C1b3rCh4ll3ng3.17 **Admission test - Feb 3, 2017**
Programming **Duration: 180 minutes**
.....

Challenge #1 — hidden in plain sight

Steganography is the science of writing hidden messages in such a way that no one apart from the sender and the intended recipient even realizes there is a hidden message. Media files are ideal carriers for steganographic transmissions due to their size, which is typically much larger than the messages they hide. A common approach is to imperceptibly alter the pixels (individual picture elements) of an image, encoding the secret text by adding small variations in their color. In practice, these small differences pass completely unnoticed to the human eye.

A secret message is hidden in image file hidden/sheldon.pgm:

Your goal is to write a program that extracts and prints it!

Input image format. The format we consider is a subset of ASCII PGM (Portable Gray Map), a simple scheme for encoding grayscale images by describing each pixel with a number in [0, MAX], where 0=black and MAX=white. The PGM documents we consider here are text files structured as follows, with each line less than 128 chars and MAX=255:

```
line 1: <ignored>
line 2: <ignored>
line 3: width w and height h of the image in pixels, with w>0 and h>0
line 4: <ignored>
line 5 to 4+w*h: unsigned 8-bit grayscale values encountered by
                 scanning the image by row
```

Example (3x2 image):

```
P2
# CREATOR: GIMP PNM Filter Version 1.1
3 2             <- w h
255
112             <- pixel 1
123             <- pixel 2
106             <- pixel 3
117             <- pixel 4
250             <- pixel 5
110             <- pixel 6
```

112	123	106
117	250	110

Encoding. Let s be the secret message and let k be its length. The text s is represented as a zero-terminated sequence of k+1 bytes containing 7-bit ASCII codes and is encoded in the least significant bit (bit 0) of the first 7*(k+1) pixels encountered by scanning the image by row. That is:

```
- bit 0 of pixel 1 contains bit 0 of s[0]
- bit 0 of pixel 2 contains bit 1 of s[0]
  ...             ...             ...
- bit 0 of pixel 7 contains bit 6 of s[0]
- bit 0 of pixel 8 contains bit 0 of s[1]
- bit 0 of pixel 9 contains bit 1 of s[1]
  ...             ...             ...
```

Bit 7 of each byte of s is always zero. You can assume that 7*(k+1) <= w*h, i.e., you'll always find the zero terminator by the end of the file.

Input (stdin): a PGM image

Output (stdout): secret message encoded in the input image

Write the program in the directory hidden/ in any of the following languages: C (89 or 99), C++ (98), Java (7), Python (2.7 or 3.4), Scala (2.11).

Create a bash script hidden/decode.sh that compiles/runs the program on stdin and writes the result to stdout.

Examples:

```
--------------------------- decode.sh for C -----------------------------
#!/bin/bash
gcc –O1 decode.c –o decode
./decode < sheldon.pgm
-------------------------------------------------------------------------
```

```
------------- decode.sh for Python (python=2.7, python3=3.4) -----------
#!/bin/bash
python3 decode.py < sheldon.pgm
-------------------------------------------------------------------------
```

Your solution will be evaluated for correctness, performance, and elegance, on the same platform you're using to develop it.

Challenge #2 – the maze

Write a C function (C89 or C99) that finds a path in a maze, not necessarily the shortest one, from one corner to another. Here's the prototype:

```
/* --------------------------- maze.h ------------------------------ */
   unsigned solve_maze(const char* maze, unsigned w, unsigned h,
                       char* path);
/* ----------------------------------------------------------------- */
```

Parameters:
- maze: pointer to the maze, given as an array of chars of size w*h, where cell (x,y) of the maze has index x+y*w. Cell (0,0) is the upper-left corner of the maze. Any char different than ' ' represents a wall that cannot be crossed by a path;
- w: width of the maze (num horizontal cells), >= 2;
- h: height of the maze (num vertical cells), >= 2;
- path: pointer to an output buffer of w*h bytes representing a path, if any, from cell (1,0) to cell (w-2,h-1), given as a sequence of moves encoded as: 0 = down, 1 = right, 2 = up, 3 = left. The path should never go through the same cell twice.

Returns: number of bytes written in path, or zero if no path was found

Example:

```
   (0,0) (1,0)
      | /
char* maze = "+ +---+"    w = 7, h = 4                          +0----+
             "| |   |"    char* path = { 0,0,1,1,2,1,1,0,0 }    |0|110|
             "|   | |"    returned length = 9 moves             |112|0|
             "+---- +";                                         +----X+
                   |
             (w-2,h-1)
```

Write your solution in maze/maze.c and test it with the given main. The solution will be evaluated for correctness, performance, and elegance, on the same platform you're using to develop it.

Challenge #3 – Wimpy's diet

Wimpy is always hungry, but his doctor put him on a diet: he's only allowed to eat sandwiches in a strictly decreasing order of weight at any given meal. His favorite restaurant, however, only serves sandwiches in a fixed order, so Wimpy has to decide which ones to pick. You are to help him: write a program that removes sandwiches from a menu so that the remaining sandwiches have the maximum possible total weight and are served in a strictly decreasing order of weight.

Input (stdin):
number of sandwiches on the menu followed by their weights, one per line

Output (stdout):
number of sandwiches in your solution followed by their weights, one per line, in the same order as they appear in the input

Example 1:

stdin ->	stdout
8	6
389	389
~~207~~	300
~~155~~	299
300	170
299	158
170	65
158	
65	

Example 2:

stdin ->	stdout
4	1
~~16~~	224
~~93~~	
~~107~~	
224	

Constraints:
- there are never more than 10000 sandwiches on a menu
- sandwich weights are integers in [1,10000]
- no two sandwiches have the same weight

Write the program in the directory wimpy/ in any of the following languages: C (89 or 99), C++ (98), Java (7), Python (2.7 or 3.4), Scala (2.11).

Create a bash script wimpy/wimpy.sh that compiles/runs the program on stdin and writes the result to stdout.

Examples:

```
-------------------------- wimpy.sh for C -----------------------------
#!/bin/bash
gcc –O1 wimpy.c –o wimpy
./wimpy
-----------------------------------------------------------------------
```

```
------------ wimpy.sh for Python (python=2.7, python3=3.4) ----------
#!/bin/bash
python3 wimpy.py
-----------------------------------------------------------------------
```

Your solution will be evaluated for correctness, performance, and elegance, on the same platform you're using to develop it.