8. MipMapping a Texture

Program Name: Mipmapping.java   Input File: mipmapping.dat

In computer graphics images are stored as textures, which are two dimensional arrays of texel data. A texel is just an individual pixel of a texture. Each texture will have its own height and width, and can have different formats (how the texel data is actually stored). There are many different texture formats which all have a different data size to visual fidelity ratio in order to serve different purposes, such as computer games or web page images. We will be focusing on standard, uncompressed data for simplicity. The format will be described below.

Mipmaps are smaller versions of a texture that are used by graphics hardware to speed up rendering at a distance. As the texture moves farther away the hardware chooses a smaller mip level, if available, in order to save room in the texture cache and decrease texture bandwidth usage. Since the image is taking up a smaller amount of screen space, the smaller texture fits better with less scaling and one cannot notice the decrease in quality. These mipmaps are stored along with the main texture in a special texture format. The hardware requires that each of the textures' dimensions be of powers of 2 if they are to have mipmaps, and each mipmap has dimensions of powers of 2 down from the previous. The mipmap chain stops when one dimension hits 1. For example, a 16 by 16 texture will have an 8 by 8 mipmap, a 4 by 4 mipmap, a 2 by 2 mipmap, and a 1 by 1 mipmap. A texture whose width or height is not a power of 2 cannot be mipmapped.

These textures are stored as 32 bits (4 bytes) per pixel, or texel. Each set of 4 bytes in a texel represents a color channel. In order the texels are *Alph*a (how transparent the texel is, with 0 being transparent and 255 being fully opaque), *Red* (0 being no red contribution, 255 being full red), *Green* and *Blue*. Opaque purple would be represented as `0xffff00ff` and transparent white would be `0x00ffffff` (all colors set to full creates white on computers).

Your boss at work needs you to write a program that can generate mipmaps for a 2D texture based on your new knowledge. For each texel of the mipmap you are generating, you should take a 2 by 2 block of texels from the mipmap above it, average each one of the color channels individually, and then combine those bytes. For example, the two colors `0x88222222` and `0x22444444` would make `0x55333333`. You shouldn't worry about manually rounding but instead let the Java integer and bit shifting math do it for you by dropping bits.

Input

The first line of input will contain a single integer `n` that indicates the number of data sets to follow. For each data set:

- the first line will contain two integers `h` and `w` that indicate the height and width of a texture
- the next `h` lines will contain the texture
- each texture line contains `w` integers separated by a single space
    - each integer will be a hexadecimal 32 bit integer with 32 bit precision
    - each integer will be in the format `0xaarrggbb`, where `a` is Alpha, `r` is Red, `g` is Green and `b` is Blue.

Output

For each data set containing a texture with dimensions that are powers of 2, you will print all mipmaps, with one row of texels per line. The texels should be printed out separated by a space in full 32 bit precision and using lowercase alpha characters, just like the input. There should not be any spaces between the mipmaps but there should be a blank line after each texture, as shown below. For a data set containing a texture that cannot be mipmapped print `NO REDUCTION POSSIBLE` followed by a blank line.

8. MipMapping a Texture (cont.)

Example Input File

```
2
4 4
0xffffffff 0xffffffff 0x00000000 0x00000000
0xffffffff 0xffffffff 0x00000000 0x00000000
0x00000000 0x00000000 0xffffffff 0xffffffff
0x00000000 0x00000000 0xffffffff 0xffffffff
8 8
0x7fcccccc 0x7fcccccc 0x7fcccccc 0x7fcccccc 0x7fcccccc 0x7fcccccc 0x7fcccccc 0x7fcccccc
0x7fcccccc 0xffff0000 0xffff0000 0xffff0000 0xffff0000 0xffff0000 0xffff0000 0x7fcccccc
0x7fcccccc 0xffff0000 0x00000000 0xffffff00 0xffffff00 0x00000000 0xffff0000 0x7fcccccc
0x7fcccccc 0xffffffff 0x00000000 0xff0000ff 0xff0000ff 0x00000000 0xffffffff 0x7fcccccc
0x7fcccccc 0xffffffff 0x00000000 0xff0000ff 0xff0000ff 0x00000000 0xffffffff 0x7fcccccc
0x7fcccccc 0xff00ff00 0x00000000 0xff00ffff 0xff00ffff 0x00000000 0xff00ff00 0x7fcccccc
0x7fcccccc 0xff00ff00 0xff00ff00 0xff00ff00 0xff00ff00 0xff00ff00 0xff00ff00 0x7fcccccc
0x7fcccccc 0x7fcccccc 0x7fcccccc 0x7fcccccc 0x7fcccccc 0x7fcccccc 0x7fcccccc 0x7fcccccc
```

Example Output to Screen

```
0xffffffff 0x00000000
0x00000000 0xffffffff
0x7f7f7f7f

0x9fd89999 0xbfe56666 0xbfe56666 0x9fd89999
0xbfe5a5a5 0x7f3f3f3f 0x7f3f3f3f 0xbfe5a5a5
0xbfa5e5a5 0x7f003f7f 0x7f003f7f 0xbfa5e5a5
0x9f99d899 0xbf66e566 0xbf66e566 0x9f99d899
0xa7b87878 0xa7b87878
0xa769b888 0xa769b888
0xa7909880
```