Icon Matching: Consider an icon matching puzzle composed of eight *candidate* icons and one *pattern* icon that matches exactly one candidate. The goal is to identify which candidate icon matches the pattern. Figure 1 shows an example puzzle: the top row of eight icons are the candidates; the bottom row contains the pattern. The solution to this puzzle is candidate icon **7**. There will always be eight candidate icons and one pattern. There will be exactly one candidate that matches the pattern pixel-for-pixel (all pixels will match in corresponding pixel locations). All the candidate icons will be unique with respect to each other (no two candidates will match). In this task, we are concerned with both functional correctness and keeping computational and storage requirements to a minimum.

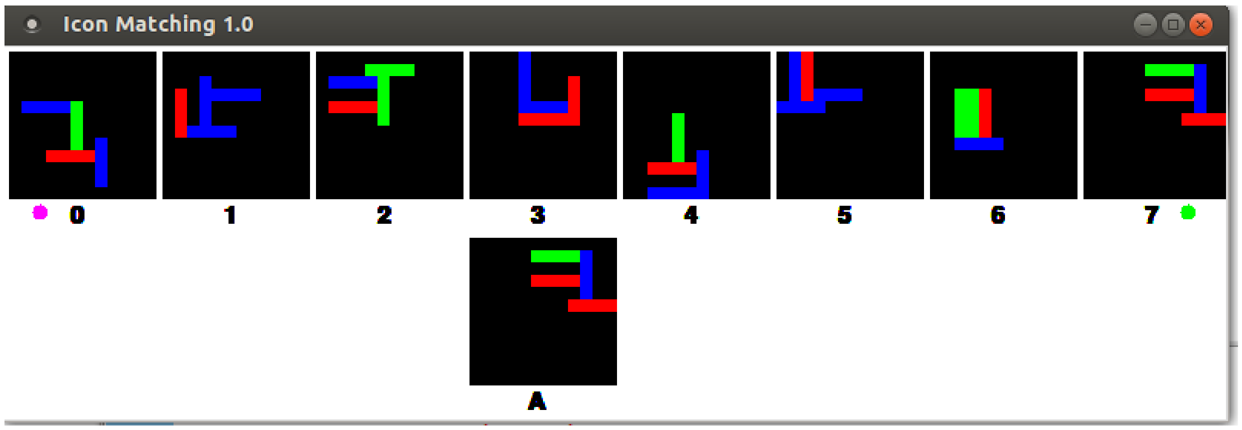

Figure 1. Icon Matching Puzzle: Top row is the candidate set; bottom row contains the pattern.

The icons are 12 by 12 arrays of pixels; each pixel is one of four colors (black=0, red=1, green=2, blue=3). The icons are provided as input to the program as a linearized array of the pixels in what is called *row-major order.* The first element of the array represents the color of the first pixel in the first row (i.e., leftmost pixel in top row). This is followed by the second pixel in that row, etc. The last pixel of the first row is followed by the first pixel of the second row. Each icon is stored contiguously in the array: the candidate icons are first, followed by the pattern icon.

The goal of this programming task is to find which of the eight candidate icons matches the pattern icon. The nonblack pixels of the pattern will never occur in the same configuration and location in more than one candidate. The parts of the pattern are not necessarily connected. Do not make assumptions about the number or size of the parts of the pattern. The nonblack pixels of the pattern will not occur as a strict subset of the nonblack pixels of any candidate. All the pixels (including the black pixels) of the pattern will occur in the same location in exactly one candidate.

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. Sometimes *back of the envelope* calculations (e.g., how many comparisons will be performed) can help illuminate the potential of an approach.

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 icon matching 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 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. 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 puzzle. 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.

Reporting your results: You can modify any part of the shell program. Just be sure that your completed assignment can read in a testcase puzzle, select the matching candidate icon, and correctly print the matching icon number using the print statement provided in the shell C code.

You must not change the print statement that reports the matching icon number. *Doing so will make your program fail when run by the autograder.* 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.*

Grading criteria: You will <u>not</u> be graded for your C implementation's performance (speed and storage efficiency). 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; although it's much easier for you if it does.

Test cases: A few test cases have been provided in the `tests.zip` file, with the file naming convention: test*i*.txt, where *n* is the answer (the matching icon number). For example, the matching candidate icon in "`puzzle5.txt`" is 5).

As in previous assignments, we 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 and place the given test cases in a subfolder called* ***tests***. 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 (the value given in $3 as described below).

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 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 puzzle (produced from Misasim). It should use the given print statement to print out a single integer identifying the matched candidate icon. 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 have 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 icon matching 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 puzzle array (do not write over the memory containing the candidate or pattern icons).*

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

SWI 584: Create Puzzle: This routine initializes memory beginning at the specified base address (e.g., `CandBase`) with the representation of 9 icons (8 candidate icons and 1 pattern icon). In particular, it initializes memory with 8 candidates followed by 1 pattern icon, with 12x12 words/icon = 1152+144 = 1296 words (each word represents a single color pixel encoded as an integer: 0, 1, 2, or 3).

INPUTS: $1 should contain the base address of the 1296 words already allocated in memory.

OUTPUTS: none.

Debugging feature: *You can load in a previously generated puzzle testcase by putting -1 into register $2 before you call swi 584. This will tell swi 584 to prompt for an input file that contains a puzzle. This should be a text file that was created by the Dump memory command in Misasim (e.g., puzzle7.txt). This is a feature that should only be used for debugging.* ***Be sure to remove the instruction assigning $2 to -1 before you submit your assignment.***

SWI 585: Mark Icon Pixel: This routine allows you to specify the address of a pixel in the icon puzzle and it marks this pixel by outlining it with a white square. It also keeps track of which pixels your program has previously marked (with previous calls to swi 585) and outlines these in gray.

> INPUTS: $2 should contain an address of a pixel. The address should be within the 1296 word region allocated for Candidates and the Pattern.
>
> 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 544: Match Ref: This routine allows you to report the number of the candidate icon that matches the pattern icon. Two dots will also appear in the puzzle display to indicate the submitted answer (magenta dot) and the correct answer (green dot).

> INPUTS: $2 should contain a number between 0 and 7, inclusive. This answer is used by an automatic grader to check the correctness of your code.
>
> OUTPUTS: $3 gives the correct answer, which is a number between 0 and 7. You can use this to validate your answer during testing. If you call swi 544 more than once in your code, only the first answer that you provide will be recorded.

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: **38** instructions, dynamic instruction length: **350** instructions (avg.), total register and memory storage required: **11** words (not including dedicated registers $0, $31, and not including the 1296 words for the input puzzle 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:

$$PercentCredit = 2 - \frac{Metric_{Your\,Program}}{Metric_{Baseline\,Program}}$$

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 (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
   # 09/12 Looping through pixels of an icon to find a nonblack one
   # 09/13 Reduced avg # dynamic insts by precomputing … before loop.
   # 09/19 Fixed bug with loop bounds causing infinite loop.
   ```

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.) 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 produce and store the correct value in $2, it must call SWI 544 to report your answer, and then 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 (comments).
7. Your solution must be properly uploaded to Canvas before the scheduled due date.

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

part	*description*	*percent*
P1-1	Icon Matching (C code)	25
P1-2	Icon Matching (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!