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 $\{A_1, ..., A_{k-1}\}$). 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 $\{A_1, ..., A_{k-1}\}$) 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:

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

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

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

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

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

Optimal Algorithm

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

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

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

Let's imagine our list looks like:

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

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

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

That is, our list above would look like:

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

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

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

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

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

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

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

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

Our pseudocode for this problem is as follows:

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

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

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

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

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

The code below implements this algorithm.

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

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

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

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

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

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

Input: 1 2 5 9 5 9 5 5 5

Output: 5

pg 187

SOLUTION

Let's start off with an example:

3 1 7 1 3 7 3 7 1 7 7

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

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

Solution #1 (Slow)

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

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

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

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

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

Solution #2 (Optimal)

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

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

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

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

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

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

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

What would this look like?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

pg 187

SOLUTION

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

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

The code below implements this algorithm.

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

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

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

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

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

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

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

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

Consider the following lists of locations.

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

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

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

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

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

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

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

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

We can implement this algorithm as shown below.

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

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

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

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

   return best;
 }

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

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

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

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