```
}

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.

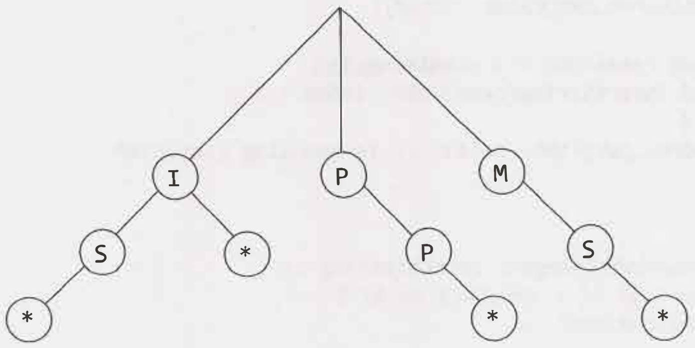
> Reminder: k is the length of the longest string in T, b is the length of the bigger string, and t is the number of smaller strings within T.

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.

```java
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 $O(SB^2)$ time, where B is the length of `bigString` and S is the length of `smallString`. This is because at each of the B characters, we potentially do $O(SB)$ work: S scans of the rest of the string, which has potentially B characters.

Optimized

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

It uses O(SB) space.

More Optimized

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

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

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

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

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

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

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

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

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

Alternative & More Optimal Solution

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

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 =
```