```
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.