University of Arkansas
Department of Computer Science and Computer Engineering
2007 High School Programming Contest
Problems

Revision Date: March 2, 2007

Please carefully read the format specifications for output. Your program will be graded using a differential program (i.e. the output produced by your program will be compared byte-for-byte with the output of the judge's solution. Thus, even though your program may produce output that is *technically* correct, it will be counted as incorrect if these specifications are not followed exactly.

Unless otherwise directed, a line of output should have no leading or trailing whitespace. For example, suppose you are to produce the integer 27 on one line of output. That line should consist of the character '2' followed by the character '7' followed by the newline ('\n') character.

Note that for all problems, an integer is defined as:

an optional minus sign,

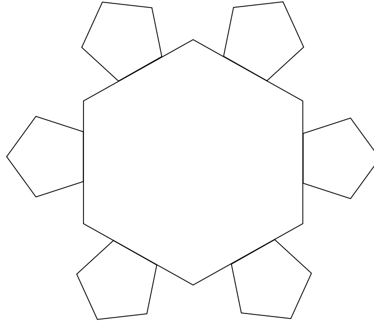
followed by one of the digits from the set 0..9,

followed by any number of digits from the set 0..9

Thus `3`, `-21`, and `309` are integers; `- 4` and `+38` are not.

1 Fractals: Silly Snowflake

Consider building a snowflake-like structure from regular polygons. Suppose we start by making the core of the snowflake a hexagon. Suppose we then glue a smaller pentagon to the center of each edge of the hexagon. If we do so, we should see a structure like this:



Note that by adding the pentagons, the snowflake is now made up of seven polygons (one hexagon plus six pentagons). Now suppose further that on each available edge of each pentagon (there are four available edges per pentagon), we glue a square. Then on each available edge of each square (there are three per square), we glue an isosceles triangle. How many polygons now make up the snowflake?

Your task is to calculate the total number of polygons that make up a snowflake whose core is a regular polygon of n sides. As above, each of the n sides has a regular polygon of n - 1 sides attached, and so on. As above, the last round attaches isosceles triangles. Your program only needs to deal with n ¿= 3. You also do not need to worry about integers overflowing (the number of polygons adds up fast!).

input file input; the name of the file is passed as a command line argument; the file contains a single integer, specifying the number of sides of the core polygon of the snowflake;

output the total number of polygons in the fully grown snowflake is written to the console (stdout)

2 Assembly Language: Generation

Your task is to write a simple expression compiler. You will read in a postfix expression and produce an equivalent assembly language program. The assembly language is composed of the following commands:

store x y	store value x in memory location y
add x y z	add the value in memory location x to the value in memory location y and place the result in memory location z
sub x y z	subtract the value in memory location y from the value in memory location x and place the result in memory location z
mul x y z	multiply the value in memory location x with the value in memory location y and place the result in memory location z
div x y z	divide the value in memory location x by the value in memory location y and place the result in memory location z — truncate any non-integer result to the next smallest integer

For example, the expression...

```
4 3 +
```

...should produce the following assembly language program:

```
store 4 0
store 3 1
add 0 1 0
```

As another example, the expression...

```
4 3 + 2 -3 * -
```

...should produce the program:

```
store 4 0
store 3 1
add 0 1 0
store 2 1
store -3 2
mul 1 2 1
sub 0 1 0
```

To be a correct program, it must use the fewest number of memory locations possible. Also, when storing to a memory location, the program must use the lowest memory location available (starting with location zero). Also, operations must be performed in the order given. In the previous example, the addition must be performed before the multiplication and the multiplication before the subtraction. The final result must be stored in memory location zero.

Note that once a value is retrieved by an operation, that memory location is immediately available for reuse. That is why the *add* operation above...

```
add 0 1 0
```

...can retrieve values from memory locations 0 and 1 (and, by the process, free them up) and store the result in memory location 0.

input file input; the name of the file is passed as a command line argument; the file contains a free-format postfix expression

output the equivalent assembly language program is written to the console (stdout), one assembly language statement per line

3 Assembly Language: Virtual Machine

Your task is to implement a virtual machine for the assembly language described in the previous problem. Your virtual machine does not need to perform any error checking (it will only be tested with well-formed assembly language programs), although you may wish to have error checks to help in debugging your virtual machine.

For example, given the assembly language program…

```
store 2 0
store 3 1
store 4 2
store 5 3
mul 2 3 2
mul 1 2 1
mul 0 1 0
```

…your program should output the value 120. Note that the given assembly program does not have to follow the convention that you must store to the lowest available memory location.

input file input; the name of the file is passed as a command line argument; the file contains a free format assembly language program

output the resulting integer value is reported to the console (stdout)

4 Audio: playing records backwards

Formerly, in the time of vinyl records and earlier, songs could be played backwards by manually spinning the disc backwards on a turntable (or by reloading a tape on a reel in reverse for play on a reel-to-reel tape player). Rock and Roll has a long history of incorporating reversed audio into songs; the technique is denoted *backwards masking*. Perhaps the most famous historical example of suspected backwards masking can be found on the Beatles' *Magical Mystery Tour* album. Playing 'I am the Walrus' in reverse supposedly reveals the phrase:

Paul is dead, Paul is dead...

This is one of the many clues conspiracy theorists claim were given by the surviving Beatles that Paul McCartney had died and had been replaced by a look-a-like.

You are to write a program for reversing an audio stream. This is a conceptually simple task, but requires a little thinking on how to accomplish this when there are multiple channels (e.g. stereo). To relieve you of some of the drudgery of dealing with compressed music formats, you will manipulate simplified human-readable versions of WAVE files, or HR-WAVE. An HR-WAVE file has the following format (all the entries are integers):

```
SampleRate
NumberOfChannels
BitsPerSample
Amplitude_1_Channel_1
Amplitude_1_Channel_2
...
Amplitude_1_Channel_M
Amplitude_2_Channel_1
Amplitude_2_Channel_2
...
Amplitude_2_Channel_M
Amplitude_3_Channel_1
Amplitude_3_Channel_2
...
Amplitude_3_Channel_M
...
...
...
Amplitude_N_Channel_0
Amplitude_N_Channel_1
...
Amplitude_N_Channel_M
```

The amplitude data is often denoted *samples*.

For CD quality, a sample rate of 44100Hz with 16 bits per sample is sufficient. That means it takes 44100 samples of audio data to make up one second of audio. The number of channels specifies how many different audio streams are combined to make the whole. For example, monaural audio (M=1) has a single channel. Stereo audio (M=2) has two channels (left and right), while quadraphonic audio (M=4) has four channels,

and so on. The amplitude data for multiple channels are interleaved. For stereo, the amplitude data/samples looks like this:

```
Amplitude_1_Channel_1
Amplitude_1_Channel_2
Amplitude_2_Channel_1
Amplitude_2_Channel_2
...
Amplitude_N_Channel_1
Amplitude_N_Channel_2
```

Your task is to read in an HR-WAVE file and write out a reversed version of the song in HR-WAVE format. For example, if the input file was...

```
44100
1
16
-16
22
68
146
21349
```

...you would write out (to console) the following data:

```
44100
1
16
21349
146
68
22
-16
```

Remember, you must not change the order of the channels. That is, the data in channel 1 should remain in channel 1, and so on.

To help you hear if you did things correctly, you may use the command-line audio player, hrw-play. Here are some example uses:

```
hrw-play song1.hrw
java Prob3 song3.hrw > temp; hrw-play temp
Prob3 song3.hrw > temp; hrw-play temp
```

The first command simply plays an HR-WAVE file. The next one saves the output of a java implementation of this problem into a file named *temp*. The *temp* file is then played. The last command is similar, but for a C or C++ implementation.

input file input; the name of the file is passed as a command line argument; the file contains free format HR-WAVE data

output the resulting reversed HR-WAVE data is reported to the console (stdout), one integer per line; no leading or trailing spaces; no blank lines

5 Audio: Changing pitch and tempo

Also in the days of vinyl records, turntables had three different speed settings, $33\frac{1}{3}$, 45, and 78. These numbers referred to the number of revolutions per minute (RPM) the record would make. Albums were played at $33\frac{1}{3}$, singles at 45, and Grandma and Grandpa's old records were played at 78 RPM.

It was always great fun (remember, it was a much simpler time) to play records at the wrong speed. Playing a single at $33\frac{1}{3}$ RPM lowered the pitch and slowed down the tempo. Playing an album at 78 RPM greatly increased the pitch and tempo of the music.

Your task is to simulate increasing the speed of playback in software. Suppose you are asked to double the speed of playback. This is a relatively simple task; simply discard the first, the third, the fifth (and so on) samples in each channel. For example, if the original HR-WAVE file contained...

```
44100
1
16
-16
22
68
146
21349
20071
```

...the resulting HR-WAVE file would be:

```
44100
1
16
22
146
20071
```

If you were asked to triple the speed of playback, you would discard two out of every three samples in each channel.

Things get trickier when 'playing' at a faster speed that is not an integer multiple. The correct way to do this is with a technique called resampling, but here is a simpler and almost as effective approach.

Suppose you wish to speed up the playback by a factor of 1.5. Since 1.5 is equivalent to the fraction $\frac{3}{2}$, this means, for every group of three samples, you would keep the last two out of the three. In general, for speeding up with a factor of $\frac{x}{y}$, you would keep the last y samples of each group of x. Of course, this process proceeds for the next group of x and so on.

Your task is to write a program for speeding up the playback speed of an HR-WAVE file by producing new HR-WAVE data that has fewer samples than the original data, respectively. Here are some example calls to your program (assuming Java implementations):

```
java Prob4 test.hrw 2
java Prob4 test.hrw 1.5
java Prob4 test.hrw 1.125
java Prob4 test.hrw 1
```

The first call doubles the playback speed, the second increases the speed by a factor of $\frac{3}{2}$, the third by $\frac{9}{8}$, and the fourth by a factor of 1.

Note: to do this task properly, you will need to convert the speed factor (a real number) to its rational number equivalent. The rational number equivalent should be in its simplest form (i.e. the numerator and denominator will have no common factors other than 1). Use this algorithm for converting the real number to a numerator and denominator:

```
numerator = (int) (factor * 1000);
denominator = 1000;
common = greatestCommonDivisor(numerator,denominator);
numerator = numerator / common;
denominator = denominator / common;
```

You may assume that the factor is a positive number.

Also, don't forget to take into account multiple channels!

input file input; the name of the file is passed as the first command line argument; the file contains free format HR-WAVE data; the second command line argument is the speed-up factor; the factor will be greater than or equal to 1; the factor will never have more than three digits after the decimal point

output the resulting modified HR-WAVE data is reported to the console (stdout); one integer per line; no leading or trailing spaces; no blank lines;

6 Audio: mixing with normalization

Your task is to combine or *mix* a number of audio streams into a single audio stream. Mixing is a relatively easy process. Suppose you are mixing three streams whose first samples are -2, 22, and 317. In the output, the first sample would then be the sum of those first samples, or 337. The second sample in the output is the sum of the second samples of the inputs, and so on. The only complication to mixing is that sometimes the sum of a set of input samples gets too large.

Suppose the HR-WAVE file has a specification of b bits per sample (the number of bits per sample is the third integer in the file). This means the largest sample in the amplitude data should not be larger than $2^b - 1$ while the most negative sample should not be less than $-2^b - 1$. If any of the samples in the output exceed these limits, the entire output must be *normalized.*

Normalization is the process of scaling data so that it fits precisely in the required range. If the output data needs to be normalized, use the following algorithm to scale the amplitude data.

Let U be the maximum allowed postive value, P be the most postive and N most negative sample. Let $R = max(P, -N)$. If R is greater than U, then each sample should be multiplied by U and then divided (integer division) by R. While it is possible to normalize the data if R is less than U (in essence, turning up the volume), do not do so for this problem.

You may assume that all the input HR-WAVEs will have the same sample rate, number of channels, and bits per sample. They may not be all the same length, in which case, you should treat the shorter ones as contributing zeros to the sum.

input file input; the names of the HR-WAVE files to be mixed are passed as the command line arguments

output the resulting HR-WAVE data is written to the console (stdout); one integer per line; no leading or trailing spaces; no blank lines; the data should be scaled appropriately, if necessary

7 Cellular automata: FireStarter

A cellular automaton is a simple computational device for simulating the evolution of a discrete system. Perhaps the most famous cellular automaton is the Game of Life, invented by John Conway. In a cellular automaton, each cell or location in the system changes its state based upon the state of its nearest neighbors in the previous time step.

You are to implement a cellular automaton called *FireStarter*. FireStarter models a flammable terrain populated with ignition points; the surrounding terrain is modeled by '.'s and the ignition points by 'x's.

The terrain is specified by a grid of ASCII characters. Here is an example of a grid:

```
.....
..x..
.....
```

A cell starts burning (modeled by a '.' becoming a '0') if its heat index exceeds a certain threshold. The heat index is calculated by summing the values of the neighborhood. For example, the heat index of the middle cell in the following grid is 5...

```
.01
..2
1.1
```

...while the heat index of the upper-left cell is 0 and the heat index of the bottom middle cell is 4. Note that '.' neighbors are treated as zeros when calculating a heat index.

Rules:

- an 'x' always becomes a '0' in the next timestep
- a cell numbered '0' through '8' always increments in the next timestep
- a '.' becomes a '0' in the next timestep if its heat index in the current timestep is greater than 10
- a '.' becomes a '0' in the next timestep if it is on the edge and has an integer-valued neighbor in the current timestep

input file input; the name of the file is passed as a command line argument; the file contains the number of time steps to simulate followed by an ASCII grid of '.'s and 'x's; the grid starts on the second line; each row of the grid starts at the beginning of a line; each row of the grid has the same length

output the state of the grid after each time step is reported to stdout (the console); the original grid is the first grid printed; an asterisk ('*') separates successive grids; each asterisk and each row of a grid is on a line by itself

8 Cellular automata: Pyro

Pyro is another cellular automaton related to FireStarter. The difference is you will be modeling the ignition of a trail of gunpowder. The gunpowder ignites more readily than the surrounding terrain. Another difference is that the '0' number is reserved to model the flashing of the gunpowder. Moreover, while the numbers '1' through '9' and the uppercase letters 'A' through 'I' represent increasingly hotter temperatures, the uppercase letters 'J' through 'Z' represent increasingly cooler temperatures. Thus, values 'G' and 'J' are the same temperatures, as are '3' and 'X', '2' and 'Y', and '1' and 'Z'. You will need to take this into account in calculating the heat index of a cell.

As with FireStarter, the surrounding terrain is modeled by '.'s and the ignition points by 'x's. In addition, gunpowder is modeled by 's's.

Rules:

- an 'x' always becomes a '0' in the next timestep but only if it is adjacent to an 's' in the current timestep. Otherwise, it is treated the same as a '.'
- an 's' next to any number in the current timestep becomes a '0' in the next timestep
- a cell numbered '0' through '8' always increments in the next timestep
- a cell numbered '9' becomes the letter 'A' in the next timestep
- a cell lettered 'A' through 'Y' becomes the next higher letter in the next timestep
- a '.' becomes a '0' in the next timestep if its heat index in the current timestep is greater than 15
- a '.' becomes a '0' in the next timestep if it is on the edge and has an integer-valued neighbor in the current timestep

input file input; the name of the file is passed as a command line argument; the file contains the number of time steps to simulate followed by an ASCII grid of '.'s and 'x's and 's's; the grid starts on the second line; each row of the grid starts at the beginning of a line; each row of the grid has the same length

output the state of the grid after each time step is reported to stdout (the console); the original grid is the first grid printed; an asterisk ('*') separates successive grids; each asterisk and each row of a grid is on a line by itself