VT ACM ICPC Handbook

ACM Programming Team 2012

September 21, 2012

Contents

Preface

This book is intended as a reference, to be used both during the competition as well in preparation for it.

It is hosted on github at https://github.com/VTACMProgrammingTeam/ICPCHandbook. If you wish to contribute, please send email to godmar@gmail.com

Chapter 1

Gotchas

Common mistakes and idiosyncrasies observed in the judge input and specification of various problems posed at competitions.

1. **Judge input not terminated as required**. Typically, the problem states that there's some way to identify the end of input without having to rely on EOF. We've observed judge input, however, where EOF terminated the input. You should try to write your input loop such that your solution works whether the input is terminated by EOF or by the specified end-of-input delimiter. This strategy may allow you to submit a correct solution even before the mistake is discovered (and may even lead to a delay in when it's discovered that would benefit your team.) Seen in 2006/E Marbles 4.2.1.

2. **Trailing Spaces**. In problems that state "there is one word per line" we have observed trailing spaces which must be trimmed. Advice: always use String.trim(), unless the spaces are significant, which we have not seen anywhere. Seen in 2007/D Witness 4.3.2.

3. **Leading Spaces**. In some problems, (insignificant) leading spaces may occur. The catch here is that naive splitting without trimming may produce an empty string in the first position. See bsh output below:

   ```
   % System.out.println(Arrays.toString("  word1  word2  ".split("\\s+")));
   [, word1, word2]
   ```

   Seen in 2011/B Raggedy 4.5.1.

Chapter 2

String Processing

2.1 Regular Expressions

2.1.1 Regular Expressions

2.1.2 Zero-width Lookahead Technique

In some problems (notably, 2007/B/Mobile 4.3.1, 2007/D/Witness 4.3.2, and 2008/G/Stems 4.4.1), the string and/or input handling of these problems can greatly benefit from using zero-width positive lookahead/lookbehind regular expressions.

To understand how they work, consider how java.util.Scanner works. By default, a Scanner splits the input stream into tokens using a delimiter pattern. The default delimiter pattern is one or more whitespaces (written as `\p{javaWhitespace}` or, when embedded in Java code, as `"\\p{javaWhitespace}+"`). The input characters that are matched by the delimiter itself are consumed by the Scanner there is no way to retrieve them.

In some cases, whitespace is not a suitable delimiter. Suppose you're asked to parse an arithmetic expression that uses +, -, *, and /. Whitespaces are optional, so both `1+1` and `1 + 1` as well as `1 +1` are valid expressions. If you made the operators '+', '-' etc. delimiters (perhaps in addition to whitespace), a Scanner would retrieve '1' and '1', but there would be no way to retrieve the '+' so you couldnt distinguish '1+1' and '1-1'. Instead, use lookaround matching by adding a zero-width delimiter that matches before or after a +, -, *, or /. "Zero-width" here means that although the delimiter matches (and thus causes the Scanner to stop and return what it has read so far!), it does not consume any characters. Thus, the scanner will stop,

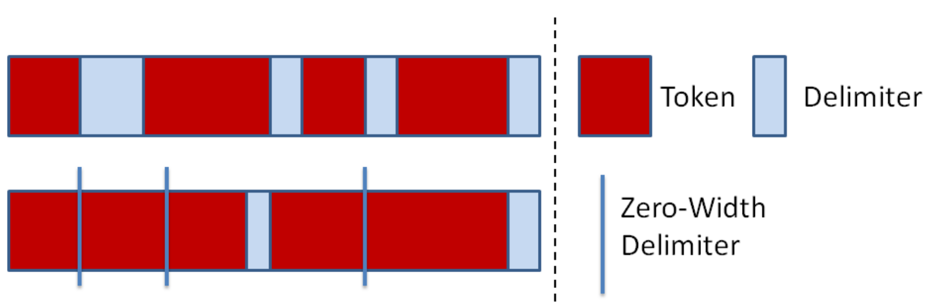

Figure 2.1: Lookahead Splitting. The top shows a traditional scanner/split which consumes delimiters. The bottom shows a scanner using delimiter expressions that may or may not consume characters.

but the delimiter (which the Scanner swallows) has zero width therefore, the characters are returned as part of the previous token. In this example, s.next() would return '+'.

Figure 2.1 shows a traditional scanner (top) and a scanner that uses both consuming and non-consuming delimiters (bottom): Note that if the delimiter used by the scanner does not consume any characters, the scanner will return the entire input stream. This is very useful if you need to manipulate a stream without losing any characters.

The idea to use String.format to turn any regular expression into a zero-width lookahead or lookbehind delimiter is taken from here.

Note that this technique can be used with a java.util.Scanner object (via useDelimiter), but also in all other functions that use regular expressions as delimiters, notably String.split().

Finally, note that you cannot use some regular expressions to describe zero-width delimiters. Notably, **expressions using repetition (* or +) cannot be used.**

Code Example. The following program shows some of the applications of this style of matching. These examples include:

- Arithmetic expressions with optional whitespace
- S-Expressions with optional whitespace before and after ( )
- Finding words in a sentence
- Finding sentences in a paragraph

```
/**
 * Examples of zero-width lookahead/lookbehind splitting.
 * For each example, study the input and output.
 *
 * http://stackoverflow.com/questions/2206378/how-to-split-a-string-but-also-keep-the-deli
 *
 * @author Godmar <godmar@gmail.com>
 */

import java.util.*;

public class Lookaround
{
    /* String.format patterns for ease of use */
    final static String MATCH_BEFORE_OR_AFTER = "((?<=%1$s)|(?=%1$s))";
    final static String MATCH_AFTER = "(?<=%1$s)";
    final static String MATCH_BEFORE = "(?=%1$s)";

    static void example(String input, String delim) {
        Scanner s = new Scanner(input).useDelimiter(delim);
        System.out.println("Delimiter: " + delim);
        System.out.println("Input: " + input);
        System.out.print("Output: '" + s.next() + "'");
        while (s.hasNext()) {
            System.out.print(", '" + s.next() + "'");
        }
        System.out.println();
        System.out.println();
    }
```

```
    public static void main(String []av) {
        // match right before or after +, -, *, /
        // consumes nothing
        String delim = String.format(MATCH_BEFORE_OR_AFTER, "[\\+\\-\\*\\/]");
        example("10+21*32-43/5+60", delim);

        // matches whitespace or  right before or after +, -, *, /
        // consumes whitespace - but no +/-/*//
        delim = "\\p{javaWhitespace}|"
            + String.format(MATCH_BEFORE_OR_AFTER, "[\\+\\-\\*\\/]");
        example("10 + 21*32 -43 / 5+60", delim);

        // match whitespace or right before or after ( )
        // consumes whitespace - but does not consume ( or )
        delim = "\\p{javaWhitespace}|"
            + String.format(MATCH_BEFORE_OR_AFTER, "[\\(\\)]");
        example("((F1 (A1 1 2 3)) (F2 (A2 4 5) ( A3 5 )))", delim);

        // match at word boundaries
        // consumes nothing
        delim = "\\b";
        example("This text has words, and some---wrongly set---punctuation characters."
            +"Note that words can contain alphanumericals such as babe1234 and"
            +" underscores.  Underscores do_not_form_a_word_boundary.", delim);

        // match at before or after anything that is not alphanumeric
        // (which matches every boundary except within a word of alphanumeric chars.)
        // consumes nothing
        delim = String.format(MATCH_BEFORE_OR_AFTER, "[^A-Za-z0-9]");
        example("This delimiter identifies word boundaries, but unlike the previous "
            +"one returns all characters between words as individual tokens. "
            +"Underscores do_form_a_word_boundary with this delimiter.",
             delim);

        // match after ., !, ? or empty line.
        delim = String.format(MATCH_AFTER, "[\\.\\!\\?]|\\n\\n");
        example("This matches sentences. And questions too?  Yes!   "
            +"Even breaks between\n\nparagraphs.", delim);
    }
}
```

2.1.3 NFA Simulation

The Regex engine in Java does not convert to a Thompson-DFA; it uses a backtracking algorithm to find out if a regular expression matches a string. This leads to pathological cases with exponential runtime increase, particularly when the regular expression contains a large number of Kleene stars.

In those situations, it may be helpful to construct your own mini-regexp interpreter by building and simulating an NFA (nondeterministic finite automaton).

Example problem is NCPC 2011/E where the input are globs such as `*a*a*a*a` that should be matched against filenames. Figure 2.2 shows an example of how to construct such a NFA. In an NFA there may be multiple transitions labeled with the same symbol: for instance, there's a transition labeled 'b' from state 0 to state 1, but there is also a transition labeled 'b' from state 0 to state 0. For the input string `abc`, the 'b'

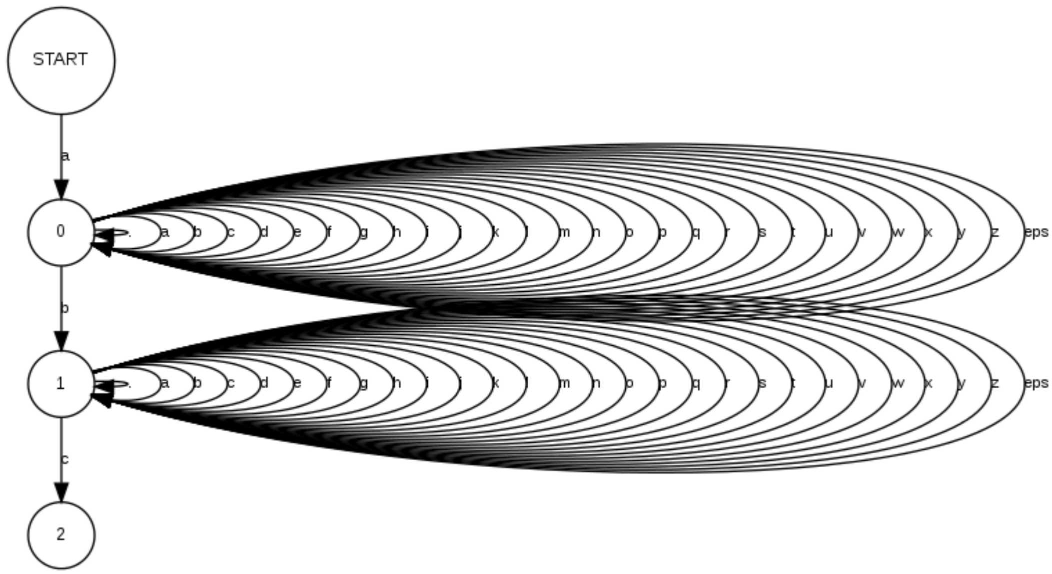

Figure 2.2: NFA for regular expression `a.*b.*c` representing glob `a*b*c` over alphabet of lowercase letters and period (.)

would transition into state 1, whereas for the input string `abbc`, the first 'b' would transition into state 0, the second into state 1.

Of course, we don't know which it's actually going to be - a NFA, in its theoretical formulation, is defined to oracle-like pick the correct transition. That's why we simulate it by simply keeping track of all possible ("active") states the NFA might be in after each symbol. This is done using a set (HashSet or BitSet if the states are nicely numbered). For each input symbol, we compute the possible set of successor states based on the current set of active states. If after the string has been exhausted the goal state is in the set of active states the string is matched. A Python solution is shown below for succinctness.

```python
import sys, string
from collections import defaultdict

def NextLine(): return sys.stdin.readline().rstrip()

class State:
    def __init__(self):
        self.transitions = defaultdict(set)

    def addTransition(self, symbol, destState):
        self.transitions[symbol].add(destState)

pattern = NextLine()
firststate = laststate = State()

for p in pattern:
    if p == '*':
        # wildcard - add self transitions for all
        # input characters and .
        for l in string.lowercase + '.':
            laststate.addTransition(l, laststate)
    else:
```

```
        # add transition to next state
        nextstate = State()
        laststate.addTransition(p, nextstate)
        laststate = nextstate

# lastState is now goal state for a full match

N = int(NextLine())
for i in range(N):
    fname = NextLine()
    # simulate NFA
    activestates = set([firststate])
    for f in fname:
        activestates = set(d for s in activestates \
                                 for d in s.transitions[f])

    if laststate in activestates:
        print fname
```

2.2 Parsing

2.2.1 Recursive Descent

Chapter 3

Geometry

3.1 Basics

A determinant of a $2x2$ matrix is defined as

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

3.2 java.awt.geom

The `java.awt.geom` and `java.awt` packages have, albeit limited, facilities for geometric problems. There are classes to represent shapes - see java.awt.Shape, including lines, ellipses, rectangles and some curves.

- ”is contained in”. java.awt.geom.Shape provide a contains() method to test if a point is contained in a shape. Contains() returns true if the point is in the interior, and false if the point is outside the shape. However, it **may return true or false if the point is on the shape boundary.**
- ”intersects.” Tests if a shape intersects with a rectangle. Can also test if two lines or line segments intersect, but cannot find the point of intersection.

3.3 Coordinate Geometry

3.3.1 Line/Line Intersection

$$P_x = \frac{\begin{vmatrix} \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} & \begin{vmatrix} x_1 & 1 \\ x_2 & 1 \end{vmatrix} \\ \\ \begin{vmatrix} x_3 & y_3 \\ x_4 & y_4 \end{vmatrix} & \begin{vmatrix} x_3 & 1 \\ x_4 & 1 \end{vmatrix} \end{vmatrix}}{\begin{vmatrix} \begin{vmatrix} x_1 & 1 \\ x_2 & 1 \end{vmatrix} & \begin{vmatrix} y_1 & 1 \\ y_2 & 1 \end{vmatrix} \\ \\ \begin{vmatrix} x_3 & 1 \\ x_4 & 1 \end{vmatrix} & \begin{vmatrix} y_3 & 1 \\ y_4 & 1 \end{vmatrix} \end{vmatrix}} \quad P_y = \frac{\begin{vmatrix} \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} & \begin{vmatrix} y_1 & 1 \\ y_2 & 1 \end{vmatrix} \\ \\ \begin{vmatrix} x_3 & y_3 \\ x_4 & y_4 \end{vmatrix} & \begin{vmatrix} y_3 & 1 \\ y_4 & 1 \end{vmatrix} \end{vmatrix}}{\begin{vmatrix} \begin{vmatrix} x_1 & 1 \\ x_2 & 1 \end{vmatrix} & \begin{vmatrix} y_1 & 1 \\ y_2 & 1 \end{vmatrix} \\ \\ \begin{vmatrix} x_3 & 1 \\ x_4 & 1 \end{vmatrix} & \begin{vmatrix} y_3 & 1 \\ y_4 & 1 \end{vmatrix} \end{vmatrix}}$$

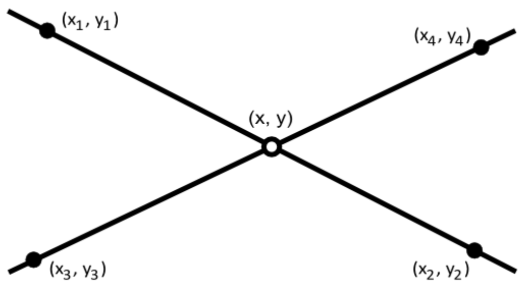

Figure 3.1: Line line intersection

The determinants can be written out as:

$$(P_x, P_y) = \Bigg(\frac{(x_1y_2 - y_1x_2)(x_3 - x_4) - (x_1 - x_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}, \frac{(x_1y_2 - y_1x_2)(y_3 - y_4) - (y_1 - y_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}\Bigg)$$

Source: http://en.wikipedia.org/wiki/Line-line_intersection.

Notes

- Does not handle parallel or coincident lines: Denominator will be zero:

$$(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4) = 0$$

- Does not handle if lines are each others' normal (i.e., at a right angle). If line is horizontal ($y_1 = y_2$ or $y_3 = y_4$), and the other vertical ($x_1 = x_2$ or $x_3 = x_4$) denominator will also be a 0 determinant, but the lines will intersect. Handle as special case if problem allows it.
- Intersection point may be outside the given segments.
- If you only need to know if two lines intersect, but not where, use java.awt.geom.Line2D.intersects.

Code This code is from a solution to 2011/F (Section 4.5.2) where the parallel and rectangular cases do not occur. (TBD: provide complete implementation.)

```
static double det(double x1, double y1, double x2, double y2) {
    return x1 * y2 - y1 * x2;
}

static Point2D.Double intersects(Point2D.Double p1, Point2D.Double p2,
                                 Point2D.Double p3, Point2D.Double p4) {
    double d = det(p1.x - p2.x, p1.y - p2.y, p3.x - p4.x, p3.y - p4.y);
    double x12 = det(p1.x, p1.y, p2.x, p2.y);
    double x34 = det(p3.x, p3.y, p4.x, p4.y);

    // assert d != 0 (lines are known to intersect and are not at right angle)
    double x = det(x12, p1.x-p2.x, x34, p3.x-p4.x) / d;
    double y = det(x12, p1.y-p2.y, x34, p3.y-p4.y) / d;
    return new Point2D.Double(x, y);
}
```

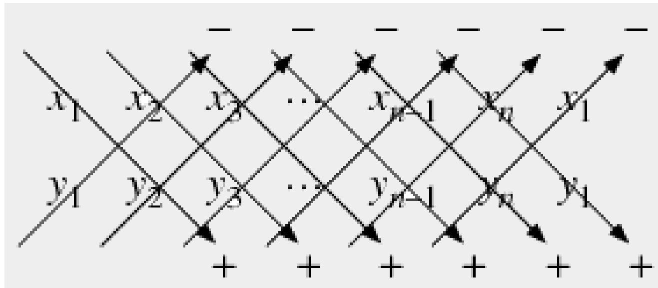

Figure 3.2: Line line intersection

3.3.2 Area of a Polygon

The signed area of a planar non-self-intersecting polygon with vertices $(x_1, y_1), \ldots, (x_n, y_n)$ is

$$A = \frac{1}{2}\left(\begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} + \begin{vmatrix} x_2 & x_3 \\ y_2 & y_3 \end{vmatrix} + \ldots + \begin{vmatrix} x_n & x_1 \\ y_n & y_1 \end{vmatrix} \right)$$

Figure 3.2 shows how to multiply this out

$$A = \frac{1}{2}\left(x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + \ldots + +x_{n-1}y_n - x_ny_{n-1} + x_ny_1 - x_1y_n\right)$$

(Source: Mathworld [1])

Notes

- Works for any simple polygon (concave or convex)
- Does not work for complex polygons (when any edges intersect)
- Points **must be ordered** if polygon has more than 3 vertices, or output is junk.
- A is positive if points are in counterclockwise order, negative if points are in clockwise order. See the use of Math.abs() in code below.
- Triangle and any Quadrilateral are, of course, just special cases. For triangles, order does not matter.

Code

```
static double areaPolygon(Point2D.Double p[]) {
    double area = 0.0;
    int n = p.length;
    for (int i = 0; i < n; i++) {
        area += p[i].x * p[(i+1) % n].y - p[i].y * p[(i+1) % n].x;
    }
    return Math.abs(area/2.0);
}
```

Chapter 4

Mid-Atlantic Problem Sets

This chapter contains some notes about the problems occurring in the Mid-Atlantic problem set. We focus on this corpus in particular because there are recurring themes since the problems have been created by the same person (or team) for multiple years.

4.1 2005

http://midatl.radford.edu/docs/pastProblems/05contest/MidAtlantic2005.pdf

4.1.1 C Extrusion

Straightforward application of polygon area formula, see Section 3.3.2.

4.2 2006

http://midatl.radford.edu/docs/pastProblems/06contest/MidAtlantic2006.pdf

4.2.1 E Marbles

A simple state space exploration problem solvable with straightforward BFS exploration. Catch: judge input data missed the ”0 0 0” line.

4.3 2007

http://midatl.radford.edu/docs/pastProblems/07contest/MidAtlantic2007.pdf

4.3.1 B Mobiles Alabama

Lexical analysis benefits from zero-width lookaround 2.1.2, although simpler solutions (replacing ’(’ and ’)’ with ’ (’ and ’) ’ before splitting on whitespace may work, too. Recursive descent parsing should be used to analyse the syntactical structure of the input.

4.3.2 D Witness Redaction

This problem can be solved with regular expressions and zero-width lookaround splitting. See Section 2.1.2.

4.4 2008

http://midatl.radford.edu/docs/pastProblems/08contest/MidAtlantic2008.pdf

4.4.1 G Stems Sell

Can be solved with regular expressions.

http://midatl.radford.edu/docs/pastProblems/08contest/JudgingData/G-stems/ appears broken, even on ICPC site.

4.5 2011

http://midatl.radford.edu/docs/pastProblems/11contest/MidAtlantic2011.pdf

4.5.1 B Raggedy, Raggedy

This problem can be solved using dynamic programming. Note that there may be leading spaces on some input lines.

4.5.2 F Line of Sight

Straightforward application of area of polygon 3.3.2 and line intersection 3.3.1. Note that parameters of the problem even exclude corner cases for line intersection (e.g. parallel lines, right angles).

Bibliography

[1] Eric W. Weisstein. Polygon area. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PolygonArea.html.

Index