ACM Programming Contest
Dixie State College
March 3, 2012

Do NOT turn this page until

10:00 am Mountain Standard Time!

Rules

1. Each team consists of up to three students.
2. Each team may use one computer.
3. Teams may use any printed references.
4. Teams may not use any electronic aids, including the internet. The help system built in to your programming environment is okay.
5. High school teams and college teams compete and are judged separately (unless otherwise requested).
6. All solution must be written in C++ or Java. Python is also permitted for high school teams, and for college teams where no member has completed CS 3005 or any other course that requires C++.
7. The team with the most correct solutions wins.
8. Ties will be broken using a time score:
   (a) Each time a team submits a correct solution, the number of minutes that have elapsed since the beginning of the competition is added to the time score.
   (b) For each incorrect solution submitted, a 20 minute penalty is added to the time score, but only if the team eventually submits a correct solution to that problem.
   (c) Multiple penalties will be added for multiple incorrect solutions to the same problem.
9. The input for each problem comes from standard in. This goes by the names `cin` and `STDIN` in C++, `System.in` in Java, and `sys.stdin` in Python. Other methods may also exist for receiving input from standard in for each language.
10. The output for each problem should be sent to standard out. This goes by the names `cout` and `STDOUT` in C++, `System.out` in Java, and `sys.stdout` in Python. Other methods may also exist for sending output to standard out for each language.
11. The output of submitted solutions must exactly match the output of the reference solution, down the the last character. Whitespace differences matter. Any other output, including debugging output, may cause an otherwise correct solution to be marked as incorrect. Each problem statement with example input and output shows exactly where newline characters are placed and where spaces are appropriate.
12. Solutions have a 10 second time limit. Any solution that runs longer than that will be considered incorrect.
13. If you need to handle last-minute registration details, please arrive at 9:00 am. Otherwise, please arrive at 9:30 am to find a computer, complete the sample problem, and sign in to the judging system.
14. The contest begins at 10:00 am and ends at 3:00 pm.

Contents

1 Gravitational Constants

You are a free-lance cargo smuggler with your Starship. Your ship is fast, but it's also a piece of junk. In order to safely leave a planet, you must enter into your ship's computer the gravitational constant for that planet. For example, on Earth, you would enter 9.8 (meters per second per second).

You devise a clever way to determine this constant without having to pay for any more high-tech equipment. You simply pay a local to jump off the top of your starship (which is exactly 13 meters tall), and precisely time how long it takes the poor fellow to splat on the ground. Given that time in seconds, and the height of 13 meters, you can calculate the gravitational constant of any planet, as long as the locals are smart enough to follow directions, yet not savvy enough to carefully think through the consequences.

The input consists of a series of times—1 per line, given in seconds. Your output should consist of a series of gravitational constants—1 per line, given in meters per second per second. Write the gravitational constants with at most 2 digits of precision. To do so, multiply your answer by 100, add .5, cast from a float to an integer, and divide by 100.0

Note: The ¶ symbol in the examples below represents a newline character.

Sample Input

```
1.62882203586¶
2.77915¶
0.8122¶
```

Sample Output

```
9.8¶
3.37¶
39.41¶
```

2 Compression

You are stuck in another galaxy, cut off from home and supplies. You have managed to devise a way to contact home and send a message, but only have microseconds of transmission time to deal with.

Therefore, you have to compress your messages to the smallest possible stream. Given that nobody remember to bring a copy of 7z along for the trip you need to implement the algorithm yourself. You have decided on a simple Huffman encoding implementation.

Huffman encoding is a way of compressing data by choosing a new, variable length symbol to represent every character of the sample set. None of the new symbols is a prefix of another to ensure that the process can be properly reversed.

The first step in Huffman encoding is to perform a frequency analysis and then construct the symbol table out of it, after which the actual encoding can be performed trivially.

The technique works by starting with one node for each symbol appearing in the input. Take the number of times that symbol appeared in the input and call it the frequency of the node. The next step is to create a binary tree of nodes by taking the two least frequent nodes and adding them as children to a new node, which has the frequency of them both combined. The new node is added to the list, replacing the original nodes that are now its children. This process is repeated until only one node remains in the list. For the sake of consistency, the "left" node will always be the one with the smaller frequency or the one with the lower ASCII value if they are equal. If the node does not have a character value (a generated node) takes the ASCII value of if its leftmost decedent. Ignore newlines (who needs them?).

Once this tree has been created, the symbol for any given symbol can be created by going down the tree to its node, for every left child append a "0" to the symbol and for every right child append a "1".

The input will consist of some body of text, potentially very large, think infinite monkeys and all that. The output should consist of the string of zeros and ones that would be the binary encoding for each character in the input ordered by the ASCII value of the character.

Note: The ¶ symbol in the examples below represents a newline character.

Sample Input

```
bob¶
```

Sample Output

```
b 1¶
o 0¶
```

Sample Input

```
How now brown cow?¶
```

Sample Output

```
  110¶
? 11110¶
H 11111¶
b 0000¶
c 0001¶
n 001¶
o 01¶
r 1110¶
w 10¶
```

3 Code Breaking

In this problem you will need to decode a message intercepted from the Nazis.

Here is what your intellegence has provided to you to aide in the decryption:

1. The Nazis are using a simple transposition cipher (where the nth character is replaced with the (n+`offset`)th character.
2. The offset is not known. It is consistent within a line of text. It does however change between lines of text.
3. If the (n+`offset`)th character is out of the range of the alphabet, it wraps around to the beginning of the alphabet.
4. The alphabet (in order) of the cipher is "`ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789,.?!`" (Note the space between "Z" and "0")
5. Any character not in the above alphabet is not replaced.
6. Each line will have at least one occurance of the word "`NAZI`".

The input consists of several lines of text that have been encrypted.

The output must consists of those lines of text but decoded.

Note: The ¶ symbol in the examples below represents a newline character.

Sample Input

```
!0L8M012¶
QD1L2456¶
5673RE2M3(RE2M3GER3FI3ER1 LIVI)¶
K.WFPXmessage:XthisXdoesXnotXgetXencoded¶
```

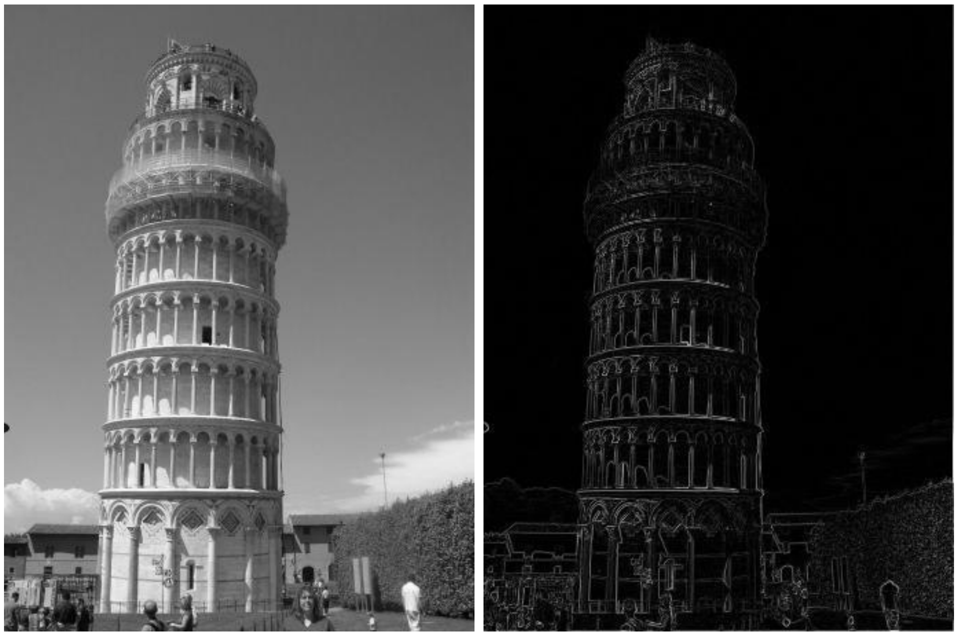
Sample Output

```
NAZI ABC¶
NAZI 123¶
123 NAZI (NAZI CAN BE ANYWHERE)¶
NAZIS message: this does not get encoded¶
```

4 Edge Detection

Computer vision relies on algorithms to automatically detect features in digital images. Many features are represented by lines. Programs that use computer vision, must have algorithms to automatically detect the lines in an image.

One mechanism for line detection relies on processing images with filters that transform the image to have bright pixels near line features and dark pixels in the other areas of the image.



An example of this is shown in the two images displayed. The first image is a gray-scale image, and the second is the image that results when the line detection filter is run on the first image.

The image filter used detects lines by looking for light intensity changes in the horizontal and vertical directions, then combining the results.

Horizontal edge detection

To detect horizontal edges we apply a simple formula to create the output pixel from the input pixel:

$$\text{horizontal\_output}(x, y) = |\text{input}(x+0, y+0) - \text{input}(x+0, y+1)|$$

For the bottom row of the image we treat the non-existent row as all 0 pixels.

Vertical edge detection

To detect vertical edges we apply a simple formula to create the output pixel from the input pixel:

$$\text{vertical\_output}(x, y) = |\text{input}(x+0, y+0) - \text{input}(x+1, y+0)|$$

For the right-most column of the image we treat the nonexistent column as all 0 pixels.

Combined edges

To show the combined edges we use the following formula:

$$\text{lines\_output}(x, y) = \sqrt{\text{horizontal\_output}(x, y)^2 + \text{vertical\_output}(x, y)^2}$$

Truncate any results after the decimal in the result of the square root operation. If the result is greater than 255, assign 255 for the value.

Input and Output

The input will describe a single image. The first line will contain two numbers: the `width` and the `height` of the image. $0 <$ `width`, `height` ≤ 1000. There will be `height` additional lines, each with `width` positive integers. Each positive integer in the image data is followed by exactly one space character.

The output file will store the results of the `line_output` above in the same format as the input. As in the input, each number in the image data must be followed by exactly one space, including the last number in each row.

Note: The ¶ symbol in the examples below represents a newline character.

Sample Input

```
6 6¶
240 240 8 8 240 240 ¶
239 239 8 8 239 238 ¶
240 8 239 7 7 8 ¶
240 7 240 7 240 240 ¶
240 240 239 7 240 239 ¶
240 7 8 240 240 240 ¶
```

Sample Output

```
6 6¶
1 232 0 232 1 240 ¶
1 255 231 231 232 255 ¶
232 231 232 0 233 232 ¶
233 255 233 233 0 240 ¶
0 233 255 255 1 239 ¶
255 7 232 240 240 255 ¶
```

5 Making Change

You work as a cashier at a fast food restaurant stationed near an intergalactic wormhole. As part of your job, you must give customers change when they pay for their purchases. This is complicated because you must deal with thousands of different kinds of money systems to accomodate your varied clientele.

You only deal in coins, and all amounts you deal with (including coid values) are whole numbers. When a customer hands you money you calculate how much change is required, then you open a drawer full of coins in that customer's money system. You may assume that you always have plenty of coins. You have a list giving the value of each coin, which is always a whole number. For example, with American money you would have this list of coins:

$$(1, 5, 10, 25, 50, 100)$$

If you had a client from England ("You're from Earth? Do you happen to know this guy named John that I met once?"), you would find this list of coins:

$$(1, 2, 5, 10, 20, 50, 100, 200)$$

Some coin systems are harder to remember and work with (like the Eulerians whose coins are all valued with prime numbers), so you must write a program to help you make change. You write it to guarantee that you give each customer the minimum possible number of coins, while also ensuring that you give him/her (it?) exactly the right amount. You may assume that each currency has a coin with value 1, so it is always possible to give change.

Your input will consist of a series of problems, each on a single line. The line is of the following format:

n v_1 v_2 v_3 $\ldots$ v_k

where n is the amount of change you must give, v_1 is the value of the least valuable coin (always 1), v_2 is the next least valuable coin, and so on until v_k, which is the most valuable coin. If there are multiple optimal solutions (using the same number of coins), you should use the solution that uses the fewest v_1 coins. If those match, use the one that uses the fewest v_2 coins, etc.

You may assume that the amount of change you need to give will never exceed 1000, and no single coin is worth more than 1000.

For each line of input, you should output a line of the format:

n=v_1*c_1+v_2*c_2+$\ldots$+v_k*c_k

Where c_1 is the number of v_1 coins in the solution, etc.

Note: The ¶ symbol in the examples below represents a newline character.

Sample Input

```
57 1 5 10 25¶
633 1 75 83¶
```

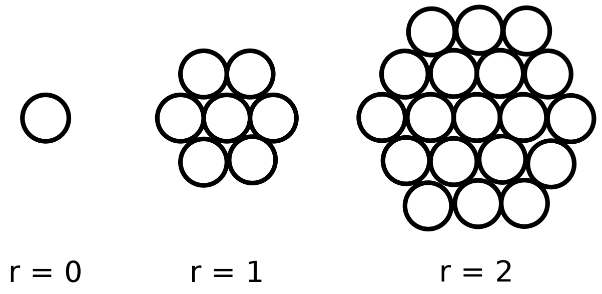
Sample Output

```
57=1*2+5*1+10*0+25*2¶
633=1*1+75*4+83*4¶
```

6 Hex Magnets

Building various shapes from spherically-shaped magnets has become a common hobby and pastime. One possible shape is a regular two-dimensional hexagon.

Given n spherically shaped magnets, what is the radius of the largest regular hexagon that you can construct? The radius is the number of magnets you must hop from the center to an outer corner.



The input consists of any number of cases. Each case is on a single line. The line contains a single number $1 \leq n \leq 150,000,000$. The input is terminated by a line with $n = 0$.

The output consists of a line for each input case. Each line contains two numbers. The first number is n from the input, and the second is r, the maximum radius obtainable. No output line is generated for the terminating case of $n = 0$.

Note: The ¶ symbol in the examples below represents a newline character.

Sample Input

```
8¶
11¶
24¶
41¶
48¶
74¶
80¶
83¶
87¶
96¶
0¶
```

Sample Output

```
8 1¶
11 1¶
24 2¶
41 3¶
48 3¶
74 4¶
80 4¶
83 4¶
87 4¶
96 5¶
```

7 Font Metrics

Proportional-width fonts look nice on the screen and on paper, but they are more work for programmers. When laying out text, one must figure out the width of each character rather than just counting the number of characters. In addition, some pairs of letters can be spaced closer together because of complementary shapes, a process called *kerning*. For example, when "V" and "A" are next to each other, they look better pushed together a little closer than two "A"s would.

Such details are called the *metrics* of a font. The Adobe Font Metric (AFM) file format is a relatively easy to parse file format for font metrics. One section in an AFM file starts with the line:

`StartCharMetrics` n

where n is an integer, followed by details for n characters, one per line, and closing with:

```
EndCharMetrics
```

Each of the n lines stores a series of fields delimited by ";" characters. For example, the line for a semicolon (";") in one font looks like:

```
C 59 ; WX 277 ; N semicolon ; B 86 -193 195 431 ;
```

The parts of this that we care about are detailed here:

- `C` n: the character code (ASCII code for our purposes) of the character (n is an integer)
- `WX` n: the normal width of the character (n is an integer)
- `N` s: the postscript name of the character (s is a word consisting only of letters (in either case).

These fields can occur in any order, and additional fields that do not concern us may be present as well. You may assume that there is a single space between each number or token.

Another significant section of the AFM file starts with the line:

`StartKernPairs` n

Followed by details for n kerning pairs and ending with:

```
EndKernPairs
```

A kerning pair line looks like:

```
KPX A V -111
```

The token "KPX" is always present. The next two fields are the postscript names of two letters, and the forth field is the adjustment that should be made to the width of the first character when it is followed by the second. In this example, an "A" followed by "V" is 111 units narrower than the combined width of "A" and "V" when not next to each other.

The units of measurement here are 1/1000ths of the font's size. So in a 12 point font, this example would nudge "A" and "V" closer together by $12 \times (111 \div 1000)$ points.

You will be given these two sections from an AFM file as input, followed by a series of lines of text. You must compute the width of each line when correctly rendered in this font, and output the width, one width per line. You may assume that every character in the line of text has a corresponding character metrics entry, though not every pair of characters has kerning data associated with it.

Note that a space in the input is a character line any other. The newline character at the end of each line just marks the end of the line; its width should not be computed.

Note: The ¶ symbol in the examples below represents a newline character.

Sample Input

```
StartCharMetrics 6¶
C 44 ; WX 277 ; N comma ; B 86 -193 203 106 ;¶
C 89 ; WX 750 ; N Y ; B 11 0 738 683 ;¶
C 101 ; WX 444 ; N e ; B 28 -11 415 448 ;¶
C 104 ; WX 555 ; N h ; B 32 0 535 694 ;¶
C 115 ; WX 394 ; N s ; B 33 -11 360 448 ;¶
C 121 ; WX 527 ; N y ; B 19 -205 508 431 ;¶
EndCharMetrics¶
StartKernPairs 2¶
KPX Y e -83¶
KPX y comma -83¶
EndKernPairs¶
Yes¶
hey,¶
```

Sample Output

```
1505¶
1720¶
```

8 ASCII Art

For this problem you are given a list of 2d coordinates that will allow you to draw some ASCII art. Your task is to draw the art according to the "map" that is given you. Note that the "map" file only indicates the row and column where a character needs to appear. If a character is not given for a particular row and column, you should put a space. Also, you must realize that ASCII art can consist of characters such as "/", "\", and others that you may need to handle carefully.

Your program should read a map file from standard input and display the correct ASCII art to standard output.

The first line of the input contains the number of rows and columns that the ASCII image should contain, separated by a comma. The remaining lines each give the row, column, and ASCII character for a single position on the image. For example, the line:

```
0,3,_
```

indicates that a "_" character should be placed at row 0, column 3.

Note: The ¶ symbol in the examples below represents a newline character.

Sample Input

```
5,21¶
0,3,_¶
0,10,_¶
0,11,_¶
0,12,_¶
1,2,/¶
1,3,_¶
1,4,\¶
1,9,/¶
1,11,_¶
1,12,_¶
1,13,\¶
1,15,/¶
1,16,\¶
1,17,/¶
1,18,\¶
2,1,/¶
2,2,/¶
2,3,_¶
2,4,\¶
2,5,\¶
2,8,/¶
```

```
2,10,/¶
2,14,/¶
2,19,\¶
3,0,/¶
3,3,_¶
3,6,\¶
3,7,/¶
3,9,/¶
3,10,_¶
3,11,_¶
3,12,_¶
3,13,/¶
3,15,/¶
3,16,\¶
3,17,/¶
3,18,\¶
3,20,\¶
4,0,\¶
4,1,_¶
4,2,/¶
4,4,\¶
4,5,_¶
4,6,/¶
4,7,\¶
4,8,_¶
4,9,_¶
4,10,_¶
4,11,_¶
4,12,/¶
4,13,\¶
4,14,/¶
4,19,\¶
4,20,/¶
```

Sample Output

```
   _      ___        ¶
  /_\    / __\ /\/\  ¶
 //_\\  / /   /    \ ¶
/  _  \/ /___/ /\/\ \¶
\_/ \_/\____/\/    \/¶
```

Sample Output Clarification

This is what the output would look like if you replaced each space in the output with a “&” symbol:

```
&&&_&&&&&&___&&&&&&&&¶
&&/_\&&&&/&__\&/\/\&&¶
&//_\\&&/&/&&&/&&&&\&¶
/&&_&&\/&/___/&/\/\&\¶
\_/&\_/\____/\/&&&&\/¶
```

9 Cipher Block Chaining

A block cipher method called chaining can be used to make a much more secure ciphertext message. In this problem you will use the cipher block chaining method to encrypt a message.

Here are the steps you should follow:

1. Convert the ASCII key to a binary (`10101010`) representation.
2. Convert the ASCII text to a binary representation.
3. Break the binary text from number 2 up into some fixed-sized blocks (you will use 12 bits as your block size, but you theoretically could use anything). If the size of the last block is less than 12, you should add alternating 1's and 0's to it until you have 12 bits. If you added some padding, you should append a final block (also of size 12) that contains the number of bits you added, i.e., if I added 4 bits of padding, I would append the block "`000000000100`" as the final block to be encoded.
4. Each of the above 12-bit blocks will now be encrypted, following this process:
   (a) Take the first block to be encrypted, reverse it, and Xor it with the first 12 bits of the key. Store this as the first block of encrypted output.
   (b) Take the next block and Xor it with the encrypted output of the first block. Reverse the new block and Xor it with the first 12 bits of the key. Add this to the encrypted output stream.
   (c) Repeat until all of the blocks are encrypted.

Your code should read input with the ASCII key on the first line and the text to be encoded on the remaining lines. It should convert them into their binary representation and use the cipher block chaining method as described above to "encrypt" them. The resulting bitstream should then be output as a series of 0's and 1's.

Note: The ¶ symbol in the examples below represents a newline character.

Sample Input

```
ABC¶
ABCDE¶
```

Sample Output

```
011010010110111010100110000001100001100100101110001001011101¶
```

Additional Hints

“ABCDE” in binary is:

```
0100000101000010010000110100010001000101
```

After padding is added (to get last block up to 12 bits), the result is:

```
010000010100001001000011010001000100010110101010
```

After the 12-bit block is appended telling how much padding was added, the result is:

```
010000010100001001000011010001000100010110101010000000001000
```

The final output is produced as follows:

Key in binary (first 12 bits)	010000010100	
First 12-bit block (reversed)	001010000010	
Xor	011010010110	(first 12 bits of encrypted output)

2nd 12-bit block	001001000011	
Output of last Xor	011010010110	
Xor	010011010101	(input to next round, call it α)

Key in binary (first 12 bits)	010000010100	
Xor’ed output (α reversed)	101010110010	
Xor	111010100110	(next 12 bits of encrypted output)

3nd 12-bit block	010001000100	
Output of last Xor	111010100110	
Xor	101011100010	(input to next round, call it β)

And so on…