One easy optimization is to notice that if the pattern starts with 'a', then the a string must start at the beginning of `value`. (Otherwise, the b *string must start at the beginning of* `value`.) Therefore, there aren't $O(n^2)$ possible values for a; there are $O(n)$.

The algorithm then is to check if the pattern starts with a or b. If it starts with b, we can "invert" it (flipping each 'a' to a 'b' and each 'b' to an 'a') so that it starts with 'a'. Then, iterate through all possible substrings for a (each of which must begin at index 0) and all possible substrings for b (each of which must begin at some character after the end of a). As before, we then compare the string for this pattern with the original string.

This algorithm now takes $O(n^4)$ time.

There's one more minor (optional) optimization we can make. We don't actually need to do this "inversion" if the string starts with 'b' instead of 'a'. The `buildFromPattern` method can take care of this. We can think about the first character in the pattern as the "main" item and the other character as the alternate character. The `buildFromPattern` method can build the appropriate string based on whether 'a' is the main character or alternate character.

```
1  boolean doesMatch(String pattern, String value) {
2    if (pattern.length() == 0) return value.length() == 0;
3
4    int size = value.length();
5    for (int mainSize = 0; mainSize < size; mainSize++) {
6      String main = value.substring(0, mainSize);
7      for (int altStart = mainSize; altStart <= size; altStart++) {
8        for (int altEnd = altStart; altEnd <= size; altEnd++) {
9          String alt = value.substring(altStart, altEnd);
10         String cand = buildFromPattern(pattern, main, alt);
11         if (cand.equals(value)) {
12           return true;
13         }
14       }
15     }
16   }
17   return false;
18 }
19
20 String buildFromPattern(String pattern, String main, String alt) {
21   StringBuffer sb = new StringBuffer();
22   char first = pattern.charAt(0);
23   for (char c : pattern.toCharArray()) {
24     if (c == first) {
25       sb.append(main);
26     } else {
27       sb.append(alt);
28     }
29   }
30   return sb.toString();
31 }
```

We should look for a more optimal algorithm.

Optimized

Let's think through our current algorithm. Searching through all values for the main string is fairly fast (it takes $O(n)$ time). It's the alternate string that is so slow: $O(n^2)$ time. We should study how to optimize that.

Suppose we have a pattern like aabab and we're comparing it to the string catcatgocatgo. Once we've picked "cat" as the value for a to try, then the a strings are going to take up nine characters (three a strings with length three each). Therefore, the b strings must take up the remaining four characters, with each having length two. Moreover, we actually know exactly where they must occur, too. If a is cat, and the pattern is aabab, then b must be go.

In other words, once we've picked a, we've picked b too. There's no need to iterate. Gathering some basic stats on pattern (number of as, number of bs, first occurrence of each) and iterating through values for a (or whichever the main string is) will be sufficient.

```
1  boolean doesMatch(String pattern, String value) {
2    if (pattern.length() == 0) return value.length() == 0;
3
4    char mainChar = pattern.charAt(0);
5    char altChar = mainChar == 'a' ? 'b' : 'a';
6    int size = value.length();
7
8    int countOfMain = countOf(pattern, mainChar);
9    int countOfAlt = pattern.length() - countOfMain;
10   int firstAlt = pattern.indexOf(altChar);
11   int maxMainSize = size / countOfMain;
12
13   for (int mainSize = 0; mainSize <= maxMainSize; mainSize++) {
14     int remainingLength = size - mainSize * countOfMain;
15     String first = value.substring(0, mainSize);
16     if (countOfAlt == 0 || remainingLength % countOfAlt == 0) {
17       int altIndex = firstAlt * mainSize;
18       int altSize = countOfAlt == 0 ? 0 : remainingLength / countOfAlt;
19       String second = countOfAlt == 0 ? "" :
20                       value.substring(altIndex, altSize + altIndex);
21
22       String cand = buildFromPattern(pattern, first, second);
23       if (cand.equals(value)) {
24         return true;
25       }
26     }
27   }
28   return false;
29 }
30
31 int countOf(String pattern, char c) {
32   int count = 0;
33   for (int i = 0; i < pattern.length(); i++) {
34     if (pattern.charAt(i) == c) {
35       count++;
36     }
37   }
38   return count;
39 }
40
41 String buildFromPattern(...) {  /* same as before */ }
```

This algorithm takes $O(n^2)$, since we iterate through O(n) possibilities for the main string and do O(n) work to build and compare the strings.

Observe that we've also cut down the possibilities for the main string that we try. If there are three instances of the main string, then its length cannot be any more than one third of value.

Optimized (Alternate)

If you don't like the work of building a string only to compare it (and then destroy it), we can eliminate this.

Instead, we can iterate through the values for a and b as before. But this time, to check if the string matches the pattern (given those values for a and b), we walk through `value`, comparing each substring to the first instance of the a and b strings.

```
1  boolean doesMatch(String pattern, String value) {
2    if (pattern.length() == 0) return value.length() == 0;
3
4    char mainChar = pattern.charAt(0);
5    char altChar = mainChar == 'a' ? 'b' : 'a';
6    int size = value.length();
7
8    int countOfMain = countOf(pattern, mainChar);
9    int countOfAlt = pattern.length() - countOfMain;
10   int firstAlt = pattern.indexOf(altChar);
11   int maxMainSize = size / countOfMain;
12
13   for (int mainSize = 0; mainSize <= maxMainSize; mainSize++) {
14     int remainingLength = size - mainSize * countOfMain;
15     if (countOfAlt == 0 || remainingLength % countOfAlt == 0) {
16       int altIndex = firstAlt * mainSize;
17       int altSize = countOfAlt == 0 ? 0 : remainingLength / countOfAlt;
18       if (matches(pattern, value, mainSize, altSize, altIndex)) {
19         return true;
20       }
21     }
22   }
23   return false;
24 }
25
26 /* Iterates through pattern and value. At each character within pattern, checks if
27  * this is the main string or the alternate string. Then checks if the next set of
28  * characters in value match the original set of those characters (either the main
29  * or the alternate. */
30 boolean matches(String pattern, String value, int mainSize, int altSize,
31                 int firstAlt) {
32   int stringIndex = mainSize;
33   for (int i = 1; i < pattern.length(); i++) {
34     int size = pattern.charAt(i) == pattern.charAt(0) ? mainSize : altSize;
35     int offset = pattern.charAt(i) == pattern.charAt(0) ? 0 : firstAlt;
36     if (!isEqual(value, offset, stringIndex, size)) {
37       return false;
38     }
39     stringIndex += size;
40   }
41   return true;
42 }
43
44 /* Checks if two substrings are equal, starting at given offsets and continuing to
45  * size. */
46 boolean isEqual(String s1, int offset1, int offset2, int size) {
47   for (int i = 0; i < size; i++) {
48     if (s1.charAt(offset1 + i) != s1.charAt(offset2 + i)) {
49       return false;
```

```
50        }
51     }
52     return true;
53  }
```

This algorithm will still take $O(n^2)$ time, but the benefit is that it can short circuit when matches fail early (which they usually will). The previous algorithm must go through all the work to build the string before it can learn that it has failed.

16.19 Pond Sizes: You have an integer matrix representing a plot of land, where the value at that location represents the height above sea level. A value of zero indicates water. A pond is a region of water connected vertically, horizontally, or diagonally. The size of the pond is the total number of connected water cells. Write a method to compute the sizes of all ponds in the matrix.

EXAMPLE

Input:

```
0 2 1 0
0 1 0 1
1 1 0 1
0 1 0 1
```

Output: 2, 4, 1 (in any order)

pg 184

SOLUTION

The first thing we can try is just walking through the array. It's easy enough to find water: when it's a zero, that's water.

Given a water cell, how can we compute the amount of water nearby? If the cell is not adjacent to any zero cells, then the size of this pond is 1. If it is, then we need to add in the adjacent cells, plus any water cells adjacent to those cells. We need to, of course, be careful to not recount any cells. We can do this with a modified breadth-first or depth-first search. Once we visit a cell, we permanently mark it as visited.

For each cell, we need to check eight adjacent cells. We could do this by writing in lines to check up, down, left, right, and each of the four diagonal cells. It's even easier, though, to do this with a loop.

```
1  ArrayList<Integer> computePondSizes(int[][] land) {
2     ArrayList<Integer> pondSizes = new ArrayList<Integer>();
3     for (int r = 0; r < land.length; r++) {
4        for (int c = 0; c < land[r].length; c++) {
5           if (land[r][c] == 0) { // Optional. Would return anyway.
6              int size = computeSize(land, r, c);
7              pondSizes.add(size);
8           }
9        }
10    }
11    return pondSizes;
12 }
13
14 int computeSize(int[][] land, int row, int col) {
15    /* If out of bounds or already visited. */
16    if (row < 0 || col < 0 || row >= land.length || col >= land[row].length ||
17        land[row][col] != 0) { // visited or not water
18       return 0;
19    }
```

```
20    int size = 1;
21    land[row][col] = -1; // Mark visited
22    for (int dr = -1; dr <= 1; dr++) {
23       for (int dc = -1; dc <= 1; dc++) {
24          size += computeSize(land, row + dr, col + dc);
25       }
26    }
27    return size;
28 }
```

In this case, we marked a cell as visited by setting its value to -1. This allows us to check, in one line (`land[row][col] != 0`), if the value is valid dry land or visited. In either case, the value will be zero.

You might also notice that the for loop iterates through nine cells, not eight. It includes the current cell. We could add a line in there to not recurse if `dr == 0` and `dc == 0`. This really doesn't save us much. We'll execute this if-statement in eight cells unnecessarily, just to avoid one recursive call. The recursive call returns immediately since the cell is marked as visited.

If you don't like modifying the input matrix, you can create a secondary `visited` matrix.

```
1  ArrayList<Integer> computePondSizes(int[][] land) {
2     boolean[][] visited = new boolean[land.length][land[0].length];
3     ArrayList<Integer> pondSizes = new ArrayList<Integer>();
4     for (int r = 0; r < land.length; r++) {
5        for (int c = 0; c < land[r].length; c++) {
6           int size = computeSize(land, visited, r, c);
7           if (size > 0) {
8              pondSizes.add(size);
9           }
10       }
11    }
12    return pondSizes;
13 }
14
15 int computeSize(int[][] land, boolean[][] visited, int row, int col) {
16    /* If out of bounds or already visited. */
17    if (row < 0 || col < 0 || row >= land.length || col >= land[row].length ||
18        visited[row][col] || land[row][col] != 0) {
19       return 0;
20    }
21    int size = 1;
22    visited[row][col] = true;
23    for (int dr = -1; dr <= 1; dr++) {
24       for (int dc = -1; dc <= 1; dc++) {
25          size += computeSize(land, visited, row + dr, col + dc);
26       }
27    }
28    return size;
29 }
```

Both implementations are $O(WH)$, where W is the width of the matrix and H is the height.

Note: Many people say "$O(N)$" or "$O(N^2)$", as though N has some inherent meaning. It doesn't. Suppose this were a square matrix. You could describe the runtime as $O(N)$ or $O(N^2)$. Both are correct, depending on what you mean by N. The runtime is $O(N^2)$, where N is the length of one side. Or, if N is the number of cells, it is $O(N)$. Be careful by what you mean by N. In fact, it might be safer to just not use N at all when there's any ambiguity as to what it could mean.

Some people will miscompute the runtime to be $O(N^4)$, reasoning that the `computeSize` method could take as long as $O(N^2)$ time and you might call it as much as $O(N^2)$ times (and apparently assuming an NxN matrix, too). While those are both basically correct statements, you can't just multiply them together. That's because as a single call to `computeSize` gets more expensive, the number of times it is called goes down.

For example, suppose the very first call to `computeSize` goes through the entire matrix. That might take $O(N^2)$ time, but then we never call `computeSize` again.

Another way to compute this is to think about how many times each cell is "touched" by either call. Each cell will be touched once by the `computePondSizes` function. Additionally, a cell might be touched once by each of its adjacent cells. This is still a constant number of touches per cell. Therefore, the overall runtime is $O(N^2)$ on an NxN matrix or, more generally, $O(WH)$.

16.20 T9: On old cell phones, users typed on a numeric keypad and the phone would provide a list of words that matched these numbers. Each digit mapped to a set of 0 - 4 letters. Implement an algorithm to return a list of matching words, given a sequence of digits. You are provided a list of valid words (provided in whatever data structure you'd like). The mapping is shown in the diagram below:

1	**2** abc	**3** def
4 ghi	**5** jkl	**6** mno
7 pqrs	**8** tuv	**9** wxyz
	0	

EXAMPLE

Input: 8733

Output: `tree, used`

pg 184

SOLUTION

We could approach this in a couple of ways. Let's start with a brute force algorithm.

Brute Force

Imagine how you would solve the problem if you had to do it by hand. You'd probably try every possible value for each digit with all other possible values.

This is exactly what we do algorithmically. We take the first digit and run through all the characters that map to that digit. For each character, we add it to a `prefix` variable and recurse, passing the prefix downward. Once we run out of characters, we print `prefix` (which now contains the full word) if the string is a valid word.

We will assume the list of words is passed in as a `HashSet`. A `HashSet` operates similarly to a hash table, but rather than offering key->value lookups, it can tell us if a word is contained in the set in $O(1)$ time.

```
1  ArrayList<String> getValidT9Words(String number, HashSet<String> wordList) {
2    ArrayList<String> results = new ArrayList<String>();
3    getValidWords(number, 0, "", wordList, results);
4    return results;
5  }
6
```

```
void getValidWords(String number, int index, String prefix,
                   HashSet<String> wordSet, ArrayList<String> results) {
   /* If it's a complete word, print it. */
   if (index == number.length() && wordSet.contains(prefix)) {
      results.add(prefix);
      return;
   }

   /* Get characters that match this digit. */
   char digit = number.charAt(index);
   char[] letters = getT9Chars(digit);

   /* Go through all remaining options. */
   if (letters != null) {
      for (char letter : letters) {
         getValidWords(number, index + 1, prefix + letter, wordSet, results);
      }
   }
}

/* Return array of characters that map to this digit. */
char[] getT9Chars(char digit) {
   if (!Character.isDigit(digit)) {
      return null;
   }
   int dig = Character.getNumericValue(digit) - Character.getNumericValue('0');
   return t9Letters[dig];
}

/* Mapping of digits to letters. */
char[][] t9Letters = {null, null, {'a', 'b', 'c'}, {'d', 'e', 'f'},
   {'g', 'h', 'i'}, {'j', 'k', 'l'}, {'m', 'n', 'o'}, {'p', 'q', 'r', 's'},
   {'t', 'u', 'v'},  {'w', 'x', 'y', 'z'}
};
```

This algorithm runs in $O(4^N)$ time, where N is the length of the string. This is because we recursively branch four times for each call to `getValidWords`, and we recurse until a call stack depth of N.

This is very, very slow on large strings.

Optimized

Let's return to thinking about how you would do this, if you were doing it by hand. Imagine the example of `33835676368` (which corresponds to `development`). If you were doing this by hand, I bet you'd skip over solutions that start with `fftf`[3383], as no valid words start with those characters.

Ideally, we'd like our program to make the same sort of optimization: stop recursing down paths which will obviously fail. Specifically, if there are no words in the dictionary that start with `prefix`, stop recursing.

The `Trie` data structure (see "Tries (Prefix Trees)" on page 105) can do this for us. Whenever we reach a string which is not a valid prefix, we exit.

```
ArrayList<String> getValidT9Words(String number, Trie trie) {
   ArrayList<String> results = new ArrayList<String>();
   getValidWords(number, 0, "", trie.getRoot(), results);
   return results;
}
```

```
void getValidWords(String number, int index, String prefix, TrieNode trieNode,
                   ArrayList<String> results) {
   /* If it's a complete word, print it. */
   if (index == number.length()) {
      if (trieNode.terminates()) { // Is complete word
         results.add(prefix);
      }
      return;
   }

   /* Get characters that match this digit */
   char digit = number.charAt(index);
   char[] letters = getT9Chars(digit);

   /* Go through all remaining options. */
   if (letters != null) {
      for (char letter : letters) {
         TrieNode child = trieNode.getChild(letter);
         /* If there are words that start with prefix + letter,
          * then continue recursing. */
         if (child != null) {
            getValidWords(number, index + 1, prefix + letter, child, results);
         }
      }
   }
}
```

It's difficult to describe the runtime of this algorithm since it depends on what the language looks like. However, this "short-circuiting" will make it run much, much faster in practice.

Most Optimal

Believe or not, we can actually make it run even faster. We just need to do a little bit of preprocessing. That's not a big deal though. We were doing that to build the trie anyway.

This problem is asking us to list all the words represented by a particular number in T9. Instead of trying to do this "on the fly" (and going through a lot of possibilities, many of which won't actually work), we can just do this in advance.

Our algorithm now has a few steps:

Pre-Computation:

1. Create a hash table that maps from a sequence of digits to a list of strings.
2. Go through each word in the dictionary and convert it to its T9 representation (e.g., `APPLE -> 27753`). Store each of these in the above hash table. For example, 8733 would map to `{used, tree}`.

Word Lookup:

1. Just look up the entry in the hash table and return the list.

That's it!

```
/* WORD LOOKUP */
ArrayList<String> getValidT9Words(String numbers,
                                  HashMapList<String, String> dictionary) {
   return dictionary.get(numbers);
}
```

```
/* PRECOMPUTATION */

/* Create a hash table that maps from a number to all words that have this
 * numerical representation. */
HashMapList<String, String> initializeDictionary(String[] words) {
   /* Create a  hash table that maps from a letter to the digit */
   HashMap<Character, Character> letterToNumberMap = createLetterToNumberMap();

   /* Create word -> number map. */
   HashMapList<String, String> wordsToNumbers = new HashMapList<String, String>();
   for (String word : words) {
      String numbers = convertToT9(word, letterToNumberMap);
      wordsToNumbers.put(numbers, word);
   }
   return wordsToNumbers;
}

/* Convert mapping of number->letters into letter->number. */
HashMap<Character, Character> createLetterToNumberMap() {
   HashMap<Character, Character> letterToNumberMap =
      new HashMap<Character, Character>();
   for (int i = 0; i < t9Letters.length; i++) {
      char[] letters = t9Letters[i];
      if (letters != null) {
         for (char letter : letters) {
            char c = Character.forDigit(i, 10);
            letterToNumberMap.put(letter, c);
         }
      }
   }
   return letterToNumberMap;
}

/* Convert from a string to its T9 representation. */
String convertToT9(String word, HashMap<Character, Character> letterToNumberMap) {
   StringBuilder sb = new StringBuilder();
   for (char c : word.toCharArray()) {
      if (letterToNumberMap.containsKey(c)) {
         char digit = letterToNumberMap.get(c);
         sb.append(digit);
      }
   }
   return sb.toString();
}

char[][] t9Letters = /* Same as before */

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

Getting the words that map to this number will run in O(N) time, where N is the number of digits. The O(N) comes in during the hash table look up (we need to convert the number to a hash table). If you know the words are never longer than a certain max size, then you could also describe the runtime as O(1).

Note that it's easy to think, "Oh, linear—that's not that fast." But it depends what it's linear *on*. Linear on the length of the word is extremely fast. Linear on the length of the dictionary is not so fast.

16.21 Sum Swap: Given two arrays of integers, find a pair of values (one value from each array) that you can swap to give the two arrays the same sum.

EXAMPLE

Input: {4, 1, 2, 1, 1, 2} and {3, 6, 3, 3}

Output: {1, 3}

pg 184

SOLUTION

We should start by trying to understand what exactly we're looking for.

We have two arrays and their sums. Although we likely aren't given their sums upfront, we can just act like we are for now. After all, computing the sum is an O(N) operation and we know we can't beat O(N) anyway. Computing the sum, therefore, won't impact the runtime.

When we move a (positive) value a from array A to array B, then the sum of A drops by a and the sum of B increases by a.

We are looking for two values, a and b, such that:

```
sumA - a + b = sumB - b + a
```

Doing some quick math:

```
2a - 2b = sumA - sumB
a - b = (sumA - sumB) / 2
```

Therefore, we're looking for two values that have a specific target difference: `(sumA - sumB) / 2`.

Observe that because that the target must be an integer (after all, you can't swap two integers to get a non-integer difference), we can conclude that the difference between the sums must be even to have a valid pair.

Brute Force

A brute force algorithm is simple enough. We just iterate through the arrays and check all pairs of values.

We can either do this the "naive" way (compare the new sums) or by looking for a pair with that difference.

Naive approach:

```
1  int[] findSwapValues(int[] array1, int[] array2) {
2    int sum1 = sum(array1);
3    int sum2 = sum(array2);
4
5    for (int one : array1) {
6      for (int two : array2) {
7        int newSum1 = sum1 - one + two;
8        int newSum2 = sum2 - two + one;
9        if (newSum1 == newSum2) {
10         int[] values = {one, two};
11         return values;
12       }
13     }
14   }
15
16   return null;
17 }
```

Target approach:

```
1  int[] findSwapValues(int[] array1, int[] array2) {
```