

# ACM International Collegiate Programming Contest 2001–2002

Sponsored by IBM

## Southwestern European Regional Contest

<http://swerc.up.pt/>

### Problem Set



University of Porto, Portugal

November 18th, 2001

This problem set should contain nine (9) problems on twenty-two (22) numbered pages. If something is missing from your problem set, please inform a runner immediately.

# ACM International Collegiate Programming Contest 2001–2002

## Southwestern European Regional Contest

University of Porto, Portugal  
November 18th, 2001

### Contents

<b>Problem A:</b> Mice and Maze . . . . .	3
<b>Problem B:</b> Multiple Morse Matches . . . . .	5
<b>Problem C:</b> Maya Calendar . . . . .	7
<b>Problem D:</b> Water Shortage . . . . .	11
<b>Problem E:</b> Puzzle . . . . .	13
<b>Problem F:</b> Reliable Programs . . . . .	15
<b>Problem G:</b> Binary Stirling Numbers . . . . .	17
<b>Problem H:</b> Project File Dependencies . . . . .	19
<b>Problem I:</b> No Change . . . . .	21

### Remark

This problem set was developed jointly by the University of Porto and University of Warsaw for the SWERC'2001 and CERC'2001 regional contests.

# Problem A

## Mice and Maze

A set of laboratory mice is being trained to escape a maze. The maze is made up of cells, and each cell is connected to some other cells. However, there are obstacles in the passage between cells and therefore there is a time penalty to overcome the passage. Also, some passages allow mice to go one-way, but not the other way round.

Suppose that all mice are now trained and, when placed in an arbitrary cell in the maze, take a path that leads them to the exit cell in minimum time.

We are going to conduct the following experiment: a mouse is placed in each cell of the maze and a count-down timer is started. When the timer stops we count the number of mice out of the maze.

### Problem

Write a program that, given a description of the maze and the time limit, predicts the number of mice that will exit the maze. Assume that there are no bottlenecks in the maze, i.e. that all cells have room for an arbitrary number of mice.

### Input specification

The maze cells are numbered  $1, 2, \dots, N$ , where  $N$  is the total number of cells. You can assume that  $N \leq 100$ .

The first three input lines contain  $N$ , the number of cells in the maze,  $E$ , the number of the exit cell, and the starting value  $T$  for the count-down timer (in some arbitrary time unit).

The fourth line contains the number  $M$  of connections in the maze, and is followed by  $M$  lines, each specifying a connection with three integer numbers: two cell numbers  $a$  and  $b$  (in the range  $1, \dots, N$ ) and the number of time units it takes to travel from  $a$  to  $b$ .

Notice that each connection is one-way, i.e., the mice can't travel from  $b$  to  $a$  unless there is another line specifying that passage. Notice also that the time required to travel in each direction might be different.

### Output specification

The output consists of a single line with the number of mice that reached the exit cell  $E$  in at most  $T$  time units.

**Sample input**

```
4
2
1
8
1 2 1
1 3 1
2 1 1
2 4 1
3 1 1
3 4 1
4 2 1
4 3 1
```

**Sample output**

```
3
```

# Problem B

## Multiple Morse Matches

Before the digital age, the most common “binary” code for radio communication was the *Morse code*. In Morse code, symbols are encoded as sequences of short and long pulses (called *dots* and *dashes*, respectively). The following table reproduces the Morse code for the alphabet, where dots and dashes are represented by ASCII characters “.” and “-”:

A	.-	B	-...	C	-.-. .	D	-..
E	.	F	..-. .	G	--.	H	....
I	..	J	.---	K	-. -	L	.-..
M	--	N	-. .	O	---	P	.---
Q	--.-	R	.-. .	S	... .	T	-
U	..- .	V	...- .	W	.-.-	X	-.-. .
Y	-.--	Z	--..				

Notice that in the absence of pauses between letters there might be multiple interpretations of a Morse sequence. For example, the sequence “-.-.-.--” can be decoded both as “CAT” or “NXT” (among others). A human Morse operator would use other context information (such as a language dictionary) to decide the appropriate decoding.

### Problem

Write a program that reads a Morse code string and a list of words (a *dictionary*) and attempts to parse the Morse code into a phrase using words that occur in the dictionary. The program’s output should be the number of distinct phrases that can be obtained.

Notice that we are interested in *full matches*, i.e. the complete Morse string must be matched to words in the dictionary.

### Input specification

The input starts with the Morse code string, made up of characters “.” and “-”, with no spaces between them, and terminated by the end-of-line character.

The next line consists of the number  $N$  of words in the dictionary, and is followed by  $N$  dictionary words, one in each line. Each word consists of upper-case characters from “A” to “Z” only.

The number of words in the dictionary is less than or equal to 10 000 and the Morse code string is at most 1000 characters long.

### Output specification

The output should be a single integer number representing the count of distinct phrases into which the Morse code can be parsed. You may assume that this number is at most  $2 \times 10^9$ .



# Problem C

## Maya Calendar

The classical Maya civilization prospered in what today is southern Mexico, Guatemala, Belize and northern Honduras. During the height of the Maya civilization they developed a sophisticated system for time keeping used both to record history and for divinatory rituals. Their calendar consisted of three components: the Tzolkin, the Haab and the Long Count.

For divinatory purposes the Maya used the Tzolkin which was composed of 20 day names to which a numeric coefficient from 1 to 13 was attached, giving a total of 260 distinct combinations. This is the size of the Tzolkin, or ritual year. From Spanish colonial sources, we know the names of the days:

**Day names:** IMIX, IK, AKBAL, KAN, CHIKCHAN, KIMI, MANIK, LAMAT, MULUK, OK, CHUEN, EB, BEN, IX, MEN, KIB, KABAN, ETZNAB, KAWAK, AJAW

For example, the sequence of days starting at 9.IMIX is: 9.IMIX / 10.IK / 11.AKBAL / 12.KAN / 13.CHIKCHAN / 1.KIMI / 2.MANIK / . . .

The Haab calendar was used for astronomy. It had 365 days divided into 19 months each with 20 days, except the last one which had only 5. In a manner similar to the Tzolkin each month name has a number from 1 to 20 indicating the day number within the month. Again, from Spanish colonial sources, we know the names of the months:

**Month names:** POHP, WO, SIP, ZOTZ, SEK, XUL, YAXKIN, MOL, CHEN, YAX, SAK, KEH, MAK, KANKIN, MUAN, PAX, KAYAB, KUMKU, WAYEB

The month WAYEB had just 5 days and was considered an unlucky time of the year.

The Tzolkin and Haab were combined in the inscriptions to create the so called Calendar Round, combining the 260 day cycle of the Tzolkin and the 365 day cycle of the Haab. A typical Calendar Round date in the inscriptions might be: “3.LAMAT 6.PAX”. Note that not all of the combination of days, months and coefficients are possible. How many days does it take to repeat a Calendar Round?

A typical sequence of days in the Calendar Round starting for example at “3.LAMAT 6.PAX”:

3.LAMAT 6.PAX / 4.MULUK 7.PAX / 5.OK 8.PAX / 6.CHUEN 9.PAX / 7.EB 10.PAX /  
8.BEN 11.PAX / 9.IX 12.PAX / 10.MEN 13.PAX / 11.KIB 14.PAX / 12.KABAN 15.PAX /  
13.ETZNAB 16.PAX / 1.KAWAK 17.PAX / 2.AJAW 18.PAX / 3.IMIX 19.PAX / 4.IK 20.PAX /  
5.AKBAL 1.KAYAB / 6.KAN 2.KAYAB / . . .

Finally, at the beginning of the Classic Period (AD 200–900) the Maya developed an absolute calendar called Long Count which counted the number of days starting from a fixed date. Currently, most researchers agree that this zero date was August 13, 3114 BC. According to Maya belief this was the date of creation of our world. Dates in the Long Count are written (for simplicity) in 5-tuples of the form: “9.2.3.4.5”. Such a date reads “9 baktuns 2 katuns 3 tuns 4 winals 5 kin since

the zero date". A "kin" is just one day. A "winal" is a group of 20 days. A "tun" is a group of 18 winals (thus a tun has  $20 \times 18 = 360$  days, 5 days short of a year). From here on all units come in multiples of 20. Thus a "katun" is 20 tuns (almost 20 years) and a "baktun" is 20 katuns (almost 400 years). Thus the date "9.2.3.4.5" means " $9 \times 144000 + 2 \times 7200 + 3 \times 360 + 4 \times 20 + 5$  days after the zero date".

Given the periodicity of the Calendar Round, a legal date such as "3.LAMAT 6.PAX" has multiple occurrences in the Long Count. Thus, one difficulty in reading Maya inscriptions is establishing the correspondence between a date given only in the Calendar Round and the absolute date in the Long Count. In this case, we must compute all the possible Long Count dates associated with the particular Calendar Round and deduce which one applies based on context information (for example, using references to a king whose lifespan is known).

## Problem

Write a program that computes all possible Long Count dates corresponding to a given Calendar Round date. Only the Long Count dates in the Baktuns 8 and 9 are of interest to us (they cover all the Classic Period).

As a starting point, you are given the information that the Long Count date 8.0.0.0.0 occurred on the Calendar Round "9.AJAW 3.SIP".

## Input specification

The input consists of one Calendar Round date in the following format:

`dayNumber.dayName dayNumber.monthName`

The day and month names are written with an upper-case first letter and lower-case letters afterwards.

## Output specification

Your output should be the corresponding sequence of Long Count dates in the Classic Period, in ascending order, each displayed with the format "baktun.katun.tun.winal.kin", separated by newlines.

If there are no corresponding Long Count date for the given Calendar Round date, your output should be "NO SOLUTION".

## Sample Input

3.Lamat 6.Pax

## Sample Output

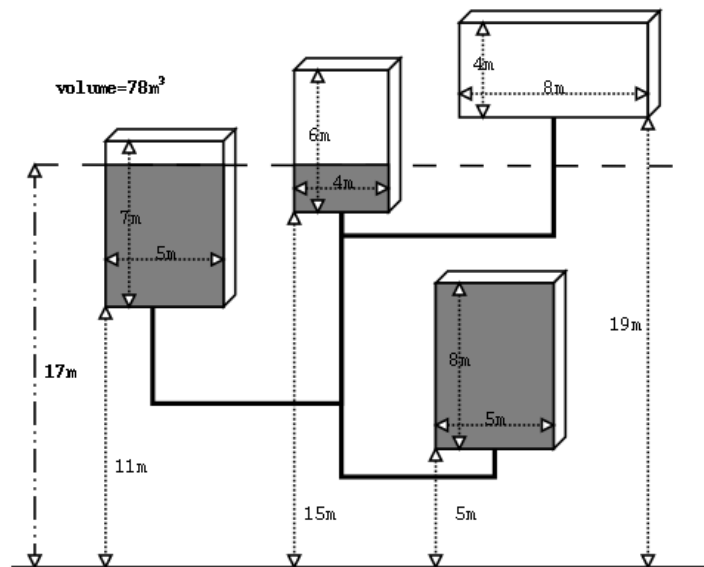
8.0.17.17.8  
8.3.10.12.8  
8.6.3.7.8  
8.8.16.2.8  
8.11.8.15.8  
8.14.1.10.8  
8.16.14.5.8  
8.19.7.0.8  
9.1.19.13.8  
9.4.12.8.8  
9.7.5.3.8  
9.9.17.16.8  
9.12.10.11.8  
9.15.3.6.8  
9.17.16.1.8



# Problem D

## Water Shortage

During the next century certain regions on earth will experience severe water shortages. The old town of Uqbar has already started to prepare itself for the worst. Recently they created a network of pipes connecting the cisterns that distribute water in each neighborhood, making it easier to fill them at once from a single source of water. But, in case of water shortage, the cisterns above a certain level will be empty since the water will flow to the cisterns below.



You have been asked to write a program to compute the level to which cisterns will be filled with a certain volume of water, given the dimensions and position of each cistern. To simplify we will neglect the volume of water in the pipes.

### Input specification

As input you receive the number of cisterns, followed by one line per cistern with 4 floating point values: its base level, height, width and depth in meters. The last input line gives you the volume in cubic meters to be injected into the network.

### Output specification

Your program must output the level that the water will reach, in meters, truncated to two fractional digits. If the volume of water exceeds the total capacity of the cisterns then your program must write "OVERFLOW".

**Sample input**

```
4
11.0 7.0 5.0 1.0
15.0 6.0 4.0 1.0
 5.0 8.0 5.0 1.0
19.0 4.0 8.0 1.0
78.0
```

**Sample output**

```
17.00
```

# Problem E

## Puzzle

This is a puzzle game to be played by two persons, Alice and Bob. Alice draws an  $n$ -vertex convex polygon and numbers its vertices with integers  $1, 2, \dots, n$  in an arbitrary way. Then she draws a number of non-crossing diagonals (the vertices of the polygon are not considered to be crossing points). She keeps the drawing secret and tells Bob the polygon's sides and diagonals, without revealing which are which. Each side or diagonal is specified by its endpoints. Bob has to guess the order of the vertices on the border of the polygon. Help him solve the puzzle.

### Example

If  $n = 4$  and  $(1,3)$ ,  $(4,2)$ ,  $(1,2)$ ,  $(4,1)$ ,  $(2,3)$  are the endpoints of the four sides and one diagonal then the ordering of the vertices on the border of the polygon is  $1, 3, 2, 4$  (up-to shifting and reversing).

### Problem

Write a program that reads the description of sides and diagonals given to Bob by Alice, computes the order of vertices on the border of the polygon and writes the result.

### Input specification

The first two lines of the input contain two integers  $n$  and  $m$ , such that  $4 \leq n \leq 10\,000$  and  $0 \leq m \leq n - 3$ . Integer  $n$  is the number of vertices of the polygon and integer  $m$  is the number of its diagonals, respectively.

The third line contains exactly  $2(m + n)$  integers separated by single spaces, specifying all sides and some diagonals of the polygon. Integers on positions  $2j - 1$  and  $2j$ ,  $1 \leq j \leq m + n$ , specify the two endpoints of a side or a diagonal. The sides and the diagonals can be given in an arbitrary order. There are no duplicates.

### Output specification

The output should consist of a single line containing a permutation of  $1, 2, \dots, n$  separated by spaces. This sequence corresponds to the numbers of vertices on the border of the polygon; the sequence should always start with vertex number 1 and its second element should be the smaller vertex of the two neighbor vertices of vertex 1.

**Sample input**

```
4
1
1 3 4 2 1 2 4 1 2 3
```

**Sample output**

```
1 3 2 4
```

# Problem F

## Reliable Programs

Consider a machine with  $n$  integer registers  $r_1, r_2, \dots, r_n$  and a single type of *compare-exchange instruction*,  $CE(i, j)$  defined as follows, where  $1 \leq i < j \leq n$  are the register indices:

$CE(i, j)$  : if  $\text{content}(r_i) > \text{content}(r_j)$  then  
exchange the contents of registers  $r_i$  and  $r_j$ .

A compare-exchange program (shortly CE-program) is any finite sequence of compare-exchange instructions. A CE-program is called *minimum-finding* if after its execution the register  $r_1$  always contains the minimum value among all the initial values in the registers. Furthermore, such program is called *reliable* if it remains a minimum-finding program after removing any single compare-exchange instruction.

Given a CE-program  $P$ , what is the minimum number of instructions that should be added at the end of program  $P$  in order to make it reliable?

### Example

Consider the following 3-register CE-program:  $CE(1, 2); CE(2, 3); CE(1, 2)$ .

In order to make this program reliable it suffices to add only two extra instructions, namely  $CE(1, 3)$  and  $CE(1, 2)$ .

### Problem

Write a program that reads the description of a CE-program, and determines the minimum number of CE-instructions that should be added to make this program reliable.

### Input specification

The first line of input contains the number of registers  $n$ , where  $0 \leq n \leq 1000$ , followed by the number of program instructions  $m$ , where  $0 \leq m \leq 3000$ .

The next line contains the program itself: a sequence of  $2m$  integers, separated by spaces, where each CE-instruction consists of two consecutive integers on positions  $2j - 1$  and  $2j$ , with  $1 \leq j \leq m$ .

### Output specification

The output consists of a single integer: the minimal number of instructions that should be added to the input program in order to make this program reliable.

**Sample input**

```
3 3  
1 2 2 3 1 2
```

**Sample output**

```
2
```

# Problem G

## Binary Stirling Numbers

The Stirling number of the second kind  $S(n, m)$  represents the number of ways to partition a set of  $n$  things into  $m$  nonempty subsets. For example, there are seven ways to split a four-element set into two parts:

$$\{1, 2, 3\} \cup \{4\}, \{1, 2, 4\} \cup \{3\}, \{1, 3, 4\} \cup \{2\}, \{2, 3, 4\} \cup \{1\}, \\ \{1, 2\} \cup \{3, 4\}, \{1, 3\} \cup \{2, 4\}, \{1, 4\} \cup \{2, 3\}.$$

We can compute  $S(n, m)$  using the recurrence,

$$S(n, m) = mS(n-1, m) + S(n-1, m-1), \quad \text{for integers } 1 < m < n.$$

but your task is slightly different: given integers  $n$  and  $m$ , compute the parity of  $S(n, m)$ , i.e.  $S(n, m) \bmod 2$ .

### Example

$$S(4, 2) \bmod 2 = 1.$$

### Problem

Write a program that reads two positive integers  $n$  and  $m$ , computes  $S(n, m) \bmod 2$ , and writes the result.

### Input specification

The input consists two integers  $n$  and  $m$  separated by a space, with  $1 \leq m \leq n \leq 1000\,000\,000$ .

### Output specification

The output should be the integer  $S(n, m) \bmod 2$ .

### Sample input

4 2

### Sample output

1



# Problem H

## Project File Dependencies

Project managers, such as the UNIX utility `make`, are used to maintain large software projects made up from many components. Users write a *project file* specifying which components (called *tasks*) depend on others and the project manager can automatically update the components in the correct order.

### Problem

Write a program that reads a project file and outputs the order in which the tasks should be performed.

### Input specification

For simplicity we represent each task by an integer number from  $1, 2, \dots, N$  (where  $N$  is the total number of tasks). The first line of input specifies the number  $N$  of tasks and the number  $M$  of rules, such that  $N \leq 100$ ,  $M \leq 100$ .

The rest of the input consists of  $M$  *rules*, one in each line, specifying dependencies using the following syntax:

$$T_0 \quad k \quad T_1 \quad T_2 \quad \dots \quad T_k$$

This rule means that task number  $T_0$  depends on  $k$  tasks  $T_1, T_2, \dots, T_k$  (we say that task  $T_0$  is the *target* and  $T_1 \dots T_k$  are *dependents*).

Note that tasks numbers are separated by single spaces and that rules end with a newline. Rules can appear in any order, but each task can appear as target only once.

Your program can assume that there are no circular dependencies in the rules, i.e. no task depends directly or indirectly on itself.

### Output specification

The output should be a single line with the permutation of the tasks  $1 \dots N$  to be performed, ordered by dependencies (i.e. no task should appear before others that it depends on).

To avoid ambiguity in the output, tasks that do not depend on each other should be ordered by their number (lower numbers first).

**Sample input**

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

**Sample output**

```
1 5 3 2 4
```

# Problem I

## No Change

Though it might be hard to imagine, the inhabitants of a small country Additiviva do not know of such thing as change, which probably has to do with them not knowing subtraction either. When they buy something, they always need to have the exact amount of addollars, their currency. The only other option, but not a really attractive one, is over-paying.

Professor Adem, one of the Additivian mathematicians came up with an algorithm for keeping a balanced portfolio. The idea is the following. Suppose you have more coins of value  $v_1$  than coins of value  $v_2$ . In this case you should try to spend at least as many coins of value  $v_1$  as those of value  $v_2$  on any buy you make. Of course spending too many  $v_1$  coins is not a good idea either, but to make the algorithm simpler professor Adem decided to ignore the problem. The algorithm became an instant hit and professor Adem is now designing a kind of “electronic portfolio” with built-in Adem’s algorithm. All he needs now is a software for these machines, that will decide whether a given amount of addollars can be paid using a given set of coins according to the rules of Adem’s algorithm. Needless to say, you are his chosen programmer for the task.

### Problem

Write a program that reads the description of a set of coins and an amount of addollars to be paid, and determines whether you can pay that amount according to Professor Adem’s rules.

### Input specification

The input starts with the amount of addollars to be paid  $x$ , where  $1 \leq x \leq 100\,000$ . The number of different coin values  $k$  follows, where  $1 \leq k \leq 5$ . The values of the coins  $v_1, \dots, v_k$  follow, where  $1 \leq v_i \leq 10\,000$ .

Notice that the order among coin values is significant: you need to spend at least as many coins of value  $v_1$  as coins of value  $v_2$ , at least as many coins of value  $v_2$  as those of value  $v_3$ , and so on. You may assume that you have a sufficiently large number of coins of each value.

### Output specification

Your program should output for each test case either a single word “YES”, if the given amount can be paid according to the rules, or a single word “NO” otherwise.

**Sample input**

13 3 9 2 1

**Sample output**

NO