Using Regular Expressions

By Dan[Popovici] & mariusmuja — *topcoder members*

Introduction
A regular expression is a special string that describes a search pattern. Many of you have surely seen and used them already when typing expressions like ls(or dir) *.txt , to get a list of all the files with the extension txt. Regular expressions are very useful not only for pattern matching, but also for manipulating text. In SRMs regular expressions can be extremely handy. Many problems that require some coding can be written using regular expressions on a few lines, making your life much easier.

(Not so) Formal Description of Regular Expressions
A regular expression(regex) is one or more non-empty branches, separated by '|'. It matches anything that matches one of the branches. The following regular expression will match any of the three words "the","top","coder"(quotes for clarity).

```
REGEX is : the|top|coder
INPUT is : Marius is one of the topcoders.
Found the text "the" starting at index 17 and ending at index 20.
Found the text "top" starting at index 21 and ending at index 24.
Found the text "coder" starting at index 24 and ending at index 29.
```

A branch is one or more pieces, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an atom possibly followed by a '*', '+', '?', or bound. An atom followed by '*' matches a sequence of 0 or more matches of the atom. An atom followed by `+' matches a sequence of 1 or more matches of the atom. An atom followed by `?' matches a sequence of 0 or 1 matches of the atom.

The following regular expression matches any successive occurrence of the words 'top' and 'coder'.

```
REGEX is: (top|coder)+
INPUT is: This regex matches topcoder and also codertop.
Found "topcoder" starting at index 19 and ending at index 27.
Found "codertop" starting at index 37 and ending at index 45.
```

A bound is '{' followed by an unsigned decimal integer, possibly followed by ',' possibly followed by another unsigned decimal integer, always followed by '}'.If there are two integers, the first may not exceed the second. An atom followed by a bound containing one integer i and no comma matches a sequence of exactly i matches of the atom. An atom followed by a bound containing one integer i and a comma matches a sequence of i or more matches of the atom. An atom followed by a bound containing two integers i and j matches a sequence of i through j (inclusive) matches of the atom.
The following regular expression matches any sequence made of '1′s having length 2,3 or 4 .

```
REGEX is: 1{2,4}
INPUT is: 101 + 10 = 111 , 11111 = 10000 + 1111
Found the text "111" starting at index 11 and ending at index 14.
Found the text "1111" starting at index 17 and ending at index 21.
Found the text "1111" starting at index 33 and ending at index 37.
```

One should observe that, greedily, the longest possible sequence is being matched and that different matches do not overlap. An atom is a regular expression enclosed in '()' (matching a match for the regular expression), a bracket expression (see below), '.' (matching any single character), '^' (matching the null string at the beginning of a line), '$' (matching the null string at the end of a line), a `\' followed by one of the characters `^.[$()|*+?{\' (matching that character taken as an ordinary character) or a single character with no other significance (matching that character). There is one more type of atom, the back reference:

`\' followed by a non-zero decimal digit d matches the same sequence of characters matched by the d-th parenthesized subexpression (numbering subexpressions by the positions of their opening parentheses, left to right), so that (e.g.) `([bc])\1′ matches `bb' or `cc' but not `bc'.

The following regular expression matches a string composed of two lowercase words separated by any character.

```
Current REGEX is: ([a-z]+).\1
Current INPUT is: top-topcoder|coder
I found the text "top-top" starting at index 0 and ending at index 7.
I found the text "coder|coder" starting at index 7 and ending at index 18.
```

A bracket expression is a list of characters enclosed in '[]'. It normally matches any single character from the list. If the list begins with '^', it matches any single character not from the rest of the list. If two characters in the list are separated by `-', this is shorthand for the full range of characters between those two inclusive (e.g. '[0-9]' matches any decimal digit). With the exception of ']','^','-' all other special characters, including `\', lose their special significance within a bracket expression.

The following regular expression matches any 3 character words not starting with 'b','c','d' and ending in 'at'.

```
Current REGEX is: [^b-d]at
Current INPUT is: bat
No match found.

Current REGEX is: [^b-d]at
Current INPUT is: hat
I found the text "hat" starting at index 0 and ending at index 3.
```

This example combines most concepts presented above. The regex matches a set of open/close pair of html tags.

```
REGEX is: <([a-zA-Z][a-zA-Z0-9]*)(()| [^>]*)>(.*)</\1>
INPUT is: <font size="2">Topcoder is the</font> <b>best</b>
Found "<font size="2">Topcoder is the</font>" starting at index 0 and ending at index 3
Found "<b>best</b>" starting at index 38 and ending at index 49.
```

([a-zA-Z][a-zA-Z0-9]*) will match any word that starts with a letter and continues with an arbitrary number of letters or digits.
(()| [^>]*) will match either the empty string or any string which does not contain '>' .
\1 will be replaced using backreferencing with the word matched be ([a-zA-Z][a-zA-Z0-9]*)

The above description is a brief one covering the basics of regular expressions. A regex written following the above rules should work in both Java(>=1.4) and C++(POSIX EXTENDED). For a more in depth view of the extensions provided by different languages you can see the links given in the References section.

Using regular expressions

In java
In java(1.4 and above) there is a package "java.util.regex" which allows usage of regular expressions.

This package contains three classes : Pattern, Matcher and PatternSyntaxException.

- A Pattern object is a compiled representation of a regular expression. The Pattern class provides no public constructors. To create a pattern, you must call one of its public static compile methods, both of which will return a Pattern object.
- A Matcher object is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. You obtain a Matcher

object by calling the public matcher method on a Pattern object.
- A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

Example(adapted from[4]):

```
Pattern pattern;
Matcher matcher;
pattern = Pattern.compile(<REGEX>);
matcher = pattern.matcher(<INPUT>);
boolean found;
while(matcher.find()) {
  System.out.println("Found the text \"" + matcher.group() +  "\" starting at index " +
      " and ending at index " + matcher.end() + ".");
  found = true;
}

if(!found){
  System.out.println("No match found.");
}
```

Java also offers the following methods in the String class:

- boolean matches(String regex) (returns if the current string matches the regular expression regex)
- String replaceAll(String regex, String replacement) (Replaces each substring of this string that matches the given regular expression with the given replacement.)
- String replaceFirst(String regex, String replacement) (Replaces the first substring of this string that matches the given regular expression with the given replacement.)
- String[] split(String regex) (Splits this string around matches of the given regular expression)

In C++
Many Topcoders believe that regular expressions are one of Java's main strengths over C++ in the arena. C++ programmers don't despair, regular expressions can be used in C++ too.

There are several regular expression parsing libraries available for C++, unfortunately they are not very compatible with each other. Fortunately as a Topcoder in the arena one does not have to cope with all this variety of "not so compatible with one another" libraries. If you plan to use regular expressions in the arena you have to choose between two flavors of regex APIs: POSIX_regex and GNU_regex. To use these APIs the header file "regex.h" must be included. Both of these work in two steps – first there is a function call that compiles the regular expression, and then there is a function call that uses that compiled regular expression to search or match a string.

Here is a short description of both of these APIs, leaving it up to the coder to choose the one that he likes the most.

POSIX_regex
Includes support for two different regular expression syntaxes, basic and extended. Basic regular expressions are similar to those in ed, while extended regular expressions are more like those in egrep, adding the '|', '+' and '?' operators and not requiring backslashes on parenthesized subexpressions or curly-bracketed bounds. Basic is the default, but extended is prefered.

With POSIX, you can only search for a given regular expression; you can't match it. To do this, you must first compile it in a pattern buffer, using `regcomp'. Once you have compiled the regular expression into a pattern buffer you can search in a null terminated string using 'regexec'. If either of the 'regcomp' or 'regexec' function fail they return an error code. To get an error string corresponding to these codes you must use 'regerror'. To free the allocated fields of a pattern buffer you must use 'regfree'.

For an in depth description of how to use these functions please consult [2] or [3] in the References section.

Example:

Here is a small piece of code showing how these functions can be used:

```
regex_t reg;

string pattern = "[^tpr]{2,}";
string str = "topcoder";

regmatch_t matches[1];

regcomp(&reg,pattern.c_str(),REG_EXTENDED|REG_ICASE);

if (regexec(&reg,str.c_str(),1,matches,0)==0) {
  cout << "Match "
  cout << str.substr(matches[0].rm_so,matches[0].rm_eo-matches[0].rm_so)
  cout << " found starting at: "
  cout << matches[0].rm_so
  cout << " and ending at "
  cout << matches[0].rm_eo
  cout << endl;
} else {
  cout << "Match not found."
  cout << endl;
}
regfree(&reg);
```

GNU_regex
The GNU_regex API has a richer set of functions. With GNU regex functions you can both match a string with a pattern and search a pattern in a string. The usage of these functions is somehow similar with the usage of the POSIX functions: a pattern must first be compiled with 're_compile_pattern', and the pattern buffer obtained is used to search and match. The functions used for searching and matching are 're_search' and 're_match'. In case of searching a fastmap can be used in order to optimize search. Without a fastmap the search algorithm tries to match the pattern at consecutive positions in the string. The fastmap tells the algorithm what the characters are from which a match can start. The fastmap is constructed by calling the 're_compile_fastmap'. The GNU_regex also provides the functions 're_search2' and 're_match2' for searching and matching with split data. To free the allocated fields of a pattern buffer you must use 'regfree'.

For an in-depth description of how to use these functions please consult [3].

Example:

```
string pattern = "([a-z]+).\\1";
string str = "top-topcoder|coder";

re_pattern_buffer buffer;
  char map[256];

buffer.translate = 0;
buffer.fastmap = map;
buffer.buffer = 0;
buffer.allocated = 0;

re_set_syntax(RE_SYNTAX_POSIX_EXTENDED);
const char* status = re_compile_pattern(pattern.c_str(),pattern.size(),&buffer);
if (status) {
    cout << "Error: " << status << endl;
}
re_compile_fastmap(&buffer);

struct re_registers regs;
int ofs = 0;
if (re_search(&buffer,str.c_str(),str.size(),0,str.size(),&regs)!=-1) {
  cout << "Match "
  cout << str.substr(regs.start[0],regs.end[0]-regs.start[0])
  cout << " found starting at: "
```

```
    cout << regs.start[0]
    cout << " and ending at "
    cout << regs.end[0]
    cout << endl;
  } else {
    cout << "Match not found."
    cout << endl;
  }
  regfree(&buffer);
```

Real SRMs Examples
The following examples are all written in Java for the sake of clarity. A C++ user can use the POSIX or the GNU regex APIs to construct functions similar to those available in Java(replace_all, split, matches).

CyberLine (SRM 187 div 1, level 1)

```
import java.util.*;
public class Cyberline
{
  public String lastCyberword(String cyberline)
  {
    String[]w=cyberline.replaceAll("-","")
          .replaceAll("[^a-zA-Z0-9]"," ")
          .split(" ") ;
    return w[w.length-1];
  }
}
```

UnLinker (SRM 203 div 2, level 3)

```
import java.util.*;
public class UnLinker
{
  public String clean(String text)
  {
    String []m = text.split("((http://)?www[.]|http://)[a-zA-Z0-9.]+[.](com|org|edu|inf
    String s = m[0] ;
    for (int i = 1 ; i < m.length ; ++i)
      s = s + "OMIT" + i + m[i] ;
    return s ;
  }
}
```

CheatCode (SRM 154 div 1, level 1)

```
import java.util.*;
public class CheatCode {
  public int[] matches(String keyPresses, String[] codes) {
    boolean []map = new boolean[codes.length] ;
    int count = 0 ;
    for (int i=0;i<codes.length; ++i)
    {
      String regex = ".*" ;
      for (int j=0; j<codes[i].length(); ) {
        int k = 1;
        while ((j+k)<codes[i].length() && codes[i].charAt(j+k)==codes[i].charAt(j)) k++
        regex = regex + codes[i].charAt(j) + "{"+k+",}";
        j+=k;
      }

      regex = regex + ".*" ;
      if (keyPresses.matches(regex))
      {
        map[i] = true ;
        count++ ;
      }
    }
    int []res = new int[count] ;
    int j=0;
    for (int i= 0 ; i < codes.length; ++i)
      if(map[i] == true)
        res[j++]=i ;
    return res ;
  }
}
```

References

1. The regex(7) linux manual page
2. The regex(3) linux manual page
3. http://docs.freebsd.org/info/regex/regex.info.Programming_with_Regex.html
4. http://www.regular-expressions.info/
5. http://java.sun.com/docs/books/tutorial/extra/regex/

More Resources

Member Tutorials

Read more than 40 data science tutorials written by topcoder members.

Problem Set Analysis

Read editorials explaining the problem and solution for each Single Round Match (SRM).

Data Science Guide

New to topcoder's data science track? Read this guide for an overview on how to get started in the arena and how competitions work.

Help Center

Need specifics about the process or the rules? Everything you need to know about competing at topcoder can be found in the Help Center.

Member Forums

Join your peers in our member forums and ask questions from the real experts - topcoder members!