Department of Computer Science

COS284 Practical and Assignment 5: Floating About

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

1 Introduction

This document contains both practical 5 and assignment 5. In general the assignments will build upon the work of the current practical.

1.1 Submission

The practical will be fitchfork only and is due at 17:00 on Friday 16 October, where as the assignment will also be fitchfork only and due at 17:00 on Friday 23 October. You may use the Discord server to ask for assistance with the assignment and practical components.

1.2 Plagiarism policy

It is in your own interest that you, at all times, act responsible and ethically. As with any work done for the purpose of your university degree, remember that the University of Pretoria will not tolerate plagiarism. Do not copy a friend's work or allow a friend to copy yours. Doing so constitutes plagiarism, and apart from not gaining the experience intended, you may face disciplinary action as a result.

For more on the University of Pretoria's plagiarism policy, you may visit the following webpage: `http://www.library.up.ac.za/plagiarism/index.htm`

1.3 Practical component [28%]

There are three tasks you must complete for this practical. All tasks will be marked by fitchfork.

1.3.1 Task 1: Simple Interest Rate Calculator [5%]

In this task you will need to implement a function interestRateCalc that takes in 2 parameters of type double, principle (P) and amount accured (A) and 1 parameter of type int, time (t). Your implementation must compute the interest rate according to the following equation:

$$r = (1/t) * ((A/P) - 1) \tag{1}$$

Your function must order the parameters in the following order:

```
double interestRateCalc(double P, double A, int t)
```

When you are finished, create a tarball containing your source code file named **interestRateCalc.asm** and upload it to the assignments.cs.up.ac.za website, under the **Practical 5 Task 1** submission slot.

1.3.2 Task 2: BMI Calculator [10%]

In this task you will need to implement a function called bmiRiskCalculator in assembly. The function will accept 2 arguments, both of type double. Your implementation should compute the BMI based on the weigth and height given using the following equation:

$$BMI = weigth/(height)^2 \tag{2}$$

- For this task, you may use the pow function from C in your ASM code.

You must then return an array of type double. The first element must be the calculated BMI (from the given equation) and the second element must be 1 of 4 values indicating the implicated risk based on the following table (you must return the third column values based on BMI calculated):

BMI Range	Implicated Risk	Value
18.5 (inclusive) to 25 (not inclusive)	Normal Risk	0.1
25 (inclusive) to 30 (not inclusive)	Average Risk	0.2
30 (inclusive) and 40 (not inclusive)	Moderate Risk	0.3
40 and above	Sever Risk	0.4

- For this task, you may also use malloc function from C in your ASM code.

When you are finished, create a tarball containing your source code file named **bmiRiskCalculator.asm** and upload it to the assignments.cs.up.ac.za website, under the **Practical 5 Task 3** submission slot.

1.3.3 Task 3: Geometric Series [13%]

For task 3 you must create a function, isGeometric, in assembly. The function must accept a variable amount of arguments (at least 1). The first argument count will be the number of arguments the function is passed bar one. The rest of the parameters, if any, will be double values. For example you may be given the following C code,

```
extern int isGeometric( int64_t count, ... );
int result = isGeometric(3, 2.0, 1.5, 0.3);
```

The first parameter, 3, is entered because the function is being passed 3 double values, 2.0, 1.5 and 0.3. It is important to note that any number of arguments may be passed into the function. You may even be passed more than 8 arguments. When this happens the parameters will not be able to fit in all the registers and will be placed on the stack. Your assembly code must handle this case.

The function should take all these arguments and determine if they form a geometric series. (`https://en.wikipedia.org/wiki/Geometric_series`)

If the sequence given to the function is geometric you must return 1 else return 0. Below is some psuedo-code for the function,

```
ratio = arg[1] / arg[0]
for i in 0 .. size - 1:
  if arg[i + 1] / arg[i] != ratio:
    return 0
return 1
```

Your code must be able to handle the case of the number of arguments being less then 2. (where arg[1]/arg[0] becomes problematic) and should return 0.

When you are finished, create a tarball containing your source code file named **isGeometric.asm** and upload it to the assignments.cs.up.ac.za website, under the **Practical 5 Task 3** submission slot.

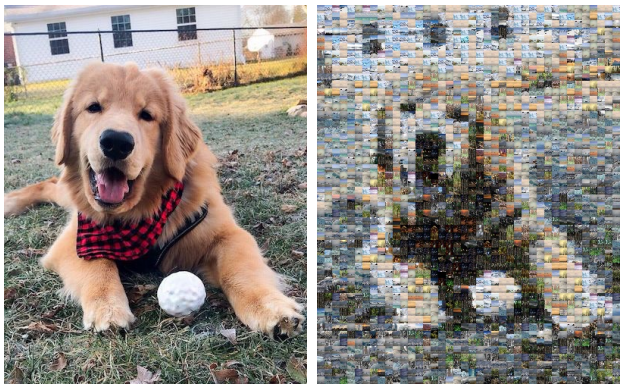
1.4 Assignment component [72%]

For the assignment you will be making a mosaic image. A mosaic image, for our purpose, is defined as an image that is constructed using only other images. An example is given below with the image on the left the original and the image on the right the constructed mosaic image.

1.4.1 Background

In this assignment you will not be required to work directly with any image formats. You will however be working with bitmaps. The images you will be working with will always be represented as 1D arrays of `unsigned chars`. The size of the arrays will always be threefold the images width multiplied by its height. The format of the image arrays is given as:

`RGBRGBRGBRGB...RGBRGBRGBRGBRGB`



"R", "G" and "B" each represent a single unsigned byte in the image. A sequence of "RGB" represents a single pixel in the image. Note that the total number of pixels in the image is equal to the width multiplied by the height of the image. The pixels are stored in row-major order.

Some tips:

- A helper function is provided for you to write your pixel data to a PPM image. You may use this to help you debug your program.
- You can view PPM files online at `http://paulcuth.me.uk/netpbm-viewer/` if you don't have an image viewer that supports them.
- You are also given an optional script, "image_converter.py", which you can use to convert images to and from PPM. The script also supports displaying the image. (Some OSes may not support image viewing in the script).
- You can use the command-line tool "xxd"/"hexdump" to view the raw binary/hex values of a file.
- See `https://en.wikipedia.org/wiki/Row-_and_column-major_order` for an explanation of row-major order.
- Every task includes a helper.asm file which you can use to include any support functions you wish to use across multiple other asm files.

1.4.2 Task 1 [7%]

In this task you will not be working directly with any image data. However, you will be working with a function that is vital to the rest of the algorithm. Specifically, you will be implementing a function that finds the distance between two colours.

You are required to provide the working assembly code for the C prototype function given below. You must implement your code in the distance.asm file.

```
float color_distance(unsigned a, unsigned b)
```

Parameters `a` and `b` each represent a single pixel. The bits 0-7 represent the red channel, 8-15 represent the green channel and 16-23 represent the blue channel. Each color channel is represented as an 8-bit `unsigned char` value.

Below we will give the formula which you must implement. But first we provide some definitions. We will let R_1, G_1 and B_1 represent the red, green and blue channels for `a` as `float`'s. We will also let R_2, G_2 and B_2 represent the red, green and blue channels for `b` as `float`'s. The average red value will be given as $\bar{r} = \frac{R_1+R_2}{2}$. Below we define the formula `color_distance` should use to calculate the return value:

$$\sqrt{(2+\frac{\bar{r}}{256})(R_1-R_2)^2+4(G_1-G_2)^2+(2+\frac{255-\bar{r}}{256})(B_1-B_2)^2}$$

Note: It is important that you follow the equation as closely as possible and avoid any mathematical simplifications as this can change the result when working with floating-point numbers.

When you are finished with the task you should upload a tarball containing your **distance.asm** and **helper.asm** files to Assignment 5 task 1 slot.

1.4.3 Task 2 [15%]

For task 2 you will be required to resize an image that is passed to you and return the newly allocated and resized image.

You are required to provide the working assembly code for the C prototype function given below. You must implement your code in the resize.asm file.

```
typedef unsigned char uchar;
uchar *resize_image(uchar *image, int w, int h, int size);
```

`image` is the bitmap of the image file which you must resize. `w` and `h` represents the width and height of the orignal image which your must resize respectively. `size` is the new width **and** height for the new pixel data image.

Below we define how one must resize the given image:

Using the dimensions of our original `image`, $\mathcal{O}$, and the new image size we can produce the floating point ratios we require to perform our resizing as $\bar{w} = \texttt{w}/\texttt{size}$ and $\bar{h} = \texttt{h}/\texttt{size}$. Finally we define the new resized image, $\mathcal{N}$, as:

$$\mathcal{N}_{r,c} = \mathcal{O}_{int(\bar{h}*r),int(\bar{w}*c)}$$

where the notation $\mathcal{I}_{i,j}$ means the pixel at row i and coloumn j for image $\mathcal{I}$. Moreover, c and r are defined for all integer values in the range $[0, \texttt{size})$. And, the function $int(x)$ returns the integer part of a floating-point number x.

Note: The return value of the function is a 1D array and the bitmap array given to you is also a 1D array. However, the algorithm description above is described using 2D notation. Hence, it is required of you to make the necessary conversions to make the algorithm work with a 1D array. As a hint, you can use the width of an image to convert 2D coordinates to-and-from 1D indices.

When you are finished with the task you should upload a tarball containing your **resize.asm** and **helper.asm** files to Assignment 5 task 2 slot.

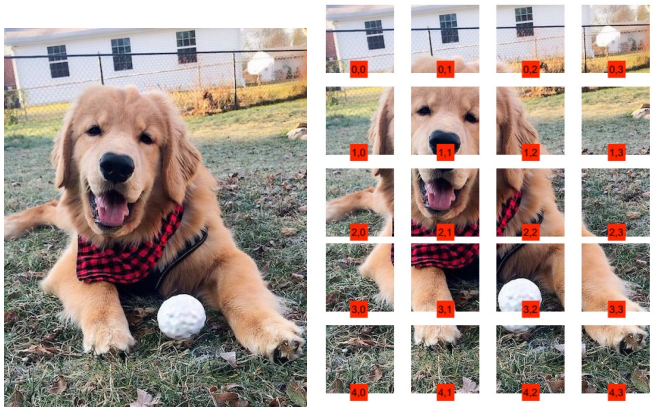
1.4.4 Task 3 [50%]

The final task will involve constructing the actual mosaic image file. The C function prototype which you will be required to provide an implementation for is given as:

```
typedef unsigned char uchar;
void draw_mosaic(uchar *image, int w, int h,
                 uchar **tiles, int tile_count, int tile_size)
```

`image` is the bitmap array of the image file which you will make the mosaic for. `w` and `h` represents the width and height of `image`. `tiles` is an array pointing to all possible tile images you can use to make the mosaic. Each tile will be generated using the `resize_image` function from task 2. `tile_count` represents the length of the `tiles` array. `tile_size` is the width and height of each tile in the `tiles` array.

Image segment row and column numbers corrospond to the coordinates of the image segments that tiles will be drawn onto. Coordinates are defined as 2D points, (r, c), in integer space. The first component, r, refers to the row number and is defined for all integers from 0 to `h/tile_size`. The second component, c, is the column number and is defined for all integers from 0 to `w/tile_size`. We use the notation $\mathcal{I}^{r,c}$ to refer to the image segments at (r, c) for an image $\mathcal{I}$. Below we give a visual example of how segments are numbered using a 500x600 image with a `tile_size` of 100.



To generate the mosaic you must find a 'good' tile image (from `tiles` array) for each image segment in the original `image`, $\mathcal{O}$, and draw it onto a new image, $\mathcal{M}$, which has the same dimensions as $\mathcal{O}$. To find a 'good' tile image for an image segment $\mathcal{O}^{r,c}$ we must compare the segment to each tile image `tiles[k]` for all `k` in $[0, \texttt{tile\_count})$. Below we define the best tile image as the image with the smallest δ which is defined as:

$$\delta = \sum_{i=0}^{\texttt{tile\_size}} \sum_{j=0}^{\texttt{tile\_size}} \texttt{color\_distance}(\mathcal{O}^{r,c}_{i,j}, \texttt{tiles[k]}_{i,j})$$

Unfortunately just picking the best image doesn't provide a pleasing 'mosaic' image. So, instead of using the best tile image, we will choose to use a 'good' tile image. We will define a 'good' tile image as a tile image, `tiles[k]`, which produces the smallest δ **and** satisfies the condition `hash_function(`r`, `c`, k) < 0.2`. `hash_function` is a function provided to you in the main.c file that takes as input the row, coloumn and tile index values as inputs respectively and returns a hashed output.

Once you have constructed your new mosaic image, $\mathcal{M}$, you must return it from your function.

Assumption: You can assume that `tile_size` will always be a proper divisor of `w` and `h` in the `draw_mosaic` function.

Performance tip: Check if `hash_function(`r`, `c`, i) < 0.2` holds then calculate δ else short circuit to the next tile.

When you are finished with the task you should upload a tarball containing your **distance.asm**, **resize.asm**, **mosaic.asm** and **helper.asm** files to Assignment 5 task 3 slot.

2 Mark Distribution

Activity	**Mark**
Practical	28
Assignment Task 1	7
Assignment Task 2	15
Assignment Task 3	50
Total	**100**