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).

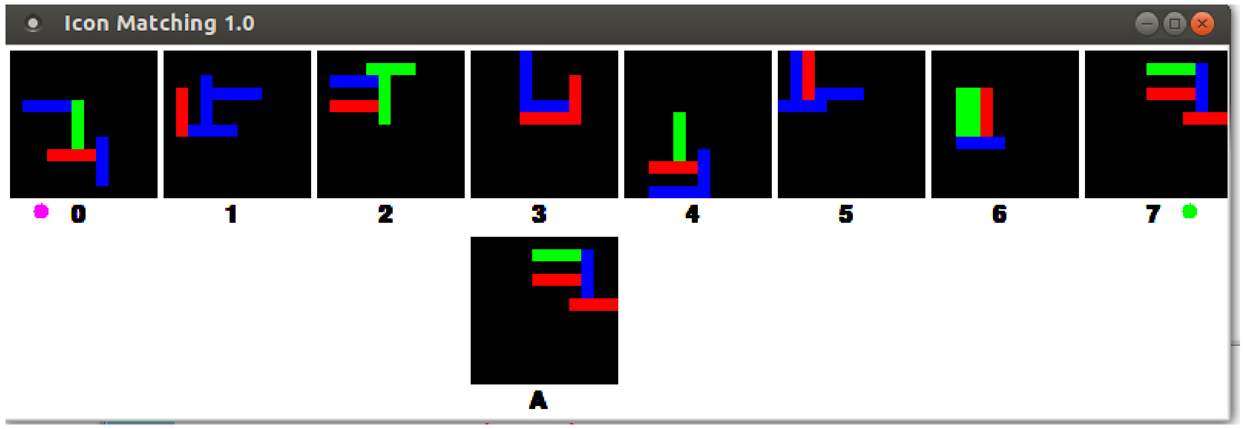

Figure 1. Icon Matching Puzzle: Top row is the candidate set; bottom row contains the pattern.

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 row-column 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. This way of linearizing a two dimensional array is called *row-column mapping*. 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.

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.

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. 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. Rename the shell file to `P1-1.c` and modify it by adding your code.

Since building the puzzle is complex, it is best to fire up Misasim, generate a puzzle, step forward until the puzzle is written in memory, and then dump memory to a file. The shell C program includes code that reads icon puzzle data in from an input file. It also includes an important print statement **used for grading** (please don't change). A few sample icon puzzles have been provided, with the answer given in the filename (e.g., the matching candidate icon in “`puzzle5.txt`” is 5).

You can modify any part of this program. Just be sure that your completed assignment can read in an arbitrary puzzle, select the matching candidate icon, and correctly complete the print statement 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. Only its accuracy and “good programming style” will be considered (e.g., using proper data types, operations, control mechanisms, etc., and documenting your code). 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.

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`.
2. Your submitted file should compile and execute on an arbitrary puzzle (produced from Misasim). It should contain the unmodified print statement, identifying the matched candidate icon. The command line parameters should not be changed. Your program must compile and run with gcc under Linux.
3. 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: There are three library routines (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.

New feature: *If $2 has value -1 when swi 584 is executed, a previously created puzzle can be loaded in. A dialog box pops up to ask for the puzzle's filename. 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 any assignments of $2 to -1 before you submit your assignment.*

SWI 544: Match Ref: This routine allows you to specify the number of one of the eight candidate icons that your code has identified as matching the pattern icon. Dots also show 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.

New routine: **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. *Be sure to remove calls to this swi in the code you submit, otherwise they will be counted in your static and dynamic instruction counts.*

In this version, execution performance and cost 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: 315 instructions (avg.), total register and memory storage required: 11 words** (not including dedicated registers $0, $31, nor the 1296 words for the input puzzle array). *The dynamic instruction length metric is the maximum of the baseline metric (315 instructions) and the average dynamic instruction length of the five fastest student submissions.*

Your score will be determined through the following equation:

$$PercentCredit = 2 - \frac{Metric_{Your\ \mathrm{Pr}\,ogram}}{Metric_{Baseline\ \mathrm{Pr}\,ogram}}$$

Percent Credit is then used to determine the number of points for the corresponding points category. 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 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 tradeoffs 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. The file must be named **`P1-2.asm`**.
2. Your program must call SWI 544 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.*
3. 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)	
	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!