out of bounds or already visited. */
    if (row < 0 || col < 0 || row >= land.length || col >= land[row].length ||
        land[row][col] != 0) { // visited or not water
        return 0;
    }
```

```
    int size = 1;
    land[row][col] = -1; // Mark visited
    for (int dr = -1; dr <= 1; dr++) {
       for (int dc = -1; dc <= 1; dc++) {
          size += computeSize(land, row + dr, col + dc);
       }
    }
    return size;
 }
```

In this case, we marked a cell as visited by setting its value to -1. This allows us to check, in one line (`land[row][col] != 0`), if the value is valid dry land or visited. In either case, the value will be zero.

You might also notice that the for loop iterates through nine cells, not eight. It includes the current cell. We could add a line in there to not recurse if `dr == 0` and `dc == 0`. This really doesn't save us much. We'll execute this if-statement in eight cells unnecessarily, just to avoid one recursive call. The recursive call returns immediately since the cell is marked as visited.

If you don't like modifying the input matrix, you can create a secondary `visited` matrix.

```
ArrayList<Integer> computePondSizes(int[][] land) {
   boolean[][] visited = new boolean[land.length][land[0].length];
   ArrayList<Integer> pondSizes = new ArrayList<Integer>();
   for (int r = 0; r < land.length; r++) {
      for (int c = 0; c < land[r].length; c++) {
         int size = computeSize(land, visited, r, c);
         if (size > 0) {
            pondSizes.add(size);
         }
      }
   }
   return pondSizes;
}

int computeSize(int[][] land, boolean[][] visited, int row, int col) {
   /* If out of bounds or already visited. */
   if (row < 0 || col < 0 || row >= land.length || col >= land[row].length ||
       visited[row][col] || land[row][col] != 0) {
      return 0;
   }
   int size = 1;
   visited[row][col] = true;
   for (int dr = -1; dr <= 1; dr++) {
      for (int dc = -1; dc <= 1; dc++) {
         size += computeSize(land, visited, row + dr, col + dc);
      }
   }
   return size;
}
```

Both implementations are $O(WH)$, where W is the width of the matrix and H is the height.

> Note: Many people say "$O(N)$" or "$O(N^2)$", as though N has some inherent meaning. It doesn't. Suppose this were a square matrix. You could describe the runtime as $O(N)$ or $O(N^2)$. Both are correct, depending on what you mean by N. The runtime is $O(N^2)$, where N is the length of one side. Or, if N is the number of cells, it is $O(N)$. Be careful by what you mean by N. In fact, it might be safer to just not use N at all when there's any ambiguity as to what it could mean.

Some people will miscompute the runtime to be O(N^4), reasoning that the `computeSize` method could take as long as O(N^2) time and you might call it as much as O(N^2) times (and apparently assuming an NxN matrix, too). While those are both basically correct statements, you can't just multiply them together. That's because as a single call to `computeSize` gets more expensive, the number of times it is called goes down.

For example, suppose the very first call to `computeSize` goes through the entire matrix. That might take O(N^2) time, but then we never call `computeSize` again.

Another way to compute this is to think about how many times each cell is "touched" by either call. Each cell will be touched once by the `computePondSizes` function. Additionally, a cell might be touched once by each of its adjacent cells. This is still a constant number of touches per cell. Therefore, the overall runtime is O(N^2) on an NxN matrix or, more generally, O(WH).

16.20 **T9:** On old cell phones, users typed on a numeric keypad and the phone would provide a list of words that matched these numbers. Each digit mapped to a set of 0 - 4 letters. Implement an algorithm to return a list of matching words, given a sequence of digits. You are provided a list of valid words (provided in whatever data structure you'd like). The mapping is shown in the diagram below:

1	**2** abc	**3** def
4 ghi	**5** jkl	**6** mno
7 pqrs	**8** tuv	**9** wxyz
	0	

EXAMPLE

Input: `8733`

Output: `tree, used`

pg 184

SOLUTION

We could approach this in a couple of ways. Let's start with a brute force algorithm.

Brute Force

Imagine how you would solve the problem if you had to do it by hand. You'd probably try every possible value for each digit with all other possible values.

This is exactly what we do algorithmically. We take the first digit and run through all the characters that map to that digit. For each character, we add it to a `prefix` variable and recurse, passing the prefix downward. Once we run out of characters, we print `prefix` (which now contains the full word) if the string is a valid word.

We will assume the list of words is passed in as a `HashSet`. A `HashSet` operates similarly to a hash table, but rather than offering key->value lookups, it can tell us if a word is contained in the set in O(1) time.

```
ArrayList<String> getValidT9Words(String number, HashSet<String> wordList) {
  ArrayList<String> results = new ArrayList<String>();
  getValidWords(number, 0, "", wordList, results);
  return results;
}

```

```
void getValidWords(String number, int index, String prefix,
                   HashSet<String> wordSet, ArrayList<String> results) {
  /* If it’s a complete word, print it. */
  if (index == number.length() && wordSet.contains(prefix)) {
    results.add(prefix);
    return;
  }

  /* Get characters that match this digit. */
  char digit = number.charAt(index);
  char[] letters = getT9Chars(digit);

  /* Go through all remaining options. */
  if (letters != null) {
    for (char letter : letters) {
      getValidWords(number, index + 1, prefix + letter, wordSet, results);
    }
  }
}

/* Return array of characters that map to this digit. */
char[] getT9Chars(char digit) {
  if (!Character.isDigit(digit)) {
    return null;
  }
  int dig = Character.getNumericValue(digit) - Character.getNumericValue(‘0’);
  return t9Letters[dig];
}

/* Mapping of digits to letters. */
char[][] t9Letters = {null, null, {‘a’, ‘b’, ‘c’}, {‘d’, ‘e’, ‘f’},
  {‘g’, ‘h’, ‘i’}, {‘j’, ‘k’, ‘l’}, {‘m’, ‘n’, ‘o’}, {‘p’, ‘q’, ‘r’, ‘s’},
  {‘t’, ‘u’, ‘v’},  {‘w’, ‘x’, ‘y’, ‘z’}
};
```

This algorithm runs in $O(4^N)$ time, where N is the length of the string. This is because we recursively branch four times for each call to `getValidWords`, and we recurse until a call stack depth of N.

This is very, very slow on large strings.

Optimized

Let's return to thinking about how you would do this, if you were doing it by hand. Imagine the example of `33835676368` (which corresponds to `development`). If you were doing this by hand, I bet you'd skip over solutions that start with `fftf` [3383], as no valid words start with those characters.

Ideally, we'd like our program to make the same sort of optimization: stop recursing down paths which will obviously fail. Specifically, if there are no words in the dictionary that start with `prefix`, stop recursing.

The `Trie` data structure (see "Tries (Prefix Trees)" on page 105) can do this for us. Whenever we reach a string which is not a valid prefix, we exit.

```
ArrayList<String> getValidT9Words(String number, Trie trie) {
  ArrayList<String> results = new ArrayList<String>();
  getValidWords(number, 0, “”, trie.getRoot(), results);
  return results;
}

```

```
void getValidWords(String number, int index, String prefix, TrieNode trieNode,
                   ArrayList<String> results) {
  /* If it's a complete word, print it. */
  if (index == number.length()) {
    if (trieNode.terminates()) { // Is complete word
      results.add(prefix);
    }
    return;
  }

  /* Get characters that match this digit */
  char digit = number.charAt(index);
  char[] letters = getT9Chars(digit);

  /* Go through all remaining options. */
  if (letters != null) {
    for (char letter : letters) {
      TrieNode child = trieNode.getChild(letter);
      /* If there are words that start with prefix + letter,
       * then continue recursing. */
      if (child != null) {
        getValidWords(number, index + 1, prefix + letter, child, results);
      }
    }
  }
}
```

It's difficult to describe the runtime of this algorithm since it depends on what the language looks like. However, this "short-circuiting" will make it run much, much faster in practice.

Most Optimal

Believe or not, we can actually make it run even faster. We just need to do a little bit of preprocessing. That's not a big deal though. We were doing that to build the trie anyway.

This problem is asking us to list all the words represented by a particular number in T9. Instead of trying to do this "on the fly" (and going through a lot of possibilities, many of which won't actually work), we can just do this in advance.

Our algorithm now has a few steps:

Pre-Computation:

1. Create a hash table that maps from a sequence of digits to a list of strings.
2. Go through each word in the dictionary and convert it to its T9 representation (e.g., `APPLE -> 27753`). Store each of these in the above hash table. For example, 8733 would map to `{used, tree}`.

Word Lookup:

1. Just look up the entry in the hash table and return the list.

That's it!

```
/* WORD LOOKUP */
ArrayList<String> getValidT9Words(String numbers,
                                  HashMapList<String, String> dictionary) {
  return dictionary.get(numbers);
}

```

```
/* PRECOMPUTATION */

/* Create a hash table that maps from a number to all words that have this
 * numerical representation. */
HashMapList<String, String> initializeDictionary(String[] words) {
  /* Create a  hash table that maps from a letter to the digit */
  HashMap<Character, Character> letterToNumberMap = createLetterToNumberMap();

  /* Create word -> number map. */
  HashMapList<String, String> wordsToNumbers = new HashMapList<String, String>();
  for (String word : words) {
    String numbers = convertToT9(word, letterToNumberMap);
    wordsToNumbers.put(numbers, word);
  }
  return wordsToNumbers;
}

/* Convert mapping of number->letters into letter->number. */
HashMap<Character, Character> createLetterToNumberMap() {
  HashMap<Character, Character> letterToNumberMap =
    new HashMap<Character, Character>();
  for (int i = 0; i < t9Letters.length; i++) {
    char[] letters = t9Letters[i];
    if (letters != null) {
      for (char letter : letters) {
        char c = Character.forDigit(i, 10);
        letterToNumberMap.put(letter, c);
      }
    }
  }
  return letterToNumberMap;
}

/* Convert from a string to its T9 representation. */
String convertToT9(String word, HashMap<Character, Character> letterToNumberMap) {
  StringBuilder sb = new StringBuilder();
  for (char c : word.toCharArray()) {
    if (letterToNumberMap.containsKey(c)) {
      char digit = letterToNumberMap.get(c);
      sb.append(digit);
    }
  }
  return sb.toString();
}

char[][] t9Letters = /* Same as before */

/* HashMapList<String, Integer> is a HashMap that maps from Strings to
 * ArrayList<Integer>. See appendix for implementation. */
```

Getting the words that map to this number will run in $O(N)$ time, where N is the number of digits. The $O(N)$ comes in during the hash table look up (we need to convert the number to a hash table). If you know the words are never longer than a certain max size, then you could also describe the runtime as $O(1)$.

Note that it's easy to think, "Oh, linear—that's not that fast." But it depends what it's linear *on*. Linear on the length of the word is extremely fast. Linear on the length of the dictionary is not so fast.

16.21 Sum Swap: Given two arrays of integers, find a pair of values (one value from each array) that you can swap to give the two arrays the same sum.

EXAMPLE

Input: `{4, 1, 2, 1, 1, 2}` and `{3, 6, 3, 3}`

Output: `{1, 3}`

pg 184

SOLUTION

We should start by trying to understand what exactly we're looking for.

We have two arrays and their sums. Although we likely aren't given their sums upfront, we can just act like we are for now. After all, computing the sum is an O(N) operation and we know we can't beat O(N) anyway. Computing the sum, therefore, won't impact the runtime.

When we move a (positive) value a from array A to array B, then the sum of A drops by a and the sum of B increases by a.

We are looking for two values, a and b, such that:

```
sumA - a + b = sumB - b + a
```

Doing some quick math:

```
2a - 2b = sumA - sumB
a - b = (sumA - sumB) / 2
```

Therefore, we're looking for two values that have a specific target difference: `(sumA - sumB) / 2`.

Observe that because that the target must be an integer (after all, you can't swap two integers to get a non-integer difference), we can conclude that the difference between the sums must be even to have a valid pair.

Brute Force

A brute force algorithm is simple enough. We just iterate through the arrays and check all pairs of values.

We can either do this the "naive" way (compare the new sums) or by looking for a pair with that difference.

Naive approach:

```
int[] findSwapValues(int[] array1, int[] array2) {
  int sum1 = sum(array1);
  int sum2 = sum(array2);

  for (int one : array1) {
    for (int two : array2) {
      int newSum1 = sum1 - one + two;
      int newSum2 = sum2 - two + one;
      if (newSum1 == newSum2) {
        int[] values = {one, two};
        return values;
      }
    }
  }

  return null;
}
```

Target approach:

```
int[] findSwapValues(int[] array1, int[] array2) {
```

```
    Integer target = getTarget(array1, array2);
    if (target == null) return null;

    for (int one : array1) {
       for (int two : array2) {
          if (one - two == target) {
             int[] values = {one, two};
             return values;
          }
       }
    }

    return null;
 }

 Integer getTarget(int[] array1, int[] array2) {
    int sum1 = sum(array1);
    int sum2 = sum(array2);

    if ((sum1 - sum2) % 2 != 0) return null;
    return (sum1 - sum2) / 2;
 }
```

We've used an `Integer` (a boxed data type) as the return value for `getTarget`. This allows us to distinguish an "error" case.

This algorithm takes O(AB) time.

Optimal Solution

This problem reduces to finding a pair of values that have a particular difference. With that in mind, let's revisit what the brute force does.

In the brute force, we're looping through A and then, for each element, looking for an element in B which gives us the "right" difference. If the value in A is 5 and the target is 3, then we must be looking for the value 2. That's the only value that could fulfill the goal.

That is, rather than writing `one - two == target`, we could have written `two == one - target`. How can we more quickly find an element in B that equals `one - target`?

We can do this very quickly with a hash table. We just throw all the elements in B into a hash table. Then, iterate through A and look for the appropriate element in B.

```
int[] findSwapValues(int[] array1, int[] array2) {
   Integer target = getTarget(array1, array2);
   if (target == null) return null;
   return findDifference(array1, array2, target);
}

/* Find a pair of values with a specific difference. */
int[] findDifference(int[] array1, int[] array2, int target) {
   HashSet<Integer> contents2 = getContents(array2);
   for (int one : array1) {
      int two = one - target;
      if (contents2.contains(two)) {
         int[] values = {one, two};
         return values;
      }
```

```
    }

    return null;
}

/* Put contents of array into hash set. */
HashSet<Integer> getContents(int[] array) {
    HashSet<Integer> set = new HashSet<Integer>();
    for (int a : array) {
        set.add(a);
    }
    return set;
}
```

This solution will take O(A+B) time. This is the Best Conceivable Runtime (BCR), since we have to at least touch every element in the two arrays.

Alternate Solution

If the arrays are sorted, we can iterate through them to find an appropriate pair. This will require less space.

```
int[] findSwapValues(int[] array1, int[] array2) {
    Integer target = getTarget(array1, array2);
    if (target == null) return null;
    return findDifference(array1, array2, target);
}

int[] findDifference(int[] array1, int[] array2, int target) {
    int a = 0;
    int b = 0;

    while (a < array1.length && b < array2.length) {
        int difference = array1[a] - array2[b];
        /* Compare difference to target. If difference is too small, then make it
         * bigger by moving a to a bigger value. If it is too big, then make it
         * smaller by moving b to a bigger value. If it's just right, return this
         * pair. */
        if (difference == target) {
            int[] values = {array1[a], array2[b]};
            return values;
        } else if (difference < target) {
            a++;
        } else {
            b++;
        }
    }

    return null;
}
```

This algorithm takes O(A + B) time but requires the arrays to be sorted. If the arrays aren't sorted, we can still apply this algorithm but we'd have to sort the arrays first. The overall runtime would be O(A log A + B log B).

16.22 Langton's Ant: An ant is sitting on an infinite grid of white and black squares. It initially faces right. At each step, it does the following:

(1) At a white square, flip the color of the square, turn 90 degrees right (clockwise), and move forward one unit.

(2) At a black square, flip the color of the square, turn 90 degrees left (counter-clockwise), and move forward one unit.

Write a program to simulate the first K moves that the ant makes and print the final board as a grid. Note that you are not provided with the data structure to represent the grid. This is something you must design yourself. The only input to your method is K. You should print the final grid and return nothing. The method signature might be something like `void printKMoves(int K)`.

pg 185

SOLUTION

At first glance, this problem seems very straightforward: create a grid, remember the ant's position and orientation, flip the cells, turn, and move. The interesting part comes in how to handle an infinite grid.

Solution #1: Fixed Array

Technically, since we're only running the first K moves, we do have a max size for the grid. The ant cannot move more than K moves in either direction. If we create a grid that has width 2K and height 2K (and place the ant at the center), we know it will be big enough.

The problem with this is that it's not very extensible. If you run K moves and then want to run another K moves, you might be out of luck.

Additionally, this solution wastes a good amount of space. The max might be K moves in a particular dimension, but the ant is probably going in circles a bit. You probably won't need all this space.

Solution #2: Resizable Array

One thought is to use a resizable array, such as Java's `ArrayList` class. This allows us to grow an array as necessary, while still offering `O(1)` amortized insertion.

The problem is that our grid needs to grow in two dimensions, but the `ArrayList` is only a single array. Additionally, we need to grow "backward" into negative values. The `ArrayList` class doesn't support this.

However, we take a similar approach by building our own resizable grid. Each time the ant hits an edge, we double the size of the grid in that dimension.

What about the negative expansions? While conceptually we can talk about something being at negative positions, we cannot actually access array indices with negative values.

One way we can handle this is to create "fake indices." Let us treat the ant as being at coordinates `(-3, -10)`, but track some sort of offset or delta to translate these coordinates into array indices.

This is actually unnecessary, though. The ant's location does not need to be publicly exposed or consistent (unless, of course, indicated by the interviewer). When the ant travels into negative coordinates, we can double the size of the array and just move the ant and all cells into the positive coordinates. Essentially, we are relabeling all the indices.

This relabeling will not impact the big O time since we have to create a new matrix anyway.

```
public class Grid {
  private boolean[][] grid;
```

```java
    private Ant ant = new Ant();

    public Grid() {
      grid = new boolean[1][1];
    }

    /* Copy old values into new array, with an offset/shift applied to the row and
     * columns. */
    private void copyWithShift(boolean[][] oldGrid, boolean[][] newGrid,
                               int shiftRow, int shiftColumn) {
      for (int r = 0; r < oldGrid.length; r++) {
        for (int c = 0; c < oldGrid[0].length; c++) {
          newGrid[r + shiftRow][c + shiftColumn] = oldGrid[r][c];
        }
      }
    }

    /* Ensure that the given position will fit on the array. If necessary, double
     * the size of the matrix, copy the old values over, and adjust the ant's
     * position so that it's in a positive range. */
    private void ensureFit(Position position) {
      int shiftRow = 0;
      int shiftColumn = 0;

      /* Calculate new number of rows. */
      int numRows = grid.length;
      if (position.row < 0) {
        shiftRow = numRows;
        numRows *= 2;
      } else if (position.row >= numRows) {
        numRows *= 2;
      }

      /* Calculate new number of columns. */
      int numColumns = grid[0].length;
      if (position.column < 0) {
        shiftColumn = numColumns;
        numColumns *= 2;
      } else if (position.column >= numColumns) {
        numColumns *= 2;
      }

      /* Grow array, if necessary. Shift ant's position too. */
      if (numRows != grid.length || numColumns != grid[0].length) {
        boolean[][] newGrid = new boolean[numRows][numColumns];
        copyWithShift(grid, newGrid, shiftRow, shiftColumn);
        ant.adjustPosition(shiftRow, shiftColumn);
        grid = newGrid;
      }
    }

    /* Flip color of cells. */
    private void flip(Position position) {
      int row = position.row;
      int column = position.column;
      grid[row][column] = grid[row][column] ? false : true;
```

```
    }

    /* Move ant. */
    public void move() {
       ant.turn(grid[ant.position.row][ant.position.column]);
       flip(ant.position);
       ant.move();
       ensureFit(ant.position); // grow
    }

    /* Print board. */
    public String toString() {
       StringBuilder sb = new StringBuilder();
       for (int r = 0; r < grid.length; r++) {
          for (int c = 0; c < grid[0].length; c++) {
             if (r == ant.position.row && c == ant.position.column) {
                sb.append(ant.orientation);
             } else if (grid[r][c]) {
                sb.append("X");
             } else {
                sb.append("_");
             }
          }
          sb.append("\n");
       }
       sb.append("Ant: " + ant.orientation + ". \n");
       return sb.toString();
    }
 }
```

We pulled the Ant code into a separate class. The nice thing about this is that if we need to have multiple ants for some reason, we can easily extend the code to support this.

```
 public class Ant {
    public Position position = new Position(0, 0);
    public Orientation orientation = Orientation.right;

    public void turn(boolean clockwise) {
       orientation = orientation.getTurn(clockwise);
    }

    public void move() {
       if (orientation == Orientation.left) {
          position.column--;
       } else if (orientation == Orientation.right) {
          position.column++;
       } else if (orientation == Orientation.up) {
          position.row--;
       } else if (orientation == Orientation.down) {
          position.row++;
       }
    }

    public void adjustPosition(int shiftRow, int shiftColumn) {
       position.row += shiftRow;
       position.column += shiftColumn;
    }
 }
```

`Orientation` is also its own enum, with a few useful functions.

```
public enum Orientation {
  left, up, right, down;

  public Orientation getTurn(boolean clockwise) {
    if (this == left) {
      return clockwise ? up : down;
    } else if (this == up) {
      return clockwise ? right : left;
    } else if (this == right) {
      return clockwise ? down : up;
    } else { // down
      return clockwise ? left : right;
    }
  }

  @Override
  public String toString() {
    if (this == left) {
      return "\u2190";
    } else if (this == up) {
      return "\u2191";
    } else if (this == right) {
      return "\u2192";
    } else { // down
      return "\u2193";
    }
  }
}
```

We've also put `Position` into its own simple class. We could just as easily track the row and column separately.

```
public class Position {
  public int row;
  public int column;

  public Position(int row, int column) {
    this.row = row;
    this.column = column;
  }
}
```

This works, but it's actually more complicated than is necessary.

Solution #3: HashSet

Although it may seem "obvious" that we would use a matrix to represent a grid, it's actually easier not to do that. All we actually need is a list of the white squares (as well as the ant's location and orientation).

We can do this by using a HashSet of the white squares. If a position is in the hash set, then the square is white. Otherwise, it is black.

The one tricky bit is how to print the board. Where do we start printing? Where do we end?

Since we will need to print a grid, we can track what should be top-left and bottom-right corner of the grid. Each time the ant moves, we compare the ant's position to the most top-left position and most bottom-right position, updating them if necessary.

```
public class Board {
  private HashSet<Position> whites = new HashSet<Position>();
  private Ant ant = new Ant();
  private Position topLeftCorner = new Position(0, 0);
  private Position bottomRightCorner = new Position(0, 0);

  public Board() { }

  /* Move ant. */
  public void move() {
    ant.turn(isWhite(ant.position)); // Turn
    flip(ant.position); // flip
    ant.move(); // move
    ensureFit(ant.position);
  }

  /* Flip color of cells. */
  private void flip(Position position) {
    if (whites.contains(position)) {
      whites.remove(position);
    } else {
      whites.add(position.clone());
    }
  }

  /* Grow grid by tracking the most top-left and bottom-right positions.*/
  private void ensureFit(Position position) {
    int row = position.row;
    int column = position.column;

    topLeftCorner.row = Math.min(topLeftCorner.row, row);
    topLeftCorner.column = Math.min(topLeftCorner.column, column);

    bottomRightCorner.row = Math.max(bottomRightCorner.row, row);
    bottomRightCorner.column = Math.max(bottomRightCorner.column, column);
  }

  /* Check if cell is white. */
  public boolean isWhite(Position p) {
    return whites.contains(p);
  }

  /* Check if cell is white. */
  public boolean isWhite(int row, int column) {
    return whites.contains(new Position(row, column));
  }

  /* Print board. */
  public String toString() {
    StringBuilder sb = new StringBuilder();
    int rowMin = topLeftCorner.row;
    int rowMax = bottomRightCorner.row;
    int colMin = topLeftCorner.column;
    int colMax = bottomRightCorner.column;
    for (int r = rowMin; r <= rowMax; r++) {
      for (int c = colMin; c <= colMax; c++) {
```

```
            if (r == ant.position.row && c == ant.position.column) {
               sb.append(ant.orientation);
            } else if (isWhite(r, c)) {
               sb.append("X");
            } else {
               sb.append("_");
            }
         }
         sb.append("\n");
      }
      sb.append("Ant: " + ant.orientation + ". \n");
      return sb.toString();
   }
```

The implementation of `Ant` and `Orientation` is the same.

The implementation of `Position` gets updated slightly, in order to support the `HashSet` functionality. The position will be the key, so we need to implement a `hashCode()` function.

```
public class Position {
   public int row;
   public int column;

   public Position(int row, int column) {
      this.row = row;
      this.column = column;
   }

   @Override
   public boolean equals(Object o) {
      if (o instanceof Position) {
         Position p = (Position) o;
         return p.row == row && p.column == column;
      }
      return false;
   }

   @Override
   public int hashCode() {
      /* There are many options for hash functions. This is one. */
      return (row * 31) ^ column;
   }

   public Position clone() {
      return new Position(row, column);
   }
}
```

The nice thing about this implementation is that if we do need to access a particular cell elsewhere, we have consistent row and column labeling.

16.23 Rand7 from Rand5: Implement a method `rand7()` given `rand5()`. That is, given a method that generates a random number between 0 and 4 (inclusive), write a method that generates a random number between 0 and 6 (inclusive).

pg 186

SOLUTION

To implement this function correctly, we must have each of the values between 0 and 6 returned with 1/7th probability.

First Attempt (Fixed Number of Calls)

As a first attempt, we might try generating all numbers between 0 and 9, and then mod the resulting value by 7. Our code for it might look something like this:

```
int rand7() {
   int v = rand5() + rand5();
   return v % 7;
}
```

Unfortunately, the above code will not generate the values with equal probability. We can see this by looking at the results of each call to `rand5()` an d the return result of the `rand7()` function.

1st Call	2nd Call	Result
0	0	0
0	1	1
0	2	2
0	3	3
0	4	4
1	0	1
1	1	2
1	2	3
1	3	4
1	4	5
2	0	2
2	1	3
2	2	4
2	3	5
2	4	6
3	0	3
3	1	4
3	2	5
3	3	6
3	4	0
4	0	4
4	1	5
4	2	6
4	3	0
4	4	1

Each individual row has a 1 in 25 chance of occurring, since there are two calls to `rand5()` and each distributes its results with $\frac{1}{5}$th probability. If you count up the number of times each number occurs, you'll note that this `rand7()` function will return 4 with $\frac{5}{25}$th probability but return 0 with just $\frac{3}{25}$th probability. This means that our function has failed; the results do not have probability $\frac{1}{7}$th.

Now, imagine we modify our function to add an if-statement, to change the constant multiplier, or to insert a new call to `rand5()`. We will still wind up with a similar looking table, and the probability of getting any one of those rows will be $\frac{1}{5^k}$, where k is the number of calls to `rand5()` in that row. Different rows may have different number of calls.

The probability of winding up with the result of the `rand7()` function being, say, 6 would be the sum of the probabilities of all rows that result in 6. That is:

$$\texttt{P(rand7() = 6)} = \frac{1}{5^i} + \frac{1}{5^j} + \ldots + \frac{1}{5^m}$$

We know that, in order for our function to be correct, this probability must equal $\frac{1}{7}$. This is impossible though. Because 5 and 7 are relatively prime, no series of reciprocal powers of 5 will result in $\frac{1}{7}$.

Does this mean the problem is impossible? Not exactly. Strictly speaking, it means that, as long as we can list out the combinations of `rand5()` results that will result in a particular value of `rand7()`, the function will not give well distributed results.

We can still solve this problem. We just have to use a while loop, and realize that there's no telling just how many turns will be required to return a result.

Second Attempt (Nondeterministic Number of Calls)

As soon as we've allowed for a while loop, our work gets much easier. We just need to generate a range of values where each value is equally likely (and where the range has at least seven elements). If we can do this, then we can discard the elements greater than the previous multiple of 7, and mod the rest of them by 7. This will get us a value within the range of 0 to 6, with each value being equally likely.

In the below code, we generate the range 0 through 24 by doing `5 * rand5() + rand5()`. Then, we discard the values between 21 and 24, since they would otherwise make `rand7()` unfairly weighted towards 0 through 3. Finally, we mod by 7 to give us the values in the range 0 to 6 with equal probability.

Note that because we discard values in this approach, we have no guarantee on the number of `rand5()` calls it may take to return a value. This is what is meant by a *nondeterministic* number of calls.

```
int rand7() {
  while (true) {
    int num = 5 * rand5() + rand5();
    if (num < 21) {
      return num % 7;
    }
  }
}
```

Observe that doing `5 * rand5() + rand5()` gives us exactly one way of getting each number in its range (0 to 24). This ensures that each value is equally probable.

Could we instead do `2 * rand5() + rand5()`? No, because the values wouldn't be equally distributed. For example, there would be three ways of getting a 6 (`6 = 2 * 1 + 4`, `6 = 2 * 2 + 2`, and `6 = 2 * 3 + 0`) but only one way of getting a 0 (`0=2*0+0`). The values in the range are not equally probable.

There *is* a way that we can use `2 * rand5()` and still get an identically distributed range, but it's much more complicated. See below.

```
int rand7() {
  while (true) {
    int r1 = 2 * rand5(); /* evens between 0 and 9 */
    int r2 = rand5(); /* used later to generate a 0 or 1 */
    if (r2 != 4) { /* r2 has extra even num-discard the extra */
      int rand1 = r2 % 2; /* Generate 0 or 1 */
      int num = r1 + rand1; /* will be in the range 0 to 9 */
      if (num < 7) {
        return num;
      }
    }
  }
}
```

In fact, there is an infinite number of ranges we can use. The key is to make sure that the range is big enough and that all values are equally likely.

16.24 Pairs with Sum: Design an algorithm to find all pairs of integers within an array which sum to a specified value.

pg 185

SOLUTION

Let's start with a definition. If we're trying to find a pair of numbers that sums to z, the *complement* of x will be z - x (that is, the number that can be added to x to make z). For example, if we're trying to find a pair of numbers that sums to 12, the complement of -5 would be 17.

Brute Force

A brute force solution is to just iterate through all pairs and print the pair if its sum matches the target sum.

```
ArrayList<Pair> printPairSums(int[] array, int sum) {
   ArrayList<Pair> result = new ArrayList<Pair>();
   for (int i = 0 ; i < array.length; i++) {
      for (int j = i + 1; j < array.length; j++) {
         if (array[i] + array[j] == sum) {
            result.add(new Pair(array[i], array[j]));
         }
      }
   }
   return result;
}
```

If there are duplicates in the array (e.g., `{5, 6, 5}`), it might print the same sum twice. You should discuss this with your interviewer.

Optimized Solution

We can optimize this with a hash map, where the value in the hash map reflects the number of "unpaired" instances of a key. We walk through the array. At each element x, check how many unpaired instances of x's complement preceded it in the array. If the count is at least one, then there is an unpaired instance of x's complement. We add this pair and decrement x's complement to signify that this element has been paired. If the count is zero, then increment the value of x in the hash table to signify that x is unpaired.

```
ArrayList<Pair> printPairSums(int[] array, int sum) {
   ArrayList<Pair> result = new ArrayList<Pair>();
   HashMap<Integer, Integer> unpairedCount = new HashMap<Integer, Integer>();
   for (int x : array) {
      int complement = sum - x;
      if (unpairedCount.getOrDefault(complement, 0) > 0) {
         result.add(new Pair(x, complement));
         adjustCounterBy(unpairedCount, complement, -1); // decrement complement
      } else {
         adjustCounterBy(unpairedCount, x, 1); // increment count
      }
   }
   return result;
}

```

```
void adjustCounterBy(HashMap<Integer, Integer> counter, int key, int delta) {
  counter.put(key, counter.getOrDefault(key, 0) + delta);
}
```

This solution will print duplicate pairs, but will not reuse the same instance of an element. It will take O(N) time and O(N) space.

Alternate Solution

Alternatively, we can sort the array and then find the pairs in a single pass. Consider this array:

`{-2, -1, 0, 3, 5, 6, 7, 9, 13, 14}.`

Let `first` point to the head of the array and `last` point to the end of the array. To find the complement of `first`, we just move `last` backwards until we find it. If `first + last < sum`, then there is no complement for `first`. We can therefore move `first` forward. We stop when `first` is greater than `last`.

Why must this find all complements for `first`? Because the array is sorted and we're trying progressively smaller numbers. When the sum of `first` and `last` is less than the sum, we know that trying even smaller numbers (as `last`) won't help us find a complement.

Why must this find all complements for `last`? Because all pairs must be made up of a `first` and a `last`. We've found all complements for `first`, therefore we've found all complements of `last`.

```
void printPairSums(int[] array, int sum) {
  Arrays.sort(array);
  int first = 0;
  int last = array.length - 1;
  while (first < last) {
    int s = array[first] + array[last];
    if (s == sum) {
      System.out.println(array[first] + " " + array[last]);
      first++;
      last--;
    } else {
      if (s < sum) first++;
      else last--;
    }
  }
}
```

This algorithm takes O(N log N) time to sort and O(N) time to find the pairs.

Note that since the array is presumably unsorted, it would be equally fast in terms of big O to just do a binary search at each element for its complement. This would give us a two-step algorithm, where each step is O(N log N).

16.25 LRU Cache: Design and build a "least recently used" cache, which evicts the least recently used item. The cache should map from keys to values (allowing you to insert and retrieve a value associated with a particular key) and be initialized with a max size. When it is full, it should evict the least recently used item. You can assume the keys are integers and the values are strings.

pg 185

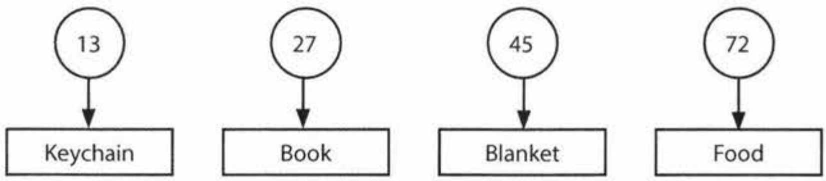
SOLUTION

We should start off by defining the scope of the problem. What exactly do we need to achieve?

- **Inserting Key, Value Pair:** We need to be able to insert a (key, value) pair.

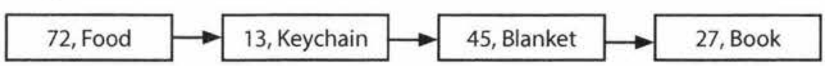
- **Retrieving Value by Key:** We need to be able to retrieve the value using the key.
- **Finding Least Recently Used:** We need to know the least recently used item (and, likely, the usage ordering of all items).
- **Updating Most Recently Used:** When we retrieve a value by key, we need to update the order to be the most recently used item.
- **Eviction:** The cache should have a max capacity and should remove the least recently used item when it hits capacity.

The (key, value) mapping suggests a hash table. This would make it easy to look up the value associated with a particular key.



Unfortunately, a hash table usually would not offer a quick way to remove the most recently used item. We could mark each item with a timestamp and iterate through the hash table to remove the item with the lowest timestamp, but that can get quite slow (O(N) for insertions).

Instead, we could use a linked list, ordered by the most recently used. This would make it easy to mark an item as the most recently used (just put it in the front of the list) or to remove the least recently used item (remove the end).

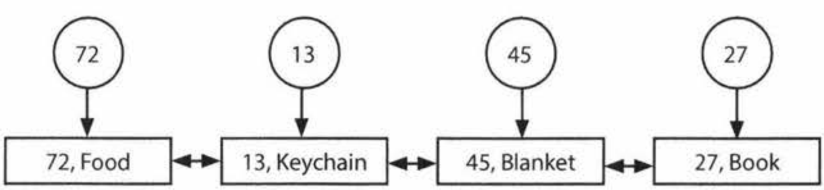


Unfortunately, this does not offer a quick way to look up an item by its key. We could iterate through the linked list and find the item by key. But this could get very slow (O(N) for retrieval).

Each approach does half of the problem (different halves) very well, but neither approach does both parts well.

Can we get the best parts of each? Yes. By using both!

The linked list looks as it did in the earlier example, but now it's a doubly linked list. This allows us to easily remove an element from the middle of the linked list. The hash table now maps to each linked list node rather than the value.



The algorithms now operate as follows:

- **Inserting Key, Value Pair:** Create a linked list node with key, value. Insert into head of linked list. Insert key -> node mapping into hash table.
- **Retrieving Value by Key:** Look up node in hash table and return value. Update most recently used item

(see below).

- **Finding Least Recently Used:** Least recently used item will be found at the end of the linked list.
- **Updating Most Recently Used:** Move node to front of linked list. Hash table does not need to be updated.
- **Eviction:** Remove tail of linked list. Get key from linked list node and remove key from hash table.

The code below implements these classes and algorithms.

```
public class Cache {
  private int maxCacheSize;
  private HashMap<Integer, LinkedListNode> map =
    new HashMap<Integer, LinkedListNode>();
  private LinkedListNode listHead = null;
  public LinkedListNode listTail = null;

  public Cache(int maxSize) {
    maxCacheSize = maxSize;
  }

  /* Get value for key and mark as most recently used. */
  public String getValue(int key) {
    LinkedListNode item = map.get(key);
    if (item == null) return null;

    /* Move to front of list to mark as most recently used. */
    if (item != listHead) {
      removeFromLinkedList(item);
      insertAtFrontOfLinkedList(item);
    }
    return item.value;
  }

  /* Remove node from linked list. */
  private void removeFromLinkedList(LinkedListNode node) {
    if (node == null)  return;

    if (node.prev != null) node.prev.next = node.next;
    if (node.next != null) node.next.prev = node.prev;
    if (node == listTail) listTail = node.prev;
    if (node == listHead) listHead = node.next;
  }

  /* Insert node at front of linked list. */
  private void insertAtFrontOfLinkedList(LinkedListNode node) {
    if (listHead == null) {
      listHead = node;
      listTail = node;
    } else {
      listHead.prev = node;
      node.next = listHead;
      listHead = node;
    }
  }

  /* Remove key/value pair from cache, deleting from hashtable and linked list. */
  public boolean removeKey(int key) {
```

```
    LinkedListNode node = map.get(key);
    removeFromLinkedList(node);
    map.remove(key);
    return true;
  }

  /* Put key, value pair in cache. Removes old value for key if necessary. Inserts
   * pair into linked list and hash table.*/
  public void setKeyValue(int key, String value) {
    /* Remove if already there. */
    removeKey(key);

    /* If full, remove least recently used item from cache. */
    if (map.size() >= maxCacheSize && listTail != null) {
      removeKey(listTail.key);
    }

    /* Insert new node. */
    LinkedListNode node = new LinkedListNode(key, value);
    insertAtFrontOfLinkedList(node);
    map.put(key, node);
  }

  private static class LinkedListNode {
    private LinkedListNode next, prev;
    public int key;
    public String value;
    public LinkedListNode(int k, String v) {
      key = k;
      value = v;
    }
  }
}
```

Note that we've chosen to make `LinkedListNode` an inner class of `Cache`, since no other classes should need access to this class and really should only exist within the scope of `Cache`.

16.26 Calculator: Given an arithmetic equation consisting of positive integers, +, -, * and / (no parentheses), compute the result.

EXAMPLE

Input: `2*3+5/6*3+15`

Output: `23.5`

pg 185

SOLUTION

The first thing we should realize is that the dumb thing—just applying each operator left to right—won't work. Multiplication and division are considered "higher priority" operations, which means that they have to happen before addition.

For example, if you have the simple expression 3+6*2, the multiplication must be performed first, and then the addition. If you just processed the equation left to right, you would end up with the incorrect result, 18, rather than the correct one, 15. You know all of this, of course, but it's worth really spelling out what it means.

Solution #1

We can still process the equation from left to right; we just have to be a little smarter about how we do it. Multiplication and division need to be grouped together such that whenever we see those operations, we perform them immediately on the surrounding terms.

For example, suppose we have this expression:

```
2 - 6 - 7*8/2 + 5
```

It's fine to compute 2 - 6 immediately and store it into a `result` variable. But, when we see `7*(something)`, we know we need to fully process that term before adding it to the result.

We can do this by reading left to right and maintaining two variables.

- The first is `processing`, which maintains the result of the current cluster of terms (both the operator and the value). In the case of addition and subtraction, the cluster will be just the current term. In the case of multiplication and division, it will be the full sequence (until you get to the next addition or subtraction).
- The second is the `result` variable. If the next term is an addition or subtraction (or there is no next term), then `processing` is applied to `result`.

On the above example, we would do the following:

1. Read +2. Apply it to `processing`. Apply processing to `result`. Clear `processing`.
   ```
   processing = {+, 2} --> null
   result = 0          --> 2
   ```
2. Read -6. Apply it to `processing`. Apply processing to `result`. Clear `processing`.
   ```
   processing = {-, 6} --> null
   result = 2          --> -4
   ```
3. Read -7. Apply it to `processing`. Observe next sign is a *. Continue.
   ```
   processing = {-, 7}
   result = -4
   ```
4. Read *8. Apply it to `processing`. Observe next sign is a /. Continue.
   ```
   processing = {-, 56}
   result = -4
   ```
5. Read /2. Apply it to `processing`. Observe next sign is a +, which terminates this multiplication and division cluster. Apply processing to `result`. Clear `processing`.
   ```
   processing = {-, 28}    --> null
   result = -4             --> -32
   ```
6. Read +5. Apply it to `processing`. Apply processing to `result`. Clear `processing`.
   ```
   processing = {+, 5} --> null
   result = -32        --> -27
   ```

The code below implements this algorithm.

```
/* Compute the result of the arithmetic sequence. This works by reading left to
 * right and applying each term to a result. When we see a multiplication or
 * division, we instead apply this sequence to a temporary variable. */
double compute(String sequence) {
  ArrayList<Term> terms = Term.parseTermSequence(sequence);
  if (terms == null) return Integer.MIN_VALUE;

  double result = 0;
  Term processing = null;
  for (int i = 0; i < terms.size(); i++) {
```

```
      Term current = terms.get(i);
      Term next = i + 1 < terms.size() ? terms.get(i + 1) : null;

      /* Apply the current term to "processing". */
      processing = collapseTerm(processing, current);

      /* If next term is + or -, then this cluster is done and we should apply
       * "processing" to "result". */
      if (next == null || next.getOperator() == Operator.ADD
          || next.getOperator() == Operator.SUBTRACT) {
        result = applyOp(result, processing.getOperator(), processing.getNumber());
        processing = null;
      }
    }

    return result;
  }

  /* Collapse two terms together using the operator in secondary and the numbers
   * from each. */
  Term collapseTerm(Term primary, Term secondary) {
    if (primary == null) return secondary;
    if (secondary == null) return primary;

    double value = applyOp(primary.getNumber(), secondary.getOperator(),
                    secondary.getNumber());
    primary.setNumber(value);
    return primary;
  }

  double applyOp(double left, Operator op, double right) {
    if (op == Operator.ADD) return left + right;
    else if (op == Operator.SUBTRACT) return left - right;
    else if (op == Operator.MULTIPLY) return left * right;
    else if (op == Operator.DIVIDE) return left / right;
    else return right;
  }

  public class Term {
    public enum Operator {
      ADD, SUBTRACT, MULTIPLY, DIVIDE, BLANK
    }

    private double value;
    private Operator operator = Operator.BLANK;

    public Term(double v, Operator op) {
      value = v;
      operator = op;
    }

    public double getNumber() { return value; }
    public Operator getOperator() { return operator; }
    public void setNumber(double v) { value = v; }

    /* Parses arithmetic sequence into a list of Terms. For example, 3-5*6 becomes
```

```
     * something like: [{BLANK,3}, {SUBTRACT, 5}, {MULTIPLY, 6}].
     * If improperly formatted, returns null. */
    public static ArrayList<Term> parseTermSequence(String sequence) {
      /* Code can be found in downloadable solutions. */
    }
}
```

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

Solution #2

Alternatively, we can solve this problem using two stacks: one for numbers and one for operators.

```
2 - 6 - 7 * 8 / 2 + 5
```

The processing works as follows:

- Each time we see a number, it gets pushed onto `numberStack`.
- Operators get pushed onto `operatorStack`—as long as the operator has higher priority than the current top of the stack. If `priority(currentOperator) <= priority(operatorStack.top())`, then we "collapse" the top of the stacks:
  - » Collapsing: pop two elements off `numberStack`, pop an operator off `operatorStack`, apply the operator, and push the result onto `numberStack`.
  - » Priority: addition and subtraction have equal priority, which is lower than the priority of multiplication and division (also equal priority).

  This collapsing continues until the above inequality is broken, at which point `currentOperator` is pushed onto `operatorStack`.
- At the very end, we collapse the stack.

Let's see this with an example: `2 - 6 - 7 * 8 / 2 + 5`

	action	numberStack	operatorStack
2	numberStack.push(2)	2	[empty]
-	operatorStack.push(-)	2	-
6	numberStack.push(6)	6, 2	-
-	collapseStacks [2 - 6] operatorStack.push(-)	-4 -4	[empty] -
7	numberStack.push(7)	7, -4	-
*****	operatorStack.push(*)	7, -4	*, -
8	numberStack.push(8)	8, 7, -4	*, -
/	collapseStack [7 * 8] numberStack.push(/)	56, -4 56, -4	- /, -
2	numberStack.push(2)	2, 56, -4	/, -
+	collapseStack [56 / 2] collapseStack [-4 - 28] operatorStack.push(+)	28, -4 -32 -32	- [empty] +
5	numberStack.push(5)	5, -32	+
	collapseStack [-32 + 5]	-27	[empty]
	return -27		

The code below implements this algorithm.

```
public enum Operator {
  ADD, SUBTRACT, MULTIPLY, DIVIDE, BLANK
}

double compute(String sequence) {
  Stack<Double> numberStack = new Stack<Double>();
  Stack<Operator> operatorStack = new Stack<Operator>();

  for (int i = 0; i < sequence.length(); i++) {
    try {
      /* Get number and push. */
      int value = parseNextNumber(sequence, i);
      numberStack.push((double) value);

      /* Move to the operator. */
      i += Integer.toString(value).length();
      if (i >= sequence.length()) {
        break;
      }

      /* Get operator, collapse top as needed, push operator. */
      Operator op = parseNextOperator(sequence, i);
      collapseTop(op, numberStack, operatorStack);
      operatorStack.push(op);
    } catch (NumberFormatException ex) {
      return Integer.MIN_VALUE;
    }
  }

  /* Do final collapse. */
  collapseTop(Operator.BLANK, numberStack, operatorStack);
  if (numberStack.size() == 1 && operatorStack.size() == 0) {
    return numberStack.pop();
  }
  return 0;
}

/* Collapse top until priority(futureTop) > priority(top). Collapsing means to pop
 * the top 2 numbers and apply the operator popped from the top of the operator
 * stack, and then push that onto the numbers stack.*/
void collapseTop(Operator futureTop, Stack<Double> numberStack,
                 Stack<Operator> operatorStack) {
  while (operatorStack.size() >= 1 && numberStack.size() >= 2) {
    if (priorityOfOperator(futureTop) <=
         priorityOfOperator(operatorStack.peek())) {
      double second = numberStack.pop();
      double first = numberStack.pop();
      Operator op = operatorStack.pop();
      double collapsed = applyOp(first, op, second);
      numberStack.push(collapsed);
    } else {
      break;
    }
  }
```

```
}

/* Return priority of operator. Mapped so that:
 *     addition == subtraction < multiplication == division. */
int priorityOfOperator(Operator op) {
  switch (op) {
     case ADD: return 1;
     case SUBTRACT: return 1;
     case MULTIPLY: return 2;
     case DIVIDE: return 2;
     case BLANK: return 0;
  }
  return 0;
}

/* Apply operator: left [op] right. */
double applyOp(double left, Operator op, double right) {
  if (op == Operator.ADD) return left + right;
  else if (op == Operator.SUBTRACT) return left - right;
  else if (op == Operator.MULTIPLY) return left * right;
  else if (op == Operator.DIVIDE) return left / right;
  else return right;
}

/* Return the number that starts at offset. */
int parseNextNumber(String seq, int offset) {
  StringBuilder sb = new StringBuilder();
  while (offset < seq.length() && Character.isDigit(seq.charAt(offset))) {
     sb.append(seq.charAt(offset));
     offset++;
  }
  return Integer.parseInt(sb.toString());
}

/* Return the operator that occurs as offset. */
Operator parseNextOperator(String sequence, int offset) {
  if (offset < sequence.length()) {
     char op = sequence.charAt(offset);
     switch(op) {
        case '+': return Operator.ADD;
        case '-': return Operator.SUBTRACT;
        case '*': return Operator.MULTIPLY;
        case '/': return Operator.DIVIDE;
     }
  }
  return Operator.BLANK;
}
```

This code also takes O(N) time, where N is the length of the string.

This solution involves a lot of annoying string parsing code. Remember that getting all these details out is not that important in an interview. In fact, your interviewer might even let you assume the expression is passed in pre-parsed into some sort of data structure.

Focus on modularizing your code from the beginning and "farming out" tedious or less interesting parts of the code to other functions. You want to focus on getting the core `compute` function working. The rest of the details can wait!