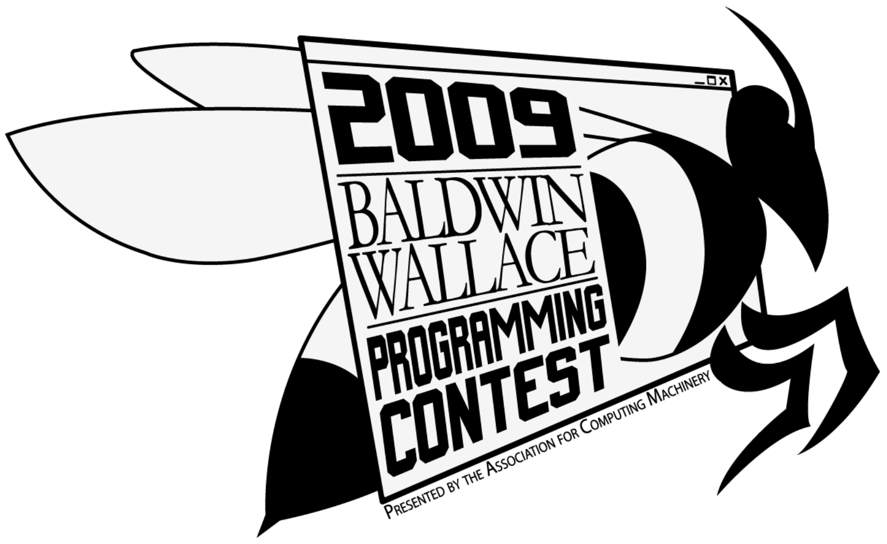
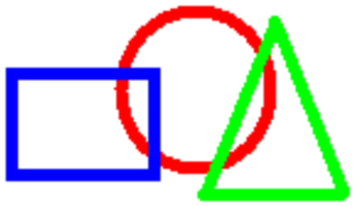
Baldwin-Wallace College

6th Annual High School Programming Contest



Do not open until instructed



Merging Shapes

A lot of graphical applications render overlapping shapes to the computer screen. For example, when windows are opened on a computer, the new window may overlay another, causing the first window to be covered in part, or in its entirety, depending upon the size and position of the newly opened window(s). Determining what image should appear on the screen in these cases requires some analysis.

If the shapes that overlap are of the same solid color, the problem becomes somewhat simpler, because the second shape does not appear to hide the first shape, but simply to merge with it. It is a variation of this simplified case that this problem will ask you to implement.

The program should merge a diamond (rhombus) and a rectangle, orienting them such that their center points coincide. The size of each rhombus will be determined by a single integer value *s* that will tell how many lines are in the rhombus from its top to its center. The top and bottom of the rhombus have only one '*', and successive rows differ by exactly two '*'s. For example, the following is the rhombus defined by the value 3:

```
  *
 ***
*****
 ***
  *
```

Note that the overall dimensions of a rhombus are always an odd number. Thus, in order for the centers of the rhombus and rectangle to coincide, the dimensions of the rectangle, given by two integer values, *l* and *w* (in that order), will always be odd numbers as well. *L* gives the length of the rectangle in number of rows and *w* its width. Thus a 3x5 rectangle looks as follows:

```
*****
*****
*****
```

When the rhombus and rectangle overlap, three outcomes can occur: The rhombus may fit entirely within the rectangle, the rectangle may fit entirely within the rhombus, or they may overlap only partially, resulting in a variety of shapes. The two figures above, when merged, result in the following shape:

```
  *
*****
*****
*****
  *
```

Your program should determine what shape the merged rhombus and rectangle create and output the outline of that shape. The leftmost boundary of your outline should appear at the extreme left of the output and there should be no extra white space, above or below the shape or at the ends of any of the individual lines of the shape.

The first line of the input will be a single integer, *n*, that will indicate the number of shapes to be drawn. Each successive line of input will contain the three integer values, *s, l,* and *w* as described above. No input value will result in a shape whose maximum dimensions exceeds 79x79. You should output each merged shape using asterisks to draw the shape. There should be no extra spaces at the ends of the lines within a figure and no blank lines between figures.

Sample input

```
3
3 3 5
5 3 3
3 5 5
```

Sample output

Timing is Everything

It is particularly important in a championship swim competition that the timing is accurate and fair. When electronic timing is not available, multiple timekeepers are assigned to each lane. Each of these timekeepers, equipped with a stopwatch, records the time of each swimmer who swims in that lane. A swimmer's official time is then determined from these individual hand times.

When three or more hand times are available, the official time is calculated by dropping the fastest and slowest times, and averaging the remaining time(s). For instance, suppose Buzzy Bee is the swimmer and his recorded times are 23.45, 24.56, 23.86, 23.13, and 25.32. In this case, 23.13 and 25.32 are dropped, and his official time is (23.45 + 23.86 + 24.56) / 3 = 23.96 sec.

For this problem, you are to write a program that reads in the hand times from a 50 m. freestyle event and reports the results, including each swimmer's name and official time, in rank order.

The first line of the input contains a single integer, *k,* which indicates the number of lines to follow. Each of the following *k* lines contains (in the given order, and separated by a blank) the first and last name of a swimmer, as well as the time for that swimmer recorded by one timekeeper. Times are reported in seconds, accurate to two decimal places. There are at least three times for each swimmer in the event, but no more than ten. There is a maximum of 100 swimmers.

Output should contain one line for each swimmer. It should include the swimmer's first name, last name, and official time (in that order), each separated by a blank. Times should be expressed rounded to two decimal places. Lines of output should appear in order by time, listing the swimmer with the fastest time first. Swimmers whose official times are the same (rounded to two decimal places) should be listed alphabetically by last name. If both time and last name are the same, those should be listed alphabetically by first name.

Sample Input

```
12
John Smith 23.12
Jane Doe 22.32
Adam Snow 25.32
Jane Doe 34.32
Adam Snow 23.43
John Smith 23.45
Jane Doe 23.47
John Smith 23.65
Adam Snow 34.34
Adam Snow 30.34
Jane Doe 23.64
John Smith 32.34
```

Sample Output

```
John Smith 23.55
Jane Doe 23.56
Adam Snow 27.83
```

Wikilations

The English version of Wikipedia, the user-editable online encyclopedia supported by the Wikimedia Foundation, currently contains over 2.7 million online articles. Some of these articles discuss very diverse topics, while others describe closely related topics. Most Wikipedia articles link to other Wikipedia articles, and as such, create a web of interconnecting articles.

Your program is to find the shortest connection between two Wikipedia articles, where distance is measured as the number of links that need to be followed in order to get to the second article from the first. For instance, if there were an article named *Baldwin-Wallace College* containing the link *Berea, Ohio* for an article which contained a link to the page *Cleveland*, then *Cleveland* would be two articles away from *Baldwin-Wallace College*. In order to navigate from any article to itself, there must be other links that lead back to the article itself. That is, the distance from an article to itself is not zero.

The input begins with a description of a set of Wikipedia articles, and ends with a listing of pairs of articles whose shortest distance your program is to determine. The first line of the input contains an integer, n ($1 \leq n \leq 20$), indicating the number of Wikipedia articles to follow. For each of these n articles, there is a set of lines with the pattern *name, number, link, link…link*. The first gives the name of the article. That is followed by an integer, m ($0 \leq m \leq 20$), indicating the number of links in that article, followed by the names of the m links.

Once these articles and their links are specified, a line containing an integer, p, is listed, indicating the number of pairs of articles to follow. There are then p pairs of articles ($2 \times p$ lines) listed, for which you are to determine the shortest distance from the first article to the second article.

Output should contain p lines, one for each pair of articles listed at the end of the input. If the second of the pair can be gotten to from the first, the output should have the form

```
      <Article two> is <x> articles away from <Article one>.
```

where `<x>` is the shortest connection from the first to the second. If Article two is not navigable from Article one, the output should have the form

```
      <Article two> is not navigable from <Article one>.
```

Sample Input

```
4
Baldwin-Wallace College
3
Liberal arts college
Berea, Ohio
WBWC
Liberal arts college
2
Undergraduate
Liberal arts
Berea, Ohio
3
Cuyahoga County, Ohio
Ohio
Cleveland
Ohio
3
Columbus
Cincinnati
Cleveland
3
Baldwin-Wallace College
Cleveland
Liberal arts college
Undergraduate
Liberal arts college
Baldwin-Wallace College
```

Sample Output

```
Cleveland is 2 articles away from Baldwin-Wallace College.
Undergraduate is 1 articles away from Liberal arts college.
Baldwin-Wallace College is not navigable from Liberal arts college.
```

Floating Holidays

Many holidays fall on the same date every year – for instance, Christmas. However, this is not true of all holidays. Those that do not are called floating holidays. Even though these do not fall on the same date every year, there is a pattern to their scheduling. Seven holidays, as well as the method of determining their dates, are given in the table below.

ML King	3rd Monday of Jan
Presidents Day	3rd Monday of Feb
Memorial Day	Last Monday of May
Labor Day	1st Monday of Sep
Election Day	Tuesday after 1st Monday of Nov
Thanksgiving Day	4th Thursday of Nov

Your program is to determine the date for each of these floating holidays for a given year (whether or not it would really have been celebrated that year).

There is some information that you might find helpful. Recall the number of days in each month (January - 31, February – 28 in general but 29 in leap years, March - 31, April - 30, May - 31, June - 30, July - 31, August - 31, September - 30, October - 31, November - 30, December - 31). Years that are a multiple of four, except those that are a multiple of 100 but not of 400, are leap years. Thus, last year (2008) was a leap year, as was 2000, but neither this year (2009) nor 1900 are. In addition, January 1 was a Thursday this year.

The first line of input will be a single integer n ($0 < n <= 3000$) indicating the number of cases to follow. Each of the following n lines contains a single integer y ($0 < y <= 3000$) indicating the particular year.

For each case (year) in the input, there should be 8 corresponding lines of output – the given year, followed by the dates of the 7 floating holidays had they been celebrated that year. Dates should be expressed in *mm/dd/yyyy* format. Notice that there should always be 8 digits in the date. The cases should appear in the order of the input, and for each year (case), the holidays should be listed in the same order as they appear on the table above.

Sample Input

```
3
1410
2057
109
```

Sample Output

```
1410
01/15/1410
02/19/1410
05/28/1410
09/03/1410
11/06/1410
11/22/1410
2057
01/15/2057
02/19/2057
05/28/2057
09/3/2057
11/6/2057
11/22/2057
109
01/21/0109
02/18/0109
05/27/0109
09/02/0109
11/05/0109
11/28/0109
```

Sequences

Numeric sequences are important in many areas of mathematics. Two very important types of mathematical sequences are arithmetic sequences and geometric sequences. It is often useful to be able to quickly determine whether a sequence is arithmetic or geometric. An arithmetic sequence is one in which the difference between any two sequential numbers in the sequence is the same. A geometric sequence is one in which the quotient of any two sequential numbers in the sequence is the same. Thus, 1, 2, 3, 4, 5 … is arithmetic, since the difference between each number and the next is always one. The sequence 3, 9, 27, 81, 243 … is geometric, since the quotient of each number divided by the previous number is always three.

Your program is to determine whether five integers could constitute the beginning of an arithmetic sequence or of a geometric sequence, or if neither is possible. If it could be one of them, the program should determine the next five integers in the sequence.

The first line of input will contain a single integer, *n*, indicating the number of cases to follow. Each of the succeeding *n* lines will each contain five integers, each in the range 0 – 1000, separated by a single space, and in non-decreasing order.

Your output should have *n* lines, one line per case. If the five integers could constitute the beginning of an arithmetic or geometric sequence, the corresponding output line should be "Arithmetic", "Geometric", or "Both" (as appropriate), followed by the next five integers in the sequence, all separated by a single blank. If it is impossible for the input integers to begin either an arithmetic or geometric sequence, only the word "Neither" should be output.

Sample Input

```
3
1 2 3 4 5
3 9 27 81 243
1 2 3 4 6
```

Sample Output

```
Arithmetic 6 7 8 9 10
Geometric 729 2187 6561 19683 59049
Neither
```

Hamming Codes

When messages are transmitted, it is possible that errors can occur during transmission, resulting in the received message differing from the one sent. Hamming Codes are used to detect when a transmitted message is in error. In some situations Hamming codes will also indicate where the error in transmission occurred and it can be corrected.

Hamming Codes work by adding extra bits, called parity bits, to a binary message. A parity bit functions by forcing the sum of a group of bits to be an even number (called even parity) or an odd number (called odd parity). Here, we are interested in even parity. Thus, if the original bits sum to an even number, the extra parity bit is set to zero; if they sum to an odd number, it is set to one. Hence, in either case, the sum including the parity bit is always an even number.
For an original message that is 8 bits long, 4 parity bits are added in strategic locations to form a 12-digit message. Each of these additional bits is the parity bit for a different combination of bits in the message, as follows. (Note: for purposes of this problem, bit 1 is defined to be the *leftmost* bit in the sequence.)

Parity bit in position 1 combines with bits at positions 3, 5, 7, 9 and 11.
Parity bit in position 2 combines with bits at positions 3, 6, 7, 10 and 11.
Parity bit in position 4 combines with bits at positions 5, 6, 7 and 12.
Parity bit in position 8 combines with bits at positions 9, 10, 11 and 12.

If a received message has been transmitted without error, each of the above combinations of bits should sum to an even number. For example, consider the transmitted message ‘011100101010’.

The above four combinations lead to the following sums:
$0 + 1 + 0 + 1 + 1 + 1 = 4$ $1 + 1 + 0 + 1 + 0 + 1 = 4$ $1 + 0 + 0 + 1 + 0 = 2$ $0 + 1 + 0 + 1 + 0 = 2$
Since these sums are all even numbers, the transmitted message is deemed valid.

Had the above transmission been ‘011100101110’ instead, the sums involving parity bits at positions 2 and 8 would have been odd. It is no coincidence that 2 + 8 = 10, and that the 10th position in the message is the incorrectly transmitted bit. Flipping the 10th bit from 1 to 0 corrects the error and makes the transmitted message valid.

Note that each bit of the real message is involved in at least two of the above sums. Consequently, if only a single parity bit is odd, the message bits themselves have not been altered and therefore the message is still valid. Also note that not all combinations of odd parity bits point to valid message bits (e.g., if all four are odd there is no corresponding 15th bit). In these cases, the message is deemed invalid, but it is unable to be corrected using this method. Your program will analyze a series of transmitted 12-bit messages to determine whether or not transmission errors have occurred and if they can be can be corrected.

The first line of input is a single integer, *n*, indicating the number of lines to follow. Each of the remaining *n* lines contains a single 12-bit transmitted message. Your program is to determine if each transmitted message is valid or not, to correct any invalid messages, and to recover the original 8-bit message.

There should be one line of output for each of the transmitted messages. The line should begin with the transmitted message, followed by the word "valid" or "invalid" (whichever is appropriate). If the message is valid or can be recovered by the parity check, the original 8-bit message should appear at the end of the line. Otherwise the word "unknown" should be printed in place of the valid message. Each of these should be separated by a single blank.

Sample Input

```
2
011100101010
011100101110
```

Sample Output

```
011100101010 valid 10011010
011100101110 invalid 10011010
```