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 $^{n(n+1)}/_2$. 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.

```
/* 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 1/10th 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.

```
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);
```

```java
    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)`.

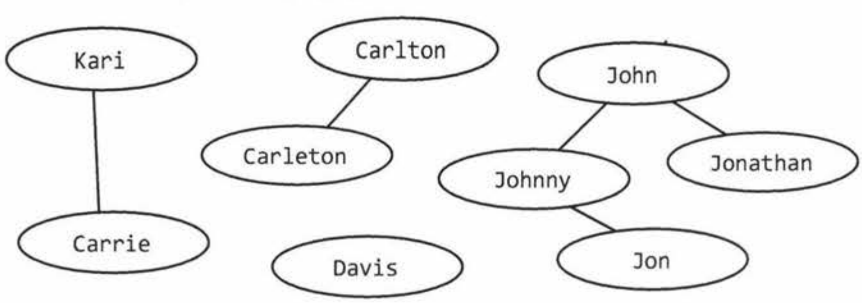
This is pretty good, but let's see if we can make it even faster.

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 $7A_1$ < `min`, and then later we check if $7A_5$ < `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 = {3A_4}
Q56 = {5A_2, 5A_4, 5A_5}
Q76 = {7A_1, 7A_2, 7A_3, 7A_4, 7A_5}
```

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.

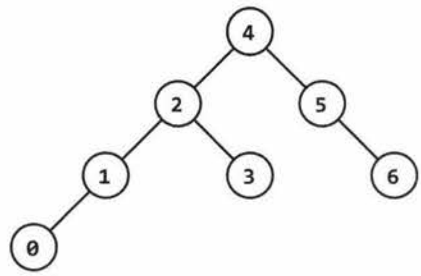
17.12 BiNode: Consider a simple data structure called BiNode, which has pointers to two other nodes. The data structure `BiNode` could be used to represent both a binary tree (where `node1` is the left node and `node2` is the right node) or a doubly linked list (where `node1` is the previous node and `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

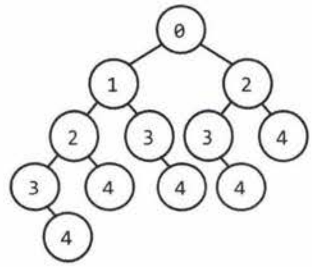
The first thing that may come to mind is a recursive solution. We have essentially a sequence of choices as we walk down the list of appointments: Do we use this appointment or do we not? If we use appointment `i`, we must skip appointment `i + 1` as we can't take back-to-back appointments. Appointment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

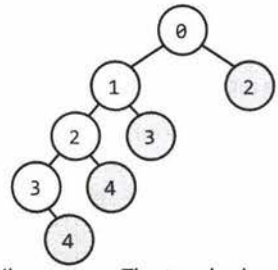
Solution #2: Recursion + Memoization

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.

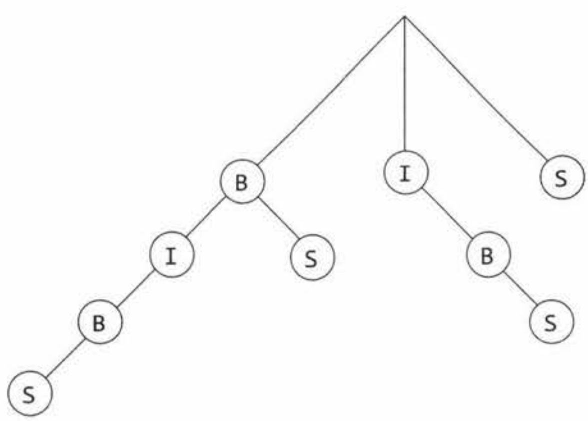
This will take `O(kbt)` time, where k is the length of the longest string in T, b is the length of the bigger string, and `t` is the number of smaller strings within T.

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.

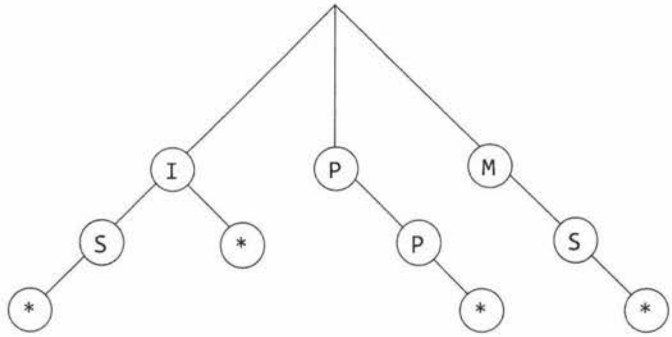
> Reminder: k is the length of the longest string in T, b is the length of the bigger string, and t is the number of smaller strings within T.

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. */
```

```java
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`.

value	7	5	9	9	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

```
1 -> {5, 10, 11}
5 -> {1, 7, 12}
9 -> {2, 3, 9, 15}
```

What is the very first valid subsequence (which contains 1, 5, and 9)? We can just look at the heads of each list to tell us this. The minimum of the heads is the start of the range and the max of the heads is the end of the range. In this case, the first range is [1, 5]. This is currently our "best" subsequence.

How can we find the next one? Well, the next one will not include index 1, so let's remove that from the list.

```
1 -> {5, 10, 11}
5 -> {7, 12}
9 -> {2, 3, 9, 15}
```

The next subsequence is [2, 7]. This is worse than the earlier best, so we can toss it.

Now, what's the next subsequence? We can remove the min from earlier (2) and find out.

```
1 -> {5, 10, 11}
5 -> {7, 12}
9 -> {3, 9, 15}
```

The next subsequence is [3, 7], which is no better or worse than our current best.

We can continue down this path each time, repeating this process. We will end up iterating through all "minimal" subsequences that start from a given point.

1. Current subsequence is [min of heads, max of heads]. Compare to best subsequence and update if necessary.
2. Remove the minimum head.
3. Repeat.

This will give us an `O(SB)` time complexity. This is because for each of B elements, we are doing a comparison to the S other list heads to find the minimum.

This is pretty good, but let's see if we can make that minimum computation faster.

What we're doing in these repeated minimum calls is taking a bunch of elements, finding and removing the minimum, adding in one more element, and then finding the minimum again.

We can make this faster by using a min-heap. First, put each of the heads in a min-heap. Remove the minimum. Look up the list that this minimum came from and add back the new head. Repeat.

To get the list that the minimum element came from, we'll need to use a HeapNode class that stores both the `locationWithinList` (the index) and the `listId`. This way, when we remove the minimum, we can jump back to the correct list and add its new head to the heap.

```
Range shortestSupersequence(int[] array, int[] elements) {
  ArrayList<Queue<Integer>> locations = getLocationsForElements(array, elements);
  if (locations == null) return null;
  return getShortestClosure(locations);
}

/* Get list of queues (linked lists) storing the indices at which each element in
 * smallArray appears in bigArray. */
ArrayList<Queue<Integer>> getLocationsForElements(int[] big, int[] small) {
  /* Initialize hash map from item value to locations. */
  HashMap<Integer, Queue<Integer>> itemLocations =
```

```
      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);
```

```
    }

    return new Range(bestRangeMin, bestRangeMax);
}
```

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 `x`th 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.

```
int missingOne(int[] array) {
  BigInteger fullProduct = productToN(array.length + 1);

  BigInteger actualProduct = new BigInteger("1");
  for (int i = 0; i < array.length; i++) {
    BigInteger value = new BigInteger(array[i] + "");
    actualProduct = actualProduct.multiply(value);
  }

  BigInteger missingNumber = fullProduct.divide(actualProduct);
  return Integer.parseInt(missingNumber.toString());
}

BigInteger productToN(int n) {
  BigInteger fullProduct = new BigInteger("1");
  for (int i = 2; i <= n; i++) {
    fullProduct = fullProduct.multiply(new BigInteger(i + ""));
  }
  return fullProduct;
}
```

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 $\frac{n}{2}$ 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.

```
int[] missingTwo(int[] array) {
  int max_value = array.length + 2;
  int rem_square = squareSumToN(max_value, 2);
  int rem_one = max_value * (max_value + 1) / 2;

  for (int i = 0; i < array.length; i++) {
    rem_square -= array[i] * array[i];
    rem_one -= array[i];
  }

  return solveEquation(rem_one, rem_square);
}

int squareSumToN(int n, int power) {
  int sum = 0;
  for (int i = 1; i <= n; i++) {
    sum += (int) Math.pow(i, power);
  }
  return sum;
}
```

```
int[] solveEquation(int r1, int r2) {
  /* ax^2 + bx + c
   * -->
   * x = [-b +- sqrt(b^2 - 4ac)] / 2a
   * In this case, it has to be a + not a - */
  int a = 2;
  int b = -2 * r1;
  int c = r1 * r1 - r2;

  double part1 = -1 * b;
  double part2 = Math.sqrt(b*b - 4 * a * c);
  double part3 = 2 * a;

  int solutionX = (int) ((part1 + part2) / part3);
  int solutionY = r1 - solutionX;

  int[] solution = {solutionX, solutionY};
  return solution;
}
```

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 $2[\text{something}]^2 - 2s[\text{something}] + s^2-t = 0$, 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: $[-b - \text{sqrt}(b^2 - 4ac)] / 2a$. What's y?

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

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

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

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

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

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.

```
Comparator<Integer> maxHeapComparator, minHeapComparator;
PriorityQueue<Integer> maxHeap, minHeap;

void addNewNumber(int randomNumber) {
  /* Note: addNewNumber maintains a condition that
   * maxHeap.size() >= minHeap.size() */
  if (maxHeap.size() == minHeap.size()) {
    if ((minHeap.peek() != null) &&
      randomNumber > minHeap.peek()) {
      maxHeap.offer(minHeap.poll());
      minHeap.offer(randomNumber);
    } else {
      maxHeap.offer(randomNumber);
    }
  } else {
    if(randomNumber < maxHeap.peek()) {
      minHeap.offer(maxHeap.poll());
      maxHeap.offer(randomNumber);
    }
    else {
      minHeap.offer(randomNumber);
    }
  }
}

double getMedian() {
  /* maxHeap is always at least as big as minHeap. So if maxHeap is empty, then
   * minHeap is also. */
  if (maxHeap.isEmpty()) {
    return 0;
  }
```

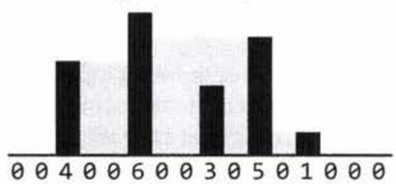
```
    if (maxHeap.size() == minHeap.size()) {
       return ((double)minHeap.peek()+(double)maxHeap.peek()) / 2;
    } else {
       /* If maxHeap and minHeap are of different sizes, then maxHeap must have one
        * extra element. Return maxHeap's top element.*/
       return maxHeap.peek();
 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

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.)`

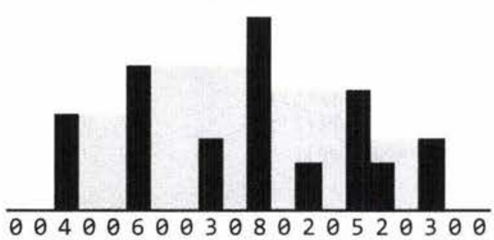


Output: 26

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

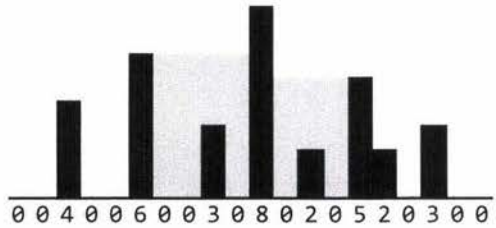
Let's look at the tallest bar, which has size 8. What role does that bar play? It plays an important role for being the highest, but it actually wouldn't matter if that bar instead had height 100. It wouldn't affect the volume.

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.

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

  int max = findIndexOfMax(histogram, start, end);
  int leftVolume =  subgraphVolume(histogram, start, max, true);
  int rightVolume = subgraphVolume(histogram, max, end, false);

  return leftVolume + rightVolume;
}

/* 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(int[] histogram, int start, int end, boolean isLeft) {
  if (start >= end) return 0;
  int sum = 0;
  if (isLeft) {
    int max = findIndexOfMax(histogram, start, end - 1);
    sum += borderedVolume(histogram, max, end);
    sum += subgraphVolume(histogram, start, max, isLeft);
  } else {
    int max = findIndexOfMax(histogram, start + 1, end);
    sum += borderedVolume(histogram, start, max);
    sum += subgraphVolume(histogram, max, end, isLeft);
  }

  return sum;
}

```

```
/* Find tallest bar in histogram between start and end. */
int findIndexOfMax(int[] histogram, int start, int end) {
  int indexOfMax = start;
  for (int i = start + 1; i <= end; i++) {
    if (histogram[i] > histogram[indexOfMax]) {
      indexOfMax = i;
    }
  }
  return indexOfMax;
}

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

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

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.

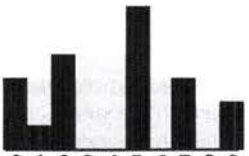
Solution #2 (Optimized)

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.

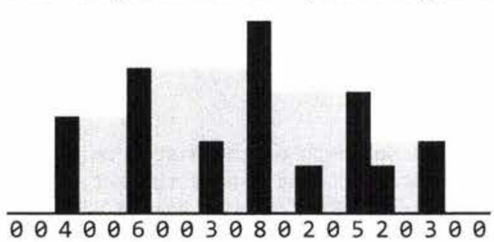
```
int computeHistogramVolume(int[] histogram) {
  int start = 0;
  int end = histogram.length - 1;

  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);
```

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

    return sum;
}

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

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

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

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

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

Solution #3 (Optimized & Simplified)

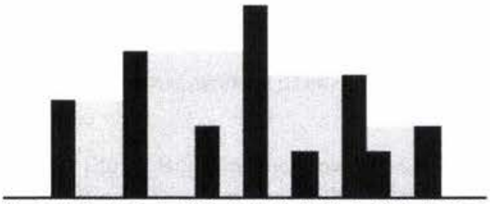
While we can't make the solution faster in terms of big O, we can make it much, much simpler. Let's look at an example again in light of what we've just learned about potential algorithms.



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

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

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



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

Our algorithm now runs in a few simple steps:

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

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

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

  int sum = 0;

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

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

```
                sum += secondTallest - histo[i];
            }
        }

        return sum;
    }
```

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

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

EXAMPLE

Input: DAMP, LIKE

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

pg 189

SOLUTION

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

Brute Force

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

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

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

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

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

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

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

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

LinkedList<String> transform(HashSet<String> visited, String startWord,
```

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

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

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

    return null;
  }

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

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

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

Optimized Solution

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

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

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

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

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

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

The algorithm is otherwise essentially the same.

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

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

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

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

  return null;
}

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

/* Get list of wildcards associated with word. */
ArrayList<String> getWildcardRoots(String w) {
  ArrayList<String> words = new ArrayList<String>();
```

```
  for (int i = 0; i < w.length(); i++) {
    String word = w.substring(0, i) + "_" + w.substring(i + 1);
    words.add(word);
  }
  return words;
}

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

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

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

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

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

Optimal Solution

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

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

Breadth-first search spans out very quickly.

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

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

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

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

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

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

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

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

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

  return null;
}

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

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

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

```

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

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

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

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

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

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

  public String getWord() {
    return word;
  }

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

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

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

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

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

pg 190

SOLUTION

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

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

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

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

Our code works as follows:

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

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

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

boolean isSquare(int[][] matrix, int row, int col, int size) {
```

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

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

Pre-Processing Solution: $O(N^3)$

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

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

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

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

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

(0s right, 0s below)

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

Original Matrix

W	B	W
B	B	W
B	B	W

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

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

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

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

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

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

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

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

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

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

```
}
```

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

pg 190

SOLUTION

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

Brute Force Solution: $O(N^6)$

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

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

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

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

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

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

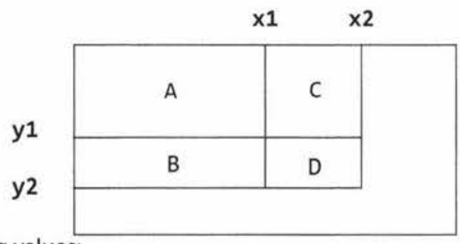
```
        row2 = r2;
        col2 = c2;
        sum = sm;
    }

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

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

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

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

Consider the following rectangle:



Suppose we knew the following values:

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

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

With these values, we know the following:

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

Or, written another way:

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

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

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

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

The following code implements this algorithm.

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

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

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

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

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

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

Optimized Solution: $O(N^3)$

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

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

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

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

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

Picture a submatrix:

rowStart

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

12 -5 3 9 -5

rowEnd

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

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

We now have pseudocode that looks like the following.

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

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

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

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

Our full code looks like this:

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

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

    for (int rowEnd = rowStart; rowEnd < rowCount; rowEnd++) {
      /* Add values at row rowEnd. */
      for (int i = 0; i < colCount; i++) {
```

```
          partialSum[i] += matrix[rowEnd][i];
        }

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

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

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

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

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

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

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

pg 190

SOLUTION

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
      }
   }
   return null;
}
```

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

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

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

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

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

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

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

```
      return true;
   }

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

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

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

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

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

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

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

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

17.26 Sparse Similarity: The similarity of two documents (each with distinct words) is defined to be the size of the intersection divided by the size of the union. For example, if the documents consist of integers, the similarity of `{1, 5, 3}` and `{1, 7, 2, 3}` is `0.4`, because the intersection has size 2 and the union has size 5.

We have a long list of documents (with distinct values and each with an associated ID) where the similarity is believed to be "sparse." That is, any two arbitrarily selected documents are very likely to have similarity 0. Design an algorithm that returns a list of pairs of document IDs and the associated similarity.

Print only the pairs with similarity greater than 0. Empty documents should not be printed at all. For simplicity, you may assume each document is represented as an array of distinct integers.

EXAMPLE

Input:

```
13: {14, 15, 100, 9, 3}
16: {32, 1, 9, 3, 5}
19: {15, 29, 2, 6, 8, 7}
24: {7, 10}
```

Output:

```
ID1, ID2 : SIMILARITY
13, 19   : 0.1
13, 16   : 0.25
19, 24   : 0.14285714285714285
```

pg 190

SOLUTION

This sounds like quite a tricky problem, so let's start off with a brute force algorithm. If nothing else, it will help wrap our heads around the problem.

Remember that each document is an array of distinct "words", and each is just an integer.

Brute Force

A brute force algorithm is as simple as just comparing all arrays to all other arrays. At each comparison, we compute the size of the intersection and size of the union of the two arrays.

Note that we only want to print this pair if the similarity is greater than 0. The union of two arrays can never be zero (unless both arrays are empty, in which case we don't want them printed anyway). Therefore, we are really just printing the similarity if the intersection is greater than 0.

How do we compute the size of the intersection and the union?

The intersection means the number of elements in common. Therefore, we can just iterate through the first array (A) and check if each element is in the second array (B). If it is, increment an `intersection` variable.

To compute the union, we need to be sure that we don't double count elements that are in both. One way to do this is to count up all the elements in A that are *not* in B. Then, add in all the elements in B. This will avoid double counting as the duplicate elements are only counted with B.

Alternatively, we can think about it this way. If we *did* double count elements, it would mean that elements in the intersection (in both A and B) were counted twice. Therefore, the easy fix is to just remove these duplicate elements.

```
union(A, B) = A + B - intersection(A, B)
```

This means that all we really need to do is compute the intersection. We can derive the union, and therefore similarity, from that immediately.

This gives us an `O(AB)` algorithm, just to compare two arrays (or documents).

However, we need to do this for all pairs of D documents. If we assume each document has at most W words then the runtime is $O(D^2 W^2)$.

Slightly Better Brute Force

As a quick win, we can optimize the computation for the similarity of two arrays. Specifically, we need to optimize the intersection computation.

We need to know the number of elements in common between the two arrays. We can throw all of A's elements into a hash table. Then we iterate through B, incrementing `intersection` every time we find an element in A.

This takes `O(A + B)` time. If each array has size W and we do this for D arrays, then this takes $O(D^2 W)$.

Before implementing this, let's first think about the classes we'll need.

We'll need to return a list of document pairs and their similarities. We'll use a `DocPair` class for this. The exact return type will be a hash table that maps from `DocPair` to a double representing the similarity.

```
public class DocPair {
   public int doc1, doc2;

   public DocPair(int d1, int d2) {
      doc1 = d1;
      doc2 = d2;
   }

   @Override
   public boolean equals(Object o) {
      if (o instanceof DocPair) {
         DocPair p = (DocPair) o;
         return p.doc1 == doc1 && p.doc2 == doc2;
      }
      return false;
   }

   @Override
   public int hashCode() { return (doc1 * 31) ^ doc2; }
}
```

It will also be useful to have a class that represents the documents.

```
public class Document {
   private ArrayList<Integer> words;
   private int docId;

   public Document(int id, ArrayList<Integer> w) {
      docId = id;
      words = w;
   }

   public ArrayList<Integer> getWords() { return words; }
   public int getId() { return docId; }
```

```
   public int size() { return words == null ? 0 : words.size(); }
}
```

Strictly speaking, we don't need any of this. However, readability is important, and it's a lot easier to read `ArrayList<Document>` than `ArrayList<ArrayList<Integer>>`.

Doing this sort of thing not only shows good coding style, it also makes your life in an interview a lot easier. You have to write a lot less. (You probably would not define the entire `Document` class, unless you had extra time or your interviewer asked you to.)

```
1  HashMap<DocPair, Double> computeSimilarities(ArrayList<Document> documents) {
2     HashMap<DocPair, Double> similarities = new HashMap<DocPair, Double>();
3     for (int i = 0; i < documents.size(); i++) {
4        for (int j = i + 1; j < documents.size(); j++) {
5           Document doc1 = documents.get(i);
6           Document doc2 = documents.get(j);
7           double sim = computeSimilarity(doc1, doc2);
8           if (sim > 0) {
9              DocPair pair = new DocPair(doc1.getId(), doc2.getId());
10             similarities.put(pair, sim);
11          }
12       }
13    }
14    return similarities;
15 }
16
17 double computeSimilarity(Document doc1, Document doc2) {
18    int intersection = 0;
19    HashSet<Integer> set1 = new HashSet<Integer>();
20    set1.addAll(doc1.getWords());
21
22    for (int word : doc2.getWords()) {
23       if (set1.contains(word)) {
24          intersection++;
25       }
26    }
27
28    double union = doc1.size() + doc2.size() - intersection;
29    return intersection / union;
30 }
```

Observe what's happening on line 28. Why did we make `union` a `double`, when it's obviously an `integer`?

We did this to avoid an integer division bug. If we didn't do this, the division would "round" down to an integer. This would mean that the similarity would almost always return 0. Oops!

Slightly Better Brute Force (Alternate)

If the documents were sorted, you could compute the intersection between two documents by walking through them in sorted order, much like you would when doing a sorted merge of two arrays.

This would take O(A + B) time. This is the same time as our current algorithm, but less space. Doing this on D documents with W words each would take $O(D^2 W)$ time.

Since we don't know that the arrays are sorted, we could first sort them. This would take O(D * W log W) time. The full runtime then is $O(D * W \log W + D^2 W)$.

We cannot necessarily assume that the second part "dominates" the first one, because it doesn't necessarily. It depends on the relative size of D and `log W`. Therefore, we need to keep both terms in our runtime expression.

Optimized (Somewhat)

It is useful to create a larger example to really understand the problem.

```
13: {14, 15, 100, 9, 3}
16: {32, 1, 9, 3, 5}
19: {15, 29, 2, 6, 8, 7}
24: {7, 10, 3}
```

At first, we might try various techniques that allow us to more quickly eliminate potential comparisons. For example, could we compute the min and max values in each array? If we did that, then we'd know that arrays with no overlap in ranges don't need to be compared.

The problem is that this doesn't really fix our runtime issue. Our best runtime thus far is $O(D^2 W)$. With this change, we're still going to be comparing all $O(D^2)$ pairs, but the $O(W)$ part might go to $O(1)$ sometimes. That $O(D^2)$ part is going to be a really big problem when D gets large.

Therefore, let's focus on reducing that $O(D^2)$ factor. That is the "bottleneck" in our solution. Specifically, this means that, given a document `docA`, we want to find all documents with some similarity—and we want to do this without "talking" to each document.

What would make a document similar to `docA`? That is, what characteristics define the documents with similarity > 0?

Suppose `docA` is `{14, 15, 100, 9, 3}`. For a document to have similarity > 0, it needs to have a 14, a 15, a 100, a 9, or a 3. How can we quickly gather a list of all documents with one of those elements?

The slow (and, really, only way) is to read every single word from every single document to find the documents that contain a 14, a 15, a 100, a 9, or a 3. That will take $O(DW)$ time. Not good.

However, note that we're doing this repeatedly. We can reuse the work from one call to the next.

If we build a hash table that maps from a word to all documents that contain that word, we can very quickly know the documents that overlap with `docA`.

```
1 -> 16
2 -> 19
3 -> 13, 16, 24
5 -> 16
6 -> 19
7 -> 19, 24
8 -> 19
9 -> 13, 16
...
```

When we want to know all the documents that overlap with `docA`, we just look up each of `docA`'s items in this hash table. We'll then get a list of all documents with some overlap. Now, all we have to do is compare `docA` to each of those documents.

If there are P pairs with similarity > 0, and each document has W words, then this will take $O(PW)$ time (plus $O(DW)$ time to create and read this hash table). Since we expect P to be much less than D^2, this is much better than before.

Optimized (Better)

Let's think about our previous algorithm. Is there any way we can make it more optimal?

If we consider the runtime—O(PW + DW)—we probably can't get rid of the O(DW) factor. We have to touch each word at least once, and there are O(DW) words. Therefore, if there's an optimization to be made, it's probably in the O(PW) term.

It would be difficult to eliminate the P part in O(PW) because we have to at least print all P pairs (which takes O(P) time). The best place to focus, then, is on the W part. Is there some way we can do less than O(W) work for each pair of similar documents?

One way to tackle this is to analyze what information the hash table gives us. Consider this list of documents:

```
12: {1, 5, 9}
13: {5, 3, 1, 8}
14: {4, 3, 2}
15: {1, 5, 9, 8}
17: {1, 6}
```

If we look up document 12's elements in a hash table for this document, we'll get:

```
1 -> {12, 13, 15, 17}
5 -> {12, 13, 15}
9 -> {12, 15}
```

This tells us that documents 13, 15, and 17 have some similarity. Under our current algorithm, we would now need to compare document 12 to documents 13, 15, and 17 to see the number of elements document 12 has in common with each (that is, the size of the intersection). The union can be computed from the document sizes and the intersection, as we did before.

Observe, though, that document 13 appeared twice in the hash table, document 15 appeared three times, and document 17 appeared once. We discarded that information. But can we use it instead? What does it indicate that some documents appeared multiple times and others didn't?

Document 13 appeared twice because it has two elements (1 and 5) in common. Document 17 appeared once because it has only one element (1) in common. Document 15 appeared three times because it has three elements (1, 5, and 9) in common. This information can actually directly give us the size of the intersection.

We could go through each document, look up the items in the hash table, and then count how many times each document appears in each item's lists. There's a more direct way to do it.

1. As before, build a hash table for a list of documents.
2. Create a new hash table that maps from a document pair to an integer (which will indicate the size of the intersection).
3. Read the first hash table by iterating through each list of documents.
4. For each list of documents, iterate through the pairs in that list. Increment the intersection count for each pair.

Comparing this runtime to the previous one is a bit tricky. One way we can look at it is to realize that before we were doing O(W) work for each similar pair. That's because once we noticed that two documents were similar, we touched every single word in each document. With this algorithm, we're only touching the words that actually overlap. The worst cases are still the same, but for many inputs this algorithm will be faster.

```
HashMap<DocPair, Double>
computeSimilarities(HashMap<Integer, Document> documents) {
```

```
    HashMapList<Integer, Integer> wordToDocs = groupWords(documents);
    HashMap<DocPair, Double> similarities = computeIntersections(wordToDocs);
    adjustToSimilarities(documents, similarities);
    return similarities;
  }

  /* Create hash table from each word to where it appears. */
  HashMapList<Integer, Integer> groupWords(HashMap<Integer, Document> documents) {
    HashMapList<Integer, Integer> wordToDocs = new HashMapList<Integer, Integer>();

    for (Document doc : documents.values()) {
      ArrayList<Integer> words = doc.getWords();
      for (int word : words) {
        wordToDocs.put(word, doc.getId());
      }
    }

    return wordToDocs;
  }

  /* Compute intersections of documents. Iterate through each list of documents and
   * then each pair within that list, incrementing the intersection of each page. */
  HashMap<DocPair, Double> computeIntersections(
      HashMapList<Integer, Integer> wordToDocs {
    HashMap<DocPair, Double> similarities = new HashMap<DocPair, Double>();
    Set<Integer> words = wordToDocs.keySet();
    for (int word : words) {
      ArrayList<Integer> docs = wordToDocs.get(word);
      Collections.sort(docs);
      for (int i = 0; i < docs.size(); i++) {
        for (int j = i + 1; j < docs.size(); j++) {
          increment(similarities, docs.get(i), docs.get(j));
        }
      }
    }

    return similarities;
  }

  /* Increment the intersection size of each document pair. */
  void increment(HashMap<DocPair, Double> similarities, int doc1, int doc2) {
    DocPair pair = new DocPair(doc1, doc2);
    if (!similarities.containsKey(pair)) {
      similarities.put(pair, 1.0);
    } else {
      similarities.put(pair, similarities.get(pair) + 1);
    }
  }

  /* Adjust the intersection value to become the similarity. */
  void adjustToSimilarities(HashMap<Integer, Document> documents,
                            HashMap<DocPair, Double> similarities) {
    for (Entry<DocPair, Double> entry : similarities.entrySet()) {
      DocPair pair = entry.getKey();
      Double intersection = entry.getValue();
      Document doc1 = documents.get(pair.doc1);
```

```
      Document doc2 = documents.get(pair.doc2);
      double union = (double) doc1.size() + doc2.size() - intersection;
      entry.setValue(intersection / union);
    }
}

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

For a set of documents with sparse similarity, this will run much faster than the original naive algorithm, which compares all pairs of documents directly.

Optimized (Alternative)

There's an alternative algorithm that some candidates might come up with. It's slightly slower, but still quite good.

Recall our earlier algorithm that computed the similarity between two documents by sorting them. We can extend this approach to multiple documents.

Imagine we took all of the words, tagged them by their original document, and then sorted them. The prior list of documents would look like this:

1_{12}, 1_{13}, 1_{15}, 1_{16}, 2_{14}, 3_{13}, 3_{14}, 4_{14}, 5_{12}, 5_{13}, 5_{15}, 6_{16}, 8_{13}, 8_{15}, 9_{12}, 9_{15}

Now we have essentially the same approach as before. We iterate through this list of elements. For each sequence of identical elements, we increment the intersection counts for the corresponding pair of documents.

We will use an Element class to group together documents and words. When we sort the list, we will sort first on the word but break ties on the document ID.

```
class Element implements Comparable<Element> {
  public int word, document;
  public Element(int w, int d) {
    word = w;
    document = d;
  }

  /* When we sort the words, this function will be used to compare the words. */
  public int compareTo(Element e) {
    if (word == e.word) {
      return document - e.document;
    }
    return word - e.word;
  }
}

HashMap<DocPair, Double> computeSimilarities(
    HashMap<Integer, Document> documents) {
  ArrayList<Element> elements = sortWords(documents);
  HashMap<DocPair, Double> similarities = computeIntersections(elements);
  adjustToSimilarities(documents, similarities);
  return similarities;
}

/* Throw all words into one list, sorting by the word and then the document. */
ArrayList<Element> sortWords(HashMap<Integer, Document> docs) {
  ArrayList<Element> elements = new ArrayList<Element>();
```

```
    for (Document doc : docs.values()) {
      ArrayList<Integer> words = doc.getWords();
      for (int word : words) {
        elements.add(new Element(word, doc.getId()));
      }
    }
    Collections.sort(elements);
    return elements;
  }

  /* Increment the intersection size of each document pair. */
  void increment(HashMap<DocPair, Double> similarities, int doc1, int doc2) {
    DocPair pair = new DocPair(doc1, doc2);
    if (!similarities.containsKey(pair)) {
      similarities.put(pair, 1.0);
    } else {
      similarities.put(pair, similarities.get(pair) + 1);
    }
  }

  /* Adjust the intersection value to become the similarity. */
  HashMap<DocPair, Double> computeIntersections(ArrayList<Element> elements) {
    HashMap<DocPair, Double> similarities = new HashMap<DocPair, Double>();

    for (int i = 0; i < elements.size(); i++) {
      Element left = elements.get(i);
      for (int j = i + 1; j < elements.size(); j++) {
        Element right = elements.get(j);
        if (left.word != right.word) {
          break;
        }
        increment(similarities, left.document, right.document);
      }
    }
    return similarities;
  }

  /* Adjust the intersection value to become the similarity. *
  void adjustToSimilarities(HashMap<Integer, Document> documents,
                            HashMap<DocPair, Double> similarities) {
    for (Entry<DocPair, Double> entry : similarities.entrySet()) {
      DocPair pair = entry.getKey();
      Double intersection = entry.getValue();
      Document doc1 = documents.get(pair.doc1);
      Document doc2 = documents.get(pair.doc2);
      double union = (double) doc1.size() + doc2.size() - intersection;
      entry.setValue(intersection / union);
    }
  }
```

The first step of this algorithm is slower than that of the prior algorithm, since it has to sort rather than just add to a list. The second step is essentially equivalent.

Both will run much faster than the original naive algorithm.