This project focuses on a constrained search problem: finding an exact match pattern, variably scaled in an image array (such as a specific face or pose that appears within a crowd scene). In this assignment, we will focus on finding George P. Burdell, whose mugshot is shown in Figure 1, in an image that contains a crowd of faces, such as the sample scene in Figure 2. The faces in the scene, including George's, may be at varying distances from the camera, so they may appear scaled. For example in Figure 3a, George is scaled two times the original size and in Figure 3b, he is scaled by a factor of three. All faces will be facing forward and upright (not rotated).

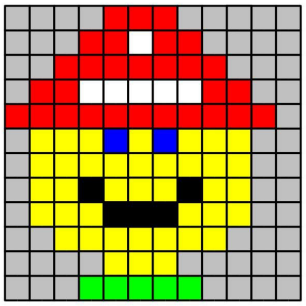

Figure 1: Mugshot of George P. Burdell
Disclaimer: Any resemblance to actual persons, roommates, or relatives is unintentional and coincidental.
All pixels shown are within the bounding box of George. Note that the rightmost *n* columns of the bounding box will always contain only background pixel colors, where *n* is the scaling factor.

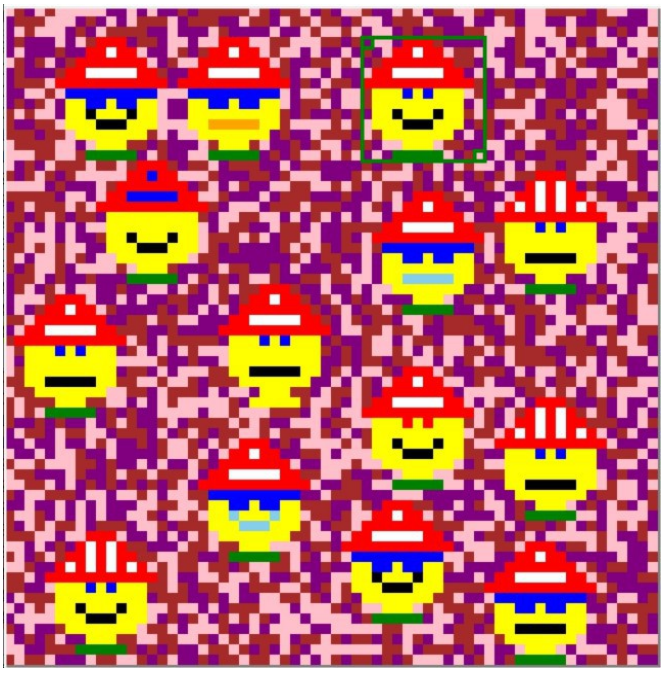

Fig. 2: Sample Crowd of Faces `(crowd-227-942.txt)`
George is found in the upper right. A bounding box is drawn around the pixels that match his mugshot.

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, or miss an important match.

George will appear exactly once (possibly scaled) in a random spot in each 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).

Like most college students, George has a limited wardrobe; he always wears a red hat and green shirt – and he's always smiling. The faces that are not George will differ from him by having different color features (e.g., an orange hat or green skin) and/or structurally (possible variations: direction of hat stripes, glasses/none, smile/frown).

Your task is to find the one face that exactly matches (possibly with scaling) George's mugshot shown in Figure 1. For example, in Figure 2, George is the face at the top and to the right.

Note: The figures of George and crowds that are shown in this document are repeated at the end of this document with color codes shown in each pixel. Please contact your instructor if you experience difficulty viewing these.

a) crowd-24-1519.txt b) crowd-1152-3427.txt

Figure 3: More Sample Crowds

The crowd scene is a 64x64 image array of pixels; each pixel is one of eleven colors whose codes are given in the color palette shown in Figure 4. Each *non-scaled* face fits in an 12x12 region of pixels. Each face may appear in the image scaled by a factor of 2, 3, 4, or 5. For example, in Figure 3a, George is scaled by a factor of 2 and fits in a 24x24 square region; in Figure 3b, he is scaled by 3 and fits in a 36x36 region. 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 4: Color Palette

The image array is provided as input to the program as a linearized array of the pixels in row-column order. The first element of the array (location 0) represents the color of the first pixel in the first row. This is followed by the second pixel (location 1) in that row, etc. The last pixel of the first row (location 63) is followed by the first pixel of the second row (location 64). This way of linearizing a two dimensional array is called *row-column mapping*. The color code (1-11) is packed in an unsigned byte integer for each pixel.

For example, if the top row of an image started with pixels in the three background colors (9, 10, 11, in that order), followed by a red pixel (2), followed by four purple (9) pixels, then the array would begin with a word packing 9 in byte 0, 10 in byte 1, 11 in byte 2, and 2 in byte 3. In Little Endian, this would create the word 0x020B0A09. (Recall 11 decimal is B hexadecimal and 10 decimal is A hexadecimal.)

For this assignment, you will write two programs, one in C and one in MIPS, as described below. You should design, implement, and test your own code. There are many possible approaches you can take for solving this problem. You are encouraged to try more than one and choose the one that best trades off constraints on code size, storage, and run time.

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 chosen, 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-first-draft.asm and the final version P1-2 at staggered due dates. All assembly files should contain a change log as described below.*

P1-1 High Level Language Implementation:

In this section, the first two steps described above will be 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. Use print statements to display the number of loops required, count the number of statements in each loop, or consider learning how to use a 'profiling' tool. 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 image search code.

Reporting your results: Your C program should print the index of the top left corner and the index of the bottom right corner of the *square* region (bounding box) containing George. For the example crowd shown in Figure 5 (below), it should print these locations as 1234 (the index of the top left pixel 18 pixels across and 19 rows down) and 2729 (the index of the bottom right pixel 41 pixels across and 42 rows down). This test case is provided as `crowd-1234-2729.txt`.

Note that the bounding box is square. Figure 1 shows that the rightmost column of an unscaled image will always include ONLY background colors (ANY background colors, not necessarily a solid color such as in Figure 1). If an image is scaled by n, there will be n columns of ONLY background colors on the right.

The shell C program 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 the shell program. Just be sure that your completed assignment can read in an arbitrary crowd, find George, and correctly print his location (corners of his bounding box) since this is how you will receive points for this part of the project. ***You must not change the statement that prints the results, as noted in the shell file.*** *Doing so will make your program fail when run by the autograder.*

Note: you will not be graded for your C implementation's performance. Only its accuracy and good programming style will be considered (e.g., using proper data types, operations, control mechanisms, etc., and documenting your code with comments). Your C implementation does not need to use the same algorithm as the assembly program; however, it's much easier for you if it does.

Test cases: A few test cases have been provided in the `tests.zip` file. As in previous assignments, we will provide a short "**smoke test" script (run_tests.sh)** with the P1-1 Assignment on Canvas. *Please place* **run_tests.sh** *in the same directory where you put your P1-1.c code.* You must run the smoke test on your code on the Linux Lab servers before submitting your code and fix any errors detected.

Be sure to run several more tests. The smoke tests do not represent the comprehensive set run by the autograder. You can create additional test files using MiSaSiM to run `P1-2-shell.asm`, go to the end of the trace, and use the "Dump" memory menu button to save the memory to a text file with the correct answer in the name of the file (derived from the value in $3 as shown in the `P1-2-shell.asm` program).

Submitting P1-1: When you 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 please 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 print out the results with the given print string. The command line parameters should not be altered from those used in the shell program.
4. Your program must compile and run with gcc on the Linux Lab servers. Compiler warnings will cause point deductions. If your program does not compile or enters an infinite loop, it will earn 0 correctness points.
5. Before submitting your code, reset DEBUG to 0 and run your code through the smoke tests on the Linux Lab servers to ensure that your code is providing the answer in the

proper format without any extraneous print statements. ***If your submitted code has a bug that the smoke test would detect, you will receive a 0.***

6. Your solution must include proper documentation (comments) and appropriate indentation.

7. Your solution must be properly uploaded to Canvas before the scheduled due date.

P1-2 Assembly Level Implementation: In this part of the project, you will write the performance-focused assembly program that solves the Find George puzzle. A shell program (`P1-2-shell.asm`) is provided to get you started. Rename it to `P1-2.asm`. *Your solution must not change the crowd image array (do not write over the memory containing the crowd).*

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

SWI 592: Create Scaled 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 pixel's color code in the 64x64 crowd image array. The color codes are defined in the palette in Fig. 3.

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

As a debugging feature, you can load in a previously dumped crowd testcase by putting -1 into register $2 before you call swi 592. This will tell swi 592 to prompt for an input txt file that contains a crowd. ***Be sure to remove the instruction putting -1 in $2 before submitting your file.***

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

> INPUTS: $2 should contain an offset into the Crowd array (an integer in the range 0 to 4095, inclusive).
>
> OUTPUTS: none.

This is intended to help you debug your code; be sure to remove calls to this software interrupt before handing in your final submission, since it will contribute to your instruction count.

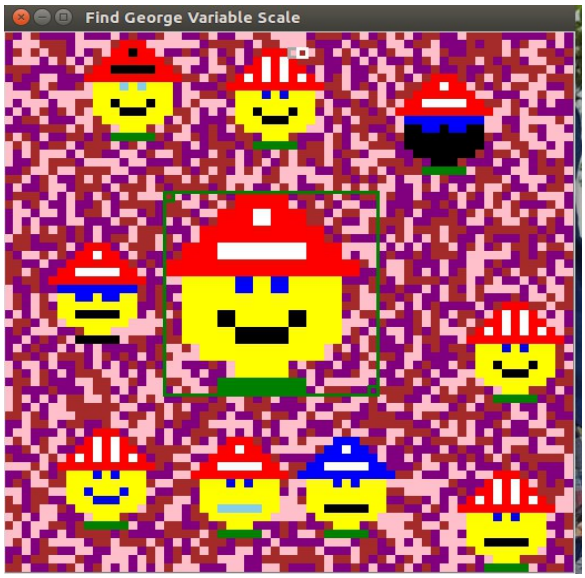

Figure 5: crowd (`crowd-1234-2729.txt`) with Correct Answer:
`George is located at: top left pixel 1234, bottom right pixel 2729.`

SWI 593: Locate Scaled George: This routine allows you to specify the position of the top left and bottom right corner pixels of George's *square* bounding box to indicate the location where George has been found.

> INPUTS: $2 should contain two packed numbers: in the upper 16 bits, the top left corner pixel location and in the lower 16 bits, the bottom right corner pixel location. Each location should 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 593 more than once in your code, only the first answer that you provide will be recorded. The visualization will draw a yellow box around the location you provide and a larger yellow box spanning the corner locations you provide. It will also draw a green box at the location of the correct answer and a larger green box showing George's actual location. If your answer is correct, the green boxes should completely cover your yellow boxes. For example, in Fig 5, the correct answer is `0x4D20AA9` (top left: 1234 = `0x4D2`; bottom right: 2729 = `0xAA9`) and when swi 593 is called with `0x4D20AA9` in $2, the green boxes appear as shown in Fig. 5.

Performance 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: **90** instructions, dynamic instruction length: **2400** instructions (avg.), storage required: **17** words (not including dedicated registers $0, $31, nor the 1024 words for the image array). *The registers used as inputs and outputs to the swi instructions and by the oracle count toward the total storage. You may use these registers for more than one purpose in optimizing your code.*

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 = 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 (code size, 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, the performance scores will be reduced by 10% for each incorrect trial (out of 100 trials). You cannot earn performance credit 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. Checkpoint1: A first draft version of your code should be submitted in a file named `P1-2-first-draft.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, you will not receive credit for this checkpoint.

   Here are some example entries:

   ```
   # CHANGE LOG: brief description of changes made from P1-2-shell.asm
   # to this version of code.
   # Date  Modification
   # 09/24 Looping through pixels to find one w/ color $3       (example)
   # 09/28 Reduced avg DI by only looking at pixels starting at row ...
   ```

2. Checkpoint2: No additional intermediate drafts need to be submitted, but about one week before the final due date, a short graded survey will ask you to describe your progress.
3. The final version of your code must be named **`P1-2.asm`**. (As mentioned above, it is OK if Canvas renames it slightly with an extended version number; we'll grade your most recent submission.) It must also contain a change log that records a brief description of changes made from the previously submitted intermediate draft to this version.
4. Your name and the date should be inserted at the beginning of the file.
5. Your program must call SWI 593 to report an answer and return to the operating system via the **`jr`** instruction. *Programs that include infinite loops or produce simulator warnings or errors will receive zero credit*.
6. Your solution must include proper documentation.

7. 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 George (C code) correct operation, technique & style	25
P1-2	Find George (MIPS assembly)	
	checkpoints, 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!

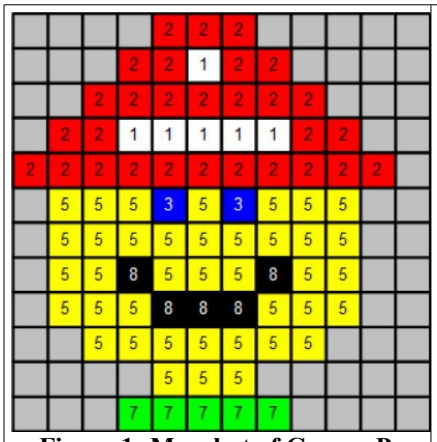

Figure 1: Mugshot of George P. Burdell

Note that the rightmost *n* columns of the bounding box will always contain only background pixel colors, where *n* is the scaling factor.

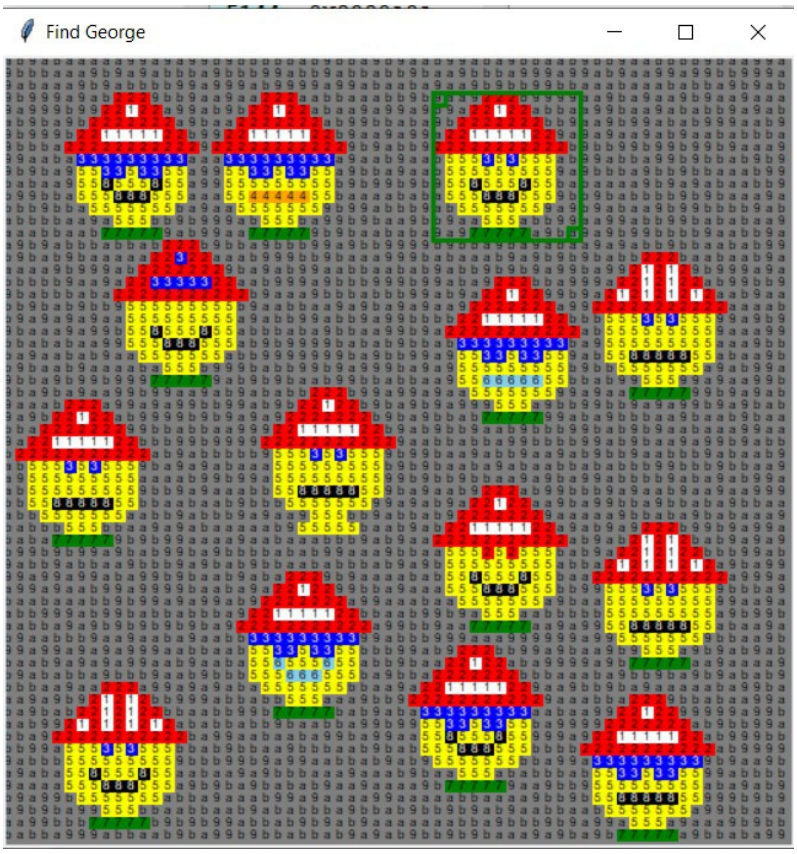

Fig. 2: Sample Crowd w/ Bounding Box `(crowd-227-942.txt)`

George is found in the upper right. A "bounding box" is drawn around the pixels that match his mugshot.

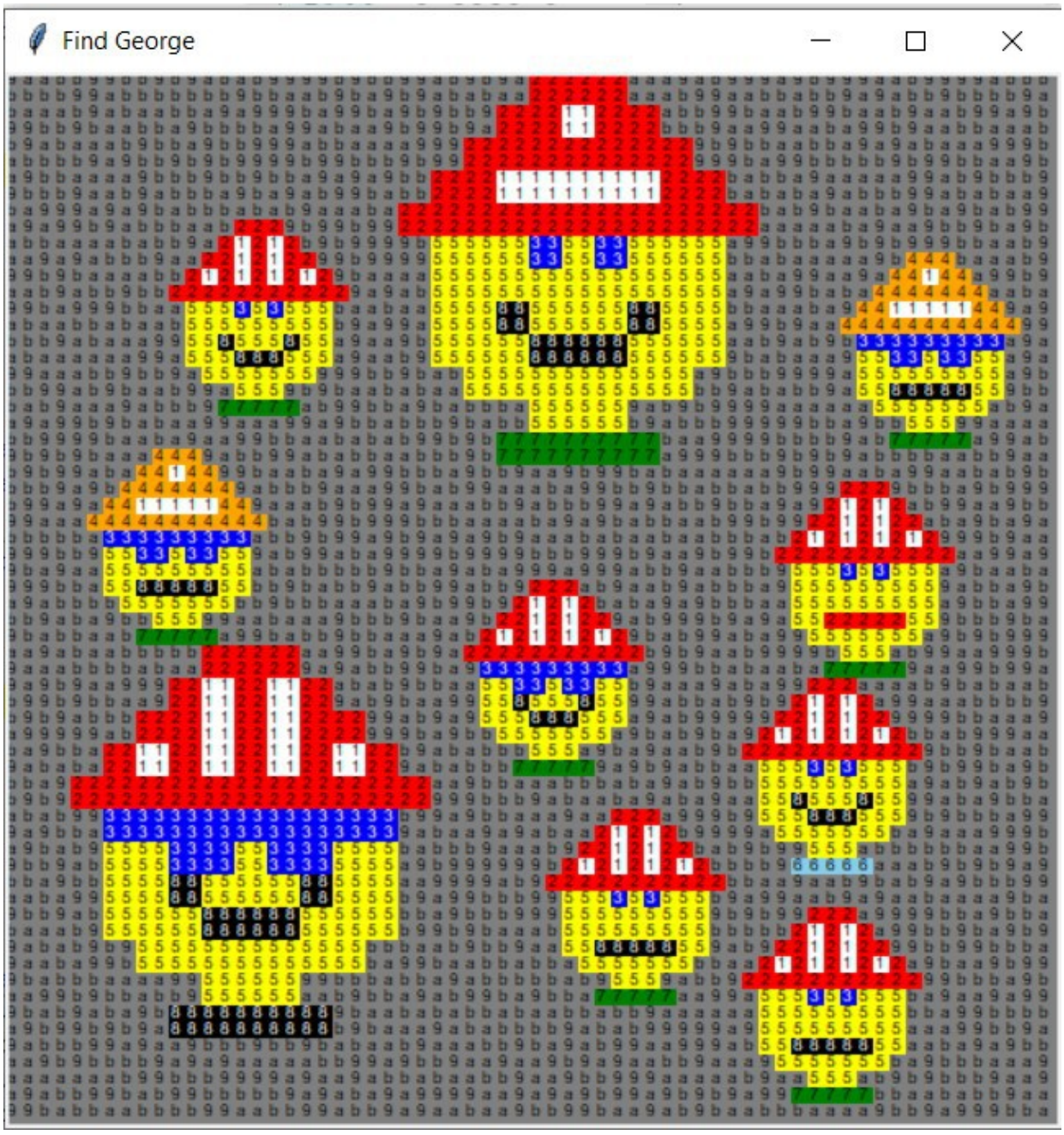

Figure 3a) crowd-24-1519.txt

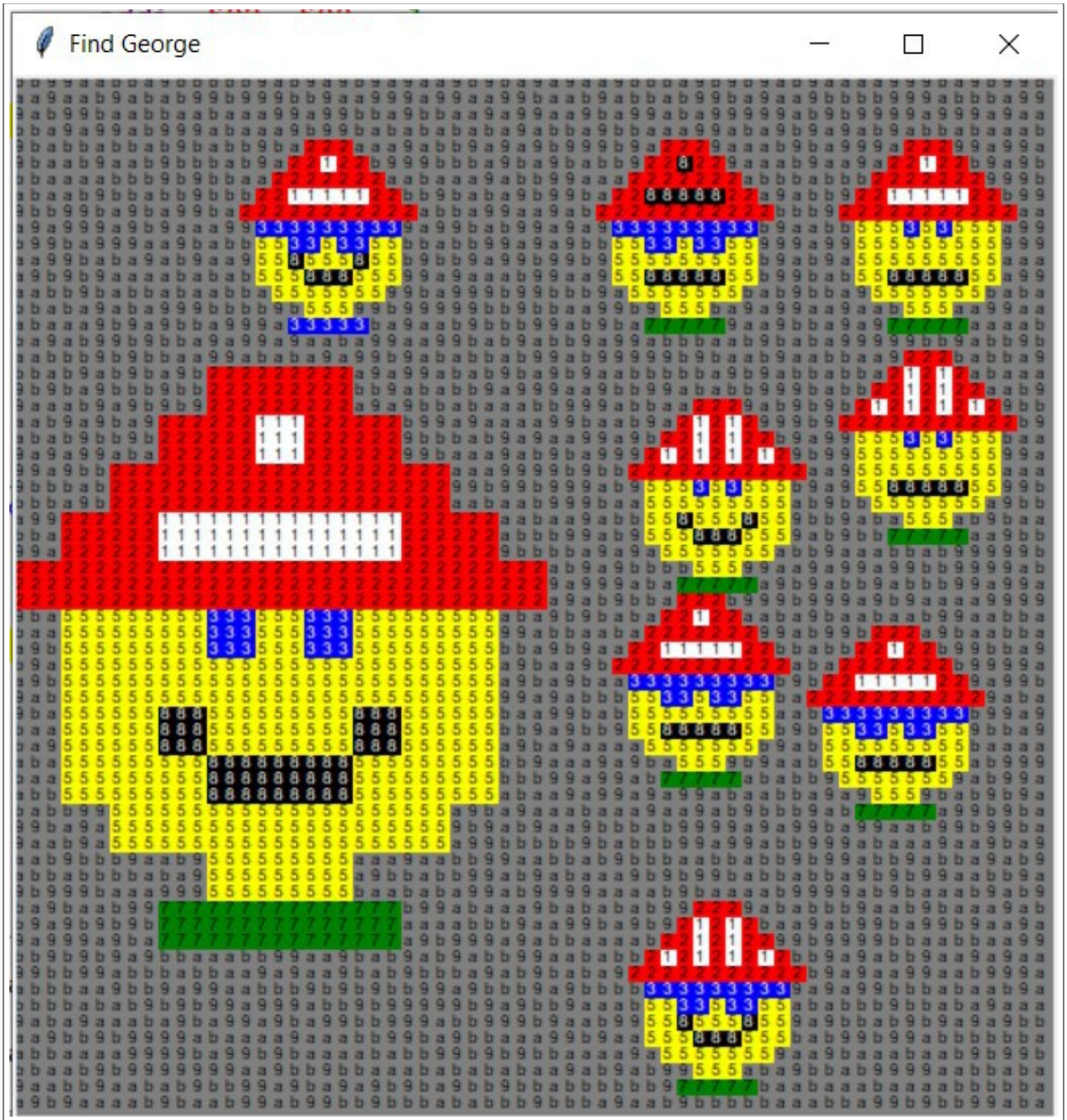

Figure 3b) crowd-1152-3427.txt

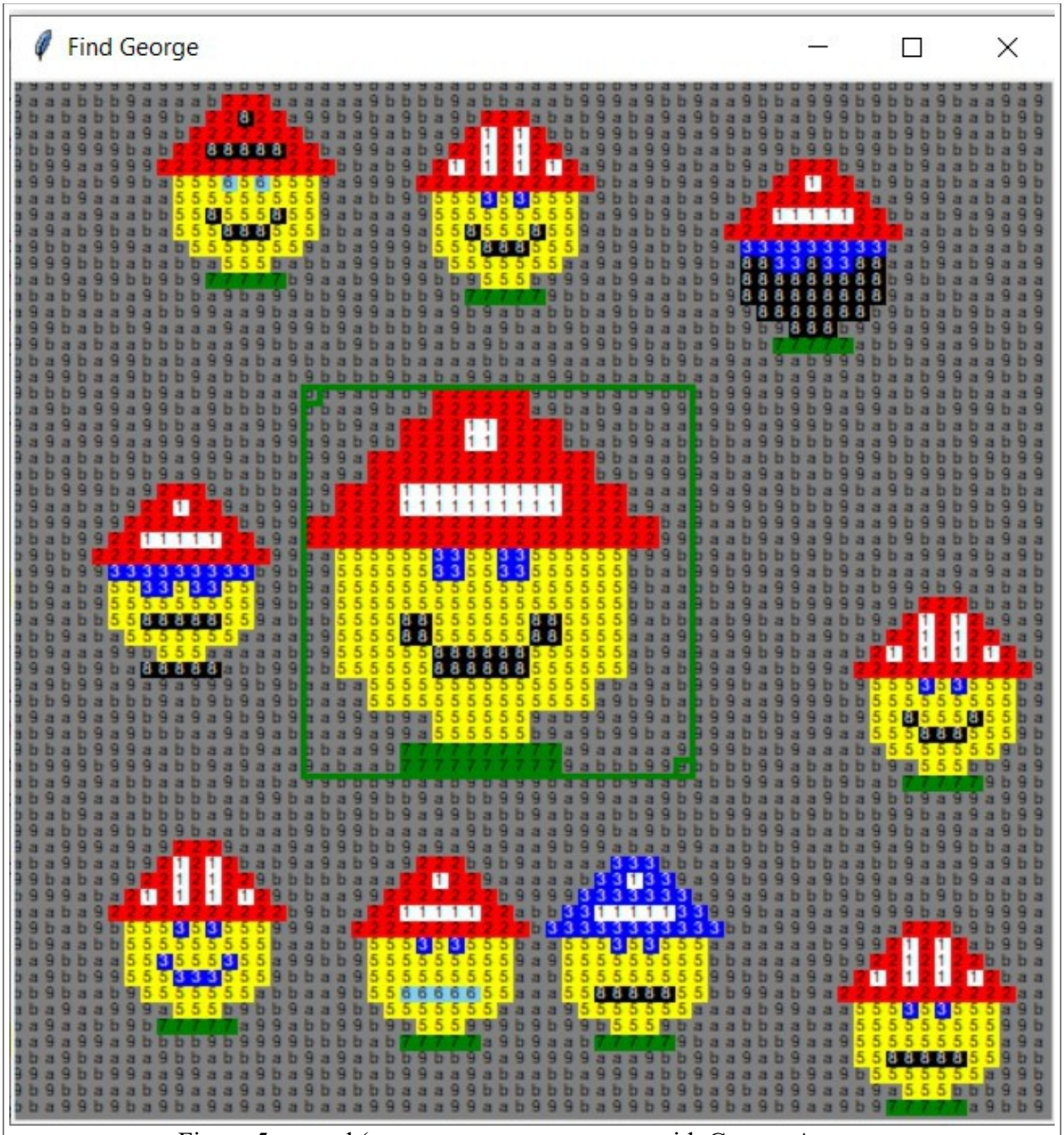

Figure 5: Crowd (`crowd-1234-2729.txt`) with Correct Answer:
`George is located at: top left pixel 1234, bottom right pixel 2729.`