This project focuses on an image search problem: finding a specific face in an image or photograph. This is a problem that arises in many video surveillance applications as well as in searching large collections of digital images (e.g., finding all photos you took five years ago that contain your grandmother). In this assignment, we will focus on finding George P. Burdell, whose mugshot is shown in Figure 1, in a cartoon image that contains a crowd of cartoon faces, such as the sample scene in Figure 2. The faces in the scene, including George's, may be upright or may be rotated by 90, 180, or 270 degrees.

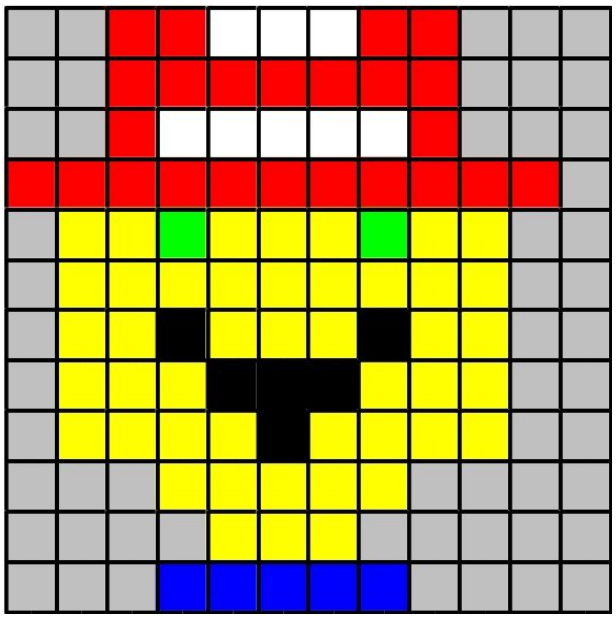

Figure 1: Mugshot of George P. Burdell

Disclaimer: Any resemblance to actual persons, roommates, or relatives is unintentional and purely coincidental.


Figure 2: Sample Crowd of Faces

We would like this task to be performed in real-time, so the computational and storage requirements must be kept to a minimum. We are also concerned with functional correctness of the algorithm (i.e., getting the correct answer), since we do not want to generate frequent false alarms while monitoring animations or miss an important match.

**Description**: George will appear exactly once (possibly rotated by 90, 180 or 270 degrees) in a random spot in a crowded scene. None of the faces in the scene will overlap with each other and all faces fit completely within the boundaries of the crowd (no partial faces hanging off the edges).

The faces that are not George will differ from him structurally (e.g., wear glasses) and/or by having different color features (e.g., a blue hat or green skin). Your task is to find the one face that exactly matches George's mugshot, but rotated either 0, 90, 180, or 270 degrees.

The crowded scene is a 64x64 image array of cells; each cell is one of eleven colors whose codes are given in the color palette shown in Figure 3. Each face fits in an 12x12 region of cells. The features of the face will always have color codes in the range 1, 2, …, 8. The background (non-face parts) of the image will always use colors whose codes are 9, 10, or 11.

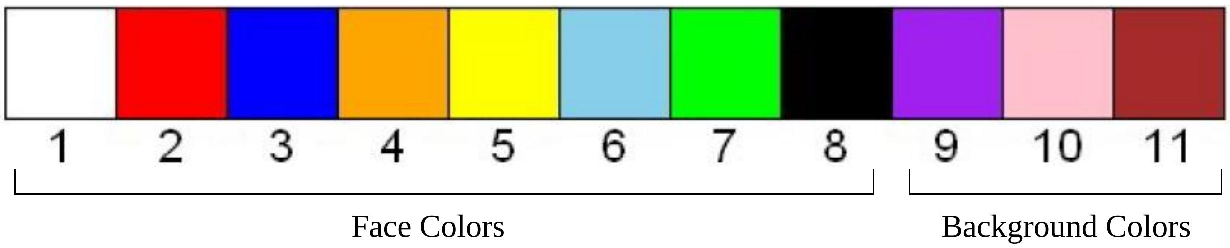


Figure 3: Color Palette

The image array is provided as input to the program as a linearized array of the cells in *row-maj*or order. The first element of the array (location 0) represents the color of the first cell in the first row. This is followed by the second cell (location 1) in that row, etc. The last cell of the first row (location 63) is followed by the first cell of the second row (location 64). The color code (1-11) is packed in an unsigned byte integer for each cell.

**Strategy**: Unlike many "function only" programming tasks where a solution can be quickly envisioned and implemented, this task requires a different strategy.

1. Before writing any code, reflect on the task requirements and constraints. *Mentally* explore different approaches and algorithms, considering their potential performance and costs. The metrics of merit are **static code length**, **dynamic execution time**, and **storage requirements**. There are often trade-offs between these parameters.
2. Once a promising approach is at hand, a high level language (HLL) implementation (e.g., in C) can deepen its understanding. The HLL implementation is more flexible and convenient for exploring the solution space and should be written before constructing the assembly version where design changes are more costly and difficult to make. For P1-1, you will write a C implementation of the program.
3. Once a working C version is created, it's time to "be the compiler" and see how it translates to MIPS assembly. This is an opportunity to see how HLL constructs are supported on a machine platform (at the ISA level). This level requires the greatest programming effort; but it also uncovers many new opportunities to increase performance and efficiency. You will write the assembly version for P1-2. *You'll hand in an intermediate draft P1-2-int.asm and the final version P1-2 at staggered due dates. Both files should contain a change log as described below.*

**P1-1 High Level Language Implementation**:

In this part, the first two steps described above are completed. It's fine to start with a simple implementation that produces an answer; this will help deepen your understanding. Then experiment with your best ideas for a better performing solution. Each hour spent exploring here will cut many hours from the assembly level coding exercise.

You should use the simple shell C program that is provided `P1-1-shell.c` to allow you to read in a linearized image array. The shell program includes a reader function `Load_Mem()` that loads the values from a text file. Rename the shell file to `P1-1.c` and modify it by adding your code.

Since generating example crowd images is complex, it is best to fire up Misasim, generate a crowd, step forward until the crowd is written in memory, and then dump memory to a file. Your C program should print the array index of the middle red pixel on the top of George's hat and the middle pixel on George's shirt. For example, for the crowd shown in Figure 4 (below), it should print this location as hat: 2731, shirt: 2742, the index of the pixels 42 rows down and 43 and 54 pixels across, respectively. (This test case is provided as `crowd-2731-2742.txt`.)

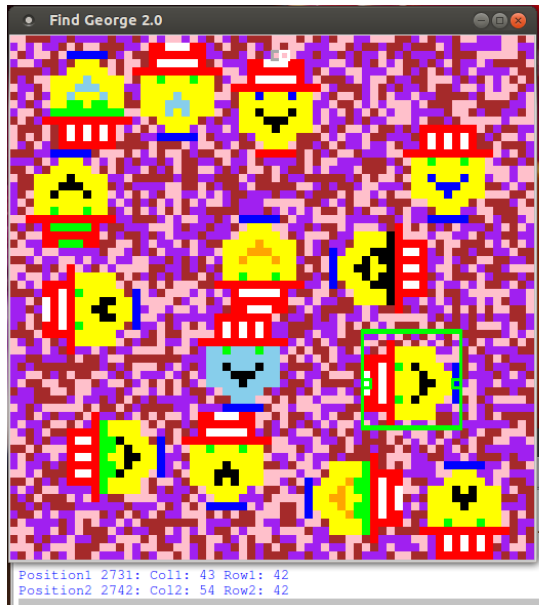


Figure 4: Crowd with Correct Answer: 2731 hat, 2742 shirt

The shell C program includes code that reads the crowd data in from an input file. It also includes an important print statement **used for grading** (please don't change it). If you would like to add more print statements as you debug your code, please wrap them in an if statement using a `DEBUG` flag – an example is given in the shell program – so that you can suppress printing them in the code you submit by setting `DEBUG` to 0. *If your submitted code prints extraneous output, it will be marked incorrect by the autograder.*

You can modify any part of this program. Just be sure that your completed assignment can read in an arbitrary crowd, find George, and correctly print his location (hat and shirt positions) since this is how you will receive points for this part of the project.

Note: you will not be graded for your C implementation's performance (speed and storage efficiency). Only its accuracy and good programming style will be considered (e.g., organizing and commenting your code). Your MIPS implementation might not even use the same algorithm; although it's much easier for you if it does.

When have completed the assignment, submit the single file `P1-1.c` to Canvas. You do not need to submit data files. Although it is good practice to employ a header file (e.g., `P1-1.h`) for declarations, external variables, etc., in this project you should just include this information at the beginning of your submitted program file. In order for your solution to be properly received and graded, there are a few requirements.

1. The file must be named `P1-1.c`. (Do not worry if canvas appends a version number.)
2. Your name and the date should be included in the header comment.
3. Your submitted file should compile and execute on an arbitrary crowd (produced from Misasim). It should contain the unmodified print statement, giving the solution. The command line parameters should not be altered from those used in the shell program.

4. Your program must compile and run with gcc under Linux. Compiler warnings will cause point deductions. If your program does not compile or enters an infinite loop, it will earn 0 correctness points.

5. Your solution must be properly uploaded to Canvas before the scheduled due date. The canvas P1-1 assignment has details on late penalties and policies.

**P1-2 Assembly Level Implementation:** In this part of the project, you will write the performance-focused assembly program that solves the Find Tumbling George puzzle. A shell program (`P1-2-shell.asm`) is provided to get you started.

Your program will call a software interrupt (570) that will generate a random crowd containing exactly one instance of George's face. *Your solution must not change the crowd image array (do not write over the memory containing the crowd).*

Your MIPS assembly code must compute George's location and report it by calling another software interrupt (571).

**Library Routines:** Here are the specifications of the three library routines you will use (accessible via the `swi` instruction).

**SWI 570: Create Tumbling Crowd**: This routine initializes memory beginning at the specified base address (e.g., `Array`). It sets each byte of the 4096 byte array to the corresponding cell's color code in the 64x64 crowd image array. The color codes are defined in the palette in Figure 3.

> INPUTS: $1 should contain the base address of the 1024 word (4096 byte) array already allocated in memory.
>
> OUTPUTS: none.

**SWI 552: Highlight Position**: This routine allows you to specify an offset into the Crowd array and it draws a white outline around the cell at that offset. Cells that have been highlighted previously in the trace are drawn with a gray outline to allow you to visualize which positions your code has visited.

> INPUTS: $2 should contain an offset into the Crowd array.
>
> OUTPUTS: none. *This is intended to help you debug your code; be sure to remove calls to this software interrupt before you hand in your final submission, since it will contribute to your instruction count.*

**SWI 571: Locate Tumbling George:** This routine allows you to specify the position of the middle red pixel at the top of George's hat and the middle pixel of his shirt to indicate the location where George has been found at a certain orientation.

> INPUTS: $2 should contain two packed numbers: in the upper 16 bits, the hat pixel position and in the lower 16 bits, the shirt pixel position. Each pixel position must be a number between 0 and 4095, inclusive. This answer is used by an automatic grader to check the correctness of your code.
>
> OUTPUTS: $3 gives the correct answer. You can use this to validate your answer during testing. If you call swi 571 more than once in your code, only the first answer that you provide will be recorded. The visualization will draw small yellow boxes at the pixel locations you provide and a larger 12x12 yellow box centered at these locations that you provide. It will also draw small green boxes at the pixel positions of the correct answer

and a larger 12x12 green box showing George's actual location. If your answer is correct, the green boxes should completely cover your yellow boxes. For example, in Figure 4, the correct answer is 0xAAB0AB6 (2731 decimal in top 16 bits; 2742 decimal in lower 16 bits) and when swi 571 is called with this value in $2, the green boxes appear as shown in Figure 4.

**Evaluation**: In this part (P1-2), correct operation and efficient performance are both important. The assessment of your submission will include functional accuracy during 100 trials and performance and efficiency. The code size, dynamic execution length, and operand storage requirements are scored empirically, relative to a baseline solution. The baseline numbers for this project are static code size: **123** instructions, dynamic instruction length: **4300** instructions (avg.), total storage required for registers, static memory, and stack memory: **16** words (not including dedicated registers $0, $31, and not including the 1024 words for the input puzzle array). *As a safety net, the dynamic instruction length metric is the maximum of the baseline metric and the average dynamic instruction length of the five fastest student submissions.*

Your score will be determined through the following equation:

$$\text{PercentCredit} = 2 - \frac{\text{Metric}_{\text{YourProgram}}}{\text{Metric}_{\text{BaselineProgram}}}$$

Percent Credit is then used to scale the number of points for the corresponding points category. For example, if your program uses half as much storage as the baseline, then PercentCredit is 1.5 and the number of points for the storage category (see Project Grading table below) is 10 scaled by 1.5 = 10*1.5 = 15. Important note: while the total score for each part can exceed 100%, especially bad performance can earn *negative credit*. The sum of the combined performance metrics scores (static code size, dynamic execution length, and storage) will be capped at zero; the sum of that portion of the grade will not be less than zero points. **Finally, these performance scores will be reduced by 10% for each incorrect trial (out of 100 trials). You cannot earn points for performance metrics if your implementation fails ten or more of the 100 trials**.

In MIPS assembly language, small changes to the implementation can have a large impact on overall execution performance. Often trade-offs arise between static code size, dynamic execution length, and operand storage requirements. Creative approaches and a thorough understanding of the algorithm and programming mechanisms will often yield impressive results. Almost all engineering problems require multidimensional evaluation of alternative design approaches. Knowledge and creativity are critical to effective problem solving.

In order for your solution to be properly received and graded, there are a few requirements.

1. A draft version of your code should be submitted in a file named `P1-2-int.asm`. This is due before the final version and will not be graded for accuracy or efficiency. It must contain a *change log:* a brief description of changes made from `P1-2-shell.asm` to this version of code. If this code does not incorporate substantive changes made to the shell code, points will be deducted from P1-2.

Here are some example entries:

```
# CHANGE LOG: brief description of changes made from P1-2-shell.asm
# to this version of code.
# Date Modification
# 02/12 Looping through pixels to find one w/ color $3
# 02/15 Reduced avg DI by only looking at pixels starting at row 10
```

2. The final version file must be named `P1-2.asm`. It must also contain a change log that records a brief description of changes made from `P1-2-int.asm` to this version.
3. Your final program must produce and store the correct value in $2, it must call SWI 571 to report your answer.
4. Your final program must return to the operating system via the **`jr`** instruction. *Programs that include infinite loops or produce simulator warnings or errors will receive zero credit*.
5. Your intermediate and final solutions must be properly uploaded to Canvas before the scheduled due dates.

**Project Grading**: The project grade will be determined as follows:

| *part* | *description* | *percent* |
|---|---|---|
| P1-1 | Find Tumbling George (C code) | 25 |
| P1-2-int | Intermediate draft of P1-2 (MIPS assembly) | -10 if missing |
| P1-2 | Find Tumbling George (MIPS assembly) | |
| | correct operation, proper commenting & style | 25 |
| | static code size | 15 |
| | dynamic execution length | 25 |
| | operand storage requirements | 10 |
| | *Total* | *100* |

**All code (MIPS and C) must be documented for full credit.**

**Honor Policy: In all programming assignments, you should design, implement, and test your own code. Any submitted assignment containing non-shell code that is not fully created and debugged by the student constitutes academic misconduct. You should not share code, debug code, or discuss its performance with anyone. Once you begin implementing your solution, you must work alone.**

*Good luck and happy coding!*