```
      new HashMap<Integer, Queue<Integer>>();
   for (int s : small) {
      Queue<Integer> queue = new LinkedList<Integer>();
      itemLocations.put(s, queue);
   }

   / *Walk through big array, adding the item locations to hash map */
   for (int i = 0; i < big.length; i++) {
      Queue<Integer> queue = itemLocations.get(big[i]);
      if (queue != null) {
         queue.add(i);
      }
   }

   ArrayList<Queue<Integer>> allLocations = new ArrayList<Queue<Integer>>();
   allLocations.addAll(itemLocations.values());
   return allLocations;
}

Range getShortestClosure(ArrayList<Queue<Integer>> lists) {
   PriorityQueue<HeapNode> minHeap = new PriorityQueue<HeapNode>();
   int max = Integer.MIN_VALUE;

   / *Insert min element from each list. */
   for (int i = 0; i < lists.size(); i++) {
      int head = lists.get(i).remove();
      minHeap.add(new HeapNode(head, i));
      max = Math.max(max, head);
   }

   int min = minHeap.peek().locationWithinList;
   int bestRangeMin = min;
   int bestRangeMax = max;

   while (true) {
      / *Remove min node. */
      HeapNode n = minHeap.poll();
      Queue<Integer> list = lists.get(n.listId);

      / *Compare range to best range. */
      min = n.locationWithinList;
      if (max - min < bestRangeMax - bestRangeMin) {
         bestRangeMax = max;
         bestRangeMin = min;
      }

      / *If there are no more elements, then there's no more subsequences and we
       * can break. */
      if (list.size() == 0) {
         break;
      }

      / *Add new head of list to heap. */
      n.locationWithinList = list.remove();
      minHeap.add(n);
      max = Math.max(max, n.locationWithinList);
```

```
68      }
69
70      return new Range(bestRangeMin, bestRangeMax);
71  }
```

We're going through B elements in `getShortestClosure`, and each time pass in the for loop will take `O(log S)` time (the time to insert/remove from the heap). This algorithm will therefore take `O(B log S)` time in the worst case.

**17.19 Missing Two:** You are given an array with all the numbers from 1 to N appearing exactly once, except for one number that is missing. How can you find the missing number in `O(N)` time and `O(1)` space? What if there were two numbers missing?

*pg 189*

## SOLUTIONS

Let's start with the first part: find a missing number in `O(N)` time and `O(1)` space.

### Part 1: Find One Missing Number

We have a very constrained problem here. We can't store all the values (that would take `O(N)` space) and yet, somehow, we need to have a "record" of them such that we can identify the missing number.

This suggests that we need to do some sort of computation with the values. What characteristics does this computation need to have?

- **Unique.** If this computation gives the same result on two arrays (which fit the description in the problem), then those arrays must be equivalent (same missing number). That is, the result of the computation must uniquely correspond to the specific array and missing number.
- **Reversible.** We need some way of getting from the result of the calculation to the missing number.
- **Constant Time:** The calculation can be slow, but it must be constant time per element in the array.
- **Constant Space:** The calculation can require additional memory, but it must be `O(1)` memory.

The "unique" requirement is the most interesting—and the most challenging. What calculations can be performed on a set of numbers such that the missing number will be discoverable?

There are actually a number of possibilities.

We could do something with prime numbers. For example, for each value `x` in the array, we multiply `result` by the xth prime. We would then get some value that is indeed unique (since two different sets of primes can't have the same product).

Is this reversible? Yes. We could take `result` and divide it by each prime number: 2, 3, 5, 7, and so on. When we get a non-integer for the `i`th prime, then we know `i` was missing from our array.

Is it constant time and space, though? Only if we had a way of getting the `i`th prime number in `O(1)` time and `O(1)` space. We don't have that.

What other calculations could we do? We don't even need to do all this prime number stuff. Why not just multiply all the numbers together?

- **Unique?** Yes. Picture `1*2*3*...*n`. Now, imagine crossing off one number. This will give us a different result than if we crossed off any other number.
- **Constant time and space?** Yes.

- **Reversible?** Let's think about this. If we compare what our product is to what it would have been without a number removed, can we find the missing number? Sure. We just divide `full_product` by `actual_product`. This will tell us which number was missing from `actual_product`.

*There's just one issue: this product is really, really, really* big. *If* n *is* 20, the product *will* be somewhere around 2,000,000,000,000,000,000.

We can still approach it this way, but we'll need to use the `BigInteger` class.

```
1  int missingOne(int[] array) {
2    BigInteger fullProduct = productToN(array.length + 1);
3
4    BigInteger actualProduct = new BigInteger("1");
5    for (int i = 0; i < array.length; i++) {
6      BigInteger value = new BigInteger(array[i] + "");
7      actualProduct = actualProduct.multiply(value);
8    }
9
10   BigInteger missingNumber = fullProduct.divide(actualProduct);
11   return Integer.parseInt(missingNumber.toString());
12 }
13
14 BigInteger productToN(int n) {
15   BigInteger fullProduct = new BigInteger("1");
16   for (int i = 2; i <= n; i++) {
17     fullProduct = fullProduct.multiply(new BigInteger(i + ""));
18   }
19   return fullProduct;
20 }
```

There's no need for all of this, though. We can use the sum instead. It too will be unique.

Doing the sum has another benefit: there is already a closed form expression to compute the sum of numbers between 1 and n. This is $\frac{n(n+1)}{2}$.

> Most candidates probably won't remember the expression for the sum of numbers between 1 and n, and that's okay. Your interviewer might, however, ask you to derive it. Here's how to think about that: you can pair up the low and high values in the sequence of `0 + 1 + 2 + 3 + ... + n` to get: `(0, n) + (1, n-1) + (2, n-3)`, and so on. Each of those pairs has a sum of n and there are $\frac{n+1}{2}$ pairs. But what if n is even, such that $\frac{n+1}{2}$ is not an integer? In this case, pair up low and high values to get $n/2$ pairs with sum n+1. Either way, the math works out to $\frac{n(n+1)}{2}$.

Switching to a sum will delay the overflow issue substantially, but it won't wholly prevent it. You should discuss the issue with your interviewer to see how he/she would like you to handle it. Just mentioning it is plenty sufficient for many interviewers.

**Part 2: Find Two Missing Numbers**

This is substantially more difficult. Let's start with what our earlier approaches will tell us when we have two missing numbers.

- Sum: Using this approach will give us the sum of the two values that are missing.
- Product: Using this approach will give us the product of the two values that are missing.

Unfortunately, knowing the sum isn't enough. If, for example, the sum is 10, that could correspond to (1, 9), (2, 8), and a handful of other pairs. The same could be said for the product.

We're again at the same point we were in the first part of the problem. We need a calculation that can be applied such that the result is unique across all potential pairs of missing numbers.

Perhaps there is such a calculation (the prime one would work, but it's not constant time), but your interviewer probably doesn't expect you to know such math.

What else can we do? Let's go back to what we can do. We can get `x + y` and we can also get x * y. Each result leaves us with a number of possibilities. But using both of them narrows it down to the specific numbers.

```
x + y = sum     -> y = sum - x
x * y = product -> x(sum - x) = product
                  x*sum - x² = product
                  x*sum - x² - product = 0
                  -x² + x*sum - product = 0
```

At this point, we can apply the quadratic formula to solve for x. Once we have x, we can then compute y.

There are actually a number of other calculations you can perform. In fact, almost any other calculation (other than "linear" calculations) will give us values for x and y.

For this part, let's use a different calculation. Instead of using the product of `1 * 2 * ... * n`, we can use the sum of the squares: $1^2 + 2^2 + \ldots + n^2$. This will make the `BigInteger` usage a little less critical, as the code will at least run on small values of n. We can discuss with our interviewer whether or not this is important.

```
x + y = s       -> y = s - x
x² + y² = t     -> x² + (s-x)² = t
                   2x² - 2sx + s²-t = 0
```

Recall the quadratic formula:

```
x = [-b +- sqrt(b² - 4ac)] / 2a
```

where, in this case:

```
a = 2
b = -2s
c = s²-t
```

Implementing this is now somewhat straightforward.

```
1  int[] missingTwo(int[] array) {
2    int max_value = array.length + 2;
3    int rem_square = squareSumToN(max_value, 2);
4    int rem_one = max_value * (max_value + 1) / 2;
5
6    for (int i = 0; i < array.length; i++) {
7      rem_square -= array[i] * array[i];
8      rem_one -= array[i];
9    }
10
11   return solveEquation(rem_one, rem_square);
12 }
13
14 int squareSumToN(int n, int power) {
15   int sum = 0;
16   for (int i = 1; i <= n; i++) {
17     sum += (int) Math.pow(i, power);
18   }
19   return sum;
20 }
```

```
21
22 int[] solveEquation(int r1, int r2) {
23    /* ax^2 + bx + c
24     * -->
25     * x = [-b +- sqrt(b^2 - 4ac)] / 2a
26     * In this case, it has to be a + not a - */
27    int a = 2;
28    int b = -2 * r1;
29    int c = r1 * r1 - r2;
30
31    double part1 = -1 * b;
32    double part2 = Math.sqrt(b*b - 4 * a * c);
33    double part3 = 2 * a;
34
35    int solutionX = (int) ((part1 + part2) / part3);
36    int solutionY = r1 - solutionX;
37
38    int[] solution = {solutionX, solutionY};
39    return solution;
40 }
```

You might notice that the quadratic formula usually gives us two answers (see the + or - part), yet in our code, we only use the (+) result. We never checked the (-) answer. Why is that?

The existence of the "alternate" solution doesn't mean that one is the correct solution and one is "fake." It means that there are exactly two values for x which will correctly fulfill our equation: $2x^2 - 2sx + (s^2-t) = 0$.

That's true. There are. What's the other one? The other value is y!

If this doesn't immediately make sense to you, remember that x and y are interchangeable. Had we solved for y earlier instead of x, we would have wound up with an identical equation: $2y^2 - 2sy + (s^2-t) = 0$. So of course y could fulfill x's equation and x could fulfill y's equation. They have the exact same equation. Since x and y are both solutions to equations that look like $2[something]^2 - 2s[something] + s^2-t = 0$, then the other something that fulfills that equation must be y.

Still not convinced? Okay, we can do some math. Let's say we took the alternate value for x: $[-b - sqrt(b^2 - 4ac)] / 2a$. What's y?

$$\begin{aligned} x + y &= r_1 \\ y &= r_1 - x \\ &= r_1 - [-b - sqrt(b^2 - 4ac)]/2a \\ &= [2a*r_1 + b + sqrt(b^2 - 4ac)]/2a \end{aligned}$$

Partially plug in values for a and b, but keep the rest of the equation as-is:

$$\begin{aligned} &= [2(2)*r_1 + (-2r_1) + sqrt(b^2 - 4ac)]/2a \\ &= [2r_1 + sqrt(b^2 - 4ac)]/2a \end{aligned}$$

Recall that $b = -2r_1$. Now, we wind up with this equation:

$$= [-b + sqrt(b^2 - 4ac)]/2a$$

Therefore, if we use x = (part1 + part2) / part3, then we'll get (part1 - part2) / part3 for the value for y.

We don't care which one we call x and which one we call y, so we can use either one. It'll work out the same in the end.

**17.20 *Continuous Median:*** Numbers are randomly generated and passed to a method. Write a program to find and maintain the median value as new values are generated.

*pg 189*

## SOLUTIONS

One solution is to use two priority heaps: a max heap for the values below the median, and a min heap for the values above the median. This will divide the elements roughly in half, with the middle two elements as the top of the two heaps. This makes it trivial to find the median.

What do we mean by "roughly in half," though? "Roughly" means that, if we have an odd number of values, one heap will have an extra value. Observe that the following is true:

- If `maxHeap.size() > minHeap.size()`, `maxHeap.top()` will be the median.
- If `maxHeap.size() == minHeap.size()`, then the average of `maxHeap.top()` and `minHeap.top()` will be the median.

By the way in which we rebalance the heaps, we will ensure that it is always `maxHeap` with extra element.

The algorithm works as follows. When a new value arrives, it is placed in the `maxHeap` if the value is less than or equal to the median, otherwise it is placed into the `minHeap`. The heap sizes can be equal, or the `maxHeap` may have one extra element. This constraint can easily be restored by shifting an element from one heap to the other. The median is available in constant time, by looking at the top element(s). Updates take `O(log(n))` time.

```
1  Comparator<Integer> maxHeapComparator, minHeapComparator;
2  PriorityQueue<Integer> maxHeap, minHeap;
3
4  void addNewNumber(int randomNumber) {
5    /* Note: addNewNumber maintains a condition that
6     * maxHeap.size() >= minHeap.size() */
7    if (maxHeap.size() == minHeap.size()) {
8      if ((minHeap.peek() != null) &&
9          randomNumber > minHeap.peek()) {
10       maxHeap.offer(minHeap.poll());
11       minHeap.offer(randomNumber);
12     } else {
13       maxHeap.offer(randomNumber);
14     }
15   } else {
16     if(randomNumber < maxHeap.peek()) {
17       minHeap.offer(maxHeap.poll());
18       maxHeap.offer(randomNumber);
19     }
20     else {
21       minHeap.offer(randomNumber);
22     }
23   }
24 }
25
26 double getMedian() {
27   /* maxHeap is always at least as big as minHeap. So if maxHeap is empty, then
28    * minHeap is also. */
29   if (maxHeap.isEmpty()) {
30     return 0;
31   }
```

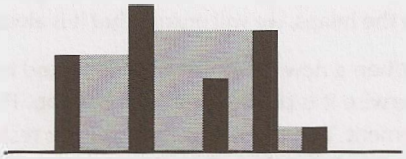
```
32    if (maxHeap.size() == minHeap.size()) {
33       return ((double)minHeap.peek()+(double)maxHeap.peek()) / 2;
34    } else {
35       /* If maxHeap and minHeap are of different sizes, then maxHeap must have one
36        * extra element. Return maxHeap's top element.*/
37       return maxHeap.peek();
38    }
39 }
```

**17.21 Volume of Histogram:** Imagine a histogram (bar graph). Design an algorithm to compute the volume of water it could hold if someone poured water across the top. You can assume that each histogram bar has width 1.

EXAMPLE

Input: {0, 0, 4, 0, 0, 6, 0, 0, 3, 0, 5, 0, 1, 0, 0, 0}

(Black bars are the histogram. Gray is water.)

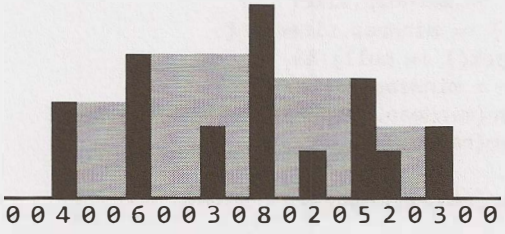



Output: 26

pg 189

## SOLUTION

This is a difficult problem, so let's come up with a good example to help us solve it.




We should study this example to see what we can learn from it. What exactly dictates how big those gray areas are?

### Solution #1

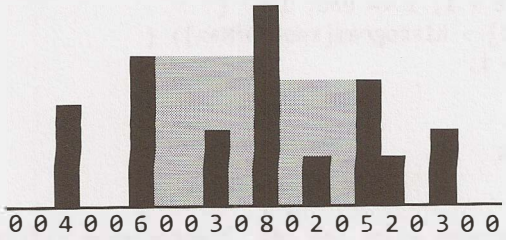
Let's look at the tallest bar, which has size 8. What role does that bar play? It plays an important role for being the highest, but it actually wouldn't matter if that bar instead had height 100. It wouldn't affect the volume.

The tallest bar forms a barrier for water on its left and right. But the volume of water is actually controlled by the next highest bar on the left and right.

- **Water on immediate left of tallest bar:** The next tallest bar on the left has height 6. We can fill up the area in between with water, but we have to deduct the height of each histogram between the tallest and next tallest. This gives a volume on the immediate left of: `(6-0) + (6-0) + (6-3) + (6-0) = 21`.
- **Water on immediate right of tallest bar:** The next tallest bar on the right has height 5. We can now

compute the volume: (5-0) + (5-2) + (5-0) = 13.

This just tells us part of the volume.




What about the rest?

We have essentially two subgraphs, one on the left and one on the right. To find the volume there, we repeat a very similar process.

1. Find the max. (Actually, this is given to us. The highest on the left subgraph is the right border (6) and the highest on the right subgraph is the left border (5).)
2. Find the second tallest in each subgraph. In the left subgraph, this is 4. In the right subgraph, this is 3.
3. Compute the volume between the tallest and the second tallest.
4. Recurse on the edge of the graph.

The code below implements this algorithm.

```
1   int computeHistogramVolume(int[] histogram) {
2     int start = 0;
3     int end = histogram.length - 1;
4
5     int max = findIndexOfMax(histogram, start, end);
6     int leftVolume =  subgraphVolume(histogram, start, max, true);
7     int rightVolume = subgraphVolume(histogram, max, end, false);
8
9     return leftVolume + rightVolume;
10  }
11
12  /* Compute the volume of a subgraph of the histogram. One max is at either start
13   * or end (depending on isLeft). Find second tallest, then compute volume between
14   * tallest and second tallest. Then compute volume of subgraph. */
15  int subgraphVolume(int[] histogram, int start, int end, boolean isLeft) {
16    if (start >= end) return 0;
17    int sum = 0;
18    if (isLeft) {
19      int max = findIndexOfMax(histogram, start, end - 1);
20      sum += borderedVolume(histogram, max, end);
21      sum += subgraphVolume(histogram, start, max, isLeft);
22    } else {
23      int max = findIndexOfMax(histogram, start + 1, end);
24      sum += borderedVolume(histogram, start, max);
25      sum += subgraphVolume(histogram, max, end, isLeft);
26    }
27
28    return sum;
29  }
30
```

```
31 /* Find tallest bar in histogram between start and end. */
32 int findIndexOfMax(int[] histogram, int start, int end) {
33    int indexOfMax = start;
34    for (int i = start + 1; i <= end; i++) {
35       if (histogram[i] > histogram[indexOfMax]) {
36          indexOfMax = i;
37       }
38    }
39    return indexOfMax;
40 }
41
42 /* Compute volume between start and end. Assumes that tallest bar is at start and
43  * second tallest is at end. */
44 int borderedVolume(int[] histogram, int start, int end) {
45    if (start >= end) return 0;
46
47    int min = Math.min(histogram[start], histogram[end]);
48    int sum = 0;
49    for (int i = start + 1; i < end; i++) {
50       sum += min - histogram[i];
51    }
52    return sum;
53 }
```

This algorithm takes $O(N^2)$ time in the worst case, where N is the number of bars in the histogram. This is because we have to repeatedly scan the histogram to find the max height.

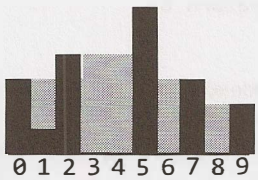
**Solution #2 (Optimized)**

To optimize the previous algorithm, let's think about the exact cause of the inefficiency of the prior algorithm. The root cause is the perpetual calls to `findIndexOfMax`. This suggests that it should be our focus for optimizing.

One thing we should notice is that we don't pass in arbitrary ranges into the `findIndexOfMax` function. It's actually always finding the max from one point to an edge (either the right edge or the left edge). Is there a quicker way we could know what the max height is from a given point to each edge?

Yes. We could precompute this information in $O(N)$ time.

In two sweeps through the histogram (one moving right to left and the other moving left to right), we can create a table that tells us, from any index `i`, the location of the max index on the right and the max index on the left.



```
INDEX: 0 1 2 3 4 5 6 7 8 9
HEIGHT: 3 1 4 0 0 6 0 3 0 2
INDEX LEFT MAX: 0 0 2 2 2 5 5 5 5 5
INDEX RIGHT MAX: 5 5 5 5 5 5 7 7 9 9
```

The rest of the algorithm precedes essentially the same way.

We've chosen to use a HistogramData object to store this extra information, but we could also use a two-dimensional array.

```
int computeHistogramVolume(int[] histogram) {
  int start = 0;
  int end = histogram.length - 1;

  HistogramData[] data = createHistogramData(histogram);

  int max = data[0].getRightMaxIndex(); // Get overall max
  int leftVolume =  subgraphVolume(data, start, max, true);
  int rightVolume = subgraphVolume(data, max, end, false);

  return leftVolume + rightVolume;
}

HistogramData[] createHistogramData(int[] histo) {
  HistogramData[] histogram = new HistogramData[histo.length];
  for (int i = 0; i < histo.length; i++) {
    histogram[i] = new HistogramData(histo[i]);
  }

  /* Set left max index. */
  int maxIndex = 0;
  for (int i = 0; i < histo.length; i++) {
    if (histo[maxIndex] < histo[i]) {
      maxIndex = i;
    }
    histogram[i].setLeftMaxIndex(maxIndex);
  }

  /* Set right max index. */
  maxIndex = histogram.length - 1;
  for (int i = histogram.length - 1; i >= 0; i--) {
    if (histo[maxIndex] < histo[i]) {
      maxIndex = i;
    }
    histogram[i].setRightMaxIndex(maxIndex);
  }

  return histogram;
}

/* Compute the volume of a subgraph of the histogram. One max is at either start
 * or end (depending on isLeft). Find second tallest, then compute volume between
 * tallest and second tallest. Then compute volume of subgraph. */
int subgraphVolume(HistogramData[] histogram, int start, int end,
                   boolean isLeft) {
  if (start >= end) return 0;
  int sum = 0;
  if (isLeft) {
    int max = histogram[end - 1].getLeftMaxIndex();
    sum += borderedVolume(histogram, max, end);
    sum += subgraphVolume(histogram, start, max, isLeft);
  } else {
    int max = histogram[start + 1].getRightMaxIndex();
    sum += borderedVolume(histogram, start, max);
```