University of Arkansas
Department of Computer Science and Computer Engineering
2012 High School Programming Contest
Problems

Wing Ning Li Hung Nguyen Adam Higgins Paul Martin

Revision Date: March 10, 2012

Please carefully read the output specification. Submitted solutions are graded by a differential program, i.e., the output produced by a submission will be compared *byte-for-byte* with the output of the judges' solutions. Thus, even though submitted solutions may produce output that is *technically* correct, it will be counted as incorrect if these specifications are not followed exactly.

For all problems, solutions receive input from standard input (*stdin*) and produce output on standard output (*stdout*).

Restricting input and output to stdin and stdout does not prevent a program from processing files. Let `Main` be the program name, `Input` be the input file name, and `Output` be the output file name. The following command at the terminal allows the program to read and write the input and output files respectively:

```
./Main < Input > Output      # for C or C++, assuming Main is the executable
java Main < Input > Output   # for Java, assuming Main is a class file
```

For all problems, an *integer* is defined as an optional minus sign followed by one digit from the set 1..9 followed by zero or more digits from the set 0..9. `3`, `-21`, and `309` are integers; `0123`, `- 4`, and `+38` are not.

A *positive integer* is an integer whose value is greater than 0. You may assume the value of a positive integer is no more than $2^{31} - 1$, or $2,147,483,647$.

A *string*'s maximum length is 16,384.

For all problems, a *line* is defined as zero or more characters followed by a newline ('\n') character. Unless otherwise directed, a line of output should have no leading or trailing whitespace. A program producing the integer 27 on one line of output should consist of the character '2' followed by the character '7' followed by the newline ('\n') character.

Input for a problem consists of multiple test cases; each test case is followed by a blank line. You must read input until the *end-of-file*. Refer to the code for the Warm Up problem for hints and ideas for handling input properly.

For each test case, your program must output the result following the specified format in each problem. Output a blank line after the output of each test case. Refer to the code for the Warm Up problem again for hints and ideas for handling output properly.

1 Letter Grade Assignment

Graded tests are given numerical scores such as 95 or 74. Letter grades such as A or C are given to students after they complete a course. Write a program to assign a letter grade for a given numerical test score based on the following scheme:

90-100	is A
80-89	is B
70-79	is C
60-69	is D
0-59	is F

Since numerical scores are entered by human, it is possible that an integer greater than 100 or less than 0 is used as a numerical score. In this case, the program should output ERROR.

input Each test case has a line containing an integer, which is the numerical score. *There is always a blank line after every test case.*

output For each test case, output two lines: the first line containing the letter grade of the numerical test score or ERROR, *and a blank line as second.*

Example:

Input

```
95

74

101

```

Output

```
A

C

ERROR
```

2 Letter Grade Assignment with Leniency

Let us use the same conversion scheme from numerical test score to letter grade as in the previous problem. In a semester, four tests are given. A student may drop the the lowest test score of the four to improve the grade. The remaining three scores are used to determine the final letter grade based on the following rules:

1. A student gets A if two of the three numerical scores result in A and the other results in C or better; or one of the three numerical scores result in A and the other two at least B.
2. A student gets B if two of the three numerical scores result in B and the other results in D or better; or one of the three numerical scores results in B and the other two at least C; or two of the three numerical scores result in A and at least B; or the three numerical scores result in A, at least C, and at least D.
3. A student gets C if two of the three numerical scores result in at least C; or one of the three numerical scores results in C and the other two at least D; or two of the three numerical scores result in B and at least D; or one of the three numerical scores results in A.
4. A student gets D if two of the three numerical scores result in at least D; or one of the three numerical scores result in C or better.
5. A student gets F if the above cannot be applied.

Write a program to assign an as high as possible letter grade to a student based on the above rules. Again if at least one the four numerical scores is great than 100 and less than 0, the program outputs ERROR.

input Each test case has a line containing four integers separated by one or more spaces for the four numerical test score. *There is always a blank line after every test case.*

output For each test case, output two lines: the first line containing the letter grade of the numerical scores or ERROR, *and a blank line as second.*

Example:

Input

```
90 82 91 63

95 45 90 30

120 30 76 82
```

Output

```
A

B

ERROR
```

3 Public keys in RSA

Two positive integers A and B are said to be relatively prime to each other if the only integer that can divide both integers evenly is 1. In this case, A is said to be relatively prime to B and B is said to be relatively prime to A. For example, 3 and 4 are relatively prime to each other, and 3 and 9 are not because 3 divides both 3 and 9. One of the tasks in the RSA public key encryption algorithm is to find the smallest positive integer greater than 1 that is relatively prime to a given positive even integer (an integer that is divisible by 2, such as 2,4,6,8, and so on). Such smallest integer along with the given integer is called public key. For example, the smallest positive integer greater than 1 that is relatively prime to 60 is 7.

Write a program to computer the smallest positive integer greater than 1 that is relatively prime to a given positive even integer.

input Each test case has one line that contains a positive even integer. *There is always a blank line after every test case.*

output For each test case, output two lines: the first line containing the smallest positive integer greater that 1 that is relatively prime to the input integer, *and a blank line as second.*

Example:

Input

```
60

192

```

Output

```
7

5
```

4 Private keys in RSA

Given any two positive integers A and B that are relatively prime to each other, it can be shown that $Ax + By = 1$ for some integers x and y. For example, let $A = 7$ and $B = 60$, we have $7 \times (-17) + 60 \times (2) = 1$, where x is -17 and y is 2. Note that we also have $7 \times (43) + 60 \times (-5) = 1$. Given two positive integers A and B that are relatively prime to each other, with $A \leq B$, we want to write a program to find $\underline{\text{the smallest positive integer}}$ x such that $Ax + By = 1$ for some integer y. This is one of the other tasks of the RSA public key encryption algorithm and x along with B is called private key.

input Each test case has one line that contains two positive integers A and B, $A \leq B$, separated by spaces, that are relatively prime to each other. *There is always a blank line after every test case.*

output For each test case, output two lines: the first line containing $\underline{\text{the smallest positive integer}}$ x such that $Ax + By = 1$ for some integer y *and a blank line as second.*

Example:

Input

```
7 60

11 60

5 192
```

Output

```
43

11

77
```

5 Pretty-Print Expressions

Arithmetic expressions such as 100*(3+8) are used in mathematics or in computer programs. Here are a few more examples: 2012, $(2-3)/((7*2)/4)$, $(2-3)$, $2+3*4$. For this problem, an arithmetic expression consists of +,-,*,/ (operators, note that each operator takes two values to perform an operation); positive integers (operands); and (,) (open and close parentheses for grouping) as its elements. An arithmetic expression is considered the same whether there are spaces between the elements or not. For example: ␣␣100␣␣*(␣3+8)␣␣␣␣, 100␣␣*(␣3+8␣␣)␣␣␣␣, and 100*(3+8) are the same, where ␣ stands for a space. The pretty-print version of an expression does not have leading and ending space, and has a single space between each element. For example, given any of the three expressions above, the pretty-print version is 100␣*␣(␣3␣+␣8␣).

Write a program that transforms each input arithmetic expression to its pretty-print version.

input Each test case has two lines. The first line contains an arithmetic expression described in the problem statement. An expression will have at most 5,000 characters, including spaces. *There is always a blank line after every test case.*

output For each test case, output two lines: the first line contains the pretty-print version, *and a blank line as second.*

Example:

Input

```
100*(3+8)

  100  *( 3+8)

100  *( 3+8  )

```

Output

```
100 * ( 3 + 8 )

100 * ( 3 + 8 )

100 * ( 3 + 8 )
```

6 Pretty-Print Expressions with Error Checking

The previous problem assumes an arithmetic expression is correct in the input and asks for a pretty-print version. In this problem, an input arithmetic expression may be incorrect or invalid. Here are a few examples: 100*(3+8, 100*(3+), 100*(3 8), 100*(a+3), 100*(3+8)), -2, and 100(3+8). Notice that a close parenthesis is missing in the first example; an operand is missing in the second example; an operator is missing in the third example; a is not a valid operand in the fourth example; an extra close parenthesis in the fifth example; the first operand is missing for - in the sixth example, and an operator is missing between 100 and the open parenthesis in the last example. Here are a few examples that are correct expressions: 2, (2), 2+3, (2+3), ((2+3)), ((2)+3), and (2+3)*3.

Write a program that is similar to the program that solves the previous problem with the enhancement that if the input arithmetic expression is invalid output INVALID to the output. You may assume any sequence of digits in an expression is a correct positive integer.

input Each test case has two lines. The first one contains an arithmetic expression which may be valid or invalid. An expression will have at most 5,000 characters, including spaces.*There is always a blank line after every test case.*

output For each test case, output two lines. The first line contains either a pretty-print version or INVALID depending on whether the input expression is valid or invalid. *The second line is a blank line.*

Example:

Input

```
100*(3+8)

(100*(3+8))

100*(((3))+8)

100*(3+8

100*(3+ )

100*(3 8)

100*(a+3)

100*(3+8))

100(3+8)

(3+8)(3+8)
```

Output

```
100 * ( 3 + 8 )

( 100 * ( 3 + 8 ) )

100 * ( ( ( 3 ) ) + 8 )

INVALID

INVALID

INVALID

INVALID

INVALID

INVALID

INVALID
```

7 Quotations Extraction

A string is a sequence of characters whose ASCII values are between 32 and 126. These characters include space, and any printable key on your keyboard. Note that tabs and newline characters are excluded.

Reading a string from left to right, the first quotation is the sequence of characters between the first pair of double-quotes, and the second quotation is the sequence of characters between the second pair of double-quotes, and so on. For example, given the following string

```
This "demonstrates" "how t"o handle unclosed "double quotes:" "abc
```

We have the first quotation: `demonstrates`; second quotation: `how t`; and third quotation: `double quotes:`. Note that the last double quote cannot be paired since no more double-quote can be found in the remaining characters.

Given a string, extract the quotations and print them out in the order of first quotation, second quotation, and so on. The program can assume that each input will have at least one quotation.

input Each test case has only one line containing a string as specified in the problem statement. The string will have length at most 6,000 characters. *There is always a blank line after every test case.*

output For each test case, output the quotations extracted from the string. Each quotation is printed out on a line and is ordered based on first quotation, second quotation, and so on. You can assume that all quotations are not empty, so they always have at least one character. *Output a blank line after each test case.*

Example:

Input

```
This is example one: "Hello, world."

This is a slightly more complicated "string" with more "double quotes."

This "demonstrates" "how t"o handle unclosed "double quotes:" ".
```

Output

```
Hello, world.

string
double quotes.

demonstrates
how t
double quotes:
```

8 A Quotation within a Quotation ... within a Quotation

Let us use the same string definition given in the previous problem. Suppose a quotation contains another quotation such as: `My friend once said: "Let me quote you a funny quote of my teacher, "The only rule we have is that there is no rule." Do you think the teacher's quote funny?"`. If we use the logic given in the previous problem, we will get `Let me quote you a funny quote of my teacher,` as the first quote and `Do you think the teacher's quote funny?` as the second quote. We totally missed the teacher's quotation. As a matter of fact, there are two quotations in the example string. The first one is:

```
Let me quote you a funny quote of my teacher "The only rule we have
is that there is no rule." Do you think the teacher's quote funny?
```

The second one is:

```
The only rule we have is that there is no rule.
```

Notice that double-quotes show up in the first quotation. We need to have a way to allow a double-quote as a character inside a quotation surrounded by two double-quotes. We need to disambiguate whether a double-quote signifies the start or the end of a quotation or a character within a quotation.

To this end, we introduce the notion of escaped characters inside of a quote: since a quotation is whatever between two double-quotes ("), if we want to insert a double-quote as part of the text of a quotation, we have to escape that double-quote. **Thus, if we want our quotation to include a double quote, we use \", two-character sequence, for " within the two double-quotes**. This solves the problem of allowing a double-quote within a quotation, but introduces a new problem of dealing with \ as a character within two double-quotes. For example, in `"abc \" abc..."`, if our intent is to have `abc \` as a quotation, we will not get it because \" is treated as a double-quote ("). So we need to escape \ as well. **We use \\, two-character sequence, for \ within the two double-quotes**. To have `abc \` as a quotation, we use the following string: `"abc \\" abc..."`. **Inside a quotation, if \ is not immediately followed by " or \, it is a syntax error**. In summary, there are only two possible types of escaped sequence as part of some quotation:

1. \" is an escaped sequence for the " character inside a quote
2. \\ is an escaped sequence for the \ character inside a quote

We will illustrate the concept of escaped sequences by the following examples:

1. The string `Hello, "Alice.\" This is \\Bob"` has only one quotation extracted and interpreted as `Alice." This is \Bob`
2. The string `Hello, "\Alice"` is an erroneous string since the sequence `\A` is not a valid construct inside two double-quotes, and no quotations can be extracted from it. Thus, between two double-quotes, a backslash must always be part of either of the previously described escaped sequence.
3. The string `Hello, "\\\\Alice"` is a valid input string with quotation extracted as `\\Alice`

Being able to allow double-quotes inside of a quotation, we are now able to allow nested quotations, or quotations inside of quotations. Thus, we consider quotations have multiple levels nested inside of each other. At the beginning, we extract the quotation A_1, if any, from the input string A_0 using the rules described above. This also means substituting escape sequences: \" and \\ by the characters they escape: " and \, respectively. Next we print out the A_1 on a line, and consider A_1 as a new string to be processed, just like A_0. We keep extracting a quotation from the previous quotation and print the new quotation out until the last quotation does not contain any quotation.

Consider the illustrative example below: initially the input string is A_0

1. A_0 = `\Zero\ "First \"Second \\\"Third \\\\\\\" None\\\"\""`
2. A_1 = `First "Second \"Third \\\" None\""`
3. A_2 = `Second "Third \" None"`
4. A_3 = `Third " None`

You should print out A_1, A_2, and A_3 in that order for the input string A_0. Note how the two backslashes surrounding `Zero` in A_0 do not result in any error since they are not inside any pair of double-quotes. Thus, A_0 is a correct input string.

An input string is erroneous if at any level, a quotation has a syntax error (i.e. inside a pair of double-quotes \ is not immediately followed by " or \). Consider the following example:

1. A_0 = `\Zero\ "First \"Second \\\"Third \\\\\\\" \\\\None\\\"\""`
2. A_1 = `First "Second \"Third \\\" \\None\""`
3. A_2 = `Second "Third \" \None"` - `ERROR`

You can see that A_2 has an error in `"Third \" \None"`, which has a syntax error due to the last backslash that is followed by N.

You may assume that for this problem, an input string has at most one (top level) quotation, and that every quotation extracted also contains at most one quotation nested *immediately* inside. However, no assumption can be made about the maximum level of quotations that an input may have.

input Each test case has only one line containing a string as specified in the problem statement. The string will have length at most 6,000 characters. *There is always a blank line after every test case.*

output For each test case, output all the quotations extracted from the string or output `ERROR` if the input string is erroneous as described above. You can assume that all quotations are not empty, so they always have at least one character. *Output a blank line after each test case.*

Example:

Input

```
This is a "very simple quote" as first example.

Hello, "Alice.\" This is \\Bob"

Hello, "\Alice"

Hello, "\\\\Alice"

\Zero\ "First \"Second \\\"Third \\\\\\\" None\\\"\""

\Zero\ "First \"Second \\\"Third \\\\\\\" \\\\None\\\"\""
```

Output

```
very simple quote

Alice." This is \Bob

ERROR

\\Alice

First "Second \"Third \\\" None\""
Second "Third \" None"
Third " None

ERROR
```