Palindrome Efficiency

Like our original,naïve *fibonacci* function this implementation is also very inefficient, but for a different reason.

- The `fib()` function was inefficient because every time we calculated a term, we first had to (re)calculate all of the lower terms. It was expensive in terms of **time**.
- The `isPalindrome()` function is inefficient because every time we enter the recursive call, we first have to create a new substring, which not only takes time, but also uses extra memory. This function is expensive in terms of **space**.

We can improve the performance by making these changes:

- Calculate the size of the string only once.
- Don’t make a new substring on each call.

The main inefficiency is the **repeated `substr()` calls**. You can avoid this by passing indices to keep track of the positions instead of creating new substrings.

Of course, that means we'll need a **helper** and a **wrapper**. Here they are:

```
bool palHelper(const string&, int i1, int i2)
{
    if (i1 >= i2) { return true; }
    return str.at(i1) == str.at(i2) &&
        palHelper(str, i1 + 1, i2 - 1);
}
// Wrapper
bool isPalindrome(const string& str)
{
    return palHelper(str, 0, str.size() - 1);
}
```