```
  public int zerosBelow = 0;
  /* declaration, getters, setters */
}

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

Subsquare findSquareWithSize(SquareCell[][] processed, int size) {
  /* equivalent to first algorithm */
}

boolean isSquare(SquareCell[][] matrix, int row, int col, int sz) {
  SquareCell topLeft = matrix[row][col];
  SquareCell topRight = matrix[row][col + sz - 1];
  SquareCell bottomLeft = matrix[row + sz - 1][col];

  /* Check top, left, right, and bottom edges, respectively. */
  if (topLeft.zerosRight < sz || topLeft.zerosBelow < sz ||
     topRight.zerosBelow < sz || bottomLeft.zerosRight < sz) {
    return false;
  }
  return true;
}

SquareCell[][] processSquare(int[][] matrix) {
  SquareCell[][] processed =
    new SquareCell[matrix.length][matrix.length];

  for (int r = matrix.length - 1; r >= 0; r--) {
    for (int c = matrix.length - 1; c >= 0; c--) {
      int rightZeros = 0;
      int belowZeros = 0;
      // only need to process if it's a black cell
      if (matrix[r][c] == 0) {
        rightZeros++;
        belowZeros++;
        // next column over is on same row
        if (c + 1 < matrix.length) {
          SquareCell previous = processed[r][c + 1];
          rightZeros += previous.zerosRight;
        }
        if (r + 1 < matrix.length) {
          SquareCell previous = processed[r + 1][c];
          belowZeros += previous.zerosBelow;
        }
      }
      processed[r][c] = new SquareCell(rightZeros, belowZeros);
    }
  }
  return processed;
```

```
}
```

17.24 Max Submatrix: Given an NxN matrix of positive and negative integers, write code to find the submatrix with the largest possible sum.

pg 190

SOLUTION

This problem can be approached in a variety of ways. We'll start with the brute force solution and then optimize the solution from there.

Brute Force Solution: O(N⁶)

Like many "maximizing" problems, this problem has a straightforward brute force solution. This solution simply iterates through all possible submatrices, computes the sum, and finds the largest.

To iterate through all possible submatrices (with no duplicates), we simply need to iterate through all ordered pairs of rows, and then all ordered pairs of columns.

This solution is $O(N^6)$, since we iterate through $O(N^4)$ submatrices and it takes $O(N^2)$ time to compute the area of each.

```
SubMatrix getMaxMatrix(int[][] matrix) {
  int rowCount = matrix.length;
  int columnCount = matrix[0].length;
  SubMatrix best = null;
  for (int row1 = 0; row1 < rowCount; row1++) {
    for (int row2 = row1; row2 < rowCount; row2++) {
      for (int col1 = 0; col1 < columnCount; col1++) {
        for (int col2 = col1; col2 < columnCount; col2++) {
          int sum = sum(matrix, row1, col1, row2, col2);
          if (best == null || best.getSum() < sum) {
            best = new SubMatrix(row1, col1, row2, col2, sum);
          }
        }
      }
    }
  }
  return best;
}

int sum(int[][] matrix, int row1, int col1, int row2, int col2) {
  int sum = 0;
  for (int r = row1; r <= row2; r++) {
    for (int c = col1; c <= col2; c++) {
      sum += matrix[r][c];
    }
  }
  return sum;
}

public class SubMatrix {
  private int row1, row2, col1, col2, sum;
  public SubMatrix(int r1, int c1, int r2, int c2, int sm) {
    row1 = r1;
    col1 = c1;
```

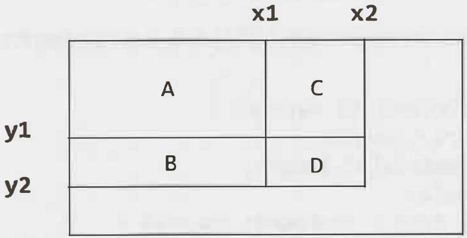
```
      row2 = r2;
      col2 = c2;
      sum = sm;
   }

   public int getSum() {
      return sum;
   }
}
```

It is good practice to pull the sum code into its own function since it's a fairly distinct set of code.

Dynamic Programming Solution: $O(N^4)$

Notice that the earlier solution is made slower by a factor of $O(N^2)$ simply because computing the sum of a matrix is so slow. Can we reduce the time to compute the area? Yes! In fact, we can reduce the time of computeSum to O(1).

Consider the following rectangle:



Suppose we knew the following values:

```
ValD = area(point(0, 0) -> point(x2, y2))
ValC = area(point(0, 0) -> point(x2, y1))
ValB = area(point(0, 0) -> point(x1, y2))
ValA = area(point(0, 0) -> point(x1, y1))
```

Each Val* starts at the origin and ends at the bottom right corner of a subrectangle.

With these values, we know the following:

```
area(D) = ValD - area(A union C) - area(A union B) + area(A).
```

Or, written another way:

```
area(D) = ValD - ValB - ValC + ValA
```

We can efficiently compute these values for all points in the matrix by using similar logic:

```
Val(x, y) = Val(x-1, y) + Val(y-1, x) - Val(x-1, y-1) + M[x][y]
```

We can precompute all such values and then efficiently find the maximum submatrix.

The following code implements this algorithm.

```
SubMatrix getMaxMatrix(int[][] matrix) {
   SubMatrix best = null;
   int rowCount = matrix.length;
   int columnCount = matrix[0].length;
   int[][] sumThrough = precomputeSums(matrix);

   for (int row1 = 0; row1 < rowCount; row1++) {
      for (int row2 = row1; row2 < rowCount; row2++) {
         for (int col1 = 0; col1 < columnCount; col1++) {
            for (int col2 = col1; col2 < columnCount; col2++) {
```

```
          int sum = sum(sumThrough, row1, col1, row2, col2);
          if (best == null || best.getSum() < sum) {
            best = new SubMatrix(row1, col1, row2, col2, sum);
          }
        }
      }
    }
  }
  return best;
}

int[][] precomputeSums(int[][] matrix) {
  int[][] sumThrough = new int[matrix.length][matrix[0].length];
  for (int r = 0; r < matrix.length; r++) {
    for (int c = 0; c < matrix[0].length; c++) {
      int left = c > 0 ? sumThrough[r][c - 1] : 0;
      int top = r > 0 ? sumThrough[r - 1][c] : 0;
      int overlap = r > 0 && c > 0 ? sumThrough[r-1][c-1] : 0;
      sumThrough[r][c] = left + top - overlap + matrix[r][c];
    }
  }
  return sumThrough;
}

int sum(int[][] sumThrough, int r1, int c1, int r2, int c2) {
  int topAndLeft = r1 > 0 && c1 > 0 ? sumThrough[r1-1][c1-1] : 0;
  int left = c1 > 0 ? sumThrough[r2][c1 - 1] : 0;
  int top = r1 > 0 ? sumThrough[r1 - 1][c2] : 0;
  int full = sumThrough[r2][c2];
  return full - left - top + topAndLeft;
}
```

This algorithm takes O(N^4) time, since it goes through each pair of rows and each pair of columns.

Optimized Solution: O(N^3)

Believe it or not, an even more optimal solution exists. If we have R rows and C columns, we can solve it in O(R^2C) time.

Recall the solution to the maximum subarray problem: "Given an array of integers, find the subarray with the largest sum." We can find the maximum subarray in O(N) time. We will leverage this solution for this problem.

Every submatrix can be represented by a contiguous sequence of rows and a contiguous sequence of columns. If we were to iterate through every contiguous sequence of rows, we would then just need to find, for each of those, the set of columns that gives us the highest sum. That is:

```
maxSum = 0
foreach rowStart in rows
  foreach rowEnd in rows
    /* We have many possible submatrices with rowStart and rowEnd as the top and
     * bottom edges of the matrix. Find the colStart and colEnd edges that give
     * the highest sum. */
    maxSum = max(runningMaxSum, maxSum)
return maxSum
```

Now the question is, how do we efficiently find the "best" `colStart` and `colEnd`?

Picture a submatrix:

rowStart

9	-8	1	3	-2
-3	7	6	-2	4
6	-4	-4	8	-7
12	-5	3	9	-5

rowEnd

Given a `rowStart` and `rowEnd`, we want to find the `colStart` and `colEnd` that give us the highest possible sum. To do this, we can sum up each column and then apply the `maximumSubArray` function explained at the beginning of this problem.

For the earlier example, the maximum subarray is the first through fourth columns. This means that the maximum submatrix is (`rowStart`, `first column`) through (`rowEnd`, `fourth column`).

We now have pseudocode that looks like the following.

```
1   maxSum = 0
2   foreach rowStart in rows
3     foreach rowEnd in rows
4       foreach col in columns
5         partialSum[col] = sum of matrix[rowStart, col] through matrix[rowEnd, col]
6       runningMaxSum = maxSubArray(partialSum)
7       maxSum = max(runningMaxSum, maxSum)
8   return maxSum
```

The sum in lines 5 and 6 takes R*C time to compute (since it iterates through `rowStart` through `rowEnd`), so this gives us a runtime of $O(R^3C)$. We're not quite done yet.

In lines 5 and 6, we're basically adding up `a[0]...a[i]` from scratch, even though in the previous iteration of the outer for loop, we already added up `a[0]...a[i-1]`. Let's cut out this duplicated effort.

```
maxSum = 0
foreach rowStart in rows
  clear array partialSum
  foreach rowEnd in rows
    foreach col in columns
      partialSum[col] += matrix[rowEnd, col]
    runningMaxSum = maxSubArray(partialSum)
  maxSum = max(runningMaxSum, maxSum)
return maxSum
```

Our full code looks like this:

```
SubMatrix getMaxMatrix(int[][] matrix) {
  int rowCount = matrix.length;
  int colCount = matrix[0].length;
  SubMatrix best = null;

  for (int rowStart = 0; rowStart < rowCount; rowStart++) {
    int[] partialSum = new int[colCount];

    for (int rowEnd = rowStart; rowEnd < rowCount; rowEnd++) {
      /* Add values at row rowEnd. */
      for (int i = 0; i < colCount; i++) {
        partialSum[i] += matrix[rowEnd][i];
      }

      Range bestRange = maxSubArray(partialSum, colCount);
      if (best == null || best.getSum() < bestRange.sum) {
        best = new SubMatrix(rowStart, bestRange.start, rowEnd,
                             bestRange.end, bestRange.sum);
      }
    }
  }
  return best;
}

Range maxSubArray(int[] array, int N) {
  Range best = null;
  int start = 0;
  int sum = 0;

  for (int i = 0; i < N; i++) {
    sum += array[i];
    if (best == null || sum > best.sum) {
      best = new Range(start, i, sum);
    }

    /* If running_sum is < 0 no point in trying to continue the series. Reset. */
    if (sum < 0) {
      start = i + 1;
      sum = 0;
    }
  }
  return best;
}

public class Range {
  public int start, end, sum;
  public Range(int start, int end, int sum) {
    this.start = start;
    this.end = end;
    this.sum = sum;
  }
}
```

This was an extremely complex problem. You would not be expected to figure out this entire problem in an interview without a lot of help from your interviewer.

17.25 Word Rectangle: Given a list of millions of words, design an algorithm to create the largest possible rectangle of letters such that every row forms a word (reading left to right) and every column forms a word (reading top to bottom). The words need not be chosen consecutively from the list, but all rows must be the same length and all columns must be the same height.

pg 190

SOLUTION

Many problems involving a dictionary can be solved by doing some pre-processing. Where can we do pre-processing?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
                if (rectangle != null) return rectangle;
            }
        }
    }
  }
  return null;
}
```

The makeRectangle method is called by maxRectangle and tries to build a rectangle of a specific length and height.

```
Rectangle makeRectangle(int length, int height) {
  if (groupList[length-1] == null || groupList[height-1] == null) {
    return null;
  }

  /* Create trie for word length if we haven't yet */
  if (trieList[height - 1] == null) {
    LinkedList<String> words = groupList[height - 1].getWords();
    trieList[height - 1] = new Trie(words);
  }

  return makePartialRectangle(length, height, new Rectangle(length));
}
```

The makePartialRectangle method is where the action happens. It is passed in the intended, final length and height, and a partially formed rectangle. If the rectangle is already of the final height, then we just check to see if the columns form valid, complete words, and return.

Otherwise, we check to see if the columns form valid prefixes. If they do not, then we immediately break since there is no way to build a valid rectangle off of this partial one.

But, if everything is okay so far, and all the columns are valid prefixes of words, then we search through all the words of the right length, append each to the current rectangle, and recursively try to build a rectangle off of {current rectangle with new word appended}.

```
Rectangle makePartialRectangle(int l, int h, Rectangle rectangle) {
  if (rectangle.height == h) { // Check if complete rectangle
    if (rectangle.isComplete(l, h, groupList[h - 1])) {
      return rectangle;
    }
    return null;
  }

  /* Compare columns to trie to see if potentially valid rect */
  if (!rectangle.isPartialOK(l, trieList[h - 1])) {
    return null;
  }

  /* Go through all words of the right length. Add each one to the current partial
   * rectangle, and attempt to build a rectangle recursively. */
  for (int i = 0; i < groupList[l-1].length(); i++) {
    /* Create a new rectangle which is this rect + new word. */
    Rectangle orgPlus = rectangle.append(groupList[l-1].getWord(i));

    /* Try to build a rectangle with this new, partial rect */
    Rectangle rect = makePartialRectangle(l, h, orgPlus);
    if (rect != null) {
      return rect;
      }
    }
    return null;
  }
```

The Rectangle class represents a partially or fully formed rectangle of words. The method isPartialOk can be called to check if the rectangle is, thus far, a valid one (that is, all the columns are prefixes of words). The method isComplete serves a similar function, but checks if each of the columns makes a full word.

```
public class Rectangle {
  public int height, length;
  public char[][] matrix;

  / *Construct an “empty” rectangule. Length is fixed, but height varies as we add
   * words. */
  public Rectangle(int l) {
    height = 0;
    length = l;
  }

  / *Construct a rectangular array of letters of the specified length and height,
   * and backed by the specified matrix of letters. (It is assumed that the length
   * and height specified as arguments are consistent with the array argument’s
   * dimensions.) */
  public Rectangle(int length, int height, char[][] letters) {
    this.height = letters.length;
    this.length = letters[0].length;
    matrix = letters;
  }

  public char getLetter (int i, int j) { return matrix[i][j]; }
  public String getColumn(int i) { ... }

  / *Check if all columns are valid. All rows are already known to be valid since
   * they were added directly from dictionary. */
  public boolean isComplete(int l, int h, WordGroup groupList) {
    if (height == h) {
      / *Check if each column is a word in the dictionary. */
      for (int i = 0; i < l; i++) {
        String col = getColumn(i);
        if (!groupList.containsWord(col)) {
          return false;
        }
      }
      return true;
    }
    return false;
  }

  public boolean isPartialOK(int l, Trie trie) {
    if (height == 0) return true;
    for (int i = 0; i < l; i++ ) {
      String col = getColumn(i);
      if (!trie.contains(col)) {
        return false;
      }
    }
    return true;
  }

  / *Create a new Rectangle by taking the rows of the current rectangle and
   * appending s. */
  public Rectangle append(String s) { ... }
}
```

The WordGroup class is a simple container for all words of a specific length. For easy lookup, we store the words in a hash table as well as in an ArrayList.

The lists in WordGroup are created through a static method called createWordGroups.

```
public class WordGroup {
  private HashMap<String, Boolean> lookup = new HashMap<String, Boolean>();
  private ArrayList<String> group = new ArrayList<String>();
  public boolean containsWord(String s) { return lookup.containsKey(s); }
  public int length() { return group.size(); }
  public String getWord(int i) { return group.get(i); }
  public ArrayList<String> getWords() { return group; }

  public void addWord (String s) {
    group.add(s);
    lookup.put(s, true);
  }

  public static WordGroup[] createWordGroups(String[] list) {
    WordGroup[] groupList;
    int maxWordLength = 0;
    / *Find the length of the longest word */
    for (int i = 0; i < list.length; i++) {
      if (list[i].length() > maxWordLength) {
        maxWordLength = list[i].length();
      }
    }

    / *Group the words in the dictionary into lists of words of same length.
     * groupList[i] will contain a list of words, each of length (i+1). */
    groupList = new WordGroup[maxWordLength];
    for (int i = 0; i < list.length; i++) {
      / *We do wordLength - 1 instead of just wordLength since this is used as
       * an index and no words are of length 0 */
      int wordLength = list[i].length() - 1;
      if (groupList[wordLength] == null) {
        groupList[wordLength] = new WordGroup();
      }
      groupList[wordLength].addWord(list[i]);
    }
    return groupList;
  }
}
```

The full code for this problem, including the code for Trie and TrieNode, can be found in the code attachment. Note that in a problem as complex as this, you'd most likely only need to write the pseudocode. Writing the entire code would be nearly impossible in such a short amount of time.