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.

```java
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;
      }
```

```java
    }

    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.

```java
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;
```

```
    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:

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

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

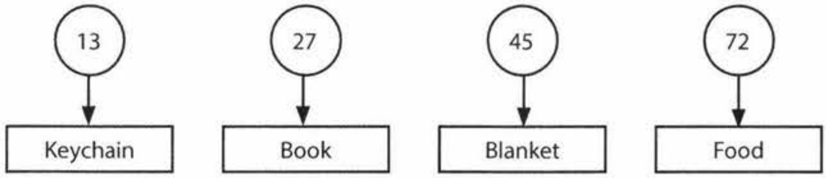
SOLUTION

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.

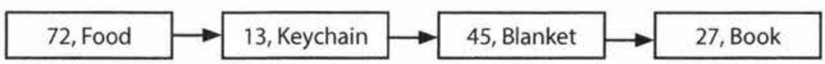
- **Retrieving Value by Key:** We need to be able to retrieve the value using the key.
- **Finding Least Recently Used:** We need to know the least recently used item (and, likely, the usage ordering of all items).
- **Updating Most Recently Used:** When we retrieve a value by key, we need to update 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).

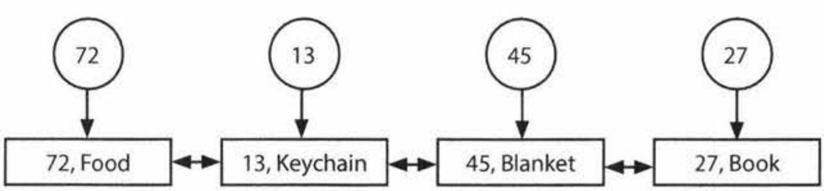


Unfortunately, this does not offer a quick way to look up an item by its key. We could iterate through the linked list and find the item by key. But this could get very slow (O(N) for retrieval).

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!

17

Solutions to Hard

17.1 Add Without Plus: Write a function that adds two numbers. You should not use + or any arithmetic operators.

pg 186

SOLUTION

Our first instinct in problems like these should be that we're going to have to work with bits. Why? Because when you take away the + sign, what other choice do we have? Plus, that's how computers do it!

Our next thought should be to deeply understand how addition works. We can walk through an addition problem to see if we can understand something new—some pattern—and then see if we can replicate that with code.

So let's do just that—let's walk through an addition problem. We'll work in base 10 so that it's easier to see.

To add 759 + 674, I would usually add `digit[0]` from each number, carry the one, add `digit[1]` from each number, carry the one, and so on. You could take the same approach in binary: add each digit, and carry the one as necessary.

Can we make this a little easier? Yes! Imagine I decided to split apart the "addition" and "carry" steps. That is, I do the following:

1. Add 759 + 674, but "forget" to carry. I then get 323.
2. Add 759 + 674 but only do the carrying, rather than the addition of each digit. I then get 1110.
3. Add the result of the first two operations (recursively, using the same process described in step 1 and 2): 1110 + 323 = 1433.

Now, how would we do this in binary?

1. If I add two binary numbers together, but forget to carry, the `i`th bit in the sum will be `0` only if a and b have the same `i`th bit (both `0` or both `1`). This is essentially an XOR.
2. If I add two numbers together but *only* carry, I will have a 1 in the `i`th bit of the sum only if bits `i - 1` of a and b are both 1s. This is an AND, shifted.
3. Now, recurse until there's nothing to carry.

The following code implements this algorithm.

```
int add(int a, int b) {
  if (b == 0) return a;
  int sum = a ^ b; // add without carrying
  int carry = (a & b) << 1; // carry, but don't add
```

```
  return add(sum, carry); // recurse with sum + carry
}
```

Alternatively, you can implement this iteratively.

```
int add(int a, int b) {
  while (b != 0) {
    int sum = a ^ b; // add without carrying
    int carry = (a & b) << 1; // carry, but don't add
    a = sum;
    b = carry;
  }
  return a;
}
```

Problems requiring us to implement core operations like addition and subtraction are relatively common. The key in all of these problems is to dig into how these operations are usually implemented, so that we can re-implement them with the constraints of the given problem.

17.2 Shuffle: Write a method to shuffle a deck of cards. It must be a perfect shuffle—in other words, each of the 52! permutations of the deck has to be equally likely. Assume that you are given a random number generator which is perfect.

pg 186

SOLUTION

This is a very well known interview question, and a well known algorithm. If you aren't one of the lucky few to already know this algorithm, read on.

Let's imagine our n-element array. Suppose it looks like this:

```
[1] [2] [3] [4] [5]
```

Using our Base Case and Build approach, we can ask this question: suppose we had a method `shuffle(...)` that worked on `n - 1` elements. Could we use this to shuffle n elements?

Sure. In fact, that's quite easy. We would first shuffle the first n - 1 elements. Then, we would take the nth element and randomly swap it with an element in the array. That's it!

Recursively, that algorithm looks like this:

```
/* Random number between lower and higher, inclusive */
int rand(int lower, int higher) {
  return lower + (int)(Math.random() * (higher - lower + 1));
}

int[] shuffleArrayRecursively(int[] cards, int i) {
  if (i == 0) return cards;

  shuffleArrayRecursively(cards, i - 1); // Shuffle earlier part
  int k = rand(0, i); // Pick random index to swap with

  /* Swap element k and i */
  int temp = cards[k];
  cards[k] = cards[i];
  cards[i] = temp;

  /* Return shuffled array */
  return cards;
```

```
}
```

What would this algorithm look like iteratively? Let's think about it. All it does is moving through the array and, for each element `i`, swapping `array[i]` with a random element between `0` and `i`, inclusive.

This is actually a very clean algorithm to implement iteratively:

```
void shuffleArrayIteratively(int[] cards) {
  for (int i = 0; i < cards.length; i++) {
    int k = rand(0, i);
    int temp = cards[k];
    cards[k] = cards[i];
    cards[i] = temp;
  }
}
```

The iterative approach is usually how we see this algorithm written.

17.3 Random Set: Write a method to randomly generate a set of `m` integers from an array of size n. Each element must have equal probability of being chosen.

pg 186

SOLUTION

Like the prior problem which was similar, (problem 17.2 on page 531), we can look at this problem recursively using the Base Case and Build approach.

Suppose we have an algorithm that can pull a random set of `m` elements from an array of size `n - 1`. How can we use this algorithm to pull a random set of `m` elements from an array of size `n`?

We can first pull a random set of size `m` from the first `n - 1` elements. Then, we just need to decide if `array[n]` should be inserted into our subset (which would require pulling out a random element from it). An easy way to do this is to pick a random number k from `0` through n. If `k < m`, then insert `array[n]` into `subset[k]`. This will both "fairly" (i.e., with proportional probability) insert `array[n]` into the subset and "fairly" remove a random element from the subset.

The pseudocode for this recursive algorithm would look like this:

```
int[] pickMRecursively(int[] original, int m, int i) {
  if (i + 1 == m) { // Base case
    /* return first m elements of original */
  } else if (i + 1 > m) {
    int[] subset = pickMRecursively(original, m, i - 1);
    int k =  random value between 0 and i, inclusive
    if (k < m) {
      subset[k] = original[i];
    }
    return subset;
  }
  return null;
}
```

This is even cleaner to write iteratively. In this approach, we initialize an array `subset` to be the first `m` elements in `original`. Then, we iterate through the array, starting at element `m`, inserting `array[i]` into the subset at (random) position k whenever `k < m`.

```
int[] pickMIteratively(int[] original, int m) {
  int[] subset = new int[m];

```

```
    /* Fill in subset array with first part of original array */
    for (int i = 0; i < m ; i++) {
       subset[i] = original[i];
    }

    /* Go through rest of original array. */
    for (int i = m; i < original.length; i++) {
       int k = rand(0, i); // Random # between 0 and i, inclusive
       if (k < m) {
          subset[k] = original[i];
       }
    }

    return subset;
}
```

Both solutions are, not surprisingly, very similar to the algorithm to shuffle an array.

17.4 Missing Number: An array A contains all the integers from 0 to n, except for one number which is missing. In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the jth bit of A[i]," which takes constant time. Write code to find the missing integer. Can you do it in O(n) time?

pg 186

SOLUTION

You may have seen a very similar sounding problem: Given a list of numbers from 0 to n, with exactly one number removed, find the missing number. This problem can be solved by simply adding the list of numbers and comparing it to the actual sum of 0 through n, which is $\frac{n(n+1)}{2}$. The difference will be the missing number.

We could solve this by computing the value of each number, based on its binary representation, and calculating the sum.

The runtime of this solution is `n * length(n)`, when `length` is the number of bits in n. Note that `length(n)` = $\log_2$`(n)`. So, the runtime is actually `O(n log(n))`. Not quite good enough!

So how else can we approach it?

We can actually use a similar approach, but leverage the bit values more directly.

Picture a list of binary numbers (the `-----` indicates the value that was removed):

00000	00100	01000	01100
00001	00101	01001	01101
00010	00110	01010	
-----	00111	01011	

Removing the number above creates an imbalance of 1s and 0s in the least significant bit, which we'll call LSB_1. In a list of numbers from 0 to n, we would expect there to be the same number of 0s as 1s (if n is odd), or an additional 0 if n is even. That is:

```
if n % 2 == 1 then count(0s) = count(1s)
if n % 2 == 0 then count(0s) = 1 + count(1s)
```

Note that this means that `count(0s)` is always greater than or equal to `count(1s)`.

When we remove a value v from the list, we'll know immediately if v is even or odd just by looking at the least significant bits of all the other values in the list.

	n % 2 == 0 **count(0s) = 1 + count(1s)**	**n % 2 == 1** **count(0s) = count(1s)**
v % 2 == 0 **LSB$_1$(v) = 0**	a 0 is removed. count(0s) = count(1s)	a 0 is removed. count(0s) < count(1s)
v % 2 == 1 **LSB$_1$(v) = 1**	a 1 is removed. count(0s) > count(1s)	a 1 is removed. count(0s) > count(1s)

So, if `count(0s) <= count(1s)`, then v is even. If `count(0s) > count(1s)`, then v is odd.

We can now remove all the evens and focus on the odds, or remove all the odds and focus on the evens.

Okay, but how do we figure out what the next bit in v is? If v were contained in our (now smaller) list, then we should expect to find the following (where count$_2$ indicates the number of 0s or 1s in the second least significant bit):

count$_2$(0s) = count$_2$(1s)   OR   count$_2$(0s) = 1 + count$_2$(1s)

As in the earlier example, we can deduce the value of the second least significant bit (LSB$_2$) of v.

	count$_2$(0s) = 1 + count$_2$(1s)	**count$_2$(0s) = count$_2$(1s)**
LSB$_2$(v) == 0	a 0 is removed. count$_2$(0s) = count$_2$(1s)	a 0 is removed. count$_2$(0s) < count$_2$(1s)
LSB$_2$(v) == 1	a 1 is removed. count$_2$(0s) > count$_2$(1s)	a 1 is removed. count$_2$(0s) > count$_2$(1s)

Again, we have the same conclusion:

- If count$_2$(0s) <= count$_2$(1s), then LSB$_2$(v) = 0.
- If count$_2$(0s) > count$_2$(1s), then LSB$_2$(v) = 1.

We can repeat this process for each bit. On each iteration, we count the number of 0s and 1s in bit `i` to check if LSB$_i$(v) is 0 or 1. Then, we discard the numbers where LSB$_i$(x) != LSB$_i$(v). That is, if v is even, we discard the odd numbers, and so on.

By the end of this process, we will have computed all bits in v. In each successive iteration, we look at n, then n / 2, then n / 4, and so on, bits. This results in a runtime of `O(N)`.

If it helps, we can also move through this more visually. In the first iteration, we start with all the numbers:

00000	00100	01000	01100
00001	00101	01001	01101
00010	00110	01010	
-----	00111	01011	

Since count$_1$(0s) > count$_1$(1s), we know that LSB$_1$(v) = 1. Now, discard all numbers x where LSB$_1$(x) != LSB$_1$(v).

~~00000~~	~~00100~~	~~01000~~	~~01100~~
00001	00101	01001	01101
~~00010~~	~~00110~~	~~01010~~	
-----	00111	01011	

Now, count$_2$(0s) > count$_2$(1s), so we know that LSB$_2$(v) = 1. Now, discard all numbers x where LSB$_2$(x) != LSB$_2$(v).

~~00000~~	~~00100~~	~~01000~~	~~01100~~
~~00001~~	~~00101~~	~~01001~~	~~01101~~
~~00010~~	~~00110~~	~~01010~~	
-----	00111	01011	

This time, $count_3$`(0s) <= `$count_3$`(1s)`, we know that LSB_3`(v) = 0`. Now, discard all numbers x where LSB_3`(x) != `LSB_3`(v)`.

~~00000~~	~~00100~~	~~01000~~	~~01100~~
~~00001~~	~~00101~~	~~01001~~	~~01101~~
~~00010~~	~~00110~~	~~01010~~	
-----	~~00111~~	01011	

We're down to just one number. In this case, $count_4$`(0s) <= `$count_4$`(1s)`, so LSB_4`(v) = 0`.

When we discard all numbers where LSB_4`(x) != 0`, we'll wind up with an empty list. Once the list is empty, then $count_i$`(0s) <= `$count_i$`(1s)`, so LSB_i`(v) = 0`. In other words, once we have an empty list, we can fill in the rest of the bits of v with 0.

This process will compute that, for the example above, `v = 00011`.

The code below implements this algorithm. We've implemented the discarding aspect by partitioning the array by bit value as we go.

```
1  int findMissing(ArrayList<BitInteger> array) {
2    /* Start from the least significant bit, and work our way up */
3    return findMissing(array, 0);
4  }
5
6  int findMissing(ArrayList<BitInteger> input, int column) {
7    if (column >= BitInteger.INTEGER_SIZE) { // We're done!
8      return 0;
9    }
10   ArrayList<BitInteger> oneBits = new ArrayList<BitInteger>(input.size()/2);
11   ArrayList<BitInteger> zeroBits = new ArrayList<BitInteger>(input.size()/2);
12
13   for (BitInteger t : input) {
14     if (t.fetch(column) == 0) {
15       zeroBits.add(t);
16     } else {
17       oneBits.add(t);
18     }
19   }
20   if (zeroBits.size() <= oneBits.size()) {
21     int v = findMissing(zeroBits, column + 1);
22     return (v << 1) | 0;
23   } else {
24     int v = findMissing(oneBits, column + 1);
25     return (v << 1) | 1;
26   }
27 }
```

In lines 24 and 27, we recursively calculate the other bits of v. Then, we insert either a 0 or 1, depending on whether or not $count_i$`(0s) <= `$count_i$`(1s)`.

17.5 Letters and Numbers: Given an array filled with letters and numbers, find the longest subarray with an equal number of letters and numbers.

pg 186

SOLUTION

In the introduction, we discussed the importance of creating a really good, general-purpose example. That's absolutely true. It's also important, though, to understand what matters.

In this case, we just want an equal number of letters and numbers. All letters are treated identically and all numbers are treated identically. Therefore, we can use an example with a single letter and a single number—or, for that matter, As and Bs, 0s and 1s, or `Thing1`s and `Thing2`s.

With that said, let's start with an example:

```
[A, B, A, A, A, B, B, B, A, B, A, A, B, B, A, A, A, A, A, A]
```

We're looking for the smallest subarray where `count(A, subarray) = count(B, subarray)`.

Brute Force

Let's start with the obvious solution. Just go through all subarrays, count the number of As and Bs (or letters and numbers), and find the longest one that is equal.

We can make one small optimization to this. We can start with the longest subarray and, as soon as we find one which fits this equality condition, return it.

```java
/* Return the largest subarray with equal number of 0s and 1s. Look at each
 * subarray, starting from the longest. As soon as we find one that's equal, we
 * return.
char[] findLongestSubarray(char[] array) {
  for (int len = array.length; len > 1; len--) {
    for (int i = 0; i <= array.length - len; i++) {
      if (hasEqualLettersNumbers(array, i, i + len - 1)) {
        return extractSubarray(array, i, i + len - 1);
      }
    }
  }
  return null;
}

/* Check if subarray has equal number of letters and numbers. */
boolean hasEqualLettersNumbers(char[] array, int start, int end) {
  int counter = 0;
  for (int i = start; i <= end; i++) {
    if (Character.isLetter(array[i])) {
      counter++;
    } else if (Character.isDigit(array[i])) {
      counter--;
    }
  }
  return counter == 0;
}

/* Return subarray of array between start and end (inclusive). */
char[] extractSubarray(char[] array, int start, int end) {
  char[] subarray = new char[end - start + 1];
  for (int i = start; i <= end; i++) {
    subarray[i - start] = array[i];
```

```
    }
    return subarray;
}
```

Despite the one optimization we made, this algorithm is still $O(N^2)$, where N is the length of the array.

Optimal Solution

What we're trying to do is find a subarray where the count of letters equals the count of numbers. What if we just started from the beginning, counting the number of letters and numbers?

	a	a	a	a	1	1	a	1	1	a	a	1	a	a	1	a	a	a	a	a
#a	1	2	3	4	4	4	5	5	5	6	7	7	8	9	9	10	11	12	13	14
#1	0	0	0	0	1	2	2	3	4	4	4	5	5	5	6	6	6	6	6	6

Certainly, whenever the number of letters equals the number of numbers, we can say that from index 0 to that index is an "equal" subarray.

That will only tell us equal subarrays that start at index 0. How can we identify all equal subarrays?

Let's picture this. Suppose we inserted an equal subarray (like `a11a1a`) after an array like `a1aaa1`. How would that impact the counts?

	a	1	a	a	a	1		a	1	1	a	1	a
#a	1	1	2	3	4	4		5	5	5	6	6	7
#1	0	1	1	1	1	2		2	3	4	4	5	5

Study the numbers before the subarray (4, 2) and the end (7, 5). You might notice that, while the values aren't the same, the differences are: `4 - 2 = 7 - 5`. This makes sense. Since they've added the same number of letters and numbers, they should maintain the same difference.

> Observe that when the difference is the same, the subarray starts one after the initial matching index and continues through the final matching index. This explains line 10 in the code below.

Let's update the earlier array with the differences.

	a	a	a	a	1	1	a	1	1	a	a	1	a	a	1	a	a	a	a	a
#a	1	2	3	4	4	4	5	5	5	6	7	7	8	9	9	10	11	12	13	14
#1	0	0	0	0	1	2	2	3	4	4	4	5	5	5	6	6	6	6	6	6
-	1	2	3	4	3	2	3	2	1	2	3	2	3	4	3	4	5	6	7	8

Whenever we return the same difference, then we know we have found an equal subarray. To find the biggest subarray, we just have to find the two indices farthest apart with the same value.

To do so, we use a hash table to store the first time we see a particular difference. Then, each time we see the same difference, we see if this subarray (from first occurrence of this index to current index) is bigger than the current max. If so, we update the max.

```
1   char[] findLongestSubarray(char[] array) {
2     /* Compute deltas between count of numbers and count of letters. */
3     int[] deltas = computeDeltaArray(array);
4
5     /* Find pair in deltas with matching values and largest span. */
6     int[] match = findLongestMatch(deltas);
7
8     /* Return the subarray. Note that it starts one *after* the initial occurence of
9      * this delta. */
10    return extract(array, match[0] + 1, match[1]);
11  }
12
```

```
/* Compute the difference between the number of letters and numbers between the
 * beginning of the array and each index. */
int[] computeDeltaArray(char[] array) {
  int[] deltas = new int[array.length];
  int delta = 0;
  for (int i = 0; i < array.length; i++) {
    if (Character.isLetter(array[i])) {
      delta++;
    } else if (Character.isDigit(array[i])) {
      delta--;
    }
    deltas[i] = delta;
  }
  return deltas;
}

/* Find the matching pair of values in the deltas array with the largest
 * difference in indices. */
int[] findLongestMatch(int[] deltas) {
  HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
  map.put(0, -1);
  int[] max = new int[2];
  for (int i = 0; i < deltas.length; i++) {
    if (!map.containsKey(deltas[i])) {
      map.put(deltas[i], i);
    } else {
      int match = map.get(deltas[i]);
      int distance = i - match;
      int longest = max[1] - max[0];
      if (distance > longest) {
        max[1] = i;
        max[0] = match;
      }
    }
  }
  return max;
}

char[] extract(char[] array, int start, int end) { /* same */ }
```

This solution takes O(N) time, where N is size of the array.

17.6 Count of 2s: Write a method to count the number of 2s between 0 and n.

pg 186

SOLUTION

Our first approach to this problem can be—and probably should be—a brute force solution. Remember that interviewers want to see how you're approaching a problem. Offering a brute force solution is a great way to start.

```
/* Counts the number of '2' digits between 0 and n */
int numberOf2sInRange(int n) {
  int count = 0;
  for (int i = 2; i <= n; i++) { // Might as well start at 2
    count += numberOf2s(i);
```

```
    }
    return count;
}

/* Counts the number of '2' digits in a single number */
int numberOf2s(int n) {
    int count = 0;
    while (n > 0) {
        if (n % 10 == 2) {
            count++;
        }
        n = n / 10;
    }
    return count;
}
```

The only interesting part is that it's probably cleaner to separate out `numberOf2s` into a separate method. This demonstrates an eye for code cleanliness.

Improved Solution

Rather than looking at the problem by ranges of numbers, we can look at the problem digit by digit. Picture a sequence of numbers:

```
  0   1   2   3   4   5   6   7   8   9
 10  11  12  13  14  15  16  17  18  19
 20  21  22  23  24  25  26  27  28  29
...
110 111 112 113 114 115 116 117 118 119
```

We know that roughly one tenth of the time, the last digit will be a 2 since it happens once in any sequence of ten numbers. In fact, any digit is a 2 roughly one tenth of the time.

We say "roughly" because there are (very common) boundary conditions. For example, between 1 and 100, the 10's digit is a 2 exactly $\frac{1}{10}$ th of the time. However, between 1 and 37, the 10's digit is a 2 much more than $1/10^{th}$ of the time.

We can work out what exactly the ratio is by looking at the three cases individually: `digit < 2`, `digit = 2`, and `digit > 2`.

Case digit < 2

Consider the value `x = 61523` and `d = 3`, and observe that `x[d] = 1` (that is, the dth digit of x is 1). There are 2s at the 3rd digit in the ranges `2000 - 2999`, `12000 - 12999`, `22000 - 22999`, `32000 - 32999`, `42000 - 42999`, and `52000 - 52999`. We will not yet have hit the range `62000 - 62999`, so there are 6000 2s total in the 3rd digit. This is the same amount as if we were just counting all the 2s in the 3rd digit between 1 and 60000.

In other words, we can round *down* to the nearest 10^{d+1}, and then divide by 10, to compute the number of 2s in the dth digit.

```
if x[d] < 2: count2sInRangeAtDigit(x, d) =
    let y = round down to nearest 10^(d+1)
    return y / 10
```

Case digit > 2

Now, let's look at the case where dth digit of x is greater than 2 (`x[d] > 2`). We can apply almost the exact same logic to see that there are the same number of 2s in the 3rd digit in the range `0 - 63525` as there as in the range `0 - 70000`. So, rather than rounding down, we round up.

```
if x[d] > 2: count2sInRangeAtDigit(x, d) =
    let y = round up to nearest 10^(d+1)
    return y / 10
```

Case digit = 2

The final case may be the trickiest, but it follows from the earlier logic. Consider `x = 62523` and `d = 3`. We know that there are the same ranges of 2s from before (that is, the ranges `2000 - 2999`, `12000 - 12999`, ..., `52000 - 52999`). How many appear in the 3rd digit in the final, partial range from `62000 - 62523`? Well, that should be pretty easy. It's just 524 (`62000, 62001, ..., 62523`).

```
if x[d] = 2: count2sInRangeAtDigit(x, d) =
    let y = round down to nearest 10^(d+1)
    let z = right side of x (i.e., x % 10^d)
    return y / 10 + z + 1
```

Now, all you need is to iterate through each digit in the number. Implementing this code is reasonably straightforward.

```java
int count2sInRangeAtDigit(int number, int d) {
   int powerOf10 = (int) Math.pow(10, d);
   int nextPowerOf10 = powerOf10 * 10;
   int right = number % powerOf10;

   int roundDown = number - number % nextPowerOf10;
   int roundUp = roundDown + nextPowerOf10;

   int digit = (number / powerOf10) % 10;
   if (digit < 2) { // if the digit in spot digit is
      return roundDown / 10;
   } else if (digit == 2) {
      return roundDown / 10 + right + 1;
   } else {
      return roundUp / 10;
   }
}

int count2sInRange(int number) {
   int count = 0;
   int len = String.valueOf(number).length();
   for (int digit = 0; digit < len; digit++) {
      count += count2sInRangeAtDigit(number, digit);
   }
   return count;
}
```

This question requires very careful testing. Make sure to generate a list of test cases, and to work through each of them.

17.7 **Baby Names:** Each year, the government releases a list of the 10,000 most common baby names and their frequencies (the number of babies with that name). The only problem with this is that some names have multiple spellings. For example, "John" and "Jon" are essentially the same name but would be listed separately in the list. Given two lists, one of names/frequencies and the other of pairs of equivalent names, write an algorithm to print a new list of the true frequency of each name. Note that if John and Jon are synonyms, and Jon and Johnny are synonyms, then John and Johnny are synonyms. (It is both transitive and symmetric.) In the final list, any name can be used as the "real" name.

EXAMPLE

Input:

Names: John (15), Jon (12), Chris (13), Kris (4), Christopher (19)

Synonyms: (Jon, John), (John, Johnny), (Chris, Kris), (Chris, Christopher)

Output: John (27), Kris (36)

pg 187

SOLUTION

Let's start off with a good example. We want an example with some names with multiple synonyms and some with none. Additionally, we want the synonym list to be diverse in which name is on the left side and which is on the right. For example, we wouldn't want Johnny to always be the name on the left side as we're creating the group of (John, Jonathan, Jon, and Johnny).

This list should work fairly well.

Name	Count
John	10
Jon	3
Davis	2
Kari	3
Johnny	11
Carlton	8
Carleton	2
Jonathan	9
Carrie	5

Name	Alternate
Jonathan	John
Jon	Johnny
Johnny	John
Kari	Carrie
Carleton	Carlton

The final list should be something like: John (33), Kari (8), Davis(2), Carleton (10).

Solution #1

Let's assume our baby names list is given to us as a hash table. (If not, it's easy enough to build one.)

We can start reading pairs in from the synonyms list. As we read the pair (Jonathan, John), we can merge the counts for Jonathan and John together. We'll need to remember, though, that we saw this pair, because, in the future, we could discover that Jonathan is equivalent to something else.

We can use a hash table (`L1`) that maps from a name to its "true" name. We'll also need to know, given a "true" name, all the names equivalent to it. This will be stored in a hash table `L2`. Note that `L2` acts as a reverse lookup of `L1`.

```
READ (Jonathan, John)
```

```
        L1.ADD Jonathan -> John
        L2.ADD John -> Jonathan
    READ (Jon, Johnny)
        L1.ADD Jon -> Johnny
        L2.ADD Johnny -> Jon
    READ (Johnny, John)
        L1.ADD Johnny -> John
        L1.UPDATE Jon -> John
        L2.UPDATE John -> Jonathan, Johnny, Jon
```

If we later find that John is equivalent to, say, Jonny, we'll need to look up the names in `L1` and `L2` and merge together all the names that are equivalent to them.

This will work, but it's unnecessarily complicated to keep track of these two lists.

Instead, we can think of these names as "equivalence classes." When we find a pair (Jonathan, John), we put these in the same set (or equivalence classes). Each name maps to its equivalence class. All items in the set map to the same instance of the set.

If we need to merge two sets, then we copy one set into the other and update the hash table to point to the new set.

```
    READ (Jonathan, John)
        CREATE Set1 = Jonathan, John
        L1.ADD Jonathan -> Set1
        L1.ADD John -> Set1
    READ (Jon, Johnny)
        CREATE Set2 = Jon, Johnny
        L1.ADD Jon -> Set2
        L1.ADD Johnny -> Set2
    READ (Johnny, John)
        COPY Set2 into Set1.
            Set1 = Jonathan, John, Jon, Johnny
        L1.UPDATE Jon -> Set1
        L1.UPDATE Johnny -> Set1
```

In the last step above, we iterated through all items in `Set2` and updated the reference to point to `Set1`. As we do this, we keep track of the total frequency of names.

```
HashMap<String, Integer> trulyMostPopular(HashMap<String, Integer> names,
                                          String[][] synonyms) {
  /* Parse list and initialize equivalence classes.*/
  HashMap<String, NameSet> groups = constructGroups(names);

  /* Merge equivalence classes together. */
  mergeClasses(groups, synonyms);

  /* Convert back to hash map. */
  return convertToMap(groups);
}

/* This is the core of the algorithm. Read through each pair. Merge their
 * equivalence classes and update the mapping of the secondary class to point to
 * the first set.*/
void mergeClasses(HashMap<String, NameSet> groups, String[][] synonyms) {
  for (String[] entry : synonyms) {
    String name1 = entry[0];
    String name2 = entry[1];
    NameSet set1 = groups.get(name1);
```

```
      NameSet set2 = groups.get(name2);
      if (set1 != set2) {
         /* Always merge the smaller set into the bigger one. */
         NameSet smaller = set2.size() < set1.size() ? set2 : set1;
         NameSet bigger = set2.size() < set1.size() ? set1 : set2;

         /* Merge lists */
         Set<String> otherNames = smaller.getNames();
         int frequency = smaller.getFrequency();
         bigger.copyNamesWithFrequency(otherNames, frequency);

         /* Update mapping */
         for (String name : otherNames) {
            groups.put(name,  bigger);
         }
      }
   }
}

/* Read through (name, frequency) pairs and initialize a mapping of names to
 * NameSets (equivalence classes).*/
HashMap<String, NameSet> constructGroups(HashMap<String, Integer> names) {
   HashMap<String, NameSet> groups = new HashMap<String, NameSet>();
   for (Entry<String, Integer> entry : names.entrySet()) {
      String name = entry.getKey();
      int frequency = entry.getValue();
      NameSet group = new NameSet(name, frequency);
      groups.put(name,  group);
   }
   return groups;
}

HashMap<String, Integer> convertToMap(HashMap<String, NameSet> groups) {
   HashMap<String, Integer> list = new HashMap<String, Integer>();
   for (NameSet group : groups.values()) {
      list.put(group.getRootName(), group.getFrequency());
   }
   return list;
}

public class NameSet {
   private Set<String> names = new HashSet<String>();
   private int frequency = 0;
   private String rootName;

   public NameSet(String name, int freq) {
      names.add(name);
      frequency = freq;
      rootName = name;
   }

   public void copyNamesWithFrequency(Set<String> more, int freq) {
      names.addAll(more);
      frequency += freq;
   }

```

```
    public Set<String> getNames() { return names; }
    public String getRootName() { return rootName; }
    public int getFrequency() { return frequency; }
    public int size() { return names.size(); }
}
```

The runtime of the algorithm is a bit tricky to figure out. One way to think about it is to think about what the worst case is.

For this algorithm, the worst case is where all names are equivalent—and we have to constantly merge sets together. Also, for the worst case, the merging should come in the worst possible way: repeated pairwise merging of sets. Each merging requires copying the set's elements into an existing set and updating the pointers from those items. It's slowest when the sets are larger.

If you notice the parallel with merge sort (where you have to merge single-element arrays into two-element arrays, and then two-element arrays into four-element arrays, until finally having a full array), you might guess it's `O(N log N)`. That is correct.

If you don't notice that parallel, here's another way to think about it.

Imagine we had the names (a, b, c, d, . . . , z). In our worst case, we'd first pair up the items into equivalence classes: (a, b), (c, d), (e, f), . . . , (y, z). Then, we'd merge pairs of those: (a, b, c, d), (e, f, g, h), . . . , (w, x, y, z). We'd continue doing this until we wind up with just one class.

At each "sweep" through the list where we merge sets together, half of the items get moved into a new set. This takes `O(N)` work per sweep. (There are fewer sets to merge, but each set has grown larger.)

How many sweeps do we do? At each sweep, we have half as many sets as we did before. Therefore, we do `O(log N)` sweeps.

Since we're doing `O(log N)` sweeps and `O(N)` work per sweep, the total runtime is `O(N log N)`.

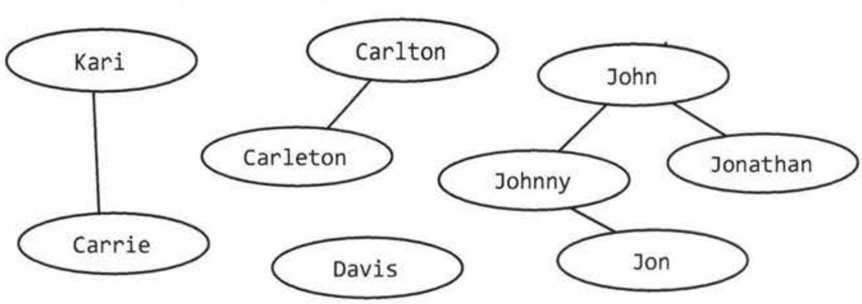
This is pretty good, but let's see if we can make it even faster.

Optimized Solution

To optimize the old solution, we should think about what exactly makes it slow. Essentially, it's the merging and updating of pointers.

So what if we just didn't do that? What if we marked that there was an equivalence relationship between two names, but didn't actually do anything with the information yet?

In this case, we'd be building essentially a graph.



Now what? Visually, it seems easy enough. Each component is an equivalent set of names. We just need to group the names by their component, sum up their frequencies, and return a list with one arbitrarily chosen name from each group.

In practice, how does this work? We could pick a name and do a depth-first (or breadth-first) search to sum the frequencies of all the names in one component. We would have to make sure that we hit each component exactly once. That's easy enough to achieve: mark a node as `visited` after it's discovered in the graph search, and only start the search for nodes where `visited` is false.

```
HashMap<String, Integer> trulyMostPopular(HashMap<String, Integer> names,
                                          String[][] synonyms) {
  /* Create data. */
  Graph graph = constructGraph(names);
  connectEdges(graph, synonyms);

  /* Find components. */
  HashMap<String, Integer> rootNames = getTrueFrequencies(graph);
  return rootNames;
}

/* Add all names to graph as nodes. */
Graph constructGraph(HashMap<String, Integer> names) {
  Graph graph = new Graph();
  for (Entry<String, Integer> entry : names.entrySet()) {
      String name = entry.getKey();
      int frequency = entry.getValue();
      graph.createNode(name, frequency);
  }
  return graph;
}

/* Connect synonymous spellings. */
void connectEdges(Graph graph, String[][] synonyms) {
  for (String[] entry : synonyms) {
      String name1 = entry[0];
      String name2 = entry[1];
      graph.addEdge(name1,  name2);
  }
}

/* Do DFS of each component. If a node has been visited before, then its component
 * has already been computed. */
HashMap<String, Integer> getTrueFrequencies(Graph graph) {
  HashMap<String, Integer> rootNames = new HashMap<String, Integer>();
  for (GraphNode node : graph.getNodes()) {
    if (!node.isVisited()) { // Already visited this component
      int frequency = getComponentFrequency(node);
      String name = node.getName();
      rootNames.put(name, frequency);
    }
  }
  return rootNames;
}

/* Do depth-first search to find the total frequency of this component, and mark
 * each node as visited.*/
int getComponentFrequency(GraphNode node) {
  if (node.isVisited()) return 0; // Already visited

  node.setIsVisited(true);
  int sum = node.getFrequency();
```

```
  for (GraphNode child : node.getNeighbors()) {
    sum += getComponentFrequency(child);
  }
  return sum;
}

/* Code for GraphNode and Graph is fairly self-explanatory, but can be found in
 * the downloadable code solutions.*/
```

To analyze the efficiency, we can think about the efficiency of each part of the algorithm.

- Reading in the data is linear with respect to the size of the data, so it takes O(B + P) time, where B is the number of baby names and P is the number of pairs of synonyms. This is because we only do a constant amount of work per piece of input data.
- To compute the frequencies, each edge gets "touched" exactly once across all of the graph searches and each node gets touched exactly once to check if it's been visited. The time of this part is O(B + P).

Therefore, the total time of the algorithm is O(B + P). We know we cannot do better than this since we must at least read in the B + P pieces of data.

17.8 Circus Tower: A circus is designing a tower routine consisting of people standing atop one another's shoulders. For practical and aesthetic reasons, each person must be both shorter and lighter than the person below him or her. Given the heights and weights of each person in the circus, write a method to compute the largest possible number of people in such a tower.

pg 187

SOLUTION

When we cut out all the "fluff" to this problem, we can understand that the problem is really the following.

We have a list of pairs of items. Find the longest sequence such that both the first and second items are in non-decreasing order.

One thing we might first try is sorting the items on an attribute. This is useful actually, but it won't get us all the way there.

By sorting the items by height, we have a relative order the items must appear in. We still need to find the longest increasing subsequence of weight though.

Solution 1: Recursive

One approach is to essentially try all possibilities. After sorting by height, we iterate through the array. At each element, we branch into two choices: add this element to the subsequence (if it's valid) or do not.

```
ArrayList<HtWt> longestIncreasingSeq(ArrayList<HtWt> items) {
  Collections.sort(items);
  return bestSeqAtIndex(items, new ArrayList<HtWt>(), 0);
}

ArrayList<HtWt> bestSeqAtIndex(ArrayList<HtWt> array, ArrayList<HtWt> sequence,
                               int index) {
  if (index >= array.size()) return sequence;

  HtWt value = array.get(index);

```

```
  ArrayList<HtWt> bestWith = null;
  if (canAppend(sequence, value)) {
    ArrayList<HtWt> sequenceWith = (ArrayList<HtWt>) sequence.clone();
    sequenceWith.add(value);
    bestWith = bestSeqAtIndex(array, sequenceWith, index + 1);
  }

  ArrayList<HtWt> bestWithout = bestSeqAtIndex(array, sequence, index + 1);

  if (bestWith == null || bestWithout.size() > bestWith.size()) {
    return bestWithout;
  } else {
    return bestWith;
  }
}

boolean canAppend(ArrayList<HtWt> solution, HtWt value) {
  if (solution == null) return false;
  if (solution.size() == 0) return true;

  HtWt last = solution.get(solution.size() - 1);
  return last.isBefore(value);
}

ArrayList<HtWt> max(ArrayList<HtWt> seq1, ArrayList<HtWt> seq2) {
  if (seq1 == null) {
    return seq2;
  } else if (seq2 == null) {
    return seq1;
  }
  return seq1.size() > seq2.size() ? seq1 : seq2;
}

public class HtWt implements Comparable<HtWt> {
  private int height;
  private int weight;
  public HtWt(int h, int w) { height = h; weight = w; }

  public int compareTo(HtWt second) {
    if (this.height != second.height) {
      return ((Integer)this.height).compareTo(second.height);
    } else {
      return ((Integer)this.weight).compareTo(second.weight);
    }
  }

  /* Returns true if "this" should be lined up before "other". Note that it's
   * possible that this.isBefore(other) and other.isBefore(this) are both false.
   * This is different from the compareTo method, where if a < b then b > a. */
  public boolean isBefore(HtWt other) {
    if (height < other.height && weight < other.weight) {
      return true;
    } else {
      return false;
    }
  }
```

```
}
```

This algorithm will take $O(2^n)$ time. We can optimize it using memoization (that is, caching the best sequences).

There's a cleaner way to do this though.

Solution #2: Iterative

Imagine we had the longest subsequence that terminates with each element, A[0] through A[3]. Could we use this to find the longest subsequence that terminates with A[4]?

```
Array: 13, 14, 10, 11, 12
Longest(ending with A[0]): 13
Longest(ending with A[1]): 13, 14
Longest(ending with A[2]): 10
Longest(ending with A[3]): 10, 11
Longest(ending with A[4]): 10, 11, 12
```

Sure. We just append A[4] on to the longest subsequence that it can be appended to.

This is now fairly straightforward to implement.

```
ArrayList<HtWt> longestIncreasingSeq(ArrayList<HtWt> array) {
  Collections.sort(array);

  ArrayList<ArrayList<HtWt>> solutions =  new ArrayList<ArrayList<HtWt>>();
  ArrayList<HtWt> bestSequence = null;

  /* Find the longest subsequence that terminates with each element. Track the
   * longest overall subsequence as we go. */
  for (int i = 0; i < array.size(); i++) {
    ArrayList<HtWt> longestAtIndex = bestSeqAtIndex(array, solutions, i);
    solutions.add(i, longestAtIndex);
    bestSequence = max(bestSequence, longestAtIndex);
  }

  return bestSequence;
}

/* Find the longest subsequence which terminates with this element. */
ArrayList<HtWt> bestSeqAtIndex(ArrayList<HtWt> array,
    ArrayList<ArrayList<HtWt>> solutions, int index) {
  HtWt value = array.get(index);

  ArrayList<HtWt> bestSequence = new ArrayList<HtWt>();

  /* Find the longest subsequence that we can append this element to. */
  for (int i = 0; i < index; i++) {
    ArrayList<HtWt> solution = solutions.get(i);
    if (canAppend(solution, value)) {
      bestSequence = max(solution, bestSequence);
    }
  }

  /* Append element. */
  ArrayList<HtWt> best = (ArrayList<HtWt>) bestSequence.clone();
  best.add(value);

```

```
  return best;
}
```

This algorithm operates in $O(n^2)$ time. An `O(n log(n))` algorithm does exist, but it is considerably more complicated and it is highly unlikely that you would derive this in an interview—even with some help. However, if you are interested in exploring this solution, a quick internet search will turn up a number of explanations of this solution.

17.9 Kth Multiple: Design an algorithm to find the kth number such that the only prime factors are 3, 5, and 7. Note that 3, 5, and 7 do not have to be factors, but it should not have any other prime factors. For example, the first several multiples would be (in order) 1, 3, 5, 7, 9, 15, 21.

pg 187

SOLUTION

Let's first understand what this problem is asking for. It's asking for the kth smallest number that is in the form $3^a * 5^b * 7^c$. Let's start with a brute force way of finding this.

Brute Force

We know that biggest this kth number could be is $3^k * 5^k * 7^k$. So, the "stupid" way of doing this is to compute $3^a * 5^b * 7^c$ for all values of a, b, and c between 0 and k. We can throw them all into a list, sort the list, and then pick the kth smallest value.

```
int getKthMagicNumber(int k) {
  ArrayList<Integer> possibilities = allPossibleKFactors(k);
  Collections.sort(possibilities);
  return possibilities.get(k);
}

ArrayList<Integer> allPossibleKFactors(int k) {
  ArrayList<Integer> values = new ArrayList<Integer>();
  for (int a = 0; a <= k; a++) { // loop 3
    int powA = (int) Math.pow(3, a);
    for (int b = 0; b <= k; b++) { // loop 5
      int powB = (int) Math.pow(5, b);
      for (int c = 0; c <= k; c++) { // loop 7
        int powC = (int) Math.pow(7, c);
        int value = powA * powB * powC;

        /* Check for overflow. */
        if (value < 0 || powA == Integer.MAX_VALUE ||
            powB == Integer.MAX_VALUE ||
            powC == Integer.MAX_VALUE) {
          value = Integer.MAX_VALUE;
        }
        values.add(value);
      }
    }
  }
  return values;
}
```

What is the runtime of this approach? We have nested for loops, each of which runs for k iterations. The runtime of the `allPossibleKFactors` is $O(k^3)$. Then, we sort the k^3 results in $O(k^3 \log (k^3))$ time (which is equivalent to $O(k^3 \log k)$. This gives us a runtime of $O(k^3 \log k)$.

There are a number of optimizations you could make to this (and better ways of handling the integer overflow), but honestly this algorithm is fairly slow. We should instead focus on reworking the algorithm.

Improved

Let's picture what our results will look like.

1	-	$3^0 * 5^0 * 7^0$
3	3	$3^1 * 5^0 * 7^0$
5	5	$3^0 * 5^1 * 7^0$
7	7	$3^0 * 5^0 * 7^1$
9	3*3	$3^2 * 5^0 * 7^0$
15	3*5	$3^1 * 5^1 * 7^0$
21	3*7	$3^1 * 5^0 * 7^1$
25	5*5	$3^0 * 5^2 * 7^0$
27	3*9	$3^3 * 5^0 * 7^0$
35	5*7	$3^0 * 5^1 * 7^1$
45	5*9	$3^2 * 5^1 * 7^0$
49	7*7	$3^0 * 5^0 * 7^2$
63	3*21	$3^2 * 5^0 * 7^1$

The question is: what is the next value in the list? The next value will be one of these:

- 3 * (some previous number in list)
- 5 * (some previous number in list)
- 7 * (some previous number in list)

If this doesn't immediately jump out at you, think about it this way: whatever the next value (let's call it `nv`) is, divide it by 3. Will that number have already appeared? As long as `nv` has factors of 3 in it, yes. The same can be said for dividing it by 5 and 7.

So, we know A_k can be expressed as `(3, 5 or 7)` * (some value in $\{A_1, \ldots, A_{k-1}\}$). We also know that A_k is, by definition, the next number in the list. Therefore, A_k will be the smallest "new" number (a number that it's already in $\{A_1, \ldots, A_{k-1}\}$) that can be formed by multiplying each value in the list by 3, 5 or 7.

How would we find A_k? Well, we could actually multiply each number in the list by 3, 5, and 7 and find the smallest element that has not yet been added to our list. This solution is $O(k^2)$. Not bad, but I think we can do better.

Rather than A_k trying to "pull" from a previous element in the list (by multiplying all of them by 3, 5 and 7), we can think about each previous value in the list as "pushing" out three subsequent values in the list. That is, each number A_i will eventually be used later in the list in the following forms:

- $3 * A_i$
- $5 * A_i$
- $7 * A_i$

We can use this thought to plan in advance. Each time we add a number A_i to the list, we hold on to the values $3A_i$, $5A_i$, and $7A_i$ in some sort of temporary list. To generate A_{i+1}, we search through this temporary list to find the smallest value.

Our code looks like this:

```
int removeMin(Queue<Integer> q) {
  int min = q.peek();
  for (Integer v : q) {
    if (min > v) {
      min = v;
    }
  }
  while (q.contains(min)) {
    q.remove(min);
  }
  return min;
}

void addProducts(Queue<Integer> q, int v) {
  q.add(v * 3);
  q.add(v * 5);
  q.add(v * 7);
}

int getKthMagicNumber(int k) {
  if (k < 0) return 0;

  int val = 1;
  Queue<Integer> q = new LinkedList<Integer>();
  addProducts(q, 1);
  for (int i = 0; i < k; i++) {
    val = removeMin(q);
    addProducts(q, val);
  }
  return val;
}
```

This algorithm is certainly much, much better than our first algorithm, but it's still not quite perfect.

Optimal Algorithm

To generate a new element A_i, we are searching through a linked list where each element looks like one of:

- 3 * previous element
- 5 * previous element
- 7 * previous element

Where is there unnecessary work that we might be able to optimize out?

Let's imagine our list looks like:

$$q_6 = \{7A_1, 5A_2, 7A_2, 7A_3, 3A_4, 5A_4, 7A_4, 5A_5, 7A_5\}$$

When we search this list for the min, we check if $7A_1 < \text{min}$, and then later we check if $7A_5 < \text{min}$. That seems sort of silly, doesn't it? Since we know that $A_1 < A_5$, we should only need to check $7A_1$.

If we separated the list from the beginning by the constant factors, then we'd only need to check the first of the multiples of 3, 5 and 7. All subsequent elements would be bigger.

That is, our list above would look like:

```
Q36 = {3A4}
Q56 = {5A2, 5A4, 5A5}
Q76 = {7A1, 7A2, 7A3, 7A4, 7A5}
```

To get the min, we only need to look at the fronts of each queue:

```
y = min(Q3.head(), Q5.head(), Q7.head())
```

Once we compute `y`, we need to insert 3y into Q3, 5y into Q5, and 7y into Q7. But, we only want to insert these elements if they aren't already in another list.

Why might, for example, 3y already be somewhere in the holding queues? Well, if y was pulled from Q7, then that means that `y = 7x`, for some smaller x. If 7x is the smallest value, we must have already seen 3x. And what did we do when we saw 3x? We inserted `7 * 3x` into Q7. Note that `7 * 3x = 3 * 7x = 3y`.

To put this another way, if we pull an element from Q7, it will look like `7 * suffix`, and we know we have already handled `3 * suffix` and `5 * suffix`. In handling `3 * suffix`, we inserted `7 * 3 * suffix` into a Q7. And in handling `5 * suffix`, we know we inserted `7 * 5 * suffix` in Q7. The only value we haven't seen yet is 7 * 7 * suffix, so we just insert `7 * 7 * suffix` into Q7.

Let's walk through this with an example to make it really clear.

```
initialize:
        Q3 = 3
        Q5 = 5
        Q7 = 7
remove min = 3. insert 3*3 in Q3, 5*3 into Q5, 7*3 into Q7.
        Q3 = 3*3
        Q5 = 5, 5*3
        Q7 = 7, 7*3
remove min = 5. 3*5 is a dup, since we already did 5*3. insert 5*5 into Q5, 7*5
into Q7.
        Q3 = 3*3
        Q5 = 5*3, 5*5
        Q7 = 7, 7*3, 7*5.
remove min = 7. 3*7 and 5*7 are dups, since we already did 7*3 and 7*5. insert 7*7
into Q7.
        Q3 = 3*3
        Q5 = 5*3, 5*5
        Q7 = 7*3, 7*5, 7*7
remove min = 3*3 = 9. insert 3*3*3 in Q3, 3*3*5 into Q5, 3*3*7 into Q7.
        Q3 = 3*3*3
        Q5 = 5*3, 5*5, 5*3*3
        Q7 = 7*3, 7*5, 7*7, 7*3*3
remove min = 5*3 = 15. 3*(5*3) is a dup, since we already did 5*(3*3). insert
5*5*3 in Q5, 7*5*3 into Q7.
        Q3 = 3*3*3
        Q5 = 5*5, 5*3*3, 5*5*3
        Q7 = 7*3, 7*5, 7*7, 7*3*3, 7*5*3
remove min = 7*3 = 21. 3*(7*3) and 5*(7*3) are dups, since we already did 7*(3*3)
and 7*(5*3). insert 7*7*3 into Q7.
        Q3 = 3*3*3
        Q5 = 5*5, 5*3*3, 5*5*3
        Q7 = 7*5, 7*7, 7*3*3, 7*5*3, 7*7*3
```

Our pseudocode for this problem is as follows:

1. Initialize `array` and queues Q3, Q5, and Q7

2. Insert 1 into `array`.
3. Insert `1*3`, `1*5` and `1*7` into Q3, Q5, and Q7 respectively.
4. Let x be the minimum element in Q3, Q5, and Q7. Append x to `magic`.
5. If x was found in:

   Q3 -> append `x*3`, `x*5` and `x*7` to Q3, Q5, and Q7. Remove x from Q3.

   Q5 -> append `x*5` and `x*7` to Q5 and Q7. Remove x from Q5.

   Q7 -> only append `x*7` to Q7. Remove x from Q7.
6. Repeat steps 4 - 6 until we've found k elements.

The code below implements this algorithm.

```
int getKthMagicNumber(int k) {
   if (k < 0) {
      return 0;
   }
   int val = 0;
   Queue<Integer> queue3 = new LinkedList<Integer>();
   Queue<Integer> queue5 = new LinkedList<Integer>();
   Queue<Integer> queue7 = new LinkedList<Integer>();
   queue3.add(1);

   /* Include 0th through kth iteration */
   for (int i = 0; i <= k; i++) {
      int v3 = queue3.size() > 0 ? queue3.peek() : Integer.MAX_VALUE;
      int v5 = queue5.size() > 0 ? queue5.peek() : Integer.MAX_VALUE;
      int v7 = queue7.size() > 0 ? queue7.peek() : Integer.MAX_VALUE;
      val = Math.min(v3, Math.min(v5, v7));
      if (val == v3) { // enqueue into queue 3, 5 and 7
         queue3.remove();
         queue3.add(3 * val);
         queue5.add(5 * val);
      } else if (val == v5) { // enqueue into queue 5 and 7
         queue5.remove();
         queue5.add(5 * val);
      } else if (val == v7) { // enqueue into Q7
         queue7.remove();
      }
      queue7.add(7 * val); // Always enqueue into Q7
   }
   return val;
}
```

When you get this question, do your best to solve it—even though it's really difficult. You can start with a brute force approach (challenging, but not quite as tricky), and then you can start trying to optimize it. Or, try to find a pattern in the numbers.

Chances are that your interviewer will help you along when you get stuck. Whatever you do, don't give up! Think out loud, wonder out loud, and explain your thought process. Your interviewer will probably jump in to guide you.

Remember, perfection on this problem is not expected. Your performance is evaluated in comparison to other candidates. Everyone struggles on a tricky problem.

17.10 Majority Element: A majority element is an element that makes up more than half of the items in an array. Given a positive integers array, find the majority element. If there is no majority element, return -1. Do this in `O(N)` time and `O(1)` space.

Input: `1 2 5 9 5 9 5 5 5`

Output: `5`

pg 187

SOLUTION

Let's start off with an example:

`3 1 7 1 3 7 3 7 1 7 7`

One thing we can notice here is that if the majority element (in this case 7) appears less often in the beginning, it must appear much more often toward the end. That's a good observation to make.

This interview question specifically requires us to do this in `O(N)` time and `O(1)` space. Nonetheless, sometimes it can be useful to relax one of those requirements and develop an algorithm. Let's try relaxing the time requirement but staying firm on the `O(1)` space requirement.

Solution #1 (Slow)

One simple way to do this is to just iterate through the array and check each element for whether it's the majority element. This takes $O(N^2)$ time and `O(1)` space.

```
int findMajorityElement(int[] array) {
   for (int x : array) {
      if (validate(array, x)) {
         return x;
      }
   }
   return -1;
}

boolean validate(int[] array, int majority) {
   int count = 0;
   for (int n : array) {
      if (n == majority) {
         count++;
      }
   }

   return count > array.length / 2;
}
```

This does not fit the time requirements of the problem, but it is potentially a starting point. We can think about optimizing this.

Solution #2 (Optimal)

Let's think about what that algorithm did on a particular example. Is there anything we can get rid of?

3	1	7	1	1	7	7	3	7	7	7
0	1	2	3	4	5	6	7	8	9	10

In the very first validation pass, we select 3 and validate it as the majority element. Several elements later, we've still counted just one 3 and several non-3 elements. Do we need to continue checking for 3?

On one hand, yes. 3 could redeem itself and be the majority element, if there are a bunch of 3s later in the array.

On the other hand, not really. If 3 does redeem itself, then we'll encounter those 3s later on, in a subsequent validation step. We could terminate this `validate(3)` step.

That logic is fine for the first element, but what about the next one? We would immediately terminate `validate(1)`, `validate(7)`, and so on.

Since the logic was okay for the first element, what if we treated all subsequent elements like they're the first element of some new subarray? This would mean that we start `validate(array[1])` at index 1, `validate(array[2])` at index 2, and so on.

What would this look like?

```
validate(3)
    sees 3 -> countYes = 1, countNo = 0
    sees 1 -> countYes = 1, countNo = 1
    TERMINATE. 3 is not majority thus far.
validate(1)
    sees 1 -> countYes = 0, countNo = 0
    sees 7 -> countYes = 1, countNo = 1
    TERMINATE. 1 is not majority thus far.
validate(7)
    sees 7 -> countYes = 1, countNo = 0
    sees 1 -> countYes = 1, countNo = 1
    TERMINATE. 7 is not majority thus far.
validate(1)
    sees 1 -> countYes = 1, countNo = 0
    sees 1 -> countYes = 2, countNo = 0
    sees 7 -> countYes = 2, countNo = 1
    sees 7 -> countYes = 2, countNo = 1
    TERMINATE. 1 is not majority thus far.
validate(1)
    sees 1 -> countYes = 1, countNo = 0
    sees 7 -> countYes = 1, countNo = 1
    TERMINATE. 1 is not majority thus far.
validate(7)
    sees 7 -> countYes = 1, countNo = 0
    sees 7 -> countYes = 2, countNo = 0
    sees 3 -> countYes = 2, countNo = 1
    sees 7 -> countYes = 3, countNo = 1
    sees 7 -> countYes = 4, countNo = 1
    sees 7 -> countYes = 5, countNo = 1
```

Do we know at this point that 7 is the majority element? Not necessarily. We have eliminated everything before that 7, and everything after it. But there could be no majority element. A quick `validate(7)` pass that starts from the beginning can confirm if 7 is actually the majority element. This `validate` step will be `O(N)` time, which is also our Best Conceivable Runtime. Therefore, this final `validate` step won't impact our total runtime.

This is pretty good, but let's see if we can make this a bit faster. We should notice that some elements are being "inspected" repeatedly. Can we get rid of this?

Look at the first `validate(3)`. This fails after the subarray `[3, 1]`, because 3 was not the majority element. But because `validate` fails the instant an element is not the majority element, it also means nothing else in that subarray was the majority element. By our earlier logic, we don't need to call `validate(1)`. We know that 1 did not appear more than half the time. If it is the majority element, it'll pop up later.

Let's try this again and see if it works out.

```
validate(3)
    sees 3 -> countYes = 1, countNo = 0
    sees 1 -> countYes = 1, countNo = 1
    TERMINATE. 3 is not majority thus far.
skip 1
validate(7)
    sees 7 -> countYes = 1, countNo = 0
    sees 1 -> countYes = 1, countNo = 1
    TERMINATE. 7 is not majority thus far.
skip 1
validate(1)
    sees 1 -> countYes = 1, countNo = 0
    sees 7 -> countYes = 1, countNo = 1
    TERMINATE. 1 is not majority thus far.
skip 7
validate(7)
    sees 7 -> countYes = 1, countNo = 0
    sees 3 -> countYes = 1, countNo = 1
    TERMINATE. 7 is not majority thus far.
skip 3
validate(7)
    sees 7 -> countYes = 1, countNo = 0
    sees 7 -> countYes = 2, countNo = 0
    sees 7 -> countYes = 3, countNo = 0
```

Good! We got the right answer. But did we just get lucky?

We should pause for a moment to think what this algorithm is doing.

1. We start off with `[3]` and we expand the subarray until 3 is no longer the majority element. We fail at `[3, 1]`. At the moment we fail, the subarray can have no majority element.
2. Then we go to `[7]` and expand until `[7, 1]`. Again, we terminate and nothing could be the majority element in that subarray.
3. We move to [1] and expand to `[1, 7]`. We terminate. Nothing there could be the majority element.
4. We go to `[7]` and expand to `[7, 3]`. We terminate. Nothing there could be the majority element.
5. We go to `[7]` and expand until the end of the array: `[7, 7, 7]`. We have found the majority element (and now we must validate that).

Each time we terminate the `validate` step, the subarray has no majority element. This means that there are at least as many non-7s as there are 7s. Although we're essentially removing this subarray from the original array, the majority element will still be found in the rest of the array—and will still have majority status. Therefore, at some point, we will discover the majority element.

Our algorithm can now be run in two passes: one to find the possible majority element and another to validate it. Rather than using two variables to count (`countYes` and `countNo`), we'll just use a single `count` variable that increments and decrements.

```
int findMajorityElement(int[] array) {
  int candidate = getCandidate(array);
  return validate(array, candidate) ? candidate : -1;
}

int getCandidate(int[] array) {
  int majority = 0;
```

```
    int count = 0;
    for (int n : array) {
       if (count == 0) { // No majority element in previous set.
          majority = n;
       }
       if (n == majority) {
          count++;
       } else {
          count--;
       }
    }
    return majority;
 }

 boolean validate(int[] array, int majority) {
    int count = 0;
    for (int n : array) {
       if (n == majority) {
          count++;
       }
    }

    return count > array.length / 2;
 }
```

This algorithm runs in O(N) time and O(1) space.

17.11 Word Distance: You have a large text file containing words. Given any two words, find the shortest distance (in terms of number of words) between them in the file. If the operation will be repeated many times for the same file (but different pairs of words), can you optimize your solution?

pg 187

SOLUTION

We will assume for this question that it doesn't matter whether `word1` or `word2` appears first. This is a question you should ask your interviewer.

To solve this problem, we can traverse the file just once. We remember throughout our traversal where we've last seen `word1` and `word2`, storing the locations in `location1` and `location2`. If the current locations are better than our best known location, we update the best locations.

The code below implements this algorithm.

```
LocationPair findClosest(String[] words, String word1, String word2) {
   LocationPair best = new LocationPair(-1, -1);
   LocationPair current = new LocationPair(-1, -1);
   for (int i = 0; i < words.length; i++) {
      String word = words[i];
      if (word.equals(word1)) {
         current.location1 = i;
         best.updateWithMin(current);
      } else if (word.equals(word2)) {
         current.location2 = i;
         best.updateWithMin(current); // If shorter, update values
      }
   }
```

```
    return best;
}

public class LocationPair {
    public int location1, location2;
    public LocationPair(int first, int second) {
        setLocations(first, second);
    }

    public void setLocations(int first, int second) {
        this.location1 = first;
        this.location2 = second;
    }

    public void setLocations(LocationPair loc) {
        setLocations(loc.location1, loc.location2);
    }

    public int distance() {
        return Math.abs(location1 - location2);
    }

    public boolean isValid() {
        return location1 >= 0 && location2 >= 0;
    }

    public void updateWithMin(LocationPair loc) {
        if (!isValid() || loc.distance() < distance()) {
            setLocations(loc);
        }
    }
}
```

If we need to repeat the operation for other pairs of words, we can create a hash table that maps from each word to the locations where it occurs. We'll only need to read through the list of words once. After that point, we can do a very similar algorithm but just iterate through the locations directly.

Consider the following lists of locations.

```
listA: {1, 2, 9, 15, 25}
listB: {4, 10, 19}
```

Picture pointers pA and pB that point to the beginning of each list. Our goal is to make pA and pB point to values as close together as possible.

The first potential pair is `(1, 4)`.

What is the next pair we can find? If we moved pB, then the distance would definitely get larger. If we moved pA, though, we might get a better pair. Let's do that.

The second potential pair is `(2, 4)`. This is better than the previous pair, so let's record this as the best pair.

We move pA again and get `(9, 4)`. This is worse than we had before.

Now, since the value at pA is bigger than the one at pB, we move pB. We get `(9, 10)`.

Next we get `(15, 10)`, then `(15, 19)`, then `(25, 19)`.

We can implement this algorithm as shown below.

```
LocationPair findClosest(String word1, String word2,
```

```
                             HashMapList<String, Integer> locations) {
    ArrayList<Integer> locations1 = locations.get(word1);
    ArrayList<Integer> locations2 = locations.get(word2);
    return findMinDistancePair(locations1, locations2);
}

LocationPair findMinDistancePair(ArrayList<Integer> array1,
                                 ArrayList<Integer> array2) {
    if (array1 == null || array2 == null || array1.size() == 0 ||
        array2.size() == 0) {
        return null;
    }

    int index1 = 0;
    int index2 = 0;
    LocationPair best = new LocationPair(array1.get(0), array2.get(0));
    LocationPair current = new LocationPair(array1.get(0), array2.get(0));

    while (index1 < array1.size() && index2 < array2.size()) {
        current.setLocations(array1.get(index1), array2.get(index2));
        best.updateWithMin(current); // If shorter, update values
        if (current.location1 < current.location2) {
            index1++;
        } else {
            index2++;
        }
    }

    return best;
}

/* Precomputation. */
HashMapList<String, Integer> getWordLocations(String[] words) {
    HashMapList<String, Integer> locations = new HashMapList<String, Integer>();
    for (int i = 0; i < words.length; i++) {
        locations.put(words[i], i);
    }
    return locations;
}

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

The precomputation step of this algorithm will take O(N) time, where N is the number of words in the string.

Finding the closest pair of locations will take O(A + B) time, where A is the number of occurrences of the first word and B is the number of occurrences of the second word.

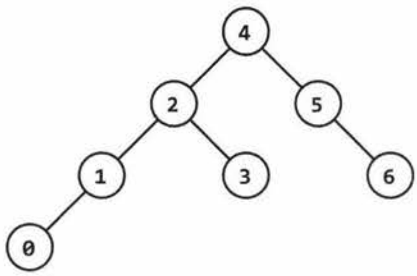
17.12 BiNode: Consider a simple data structure called BiNode, which has pointers to two other nodes. The data structure `BiNode` could be used to represent both a binary tree (where `node1` is the left node and `node2` is the right node) or a doubly linked list (where `node1` is the previous node and `node2` is the next node). Implement a method to convert a binary search tree (implemented with `BiNode`) into a doubly linked list. The values should be kept in order and the operation should be performed in place (that is, on the original data structure).

pg 188

SOLUTION

This seemingly complex problem can be implemented quite elegantly using recursion. You will need to understand recursion very well to solve it.

Picture a simple binary search tree:



The `convert` method should transform it into the below doubly linked list:

```
0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6
```

Let's approach this recursively, starting with the root (node 4).

We know that the left and right halves of the tree form their own "sub-parts" of the linked list (that is, they appear consecutively in the linked list). So, if we recursively converted the left and right subtrees to a doubly linked list, could we build the final linked list from those parts?

Yes! We would simply merge the different parts.

The pseudocode looks something like:

```
BiNode convert(BiNode node) {
   BiNode left = convert(node.left);
   BiNode right = convert(node.right);
   mergeLists(left, node, right);
   return left; // front of left
}
```

To actually implement the nitty-gritty details of this, we'll need to get the head and tail of each linked list. We can do this several different ways.

Solution #1: Additional Data Structure

The first, and easier, approach is to create a new data structure called `NodePair` which holds just the head and tail of a linked list. The `convert` method can then return something of type `NodePair`.

The code below implements this approach.

```
private class NodePair {
```

```
    BiNode head, tail;

    public NodePair(BiNode head, BiNode tail)  {
       this.head = head;
       this.tail = tail;
    }
 }

 public NodePair convert(BiNode root) {
    if (root == null) return null;

    NodePair part1 = convert(root.node1);
    NodePair part2 = convert(root.node2);

    if (part1 != null) {
       concat(part1.tail, root);
    }

    if (part2 != null) {
       concat(root, part2.head);
    }

    return new NodePair(part1 == null ? root : part1.head,
                        part2 == null ? root : part2.tail);
 }

 public static void concat(BiNode x, BiNode y) {
    x.node2 = y;
    y.node1 = x;
 }
```

The above code still converts the `BiNode` data structure in place. We're just using `NodePair` as a way to return additional data. We could have alternatively used a two-element `BiNode` array to fulfill the same purposes, but it looks a bit messier (and we like clean code, especially in an interview).

It'd be nice, though, if we could do this without these extra data structures—and we can.

Solution #2: Retrieving the Tail

Instead of returning the head and tail of the linked list with `NodePair`, we can return just the head, and then we can use the head to find the tail of the linked list.

```
BiNode convert(BiNode root) {
   if (root == null) return null;

   BiNode part1 = convert(root.node1);
   BiNode part2 = convert(root.node2);

   if (part1 != null) {
      concat(getTail(part1), root);
   }

   if (part2 != null) {
      concat(root, part2);
   }

   return part1 == null ? root : part1;
```

```
}

public static BiNode getTail(BiNode node) {
   if (node == null) return null;
   while (node.node2 != null) {
      node = node.node2;
   }
   return node;
}
```

Other than a call to `getTail`, this code is almost identical to the first solution. It is not, however, very efficient. A leaf node at depth d will be "touched" by the `getTail` method d times (one for each node above it), leading to an $O(N^2)$ overall runtime, where N is the number of nodes in the tree.

Solution #3: Building a Circular Linked List

We can build our third and final approach off of the second one.

This approach requires returning the head and tail of the linked list with `BiNode`. We can do this by returning each list as the head of a *circular* linked list. To get the tail, then, we simply call `head.node1`.

```
BiNode convertToCircular(BiNode root) {
   if (root == null) return null;

   BiNode part1 = convertToCircular(root.node1);
   BiNode part3 = convertToCircular(root.node2);

   if (part1 == null && part3 == null) {
      root.node1 = root;
      root.node2 = root;
      return root;
   }
   BiNode tail3 = (part3 == null) ? null : part3.node1;

   /* join left to root */
   if (part1 == null) {
      concat(part3.node1, root);
   } else {
      concat(part1.node1, root);
   }

   /* join right to root */
   if (part3 == null) {
      concat(root, part1);
   } else {
      concat(root, part3);
   }

   /* join right to left */
   if (part1 != null && part3 != null) {
      concat(tail3, part1);
   }

   return part1 == null ? root : part1;
}

/* Convert list to a circular linked list, then break the circular connection. */
```

```
BiNode convert(BiNode root) {
   BiNode head = convertToCircular(root);
   head.node1.node2 = null;
   head.node1 = null;
   return head;
}
```

Observe that we have moved the main parts of the code into `convertToCircular`. The `convert` method calls this method to get the head of the circular linked list, and then breaks the circular connection.

The approach takes `O(N)` time, since each node is only touched an average of once (or, more accurately, `O(1)` times).

17.13 Re-Space: Oh, no! You have accidentally removed all spaces, punctuation, and capitalization in a lengthy document. A sentence like "`I reset the computer. It still didn't boot!`" became "`iresetthecomputeritstilldidntboot`". You'll deal with the punctuation and capitalization later; right now you need to re-insert the spaces. Most of the words are in a dictionary but a few are not. Given a dictionary (a list of strings) and the document (a string), design an algorithm to unconcatenate the document in a way that minimizes the number of unrecognized characters.

EXAMPLE

Input: `jesslookedjustliketimherbrother`

Output: `jess looked just like tim her brother` (7 unrecognized characters)

pg 188

SOLUTION

Some interviewers like to cut to the chase and give you the specific problems. Others, though, like to give you a lot of unnecessary context, like this problem has. It's useful in such cases to boil down the problem to what it's really all about.

In this case, the problem is really about finding a way to break up a string into separate words such that as few characters as possible are "left out" of the parsing.

Note that we do not attempt to "understand" the string. We could just as well parse "`thisisawesome`" to be "`this is a we some`" as we could "`this is awesome`."

Brute Force

The key to this problem is finding a way to define the solution (that is, parsed string) in terms of its subproblems. One way to do this is recursing through the string.

The very first choice we make is where to insert the first space. After the first character? Second character? Third character?

Let's imagine this in terms of a string like `thisismikesfavoritefood`. What is the first space we insert?

- If we insert a space after `t`, this gives us one invalid character.
- After `th` is two invalid characters.
- After `thi` is three invalid characters.
- At `this` we have a complete word. This is zero invalid characters.
- At `thisi` is five invalid characters.
- ... and so on.

After we choose the first space, we can recursively pick the second space, then the third space, and so on, until we are done with the string.

We take the best (fewest invalid characters) out of all these choices and return.

What should the function return? We need both the number of invalid characters in the recursive path as well as the actual parsing. Therefore, we just return both by using a custom-built `ParseResult` class.

```
1  String bestSplit(HashSet<String> dictionary, String sentence) {
2    ParseResult r = split(dictionary, sentence, 0);
3    return r == null ? null : r.parsed;
4  }
5
6  ParseResult split(HashSet<String> dictionary, String sentence, int start) {
7    if (start >= sentence.length()) {
8      return new ParseResult(0, “”);
9    }
10
11   int bestInvalid = Integer.MAX_VALUE;
12   String bestParsing = null;
13   String partial = “”;
14   int index = start;
15   while (index < sentence.length()) {
16     char c = sentence.charAt(index);
17     partial += c;
18     int invalid = dictionary.contains(partial) ? 0 : partial.length();
19     if (invalid < bestInvalid) { // Short circuit
20       /* Recurse, putting a space after this character. If this is better than
21        * the current best option, replace the best option. */
22       ParseResult result = split(dictionary, sentence, index + 1);
23       if (invalid + result.invalid < bestInvalid) {
24         bestInvalid = invalid + result.invalid;
25         bestParsing = partial + “ ” + result.parsed;
26         if (bestInvalid == 0) break; // Short circuit
27       }
28     }
29
30     index++;
31   }
32   return new ParseResult(bestInvalid, bestParsing);
33 }
34
35 public class ParseResult {
36   public int invalid = Integer.MAX_VALUE;
37   public String parsed = “ ”;
38   public ParseResult(int inv, String p) {
39     invalid = inv;
40     parsed = p;
41   }
42 }
```

We've applied two short circuits here.

- Line 22: If the number of current invalid characters exceeds the best known one, then we know this recursive path will not be ideal. There's no point in even taking it.
- Line 30: If we have a path with zero invalid characters, then we know we can't do better than this. We might as well accept this path.

What's the runtime of this? It's difficult to truly describe in practice as it depends on the (English) language.

One way of looking at it is to imagine a bizarre language where essentially all paths in the recursion are taken. In this case, we are making both choices at each character. If there are n characters, this is an $O(2^n)$ runtime.

Optimized

Commonly, when we have exponential runtimes for a recursive algorithm, we optimize them through memoization (that is, caching results). To do so, we need to find the common subproblems.

Where do recursive paths overlap? That is, where are the common subproblems?

Let's again imagine the string `thisismikesfavoritefood`. Again, imagine that everything is a valid word.

In this case, we attempt to insert the first space after `t` as well as after `th` (and many other choices). Think about what the next choice is.

```
split(thisismikesfavoritefood) ->
       t  + split(hisismikesfavoritefood)
    OR th + split(isismikesfavoritefood)
    OR ...

split(hisismikesfavoritefood) ->
       h + split(isismikesfavoritefood)
    OR ...

...
```

Adding a space after `t` and h leads to the same recursive path as inserting a space after `th`. There's no sense in computing `split(isismikesfavoritefood)` twice when it will lead to the same result.

We should instead cache the result. We do this using a hash table which maps from the current substring to the `ParseResult` object.

We don't actually need to make the current substring a key. The `start` index in the string sufficiently represents the substring. After all, if we were to use the substring, we'd really be using `sentence.substring(start, sentence.length)`. This hash table will map from a start index to the best parsing from that index to the end of the string.

And, since the start index is the key, we don't need a true hash table at all. We can just use an array of `ParseResult` objects. This will also serve the purpose of mapping from an index to an object.

The code is essentially identical to the earlier function, but now takes in a memo table (a cache). We look up when we first call the function and set it when we return.

```
String bestSplit(HashSet<String> dictionary, String sentence) {
  ParseResult[] memo = new ParseResult[sentence.length()];
  ParseResult r = split(dictionary, sentence, 0, memo);
  return r == null ? null : r.parsed;
}

ParseResult split(HashSet<String> dictionary, String sentence, int start,
                  ParseResult[] memo) {
  if (start >= sentence.length()) {
    return new ParseResult(0, "");
  } if (memo[start] != null) {
    return memo[start];
  }
```

```

    int bestInvalid = Integer.MAX_VALUE;
    String bestParsing = null;
    String partial = “”;
    int index = start;
    while (index < sentence.length()) {
       char c = sentence.charAt(index);
       partial += c;
       int invalid = dictionary.contains(partial) ? 0 : partial.length();
       if (invalid < bestInvalid) { // Short circuit
          /* Recurse, putting a space after this character. If this is better than
           * the current best option, replace the best option. */
          ParseResult result = split(dictionary, sentence, index + 1, memo);
          if (invalid + result.invalid < bestInvalid) {
             bestInvalid = invalid + result.invalid;
             bestParsing = partial + “ ” + result.parsed;
             if (bestInvalid == 0) break; // Short circuit
          }
       }

       index++;
    }
    memo[start] = new ParseResult(bestInvalid, bestParsing);
    return memo[start];
 }
```

Understanding the runtime of this is even trickier than in the prior solution. Again, let's imagine the truly bizarre case, where essentially everything looks like a valid word.

One way we can approach it is to realize that `split(i)` will only be computed once for each value of `i`. What happens when we call `split(i)`, assuming we've already called `split(i+1)` through `split(n - 1)`?

```
split(i) -> calls:
    split(i + 1)
    split(i + 2)
    split(i + 3)
    split(i + 4)
    ...
    split(n - 1)
```

Each of the recursive calls has already been computed, so they just return immediately. Doing `n - i` calls at `O(1)` time each takes `O(n - i)` time. This means that `split(i)` takes `O(i)` time at most.

We can now apply the same logic to `split(i - 1)`, `split(i - 2)`, and so on. If we make 1 call to compute `split(n - 1)`, 2 calls to compute `split(n - 2)`, 3 calls to compute `split(n - 3)`, ..., n calls to compute `split(0)`, how many calls total do we do? This is basically the sum of the numbers from 1 through n, which is $O(n^2)$.

Therefore, the runtime of this function is $O(n^2)$.

17.14 Smallest K: Design an algorithm to find the smallest K numbers in an array.

pg 188

SOLUTION

There are a number of ways to approach this problem. We will go through three of them: sorting, max heap, and selection rank.

Some of these algorithms require modifying the array. This is something you should discuss with your interviewer. Note, though, that even if modifying the original array is not acceptable, you can always clone the array and modify the clone instead. This will not impact the overall big O time of any algorithm.

Solution 1: Sorting

We can sort the elements in ascending order and then take the first million numbers from that.

```
int[] smallestK(int[] array, int k) {
  if (k <= 0 || k > array.length) {
    throw new IllegalArgumentException();
  }

  /* Sort array. */
  Arrays.sort(array);

  /* Copy first k elements. */
  int[] smallest = new int[k];
  for (int i = 0; i < k; i++) {
    smallest[i] = array[i];
  }
  return smallest;
}
```

The time complexity is `O(n log(n))`.

Solution 2: Max Heap

We can use a max heap to solve this problem. We first create a max heap (largest element at the top) for the first million numbers.

Then, we traverse through the list. On each element, if it's smaller than the root, we insert it into the heap and delete the largest element (which will be the root).

At the end of the traversal, we will have a heap containing the smallest one million numbers. This algorithm is `O(n log(m))`, where `m` is the number of values we are looking for.

```
int[] smallestK(int[] array, int k) {
  if (k <= 0 || k > array.length) {
    throw new IllegalArgumentException();
  }

  PriorityQueue<Integer> heap = getKMaxHeap(array, k);
  return heapToIntArray(heap);
}

/* Create max heap of smallest k elements. */
PriorityQueue<Integer> getKMaxHeap(int[] array, int k) {
  PriorityQueue<Integer> heap =
```

```
      new PriorityQueue<Integer>(k, new MaxHeapComparator());
  for (int a : array) {
    if (heap.size() < k) { // If space remaining
      heap.add(a);
    } else if (a < heap.peek()) { // If full and top is small
      heap.poll(); // remove highest
      heap.add(a); // insert new element
    }
  }
  return heap;
}

/* Convert heap to int array. */
int[] heapToIntArray(PriorityQueue<Integer> heap) {
  int[] array = new int[heap.size()];
  while (!heap.isEmpty()) {
    array[heap.size() - 1] = heap.poll();
  }
  return array;
}

class MaxHeapComparator implements Comparator<Integer> {
    public int compare(Integer x, Integer y) {
        return y - x;
    }
}
```

Java's uses the `PriorityQueue` class to offer heap-like functionality. By default, it operates as a min heap, with the smallest element on the top. To switch it to the biggest element on the top, we can pass in a different comparator.

Approach 3: Selection Rank Algorithm (if elements are unique)

Selection Rank is a well-known algorithm in computer science to find the `i`th smallest (or largest) element in an array in linear time.

If the elements are unique, you can find the `i`th smallest element in expected `O(n)` time. The basic algorithm operates like this:

1. Pick a random element in the array and use it as a "pivot." Partition elements around the pivot, keeping track of the number of elements on the left side of the partition.
2. If there are exactly `i` elements on the left, then you just return the biggest element on the left.
3. If the left side is bigger than `i`, repeat the algorithm on just the left part of the array.
4. If the left side is smaller than `i`, repeat the algorithm on the right, but look for the element with rank `i - leftSize`.

Once you have found the `i`th smallest element, you know that all elements smaller than this will be to the left of this (since you've partitioned the array accordingly). You can now just return the first `i` elements.

The code below implements this algorithm.

```
int[] smallestK(int[] array, int k) {
  if (k <= 0 || k > array.length) {
    throw new IllegalArgumentException();
  }

```

```
    int threshold = rank(array, k - 1);
    int[] smallest = new int[k];
    int count = 0;
    for (int a : array) {
       if (a <= threshold) {
          smallest[count] = a;
          count++;
       }
    }
    return smallest;
 }

 /* Get element with rank. */
 int rank(int[] array, int rank) {
    return rank(array, 0, array.length - 1, rank);
 }

 /* Get element with rank between left and right indices. */
 int rank(int[] array, int left, int right, int rank) {
    int pivot = array[randomIntInRange(left, right)];
    int leftEnd = partition(array, left, right, pivot);
    int leftSize = leftEnd - left + 1;
    if (rank == leftSize - 1) {
       return max(array, left, leftEnd);
    } else if (rank < leftSize) {
       return rank(array, left, leftEnd, rank);
    } else {
       return rank(array, leftEnd + 1, right, rank - leftSize);
    }
 }

 /* Partition array around pivot such that all elements <= pivot come before all
  * elements > pivot. */
 int partition(int[] array, int left, int right, int pivot) {
    while (left <= right) {
       if (array[left] > pivot) {
          /* Left is bigger than pivot. Swap it to the right side, where we know it
           * should be. */
          swap(array, left, right);
          right--;
       } else if (array[right] <= pivot) {
          /* Right is smaller than the pivot. Swap it to the left side, where we know
           * it should be. */
          swap(array, left, right);
          left++;
       } else {
          /* Left and right are in correct places. Expand both sides. */
          left++;
          right--;
       }
    }
    return left - 1;
 }

 /* Get random integer within range, inclusive. */
 int randomIntInRange(int min, int max) {
```

```
  Random rand = new Random();
  return rand.nextInt(max + 1 - min) + min;
}

/* Swap values at index i and j. */
void swap(int[] array, int i, int j) {
  int t = array[i];
  array[i] = array[j];
  array[j] = t;
}

/* Get largest element in array between left and right indices. */
int max(int[] array, int left, int right) {
  int max = Integer.MIN_VALUE;
  for (int i = left; i <= right; i++) {
    max = Math.max(array[i], max);
  }
  return max;
}
```

If the elements are not unique, we can tweak this algorithm slightly to accommodate this.

Approach 4: Selection Rank Algorithm (if elements are not unique)

The major change that needs to be made is to the `partition` function. When we partition the array around a pivot element, we now partition it into three chunks: less than `pivot`, equal to `pivot`, and greater than `pivot`.

This requires minor tweaks to `rank` as well. We now compare the size of left and middle partitions to `rank`.

```
class PartitionResult {
  int leftSize, middleSize;
  public PartitionResult(int left, int middle) {
    this.leftSize = left;
    this.middleSize = middle;
  }
}

int[] smallestK(int[] array, int k) {
  if (k <= 0 || k > array.length) {
    throw new IllegalArgumentException();
  }

  /* Get item with rank k - 1. */
  int threshold = rank(array, k - 1);

  /* Copy elements smaller than the threshold element. */
  int[] smallest = new int[k];
  int count = 0;
  for (int a : array) {
    if (a < threshold) {
      smallest[count] = a;
      count++;
    }
  }

  /* If there's still room left, this must be for elements equal to the threshold
```

```
   * element. Copy those in. */
  while (count < k) {
    smallest[count] = threshold;
    count++;
  }

  return smallest;
}

/* Find value with rank k in array. */
int rank(int[] array, int k) {
  if (k >= array.length) {
    throw new IllegalArgumentException();
  }
  return rank(array, k, 0, array.length - 1);
}

/* Find value with rank k in sub array between start and end. */
int rank(int[] array, int k, int start, int end) {
  /* Partition array around an arbitrary pivot. */
  int pivot = array[randomIntInRange(start, end)];
  PartitionResult partition = partition(array, start, end, pivot);
  int leftSize = partition.leftSize;
  int middleSize = partition.middleSize;

  /* Search portion of array. */
  if (k < leftSize) { // Rank k is on left half
    return rank(array, k, start, start + leftSize - 1);
  } else if (k < leftSize + middleSize) { // Rank k is in middle
    return pivot; // middle is all pivot values
  } else { // Rank k is on right
    return rank(array, k - leftSize - middleSize, start + leftSize + middleSize,
                end);
  }
}

/* Partition result into < pivot, equal to pivot -> bigger than pivot. */
PartitionResult partition(int[] array, int start, int end, int pivot) {
  int left = start; /* Stays at (right) edge of left side. */
  int right = end;  /* Stays at (left) edge of right side. */
  int middle = start; /* Stays at (right) edge of middle. */
  while (middle <= right) {
    if (array[middle] < pivot) {
      /* Middle is smaller than the pivot. Left is either smaller or equal to
       * the pivot. Either way, swap them. Then middle and left should move by
       * one. */
      swap(array, middle, left);
      middle++;
      left++;
    } else if (array[middle] > pivot) {
      /* Middle is bigger than the pivot. Right could have any value. Swap them,
       * then we know that the new right is bigger than the pivot. Move right by
       * one. */
      swap(array, middle, right);
      right--;
    } else if (array[middle] == pivot) {
```

```
            /* Middle is equal to the pivot. Move by one. */
            middle++;
        }
    }

    /* Return sizes of left and middle. */
    return new PartitionResult(left - start, right - left + 1);
}
```

Notice the change made to `smallestK` too. We can't simply copy all elements less than or equal to `threshold` into the array. Since we have duplicates, there could be many more than k elements that are less than or equal to `threshold`. (We also can't just say "okay, only copy k elements over." We could inadvertently fill up the array early on with "equal" elements, and not leave enough space for the smaller ones.)

The solution for this is fairly simple: only copy over the smaller elements first, then fill up the array with equal elements at the end.

17.15 Longest Word: Given a list of words, write a program to find the longest word made of other words in the list.

pg 188

SOLUTION

This problem seems complex, so let's simplify it. What if we just wanted to know the longest word made of *two* other words in the list?

We could solve this by iterating through the list, from the longest word to the shortest word. For each word, we would split it into all possible pairs and check if both the left and right side are contained in the list.

The pseudocode for this would look like the following:

```
String getLongestWord(String[] list) {
    String[] array = list.SortByLength();
    /* Create map for easy lookup */
    HashMap<String, Boolean> map = new HashMap<String, Boolean>;

    for (String str : array) {
        map.put(str, true);
    }

    for (String s : array) {
        // Divide into every possible pair
        for (int i = 1; i < s.length(); i++) {
            String left = s.substring(0, i);
            String right = s.substring(i);
            // Check if both sides are in the array
            if (map[left] == true && map[right] == true) {
                return s;
            }
        }
    }
    return str;
}
```

This works great for when we just want to know composites of two words. But what if a word could be formed by any number of other words?

In this case, we could apply a very similar approach, with one modification: rather than simply looking up if the right side is in the array, we would recursively see if we can build the right side from the other elements in the array.

The code below implements this algorithm:

```
String printLongestWord(String arr[]) {
  HashMap<String, Boolean> map = new HashMap<String, Boolean>();
  for (String str : arr) {
    map.put(str, true);
  }
  Arrays.sort(arr, new LengthComparator()); // Sort by length
  for (String s : arr) {
    if (canBuildWord(s, true, map)) {
      System.out.println(s);
      return s;
    }
  }
  return “”;
}

boolean canBuildWord(String str, boolean isOriginalWord,
                     HashMap<String, Boolean> map) {
  if (map.containsKey(str) && !isOriginalWord) {
    return map.get(str);
  }
  for (int i = 1; i < str.length(); i++) {
    String left = str.substring(0, i);
    String right = str.substring(i);
    if (map.containsKey(left) && map.get(left) == true &&
        canBuildWord(right, false, map)) {
      return true;
    }
  }
  map.put(str, false);
  return false;
}
```

Note that in this solution we have performed a small optimization. We use a dynamic programming/memoization approach to cache the results between calls. This way, if we repeatedly need to check if there's any way to build "`testingtester`," we'll only have to compute it once.

A boolean flag `isOriginalWord` is used to complete the above optimization. The method `canBuildWord` is called for the original word and for each substring, and its first step is to check the cache for a previously calculated result. However, for the original words, we have a problem: `map` is initialized to `true` for them, but we don't want to return `true` (since a word cannot be composed solely of itself). Therefore, for the original word, we simply bypass this check using the `isOriginalWord` flag.

17.16 The Masseuse: A popular masseuse receives a sequence of back-to-back appointment requests and is debating which ones to accept. She needs a 15-minute break between appointments and therefore she cannot accept any adjacent requests. Given a sequence of back-to-back appointment requests (all multiples of 15 minutes, none overlap, and none can be moved), find the optimal (highest total booked minutes) set the masseuse can honor. Return the number of minutes.

EXAMPLE

Input: `{30, 15, 60, 75, 45, 15, 15, 45}`

Output: `180 minutes ({30, 60, 45, 45})`.

pg 188

SOLUTION

Let's start with an example. We'll draw it visually to get a better feel for the problem. Each number indicates the number of minutes in the appointment.

r_0 = 75	r_1 = 105	r_2 = 120	r_3 = 75	r_4 = 90	r_5 = 135

Alternatively, we could have also divided all the values (including the break) by 15 minutes, to give us the array `{5, 7, 8, 5, 6, 9}`. This would be equivalent, but now we would want a 1-minute break.

The best set of appointments for this problem has 330 minutes total, formed with {r_0 = 75, r_2 = 120, r_5 = 135}. Note that we've intentionally chosen an example in which the best sequence of appointments was not formed through a strictly alternating sequence.

We should also recognize that choosing the longest appointment first (the "greedy" strategy) would not necessarily be optimal. For example, a sequence like `{45, 60, 45, 15}` would not have 60 in the optimal set.

Solution #1: Recursion

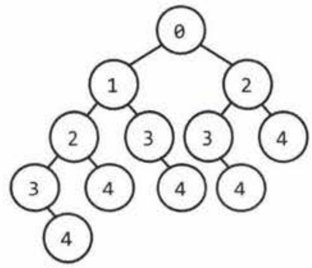
The first thing that may come to mind is a recursive solution. We have essentially a sequence of choices as we walk down the list of appointments: Do we use this appointment or do we not? If we use appointment `i`, we must skip appointment `i + 1` as we can't take back-to-back appointments. Appointment `i + 2` is a possibility (but not necessarily the best choice).

```
int maxMinutes(int[] massages) {
  return maxMinutes(massages, 0);
}

int maxMinutes(int[] massages, int index) {
  if (index >= massages.length) { // Out of bounds
    return 0;
  }

  /* Best with this reservation. */
  int bestWith = massages[index] + maxMinutes(massages, index + 2);

  /* Best without this reservation. */
  int bestWithout = maxMinutes(massages, index + 1);

  /* Return best of this subarray, starting from index. */
  return Math.max(bestWith, bestWithout);
}
```

The runtime of this solution is O(2^n) because at each element we're making two choices and we do this n times (where n is the number of massages).

The space complexity is O(n) due to the recursive call stack.

We can also depict this through a recursive call tree on an array of length 5. The number in each node represents the `index` value in a call to `maxMinutes`. Observe that, for example, `maxMinutes(massages, 0)` calls `maxMinutes(massages, 1)` and `maxMinutes(massages, 2)`.



As with many recursive problems, we should evaluate if there's a possibility to memoize repeated subproblems. Indeed, there is.

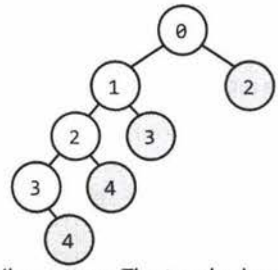
Solution #2: Recursion + Memoization

We will repeatedly call `maxMinutes` on the same inputs. For example, we'll call it on index 2 when we're deciding whether to take appointment 0. We'll also call it on index 2 when we're deciding whether to take appointment 1. We should memoize this.

Our memo table is just a mapping from `index` to the max minutes. Therefore, a simple array will suffice.

```
int maxMinutes(int[] massages) {
  int[] memo = new int[massages.length];
  return maxMinutes(massages, 0, memo);
}

int maxMinutes(int[] massages, int index, int[] memo) {
  if (index >= massages.length) {
    return 0;
  }

  if (memo[index] == 0) {
    int bestWith = massages[index] + maxMinutes(massages, index + 2, memo);
    int bestWithout = maxMinutes(massages, index + 1, memo);
    memo[index] = Math.max(bestWith, bestWithout);
  }

  return memo[index];
}
```

To determine the runtime, we'll draw the same recursive call tree as before but gray-out the calls that will return immediately. The calls that will never happen will be deleted entirely.



If we drew a bigger tree, we'd see a similar pattern. The tree looks very linear, with one branch down to the left. This gives us an `O(n)` runtime and `O(n)` space. The space usage comes from the recursive call stack as well as from the memo table.

Solution #3: Iterative

Can we do better? We certainly can't beat the time complexity since we have to look at each appointment. However, we might be able to beat the space complexity. This would mean not solving the problem recursively.

Let's look at our first example again.

r_0 = 30	r_1 = 15	r_2 = 60	r_3 = 75	r_4 = 45	r_5 = 15	r_6 = 15	r_7 = 45

As we noted in the problem statement, we cannot take adjacent appointments.

There's another observation, though, that we can make: We should never *skip* three consecutive appointments. That is, we might skip r_1 and r_2 if we wanted to take r_0 and r_3. But we would never skip r_1, r_2, and r_3. This would be suboptimal since we could always improve our set by grabbing that middle element.

This means that if we take r_0, we know we'll definitely skip r_1 and definitely take either r_2 or r_3. This substantially limits the options we need to evaluate and opens the door to an iterative solution.

Let's think about our recursive + memoization solution and try to reverse the logic; that is, let's try to approach it iteratively.

A useful way to do this is to approach it from the back and move toward the start of the array. At each point, we find the solution for the subarray.

- `best(7)`: What's the best option for {r_7 = 45}? We can get 45 min. if we take r_7, so `best(7) = 45`
- `best(6)`: What's the best option for {r_6 = 15, ...}? Still 45 min., so `best(6) = 45`.
- `best(5)`: What's the best option for {r_5 = 15, ...}? We can either:
  - take r_5 = 15 and merge it with `best(7) = 45`, or:
  - take `best(6) = 45`.

  The first gives us 60 minutes, `best(5) = 60`.
- `best(4)`: What's the best option for {r_4 = 45, ...}? We can either:
  - take r_4 = 45 and merge it with `best(6) = 45`, or:
  - take `best(5) = 60`.

  The first gives us 90 minutes, `best(4) = 90`.
- `best(3)`: What's the best option for {r_3 = 75, ...}? We can either:
  - take r_3 = 75 and merge it with `best(5) = 60`, or:

  - » take best(4) = 90.

  The first gives us 135 minutes, best(3) = 135.

- best(2): What's the best option for {r_2 = 60, ...}? We can either:
  - » take r_2 = 60 and merge it with best(4) = 90, or:
  - » take best(3) = 135.

  The first gives us 150 minutes, best(2) = 150.

- best(1): What's the best option for {r_1 = 15, ...}? We can either:
  - » take r_1 = 15 and merge it with best(3) = 135, or:
  - » take best(2) = 150.

  Either way, best(1) = 150.

- best(0): What's the best option for {r_0 = 30, ...}? We can either:
  - » take r_0 = 30 and merge it with best(2) = 150, or:
  - » take best(1) = 150.

  The first gives us 180 minutes, best(0) = 180.

Therefore, we return 180 minutes.

The code below implements this algorithm.

```
1  int maxMinutes(int[] massages) {
2    /* Allocating two extra slots in the array so we don't have to do bounds
3     * checking on lines 7 and 8. */
4    int[] memo = new int[massages.length + 2];
5    memo[massages.length] = 0;
6    memo[massages.length + 1] = 0;
7    for (int i = massages.length - 1; i >= 0; i--) {
8      int bestWith = massages[i] + memo[i + 2];
9      int bestWithout = memo[i + 1];
10     memo[i] = Math.max(bestWith, bestWithout);
11   }
12   return memo[0];
13 }
```

The runtime of this solution is O(n) and the space complexity is also O(n).

It's nice in some ways that it's iterative, but we haven't actually "won" anything here. The recursive solution had the same time and space complexity.

Solution #4: Iterative with Optimal Time and Space

In reviewing the last solution, we can recognize that we only use the values in the memo table for a short amount of time. Once we are several elements past an index, we never use that element's index again.

In fact, at any given index i, we only need to know the best value from i + 1 and i + 2. Therefore, we can get rid of the memo table and just use two integers.

```
int maxMinutes(int[] massages) {
  int oneAway = 0;
  int twoAway = 0;
  for (int i = massages.length - 1; i >= 0; i--) {
    int bestWith = massages[i] + twoAway;
    int bestWithout = oneAway;
```

```
      int current = Math.max(bestWith, bestWithout);
      twoAway = oneAway;
      oneAway = current;
    }
    return oneAway;
  }
```

This gives us the most optimal time and space possible: O(n) time and O(1) space.

Why did we look backward? It's a common technique in many problems to walk backward through an array.

However, we can walk forward if we want. This is easier for some people to think about, and harder for others. In this case, rather than asking "What's the best set that starts with a[i]?", we would ask "What's the best set that ends with a[i]?"

17.17 Multi Search: Given a string b and an array of smaller strings T, design a method to search b for each small string in T.

pg 189

SOLUTION

Let's start with an example:

```
T = {"is", "ppi", "hi", "sis", "i", "ssippi"}
b = "mississippi"
```

Note that in our example, we made sure to have some strings (like "is") that appear multiple times in b.

Solution #1

The naive solution is reasonably straightforward. Just search through the bigger string for each instance of the smaller string.

```
HashMapList<String, Integer> searchAll(String big, String[] smalls) {
  HashMapList<String, Integer> lookup =
    new HashMapList<String, Integer>();
  for (String small : smalls) {
    ArrayList<Integer> locations = search(big, small);
    lookup.put(small, locations);
  }
  return lookup;
}

/* Find all locations of the smaller string within the bigger string. */
ArrayList<Integer> search(String big, String small) {
  ArrayList<Integer> locations = new ArrayList<Integer>();
  for (int i = 0; i < big.length() - small.length() + 1; i++) {
    if (isSubstringAtLocation(big, small, i)) {
      locations.add(i);
    }
  }
  return locations;
}

/* Check if small appears at index offset within big. */
boolean isSubstringAtLocation(String big, String small, int offset) {
  for (int i = 0; i < small.length(); i++) {
    if (big.charAt(offset + i) != small.charAt(i)) {
```

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

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

We could have also used a `substring` and `equals` function, instead of writing `isAtLocation`. This is slightly faster (though not in terms of big O) because it doesn't require creating a bunch of substrings.

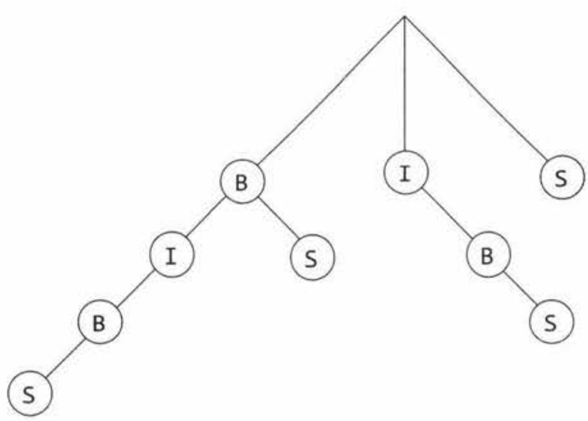
This will take `O(kbt)` time, where k is the length of the longest string in T, b is the length of the bigger string, and `t` is the number of smaller strings within T.

Solution #2

To optimize this, we should think about how we can tackle all the elements in T at once, or somehow re-use work.

One way is to create a trie-like data structure using each suffix in the bigger string. For the string `bibs`, the suffix list would be: `bibs`, `ibs`, `bs`, `s`.

The tree for this is below.



Then, all you need to do is search in the suffix tree for each string in T. Note that if "B" were a word, you would come up with two locations.

```
HashMapList<String, Integer> searchAll(String big, String[] smalls) {
    HashMapList<String, Integer> lookup = new HashMapList<String, Integer>();
    Trie tree = createTrieFromString(big);
    for (String s : smalls) {
        /* Get terminating location of each occurrence.*/
        ArrayList<Integer> locations = tree.search(s);

        /* Adjust to starting location. */
        subtractValue(locations, s.length());

        /* Insert. */
        lookup.put(s, locations);
    }
    return lookup;
```

```
}

Trie createTrieFromString(String s) {
  Trie trie = new Trie();
  for (int i = 0; i < s.length(); i++) {
    String suffix = s.substring(i);
    trie.insertString(suffix, i);
  }
  return trie;
}

void subtractValue(ArrayList<Integer> locations, int delta) {
  if (locations == null) return;
  for (int i = 0; i < locations.size(); i++) {
    locations.set(i, locations.get(i) - delta);
  }
}

public class Trie {
  private TrieNode root = new TrieNode();

  public Trie(String s) { insertString(s, 0); }
  public Trie() {}

  public ArrayList<Integer> search(String s) {
    return root.search(s);
  }

  public void insertString(String str, int location) {
    root.insertString(str, location);
  }

  public TrieNode getRoot() {
    return root;
  }
}

public class TrieNode {
  private HashMap<Character, TrieNode> children;
  private ArrayList<Integer> indexes;
  private char value;

  public TrieNode() {
    children = new HashMap<Character, TrieNode>();
    indexes = new ArrayList<Integer>();
  }

  public void insertString(String s, int index) {
    indexes.add(index);
    if (s != null && s.length() > 0) {
      value = s.charAt(0);
      TrieNode child = null;
      if (children.containsKey(value)) {
        child = children.get(value);
      } else {
        child = new TrieNode();
```

```
            children.put(value, child);
         }
         String remainder = s.substring(1);
         child.insertString(remainder, index + 1);
      } else {
         children.put('\0', null); // Terminating character
      }
   }

   public ArrayList<Integer> search(String s) {
      if (s == null || s.length() == 0) {
         return indexes;
      } else {
         char first = s.charAt(0);
         if (children.containsKey(first)) {
            String remainder = s.substring(1);
            return children.get(first).search(remainder);
         }
      }
      return null;
   }

   public boolean terminates() {
      return children.containsKey('\0');
   }

   public TrieNode getChild(char c) {
      return children.get(c);
   }
}

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

It takes $O(b^2)$ time to create the tree and `O(kt)` time to search for the locations.

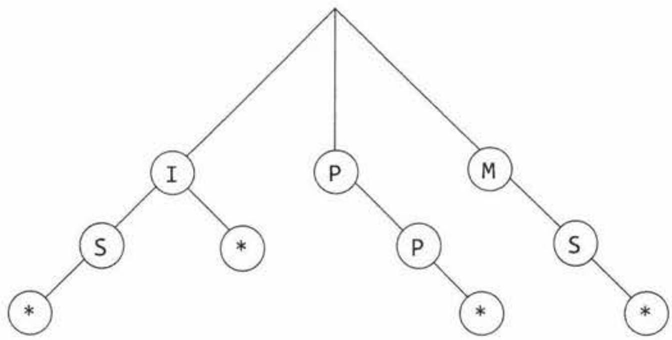
> Reminder: k is the length of the longest string in T, b is the length of the bigger string, and t is the number of smaller strings within T.

The total runtime is $O(b^2 + kt)$.

Without some additional knowledge of the expected input, you cannot directly compare `O(bkt)`, which was the runtime of the prior solution, to $O(b^2 + kt)$. If b is very large, then `O(bkt)` is preferable. But if you have a lot of smaller strings, then $O(b^2 + kt)$ might be better.

Solution #3

Alternatively, we can add all the smaller strings into a trie. For example, the strings `{i, is, pp, ms}` would look like the trie below. The asterisk (*) hanging from a node indicates that this node completes a word.



Now, when we want to find all words in mississippi, we search through this trie starting with each word.

- m: We would first look up in the trie starting with m, the first letter in mississippi. As soon as we go to mi, we terminate.
- i: Then, we go to i, the second character in mississippi. We see that i is a complete word, so we add it to the list. We also keep going with i over to is. The string is is also a complete word, so we add that to the list. This node has no more children, so we move onto the next character in mississippi.
- s: We now go to s. There is no upper-level node for s, so we go onto the next character.
- s: Another s. Go on to the next character.
- i: We see another i. We go to the i node in the trie. We see that i is a complete word, so we add it to the list. We also keep going with i over to is. The string is is also a complete word, so we add that to the list. This node has no more children, so we move onto the next character in mississippi.
- s: We go to s. There is no upper-level node for s.
- s: Another s. Go on to the next character.
- i: We go to the i node. We see that i is a complete word, so we add it to the trie. The next character in mississippi is a p. There is no node p, so we break here.
- p: We see a p. There is no node p.
- p: Another p.
- i: We go to the i node. We see that i is a complete word, so we add it to the trie. There are no more characters left in mississippi, so we are done.

Each time we find a complete "small" word, we add it to a list along with the location in the bigger word (mississippi) where we found the small word.

The code below implements this algorithm.

```
HashMapList<String, Integer> searchAll(String big, String[] smalls) {
  HashMapList<String, Integer> lookup = new HashMapList<String, Integer>();
  int maxLen = big.length();
  TrieNode root = createTreeFromStrings(smalls, maxLen).getRoot();

  for (int i = 0; i < big.length(); i++) {
    ArrayList<String> strings = findStringsAtLoc(root, big, i);
    insertIntoHashMap(strings, lookup, i);
  }

  return lookup;
```

```
}

/* Insert each string into trie (provided string is not longer than maxLen). */
Trie createTreeFromStrings(String[] smalls, int maxLen) {
  Trie tree = new Trie("");
  for (String s : smalls) {
    if (s.length() <= maxLen) {
      tree.insertString(s, 0);
    }
  }
  return tree;
}

/* Find strings in trie that start at index "start" within big. */
ArrayList<String> findStringsAtLoc(TrieNode root, String big, int start) {
  ArrayList<String> strings = new ArrayList<String>();
  int index = start;
  while (index < big.length()) {
    root = root.getChild(big.charAt(index));
    if (root == null) break;
    if (root.terminates()) { // Is complete string, add to list
      strings.add(big.substring(start, index + 1));
    }
    index++;
  }
  return strings;
}

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

This algorithm takes `O(kt)` time to create the trie and `O(bk)` time to search for all the strings.

> Reminder: k is the length of the longest string in T, b is the length of the bigger string, and `t` is the number of smaller strings within T.

The total time to solve the question is `O(kt + bk)`.

Solution #1 was `O(kbt)`. We know that `O(kt + bk)` will be faster than `O(kbt)`.

Solution #2 was $O(b^2 + kt)$. Since b will always be bigger than k (or if it's not, then we know this really long string k cannot be found in b), we know Solution #3 is also faster than Solution #2.

17.18 Shortest Supersequence: You are given two arrays, one shorter (with all distinct elements) and one longer. Find the shortest subarray in the longer array that contains all the elements in the shorter array. The items can appear in any order.

EXAMPLE

Input:

```
{1, 5, 9}
{7, 5, 9, 0, 2, 1, 3, 5, 7, 9, 1, 1, 5, 8, 8, 9, 7}
```

Output: `[7, 10] (the underlined portion above)`

pg 189

SOLUTIONS

As usual, a brute force approach is a good way to start. Try thinking about it as if you were doing it by hand. How would you do it?

Let's use the example from the problem to walk through this. We'll call the smaller array `smallArray` and the bigger array `bigArray`.

Brute Force

The slow, "easy" way to do this is to iterate through `bigArray` and do repeated small passes through it.

At each index in `bigArray`, scan forward to find the next occurrence of each element in `smallArray`. The largest of these next occurrences will tell us the shortest subarray that starts at that index. (We'll call this concept "closure." That is, the closure is the element that "closes" a complete subarray starting at that index. For example, the closure of index 3—which has value `0`—in the example is index 9.)

By finding the closures for each index in the array, we can find the shortest subarray overall.

```
Range shortestSupersequence(int[] bigArray, int[] smallArray) {
  int bestStart = -1;
  int bestEnd = -1;
  for (int i = 0; i < bigArray.length; i++) {
    int end = findClosure(bigArray, smallArray, i);
    if (end == -1) break;
    if (bestStart == -1 || end - i < bestEnd - bestStart) {
      bestStart = i;
      bestEnd = end;
    }
  }
  return new Range(bestStart, bestEnd);
}

/* Given an index, find the closure (i.e., the element which terminates a complete
 * subarray containing all elements in smallArray). This will be the max of the
 * next locations of each element in smallArray. */
int findClosure(int[] bigArray, int[] smallArray, int index) {
  int max = -1;
  for (int i = 0; i < smallArray.length; i++) {
    int next = findNextInstance(bigArray, smallArray[i], index);
    if (next == -1) {
      return -1;
    }
    max = Math.max(next, max);
  }
```

```
  return max;
}

/* Find next instance of element starting from index. */
int findNextInstance(int[] array, int element, int index) {
  for (int i = index; i < array.length; i++) {
    if (array[i] == element) {
      return i;
    }
  }
  return -1;
}

public class Range {
  private int start;
  private int end;
  public Range(int s, int e) {
    start = s;
    end = e;
  }

  public int length() { return end - start + 1; }
  public int getStart() { return start; }
  public int getEnd() { return end; }

  public boolean shorterThan(Range other) {
    return length() < other.length();
  }
}
```

This algorithm will potentially take `O(SB²)` time, where B is the length of `bigString` and S is the length of `smallString`. This is because at each of the B characters, we potentially do `O(SB)` work: S scans of the rest of the string, which has potentially B characters.

Optimized

Let's think about how we can optimize this. The core reason why it's slow is the repeated searches. Is there a faster way that we can find, given an index, the next occurrence of a particular character?

Let's think about it with an example. Given the array below, is there a way we could quickly find the next 5 from each location?

```
7, 5, 9, 0, 2, 1, 3, 5, 7, 9, 1, 1, 5, 8, 8, 9, 7
```

Yes. Because we're going to have to do this repeatedly, we can precompute this information in just a single (backwards) sweep. Iterate through the array backwards, tracking the last (most recent) occurrence of 5.

value	7	5	9	0	2	1	3	5	7	9	1	1	5	8	8	9	7
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
next 5	1	1	7	7	7	7	7	7	12	12	12	12	12	x	x	x	x

Doing this for each of `{1, 5, 9}` takes just 3 backwards sweeps.

Some people want to merge this into one backwards sweep that handles all three values. It feels faster—but it's not really. Doing it in one backwards sweep means doing three comparisons at each iteration. N moves through the list with three comparisons at each move is no better than 3N moves and one comparison at each move. You might as well keep the code clean by doing it in separate sweeps.

value	7	5	9	0	2	1	3	5	7	9	1	1	5	8	8	9	7
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
next 1	5	5	5	5	5	5	10	10	10	10	10	11	x	x	x	x	x
next 5	1	1	7	7	7	7	7	7	12	12	12	12	12	x	x	x	x
next 9	2	2	2	9	9	9	9	9	9	9	15	15	15	15	15	15	x

The `findNextInstance` function can now just use this table to find the next occurrence, rather than doing a search.

But, actually, we can make it a bit simpler. Using the table above, we can quickly compute the closure of each index. It's just the max of the column. If a column has an x in it, then there is no closure, at this indicates that there's no next occurrence of that character.

The difference between the index and the closure is the smallest subarray starting at that index.

value	7	5	9	0	2	1	3	5	7	9	1	1	5	8	8	9	7
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
next 1	5	5	5	5	5	5	10	10	10	10	10	11	x	x	x	x	x
next 5	1	1	7	7	7	7	7	7	12	12	12	12	12	x	x	x	x
next 9	2	2	2	9	9	9	9	9	9	9	15	15	15	15	15	15	x
closure	5	5	7	9	9	9	10	10	12	12	15	15	x	x	x	x	x
diff.	5	4	5	6	5	4	4	3	4	3	5	4	x	x	x	x	x

Now, all we have to do is to find the minimum distance in this table.

```
Range shortestSupersequence(int[] big, int[] small) {
  int[][] nextElements = getNextElementsMulti(big, small);
  int[] closures = getClosures(nextElements);
  return getShortestClosure(closures);
}

/* Create table of next occurrences. */
int[][] getNextElementsMulti(int[] big, int[] small) {
  int[][] nextElements = new int[small.length][big.length];
  for (int i = 0; i < small.length; i++) {
    nextElements[i] = getNextElement(big, small[i]);
  }
  return nextElements;
}

/* Do backwards sweep to get a list of the next occurrence of value from each
 * index. */
int[] getNextElement(int[] bigArray, int value) {
  int next = -1;
  int[] nexts = new int[bigArray.length];
  for (int i = bigArray.length - 1; i >= 0; i--) {
    if (bigArray[i] == value) {
      next = i;
    }
    nexts[i] = next;
  }
  return nexts;
}

/* Get closure for each index. */
```

```
int[] getClosures(int[][] nextElements) {
  int[] maxNextElement = new int[nextElements[0].length];
  for (int i = 0; i < nextElements[0].length; i++) {
    maxNextElement[i] = getClosureForIndex(nextElements, i);
  }
  return maxNextElement;
}

/* Given an index and the table of next elements, find the closure for this index
 * (which will be the min of this column). */
int getClosureForIndex(int[][] nextElements, int index) {
  int max = -1;
  for (int i = 0; i < nextElements.length; i++) {
    if (nextElements[i][index] == -1) {
      return -1;
    }
    max = Math.max(max, nextElements[i][index]);
  }
  return max;
}

/* Get shortest closure. */
Range getShortestClosure(int[] closures) {
  int bestStart = -1;
  int bestEnd = -1;
  for (int i = 0; i < closures.length; i++) {
    if (closures[i] == -1) {
      break;
    }
    int current = closures[i] - i;
    if (bestStart == -1 || current < bestEnd - bestStart) {
      bestStart = i;
      bestEnd = closures[i];
    }
  }
  return new Range(bestStart, bestEnd);
}
```

This algorithm will potentially take O(SB) time, where B is the length of `bigString` and S is the length of `smallString`. This is because we do S sweeps through the array to build up the next occurrences table and each sweep takes O(B) time.

It uses O(SB) space.

More Optimized

While our solution is fairly optimal, we can reduce the space usage. Remember the table we created:

value	7	5	9	0	2	1	3	5	7	9	1	1	5	8	8	9	7
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
next 1	5	5	5	5	5	5	10	10	10	10	10	11	x	x	x	x	x
next 5	1	1	7	7	7	7	7	7	12	12	12	12	12	x	x	x	x
next 9	2	2	2	9	9	9	9	9	9	9	15	15	15	15	15	15	x
closure	5	5	7	9	9	9	10	10	12	12	15	15	x	x	x	x	x

In actuality, all we need is the closure row, which is the minimum of all the other rows. We don't need to store all the other next occurrence information the entire time.

Instead, as we do each sweep, we just update the closure row with the minimums. The rest of the algorithm works essentially the same way.

```
Range shortestSupersequence(int[] big, int[] small) {
  int[] closures = getClosures(big, small);
  return getShortestClosure(closures);
}

/* Get closure for each index. */
int[] getClosures(int[] big, int[] small) {
  int[] closure = new int[big.length];
  for (int i = 0; i < small.length; i++) {
    sweepForClosure(big, closure, small[i]);
  }
  return closure;
}

/* Do backwards sweep and update the closures list with the next occurrence of
 * value, if it's later than the current closure. */
void sweepForClosure(int[] big, int[] closures, int value) {
  int next = -1;
  for (int i = big.length - 1; i >= 0; i--) {
    if (big[i] == value) {
      next = i;
    }
    if ((next == -1 || closures[i] < next) &&
        (closures[i] != -1)) {
      closures[i] = next;
    }
  }
}

/* Get shortest closure. */
Range getShortestClosure(int[] closures) {
  Range shortest = new Range(0, closures[0]);
  for (int i = 1; i < closures.length; i++) {
    if (closures[i] == -1) {
      break;
    }
    Range range = new Range(i, closures[i]);
    if (!shortest.shorterThan(range)) {
      shortest = range;
    }
  }
  return shortest;
}
```

This still runs in O(SB) time, but it now only takes O(B) additional memory.

Alternative & More Optimal Solution

There's a totally different way to approach it. Let's suppose we had a list of the occurrences of each element in `smallArray`.