

Problems from the Boston Preliminary Contest, October 26th

Authorship Verification

Slippery Redux, the Text Private Eye, is trying to set up a system to catch text copycats. To do this he needs a program that will take input text and count the words that match a pattern. The patterns he is interested in are fairly simple: they merely match the beginning of each word to see if there is some sequence of vowels and consonants and particular letters. Thus the pattern `CVC' matches all words beginning with a consonant followed by a vowel followed by a consonant, and the pattern `thV' matches all words beginning with the letters `th' followed by a vowel.

The input has been preprocessed so it is just a sequence of data words each of which contain nothing but lower case letters. The pattern is a word consisting of lower case letters and the letters `C' and `V'. In a pattern, a lower case letter matches only the letter itself in a data word, while `C' in a pattern matches any consonant in a data word, and `V' in a pattern matches any vowel in a data word.

The letter `y' in a data word is to be treated as both a vowel and a consonant. Thus BOTH `V' AND `C' ALWAYS match `y'.

Input

The input is a sequence of items, each of which is a pattern, a data word, or the special item `*'. Items are separated by whitespace, which consists of one or more spaces, tabs, or newlines.

The input contains one or more test cases, each consisting of a pattern followed by 1 or more data words followed by the item `*'.

Input ends with the item '*' by itself (where the next pattern would be).

No data word or pattern is longer than 80 characters.

Output

For each test case, the single line

Case #: #/# = #%

where the #'s denote numbers and are the following in order: the test case number (chosen from 1, 2, 3, etc.), the number of matching data words, the total number of data words, and the ratio of matching to total data words expressed as a percentage with exactly one decimal place.

Sample Input

```
thV
i like the best of the noise sometimes *
V
i like the best of the noise sometimes *
CVC
i like the best
of the noise sometimes *
CV
do you like my new nonsense *
*
```

Sample Output

Case 1: 2/8 = 25.0%
Case 2: 2/8 = 25.0%
Case 3: 3/8 = 37.5%
Case 4: 6/6 = 100.0%

File: authorship.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Sat Oct 26 16:02:39 EDT 2002

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

\$Author: hc3-judge \$
\$Date: 2002/10/26 20:04:51 \$
\$RCSfile: authorship.txt,v \$
\$Revision: 1.8 \$

Achieving Better Bias

The Better Bias Bureau takes M faced biased dice with known biases and makes from them N faced biased dice with desired biases. More specifically, if you have an M faced biased die with probability $pm[j]$ of throwing face j , for j from 1 through M, but you really want instead an N faced biased die with probability $pn[i]$ of throwing face i , for i from 1 through N, then good old B3 will give you a method of achieving your desires.

The method is this. You are to consider throwing your M faced die an infinite number of times and writing out the resulting faces, f_1, f_2, f_3, \dots , as an infinite fraction $.f_1f_2f_3\dots$ base M. The fraction will be a number, call it F , in the range from 0 through 1. For any number r , $0 \leq r \leq 1$, we can compute the probability that $F \leq r$.

Let $rn[i]$ be defined so that

probability $F \leq rn[i]$
 = probability that an N faced die throw is k for
 some $k \leq i$
 = sum of $pn[k]$ for k from 1 through i

Then when you throw a value F with your M faced die, you should declare it to be face i of the N faced die if

$F < rn[i]$ if $i = 1$

or

$rn[i-1] < F < rn[i]$ if $1 < i < N$

or

$rn[i-1] < F$ if $i = N$

or

This method produces a properly biased N faced die. Here one ignores the cases where $F = rn[i]$ for some i , as the probability of this happening is 0.

The neat thing about all this is that you do not actually have to throw the M faced die an infinite number of times to decide what the value of i is. After some finite number of throws, depending on the value of F , you can stop, knowing the answer i .

Suppose you stop as soon as possible. What is the expected number of throws of the M faced die necessary to produce one throw of the N faced die?

Note:

For technical reasons, relating to the elimination of ambiguity in judging a contest, we use only one inequality if $i = 1$ or $i = N$. To be a little more specific, the judge's test data has been adjusted so that for all cases that must actually be tested, either both the inequalities $rn[i-1] < F$ and $F < rn[i]$ will be true by a margin definitely larger than the accuracy of the computation, or one of these inequalities will be false by a margin definitely larger than the accuracy of the computation. But it is NOT possible to adjust the judge's data so the inequalities at the ends, $rn[0] = 0.0 < F$ and $F < rn[N] = 1.0$, will be true or false by a margin definitely larger than the accuracy of the computation. So we suppress these two inequalities, which we can do without changing the validity of the method.

Input

For each of several test cases:

* M followed by $pm[1]$, $pm[2]$, ..., $pm[M]$ in order.

* N followed by $pn[1]$, $pn[2]$, ..., $pn[N]$ in order.

Here $2 \leq M \leq 20$, $2 \leq N \leq 20$, and for all j and i
 $0.01 \leq pm[j] \leq 0.99$, $0.01 \leq pn[i] \leq 0.99$. The sum
of all the $pm[j]$ equals 1.0, and the sum of all $pn[i]$
equals 1.0.

Numbers are separated by whitespace, which may consist
of any sequence of spaces, tabs, or newlines.

Input ends with a line containing just a single 0.

You should use double precision floating point arithmetic
to do input and computation, as more than 6 significant
figures of accuracy are necessary.

Output

For each test case, one line containing the expected
number of throws of the M sided die needed to generate
one throw of the N sided die. This number should have
exactly 2 decimal places.

Sample Input

```
2 0.5 0.5
4 0.1 0.2
  0.3 0.4
4 0.1 0.2 0.3 0.4
2 0.5 0.5
2 0.5 0.5
3 0.333333333333 0.333333333333 0.333333333333
0
```

Sample Output

```
3.50
1.45
3.00
```

Note

You might find it interesting to consider at your leisure whether B3's method is optimal, i.e., achieves the minimal expected number of throws of the M faced die to generate one throw of the N faced die.

File: betterbias.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Mon Oct 21 01:22:35 EDT 2002

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

\$Author: walton \$
\$Date: 2002/10/21 05:23:01 \$
\$RCSfile: betterbias.txt,v \$
\$Revision: 1.11 \$

GSAT Tracing

GSAT is a 'greedy local search' procedure for satisfying propositional formulae in CNF form. You have been asked to code the GSAT algorithm, and as a first step, you are to code a variation of the core of this algorithm and output a trace of its execution.

A propositional formula is a formula containing boolean variables and the operators AND, OR, and NOT. A propositional formula is in 'conjunctive normal form', or CNF, if it has the syntax:

```
CNF-formula ::= clause { AND clause }*
clause ::= ( literal { OR literal }* )
literal ::= variable | NOT variable
```

Thus if v_1, v_2, v_3, \dots are boolean variables, an example is

$$(v_1 \text{ OR } \text{NOT } v_3) \text{ AND } (\text{NOT } v_4 \text{ OR } v_5 \text{ OR } v_2)$$

As a shorthand we will write such formula with the following modifications:

- A. We will omit OR's and AND's.
- B. We will omit v's, representing variables by integers.
- C. We will replace NOT's by -'s, negating the integer representing a variable that is to be NOT'ed. We will omit space between the '-' and its following integer.

Thus the above example becomes:

$$(1 -3) (-4 5 2)$$

The goal of the GSAT algorithm is to find an assignment of values (true or false) to variables so that the given input CNF formula is satisfied (evaluates to true by the rules of boolean algebra).

The core of the GSAT algorithm is:

1. Assign values (true or false) to all the variables randomly.
2. If all clauses are satisfied (so the formula is satisfied), stop.
3. For each variable i , compute the number of clauses $N[i]$ that will be satisfied if just the value of variable i is changed.
4. Form the set of variables S whose $N[i]$ is maximal.
5. Pick one variable from S randomly and change its value.
6. Return to step 2.

You have been asked to program a first version of this GSAT core that replaces random choices as follows:

1. In step 1, the initial variable assignments are input.

5. In step 5, instead of picking a variable from S randomly, the variable i with the smallest value of

$$(i - \text{iteration}) \bmod V$$

is chosen, where 'iteration' is the iteration number, 1, 2, 3, etc., and V is the number of variables. This amounts to searching variables beginning with variable

$$((\text{iteration} - 1) \bmod V) + 1$$

and wrapping around from variable V to variable 1, picking the first variable with maximum $N[i]$ that is found.

You are asked to print out all variable changes in step 5 and the set of currently unsatisfied clauses in step 2.

To simplify implementation, GSAT is usually programmed to run only on CNF formula whose clauses contain at most 3 literals. All CNF formula can be easily converted to such a form. To make coding easier, we require that all clauses have exactly 3 literals. To make this work, we introduce the special variable v_0 whose value is ALWAYS false. Thus the clause $(v_1 \text{ OR } v_2)$ with two literals is changed to the clause $(v_1 \text{ OR } v_2 \text{ OR } v_0)$ with three literals, the last of which, v_0 , is always false.

Input

For each formula, integers V, C, and I, in that order, which are respectively the number of variables, the number of clauses, and the maximum number of algorithm iterations. $1 \leq V \leq 100$, $1 \leq C \leq 100$, and $1 \leq I \leq 100$. These are followed by V integers, each 0 representing 'false' or 1 representing 'true', that are the initial values of the variables, in order from variable 1 through variable V. And these are followed by the clauses, represented as a sequence of integers, using the abbreviations given above. As each clause has exactly 3 literals, parentheses would be redundant information, and are therefore omitted. If a clause contains variable 0, it is the last one or two variables of the clause.

The input ends with a line containing 3 zeros.

Output

For each data set, first a single line containing 'Formula #' where # is 1, 2, 3, etc., the number of the formula in the input.

Then every time step 2 is executed, one or more lines containing the clauses that are not satisfied. Each clause is represented by three integers enclosed in parentheses, with clauses being separated by a space. Consecutive integers are separated by a space, but parentheses are NOT separated from integers by space. If there are more than 5 clauses, then exactly 5 clauses are printed on each line but the last, which may have 1 to 5 clauses. The clauses MUST be printed the order they were input.

As a special case, if there are no unsatisfied clauses at step 2, print a single line containing just `DONE' and stop the algorithm.

Lastly, every time step 5 is execute print either `# = true' or `# = false' to indicate that variable # has been assigned the new value `true' or `false' respectively.

Execution of the algorithm should stop immediately after the algorithm executes step 2 for the I+1'st time, if execution has not previously stopped because the formula was satisfied.

Sample Input

```
4 5 10
0 0 0 0
 1  2  0
 2  3  0
 3  4  0
 4 -1  0
-1 -2  0
4 4 10
0 0 0 0
 1 -2  0
 2 -3  0
 3 -4  0
 4  0  0
4 5 8
1 0 0 1
-2  3  0
-3  4  0
-4 -2  0
 1  2  0
-1  2  0
0 0 0
```

Sample Output

Formula 1

(1 2 0) (2 3 0) (3 4 0)

2 = true

(3 4 0)

3 = true

DONE

Formula 2

(4 0 0)

1 = true

(4 0 0)

2 = true

(4 0 0)

3 = true

(4 0 0)

4 = true

DONE

Formula 3

(-1 2 0)

1 = false

(1 2 0)

3 = true

(1 2 0)

3 = false

(1 2 0)

4 = false

(1 2 0)

1 = true

(-1 2 0)

2 = true

(-2 3 0)

3 = true

(-3 4 0)

4 = true

(-4 -2 0)

Notes:

The full GSAT algorithm runs the core algorithm some number of times, and claims, perhaps incorrectly, that the formula is unsatisfiable if none of these runs produces a satisfying variable assignment. Each core run stops after some number of iterations if it fails to find a satisfying variable assignment. Full GSAT has a good record of finding satisfying variable assignments if they exist for some important classes of formulae.

The original GSAT paper is Selman, Levesque, and Mitchell, A New Method for Solving Hard Satisfiability Problems, Proc 10th Conf on AI (AAAI-92), July 1992, 440-446.

File: gsat.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Mon Oct 21 00:36:45 EDT 2002

The authors have placed this file in the public domain; they make no warranty and accept no liability for this file.

RCS Info (may not be true date or author):

\$Author: walton \$
\$Date: 2002/10/21 04:39:01 \$
\$RCSfile: gsat.txt,v \$
\$Revision: 1.7 \$

Love Log

Olivia, the arrow beetle, is on the ground near a log. On the other side of the log is Oliver, the love of her life. She wants to get to him. How far must she travel?

Olivia and Oliver are on the ground, the log rests on the ground, and Olivia must fly over the top of the log to get to Oliver. You can assume that the ground is perfectly flat and the log is a perfect cylinder with axis perpendicular to the line between Olivia and Oliver. The distance B from Olivia to the point where the log touches the ground, the distance C from this point to Oliver, and the diameter D of the log are given. Arrow beetles fly in perfectly straight lines when they need to get to a point they can see, and otherwise can fly right next to a curved surface. You can assume 'right next to' means infinitesimally close to.

Olivia or Oliver can be in the shadow of the log; i.e., $B < D/2$ or $C < D/2$ are possible.

Input

Several test cases, each with one line containing B , C , and D , in that order. $0 < B, C, D < 1000000$. Input ends with a line containing three zeros.

You should use double precision floating point arithmetic to do input and computation, as input with 3 decimal places can have up to 9 significant digits.

Output

For each test case, one line containing the length of the shortest arrow beetle flying route from Olivia to Oliver. This length must have exactly 3 decimal places.

Sample Input

```
10.0 10.0 20.0
 0.1  0.1 20.0
10.0 80.0 20.0
0 0 0
```

Sample Output

```
51.416
62.632
108.195
```

```
File:      lovelog.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Sun Oct 20 22:44:48 EDT 2002
```

The authors have placed this file in the public domain; they make no warranty and accept no liability for this file.

RCS Info (may not be true date or author):

```
$Author: walton $
$Date: 2002/10/21 02:46:01 $
$RCSfile: lovelog.txt,v $
$Revision: 1.3 $
```

Marriage Mayhem

Alex and Betty are soon to be married, but have each forgotten a crucial item at home. They must each collect their item before the wedding may take place; thus they wish to arrive there as soon as possible. However, as tradition has it that the bride and groom must not meet before the wedding, they must take care to avoid running into each other.

Being decent people, their world consists of a graph. The problem will specify Alex and Betty's start and target vertices. Traversing an edge will take one second. (It is also allowed to stand still at a vertex.) It is not allowed for Alex and Betty to:

- be at the same vertex at the same time;
- traverse the same edge at the same time.

Your job is to compute the earliest time at which both Alex and Betty can be home, or to say that no solution is possible.

Input

For each test case, the input is:

Line 1: 5 integers: the number of vertices N ;
Alex's start vertex `as` and finish vertex
`af`; Betty's start vertex `bs` and finish
vertex `bf`; Vertices are numbered $1..N$.
 $1 \leq N \leq 30$.

Lines $2..N+1$:
An $n \times n$ array of 0's and 1's, where 1's indicate
the presence of edges. Each row is on a single
line. The vertex numbers increase from left to
right and from top to bottom. The matrix is
symmetric and the diagonal is all 1's.

Input will be terminated a line containing five 0's.

Output

A single line beginning with `Case #:`. Then if a
solution is possible, this is followed by the integer
number of seconds the quickest solution takes; but if
no solution is possible, the word `impossible` is to
be output in place of an integer.

Sample Input

```
3 1 3 3 1
1 1 1
1 1 1
1 1 1
3 1 3 3 1
1 1 0
1 1 1
0 1 1
0
```

Sample Output

```
Case 1: 2
Case 2: impossible
```

```
File:      marmay.txt
Author:    Vince Conitzer <conitzer@post.harvard.edu>
           with minor revisions by
           Bob Walton <walton@deas.harvard.edu>
Date:      Sat Oct 26 16:03:37 EDT 2002
```

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

```
$Author: hc3-judge $
$Date: 2002/10/26 20:04:51 $
$RCSfile: marmay.txt,v $
$Revision: 1.9 $
```

Nasty Grammar

You have been asked to write a parser for the grammar

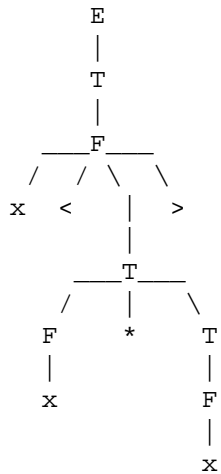
```

E -> T
E -> T < E
T -> F
T -> F * T
F -> x
F -> ( E )
F -> x < T >

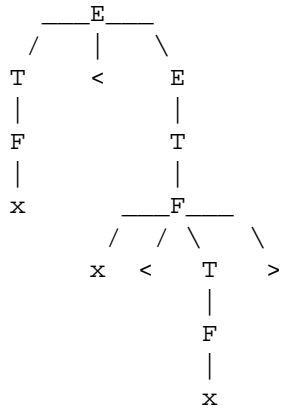
```

where in the second rule < is used as an operator and in the last rule < > are used as brackets. Note that the grammar is unambiguous (not obvious, but true). To keep things simple, the only terminals are x, *, (,), <, and >.

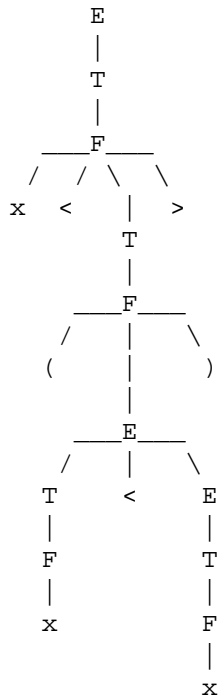
With this grammar the parse tree of $x<x*x>$ is



while the parse tree of $x<x<x>$ is



and the parse tree of `x<(x<x)>` is



Note

This grammar is a little nastier than JAVA in which `(t<x.y>)z` is a cast of `z` to the type `t<x.y>` where `x.y` is a parameter to the parameterized type `t`.

Input

For each test case, an input string made of terminals on a line by itself. Only terminal characters will be on the line. The maximum line length is 80 characters. Input ends with an end of file.

Output

For each test case, a single line. If the input string is an E, the line contains `accept' followed by the number of E, F, and T nodes in the parse tree, in the format indicated by the sample output. If the input string is NOT an E, the line merely contains `reject'.

Sample Input

```
x*<x>
x<x*x>
x<x<x>
x<(x<x)>
x<x<x>><x>
```

Sample Output

```
reject
accept 1E 3T 3F
accept 2E 3T 3F
accept 3E 4T 4F
reject
```

File: nastygrammar.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Thu Oct 17 10:11:07 EDT 2002

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

\$Author: walton \$
\$Date: 2002/10/17 14:26:03 \$
\$RCSfile: nastygrammar.txt,v \$
\$Revision: 1.3 \$

Tom's Ski Shop

Tom owns a ski shop at the foot of Mt. Big Toe. He both sells and rents skis. Tom's customers can either buy skis for X dollars, or rent them for a weekend for Y dollars. If a customer skis for T weekends before he tires of skiing and stops, and if the customer rents for the first K weekends and then buys on the $K+1$ 'st weekend, with $K+1 \leq T$, then the customer pays $K*Y + X$ dollars to have skis for T weekends. If $X < T*Y$, the customer would find it least expensive to have bought skis on his very first weekend ($K = 0$), but if $X > T*Y$, the customer would find it least expensive to always rent ($K \geq T$). A customer does not know his T in advance of actually getting tired of skiing, so these last facts are not very useful.

Tom has studied this situation statistically and devised 'optimal' advice for his customers. By observation, Tom knows the probability $P[T]$ that an arbitrary customer will tire of skiing after exactly T weekends. Using this Tom finds the number K such that if every customer rents for her first K weekends and buys on her $K+1$ 'st weekend, the expected cost to the customer is minimized.

Specifically, if the customer follows Tom's strategy for a given K , and the customer tires after T weekends, the cost to the customer is $T * Y$ if $T \leq K$ and $K * Y + X$ if $T > K$. If we denote this cost by $C[T]$, then the expected cost is the sum of $P[T]*C[T]$ over all T , and this expected cost, being a function of K , can be minimized by selecting K properly.

You are asked to compute the optimal K given $P[T]$. Somewhat artificially, you are to assume that $P[T]$ is 0 for all T above some value N , and you are not to consider any K above N .

Input

For each of a number of data sets, the numbers X, Y, and N in that order, followed by N probabilities, P[1], P[2], through P[N]. $0 < X$, $0 < Y$, $1 \leq N \leq 100$. The probabilities are all between 0.0 and 1.0 inclusive, and the probabilities sum to 1.0. The numbers are separated by whitespace, consisting of spaces, tabs, and newlines in any combination.

Input ends with a line containing three zeros.

Output

For each data set, one line containing first the optimal value of K and second the expected cost in dollars for that K. The cost must have exactly 2 decimal digits.

Sample Input

```
50.00 30.00 4 0.5 0.0 0.0 0.5
70.00 30.00 4 0.5 0.0 0.0 0.5
100.00 30.00
    4 0.5 0.0 0.0 0.5
0 0 0
```

Sample Output

```
0 50.00
1 65.00
4 75.00
```

File: skishop.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Mon Oct 21 00:14:50 EDT 2002

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

\$Author: walton \$
\$Date: 2002/10/21 04:16:23 \$
\$RCSfile: skishop.txt,v \$
\$Revision: 1.4 \$