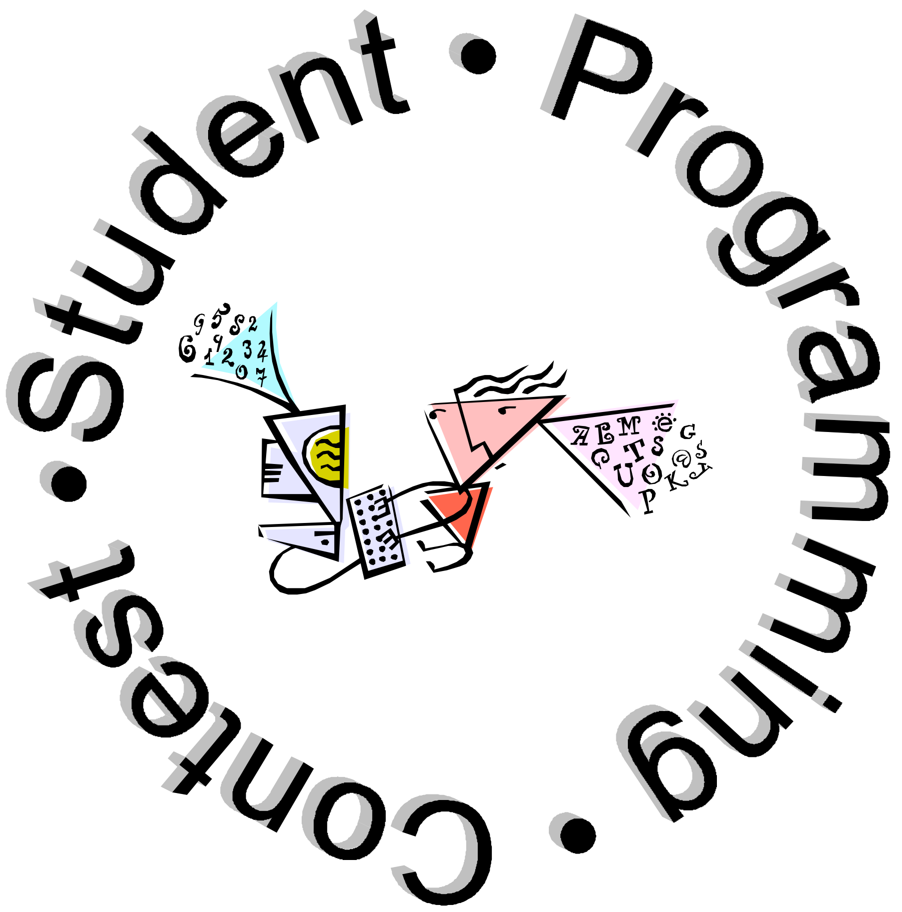
The Fifth Annual

Student Programming Contest

of the
CCSC Southeastern Region

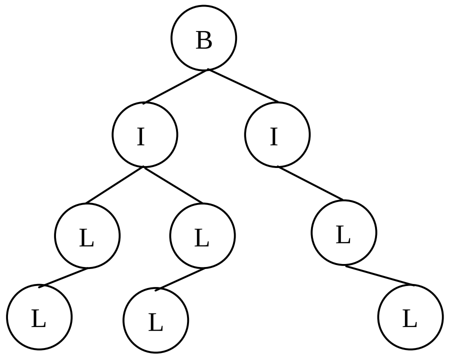


Saturday, November 7, 1998
8:00 A.M. – 12:00 P.M.

Devil in a Blue Dress

The Problem

You have just assumed your job as a co-op student programmer at the FBI Crime Lab. One of your first duties involves the microscopic analysis of a blue cocktail dress – important evidence in a high-profile case. The dress has been subjected to a sophisticated analysis process that exploits a little-known fact about human DNA. One particular strand of our DNA, when rotated into a hierarchical form, actually contains the first name of its "owner." The picture to the right is a highly magnified image of this so-called "name strand" taken from a sample of President Clinton's hair.



The exact hierarchical form of the name strand varies according to the bodily source of the DNA, but they all share this property: Paths from the "root" of the hierarchy to the "leaves" of the hierarchy represent character strings. Name strands in which all root-to-leaf paths contain the same character string (which is a person's first name) are considered to be exact matches to a suspect, while name strands in which only some of the root-to-leaf paths contain the same name are considered partial matches. Obviously name strands in which none of the root-to-leaf paths contain the same name are considered non-matching.

Analysis of the blue dress revealed that human DNA was present in various forms in several locations on the dress. The name strand for each sample has been isolated, rotated into hierarchical form, and encoded in a text file. Your task is to write a program that processes this file and classifies each DNA sample as exactly matching the suspect (in this case, President Clinton), closely matching the suspect, or not matching the suspect at all.

Sample Input

Your program must take its input from the text file `prob1.in`. This file contains an unspecified number of name strand encodings. Each encoding has two parts. The first part is a single line that contains three values in the following order: an integer N representing the number of nodes in the name strand hierarchy, an integer M representing the number of arcs (parent-child connectors) in the hierarchy, and a single character that labels the root node in the name strand hierarchy. The value of N will never be greater than 100. The second part consists of M lines, one for each parent-child connection in the hierarchy. Each of these lines contains four values. The first two are integers X and Y in the range 1..N inclusive and represent the fact that X is a parent of Y in the hierarchy. (You must assume that the nodes of each name strand hierarchy have been labeled randomly.) The third value is one of the characters 'L' or 'R' indicating whether Y is a left or right child respectively of X. The fourth and final entry on a line is an uppercase character from the English alphabet that represents the label of the Y node.

Sample contents of `prob1.in`, which encodes the name string pictured above (assuming that the nodes are labeled from 1 to 9 in top-to-bottom, left-to-right order) would be:

```
9 8 B
1 2 L I
1 3 R I
2 4 L L
2 5 R L
3 6 R L
4 7 L L
5 8 L L
6 9 R L
```

Sample Output

Your program must direct its output to the screen and must format its output exactly as shown. There must be one line of output for each DNA sample in the input file. These lines must begin by labeling the sample (e.g., “Sample I”, where I is the number of the DNA sample from the beginning of the file). After this, the sample must be categorized as an exact match, a partial match, or no match according to the following table:

Category	Appropriate Output
Exact match	`Exact match – summon the grand jury.`
Partial match	`Partial match – close, but no cigar.`
No match	`No match – sorry, Ken.`

Appropriate output for the sample input above would be:

```
Sample 1: Exact match – summon the grand jury.
```

On Target

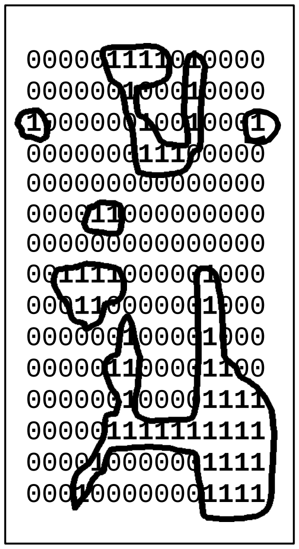
The Problem

Target identification from satellite imagery is an important application in military software systems. Data from satellite images are encoded in a machine-readable form, and software is then used to interpret this data. Your task is to write a program that inputs satellite images as a text file of zeroes and ones and returns the total number of enemy targets present in the image. Targets can be identified as any collection of one or more ones connected horizontally, vertically, or diagonally. Thus an image file consisting of all ones represents a single (very large) target.

Sample Input

Your program must take its input from the text file `prob2.in`. The file begins with a line containing a single non-negative integer N that indicates the (square) dimensions of the image (N x N). The value of N will never exceed 50. The next N lines will contain N characters each, in some combination of zeroes and ones (no blank spaces). Sample contents of `prob2.in` might be:

```
15
000001111010000
000000100010000
100000010010001
000000011100000
000000000000000
000011000000000
000000000000000
001111000001000
000110000001000
000000100001000
000001100001100
000000100001111
000001111111111
000010000001111
000100000001111
```



The figure to the right has been provided for your convenience in visually identifying the targets.

Sample Output

Your program must direct its output to the screen and format its output in exactly the following form. Appropriate output for the sample input above would be:

```
There are 6 targets in the current image.
```

Pseudo-Random, Pseudo-Hard

The Problem

Computers can normally not generate really random numbers, but frequently they are used to generate pseudo-random numbers. These numbers are in fact generated by some algorithm, but appear for all practical purposes to be random.

A common pseudo-random number generation technique is called the linear congruential method, which works as follows. If the last pseudo-random number generated was L, then the next number is generated by evaluating

$$(Z \times L + I) \bmod M$$

where Z is a constant multiplier, I is a constant increment, and M is a constant modulus. For example, suppose Z is 7, I is 5, and M is 12. If the first random number (usually called the seed) is 4, then the next random number is (7 x 4 + 5) mod 12 which is 33 mod 12 which is 9 (now the new L). Then next L's will be 8, 1, 0, 5, and 4. As you can see, the sequence of pseudo-random numbers generated by this technique repeats after six numbers (4, 9, 8, 1, 0, 5).

Your task is to write a program that takes a sequence of four integer values for Z, I, M, and L, each of which will have no more than four digits. Your program is to print out the length of the longest non-repeating sequence of pseudo-random numbers.

Sample Input

Your program must take its input from the text file `prob3.in`. This file contains an unspecified number of lines, each of which contains four integer values for Z, I, M, and L (in that order). Sample contents of `prob3.in` might be:

```
7 5 12 4
11 7 50 3
1 1 1 1
```

Sample Output

Your program must direct its output to the screen and format its output in exactly the following form. Appropriate output for the sample input above would be:

```
Z  I  M  L  Length
------------------
7  5  12 4    6
11 7  50 3   50
1  1   1 1  ERROR
```

Off the Scale

The Problem

In our culture, the twelve notes used in musical notation are arranged in the following order:

C/B♯, C♯/D♭, D, D♯/E♭, E/F♭, F/E♯, F♯/G♭, G, G♯/A♭, A, A♯/B♭, B/C♭

Note that the slash in the above list indicates alternate notations for the same note. For example, C♯and D♭ are two notations for the same note.

Any two notes that are adjacent to each other in the above list are known as a *semitone*. Any two notes that have one note separating them in the above list are known as a *tone*. A *major scale* is made up of eight notes, beginning with one of the notes in the above list and moves (toward the right in the above list) in the progression tone-tone-semitone-tone-tone-tone-semitone. For example, the major scale starting on D♭ is composed of the following notes:

D♭, E♭, F, G♭, A♭, B♭, C, D♭

Major scales contain a given note only once, except for the beginning note of the scale, which is repeated as the last note of the scale. Major scales cannot contain a combination of both flat and sharp notes.

For the purposes of this problem, you need only be concerned with the following major scales: C, D♭, D, E♭, E, F, G♭, G, A♭, A, B♭, and B.

An *interval* is the distance between two notes on a scale. It can either be ascending or descending. The names of the intervals are derived from the total number of notes involved in the interval. Two notes that are adjacent to each other (that is, with no notes separating them) in a major scale is known as a major *second*. Two notes in a major scale with three notes separating them (thus five notes in the interval) is known as a major *fifth*. An *octave* is a special interval (that is, it isn't counted like other intervals) which spans from one occurrence of a note in a major scale to the next occurrence of that note in the major scale.

Your task is to write a program that, given a major scale, computes major intervals in that scale.

Sample Input

Your program must take its input from the text file `prob4.in`. This file contains an unspecified number of pairs of lines. The first line in each pair will contain the major scale to be used. The second line in each pair will contain one or more intervals to be calculated. Each interval to be calculated will be specified with a starting note, a direction (up or down), and the interval itself (second, third, fourth, fifth, sixth, seventh, or octave). When specifying notes, the lowercase letter "b" will be used to represent a flat and the character "#" will be used to represent a sharp. Exactly one space will separate entries on the second line of the pairs. Sample contents of `prob4.in` might be:

```
C
F up second G down third
E
F# down fourth Bb down seventh B up octave
```

Sample Output

Your program must direct its output to the screen and format its output exactly as shown. Appropriate output for the sample input above is:

```
Key of C: F up second => G; G down third => E
Key of E: F# down fourth => C#; Bb is an invalid note for this key; B up octave => B
```

Perfectly Big Numbers

The Problem

If N is a positive integer, let S(N) be the sum of the proper divisors of N (i.e., divisors of N other than N itself). For example, S(12) = 1+2+3+4+6 = 16, S(7) = 1, S(6) = 1+2+3 = 6. Note that sometimes S(N) is larger than N (e.g., N=12) , sometimes smaller (e.g., N=7), and very rarely, S(N)=N (e.g., N=6). Numbers N for which S(N)=N are called *perfect numbers*, and have been a curiosity in the mathematics of number theory for over 2000 years.

Your task is to write a program that finds the first four perfect numbers and the number of digits in each. Don't try to go beyond this limit of four, because these numbers grow very quickly. For example, the ninth perfect number has thirty-seven digits.

Sample Input

There is no input for this problem.

Sample Output

Your program must direct its output to the screen and format its output in exactly the following form.

```
Perfect Number    Number of Digits
----------------------------------
     xxxx              xxxx
     xxxx              xxxx
     xxxx              xxxx
     xxxx              xxxx
```

Where `xxxx` represents the appropriate integer values.

Y2K, U2K

The Problem

A date in a given century can be expressed in the form yyddd where yy represents the last two digits of the year and ddd is a number in the range 1..365 (366 for leap years) indicating the relative number of the day measured from January 1. Such a format is called a *Julian* date. For example (assuming the 20th century), 43200 is July 19, 1943 (1943 was not a leap year) and 44201 is July 19, 1944. Julian dates are obviously not Y2K compliant.

Your task is to write a program that reads a sequence of Julian dates and converts them to the following form, which is Y2K compliant:

mm/dd/yyyy

where mm is the two digit month designation, dd is the two digit day designation, and yyyy is the four digit year designation. Assume that all Julian dates in the input are for the 20th century.

Sample Input

Your program must take its input from the text file `prob6.in`. This file contains an unspecified number of Julian dates, one per line. Sample contents of `prob6.in` might be:

```
98001
43200
67365
```

Sample Output

Your program must direct its output to the screen and format its output in exactly the following form. For each Julian date in the input file, your program must produce the corresponding Y2K-compliant date, one per line. Appropriate output for the sample input above would be:

```
01/01/1998
07/19/1943
12/31/1967
```