```
/*
 * Programming Challenge 25 - UNIT TEST
 */

#include <cassert>
#include <iostream>
#include <sstream>
using namespace std;

/*
 * Compute and return the factorial of a value, using a recursive algorithm. Zero factorial
 * will return 1.
 * @param value an unsigned integer containing the value for which the factorial will be computed
 * @return an unsigned integer containing the computed factorial of the value
*/
unsigned int factorial (unsigned int value);

/*
 * Return a specified value in a Fibonacci sequence, using a recursive algorithm. The lowest value requested in the sequence
 * must never be less than one.
 * @param fibValue an unsigned integer specifying which value in the Fibonacci sequence to return
 * @return an unsigned integer containing the requested value in the Fibonacci sequence
 */
unsigned int fibonacci (unsigned int fibValue);

/*
 * Test a single word to see if it is a palindrome, using a recursive algorithm. The word must be all in the same case
 * (upper or lower) and cannot contain spaces, digits, or punctuation.
 * @param word a string containing the word to be tested
 * @return true of word is a palindrome, else false; empty string and single character strings
 *         are considered palindromes
 */
bool wordIsPalindrome (string word);

/*
 * Produce a string containing the contents of an array, separated by single spaces,
 * starting at a specified index and going forward to the end of the array, using a recursive algorithm. The returned
 * string will contain an extra space character after the last value added.
 * @param array an integer array containing the values to be added to the string
 * @param startIndex an unsigned integer containing the index of the first value in the array to be added
 *        to the output string
 * @param size an unsigned integer containing the number of elements in the array
 * @return a string containing the contents of the array, separated by spaces; returns empty string
 *         if the startIndex is >= the size of the array
 */
string arrayForwardsAsString (int array[], unsigned int startIndex, unsigned int size);

/*
 * Produce a string containing the contents of an array, separated by single spaces,
 * starting at a specified index and going backward to the beginning of the array, using a recursive
```

```
 * algorithm. The returned string will contain an extra space character after the last value added.
 * @param array an integer array containing the values to be added to the string
 * @param startIndex an unsigned integer containing the index of the first value in the array to be added
 *        to the output string
 * @param size an unsigned integer containing the number of elements in the array
 * @return a string containing the contents of the array, separated by spaces, in reverse order; returns empty string
 *         if the startIndex is < zero
 */
string arrayBackwardsAsString (int array[], int startIndex, unsigned int size);

/* for unit testing -- do not alter */
template <typename X, typename A>
void btassert(A assertion);
void unittest ();

int main (int argc, char* argv[])
{
    unittest();

    return 0;
}

/* function declarations */

unsigned int factorial (unsigned int value)
{
    if( value > 1 )
    {
        return (value * factorial ( value - 1 ));
    }
    else
    {
        return 1;
    }
}

unsigned int fibonacci (unsigned int fibValue)
{
    if (fibValue == 0 || fibValue == 1)
    {
      return fibValue;
    }
    else
    {
      return fibonacci( (fibValue - 1) ) + fibonacci( (fibValue - 2) );
    }
}

bool wordIsPalindrome (string word)
{
```

```
    if( word.length() == 0 || word.length() == 1 )
    {
        return true;
    }
    else if( ( ( word.length() != 0 ) ) && ( word.at( 0 ) == word.at( (word.length() - 1) ) ) )
    {
        return wordIsPalindrome( word.substr( 1, (word.length() - 2) ) );
    }
    else
    {
        return false;
    }
}

string arrayForwardsAsString (int array[], unsigned int startIndex, unsigned int size)
{
    if( startIndex < size )
    {
        stringstream ss;
        ss << array[startIndex] << " " << arrayForwardsAsString( array, (startIndex + 1), size );
        return ss.str();
    }

    return "";
}

string arrayBackwardsAsString (int array[], int startIndex, unsigned int size)
{
    if( startIndex >= 0 )
    {
        stringstream ss;
        ss << array[startIndex] << " " << arrayBackwardsAsString( array, (startIndex - 1), size );
        return ss.str();
    }

    return "";
}

/*
 * Unit testing functions. Do not alter.
 */
void unittest ()
{
    cout << "\nSTARTING UNIT TEST\n\n";

    try {
        btassert<bool>(factorial(0) == 1);
        cout << "Passed TEST 1: factorial(0) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 1: factorial(0) #\n";
```

```
    }

    try {
        btassert<bool>(factorial(1) == 1);
        cout << "Passed TEST 2: factorial(1) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 2: factorial(1) #\n";
    }

    try {
        btassert<bool>(factorial(2) == 2);
        cout << "Passed TEST 3: factorial(2) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 3: factorial(2) #\n";
    }

    try {
        btassert<bool>(factorial(5) == 120);
        cout << "Passed TEST 4: factorial(5) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 4: factorial(5) #\n";
    }

    try {
        btassert<bool>(fibonacci(1) == 1);
        cout << "Passed TEST 5: fibonacci(1) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 5: fibonacci(1) #\n";
    }

    try {
        btassert<bool>(fibonacci(2) == 1);
        cout << "Passed TEST 6: fibonacci(2) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 6: fibonacci(2) #\n";
    }

    try {
        btassert<bool>(fibonacci(3) == 2);
        cout << "Passed TEST 7: fibonacci(3) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 7: fibonacci(3) #\n";
    }

    try {
        btassert<bool>(fibonacci(15) == 610);
        cout << "Passed TEST 8: fibonacci(15) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 8: fibonacci(15) #\n";
    }
```

```
    try {
        btassert<bool>(wordIsPalindrome("") == true);
        cout << "Passed TEST 9: wordIsPalindrome(\"\") \n";
    } catch (bool b) {
        cout << "# FAILED TEST 9: wordIsPalindrome(\"\") #\n";
    }

    try {
        btassert<bool>(wordIsPalindrome("a") == true);
        cout << "Passed TEST 10: wordIsPalindrome(\"a\") \n";
    } catch (bool b) {
        cout << "# FAILED TEST 10: wordIsPalindrome(\"a\") #\n";
    }

    try {
        btassert<bool>(wordIsPalindrome("sitonapotatopanotis") == true);
        cout << "Passed TEST 11: wordIsPalindrome(\"sitonapotatopanotis\") \n";
    } catch (bool b) {
        cout << "# FAILED TEST 11: wordIsPalindrome(\"sitonapotatopanotis\") #\n";
    }

    try {
        btassert<bool>(wordIsPalindrome("baseball") == false);
        cout << "Passed TEST 12: wordIsPalindrome(\"baseball\") \n";
    } catch (bool b) {
        cout << "# FAILED TEST 12: wordIsPalindrome(\"baseball\") #\n";
    }

    int numbers[5] = {5, 10, 15, 20, 25};

    try {
        btassert<bool>(arrayForwardsAsString(numbers, 0, 5) == "5 10 15 20 25 ");
        cout << "Passed TEST 13: arrayForwardsAsString(numbers, 0, 5) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 13: arrayForwardsAsString(numbers, 0, 5) #\n";
    }

    try {
        btassert<bool>(arrayForwardsAsString(numbers, 3, 5) == "20 25 ");
        cout << "Passed TEST 14: arrayForwardsAsString(numbers, 0, 5) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 14: arrayForwardsAsString(numbers, 0, 5) #\n";
    }

    try {
        btassert<bool>(arrayForwardsAsString(numbers, 5, 5) == "");
        cout << "Passed TEST 15: arrayForwardsAsString(numbers, 5, 5) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 15: arrayForwardsAsString(numbers, 5, 5) #\n";
```

```
    }

    try {
        btassert<bool>(arrayBackwardsAsString(numbers, 4, 5) == "25 20 15 10 5 ");
        cout << "Passed TEST 16: arrayBackwardsAsString(numbers, 4, 5) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 16: arrayBackwardsAsString(numbers, 4, 5) #\n";
    }

    try {
        btassert<bool>(arrayBackwardsAsString(numbers, 1, 5) == "10 5 ");
        cout << "Passed TEST 17: arrayBackwardsAsString(numbers, 1, 5) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 17: arrayBackwardsAsString(numbers, 1, 5) #\n";
    }

    try {
        btassert<bool>(arrayBackwardsAsString(numbers, -1, 5) == "");
        cout << "Passed TEST 18: arrayBackwardsAsString(numbers, -1, 5) \n";
    } catch (bool b) {
        cout << "# FAILED TEST 18: arrayBackwardsAsString(numbers, -1, 5) #\n";
    }

    cout << "\nUNIT TEST COMPLETE\n\n";
}

template <typename X, typename A>
void btassert (A assertion)
{
    if (!assertion)
        throw X();
}
```