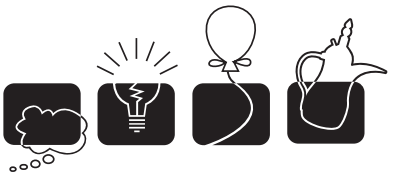


The 30th Annual ACM International Collegiate Programming Contest

Sponsored by IBM

Arab and North African Eighth Regional Contest

sponsored by:
Kuwait University
Kuwait Foundation for the Advancement of Sciences
NCS, CBK, NTEC, HumanSoft, Universe, GBM & Al-Alamiah

Kuwait University
December 1, 2005

The problem set is made of 16 numbered pages

[A] The Earth is Flat!

Program:	`flat.(c|cpp|java)`
Input:	`flat.in`
Balloon Color:	Gold

Description

Consider the following language:

$$expression = \begin{cases} c & \textit{where } c \textit{ is a single, lower-case letter} \\ (\ e_1\ e_2\ \cdots\ e_t\ n\ ) & \textit{zero or more expressions followed by a natural number} \end{cases}$$

The left column of the table below includes sample expressions of this language. Now the flattening of an expression is defined as follows: A single letter is flattened to itself. An expression of the form $(\ e_1\ e_2\ \cdots\ e_t\ n\ )$ is flattened by concatenating n copies of the concatenation of the flattening of e_i. In other words, if $f(e)$ is the flattening of e, and + symbolizes concatenation, then $f(\ \ (e_1\ e_2\ \cdots\ e_t\ n)\ \ )$ is:

$$\underbrace{f(e_1)+f(e_2)+\cdots+f(e_t)}_{\text{once}} \ + \ \overbrace{f(e_1)+f(e_2)+\cdots+f(e_t)}^{\text{twice}} \ + \ \cdots \ + \ \underbrace{f(e_1)+f(e_2)+\cdots+f(e_t)}_{\text{the n}^{th}\text{ time}}$$

The following tables show some sample expressions and the result of flattening each.

expression	flattened expression
`w`	w
`(c 4)`	cccc
`(a (b c 2) 3)`	abcbcabcbcabcbc

Write a program to flatten a given expression.

Input Format

Your program will be tested on one or more test cases. Each test case is made of one correctly formed expression written on a separate line. A `'$'` character identifies the end of line. The last line of the input, which is not part of the test cases, contains a `'$'` by itself (possibly with leading and/or trailing white spaces). Every expression in the input is grammatically correct according to the grammar specified above. Note that an expression may contain leading, trailing, and/or embedded spaces. Such spaces should be ignored. Letters and numbers are separated from each other by at least one space character.

Output Format

For each test case expression, write its flattening on a separate line. There should be no spaces (other than newlines) in the output.

Sample Input/Output

flat.in

```
w$
(c 4)$
(a (b c 2) 3)$
$
```

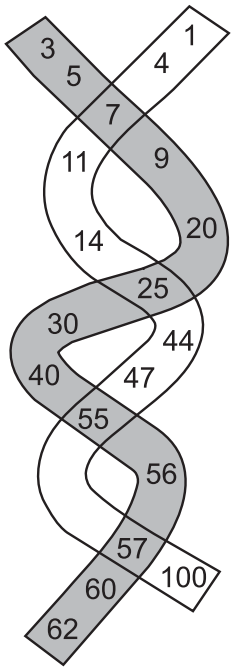
OUTPUT

```
w
cccc
abcbcabcbcabcbc
```

[B] The Double HeLiX

Program:	`helix.(c|cpp|java)`
Input:	`helix.in`
Balloon Color:	Green

Description



Two finite, strictly increasing, integer sequences are given. Any common integer between the two sequences constitute an intersection point. Take for example the following two sequences where intersection points are printed in bold:

First= 3 5 **7** 9 20 **25** 30 40 **55** 56 **57** 60 62
Second= 1 4 **7** 11 14 **25** 44 47 **55 57** 100

You can 'walk" over these two sequences in the following way:

1. You may start at the beginning of any of the two sequences. Now start moving forward.
2. At each intersection point, you have the choice of either continuing with the same sequence you're currently on, or switching to the other sequence.

The objective is finding a path that produces the maximum sum of data you walked over. In the above example, the largest possible sum is 450 which is the result of adding 3, 5, 7, 9, 20, 25, 44, 47, 55, 56, 57, 60, and 62.

Input Format

Your program will be tested on a number of test cases. Each test case will be specified on two separate lines. Each line denotes a sequence and is specified using the following format:

$$n \quad v_1 \quad v_2 \quad \ldots \quad v_n$$

Where n is the length of the sequence and v_i is the ith element in that sequence. Each sequence will have at least one element but no more than 10,000. All elements are between -10,000 and 10,000 (inclusive).

The last line of the input includes a single zero, which is not part of the test cases.

Output Format

For each test case, write on a separate line, the largest possible sum that can be produced.

Sample Input/Output

helix.in

```
13 3 5 7 9 20 25 30 40 55 56 57 60 62
11 1 4 7 11 14 25 44 47 55 57 100
4 -5 100 1000 1005
3 -12 1000 1001
0
```

OUTPUT

```
450
2100
```

[C] Alpha of Degree k

Program:	`alpha.(c|cpp|java)`
Input:	`alpha.in`
Balloon Color:	Pink

Description

Given two strings s and t, we'll say that the relation *"alpha of degree k"* holds between s and t, written as ($s \xrightarrow{\alpha^k} t$), iff there exists a suffix of s with a minimum length of k characters which is also a prefix of t. Note that the alpha relation is not defined for $k = 0$.

For example, the word `"telnet"` is α^3 to `"network"` while `"block"` is α^4 (and α^3, and α^2, and α^1) related to `"locker"`.

An α^k chain of length L (where $L > 0$) from word s to word t is a list of $L + 1$ words where s is the first word, t is the last word, and an α^k relation holds between each two consecutive words. For example, here's an α^2 chain of length 4 between `"cartoon"` and `"manual"`.

$$\texttt{cartoon} \xrightarrow{\alpha^2} \texttt{one} \xrightarrow{\alpha^2} \texttt{new} \xrightarrow{\alpha^2} \texttt{newsman} \xrightarrow{\alpha^2} \texttt{manual}$$

Given a dictionary of words C, two words s and t (taken from the dictionary), and two integers k and L, write a program that determines if there is an α^k chain between s and t where the chain's length doesn't exceed L and all words in the chain are from C.

Input Format

Your program will be tested on one or more test cases. The first line of the input specifies a single integer D which represents the number of test cases.

Each test case specifies a dictionary of words and a number of queries on that dictionary. The first line of each test case specifies two integers W and Q where W is the number of words in this test case's dictionary and Q is the number queries to be performed on that dictionary. Note that $0 < W < 50,000$ and $0 < Q < 100$.

The dictionary is listed starting at line 2 of each test case, one word on each line, all words are composed of lower case letters, there are no duplicates, no spaces, and no word is longer than 64 characters.

Following the dictionary, Q queries are specified. Each query is specified on a separate line by specifying two strings s and t and two integers k and L.

For each query, determine if, using words found in the dictionary, there is an α^k chain from s to t whose length doesn't exceed L.

Output Format

For each query, write the result on a separate line. If there is no chain that satisfies the given constrains, your program should print:

`a.b␣none`

where `a` is the test case number (starting at 1,) and `b` is the query number within this test case (again starting at 1.)

(continued on next page)

If, however, there is a chain, your program should print the following:

`a.b␣c␣word-1␣word-2␣...␣word-n`

Where `a`, `b` are as described previously, `c` is the length of the chain. `word-1`, `word-2`, ... `word-n` is the chain, where each two consecutive words are separated by a single space.

If there are more than one solution for a query, just print anyone of them.

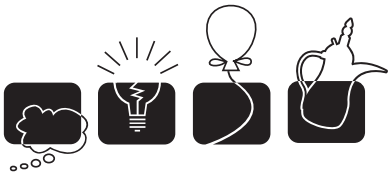
Sample Input/Output

alpha.in

```
2
8 3
news
perusal
symbolic
newspaper
salon
longstreet
cartoon
streetcar
news cartoon 3 8
news cartoon 3 4
news cartoon 1 4
2 2
link
blink
blink link 3 10
link blink 2 100
```

OUTPUT

```
1.1 6 news newspaper perusal salon longstreet streetcar cartoon
1.2 none
1.3 4 news salon longstreet streetcar cartoon
2.1 1 blink link
2.2 none
```



[D] Sort that Queue

Program:	`queue.(c|cpp|java)`
Input:	`queue.in`
Balloon Color:	`Silver`

Description

Given a queue `Q`, a stack `A`, and a stack `B`, and the following operations:

`QA(n)`: dequeue an item from `Q`, pushing it on `A`; repeat `n` times

`QB(n)`: dequeue an item from `Q`, pushing it on `B`; repeat `n` times

`QQ(n)`: dequeue an item from `Q`, enqueue it again in `Q`; repeat n times

`AQ(n)`: pop an item from `A`, enqueue it in `Q`; repeat `n` times

`BQ(n)`: pop an item from `B`, enqueue it in `Q`; repeat `n` times

`AB(n)`: pop an item from `A`, push it on `B`; repeat `n` times

`BA(n)`: pop an item from `B`, push it on `A`; repeat `n` times.

Note that each of the above is considered a single operation, regardless of the value of `n`.

Now assume that the queue is already populated with numbers and that both stacks are empty, what is the minimum number of operations needed to have the same numbers in the queue but sorted in an ascending order? (smallest in front.)

For example, the queue `(4 3 1 2 0)` where 4 is at the front, can be sorted in three steps as follows: `QA(2)`, `QQ(2)`, then AQ(2). The queue (5 4 1 3 2 0) can be sorted in four operations as follows: QB(2), QQ(1), QB(2), BQ(4).

Write a program that determines the minimum number of operations needed to sort a given queue.

Input Format

Your program will be tested on a number of test cases. Each test case is specified on a single line. Each test case is made of $N + 1$ integers. The first integer specifies N which is the number of elements in the queue. A queue of N elements will have in it the integers from 0 to $N - 1$ in some random order. The integers in the queue are specified from the front of the queue to the back. No queue will have more than 10 elements.

The end of the test cases is identified with an input line that contains a single integer $N = 0$ (which is not part of the test cases.)

Output Format

For each test case, write the result on a separate line.

Sample Input/Output

`queue.in`

```
5 4 3 1 2 0
6 5 4 1 3 2 0
0
```

`OUTPUT`

```
3
4
```

[E] Still Johnny Can't Add

Program:	`add.(c|cpp|java)`
Input:	`add.in`
Balloon Color:	Red

Description

One way for young children in elementary schools to practice addition is to make them write down an addition table. An addition table of size N is an $(N+1)*(N+1)$ square matrix, where the top row and the left column are labeled with some random integers, (except for their intersection cell where we normally put the plus sign.) The child's task is now to put in each cell the result of adding the label of the row, and the label of the column. For example, the table on the right is an addition table of size 3.

+	**3**	**-2**	**5**
1	4	-1	6
4	7	2	9
-2	1	-4	3

Once students grow up and enter intermediate-level schools, we can give them the opposite. Give them an $N*N$ table, and let them decide how to add labels for it to be a valid addition table (if it is indeed an addition table.) Given an $N*N$ table, which does not include any labels, your job is to decide whether it is possible to properly label it or not. We're not interested in the labels themselves, just decide if it is an addition table or not. For example, the $2*2$ table on the left is not an addition table, while the one on the right is.

1	4
3	5

3	6
2	5

Input Format

Your program will be tested on one or more test cases. The first line in the input is an integer `D` representing the number of cases. The first line of each test case is an integer N, where $N \leq 10$, representing the size of the table. Following that, there will be N lines, each with N integers representing the $N*N$ table in a row-major format. Each number in the table is between -10,000 and 10,000 (inclusive).

Output Format

For each test case, output the result on a single line using the following format:

`k.␣result`

Where `k` is the test case number (starting at 1,) and `result` is `"YES"` if the test case is an addition table, or `"NO"` if it's not.

Sample Input/Output

add.in

```
3
3
4 -1 6
7  2 9
1 -4 3
2
1 4
3 5
2
3 6
2 5
```

OUTPUT

```
1. YES
2. NO
3. YES
```

[F] Newton's Apple

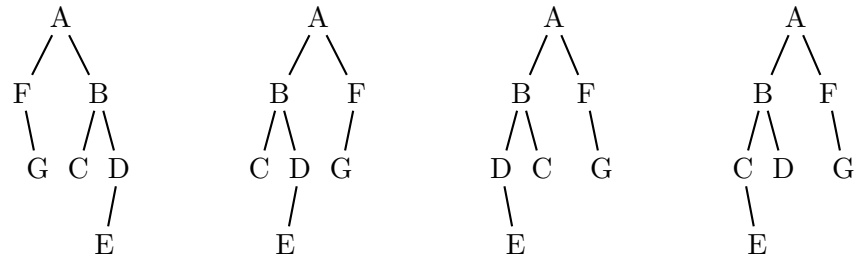
Program:	`newton.(c|cpp|java)`
Input:	`newton.in`
Balloon Color:	Purple

Description

Two binary trees (called `A` and `B`) are *equivalent* if and only if one of the following two conditions holds:

1. Both trees are empty. Or,
2. The root nodes of both trees are equal, and either:
   (a) the left subtree of `A` is *equivalent* to the left subtree of `B`, and the right subtree of `A` is *equivalent* to the right subtree of `B`. Or,
   (b) the left subtree of `A` is *equivalent* to the right subtree of `B`, and the right subtree of `A` is *equivalent* to the left subtree of `B`.

For example, the three trees on the left of the following figure are all equivalent to each other but none is equivalent to the right-most tree.



Write a program that determines if two given binary trees are equivalent or not.

Input Format

Your program will be tested on a number of test cases. The first line of the input contains an integer D which represents the number of test cases. Each test case is specified on two lines. The first line specifies the first tree, with the second tree specified on the second line. Each tree is specified using a left-to-right *postfix* notation where each empty subtree is explicitly specified using the keyword `nil`. All data in the tree are upper-case letters. The end of the line is specified using the keyword `end`. For example, the tree on the left of the figure is specified as:

```
nil nil nil G F nil nil C nil nil E nil D B A end
```

Output Format

For each test case, print on a separate line, the word `"true"` if the two trees are equivalent. Otherwise print `"false"`.

The two test cases in the following sample I/O represent the four trees drawn on the previous page (from left to right)

Sample Input/Output

newton.in

```
2
nil nil nil G F nil nil C nil nil E nil D B A end
nil nil C nil nil E nil D B nil nil G nil F A end
nil nil nil E D nil nil C B nil nil nil G F A end
nil nil nil E C nil nil D B nil nil nil G F A end
```

OUTPUT

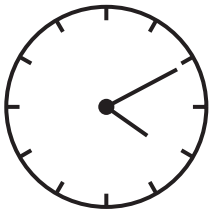
```
true
false
```

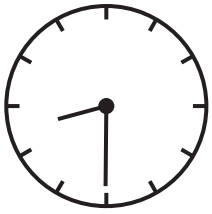
[G] The Euclidian Clock

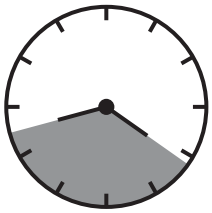
Program:	`clock.(c|cpp|java)`
Input:	`clock.in`
Balloon Color:	`Yellow`

Description

Consider a classical 2-arm 12-hour clock. Now imagine a really precise one. One that can indicate the time in hours, minutes, seconds, and hundredths of a second. Such clock can specify the time between 0:0:0.00 and 11:59:59.99 inclusive. (We'll use the format hour:minute:second.hundredths) to write the time displayed on such clock. Now, you are given two identical clocks of this type, each of them showing some time where the first is strictly before the second. You also know the radius of the clock face. We're interested in computing the area of the clock face determined by the two small (hour) clock-arms on the two clocks. The area starts from the position of the first hour clock-arm and continues in the clockwise way till the position of the second hour clock-arm.







Input Format

Your program will be tested on one or more test cases. The first line in the input specifies a single integer `D` representing the number of test cases. The first line will be followed by `D` test cases, where each case is specified on three lines. The first line in a test case specifies four integers denoting the time on the first clock using the following format:

```
H M S U
```

with `H` for hours, `M` for minutes, `S` for seconds, and `U` for hundredth of a second. Note that $0 \leq H < 12$, $0 \leq M < 60$, $0 \leq S < 60$, and $0 \leq U < 100$. The second line of a test case specifies the time on the second clock, using the same format as the first.
The third line specifies a real number denoting the radius of the clock. The maximum for the radius is 10,000.

Output Format

For each test case, output the result on a single line using the following format:

```
k.␣f
```

Where `k` is the test case number (starting at 1,) and `f` is the answer, rounded to three decimal places.

Sample Input/Output

clock.in

```
2
4 10 20 55
8 30 5 10
20.5
3 58 58 44
10 10 7 22
22.25
```

OUTPUT

```
1. 476.286
2. 801.720
```

[H] Chop Ahoy! Revisited!

Program:	`chop.(c|cpp|java)`
Input:	`chop.in`
Balloon Color:	`Cyan (light blue)`

Description

Given a non-empty string composed of digits only, we may group these digits into sub-groups (but maintaining their original order) if, for every sub-group but the last one, the sum of the digits in a sub-group is less than or equal to the sum of the digits in the sub-group immediately on its right. Needless to say, each digit will be in exactly one sub-group.

For example, the string `635` can only be grouped in one sub-group `[635]` or in two sub-groups as follows: `[6-35]` (since $6 < 8$.)

Another example is the string `1117` which can be grouped in one sub-group `[1117]` or as in the following: `[1-117]`, `[1-1-17]`, `[1-11-7]`, `[1-1-1-7]`, `[11-17]`, and `[111-7]` but not any more, hence the total number of possibilities is 7.

Write a program that computes the total number of possibilities of such groupings for a given string of digits.

Input Format

Your program will be tested on a number of test cases. Each test case is specified on a separate line. Each line contains a single string no longer than 25, and is made of decimal digits only.

The end of the test cases is identified by a line made of the word `"bye"` (without the quotes.) Such line is not part of the test cases.

Output Format

For each test case, write the result using the following format:

`k.␣n`

where `k` is the test case number (starting at 1,) and `n` is the result of this test case.

Sample Input/Output

chop.in

```
635
1117
9876
bye
```

OUTPUT

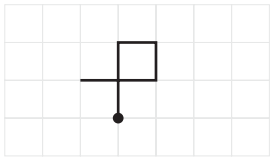
```
1. 2
2. 7
3. 2
```

[I] What's your Logo?

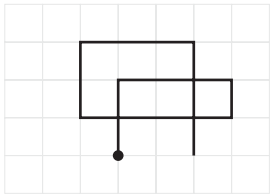
Program:	logo.(c|cpp|java)
Input:	logo.in
Balloon Color:	Black

Description

Imagine a 2D diagram drawn in the following way: Starting at the origin, you're given a sequence of letters which is entirely made of the following four letters 'U', 'D', 'L', and 'R'. A 'U' is an instruction for you to move one unit upward and drawing a segment at the same time. Similarly, 'D' is for moving down, 'L' for left, and 'R' for right. For example, figure (a) is drawn by giving the sequence 'UURDLL' while figure (b) is the result of 'UURRRDLLLLUURRRDDD' (in both figures, the starting point is identified by a small circle.)
While segments are allowed to intersect, they're not allowed to overlap. In other words, any two segments will have, at most, one point in common. We're interested in knowing the number of closed polygons, not containing any lines inside, in such diagrams. Figure (a), has only one closed polygon while figure (b) has three. Write a program to do exactly that.



(a)



(b)

Input Format

Your program will be tested on one or more test cases. Each test case is specified on a separate line. The diagram is specified using a sequence made entirely of (U|D|L|R) and terminated by the letter 'Q'. All letters are capital letters. None of the segments in a test case will overlap. The end of test cases is identified by the letter 'Q' on a line by itself.

Output Format

For each test case, write the answer on a separate line.

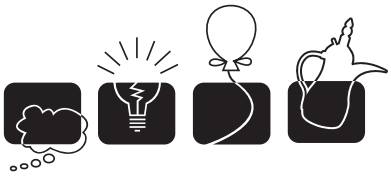
Sample Input/Output

logo.in

```
UURDLLQ
UURRRDLLLLUURRRDDDQ
Q
```

OUTPUT

```
1
3
```



[J] It's All About Three

Program:	three.(c|cpp|java)
Input:	three.in
Balloon Color:	Blue

Description

Consider the series of numbers whose all prime factors have 3 as their least (right-most) digit. For example, the first 10 numbers in this series are:

```
3 9 13 23 27 39 43 53 69 73
```

The numbers 3, 13, 23, 43, 53, and 73 are in this series since they're all primes whose least digit is a 3. Whereas 9 $(3 * 3,)$ 27 $(3 * 3 * 3,)$ 39 $(3 * 13,)$ and 69 $(23 * 3)$ are in since all their prime factors have a 3 as their least digit.

Write a program that takes a list of positive integers and determines if each integer is in this series or not.

Input Format

The input file contains a list of one or more positive integers, each given on a separate line. Each integer is less than a million. The last line of the file contains a -1 (which is not part of the list.)

Output Format

For each number in the input, write, on a separate line, the number itself followed by the word "YES" if the number is in the series described above, or "NO" if it isn't. Separate the number from the answer by a single space.

Sample Input/Output

three.in

```
3
13
33
-1
```

OUTPUT

```
3 YES
13 YES
33 NO
```