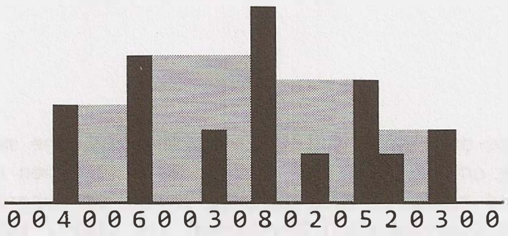
```
    sum += subgraphVolume(histogram, max, end, isLeft);
  }

  return sum;
}

/* Compute volume between start and end. Assumes that tallest bar is at start and
 * second tallest is at end. */
int borderedVolume(HistogramData[] data, int start, int end) {
  if (start >= end) return 0;

  int min = Math.min(data[start].getHeight(), data[end].getHeight());
  int sum = 0;
  for (int i = start + 1; i < end; i++) {
    sum += min - data[i].getHeight();
  }
  return sum;
}

public class HistogramData {
  private int height;
  private int leftMaxIndex = -1;
  private int rightMaxIndex = -1;

  public HistogramData(int v) { height = v; }
  public int getHeight() { return height; }
  public int getLeftMaxIndex() { return leftMaxIndex; }
  public void setLeftMaxIndex(int idx) { leftMaxIndex = idx; };
  public int getRightMaxIndex() { return rightMaxIndex; }
  public void setRightMaxIndex(int idx) { rightMaxIndex = idx; };
}
```

This algorithm takes O(N) time. Since we have to look at every bar, we cannot do better than this.

Solution #3 (Optimized & Simplified)

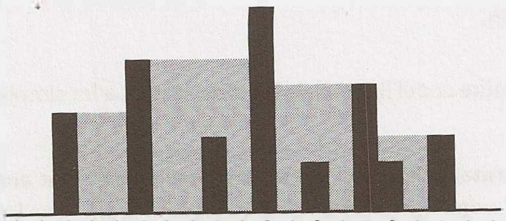
While we can't make the solution faster in terms of big O, we can make it much, much simpler. Let's look at an example again in light of what we've just learned about potential algorithms.



As we've seen, the volume of water in a particular area is determined by the tallest bar to the left and to the right (specifically, by the shorter of the two tallest bars on the left and the tallest bar on the right). For example, water fills in the area between the bar with height 6 and the bar with height 8, up to a height of 6. It's the second tallest, therefore, that determines the height.

The total volume of water is the volume of water above each histogram bar. Can we efficiently compute how much water is above each histogram bar?

Yes. In Solution #2, we were able to precompute the height of the tallest bar on the left and right of each index. The minimums of these will indicate the "water level" at a bar. The difference between the water level and the height of this bar will be the volume of water.



```
   HEIGHT: 0 0 4 0 0 6 0 0 3 0 8 0 2 0 5 2 0 3 0 0
 LEFT MAX: 0 0 4 4 4 6 6 6 6 6 8 8 8 8 8 8 8 8 8 8
RIGHT MAX: 8 8 8 8 8 8 8 8 8 8 8 5 5 5 5 3 3 3 0 0
      MIN: 0 0 4 4 4 6 6 6 6 6 8 5 5 5 5 3 3 3 0 0
    DELTA: 0 0 0 4 4 0 6 6 3 6 0 5 3 5 0 1 3 0 0 0
```

Our algorithm now runs in a few simple steps:

1. Sweep left to right, tracking the max height you've seen and setting left max.
2. Sweep right to left, tracking the max height you've seen and setting right max.
3. Sweep across the histogram, computing the minimum of the left max and right max for each index.
4. Sweep across the histogram, computing the delta between each minimum and the bar. Sum these deltas.

In the actual implementation, we don't need to keep so much data around. Steps 2, 3, and 4 can be merged into the same sweep. First, compute the left maxes in one sweep. Then sweep through in reverse, tracking the right max as you go. At each element, calculate the min of the left and right max and then the delta between that (the "min of maxes") and the bar height. Add this to the sum.

```
/* Go through each bar and compute the volume of water above it.
 * Volume of water at a bar =
 *   height - min(tallest bar on left, tallest bar on right)
 *   [where above equation is positive]
 * Compute the left max in the first sweep, then sweep again to compute the right
 * max, minimum of the bar heights, and the delta. */
int computeHistogramVolume(int[] histo) {
  /* Get left max */
  int[] leftMaxes = new int[histo.length];
  int leftMax = histo[0];
  for (int i = 0; i < histo.length; i++) {
    leftMax = Math.max(leftMax, histo[i]);
    leftMaxes[i] = leftMax;
  }

  int sum = 0;

  /* Get right max */
  int rightMax = histo[histo.length - 1];
  for (int i = histo.length - 1; i >= 0; i--) {
    rightMax = Math.max(rightMax, histo[i]);
    int secondTallest = Math.min(rightMax, leftMaxes[i]);

    /* If there are taller things on the left and right side, then there is water
     * above this bar. Compute the volume and add to the sum. */
      if (secondTallest > histo[i]) {
        sum += secondTallest - histo[i];
      }
    }

    return sum;
  }
```

Yes, this really is the entire code! It is still O(N) time, but it's a lot simpler to read and write.

17.22 Word Transformer: Given two words of equal length that are in a dictionary, write a method to transform one word into another word by changing only one letter at a time. The new word you get in each step must be in the dictionary.

EXAMPLE

Input: DAMP, LIKE

Output: DAMP -> LAMP -> LIMP -> LIME -> LIKE

pg 189

SOLUTION

Let's start with a naive solution and then work our way to a more optimal solution.

Brute Force

One way of solving this problem is to just transform the words in every possible way (of course checking at each step to ensure each is a valid word), and then see if we can reach the final word.

So, for example, the word bold would be transformed into:

- aold, bold, ..., zold
- bald, bbld, ..., bzld
- boad, bobd, ..., bozd
- bola, bolb, ..., bolz

We will terminate (not pursue this path) if the string is not a valid word or if we've already visited this word.

This is essentially a depth-first search where there is an "edge" between two words if they are only one edit apart. This means that this algorithm will not find the shortest path. It will only find a path.

If we wanted to find the shortest path, we would want to use breadth-first search.

```
LinkedList<String> transform(String start, String stop, String[] words) {
  HashSet<String> dict = setupDictionary(words);
  HashSet<String> visited = new HashSet<String>();
  return transform(visited, start, stop, dict);
}

HashSet<String> setupDictionary(String[] words) {
  HashSet<String> hash = new HashSet<String>();
  for (String word : words) {
    hash.add(word.toLowerCase());
  }
  return hash;
}

LinkedList<String> transform(HashSet<String> visited, String startWord,
                               String stopWord, Set<String> dictionary) {
  if (startWord.equals(stopWord)) {
    LinkedList<String> path = new LinkedList<String>();
    path.add(startWord);
    return path;
  } else if (visited.contains(startWord) || !dictionary.contains(startWord)) {
    return null;
  }

  visited.add(startWord);
  ArrayList<String> words = wordsOneAway(startWord);

  for (String word : words) {
    LinkedList<String> path = transform(visited, word, stopWord, dictionary);
    if (path != null) {
      path.addFirst(startWord);
      return path;
    }
  }

  return null;
}

ArrayList<String> wordsOneAway(String word) {
  ArrayList<String> words = new ArrayList<String>();
  for (int i = 0; i < word.length(); i++) {
    for (char c = 'a'; c <= 'z'; c++) {
      String w = word.substring(0, i) + c + word.substring(i + 1);
      words.add(w);
    }
  }
  return words;
}
```

One major inefficiency in this algorithm is finding all strings that are one edit away. Right now, we're finding the strings that are one edit away and then eliminating the invalid ones.

Ideally, we want to only go to the ones that are valid.

Optimized Solution

To travel to only valid words, we clearly need a way of going from each word to a list of all the valid related words.

What makes two words "related" (one edit away)? They are one edit away if all but one character is the same. For example, `ball` and `bill` are one edit away, because they are both in the form `b_ll`. Therefore, one approach is to group all words that look like `b_ll` together.

We can do this for the whole dictionary by creating a mapping from a "wildcard word" (like `b_ll`) to a list of all words in this form. For example, for a very small dictionary like `{all, ill, ail, ape, ale}` the mapping might look like this:

```
_il -> ail
_le -> ale
_ll -> all, ill
_pe -> ape
a_e -> ape, ale
a_l -> all, ail
```

```
i_l -> ill
ai_ -> ail
al_ -> all, ale
ap_ -> ape
il_ -> ill
```

Now, when we want to know the words that are one edit away from a word like ale, we look up _le, a_e, and al_ in the hash table.

The algorithm is otherwise essentially the same.

```
LinkedList<String> transform(String start, String stop, String[] words) {
  HashMapList<String, String> wildcardToWordList = createWildcardToWordMap(words);
  HashSet<String> visited = new HashSet<String>();
  return transform(visited, start, stop, wildcardToWordList);
}

/* Do a depth-first search from startWord to stopWord, traveling through each word
 * that is one edit away. */
LinkedList<String> transform(HashSet<String> visited, String start, String stop,
                             HashMapList<String, String> wildcardToWordList) {
  if (start.equals(stop)) {
    LinkedList<String> path = new LinkedList<String>();
    path.add(start);
    return path;
  } else if (visited.contains(start)) {
    return null;
  }

  visited.add(start);
  ArrayList<String> words = getValidLinkedWords(start, wildcardToWordList);

  for (String word : words) {
    LinkedList<String> path = transform(visited, word, stop, wildcardToWordList);
    if (path != null) {
      path.addFirst(start);
      return path;
    }
  }

  return null;
}

/* Insert words in dictionary into mapping from wildcard form -> word. */
HashMapList<String, String> createWildcardToWordMap(String[] words) {
  HashMapList<String, String> wildcardToWords = new HashMapList<String, String>();
  for (String word : words) {
    ArrayList<String> linked = getWildcardRoots(word);
    for (String linkedWord : linked) {
      wildcardToWords.put(linkedWord, word);
    }
  }
  return wildcardToWords;
}

/* Get list of wildcards associated with word. */
ArrayList<String> getWildcardRoots(String w) {
  ArrayList<String> words = new ArrayList<String>();
  for (int i = 0; i < w.length(); i++) {
    String word = w.substring(0, i) + "_" + w.substring(i + 1);
    words.add(word);
  }
  return words;
}

/* Return words that are one edit away. */
ArrayList<String> getValidLinkedWords(String word,
    HashMapList<String, String> wildcardToWords) {
  ArrayList<String> wildcards = getWildcardRoots(word);
  ArrayList<String> linkedWords = new ArrayList<String>();
  for (String wildcard : wildcards) {
    ArrayList<String> words = wildcardToWords.get(wildcard);
    for (String linkedWord : words) {
      if (!linkedWord.equals(word)) {
        linkedWords.add(linkedWord);
      }
    }
  }
  return linkedWords;
}

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

This will work, but we can still make it faster.

One optimization is to switch from depth-first search to breadth-first search. If there are zero paths or one path, the algorithms are equivalent speeds. However, if there are multiple paths, breadth-first search may run faster.

Breadth-first search finds the shortest path between two nodes, whereas depth-first search finds any path. This means that depth-first search might take a very long, windy path in order to find a connection when, in fact, the nodes were quite close.

Optimal Solution

As noted earlier, we can optimize this using breadth-first search. Is this as fast as we can make it? Not quite.

Imagine that the path between two nodes has length 4. With breadth-first search, we will visit about 15^4 nodes to find them.

Breadth-first search spans out very quickly.

Instead, what if we searched out from the source and destination nodes simultaneously? In this case, the breadth-first searches would collide after each had done about two levels each.

- Nodes travelled to from source: 15^2
- Nodes travelled to from destination: 15^2
- Total nodes: $15^2 + 15^2$

This is much better than the traditional breadth-first search.

We will need to track the path that we've travelled at each node.

To implement this approach, we've used an additional class BFSData. BFSData helps us keep things a bit clearer, and allows us to keep a similar framework for the two simultaneous breadth-first searches. The alternative is to keep passing around a bunch of separate variables.

```java
LinkedList<String> transform(String startWord, String stopWord, String[] words) {
  HashMapList<String, String> wildcardToWordList = getWildcardToWordList(words);

  BFSData sourceData = new BFSData(startWord);
  BFSData destData = new BFSData(stopWord);

  while (!sourceData.isFinished() && !destData.isFinished()) {
    /* Search out from source. */
    String collision = searchLevel(wildcardToWordList, sourceData, destData);
    if (collision != null) {
      return mergePaths(sourceData, destData, collision);
    }

    /* Search out from destination. */
    collision = searchLevel(wildcardToWordList, destData, sourceData);
    if (collision != null) {
      return mergePaths(sourceData, destData, collision);
    }
  }

  return null;
}

/* Search one level and return collision, if any. */
String searchLevel(HashMapList<String, String> wildcardToWordList,
                   BFSData primary, BFSData secondary) {
  /* We only want to search one level at a time. Count how many nodes are
   * currently in the primary's level and only do that many nodes. We'll continue
   * to add nodes to the end. */
  int count = primary.toVisit.size();
  for (int i = 0; i < count; i++) {
    /* Pull out first node. */
    PathNode pathNode = primary.toVisit.poll();
    String word = pathNode.getWord();

    /* Check if it's already been visited. */
    if (secondary.visited.containsKey(word)) {
      return pathNode.getWord();
    }

    /* Add friends to queue. */
    ArrayList<String> words = getValidLinkedWords(word, wildcardToWordList);
    for (String w : words) {
      if (!primary.visited.containsKey(w)) {
        PathNode next = new PathNode(w, pathNode);
        primary.visited.put(w, next);
        primary.toVisit.add(next);
      }
    }
  }
  return null;
}
```

```
LinkedList<String> mergePaths(BFSData bfs1, BFSData bfs2, String connection) {
  PathNode end1 = bfs1.visited.get(connection); // end1 -> source
  PathNode end2 = bfs2.visited.get(connection); // end2 -> dest
  LinkedList<String> pathOne = end1.collapse(false); // forward
  LinkedList<String> pathTwo = end2.collapse(true); // reverse
  pathTwo.removeFirst(); // remove connection
  pathOne.addAll(pathTwo); // add second path
  return pathOne;
}

/* Methods getWildcardRoots, getWildcardToWordList, and getValidLinkedWords are
 * the same as in the earlier solution. */

public class BFSData {
  public Queue<PathNode> toVisit = new LinkedList<PathNode>();
  public HashMap<String, PathNode> visited = new HashMap<String, PathNode>();

  public BFSData(String root) {
    PathNode sourcePath = new PathNode(root, null);
    toVisit.add(sourcePath);
    visited.put(root, sourcePath);
  }

  public boolean isFinished() {
    return toVisit.isEmpty();
  }
}

public class PathNode {
  private String word = null;
  private PathNode previousNode = null;
  public PathNode(String word, PathNode previous) {
    this.word = word;
    previousNode = previous;
  }

  public String getWord() {
    return word;
  }

  /* Traverse path and return linked list of nodes. */
  public LinkedList<String> collapse(boolean startsWithRoot) {
    LinkedList<String> path = new LinkedList<String>();
    PathNode node = this;
    while (node != null) {
      if (startsWithRoot) {
        path.addLast(node.word);
      } else {
        path.addFirst(node.word);
      }
      node = node.previousNode;
    }
    return path;
  }
}
```

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

This algorithm's runtime is a bit harder to describe since it depends on what the language looks like, as well as the actual source and destination words. One way of expressing it is that if each word has E words that are one edit away and the source and destination are distance D, the runtime is $O(E^{D/2})$. This is how much work each breadth-first search does.

Of course, this is a lot of code to implement in an interview. It just wouldn't be possible. More realistically, you'd leave out a lot of the details. You might write just the skeleton code of `transform` and `searchLevel`, but leave out the rest.

17.23 Max Square Matrix: Imagine you have a square matrix, where each cell (pixel) is either black or white. Design an algorithm to find the maximum subsquare such that all four borders are filled with black pixels.

pg 190

SOLUTION

Like many problems, there's an easy way and a hard way to solve this. We'll go through both solutions.

The "Simple" Solution: $O(N^4)$

We know that the biggest possible square has a length of size N, and there is only one possible square of size NxN. We can easily check for that square and return if we find it.

If we do not find a square of size NxN, we can try the next best thing: `(N-1) x (N-1)`. We iterate through all squares of this size and return the first one we find. We then do the same for `N-2`, `N-3`, and so on. Since we are searching progressively smaller squares, we know that the first square we find is the biggest.

Our code works as follows:

```
Subsquare findSquare(int[][] matrix) {
  for (int i = matrix.length; i >= 1; i--) {
    Subsquare square = findSquareWithSize(matrix, i);
    if (square != null) return square;
  }
  return null;
}

Subsquare findSquareWithSize(int[][] matrix, int squareSize) {
  /* On an edge of length N, there are (N - sz + 1) squares of length sz. */
  int count = matrix.length - squareSize + 1;

  /* Iterate through all squares with side length squareSize. */
  for (int row = 0; row < count; row++) {
    for (int col = 0; col < count; col++) {
      if (isSquare(matrix, row, col, squareSize)) {
        return new Subsquare(row, col, squareSize);
      }
    }
  }
  return null;
}

boolean isSquare(int[][] matrix, int row, int col, int size) {
  // Check top and bottom border.
  for (int j = 0; j < size; j++){
    if (matrix[row][col+j] == 1) {
      return false;
    }
    if (matrix[row+size-1][col+j] == 1){
      return false;
    }
  }

  // Check left and right border.
  for (int i = 1; i < size - 1; i++){
    if (matrix[row+i][col] == 1){
      return false;
    }
    if (matrix[row+i][col+size-1] == 1) {
      return false;
    }
  }
  return true;
}
```

Pre-Processing Solution: O(N^3)

A large part of the slowness of the "simple" solution above is due to the fact we have to do O(N) work each time we want to check a potential square. By doing some pre-processing, we can cut down the time of `isSquare` to O(1). The time of the whole algorithm is reduced to O(N^3).

If we analyze what `isSquare` does, we realize that all it ever needs to know is if the next `squareSize` items, on the right of as well as below particular cells, are zeros. We can pre-compute this data in a straight-forward, iterative fashion.

We iterate from right to left, bottom to top. At each cell, we do the following computation:

```
if A[r][c] is white, zeros right and zeros below are 0
else A[r][c].zerosRight = A[r][c + 1].zerosRight + 1
     A[r][c].zerosBelow = A[r + 1][c].zerosBelow + 1
```

Below is an example of these values for a potential matrix.

(0s right, 0s below)

0,0	1,3	0,0
2,2	1,2	0,0
2,1	1,1	0,0

Original Matrix

W	B	W
B	B	W
B	B	W

Now, instead of iterating through O(N) elements, the `isSquare` method just needs to check `zerosRight` and `zerosBelow` for the corners.

Our code for this algorithm is below. Note that `findSquare` and `findSquareWithSize` is equivalent, other than a call to `processMatrix` and working with a new data type thereafter.

```
public class SquareCell {
  public int zerosRight = 0;
```