Assignment 11: Recursion

Course 'Imperative Programming' (IPC031)

Make sure to start working on the assignment **before the lab** session, otherwise you will be too late for the deadline

1 Background

In this assignment you implement a number of directly recursive functions.

Important: You must solve all assignment parts using purely recursive functions. This means using iterative code such as `for`, `while`, and `do-while` loops is not allowed, even if used in addition to recursive code.

2 Learning objectives

After doing this assignment you are able to:

- to implement recursive algorithms by directly recursive functions;
- to reason about recursive functions;
- to realize that order of run-time complexity is a property of an algorithm and not of a problem (there can be many different algorithms solving the same problem).

3 Assignment

Part 1: The power function

Part 1.1: Naive power

Implement the recursive equation of the power function as a directly recursive function in C++ (parameter n must be a non-negative integer, whereas parameter x can be any integer):

$$\mathsf{power}(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot \mathsf{power}(x, n-1) & \text{if } n > 0 \end{cases}$$

Test your code for different, representative values (hint: check the conditions in the recursive equation).

Part 1.2: Power, more effective

The above realization of the power function is naive in the sense that the computation of $\mathsf{power}(x, n)$ requires n multiplications of x and the use of intermediate results. The order of run-time complexity is $\mathcal{O}(n)$. By making use of the property $x^{2n} = x^n \cdot x^n$, or equivalently, $x^{2n} = (x^n)^2$ you can implement a more efficient version. Implement this more efficient version as a directly recursive function. What is the order of run-time complexity of this more efficient algorithm? Write your answer as a comment in your function implementation.

Part 2: Palindromes

A palindrome is a text that is identical to its reversed version:

$$\mathsf{palindrome}(w) = \begin{cases} \mathsf{true} & \text{if } w \text{ is the empty string} \\ \mathsf{true} & \text{if } w \text{ is a single character} \\ \mathsf{false} & \text{if } w = cw'c',\ \text{and } c \neq c' \\ \mathsf{palindrome}(w') & \text{if } w = cw'c \end{cases}$$

For instance, "otto" and "lepel" are palindromes. In this part you develop directly recursive functions that determine whether a string is a (variant of a) palindrome. These functions use string index lower bound `i` and string index upper bound `j` to determine the slice of the string that is inspected. When testing a text `w`, the functions are called with actual parameters `w, 0, ssize(w)-1`. Use `w.at(i)` to access the character at index `i`, not `w[i]`, as this latter version does not perform bounds checking.

Part 2.1: Straight palindromes

Develop the directly recursive function:

```
bool palindrome1 (string text, int i, int j)
```

which decides if `text.at(i) ... text.at(j)` is a palindrome.

Examples:

- `palindrome1 ("otto", 0, 3)`
  returns `true`.
- `palindrome1 ("Otto", 0, 3)`
  returns `false` because `'O'` is not equal to `'o'`.
- `palindrome1 ("Madam,␣I'm␣Adam.", 0, 15)`
  returns `false` because `'M'` is not equal to `'.'`.

Part 2.2: Case-insensitive palindromes

Develop the directly recursive function:

```
bool palindrome2 (string text, int i, int j)
```

which decides if `text.at(i) ... text.at(j)` is a palindrome, but this time it should consider `'a'` also equal to `'A'`, `'b'` also equal to `'B'`, and so on for all the letter characters.

Examples:

- `palindrome2 ("otto", 0, 3)`
  returns `true`.
- `palindrome2 ("Otto", 0, 3)`
  returns `true` because `'O'` is considered equal to `'o'`.
- `palindrome2 ("Madam,␣I'm␣Adam.", 0, 15)`
  returns `false` because `'M'` is not equal to `'.'`.

Part 2.3: Case-and-space-insensitive palindromes

Develop the directly recursive function:

```
bool palindrome3 (string text, int i, int j)
```

which decides if `text.at(i) ... text.at(j)` is a palindrome, but this time it should consider `'a'` also equal to `'A'`, `'b'` also equal to `'B'`, and so on for all the letter characters. Moreover, it should ignore all space characters (`'␣'`) and punctuation marks (`'.'`, `','`, `':'`, `';'`, `'''`, `'!'`, `'?'`, `'-'`).

Examples:

- `palindrome3 ("otto", 0, 3)`
  returns `true`.
- `palindrome3 ("Otto", 0, 3)`
  returns `true` because `'O'` is considered equal to `'o'`.
- `palindrome3 ("Madam,␣I'm␣Adam.", 0, 15)`
  returns `true` because case, space, and punctuation marks are ignored (`'.'` at the end), and `'M'` is considered equal to `'m'`.

Part 3: Matching characters in a string

Develop the directly recursive function:

```
bool match_chars (string chars, int i, string source, int j)
```

which decides if the characters `chars.at(i) ... chars.at(ssize(chars)-1)` occur in `source.at(j) ... source.at(ssize(source)-1)` in that order, but allowing to skip characters in source.

Examples:

- `match_chars ("abc", 0, "It␣is␣a␣bag␣of␣cards", 0)`
  returns `true` because all characters in `"abc"` occur in order: `"It␣is␣a␣bag␣of␣cards"`.
- `match_chars ("abc", 0, "It␣is␣a␣bag␣of␣books", 0)`
  returns `false` because character `'c'` does not occur: `"It␣is␣a␣bag␣of␣books"`.
- `match_chars ("abc", 0, "It␣is␣a␣classy␣bag", 0)`
  returns `false` because character `'c'` does not occur after `'b'`: `"It␣is␣a␣classy␣bag"`.

4 Products

As product-to-deliver you upload to Brightspace:

- “`main.cpp`” that you have created with solutions for each part of the assignment.
- “`main_test.cpp`” that has been extended with non-trivial unit tests for each new function that you have developed in “`main.cpp`”.

Deadline

Lab assignment: Friday, December 1, 2023, 23:59h