CISS350: Data Structures and Advanced Algorithms
Quiz q04

Name: YOUR EMAIL Score:

Q1. Write a function that accepts two C-strings `x` and `y` and computes the longest common substring in `x` and `y`:

```
void longest_common_substring(const char * x, const char * y, char * z);
```

The longest common substring is placed in `z`.

For instance if `x` is `"hello world"` and `y` is `"Hello world"`, then `z` is `"ello world"`. When `x` and `y` are empty strings or when `x` is `"abc"` and `y` is `"def"`, the longest common substring is `""`. If there are two longest common substrings, the first one that appears in `x` is used. For instance if `x` is `"hello!world."` and `y` is `"world.hello!"`, then there are two longest substrings that appear in `x` and `y`: `"hello!"` and `"world."`, both of length 6. However `"hello!"` appears earlier in `x`. Therefore `z` is `"hello!"`.

Just use the simplest algorithm you can come up with. You don't have to be super efficient. The "correct" algorithm is covered in CISS358. You may use the `strlen` function from the C-string library. You can add helper functions.

Recall that if `x` has type `const char *`, it means that `x` points to a `char` and cannot change the value that `x` points to. If you have a function of the form `void f(const char *)` then you can call `f("abc")`. However you cannot call `g("abc")` if `g` has a prototype of `void g(char *)`. This is because something like `"abc"` is considered a constant C-string.

TURN PAGE

Answer:

```
void longest_common_substring(const char * x, const char * y, char * z)
{

}
```

TURN PAGE FOR SPOILER ONLY IF YOU WANT SOME HELP

Spoiler

Forget about the code. Think first. This problem involves comparing substrings of two strings `x` and `y`. This means that if you need to systematically iterate over all substrings of `x` and all the substrings of `y`, and then compare the two substrings:

```
for s running over all substrings of x:
    for t running over all substrings of y:
        compare s and t
```

The question is then: What is a substring of `x`?

A substring of `x` can be described by `x`, `start`, and `end` where `start` and `end` are two index values. The substring is made up of the characters from index `start` up to but not including `end`, i.e., `x[start .. end - 1]`.

For instance if `x` is `"hello"` and `start` is 1 and `end` is 4, then the substring is `"ell"`. With the above idea you should be able to iterate over all substrings of `"hello"` and get `""`, `"h"`, `"he"`, `"hel"`, `"hell"`, `"hello"`, `""`, `"e"`, `"el"`, `"ell"`, `"ello"`, `""`, `"l"`, `"ll"`, etc. Believe it or not, you have seen this double for-loop before in my notes on for-loops from CISS240 (the second set of notes on for-loops). If you print the index values processed by this double for-loop you might recognize it.

Once you can do the above, for each substring of `x`, you iterate over all substrings of `y`, and compare the two substrings. There are some obvious optimizations of the above idea. But even with these optimizations, the algorithm is still going to be extremely slow. The "correct" algorithm for this very important algorithm will be covered in CISS358.

Also, instead of using two index values for `x`, you can use two pointers. In fact it's a good idea to use pointers just for the sake of reviewing pointers.

Here's a test code:

```
void test(const char * x, const char * y)
{
    char z[1024];
    std::cout << "Test [" << x << "] " << "[" << y << "]: ";
    longest_common_substring(x, y, z);
    std::cout << "[" << x << "] "
              << "[" << y << "] "
              << "[" << z << "]\n";
}

```

```
int main()
{
    test("abc", "def");
    test("", "def");
    test("abc", "");
    test("abc", "abc");
    test("abc", "Abc");
    test("Abc", "abc");
    test("abC", "abc");
    test("abc", "abC");
    test("abcd", "AbcD");
    test("AbcD", "abcd");
    test("abcd", "AAAAAAbcD");
    test("AAAAAAbcD", "abcd");
    test("abcd", "AAAAAAbcDDDDDDD");
    test("abcd", "AAAAAAbcDDDDDDD");
    test("AAAAAAbcDDDDDDD", "abcd");
    test("abcd", "AabAabAAabcDDDDDabcdDD");
    test("AabAabAAabcDDDDDabcdDD", "abcd");
    test("abcd", "AabcdAAAabcDDDDDabdDaD");
    test("abcd", "AAAAAAbcDDDDabDDD");
    test("abcd", "AAAAAAbcdDDDDabcDDD");
    test("AAAAAAbcdDDDDabcDDD", "!!!!!!!abcd@@@@@@@");

    return 0;
}
```

TURN PAGE FOR MORE SPOILER

More spoilers

Here's a function that will be helpful for printing while debugging:

```
// Print p[0..len-1] where len = q - p.
// ".............abcde.................."
//                ^    ^
//                |    |
//                p    q
void print(const char * p, const char * q)
{
    while (p < q)
    {
        std::cout << (*p);
        ++p;
    }
}
```

And here's a function that will compare two subarrays of characters:

```
// Return true if r[0 .. len-1] is p[0 .. len-1] where len = s - r.
//
// "...abcde.................."
//     ^
//     |
//     p
// ".............abcde.................."
//                ^    ^
//                |    |
//                r    s
bool isprefix(const char * p, const char * r, const char * s)
{
    while (r < s)
    {
        if (*p != *r)
        {
            return false;
        }
        ++p; ++r;
    }
    return true;
}
```

Instructions

In the file `thispreamble.tex` look for

```
\renewcommand\AUTHOR{}
```

and enter your email address:

```
\renewcommand\AUTHOR{jdoe5@cougars.ccis.edu}
```

(This is not really necessary since alex will change that for you when you execute `make`.) In your bash shell, execute "`make`" to recompile `main.pdf`. Execute "`make v`" to view `main.pdf`.

Enter your answers in `main.tex`. In the bash shell, execute "`make`" to recompile `main.pdf`. Execute "`make v`" to view `main.pdf`.

For each question, you'll see boxes for you to fill. For small boxes, if you see

```
1 + 1 = \answerbox{}.
```

you do this:

```
1 + 1 = \answerbox{2}.
```

`answerbox` will also appear in "true/false" and "multiple-choice" questions.

For longer answers that need typewriter font, if you see

```
Write a C++ statement that declares an integer variable name x.
\begin{answercode}
\end{answercode}
```

you do this:

```
Write a C++ statement that declares an integer variable name x.
\begin{answercode}
int x;
\end{answercode}
```

`answercode` will appear in questions asking for code, algorithm, and program output. In this case, indentation and spacing is significant. For program output, I do look at spaces and newlines.

For long answers (not in typewriter font) if you see

```
What is the color of the sky?
\begin{answerlong}
\end{answerlong}
```

you can write

```
What is the color of the sky?
\begin{answerlong}
The color of the sky is blue.
\end{answerlong}
```

A question that begins with "T or F or M" requires you to identify whether it is true or false, or meaningless. "Meaningless" means something's wrong with the question and it is not well-defined. Something like "$1 + 2 = 4$" is either true or false (of course it's false). Something like "$1+_2 = 4$?" does not make sense.

When writing results of computations, make sure it's simplified. For instance write 2 instead of $1 + 1$.

Higher level classes.

For students beyond 245: You can put LaTeX commands in `answerlong`.

More examples of meaningless statements: Questions such as "Is $42 = 1+_2$ true or false?" or "Is $42 = \{2\}^{\{3\}}$ true or false?" does not make sense. "Is $P(42) = \{42\}$ true or false?" is meaningless because $P(X)$ is only defined if X is a set. For "Is 1 + 2 + 3 true or false?", "1 + 2 + 3" is well–defined but as a "numerical expression", not as a "proposition", i.e., it cannot be true or false. Therefore "Is 1 + 2 + 3 true or false?" is also not a well-defined question.

More examples of simplification: When you write down sets, if the answer is $\{1\}$, do not write $\{1, 1\}$. And when the values can be ordered, write the elements of the set in ascending order. When writing polynomials, begin with the highest degree term.

When writing a counterexample, always write the simplest.