Project 2: Palindromes

Before smart phones and Snapchat people whittled their free time away in dark, dank hovels playing silly word games and puzzles. One such word puzzle is to try to find sentences that make sense and are palindromes. Recall that a palindrome is a sequence of characters that read the same forward and reversed. For example, “kayak” and “noon” are words that are palindromes because if you reverse the order of the characters, you get the same word back. An example of a sentence palindrome would be “Was it a car or a cat I saw?”.

Note that for a sentence palindrome, the convention is that:

1. you remove the spaces and punctuation,
2. make the letters all the same case,
3. and then check if that string is a palindrome.

We would easily be able to write a C++ function to do this preprocessing. Let’s call our function `sentenceToString(const string & value)`, and if our program called this function as: `sentenceToString(“Was it a car or a cat I saw?”);` it should return the string “wasitacaroracatisaw”, which we would then verify was a palindrome.

Description

For this project we will be implementing a C++ class called FindPalindrome. The idea is that if we create an instance of this class, we can add words to the instance, and the instance will figure out all orders of the words that form sentence palindromes. For example, if I were to add the words: “Odd”, “Even”, “Never”, and “or” to an instance, it should figure out that “Never Odd or Even” is a sentence palindrome.

Allowable Words and Word Restrictions: It is a bit too complicated for us to figure out if character string is really an English word, so for this project a word will simply be a character string such as “Aaa” or “hAppT”. When we add words to the class, we will add each word as a string data type, and we impose several restrictions on what constitutes an allowable word. The restrictions are:

1. The word (as a string) must consist only of characters from ‘a’—‘z’ and ‘A’—‘Z’ (that is, no numbers, special characters, punctuation, etc.; thus, “don’t” and “Whatz up”, and “Ace1Hardware” are not allowable words),

2. We will preserve the case of letters in the word (that is, if we add the word “jUXz”, when we return a sentence palindrome with this word, the characters in the string will still be “jUXz”).

3. Although letter case has to preserved when we add words, case is ignored when determining whether or not a sentence is a palindrome (e.g., “Kayak” is a palindrome).

4. The words that you add to an instance of the class have to be unique; thus, you cannot add “Happy” if “happy” has already been added. Note that we ignore case in determining if a word is unique.

The Number of Possible Sentence Palindromes for N Words is Large: Suppose we added the words “a”, “AA”, and “AaA” to an instance of our class. Note that these words are allowable based on the above rules. The class instance should figure out that there are 6 possible sentence palindromes, namely:

1. “a” “AA” “AaA”
2. “a” “AaA” “AA”
3. “AA” “a” “AaA”
4. “AA” “AaA” “a”
5. “AaA” “a” “AA”
6. “AaA” “AA” “a”

In general, if we have N unique words, there are N! possible orderings of these N words. Yes indeed, N! gets large quickly as N increases. In fact, an algorithmic analysis of the approach might say, the number of possible sentence palindromes is “exponential in N”. As a bit of an aside, why it is “exponential” is sort of interesting. If you can remember back to Kindergarten, you might remember the substitute teacher talking about Stirling’s approximation or formula, namely that:

$$N! \sim \sqrt{\boxed{2\pi N}}(N/e)^N$$

It is the N^N part that makes N! exponential in N. In any case, the bottom line is that the number of possible sentence palindromes is N! for N words in our class.

Hint: if your words are all made from the character ‘a’, as in the above example, then the number of sentence palindromes is N!. Reminder to self: This could be a test! :^)

A Recursive Algorithm to Generate All Possible Sentences: The crux of this project is that we are going to explore the possible sentence palindromes given N words using a recursive algorithm. In fact, we are going to give you the recursive function (and you have to use it in your code). The function is:

```
void recursiveFindPalindromes(std::vector<std::string> currentCandidateVector,

                              std::vector<std::string> currentStringVector);
```

Let’s spend a bit of time looking at how this function is going to be called. To explain this, we will use a notation for the vector<string> objects. These objects are vectors of words; for example, we could write the currentCandidateVector object as $\{V_1, V_2, \ldots, V_R\}$, where this set represents the ordered words in the object; that is, there are R words in this vector, and each word V_i, $1 \le i \le R$, is some allowable word represented as a string, such as “CAT”. Likewise, the object currentStringVector is also a set of allowable words say $\{W_1, W_2, \ldots, W_S\}$, again stored as a vector of strings, but these words are not yet ordered.

This function is going to call itself recursively. The idea is that the currentCandidateVector object is part of “candidate” sentence for which we want to determine if it is a palindrome when we add any order of the words from the currentStringVector object.

Well, from the discussion above, we know that there are S! possible orderings of the words in the currentStringVector object, and we are *not* going to check them all inside a single call to this function. Instead, we are going to recursively call recursiveFindPalindromes just S times by choosing, in turn, one of the W_i words and adding them to the candidate vector, and then making a recursive call to the method recursiveFindPalindromes.

Below is a table of what should be in the argument vectors for each recursive call. Note that in this list the table shows the arguments as being the original arguments that the function received. You could do this, or make copies of these arguments to pass to the recursive call. In any case, the arguments are being passed by value, not be reference. Hence, we can modify them however we like and the vectors will be unchanged upon return within the calling function.

Recursive Call Number	**currentCandidateVector**	**currentStringVector**
1	{V_1, V_2, …, $V_{R,}$,W_1}	{W_2, W_3, …, W_S}
2	{V_1, V_2, …, $V_{R,}$,W_2}	{W_1, W_3, …, W_S}
…	…	…
S	{V_1, V_2, …, $V_{R,}$,W_S}	{W_1, W_2, …, W_{S-1}}

Optimizations

Whenever one is faced with an algorithm with exponential complexity, it worth searching for shortcuts or optimizations. We will take advantage of two observations.

Property #1: When you count the number of times each character appears in a palindrome, at most one character may appear an odd number of times.

Proof: If a palindrome is of even length, the count of each character must be even as if we were to split the palindrome in half, each character that appears in the left half clearly must also be in the right half (by the definition of palindrome and its reversal property). If the palindrome is of odd length, the character in the center appears an odd number of times, all other characters appear an even number of times (unless they are the same as the center character) per the argument for an even length palindrome.

Property #2: Whenever you split a palindrome into two substrings, the characters in the smaller (or equal length) substring must all appear in the larger substring.

Proof: Suppose we split a palindrome into two substrings, call these “left” and “right”. Since a palindrome is symmetric, we can assume that “left” is smaller (otherwise we could simply reverse the order of the substrings). By the definition of a palindrome, each character in “left” at location i must appear at the reversed location (right.length() – i) in “right”. Hence, all the characters in the substring “left” must appear in the substring “right”.

We will use these two observations to enable us to implement “cuts” in our search algorithm. We call these “cuts” because this strategy is used in “branch and cut” search algorithms. The idea in “branch and cut” is that if a “branch” (a set of possible solutions) can be shown to all be infeasible (because of some property of the branch), then you can “cut” that entire branch off from your search (and save the time of searching that branch). Let’s see how we can use the above two observations to implement such “cuts.”

First, consider how we can use Property #1. Suppose we have some word list {W_1, W_2, …, W_N} in our FindPalindrome instance. If the characters from these words to not satisfy Property #1, then there is no point in searching for sentence palindromes for this word list as there cannot be one.

Second, consider how we can use Property #2. Suppose we enter the function recursiveFindPalindromes and we find that the characters in word lists represented by the vectors currentCandidateVector and currentStringVector do not satisfy Property #2. In this case, there is no reason to make the recursive calls, instead we can just return without performing the searches as there cannot be a solution given these vectors.

For this project we will implement these two "cut tests" as the functions:

```
bool cutTest1(const std::vector<std::string> & stringVector);


bool cutTest2(const std::vector<string> & stringVector1,
        const std::vector<std::string> & stringVector2);
```

We will also "expose" these cut tests by making them public in the class so that we can test them externally in our class unit tests. In practice, we should make them private once we are satisfied that they work, as they do not provide any useful functionality to user of our class.

Specification

The class specification for FindPalindrome is given in formatted comments in the starter code file FindPalindrome.hpp.

Do not modify the public portion of the class definition. Obviously, you may use std::vector, however you may not use any other container classes in your implementation (in fact if you want to I would discuss it with us as it may indicate a misunderstanding). Your implementation should not leak memory or have invalid read/writes.

You will need to define the missing internal members and methods (marked TODO) and implement all methods in the FindPalindrome.cpp file. You should add appropriate comment blocks to any methods that you add in FindPalindrome.hpp. You will need to write tests in the file FindPalindrome_test.cpp using the Catch testing framework. The included CMakeLists.txt file sets up everything for you. There is also a main program FindPalindrome_main.cpp for you to play with (it is not part of your grade).

As done for Project 1, create and generate your build directory for your development environment as described in class.