University of Arkansas
Department of Computer Science and Computer Engineering
2008 High School Programming Contest
Problems

Revision Date: March 1, 2008

Please carefully read the format specifications for output. Your program will be graded using a differential program (i.e. the output produced by your program will be compared byte-for-byte with the output of the judge's solution. Thus, even though your program may produce output that is *technically* correct, it will be counted as incorrect if these specifications are not followed exactly.

Unless otherwise directed, a line of output should have no leading or trailing whitespace. For example, suppose you are to produce the integer 27 on one line of output. That line should consist of the character '2' followed by the character '7' followed by the newline ('\n') character.

Note that for all problems, an integer is defined as:

an optional minus sign,

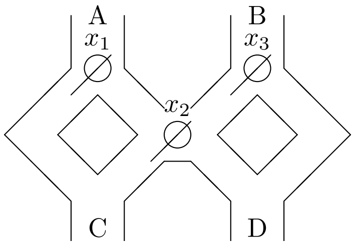
followed by one of the digits from the set 0..9,

followed by any number of digits from the set 0..9

Thus `3`, `-21`, and `309` are integers; `- 4` and `+38` are not.

1 Simulation: A Marble-Rolling Toy

The diagram below is a marble-rolling toy.



A marble is dropped at A or B. Levers x_1, x_2, and x_3 cause the marble to fall either to the left or to the right. Initially all levers cause the marble to fall to the left as shown in the diagram. Whenever a marble encounters a lever, it causes the lever to reverse after the marble passes, so the next marble will take the opposite branch. A sequence of A's and B's specifies a sequence of marbles dropped at A or B. For example, AAB specifies that the first marble is dropped at A, the second marble is dropped at A, and the third marble is dropped at B. Given such a A and B sequence specifying how marbles are dropped, we would like to know if the last marble exits at D or not. For example, for sequence AAB, the first marble dropped at A exits C (x_1 lever causes the marble to fall to the left and then reverses); the second marble dropped at A exits C (x_1 lever causes the marble to fall to the right then reverses back, and x_2 lever causes the marble to fall to the left then reverses); the third marble dropped at B exits D (x_3 lever causes the marble to fall to the left then reverses, and x_2 lever causes the marble to fall to the right then reverses). You may verify that for sequence AABB the last marble also exits D. Our task is to write a program which reads a A and B sequence and writes ACCEPTED! if the last marble exits D and REJECTED! otherwise.

input file input; the name of the file is passed as a command line argument; the file contains one line of A and B sequence.

output ACCEPTED! or REJECTED! to the console (stdout) depending on whether the last marble exits D or C respectively.

Example:

Input

```
AABB
```

Output

```
ACCEPTED!
```

Example:

Input

```
BA
```

Output

```
REJECTED!
```

2 Algorithm Analysis: Cutoff Value

In algorithm analysis, the run time of an algorithm is expressed as a function $T(n)$ of its input size n. $T(n)$ is called the time complexity of the algorithm. For instance, for a given problem one algorithm may have $T_1(n) = n^2$ while another algorithm may have $T_2(n) = 100nlogn + 100n$ (the base of the log is 2). We would like to pick a faster algorithm to be implemented in our program. The question becomes which of the two algorithms is faster. It seems that the answer may depend on the input size. For example, if $n = 16$, $T_1(n) = 256$ and $T_2(n) = 8,000$. In this case the algorithm with time complexity $T_1(n)$ is faster than the algorithm with time complexity $T_2(n)$. Now suppose $n = 2048$. Now $T_1(n) = 4,194,304$ and $T_2(n) = 2,457,600$, and hence the algorithm with time complexity $T_2(n)$ is faster than the algorithm with time complexity $T_1(n)$. $T_1(n) = n^2$ is a special case of the general form $a_2n^2 + a_1n + a_0$ when $a_2 = 1$, $a_1 = 0$, and $a_0 = 0$. Similarly, $T_2(n) = 100nlogn + 100n$ is a special case of the general form $b_2nlogn + b_1n + b_0$ when $b_2 = 100$, $b_1 = 100$, and $b_0 = 0$. Note that in the discussion above, $a_2, a_1, a_0, b_2, b_1, b_0$ are place holders for non-negative integers. It is claimed that no matter what the values of $a_2, a_1, a_0, b_2, b_1, b_0$ are, there is a smallest value for n at and after which $T_1(n) > T_2(n)$. Our task is to write a program to find such value for a given $a_2, a_1, a_0, b_2, b_1, b_0$.

input file input; the name of the file is passed as a command line argument; the file contains two lines; the first line contains three integers separated by spaces for a_2, a_1, a_0 respectively; the second line contains three integers separated by spaces for b_2, b_1, b_0 respectively.

output the smallest value n properly formatted to the console (stdout).

Example:

Input

```
1  0  0
100 100 0
```

Output

```
1,112
```

3 Postfix Notation: Evaluation

Typically an arithmetic expression is given in infix notation, such as 10+70. There is another notation called postfix, which also can be used to denote arithmetic expression. As the name suggests, in postfix notation the operator is after the operands. The infix expression 10+70 becomes 10 70 + (a space is used to separate the two operands 10 and 70 and the operator). A nice property of postfix is that we do not need to worry about operator precedence and therefore eliminating the need for using parentheses. Note that a+b*c and (a+b)*c are different in infix due to operator precedence. In postfix, a+b*c becomes abc*+, and (a+b)*c becomes ab+c* (note that the parentheses are gone).

Our task is to write a program to evaluate an arithmetic expression in postfix notation that involves positive integer operands and the following binary operators $+$, $-$, $*$.

input file input; the name of the file is passed as a command line argument; the file contains one line of a valid postfix with operands and operators separated by spaces.

output the integer resulted from the evaluation of postfix expression to the console (stdout).

Example:

Input

```
10 5 + 2 3 - - 2 *
```

Output

```
32
```

4 Postfix Notation: Conversion

The previous problem asks us to evaluate a postfix expression. In this problem we are asked to convert a postfix expression to an equivalent infix expression. The operands are single character lower or upper case letters, and the operators are binary operators $+$, $-$, $*$. Our program will produce two infix expressions in two lines. The first line is a fully parenthesized infix. The second line is an infix in which unnecessary parentheses are gone. For example, the input postfix abc*+ will result (a+(b*c)) as a fully parenthesized infix, and a+b*c as infix without unnecessary parentheses. Similarly, for postfix ab+c* we will have ((a+b)*c) and (a+b)*c as the two infix expressions.

input file input; the name of the file is passed as a command line argument; the file contains one line of a valid postfix.

output two lines of output to the console (stdout), where the first line is a fully parenthesized infix and the second line is a infix without unnecessary parentheses.

Example:

Input

```
ab+CA--d*
```

Output

```
(((a+b)-(C-A))*d)
(a+b-(C-A))*d
```

5 Binary String: Generator

Our task is to write a binary string generator program. The program takes two integer values, which are the initial length, and the final length in that order. The initial length is an integer with value at least 1. The final length is also an integer whose value is at least as large as the initial length. The generator program should write all the binary strings whose lengths are between the initial length and the final length in lexicographical order, one line per string, to the console.

input file input; the name of the file is passed as a command line argument; the file contains two lines each containing a single number. The number in the first line is the initial length. The number in the second line is the final length.

output all binary strings between the initial length and the final length in lexicographical to the console (stdout), where each string takes a line.

Example:

Input

```
2
3
```

Output

```
00
01
10
11
000
001
010
011
100
101
110
111
```

Example:

Input

```
2
2
```

Output

```
00
01
10
11
```

6 Binary String: Checker

Our task is to write a binary string checker program. The program takes one command line argument, which is the name of the input file. The input file is a text file in ASCII. The input file should contain a binary string. We do not know how long the binary string is, meaning how large the input file is. We will interpret the binary string as a binary integer. For instance, 010 is invalid (due to leading zero), 10 is the integer 2 in decimal, 110 is integer 6 in decimal, and 1001 is integer 9 in decimal. Our checker program should output INVALID to the console if the binary string begins with a 0, otherwise it should output YES to the console if the binary integer is divisible by 3 and NO otherwise. We may assume the characters in input file are 0, 1, and end of file character.

Example:

Input file contains:

```
011
```

Console Output

```
INVALID
```

Input file contains:

```
1111
```

Console Output

```
YES
```

Input file contains:

```
1010
```

Console Output

```
NO
```

7 Partition: Two Triangular Number

The n^{th} *triangular* number T_n is given by $n + (n-1) + (n-2) + \cdots + 1$. Sometimes $T_x + T_y = xy$. For example, $T_{90} + T_{415} = 90415$; $T_{585} + T_{910} = 585910$.

input input file input; the name of the file is passed as a command line argument; the file contains lines each of which consists of one integer representing the sum of two triangular numbers.

output output an integer x, a space, an integer y, (with the smaller number first) such that the sum of the x^{th} and y^{th} triangular numbers equals the input number to the console (stdout)

Example:

Input

```
90415
585910
```

Output

```
90 415
585 910
```

8 Seeking: Maximum Subsequence Sum

Given (possibly negative) integers $a_0, a_1, \ldots, a_n$, find a subsequence $a_i, \ldots, a_j$ such that the sum $(\Sigma_{k=i}^{j} a_k)$ or $a_i + a_{i+1} + \cdots + a_j$ of the elements in the subsequence is maximum among all possible subsequences (including the empty sequence whose sum is 0) of $a_0, a_1, a_2, \ldots, a_n$. Therefore the maximum subsequence sum is 0 if all the integers are negative. In this case, the output is -1 -1 0.

input file input; the name of the file is passed as a command line argument; the file contains a sequence of integers separated by spaces.

output start position of the subsequence, end position of the subsequence, sum of the subsequence to the console (stdout).

Example:

Input

```
-2 11 -4 13 -5 -2
```

Output

```
1 3 20
```

Example:

Input

```
3  -2  11  -4  13  -5  -2  6
```

Output

```
0 4 21
```