Programming Contest

Problems

October 12, 2013

All solutions:
Read from standard input
Write to standard output

Sponsored by

The Association for Computing Machinery
University of Arizona, Student Chapter

University of Arizona
Department of Computer Science

Microsoft

1. Calce

Source file: **`calce.{c, java, cpp}`**

Input file: **`none`**

Output: **`{stdout, System.out, cout}`**

Background

A simple mathematical formula for *e* is

$$e = \sum_{i=0}^{n} \frac{1}{i!}$$

where *n* is allowed to go to infinity. This can actually yield very accurate approximations of *e* using relatively small values of *n*.

The Input

There is no input for this problem.

The Output

Output the approximations of *e* generated by the above formula for the values of *n* from 0 to 9. The beginning of your output shoudl appear similar to that shown below.

Sample Input

None

Sample Output

```
n e
- -----------
0 1
1 2
2 2.5
3 2.666666667
4 2.708333334
```

2. Candy

Source file: **`candy.{c, java, cpp}`**

Input file: **`{stdin, System.in, cin}`**

Output: **`{stdout, System.out, cout}`**

A number of students sit in a circle facing their teaching in the center. Each student initially has an even number of pieces of candy. When the teacher blows a whistle, each student simultaneously gives half of his or her candy to the neighbor on the right. Any student who ends up with an odd number of pieces of candy is given another piece by the teacher. The game ends when all students have the same number of pieces of candy.

Write a program which determines the number of times the teacher blows the whistle and the final number of pieces of candy for each student from the amount of candy each child starts with.

Input

The input may describe more than one game. For each game, the input begins with the number N of students, followed by N (even) candy counts for the children counter-clockwise around the circle. The input ends with a student count of 0. Each input number is on a line by itself.

Output

For each game, output the number of rounds of the game followed by the amount of candy each child ends up with, both on one line.

Sample Input

```
6
36
2
2
2
2
2
11
22
20
18
16
14
12
10
8
6
4
2
4
2
4
6
8
0
```

Sample Output

```
15 14
17 22
4 8
```

Notes:

The game ends in a finite number of steps because:
1. The maximum candy count can never increase.
2. The minimum candy count can never decrease.
3. No one with more than the minimum amount will ever decrease to the minimum.
4. If the maximum and minimum candy count are not the same, at least one student with the minimum amount must have their count increase.

3. Cyclic: Round and Round We Go

Source file:	`cyclic.{c, java, cpp}`
Input file:	`{stdin, System.in, cin}`
Output:	`{stdout, System.out, cout}`

A *cyclic number* is an integer *n* digits in length which, when multiplied by any integer from 1 to *n*, yields a “cycle” of the digits of the original number. That is, if you consider the number after the last digit to “wrap around” back to the first digit, the sequence of digits in both numbers will be the same, though they may start at different positions.

For example, the number 142857 is cyclic, as illustrated by the following:

```
142857 * 1 = 142857
142857 * 2 = 285714
142857 * 3 = 428571
142857 * 4 = 571428
142857 * 5 = 714285
142857 * 6 = 857142
```

Write a program which will determine whether or not numbers are cyclic. The input file is a list of integers from 2 to 60 digits in length. (Note that preceding zeros should not be removed, they are considered part of the number and count in determining *n*. Thus, “01” is a two-digit number, distinct from “1” which is a one-digit number.)

Sample Input

```
142857
142856
142858
01
0588235294117647
```

Sample Output

```
142857 is cyclic
142856 is not cyclic
142858 is not cyclic
01 is not cyclic
0588235294117647 is cyclic
```

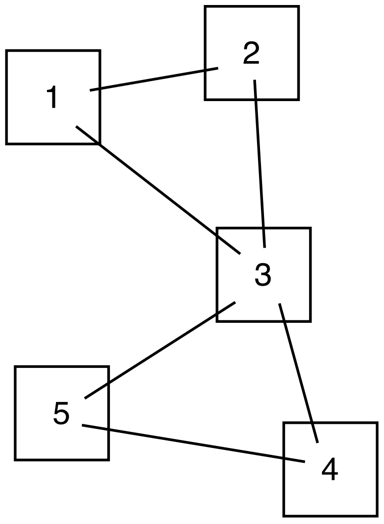
4. Single Point of Failure

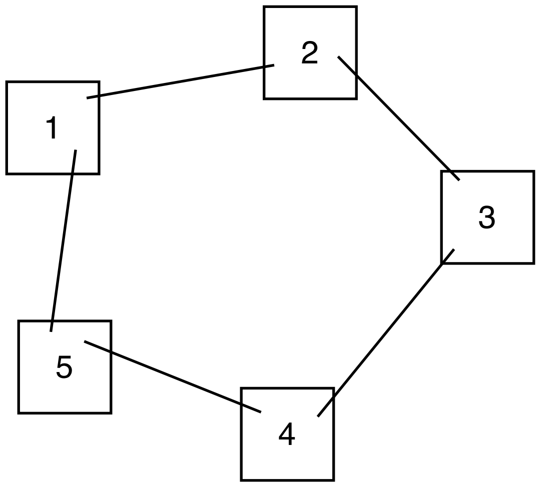
Source file: **`fail.{c, java, cpp}`**

Input file: **`{stdin, System.in, cin}`**

Output: **`{stdout, System.out, cout}`**

Consider the two networks shown below. Assuming that data moves around these networks only between directly connected nodes on a peer-to-peer basis, a failure of a single node, 3, in the network on the left would prevent some of the still available nodes from communicating with each other. Nodes 1 and 2 could still communicate with each other as could nodes 4 and 5, but communication between any other pairs of nodes would no longer be possible. Node 3 is therefore a Single Point of Failure (SPF) for this network. Strictly, an SPF will be defined as any node that, if unavailable, would prevent at least one pair of available nodes from being able to communicate on what was previously a fully connected network. Note that the network on the right has no such node; there is no SPF in the network. At least two machines must fail before there are any pairs of available nodes which cannot communicate.





Input

The input will contain the description of several networks. A network description will consist of pairs of integers, one pair per line, that identify connected nodes. Ordering of the pairs is irrelevant; 1 2 and 2 1 specify the same connection. All node numbers will range from 1 to 1000. A line containing a single zero ends the list of connected nodes. An empty network description flags the end of the input. Blank lines in the input file should be ignored.

Output

For each network in the input, you will output its number in the file, followed by a list of any SPF nodes that exist. The first network in the file should be identified as "Network #1", the second as "Network #2", etc. For each SPF node, output a line, formatted as shown in the examples below, that identifies the node and the number of fully connected subnets that remain when that node fails. If the network has no SPF nodes, simply output the text "No SPF nodes" instead of a list of SPF nodes.

Sample Input

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

1 2
2 3
3 4
4 5
5 1
0

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

0
```

Sample Output

```
Network #1
  SPF node 3 leaves 2 subnets
Network #2
  No SPF nodes
Network #3
  SPF node 2 leaves 2 subnets
  SPF node 3 leaves 2 subnets
```

5. To the Max

Source file: **`maxsum.{c, java, cpp}`**

Input file: **`{stdin, System.in, cin}`**

Output: **`{stdout, System.out, cout}`**

Given a two-dimensional array of positive and negative integers, a *sub-rectangle* is any contiguous sub-array of size 1 × 1 or greater located within the whole array. The sum of a rectangle is the sum of all the elements in that rectangle. In this problem the sub-rectangle with the largest sum is referred to as the *maximal sub-rectangle*.

As an example, the maximal sub-rectangle of the array:

```
  0 –2 –7  0
  9  2 –6  2
 -4  1 –4  1
 -1  8  0 –2
```

is in the lower left corner:

```
  9 2
 -4 1
 -1 8
```

and has a sum of 15.

Input

The input consists of an *N* × *N* array of integers. The input begins with a single positive integer *N* on a line by itself, indicating the size of the square two-dimensional array. This is followed by N^2 integers separated by whitespace (spaces and newlines). These are the N^2 integers of the array, presented in row-major order. That is, all numbers in the first row, left to right, then all numbers in the second row, left to right, etc. *N* may be as large as 100. The numbers in the array will be in the range [-127,127].

Output

Output the sum of the maximal sub-rectangle.

Sample Input

```
4
0 –2 –7  0  9  2 –6  2
-4  1 –4  1 –1

8  0 -2
```

Sample Output

```
15
```

6. N-Credible Mazes

Source file: **`maze.{c, java, cpp}`**

Input file: **`{stdin, System.in, cin}`**

Output: **`{stdout, System.out, cout}`**

An n-tersection is defined as a location in n-dimensional space, n being a positive integer, having all non-negative integer coordinates. For example, the location (1,2,3) represents an n-tersection in three dimensional space. Two n-tersections are said to be adjacent if they have the same number of dimensions and their coordinates differ by exactly 1 in a single dimension only. For example, (1,2,3) is adjacent to (0,2,3) and (2,2,3) and (1,2,4), but not to (2,3,3) or (3,2,3) or (1,2). An n-teresting space is defined as a collection of paths between adjacent n-tersections. Finally, an n-credible maze is defined as an n-teresting space combined with two specific n-tersections in that space, one of which is identified as the starting n-tersection and the other as the ending n-tersection.

Input

The input file will consist of the descriptions of one or more n-credible mazes. The first line of the description will specify *n*, the dimension of the n-teresting space. (For this problem, *n* will not exceed 10, and all coordinate values will be less than 10.) The next line will contain 2*n* non-negative integers, the first *n* of which describe the starting n-tersection, least dimension first, and the next *n* of which describe the ending n-tersection. Next will be a non-negative number of lines containing 2*n* non-negative integers each, identifying paths between adjacent n-tersections in the n-teresting space. The list is terminated by a line containing only the value –1. Several such maze descriptions may be present in the file. The end of the input is signaled by space dimension of zero. No further data will follow this terminating zero.

Output

For each maze output it's position in the input; e.g. the first maze is "Maze #1", the second is "Maze #2", etc. If it is possible to travel through the n-credible maze's n-teresting space from the starting n-tersection to the ending n-tersection, also output "can be travelled" on the same line. If such travel is not possible, output "cannot be travelled" instead.

(Sample Input and Output are on the back)

Sample Input

```
2
0 0 2 2
0 0 0 1
0 1 0 2
0 2 1 2
1 2 2 2
-1
3
1 1 1 1 2 3
1 1 2 1 1 3
1 1 3 1 2 3
1 1 1 1 1 0
1 1 0 1 0 0
1 0 0 0 0 0
-1
0
```

Sample Output

```
Maze #1 can be travelled
Maze #2 cannot be travelled
```

7. Rhyme Schemes

Source file: **`rhyme.{c, java, cpp}`**

Input file: **`{stdin, System.in, cin}`**

Output: **`{stdout, System.out, cout}`**

The *rhyme scheme* for a poem (or stanza of a longer poem) tells which lines of the poem rhyme with which other lines. For example, a *limerick* such as

> If computers that you build are quantum
> Then spies of all factions will want 'em Our codes will all fail
> And they'll read our email
> 'Til we've crypto that's quantum and daunt 'em
>
> by Jennifer and Peter Shor
>
> R(n,m) = 0 for n < 0 or m < 0 or m > n

Has a rhyme scheme of *aabba*, indicating that the first, second and fifth lines rhyme and the third and fourth lines rhyme.

For a poem or stanza of four lines, there are 15 possible rhyme schemes:

aaaa, *aaab*, *aaba*, *aabb*, *aabc*, *abaa*, *abab*, *abac*, *abba*, *abbb*, *abbc*, *abca*, *abcb*, *abcc*, and *abcd*.

Write a program to compute the number of rhyme schemes for a poem or stanza of *N* lines where *N* is an input value.

Input

Input will consist of a sequence of integers *N*, one per line, ending with a 0 (zero) to indicate the end of the data. *N* is the number of lines in a poem.

Output

For each input integer *N*, your program should output the value of *N*, followed by a space, followed by the number of rhyme schemes for a poem with *N* lines as a decimal integer with at least 12 correct significant digits (use double precision floating point for your computations).

Sample Input	Sample Output
`1`	`1 1`
`2`	`2 2`
`3`	`3 5`
`4`	`4 15`
`20`	`20 51724158235372`
`30`	`30 846749014511809120000000`
`10`	`10 115975`
`0`	

8. Digital Roots

Source file: **`roots.{c, java, cpp}`**

Input file: **`{stdin, System.in, cin}`**

Output: **`{stdout, System.out, cout}`**

The *digital root* of a positive integer is found by summing the digits of the integer. If the resulting value is a single digit then that digit is the digital root. If the resulting value contains two or more digits, those digits are summed and the process is repeated. This is continued as long as necessary to obtain a single digit.

For example, consider the positive integer 24. Adding the 2 and the 4 yields a value of 6. Since 6 is a single digit, 6 is the digital root of 24. Now consider the positive integer 39. Adding the 3 and the 9 yields 12. Since 12 is not a single digit, the process must be repeated. Adding the 1 and the 2 yields 3, a single digit and also the digital root of 39.

Input

The input file will contain a list of positive integers, one per line. The end of the input will be indicated by an integer value of zero.

Output

For each integer in the input, output its digital root on a separate line of the output.

Sample Input

```
24
39
0
```

Sample Output

```
6
3
```