```
        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;
}
```

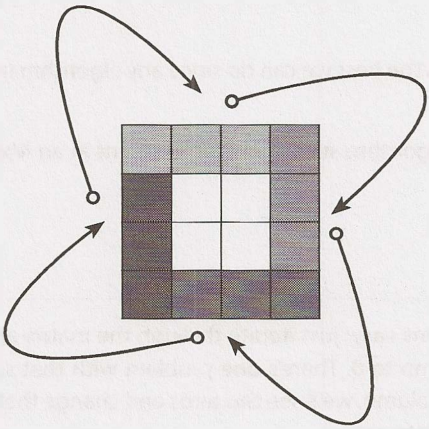
One other benefit of this approach is that we can initialize `StringBuilder` to its necessary capacity up-front. Without this, `StringBuilder` will (behind the scenes) need to double its capacity every time it hits capacity. The capacity could be double what we ultimately need.

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(N^2) time.

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

pg 223

SOLUTION

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

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

Solution #1: If linked list size is known

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

Solution #2: Recursive

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

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

Approach A: Don't Return the Element.

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

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