Kent Programming Contest

November 14, 2016

Rules

1. There are *nine* problems this time around; try to complete as many as you can.
2. All problems require you to read the test data from standard input and write results to standard output. You cannot use files for any purpose, including input, output or scratch. Ignore any input specifications in the problem statement.
3. When displaying results, follow the format in the Sample Output for each problem. Unless otherwise stated, all whitespace in the Sample Output consists of exactly one blank character. Watch out for extra whitespace at the end of lines and/or extra blank lines. Output will be checked mechanically, you have been warned!
4. To emphasize: your output is compared character by character with our canned output: this means that extra spaces, etc. will yield a "wrong answer" diagnostic.
5. The allowed programming languages are C, C++, Java and Python (2 or 3). In the case of Python, be sure to specify in the first line (with a comment) whether you are using python2 or python3.
6. All programs will be re-compiled prior to testing with the judges' data.
7. Non-standard libraries cannot be used in your solutions. The Standard Template Library (STL) and C++ string librarues are allowed. The standard Java API is available, except for those packages that are deemed dangerous by contest officials (i.e. that might generate a security violation).
8. The input to all problems will consist of multiple test cases; in fact, your program may be run several times, if need be.
9. Programming style is not considered here. You are free to code in whatever style you prefer; documentation is not required.
10. Submit your program using the command:

    `~mrothste/submit␣prog.ext`

    where prog is the problem name and ext is the extension used for the programming language of your choice. Instead of the problem name, you can also use a letter (a-??) cap or small.
11. In case of difficulty, please contact the instructor.

Problem A: Judges' Time Calculation

Source file: `calc.{c, cpp, java}`
Input file: `calc.in`

In our region, the contest traditionally starts at 12:30 and lasts for 5 hours. If you are able to submit a solution at 12:39, the wise judges would determine that 9 minutes had elapsed since the start of the contest. Sadly, as the day grows longer, the judges have more trouble doing the calculations accurately (how quickly can you determine the elapsed time for a 3:21 submission?)

Having struggled for many years, the judges developed the following system. Before the contest starts, they place the following table on the board at the front of the room

time	elapsed
12:XX	XX − 30
1:XX	XX + 30
2:XX	XX + 90
3:XX	XX + 150
4:XX	XX + 210
5:XX	XX + 270

When a problem is submitted with a given time-stamp, they determine which row of the table to use, based upon the hour of the time-stamp. Then, the formula in the right column is used to compute the number of elapsed minutes. For example, with a submission time of 12:39, the top row is applied with XX=39, leading to the elapsed minutes calculated as 39 - 30 = 9. For a program submitted at 3:21, the fourth row is used to calculate 21 + 150 = 171 elapsed minutes.

Your goal is to develop a program that generates the appropriate table given knowledge of the starting time and duration of a contest.

Input: The input starts with a line containing a single integer $1 \leq N \leq 30$ that is the number of cases. Following this are N lines, with each line containing integral values SH, SM, DH, DM separated by spaces. The values $1 \leq SH \leq 12$ and $0 \leq SM \leq 59$ respectively represent the hour and minute at which a contest starts. The values $0 \leq DH \leq 10$ and $0 \leq DM \leq 59$ represent the duration of the contest in terms of hours and minutes. A contest will last at least 1 minute and at most 10 hours and 59 minutes. This allows us to omit any A.M. or P.M. designations for the times.

Output: For each case, you are to produce a table formatted as shown in the Example Output. Any row in which the hour designator is a single digit (e.g., 5:XX) should have a single leading space, as should the header of the table just before the word "time".

The table must have a row for every hour block in which a program might be submitted. Assume that the earliest possible submission is precisely the contest starting time (i.e., 0 elapsed minutes), and that the latest possible submission has an elapsed time of the full duration of the contest (e.g., 5:30 in our region).

Example Input:	**Example Output:**

<pre>3
12 30 5 0
7 0 2 59
9 59 4 1</pre>
	<pre>------+---------
	 time | elapsed
	------+---------
	12:XX | XX - 30
	 1:XX | XX + 30
	 2:XX | XX + 90
	 3:XX | XX + 150
	 4:XX | XX + 210
	 5:XX | XX + 270
	------+---------
	 time | elapsed
	------+---------
	 7:XX | XX
	 8:XX | XX + 60
	 9:XX | XX + 120
	------+---------
	 time | elapsed
	------+---------
	 9:XX | XX - 59
	10:XX | XX + 1
	11:XX | XX + 61
	12:XX | XX + 121
	 1:XX | XX + 181
	 2:XX | XX + 241</pre>

Problem B: Mad Scientist

Source file: `mad.{c, cpp, java}`
Input file: `mad.in`

A mad scientist performed a series of experiments, each having n phases. During each phase, a measurement was taken, resulting in a positive integer of magnitude at most k. The scientist knew that an individual experiment was designed in a way such that its measurements were monotonically increasing, that is, each measurement would be at least as big as all that precede it. For example, here is a sequence of measurements for one such experiment with $n=13$ and $k=6$:

1, 1, 2, 2, 2, 2, 2, 4, 5, 5, 5, 5, 6

It was also the case that n was to be larger than k, and so there were typically many repeated values in the measurement sequence. Being mad, the scientist chose a somewhat unusual way to record the data. Rather than record each of n measurements, the scientist recorded a sequence P of k values defined as follows. For $1 \leq j \leq k$, $P(j)$ denoted the number of phases having a measurement of j or less. For example, the original measurements from the above experiment were recorded as the P-sequence:

2, 7, 7, 8, 12, 13

as there were two measurements less than or equal to 1, seven measurements less than or equal to 2, seven measurement less than or equal to 3, and so on.

Unfortunately, the scientist eventually went insane, leaving behind a notebook of these P-sequences for a series of experiments. Your job is to write a program that recovers the original measurements for the experiments.

Input: The input contains a series of P-sequences, one per line. Each line starts with the integer k, which is the length of the P-sequence. Following that are the k values of the P-sequence. The end of the input will be designated with a line containing the number 0. All of the original experiments were designed with $1 \leq k < n \leq 26$.

Output: For each P-sequence, you are to output one line containing the original experiment measurements separated by spaces.

Example Input:	**Example Output:**

`6 2 7 7 8 12 13`
`1 4`
`3 4 4 5`
`3 0 4 5`
`5 2 2 4 7 7`
`0`
	`1 1 2 2 2 2 2 4 5 5 5 5 6`
	`1 1 1 1`
	`1 1 1 1 3`
	`2 2 2 2 3`
	`1 1 3 3 4 4 4`

Problem C: Voting

Source file: `voting.{c, cpp, java}`
Input file: `voting.in`

A committee clerk is good at recording votes, but not so good at counting and figuring the outcome correctly. As a roll call vote proceeds, the clerk records votes as a sequence of letters, with one letter for every member of the committee:

Y means a yes vote
N means a no vote
P means present, but choosing not to vote
A indicates a member who was absent from the meeting

Your job is to take this recorded list of votes and determine the outcome.

Rules:
There must be a quorum. If at least half of the members were absent, respond "need quorum". Otherwise votes are counted. If there are more yes than no votes, respond "yes". If there are more no than yes votes, respond "no". If there are the same number of yes and no votes, respond "tie".

Input: The input contains of a series of votes, one per line, followed by a single line with the # character. Each vote consists entirely of the uppercase letters discussed above. Each vote will contain at least two letters and no more than 70 letters.

Output: For each vote, the output is one line with the correct choice "need quorum", "yes", "no" or "tie".

Example Input:	Example Output:
YNNAPYYNY YAYAYAYA PYPPNNYA YNNAA NYAAA #	yes need quorum tie no need quorum

Problem D: Queen Collisions

Source file: `collision.{c, cpp, java}`
Input file: `collision.in`

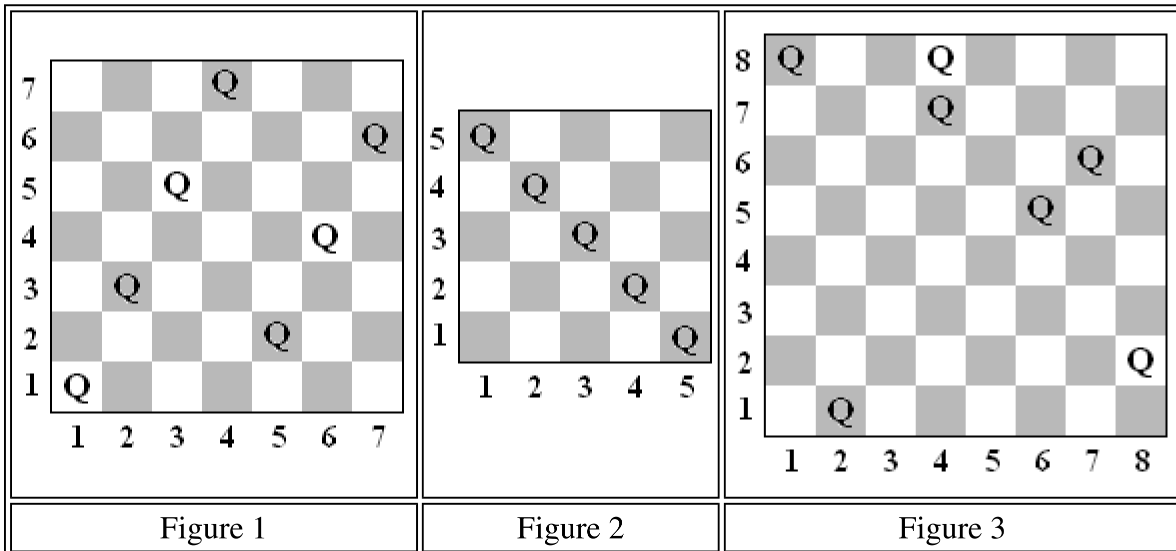


Figure 1 | Figure 2 | Figure 3

Lots of time has been spent by computer science students dealing with queens on a chess board. Two queens on a chessboard *collide* if they lie on the same row, column or diagonal, and there is no piece between them. Various sized square boards and numbers of queens are considered. For example, Figure 1, with a 7 x 7 board, contains 7 queens with no collisions. In Figure 2 there is a 5 x 5 board with 5 queens and 4 collisions. In Figure 3, a traditional 8 x 8 board, there are 7 queens and 5 collisions.

On an *n* x *n* board, queen positions are given in Cartesian coordinates (*x*, *y*) where *x* is a column number, 1 to *n*, and *y* is a row number, 1 to *n*. Queens at distinct positions (x_1, y_1) and (x_2, y_2) lie on the same diagonal if $(x_1 - x_2)$ and $(y_1 - y_2)$ have the same magnitude. They lie on the same row or column if $x_1 = x_2$ or $y_1 = y_2$, respectively. In each of these cases the queens have a collision if there is no other queen directly between them on the same diagonal, row, or column, respectively. For example, in Figure 2, the collisions are between the queens at (5, 1) and (4, 2), (4, 2) and (3, 3), (3, 3) and (2, 4), and finally (2, 4) and (1, 5). In Figure 3, the collisions are between the queens at (1, 8) and (4, 8), (4, 8) and (4, 7), (4, 7) and (6, 5), (7, 6) and (6, 5), and finally (6, 5) and (2, 1). Your task is to count queen collisions.

In many situations there are a number of queens in a regular pattern. For instance in Figure 1 there are 4 queens in a line at (1,1), (2, 3), (3, 5), and (4, 7). Each of these queens after the first at (1, 1) is one to the right and 2 up from the previous one. Three queens starting at (5, 2) follow a similar pattern. Noting these patterns can allow the positions of a large number of queens to be stated succinctly.

Input: The input will consist of one to twenty data sets, followed by a line containing only 0.

The first line of a dataset contains blank separated positive integers *n g*, where *n* indicates an *n* x *n* board size, and *g* is the number of linear patterns of queens to be described, where $n < 30000$, and $g < 250$. The next *g* lines each contain five blank separated integers, *k x y s t*, representing a linear pattern of *k* queens at locations ($x + i*s$, $y + i*t$), for $i = 0, 1, ..., k-1$. The value of *k* is positive. If *k* is 1, then the values of s and t are irrelevant, and they will be given as 0. All queen positions will be on the board. The total number of queen positions among all the linear patterns will be no more than *n*, and all these queen positions will be distinct.

Output: There is one line of output for each data set, containing only the number of collisions between the queens.

The sample input data set corresponds to the configuration in the Figures.

Take some care with your algorithm, or else your solution may take too long.

Example input:	Example output:
7 2 4 1 1 1 2 3 5 2 1 2 5 1 5 5 1 -1 1 8 3 1 2 1 0 0 3 1 8 3 -1 3 4 8 2 -3 0	0 4 5

Problem E: Mirror, Mirror on the Wall

Source file: `mirror.{c, cpp, java}`
Input file: `mirror.in`

For most fonts, the lowercase letters `b` and `d` are mirror images of each other, as are the letters `p` and `q`. Furthermore, letters `i`, `o`, `v`, `w`, and `x` are naturally mirror images of themselves. Although other symmetries exists for certain fonts, we consider only those specifically mentioned thus far for the remainder of this problem.

Because of these symmetries, it is possible to encode certain words based upon how those words would appear in the mirror. For example the word `boxwood` would appear as `boowxod`, and the word `ibid` as `bidi`. Given a particular sequence of letters, you are to determine its mirror image or to note that it is invalid.

Input: The input contains a series of letter sequences, one per line, followed by a single line with the `#` character. Each letter sequence consists entirely of lowercase letters.

Output: For each letter sequence in the input, if its mirror image is a legitimate letter sequence based upon the given symmetries, then output that mirror image. If the mirror image does not form a legitimate sequence of characters, then output the word `INVALID`.

Example Input:	Example Output:
boowxod bidi bed bbb #	boxwood ibid INVALID ddd

Problem F: Top This

Source file: `top.{c, cpp, java}`
Input file: `top.in`

The game *Top This* is played with pieces made up of four unit squares placed so that each square shares a side with at least one other square. This results in the following seven pieces:

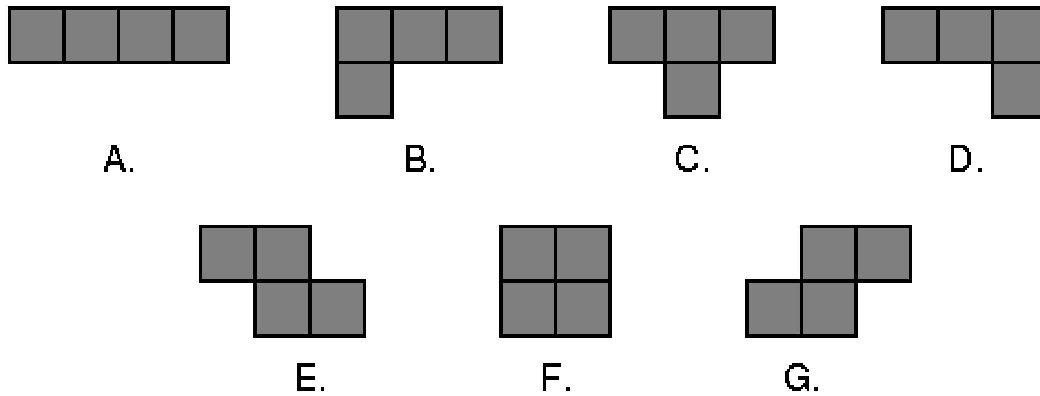


Fig. 1: The playing pieces

To play the game you are given three red pieces and three blue pieces. Your goal is to arrange the three red pieces on a grid so that they do not overlap each other (although they may have incident sides), and then to arrange the three blue pieces on top of the red pieces so that they cover precisely the same squares.

For example, suppose our red pieces include shape B and two copies of shape C, and our blue pieces were shape B, shape D, and shape E. We could create identical shapes by combining them like so:

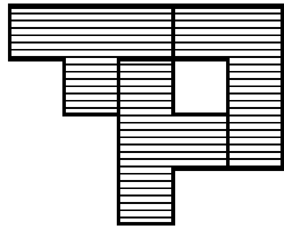


Fig. 2: Combination of B, C, and C

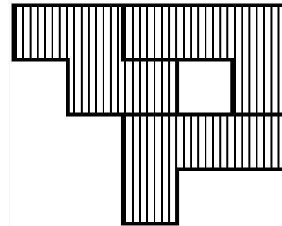


Fig. 3: Combination of B, D, and E

Notice that you may rotate each piece independently to whatever orientation you choose. However, you may *not* flip the piece over. (Doing so would change a piece of shape B to a piece of shape D, and vice versa; it would similarly swap E and G.)

Your job is to solve a *Top This* puzzle and report a shape that can be made from both groups of pieces, if possible. We further constrain the problem by only considering solutions that can be placed on a six-by-six grid. It is possible that there will be more than one solution to the puzzle—and even if there is only one solution, there will be many different positions and orientations that the shape could be in. You are to give the shape that comes first based on the following ordering:

> Given two solutions, examine the squares in row-major order (i.e., from left-to-right in each row, starting with the top row) until you find the first square that is different between the two. The solution in which the square is filled comes *before* the solution in which the square is empty.

Input: The first line of input contains a number T, where $1 \leq T \leq 5$. The remaining T lines represent T data sets, one per line. Each data set consists of a string of three characters, a single space, and another string of three characters. Each character will be an uppercase letter from A to G representing one of the seven pieces as labeled in *Figure 1* above. The first three characters will be the red pieces; the last three characters will be the blue pieces.

Output: For each data set you should first output the ordinal of the data set on its own line: 1 for the first set, 2 for the second, and so on. After the ordinal should be the solution, printed as six lines with six characters per line. Each character represents a unit square—a hash mark (`#`) indicates a filled square, while a period (`.`) indicates an empty square. The filled squares should describe the first shape that can be formed by both the red pieces and the blue pieces, as per the ordering defined above. If there is no possible solution, simply print "No solution" on a single line.

Example input:	**Example output:**
4 BCC BDE BCC EFG AAA CCE ADG DEE	1 #####. .##.#. ..###. ..#... .... 2 No solution 3 ####.. .####. ..#### .... 4 ####.. .####. ...### .....# ....

Problem G: Quick Search

Source file: `search.{c, cpp, java}`
Input file: `search.in`

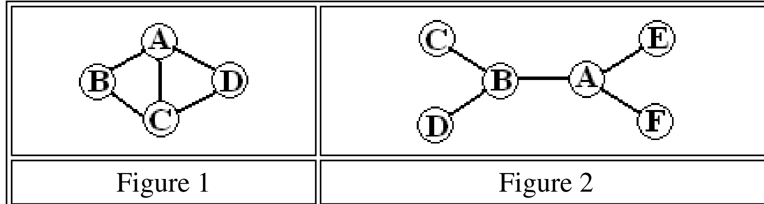


Figure 1 | Figure 2

A police unit has dealt with a number of situations where they need to search multiple sites as rapidly as possible with a small group of officers. Figures 1 and 2 are diagrams for two situations. Sites to search are labeled with capital letters. The site labeled A is the common starting point. Some of these locations have connecting paths. Assume each connecting path takes one unit of time to traverse, and individual sites take no time for a visual search.

Consider Figure 1. If there are 3 or more officers it takes only one unit of time to search. Different officers follow paths AB, AC, and AD. If there are two officers, the complete search will take two time units. For example one officer could follow the path ABC and the other AD.

Consider Figure 2. If there were 3 officers, they could follow paths ABC, ABD, and AEAF and take 3 units of time. With 2 officers they could follow paths ABCBD and AEAF, and take 4 units of time.

These are simple enough examples to figure out by hand. For more complicated arrangements they want your programming help.

Input: The input will consist of 1 to 25 data sets, followed by a line containing only 0.

A dataset is a single line starting with blank separated positive integers *s n p*, where *s* is the number of sites to search, *n* is the number of officers, and *p* is the number of connecting paths between search sites. Limits are $2 \le s \le 10$, $1 \le n \le 4$, and $p \le 20$. The rest of the line contains *p* pairs of capital letters, indicating paths between sites, each preceded by a blank. Sites are labeled with the first *s* capital letters. All sites will be connected by some sequence of connecting paths. The two letters in each letter pair will be in increasing alphabetical order. No letter pair will be repeated.

Output: There is one line of output for each data set, containing only the minimum search time for *n* officers starting at site A.

Take some care with your algorithm, or else your solution may take too long.

The sample input data sets correspond to the scenarios discussed.

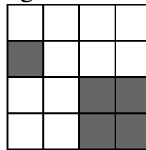
Example input:	**Example output:**

`4 4 5 AB BC CD AD AC`
`4 2 5 AB BC CD AD AC`
`6 3 5 AB BC BD AE AF`
`6 2 5 AB BC BD AE AF`
`0`
	`1`
	`2`
	`3`
	`4`

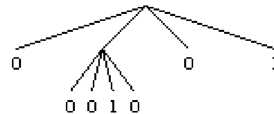
Problem H: Image Compression

Source file: `compress.{c, cpp, java}`
Input file: `compress.in`

Strategies for compressing two-dimensional images are often based on finding regions with high similarity. In this problem, we explore a particular approach based on a hierarchical decomposition of the image. For simplicity, we consider only bitmapped images such as the following:

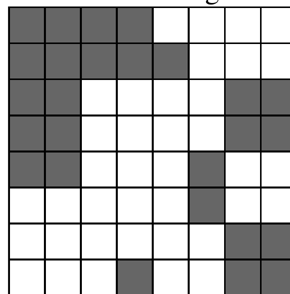


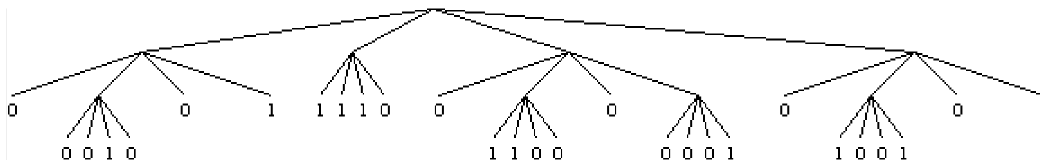
The image is encoded as a tree, with the root representing the entire image region. If a region is monochromatic, then the node for that region is a leaf storing the color of the region. Otherwise, the region is divided into four parts about its center, and the approach is applied recursively to each quadrant. For a non-leaf node, its four children represent the four quadrants ordered as upper-right, upper-left, lower-left, lower-right respectively. As an example, here is the tree encoding of the above image.



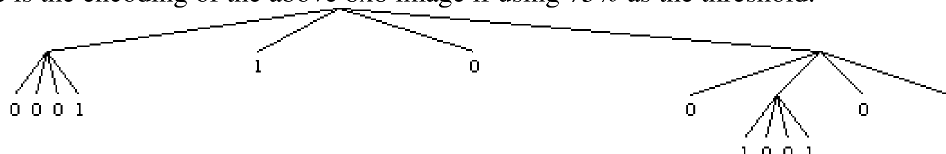
The original image is not monochromatic, so we considered the four quadrants. The top-right quadrant is monochromatic white, so the first child of the root node is a leaf with value 0. The top-left quadrant is not monochromatic, so it is further divided into four subquadrants, each of which is trivially monochromatic. This results in the subtree with leaf values 0, 0, 1, 0. The final two quadrants are monochromatic with respective values 0 and 1.

As a larger example, here is an 8x8 image and the tree encoding of it.

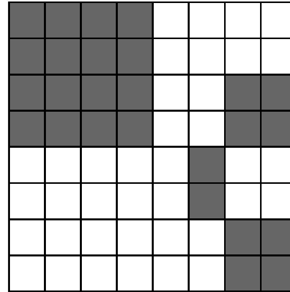




Thus far we have described a *lossless* compression scheme, but the approach can be used for *lossy* compression with the following adjustment. Instead of continuing the decomposition until reaching a monochromatic region, a threshold such as 75% is used, and a leaf is created whenever a region has at least that percentage of either color. As an example, here is the encoding of the above 8x8 image if using 75% as the threshold.



Notice that 75% of the top-left quadrant of the full image is black, and therefore the second child of the root is 1, and that more than 75% of the bottom-left quadrant of the full image is white, and therefore the third child of the root is 0. However, neither white nor black reaches 75% in the top-right quadrant, so the recursive decomposition continues, but all four of those subquadrants achieve the 75% threshold and become leaves. If we were to uncompress the image based on this new lossy encoding, we get back the following result.



Your goal is to determine the image that results from this lossy compression scheme, given an original bitmap image and a specific threshold percentage.

Input: The input will consist of a series of data sets, followed by a line containing only 0. Each data set begins with a line containing values W and T, where W is the width of the bitmap and T is the threshold percentage. Images will always be square with $1 \leq W \leq 64$ being a power of two. Threshold T will be an integer with $51 \leq T \leq 100$. Following the specification of W and T are W additional lines, each of which is a string of width W containing only characters 0 and 1, representing a row of the image bitmap, from top to bottom.

Output: For each data set, you should print an initial line of the form "`Image 1:`" numbering the images starting with 1. Following that should be `W` lines, with each line representing a row of the resulting bitmap as a string of characters 0 and 1, from top to bottom.

<table>
<tr><th>Example Input:</th><th>Example Output:</th></tr>
<tr><td>
<pre>
4 80
0000
1000
0011
0011
8 75
11111000
11110000
11000011
11000011
11000100
00000100
00010011
00010011
4 75
1101
1111
0111
0011
0
</pre>
</td><td>
<pre>
Image 1:
0000
1000
0011
0011
Image 2:
11110000
11110000
11110011
11110011
00000100
00000100
00000011
00000011
Image 3:
1111
1111
1111
1111
</pre>
</td></tr>
</table>

Problem I: Egyptian Fractions

Source file: `egypt.{c, cpp, java}`
Input file: `egypt.in`

During the Middle Kingdom of Egypt (circa 2000 BC), the Egyptians developed a novel way of writing fractions. Adding a certain symbol to the hieroglyph for an integer was understood to represent the reciprocal of that integer. This made it easy to write any fraction of the form $\frac{1}{N}$ (called a *unit fraction*)—simply add the reciprocal symbol to the hieroglyph for N.

There was no direct way to represent a fraction of the form $\frac{M}{N}$, where M > 1. Instead, any such fraction was written as the sum of unit fractions. For example, the fraction $\frac{3}{4}$ could be written as:

$$\frac{3}{4} = \frac{1}{2} + \frac{1}{4}$$

Notice that multiple sums may yield the same result; for example, $\frac{3}{4}$ can also be written as

$$\frac{3}{4} = \frac{1}{3} + \frac{1}{4} + \frac{1}{6}$$

Your job is to take a fraction $\frac{M}{N}$ and write it as a sum of unit fractions by way of a “greedy” search. In a greedy search you continually subtract the largest unit fraction possible until you are left with zero. For example, consider the fraction $\frac{9}{20}$. A greedy search would find the sum $\frac{1}{3} + \frac{1}{9} + \frac{1}{180}$ in three steps, like so:

$$(1.)\ \frac{9}{20} - \frac{1}{3} = \frac{7}{60}, \quad (2.)\ \frac{7}{60} - \frac{1}{9} = \frac{1}{180}, \quad (3.)\ \frac{1}{180} - \frac{1}{180} = 0$$

Note that at each step we subtracted the largest possible unit fraction from whatever remained of our original fraction.

One additional restriction must be added to keep our solutions from becoming too large: Each time we subtract a unit fraction, we must be left with a fraction whose denominator is strictly less than 1,000,000. For example, consider the fraction $\frac{17}{69}$. The first two unit fractions we would subtract would be $\frac{1}{5}$ and $\frac{1}{22}$, leaving us with $\frac{7}{7590}$. At this point, the largest unit fraction we could subtract would be $\frac{1}{1085}$, leaving us with

$$\frac{7}{7590} - \frac{1}{1085} = \frac{1519}{1647030} - \frac{1518}{1647030} = \frac{1}{1647030}$$

Unfortunately, this leaves us with a fraction whose denominator is larger than 1,000,000, so we can *not* use this unit fraction in our sum. We move on to the next largest unit fraction, $\frac{1}{1086}$, which leaves us with

$$\frac{7}{7590} - \frac{1}{1086} = \frac{1267}{1373790} - \frac{1265}{1373790} = \frac{2}{1373790} = \frac{1}{686895}$$

Since the final answer reduces to a fraction with a denominator less than 1,000,000, we *can* use this unit fraction, leaving us with a final answer of $\frac{1}{5} + \frac{1}{22} + \frac{1}{1086} + \frac{1}{686895}$.

In this case we only had to skip over a single fraction. In general, however, you may have to skip over many fractions in order to find one that will work. While you are searching, you will have to perform calculations on many fractions with large denominators; make sure you do so efficiently, or your program may take too long to execute.

You should also make sure you are using a data type large enough to hold the large numbers you are working with. Even though we have restricted denominators to 1,000,000, you may have to calculate intermediate values with denominators up to 10^{12}. A 64-bit integer will be required to hold such values (`long` in Java, `long long` in C/C++).

Finally, it might be worth noting that, by its very nature, the greedy algorithm will always find some answer consisting of fractions with denominators less than 1,000,000 since, at the very least, any fraction can be represented as a sum of unit fractions with its own denominator. For example:

$$\frac{3}{999983} = \frac{1}{999983} + \frac{1}{999983} + \frac{1}{999983}$$

Input: The input will consist of a sequence of fractions; one per line. Each line will contain only two integers *M* and *N*, where $1 < M < N < 100$, representing the fraction $\frac{M}{N}$. M and N will have no common divisors greater than 1. The end of the input will be indicated by two zeros: 0 0.

Output: For each input fraction $\frac{M}{N}$ you are to output a single line containing numbers $D_1 \le D_2 \le D_3 \le \ldots$ such that:

$$\frac{M}{N} = \frac{1}{D_1} + \frac{1}{D_2} + \frac{1}{D_3} + \ldots$$

This should be the first sum arrived at using a greedy search while enforcing the denominator bound of 1,000,000. Each number should be separated by a single space, with no leading or trailing whitespace.

Example input:	Example output:
3 4 2 7 9 20 17 69 0 0	2 4 4 28 3 9 180 5 22 1086 686895