Q1. Explain the difference between greedy and non-greedy syntax with visual terms in as few words as possible. What is the bare minimum effort required to transform a greedy pattern into a non-greedy one? What characters or characters can you introduce or change?

The greedy match will try to match as many repetitions of the quantified pattern as possible. The non-greedy match will try to match as few repetitions of the quantified pattern as possible.

The usual rule for matching in REs is sometimes called “left-most longest“: when a pattern can be matched at more than one place within a string, the chosen match will be the one that starts at the earliest possible position within the string, and then extends as far as possible. Normally Perl pattern matching is greedy. By greedy, we mean that the parser tries to match as much as possible.

Q2. When exactly does greedy versus non-greedy make a difference?  What if you're looking for a non-greedy match but the only one available is greedy?

Greedy matching can return unexpected results when a regular expression contains repeating or optional elements. For example, when a regular expression contains a \* or + quantifier, greedy matching will match as much text as possible, while non-greedy matching will match as little text as possible.

Q3. In a simple match of a string, which looks only for one match and does not do any replacement, is the use of a nontagged group likely to make any practical difference?

Q4. Describe a scenario in which using a nontagged category would have a significant impact on the program's outcomes.

Q5. Unlike a normal regex pattern, a look-ahead condition does not consume the characters it examines. Describe a situation in which this could make a difference in the results of your programme.

A string that is made of six-to-ten word characters can be written like this: \A\w{6,10}\z

The \A anchor asserts that the current position is the beginning of the string. After matching the six to ten word characters, the \z anchor asserts that the current position is the end of the string.

Within a lookahead, this pattern becomes (?=\A\w{6,10}\z). This lookahead asserts: at the current position in the string, what follows is the beginning of the string, six to ten word characters, and the very end of the string.

We want to make this assertion at the very beginning of the string. Therefore, to continue building our pattern, we want to anchor the lookahead with an \A. There is no need to duplicate the \A, so we can take it out of the lookahead. Our pattern becomes:

\A(?=\w{6,10}\z)

So far, we have an expression that validates that a string is entirely composed of six to ten word characters. Note that we haven't matched any of these characters yet: we have only looked ahead. The current position after the lookahead is still the beginning of the string. To check the other conditions, we just add lookaheads.

Q6. In standard expressions, what is the difference between positive look-ahead and negative look-ahead?

Positive lookahead: (?=«pattern») matches if pattern matches what comes after the current location in the input string.

Negative lookahead: (?!«pattern») matches if pattern does not match what comes after the current location in the input string.

Q7. What is the benefit of referring to groups by name rather than by number in a standard expression?

The advantage to named groups is that it adds readability and understandability to the code, so that you can easily see what part of a regular expression match is being referenced.

Q8. Can you identify repeated items within a target string using named groups, as in "The cow jumped over the moon"?

using std::regex;

using std::regex\_replace;

using std::string;

regex reg1("([A-Za-z]+) \\1"); // Find double word.

string replacement = "$1"; // Replace with one word.

string target = "The cow cow jumped over the the moon.";

string result = regex\_replace(target, reg1, replacement);

std::cout << result << std::endl;

Q9. When parsing a string, what is at least one thing that the Scanner interface does for you that the re.findall feature does not?

Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found.

Q10. Does a scanner object have to be named scanner?

Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings. It is the easiest way to read input in a Java program, though not very efficient if you want an input method for scenarios where time is a constraint like in competitive programming.