1.4 **Palindrome Permutation:** Given a string, write a function to check if it is a permutation of a palindrome. A palindrome is a word or phrase that is the same forwards and backwards. A permutation is a rearrangement of letters. The palindrome does not need to be limited to just dictionary words.

EXAMPLE

Input: `Tact Coa`

Output: `True (permutations: "taco cat", "atco cta", etc.)`

Hints: #106, #121, #134, #136

pg 195

We will use this approach in this problem. The algorithm employs a two-scan approach. In the first scan, we count the number of spaces. By tripling this number, we can compute how many extra characters we will have in the final string. In the second pass, which is done in reverse order, we actually edit the string. When we see a space, we replace it with %20. If there is no space, then we copy the original character.

The code below implements this algorithm.

```
void replaceSpaces(char[] str, int trueLength) {
  int spaceCount = 0, index, i = 0;
  for (i = 0; i < trueLength; i++) {
    if (str[i] == ' ') {
      spaceCount++;
    }
  }
  index = trueLength + spaceCount * 2;
  if (trueLength < str.length) str[trueLength] = '\0'; // End array
  for (i = trueLength - 1; i >= 0; i--) {
    if (str[i] == ' ') {
      str[index - 1] = '0';
      str[index - 2] = '2';
      str[index - 3] = '%';
      index = index - 3;
    } else {
      str[index - 1] = str[i];
      index--;
    }
  }
}
```

We have implemented this problem using character arrays, because Java strings are immutable. If we used strings directly, the function would have to return a new copy of the string, but it would allow us to implement this in just one pass.

1.4 **Palindrome Permutation:** Given a string, write a function to check if it is a permutation of a palindrome. A palindrome is a word or phrase that is the same forwards and backwards. A permutation is a rearrangement of letters. The palindrome does not need to be limited to just dictionary words.

EXAMPLE

Input: `Tact Coa`

Output: `True` (permutations: "`taco cat`", "`atco cta`", etc.)

pg 91

SOLUTION

This is a question where it helps to figure out what it means for a string to be a permutation of a palindrome. This is like asking what the "defining features" of such a string would be.

A palindrome is a string that is the same forwards and backwards. Therefore, to decide if a string is a permutation of a palindrome, we need to know if it can be written such that it's the same forwards and backwards.

What does it take to be able to write a set of characters the same way forwards and backwards? We need to have an even number of almost all characters, so that half can be on one side and half can be on the other side. At most one character (the middle character) can have an odd count.

For example, we know `tactcoapapa` is a permutation of a palindrome because it has two Ts, four As, two

Cs, two Ps, and one O. That O would be the center of all possible palindromes.

To be more precise, strings with even length (after removing all non-letter characters) must have all even counts of characters. Strings of an odd length must have exactly one character with an odd count. Of course, an "even" string can't have an odd number of exactly one character, otherwise it wouldn't be an even-length string (an odd number + many even numbers = an odd number). Likewise, a string with odd length can't have all characters with even counts (sum of evens is even). It's therefore sufficient to say that, to be a permutation of a palindrome, a string can have no more than one character that is odd. This will cover both the odd and the even cases.

This leads us to our first algorithm.

Solution #1

Implementing this algorithm is fairly straightforward. We use a hash table to count how many times each character appears. Then, we iterate through the hash table and ensure that no more than one character has an odd count.

```
1  boolean isPermutationOfPalindrome(String phrase) {
2    int[] table = buildCharFrequencyTable(phrase);
3    return checkMaxOneOdd(table);
4  }
5
6  /* Check that no more than one character has an odd count. */
7  boolean checkMaxOneOdd(int[] table) {
8    boolean foundOdd = false;
9    for (int count : table) {
10     if (count % 2 == 1) {
11       if (foundOdd) {
12         return false;
13       }
14       foundOdd = true;
15     }
16   }
17   return true;
18 }
19
20 /* Map each character to a number. a -> 0, b -> 1, c -> 2, etc.
21  * This is case insensitive. Non-letter characters map to -1. */
22 int getCharNumber(Character c) {
23   int a = Character.getNumericValue('a');
24   int z = Character.getNumericValue('z');
25   int val = Character.getNumericValue(c);
26   if (a <= val && val <= z) {
27     return val - a;
28   }
29   return -1;
30 }
31
32 /* Count how many times each character appears. */
33 int[] buildCharFrequencyTable(String phrase) {
34   int[] table = new int[Character.getNumericValue('z') -
35                         Character.getNumericValue('a') + 1];
36   for (char c : phrase.toCharArray()) {
37     int x = getCharNumber(c);
```

```
38          if (x != -1) {
39            table[x]++;
40          }
41        }
42        return table;
43    }
```

This algorithm takes O(N) time, where N is the length of the string.

Solution #2

We can't optimize the big O time here since any algorithm will always have to look through the entire string. However, we can make some smaller incremental improvements. Because this is a relatively simple problem, it can be worthwhile to discuss some small optimizations or at least some tweaks.

Instead of checking the number of odd counts at the end, we can check as we go along. Then, as soon as we get to the end, we have our answer.

```
1   boolean isPermutationOfPalindrome(String phrase) {
2     int countOdd = 0;
3     int[] table = new int[Character.getNumericValue('z') -
4                           Character.getNumericValue('a') + 1];
5
6     for (char c : phrase.toCharArray()) {
7       int x = getCharNumber(c);
8       if (x != -1) {
9         table[x]++;
10        if (table[x] % 2 == 1) {
11          countOdd++;
12        } else {
13          countOdd--;
14        }
15      }
16    }
17    return countOdd <= 1;
18  }
```

It's important to be very clear here that this is not necessarily more optimal. It has the same big O time and might even be slightly slower. We have eliminated a final iteration through the hash table, but now we have to run a few extra lines of code for each character in the string.

You should discuss this with your interviewer as an alternate, but not necessarily more optimal, solution.

Solution #3

If you think more deeply about this problem, you might notice that we don't actually need to know the counts. We just need to know if the count is even or odd. Think about flipping a light on/off (that is initially off). If the light winds up in the `off` state, we don't know how many times we flipped it, but we do know it was an even count.

Given this, we can use a single integer (as a bit vector). When we see a letter, we map it to an integer between 0 and 26 (assuming an English alphabet). Then we toggle the bit at that value. At the end of the iteration, we check that at most one bit in the integer is set to 1.

We can easily check that no bits in the integer are 1: just compare the integer to 0. There is actually a very elegant way to check that an integer has exactly one bit set to 1.

Picture an integer like `00010000`. We could of course shift the integer repeatedly to check that there's only a single 1. Alternatively, if we subtract 1 from the number, we'll get `00001111`. What's notable about this

is that there is no overlap between the numbers (as opposed to say 00101000, which, when we subtract 1 from, we get 00100111.) So, we can check to see that a number has exactly one 1 because if we subtract 1 from it and then AND it with the new number, we should get 0.

```
00010000 - 1 = 00001111
00010000 & 00001111 = 0
```

This leads us to our final implementation.

```
1  boolean isPermutationOfPalindrome(String phrase) {
2    int bitVector = createBitVector(phrase);
3    return bitVector == 0 || checkExactlyOneBitSet(bitVector);
4  }
5
6  /* Create a bit vector for the string. For each letter with value i, toggle the
7   * ith bit. */
8  int createBitVector(String phrase) {
9    int bitVector = 0;
10   for (char c : phrase.toCharArray()) {
11     int x = getCharNumber(c);
12     bitVector = toggle(bitVector, x);
13   }
14   return bitVector;
15 }
16
17 /* Toggle the ith bit in the integer. */
18 int toggle(int bitVector, int index) {
19   if (index < 0) return bitVector;
20
21   int mask = 1 << index;
22   if ((bitVector & mask) == 0) {
23     bitVector |= mask;
24   } else {
25     bitVector &= ~mask;
26   }
27   return bitVector;
28 }
29
30 /* Check that exactly one bit is set by subtracting one from the integer and
31  * ANDing it with the original integer. */
32 boolean checkExactlyOneBitSet(int bitVector) {
33   return (bitVector & (bitVector - 1)) == 0;
34 }
```

Like the other solutions, this is O(N).

It's interesting to note a solution that we did not explore. We avoided solutions along the lines of "create all possible permutations and check if they are palindromes." While such a solution would work, it's entirely infeasible in the real world. Generating all permutations requires factorial time (which is actually worse than exponential time), and it is essentially infeasible to perform on strings longer than about 10–15 characters.

I mention this (impractical) solution because a lot of candidates hear a problem like this and say, "In order to check if A is in group B, I must know everything that is in B and then check if one of the items equals A." That's not always the case, and this problem is a simple demonstration of it. You don't need to generate all permutations in order to check if one is a palindrome.