Maths Games: Enhancing Sugar with 8 New Educational Math Activities

GSoC 2025 | Sugar Labs

Basic Information

Project Details

Project Name	Math Games
Skills	Python, Sugar
Mentors	Ibiam Chihurumnaya, Walter Bender
Project Size	350 Hours

Personal Details

Name	Ravjot Singh
Date of Birth	07-04-2004
Country	India (Nationality & Current Residence)
Email	ravu2004@gmail.com
Resume	Resume
GitHub	GitHub
CodeForces	CodeForces
Leetcode	LeetCode
Linkedin	Linkedin
Matrix	ravjot07
University	Atal Bihari Vajpayee Indian Institute of Information Technology and Management
Course	Bachelor of Technology (BTech) in Computer Science & Engineering
Contact Number	(+91) 8727954660
Time Zone	UTC+05:30 (Asia/Kolkata)
Preferred Language	English

Introduction

I'm Ravjot Singh—a passionate developer from Chandigarh, Punjab—with a deep commitment to the open source community. Over the past years, I've actively contributed to projects that make a real impact on education and technology. My journey with sugar labs began with contributions, where I enhanced projects such as **Music Blocks and Connect the Dots, and many more sugar activities** optimizing codebases and introducing new features to enrich children's interactive learning experiences. Working with Sugar Labs has not only deepened my technical expertise but also reinforced my belief in the transformative power of collaborative development.

In addition to my Sugar Labs contributions, I have also been an active member of the **Confidential Computing Consortium at Verasion,** where I implemented critical improvements like **command functionality enhancements, continuous integration pipelines, and security validations**. These experiences have allowed me to work on both the front-end and back-end challenges of modern open source projects, ensuring robust performance and reliability. **Furthermore I have created a prototype for 4 maths games till now.**

Furthermore, my role as an **LFX Mentorship Project Contributor** at the **Cloud Native Computing Foundation on the Kmesh** project has provided me with invaluable insights into designing and deploying **end-to-end testing strategies, streamlining CI workflows, and mentoring new contributors**. This mentorship experience not only honed my technical skills but also reinforced my passion for making a difference through open source innovation.

My motivation to contribute to open source stems from **a desire to build accessible and engaging educational tools**. I firmly believe that technology should empower learners—especially children—to explore, experiment, and develop critical thinking skills. By combining my experience with Sugar Labs' rich educational ecosystem and my drive for continuous improvement, I am eager to create innovative math games that foster curiosity, creativity, and a love for learning.

Contributions to Sugar Labs

As an active contributor to the Sugar Labs community, I have worked across multiple repositories including **Music Blocks**, **Connect the Dots**, **Mateton**, and various game-based Sugar activities. My contributions span UI improvements, bug fixes, code refactoring, performance enhancements, and documentation updates. I've focused on making the user experience more intuitive and the codebase more maintainable, while also contributing new features and fixing critical gameplay and rendering bugs. These efforts reflect my dedication to creating impactful, child-friendly educational tools.

PR	Repository	Contribution Summary
#3	`mateton-activity`	Remove simplejson dependency and use standard library json instead

#2	fourier-series-visualisation	feat: display real-time frequency and circle count in UI
#1	fourier-series-visualisation	Fix: Prevent "width greater than radius" error in Pygame circle drawing
#21	connect-the-dots-activity	Updated README with clearer installation and contribution instructions.
#26	connect-the-dots-activity	Feat: Add Undo and Redo functionality
#24	connect-the-dots-activity	Fixes #23: Updated- lib/sugar-web
#22	connect-the-dots-activity	Refactored redundant JavaScript code to enhance maintainability.
#19	connect-the-dots-activity	Added clear button and refractored code
#4150	musicblocks	Add: Test for utils.js
#4117	musicblocks	Feat: Added live visual feedback for audio input in Sampler Widget
#42	make-them-fall-activity	Fix: Activity Start error
#18	hittheballs-activity	Resolved ball movement issues after game reset.
#562	www	Fixed typos

Besides my work with Sugar Labs, I have also contributed to other significant open-source projects, including **Veraison** under the **Confidential Computing Consortium** and **Kmesh** under the **Cloud Native Computing Foundation (CNCF)**.

My Contributuions to Project Version:

PR	Repository	Contribution Summary
#14	cocli	Modified command functionality to support unsigned CoRIMs, improving system compatibility.

#10	`gen-corim`	Developed and integrated a CI pipeline for running unit tests on each commit.
#20	`cocli`	Authored detailed documentation for CoMID templates and extension points.
#22	`cocli`	Redesigned display options for enhanced modularity and maintainability.
#143	`corim`	Implemented MAC/IP address validation in Measurement.Valid to enhance reliability and security.

My Contributuions to Project Kmesh:

PR	Title	Description
#1277	Add E2E test for Locality Load Balancing	Developed an end-to-end test validating locality-based traffic distribution, ensuring correct behavior under load balancing rules.
#1243	feat(authz): Add E2E test for kmeshctl authz functionality	Implemented tests for authorization CLI commands, improving coverage and reliability of `kmeshctl`.
#1238	feat: Add readiness probe for Kmesh daemon	Added a readiness probe to ensure safe and stable daemon startup before traffic is accepted.
#1231	Fix CI: Regenerate docs for authz subcommands	Fixed CI issues by updating auto-generated documentation for authorization commands.
#1226	Add Unit Tests for logs Package	Introduced unit tests to increase coverage and validate logging functionality.

Topic	Page No.
Personal and Project Details	1
Introduction	2
Contributions to Open source	3-5
Project Idea	
Synopsis	6
Summary of the Project	6
What will be Achieved	7
Deliverables	8-35
Proposed Timeline	35-37
Availability	37
Post GSoC Plans	38
Conclusion	38

PROJECT IDEA

Synopsis:

This proposal outlines a plan to develop "Math Games", a Sugar Labs project **to create eight new interactive math activities for the Sugar learning platform as part of Google Summer of Code (GSoC) 2025.** The project will deliver a suite of puzzle-style games covering a range of mathematical concepts – from logic puzzles like the Four Color Map and Fifteen Puzzle to arithmetic challenges and two AI-powered activities. These games are designed to engage children in learning mathematics through play, reinforcing concepts such as graph coloring, arithmetic operations, spatial reasoning, number theory, and introductory artificial intelligence. By leveraging my experience as an active Sugar Labs contributor and strong Python developer, I will implement each activity as a Sugar-compatible program with intuitive user interfaces and educational value. The end result will be a collection of polished math games that enrich Sugar's library of activities, encourage problem-solving and curiosity in children, and demonstrate Sugar Labs' continued innovation in educational technology.

Summary of the Project:

Sugar Labs' mission is to provide learners with tools for exploration and discovery, and mathematics is a core skill that benefits greatly from interactive learning. While Sugar already includes many activities, there is always room for more math-focused games to captivate young minds Link. **"Math Games"** addresses this by introducing eight new math puzzle activities into the Sugar environment. Each activity is a standalone Sugar app focusing on a specific concept or puzzle:

- **Four Color Map Game:** A puzzle based on the Four Color Theorem, where children color regions of a map with at most four colors without adjacent regions sharing a color.
- **Broken Calculator:** An arithmetic challenge where the "calculator" doesn't function normally – instead of giving answers, it asks the player to form equations equal to a target number using limited operations, fostering creative problem solving with addition, subtraction, multiplication, and division.
- **Soma Cubes:** A 3D spatial reasoning puzzle that lets learners assemble virtual 3D "Soma" pieces to form a cube (and possibly other shapes), teaching geometry and visualization.
- **Fifteen Puzzle:** The classic 4x4 sliding tile puzzle that exercises planning and sequencing as players arrange numbered tiles in order.
- **Euclid's Game:** A two-player number game based on the Euclidean algorithm, where players take turns subtracting differences of numbers aiming to be the one with an even number of moves at game's end, introducing concepts of gcd (greatest common divisor) and strategic thinking.
- **Odd Scoring:** Another two-player strategy game involving moving a token on a number line by 1, 2, or 3 steps per turn, with the twist that the winner is determined by parity (even/odd count of moves), encouraging players to think ahead about parity of outcomes.

- **Make an Identity:** An algebraic puzzle where players are given a sequence of digits and a target number, and must insert arithmetic operations to form an equation (identity) that evaluates to the target, providing excellent practice in arithmetic and order of operations.

- **Number Detective (AI-based):** An interactive pattern-recognition game in which the child enters a sequence of numbers and an AI "detective" guesses the next number. If the guess is wrong, the child provides the correct answer, and the AI learns from this feedback over time, demonstrating basic machine learning concepts in a fun way.

- **Sorting Hat AI (AI-based):** A classification game (inspired by the idea of a sorting hat) where the child labels a set of items (such as shapes, animals, or numbers) into categories, and an AI tries to classify new items. The child corrects any mistakes, thereby training the AI. This activity introduces children to how AI algorithms like decision trees or k-nearest neighbors work for classifying data.

Over the course of the GSoC project, I will design and implement each of these activities in Python using the **Sugar toolkit (GTK-based UI framework)**. The activities will integrate seamlessly with the Sugar desktop, **following its UI/UX guidelines so that they feel like native Sugar apps**. Each game will include features such as intuitive controls (point-and-click or drag-and-drop interfaces appropriate for kids), visual feedback (colors, messages, or sounds to guide and congratulate the user), and resetting or difficulty options where applicable. Two of the activities **(Number Detective and Sorting Hat AI) will incorporate simple artificial intelligence logic, using rule-based algorithms and basic machine learning techniques to enable the software to learn from the player's input.** By the end of the summer, the project will produce eight fully-functional, tested, and documented Sugar activities ready for distribution. This will expand Sugar's collection of math activities, offering new ways for children to explore mathematical ideas and even experiment with teaching a computer, all within Sugar's collaborative learning ecosystem.

What will be Achieved:

By the completion of this project, the following will be achieved:

- **Eight new Sugar activities** focusing on mathematics will be developed, each thoroughly tested and packaged for Sugar. These activities will be ready to install and use on Sugar deployments (such as on the XO laptops or Sugar on a Stick), expanding the available content for learners.

- Each activity will have a polished user interface, instructions or help sections for students (and optionally teachers) to understand how to play, and integration with Sugar's standard features (for example, using the Sugar Journal to save progress or scores if appropriate).

- The math games will collectively cover diverse topics: coloring problems, arithmetic operations, spatial puzzles, number theory games, logic and strategy games, and introductory AI concepts. This comprehensive set ensures a wide educational impact across different areas of math and computing.

- The two AI-based games will demonstrate a practical integration of basic machine learning within Sugar activities. This includes implementing or utilizing simple algorithms (like decision tree learning or k-NN classification) in an offline environment. The achievement here is not just the games themselves, but also creating a template for how AI can be used in Sugar activities in a child-friendly way.

- All source code developed will be contributed to Sugar Labs' repositories, **following best practices (well-documented, with readable code and tests).** This makes the work sustainable: other contributors can maintain or enhance the games in the future, and pieces of the code (for example, a generic puzzle framework or a simple AI module) could potentially be reused in other Sugar activities.

- A successful GSoC completion will also mean thorough documentation is produced. This includes **developer documentation (so that Sugar Labs community can continue the work) and user documentation (such as a short guide for each game explaining rules and how to use it, which could be included in the activity or on the Sugar Labs wiki).**

- Finally, as the developer, **I will have deepened my experience in creating educational software.** This personal achievement translates to benefit for Sugar Labs: **I plan to remain involved beyond GSoC, using the knowledge gained to help in future projects or mentoring others**. In essence, this project will result in eight new activities and a more experienced contributor, thereby strengthening the Sugar community.

Deliverables:

The project will deliver eight new interactive math game activities for Sugar. Below are detailed descriptions of each activity, including what the activity is, how it will be implemented **(plan, features, and required technology)**, and its educational impact:

1. Four Color Map Game – *Map Coloring Logic Puzzle*

Description & Educational Goal: This activity is based on the Four Color Theorem, which states that any planar map can be colored with at most four colors such that no two adjacent regions share a color. The game presents the learner with a map (or a pattern of regions) and a palette of four colors. The **objective is to color all regions without violating the adjacency rule**. By solving these puzzles, children practice logical reasoning and learn about graph coloring in a hands-on way. It subtly introduces them to a famous idea in mathematics in a playful format and improves their planning skills (since a wrong color choice early on might force a conflict later). It's challenging but not too difficult, making it suitable for a range of ages.

Implementation Plan & Features: The Four Color Map Game will be implemented as a graphical activity using the Sugar graphical toolkit (PyGTK/GTK3). I will create or include several pre-drawn map layouts (for example: a simple abstract pattern, a map of regions shaped like countries or puzzle pieces, maybe even a map of a fictional place or the United States outlines as in some examples (transum.org). These maps will be vector shapes or predefined regions so that the program "knows" which areas count as adjacent. When the activity starts, it will display the chosen map in an uncolored state. The UI will include a set of four color buttons (or swatches) that the player can select from. The player colors a region by clicking on a region in the map; this will fill that region with the currently selected color. The core logic will track adjacency: for each region, the program has a list of which other regions are its neighbors. After the player colors regions, a "Check" button (or an automatic check)

will verify whether any two neighboring regions share the same color. If a conflict is found, the game can highlight the problematic regions (e.g., outline them in red) to prompt the player to reconsider those choices. If the coloring is successful (all adjacency constraints satisfied and all regions colored), the game congratulates the player – possibly with a fun animation or a message – and perhaps offers another map to color.

Features to include:

- Multiple map puzzles of varying difficulty. For example, an easy level might have a small number of regions or a map that is almost tree-like (few adjacencies), while a hard level could be a complex map with many regions tightly adjacent.

- A palette of exactly four colors (enforcing the concept of the four-color limit). We might allow reuse of colors freely (the challenge isn't limiting color use, but placing them correctly).

- Undo/reset functionality: players should be able to clear a region or reset the whole puzzle to try again. Also, selecting a different color and recoloring a region should be possible.

- Feedback: a "Check solution" function that tells the player if the puzzle is solved correctly. Optionally, we can have a live check that immediately warns if a move causes an adjacent conflict (though we might allow conflicts while they are in the middle of coloring and only finalize checking when they hit "Done", to let them experiment).

- Hints (if time permits): Perhaps a hint button that colors one region correctly or points out a region that can be colored without conflict as a next step.

- Internally, the game will use data structures for the graph of regions. We'll likely represent the map as a set of nodes (regions) and edges (adjacency relations). This way, checking the coloring is efficient (just ensure no edge connects two same-colored nodes). This is a straightforward use of graph logic in Python.

- Technology: I will use Cairo or a similar drawing API via Sugar to render the map shapes and fill colors. The Sugar toolkit supports SVG or canvas drawing, which I can use for scalable graphics. Python's data structures will handle the adjacency logic. No external libraries are needed beyond what Sugar provides; it's mostly custom logic and UI event handling for clicks.

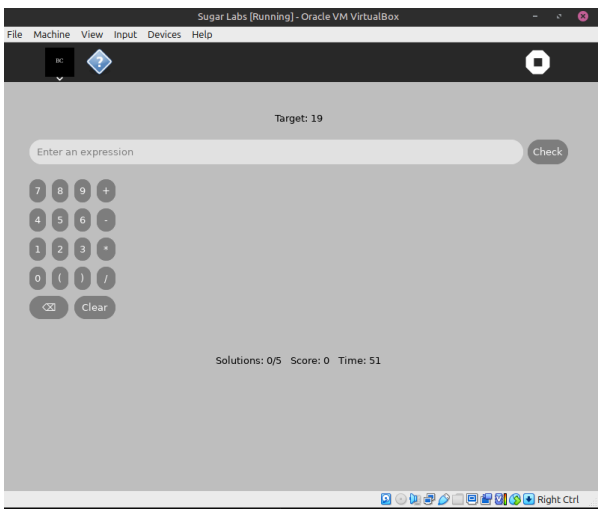
Tools Required: Sugar's activity API (for setting up the activity window, toolbars, etc.), a graphics library (Cairo via PyGTK for drawing regions, or possibly Pygame if needed, but likely not necessary), and basic Python for logic. I may design the maps using an external tool (like Inkscape) and then import the coordinates or images into the activity.

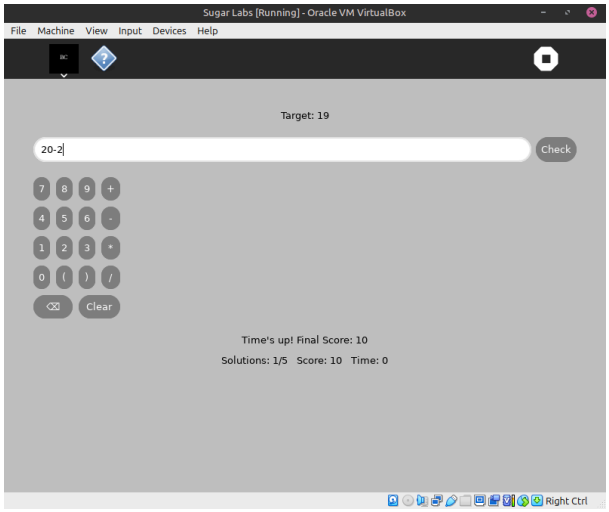
Educational Impact: This game teaches logical thinking and introduces graph theory concept in a simple way. Children learn through trial and error that you sometimes need to leave a color available for a tricky region and plan your coloring accordingly. They also learn persistence – if their first coloring doesn't work, they can try a different combination. The thematic element of coloring maps makes it enjoyable and less like a "math exercise"

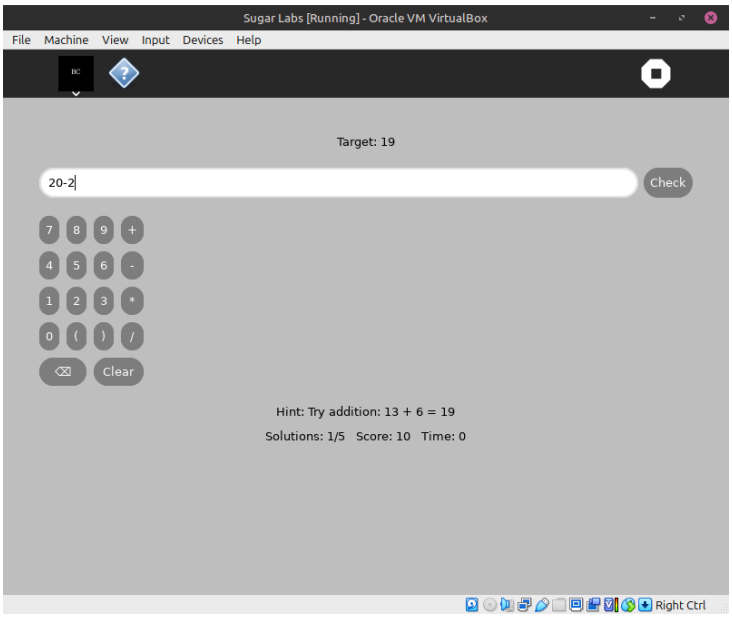
and more like a creative activity, which can engage students who are visual or artistic while still sneaking in a math/logic concept. It could even lead to discussions: a teacher or parent might ask, **“Did you know mathematicians proved you never need more than four colors for any map? You just discovered one example of that!”**. Thus it can inspire curiosity about math beyond arithmetic.

2. Broken Calculator – *Target Number Arithmetic Challenge*

Prototype:- GitHub







Description & Educational Goal: In Broken Calculator, the usual roles of a calculator are reversed. Normally, you input an expression and the calculator gives the answer; here, the game gives a target number and the student must come up with expressions that evaluate to that number. The “calculator” appears to be broken because it won’t directly compute for the student – instead, the student does the thinking. The goal is to find **five different equations** that all result in the target number toytheater.com. For example, if the target number is 10, a player

might input `5+5 = 10`, `12-2 = 10`, `2*5 = 10`, `20-10 = 10`, and `11-1 = 10` as five distinct ways to make 10. This activity builds fluency in arithmetic operations (addition, subtraction, multiplication, division) and encourages flexible thinking about numbers. Students practice composing and decomposing numbers in various ways and become more comfortable with mental math and order of operations.

Implementation Plan & Features: The Broken Calculator activity will present a calculator-like interface with an important difference: it displays a **target number** prominently on the screen. The user's task is to enter equations (using an on-screen keypad or keyboard input) that equal that target. The interface will likely have:

- A display that shows the target (e.g., "Target: 37") and maybe a list or count of equations found so far.
- An input area where the student can construct an equation. This could be done by clicking number buttons (0-9) and operation buttons (+, –, ×, ÷) and an equals sign, or by typing into a field that is restricted to valid characters.
- A submit button (or pressing Enter) to check the equation.

When the student submits an equation:

- The program will parse the equation and evaluate it (taking care to handle the operations in the correct order, or perhaps we enforce them to input in a linear fashion but then use Python's eval safely on it after validation).
- It will check if the equation is valid (does it actually compute to the target number, and is it formatted correctly).
- If valid and equals the target, it will check that this equation hasn't been used before (to ensure uniqueness). If it's new, it gets added to the list of found solutions and the score count increases. The game might give positive feedback ("Great! 37 = 40–3. Find 4 more ways to make 37.").
- If the input doesn't equal the target or is not a valid expression, the game will notify the user ("That doesn't equal 37, try again" or "Invalid equation, please use proper format").
- The challenge is complete when the user finds 5 correct equations for the target (or some preset number of equations). At that point, the game can celebrate the success and optionally present a new target number to continue play for more practice.

Additional features:

- **Random target generation:** The game will generate a new target number each round. We might configure the range of the target based on difficulty level or grade (e.g., for younger kids, targets between 10 and 50; for older, include larger numbers or even allow negative targets with subtraction).

- **Operation limitations (optional):** For variation, sometimes some buttons could be "broken" (disabled) to increase challenge – e.g., "the 7 key is broken, find ways to make 50 without using 7." This adds a twist that forces creativity. This wasn't explicitly described in the original idea, but could be an extension because "broken calculator" puzzles often have that variant. If included, the activity might occasionally indicate that certain digits or operations cannot be used, adding an extra constraint.

- **Points or timer (optional):** We could introduce a scoring system or time challenge. For instance, give points for each unique correct equation, or see how many equations the student can find in a fixed time. However, such gamification is secondary to the educational goal and will only be added if it enhances motivation without causing stress.

- **Help features:** Perhaps an "Example" button that gives one equation (especially for demonstration when a new user starts) to show how it works. Or a "Hint" that suggests an operation to use ("Try using subtraction") if the student is stuck.

From a technical standpoint, parsing and evaluating user-entered equations is a key part. I will likely leverage Python's evaluation for arithmetic expressions but in a safe manner: ensure the input only contains numbers, +, -, *, /, and perhaps parentheses. I can write a simple parser or use Python's `eval` after sanitizing input (since we are offline and it's a controlled input, this might be acceptable, but I will be careful). Alternatively, implement a small expression evaluator manually to be safe. Ensuring the expression equals the target is straightforward after evaluation. The UI will be done with Sugar's standard widgets: buttons for digits/ops or a text field for input, and labels for display. The activity loop will manage the state of equations found.

Tools Required: Mostly just Python and the Sugar UI toolkit. No external math libraries are needed beyond what Python offers for arithmetic. If implementing a custom parser, that will be pure Python logic. If using the on-screen keypad approach, I'll need to design the layout of buttons using Sugar's layout containers. This activity is largely about logic and UI, not requiring additional frameworks.

Educational Impact: Broken Calculator directly strengthens arithmetic skills and number flexibility. Students learn that there are many ways to compose a number, which reinforces the concept of equivalence and that arithmetic operations can undo each other (addition vs subtraction, multiplication vs division). It also subtly encourages understanding of order of operations and use of parentheses, especially if we allow more complex expressions; for example, to express 37, a student might do (40 + 8) / 2 = 24 (which is incorrect) and realize they need to adjust or use a different approach. This trial and error builds a deeper intuition for how operations work. Additionally, it fosters creativity and perseverance: some targets might be easy (like 10, which kids can find many ways), while others require more thought. The process of searching for multiple solutions means even after finding one correct solution, they must continue to think of alternatives, thereby extending their exploration of that number's properties. For instructors, the game provides insight into a student's understanding – the equations a learner comes up with can reveal their grasp of operations. In a classroom setting, students could share particularly clever or complex equations they found, further learning from each other. Overall, Broken Calculator turns what could be rote practice (drilling addition or subtraction) into a game of discovery with numbers.

3. Soma Cubes – *3D Spatial Puzzle*

Description & Educational Goal: The Soma cube is a classic 3D puzzle consisting of seven pieces made up of unit cubes that assemble into a 3x3x3 cube (there are also many other shapes these pieces can form). The educational value of Soma Cubes lies in spatial visualization, problem-solving, and patience. By manipulating 3D pieces, children develop an intuition for how shapes fit together in space. This translates to improved geometry skills and can spark interest in 3D design or architecture. In the Sugar activity version, the goal will be to assemble the pieces into certain shapes – primarily the 3x3x3 cube, but possibly other interesting target shapes (for example, a chair shape, pyramid, etc., which are known challenges with Soma pieces). The puzzle is inherently open-ended and trial-and-error, which encourages exploration; learners can try different piece orientations and positions until they complete the shape.

Implementation Plan & Features: Implementing a 3D puzzle in a 2D environment is challenging, but there are a couple of approaches:

- One approach is to create a simplified **3D visualization** within the Sugar activity. I can utilize a Python library or module for 3D graphics (perhaps something like PyOpenGL, or even a simpler 3D engine if one is compatible with Sugar). This would allow the pieces to be shown in perspective, and the user could rotate the view or the pieces. The seven Soma pieces (which are combinations of 3 or 4 unit cubes in various configurations) can be modeled as collections of small cubelets. The user interface would let the player select a piece, rotate it around axes, and position it on a virtual grid.

- Another approach is a more **planar (2D) representation** of the 3D puzzle: for example, showing projections or layers. We could display the puzzle as three 2D layers (each layer is a 3x3 arrangement of cubes). The user might place piece segments on these layers. However, this might be hard for children to use, since visualizing from layers is almost as hard as visualizing in 3D.

- Perhaps the most user-friendly method is a semi-3D (isometric) view with drag-and-drop. I might use an isometric grid to represent the 3D space so that pieces can be shown somewhat in 3D without full perspective rotation. This could be done by drawing cubes in isometric projection (which essentially looks like a 3D-ish board). The player could then drag pieces onto the board and use buttons to rotate them in 90-degree increments.

Given the time constraints, I plan to attempt an interactive isometric view:

- Each piece will have predefined orientations it can take (all rotations of that polycube). We can store these as sets of 3D coordinates for each piece.

- The puzzle space is a 3x3x3 grid. We need to ensure pieces do not overlap and stay within bounds.

- The UI might allow selecting a piece from a side panel, then using arrow keys or on-screen arrows to rotate it around X, Y, or Z axes to the desired orientation, then clicking on a cell of the 3D grid to place the piece (the placement would snap the piece's reference point to that cell, and the rest of the piece fills adjacent cells accordingly).

- We will highlight the cells as the piece is moved to indicate which grid cells it would occupy if placed, giving visual feedback to avoid illegal placements (like if part of the piece would go out of bounds or collide with an already placed piece, we can show that in red).

- The player can place all pieces one by one. Once all seven pieces are placed and if they perfectly fill the target shape (e.g., the 3x3x3 cube: all 27 unit cube positions filled), the puzzle is solved.

- We will provide controls to remove a piece or reposition it, so the user can backtrack. Also, a reset to start over if needed.

To simplify interactions, the first version might focus only on assembling the cube shape. This is already a substantial challenge. If time permits or as future enhancement, we can add alternate target shapes (like puzzles: “make a pyramid” or “make a sofa” etc.) with corresponding goal configurations. The game could then have levels or selection of which shape to make.

Features:

- **Interactive 3D piece manipulation:** including rotation and placement. Because dealing with 3D rotation via user input can be tricky, I will make the controls as simple as possible (e.g., fixed 90° rotations rather than free dragging in 3D space).

- **Visualization toggle:** possibly allow the user to toggle between a 3D view and a “layer” view to help them see filled spots. For example, a button that shows each 2D layer of the cube with pieces filled could help to understand the current state.

- **Snap guidance:** when a piece is selected, show an outline of the grid or available space to encourage correct positioning.

- **Completion check:** automatically detect when the target shape is completely filled with no overlaps. Then provide success feedback. If the target is the cube, maybe show a nice rendered cube and a message like “Congratulations, you assembled the cube!”

- **Educational info:** maybe an embedded note or fun fact about Soma cubes (like when it was invented, or how many solutions exist, etc.) to enrich the learning if the child or teacher is interested.

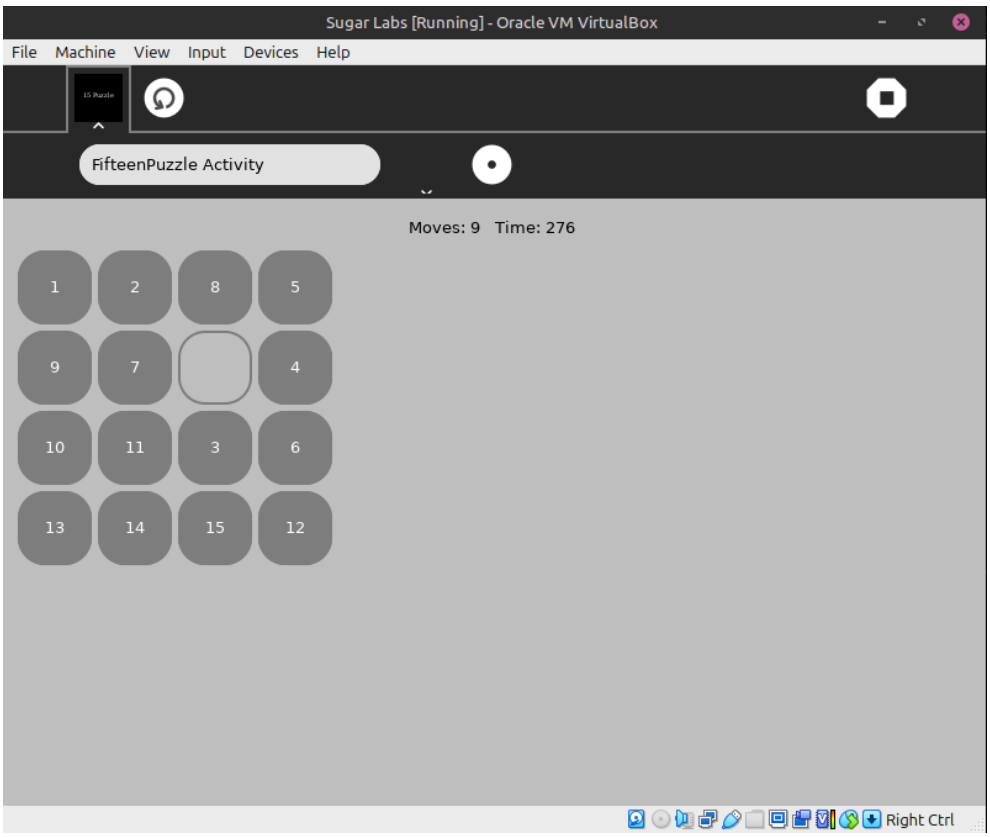
Tools Required: This activity will push the boundaries of typical Sugar 2D activities, since we need to simulate 3D. I will likely use a combination of drawing techniques. If PyGame or Pyglet (an OpenGL-based library) is allowed inside Sugar, I might leverage that for drawing cubes in perspective. If not, I can use PyGTK’s drawing area to manually draw an isometric cube using polygons for faces (which is doable with Cairo: drawing six-sided cubes by drawing squares at appropriate offsets). The logic for handling 3D rotations and collision can be done in pure Python with 3D coordinate math. No heavy external library is strictly required for the logic, but a small math utility could help (perhaps NumPy for matrix rotation math if available, or I can code the rotation matrices myself given the simplicity of 90-degree rotations). I will also require event handling for drag-and-drop or click

placement which the Sugar toolkit can provide. Managing state of a 3D grid (3x3x3 array) is straightforward in Python.

Educational Impact: Soma Cubes provide a rich spatial reasoning exercise. For many students, thinking in three dimensions is a new challenge since most schoolwork is flat or symbolic. By attempting to solve the Soma puzzle, children learn to mentally rotate objects and predict outcomes (**"If I rotate this L-shaped piece this way, will it fit in that corner?"**). This strengthens their ability to visualize, a skill useful in geometry, engineering, graphics, and everyday problem solving (like packing or fitting objects). Furthermore, because the puzzle doesn't have a single obvious solution path, it teaches **perseverance and systematic problem-solving**.

4. Fifteen Puzzle – *Sliding Tile Puzzle*

Prototype:- GitHub



Description & Educational Goal: The Fifteen Puzzle is a well-known classic: a 4x4 grid containing 15 numbered tiles and one empty space. The tiles are initially scrambled, and the goal is to slide the tiles one at a time into the empty space until the numbers are arranged in order (typically 1 through 15). This puzzle teaches **planning, sequencing, and problem-solving**. It helps improve a child's cognitive skills like short-term memory (remembering which moves they have tried), and pattern recognition (seeing which tiles need to go where). It also has a mathematical aspect in terms of permutations and parity (not that a child has to know that formally, but it's an underlying property that not all starting positions are solvable – which is a fun fact that can be shared). The goal for learners is to develop strategies (for instance, one common strategy is to place the numbers in order one row at a time) and to experience the satisfaction of completing a complex multi-step problem.

Implementation Plan & Features: The Fifteen Puzzle is relatively straightforward to implement in terms of logic. For the Sugar activity:

- We will represent the board as a 4x4 grid of positions. One position is empty at any time.
- The UI will display a grid (perhaps using a simple table layout or drawing the grid on a canvas) with 15 buttons or tiles that show numbers 1–15, and a blank space for the empty slot.
- The user can click or tap a tile adjacent to the empty space (above, below, left, or right of it) to slide it into the empty. In terms of implementation, when a tile is clicked, the program checks if it's next to the empty; if yes, we swap the tile and empty in the data structure and update the display (possibly with a sliding animation for polish, or at least an instantaneous move).
- Alternatively, the user could use arrow keys to move the empty space (which effectively moves a tile). But clicking tiles is more intuitive for touch or mouse usage, which fits Sugar's typical hardware (XO laptops have touchpad and some have touch screen).
- The puzzle needs a start state generator: We will generate a random solvable permutation of the numbers to start each game. There's a known method to ensure the initial shuffle is solvable by calculating the inversion parity; but an easier approach is to start from solved state and perform a series of random valid moves, say 100 moves, to jumble the puzzle. This guarantees solvability. I'll use that approach to avoid the complexity of parity calculation.
- The activity will have a "New puzzle" or "Shuffle" button to start a new scramble, and possibly a "Solve" button (mostly for demonstration or if a user is frustrated, it could auto-solve, but that might diminish the challenge – maybe more appropriate for teacher use or as a hint system).
- As the user makes moves, we might keep a move counter and a timer so they can see how many moves they've made and how long they've been at it – this adds a gentle gamification (trying to solve in fewer moves or faster time could be a personal goal).
- When the puzzle is solved (all tiles in order), the game will detect that (the data structure will match the goal state) and then celebrate (perhaps play a cheerful sound or animation and show "Puzzle Solved in X moves!").

Features:

- **Responsive tile movement:** Only legal moves should be allowed. Illegal moves (clicks on non-adjacent tiles) will simply not do anything, maybe with a brief blink to indicate "you can't move that".
- **Shuffle options:** Perhaps difficulty selection by shuffling more or fewer moves (though any solvable shuffle is essentially the same difficulty, except trivial ones with just a few moves).

- **Visual design:** The tiles can be simply numbered squares, but I might enhance it with colors or a picture. One variant of the 15-puzzle is using an image cut into 15 pieces. That might be an advanced feature: allow the player to switch from numbers to an image (which they then have to reassemble correctly). This can be more challenging since they must identify picture parts. However, for initial version, numbered tiles are clear and educational (number sequencing).

- **Undo (optional):** Typically, sliding puzzles don't need an undo because you can just move tiles back, but maybe providing an "undo last move" could be a user-friendly addition for younger players.

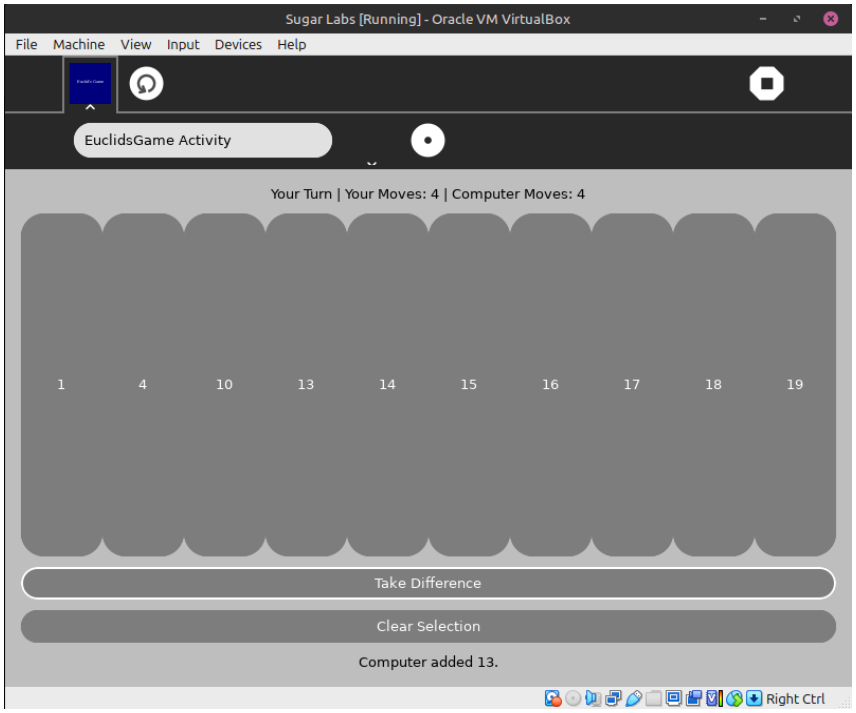
- Possibly incorporate Sugar's collaboration where two players can take turns moving tiles on the same puzzle over the network – this would be interesting but might be beyond scope initially. It could be a future improvement to allow a cooperative solve.

Tools Required: Standard Sugar GUI components will suffice. We can use a Grid or Table container to place 16 elements (15 labels or buttons and one blank). Or draw the grid in a canvas and handle click events with coordinate mapping. Either approach is fine. I'll likely use buttons because they come with click handling easily and can display text (the numbers). We might style them to be larger and square. No special libraries needed, this is pure Python UI logic. The random shuffle logic uses Python's random module or just pre-coded moves.

Educational Impact: The Fifteen Puzzle helps children improve their problem-solving strategy. It is not a trivial puzzle – it could take many moves and planning, especially if done methodically. Kids learn **Sequential reasoning, Pattern recognition and subgoals,Perseverance.** A teacher or mentor can draw out some math from it: for example, asking **"Do you think all scrambles can be solved? Why or why not?"** which can lead to discussions about parity and permutations – a high-level concept usually taught in higher math, but here it can be observed empirically (some arrangement might never occur from sliding moves if parity is wrong). Even if that's not explicitly taught, just solving the puzzle gives the brain a good workout.

5. Euclid's Game – *Two-Player Numerical Strategy Game*

Prototype:- GitHub



Description & Educational Goal: Euclid's Game is a turn-based game involving two numbers and the operation of taking differences. It is based on the Euclidean algorithm for finding the greatest common divisor (GCD). The typical rules (from the description and known versions cut-the-knot.org) are: start with two positive integers on the board. Players take turns; on a turn, a player can add to the board the positive difference of any two numbers currently on the board. Typically, we start with just the two initial numbers; then allowed moves generate new numbers. The game ends when no new number can be added (which happens when the only difference you could take is 0 or a number already present). At that point, one of the players cannot move and loses. In this variant described for the project, they added a twist for winning condition: "the player who made an even number of steps is declared the winner"cut-the-knot.org. This means instead of the usual "last player to move wins", here it specifically depends on the parity of the number of moves each made. The educational goal is to familiarize children with the concept of the Euclidean algorithm (repeated differences) and the notion of divisors/GCD in a playful way. It's also a pure strategy game where players must think ahead: it has a finite set of states and an underlying mathematical theory. It cultivates strategic thinking, planning, and a bit of mental arithmetic.

Implementation Plan & Features: In the Sugar activity, Euclid's Game will be presented such that a single player can play against the computer (AI opponent) or two players can play together (either two kids passing the device or possibly via network if feasible). Key elements of the implementation:

- **Game State:** We maintain a list (or set) of numbers that are currently "on the board". Initially, this list has two random integers. We can choose a range for these initial numbers, say 1 to 50, ensuring they are not equal (if they were equal, the game ends immediately since their difference is 0 and no move can be made). The difficulty could be influenced by the size of these numbers (larger numbers mean the game can last longer and might be slightly more complex strategy).

- On each turn, the current player must choose two distinct numbers from the board, compute their positive difference, and if that difference is not already on the board (and is >0), add it to the board. In practice, the interface can simplify this: the player can pick one number from the board and subtract it from another. For instance, if the board shows [8, 20], a player could select 20 and 8 and click a “subtract” button, resulting in 12 being added (20-8=12).

- We must track move count for each player. Let’s say we have a counter for moves: player (user) moves count and computer moves count.

- The game continues until no move is possible. When will that be? In the Euclidean algorithm, when numbers reduce to multiples of the GCD, eventually the GCD itself appears and then all new differences are multiples of that GCD. Eventually, the set of numbers stabilizes (when the smallest number on the board is the GCD, subtracting it from any multiple yields either the same or another multiple present). Practically, the game ends when the smallest number on the board is the GCD of the initial two (and now everything on board is a multiple of that) such that any difference you take is already present or zero. The losing condition for a player is “cannot make a move on your turn”.

- However, since they defined winner as the one with even number of moves, we’ll adhere to that: after the game ends, count each player’s moves and determine who has even. (It could be that both have even, or both odd in rare cases – but they indicated one is winner presumably always, and they mentioned to avoid draws the number of available steps is always odd, which might ensure asymmetry).

- **Computer Opponent (AI):** We need to implement a strategy for the computer. Ideally, we want the computer to be reasonably challenging. There is known mathematical theory: typically, the player who can force making the number of moves even or odd might have a winning strategy depending on N/gcd(N,M) cut-the-knot.org. However, for an educational game it might be sufficient that the computer makes legal moves and perhaps semi-intelligent ones (like not immediately losing if a winning move is obvious).

    - A simple approach: have the computer mimic the Euclidean algorithm (always take the difference of the two largest numbers, which is a greedy approach). This will quickly drive the game to completion. But we have to be careful – as a game, the opponent might want to influence parity of moves. A more sophisticated approach could analyze if the position is winning or losing (combinatorial game theory can be applied, but given time, I might not fully solve that).

    - At minimum, the computer will never skip a move (since it cannot) and will choose a difference that is >0 and not already present. We can randomize which difference it takes to vary the gameplay.

    - If possible, I will incorporate a known strategy: Usually in Euclid’s algorithm game, it might be optimal to take the largest possible difference (which reduces one number mod the other). I can implement that as a heuristic: e.g., if numbers [a, b, …] with a<b, then consider b mod a (if not zero) or some multiple. But given the parity win condition, the strategy might shift slightly.

    - For now, I'll plan: **computer move generation**: look at all possible new differences it could add, and choose one that leads to a position favorable to it (if it can evaluate a move or two ahead). If that's too complex, at least avoid obviously bad moves (like adding a number that immediately allows the human to win next turn).

- **User Interface:** The screen will display the current set of numbers, perhaps in a row or a grid. On the user's turn, they need to create a new number by difference. We can allow them to input the result directly ("Enter a number to add") but it's better to ensure they follow the rule of picking two numbers. So the UI might be:

    - Click first number (it highlights), click second number (highlights), then click "Take difference" button. The activity then computes the difference and, if valid, adds it.

    - Alternatively, have a small form: a dropdown or two selection boxes for "Select two numbers:" and then a button.

    - We should show whose turn it is ("Your turn" or "Computer's turn…") to avoid confusion.

    - When it's the computer's turn, maybe briefly show what move the computer chose (e.g., "Computer adds |8-5| = 3 to the board.").

    - The board updates dynamically with the new number.

    - Turn alternates and this continues.

- **Game End and Result:** When no moves are possible (which we detect when the smallest number d on board divides all other numbers and hence any difference is a multiple of d which is already on board), the game will end. We then count the moves each player made (we have those counters) and declare the winner: "Game over. You made 5 moves (odd), Computer made 4 moves (even). Computer wins because it made an even number of moves." Or vice versa. If playing two humans, it would say "Player 1 vs Player 2" with their move counts.

- Possibly incorporate a **"Play Again"** option on game over to start a fresh round with new starting numbers.

- We could allow the user to customize the starting numbers (enter their own) for exploration, but random is fine for general play.

Tools Required: Pure Python for logic. The UI can be built with basic widgets: labels for numbers that can be clicked (I might use buttons with number labels here too, for easy click handling). Or just display them as text and let the user type a result – but I prefer the click method to ensure the rule is followed. We will also use standard control flow to manage turns. For the computer logic, no extra library is needed – it's simple arithmetic.

Educational Impact: Euclid's Game sneaks in a fundamental mathematical algorithm (the Euclidean algorithm for GCD) under the guise of a game. While playing, children are effectively performing repeated subtraction, which is one way to find the greatest common divisor.

6. Odd Scoring – *Parity-based Move Game*

Description & Educational Goal: Odd Scoring is another two-player game (user vs computer) described as follows: imagine a chip or token placed on a linear track of N cells. The chip starts on one end of the track (say the far right cell of N). Players alternate turns moving this chip to the left by 1, 2, or 3 cells. So on any move, the token advances towards the left end by one to three steps. The game ends when the chip reaches the last cell (the leftmost cell). At that point, the total number of moves made by each player is tallied, and **the player who made an even number of moves wins**. The rule "to avoid a draw, the number of available steps is always odd" is essentially in the game's design (allowed steps are 1,2,3 which is an odd count of options, likely ensuring the final parity situation is decisive). Educationally, Odd Scoring teaches children about parity (even and odd numbers) in a strategic context. It's a simplified impartial game that requires foresight: players must plan not just how to be the one to land on the last cell, but how to ensure the parity of their move count is in their favor. It's an exercise in logical planning and also gives practice in basic addition/subtraction as they consider how far the chip moves and how many moves remain.

Implementation Plan & Features: For the Sugar activity:

- We will represent the track of N cells visually, perhaps as a horizontal row of boxes or circles. We can choose a default N (for example, N = 20 cells) or potentially allow the user to pick a length for a new game. The last cell (cell #1) could be specially marked (like a finish line).

- A piece (chip) is shown at the starting cell (cell #N). This chip will move leftwards as turns progress. We can label cells by their number or just show positions.

- On a player's turn, they have up to three move options: move 1, move 2, or move 3 cells forward (to the left). We will provide three buttons or clickable options for these moves. If the chip is very near the end (say at cell 3), and you have the option to move 3 which exactly lands at cell 0 (beyond the last), we will treat reaching the last cell as the end. Actually, likely the rule is you cannot move beyond the last cell – you must land exactly on or before it. We can clarify that as: if the chip is at position k (counting from left), moves available are min(3, k-1) because you can't go past cell 1. So at cell 3, you can move 1,2 or exactly 2 to land on cell 1? Actually if at cell 3, a 2-step move lands at cell 1 (finish), a 3-step would try to land at cell 0 which is out of bounds, so disallowed).

- The game flow: user always moves first (as described in the problem). So initial turn: user chooses to move the chip by 1, 2, or 3 steps (depending on what they think is a good strategy). Then the chip's position updates (we animate it sliding left or simply jump it and update a position indicator).

- Then it's the computer's turn. The computer will also choose 1,2, or 3 to move the chip further left.

- This alternation continues until the chip reaches cell 1 (the last cell). At that moment, we stop and count how many moves each player took.

- Determine winner: The one with an even number of moves wins. Note that if the total number of moves is odd, one player will have one more move than the other, meaning one has odd count, the other even, so there's a clear winner. If total moves is even, both players made the same number of moves (since turns alternate, an even total implies each had half which is an integer; half of an even is either even or odd? Example: total 6 moves, each got 3 moves, which is odd for both – that would mean both odd, which creates a contradiction because then who wins? The rule "to avoid a draw, number of available steps is always odd" might imply that scenario can't happen if both play optimally or due to initial conditions. But to handle robustly, if it does happen that both made odd moves, perhaps by the rule of the game one might consider that scenario not possible or maybe then the last mover loses by default. However, since the description emphasizes even wins, I'll assume the design is such that one will have even.)

- We'll implement according to the rule: after final move, compare parity: if user_moves % 2 == 0 and computer_moves % 2 != 0, user wins; if computer_moves is even and user's is odd, computer wins. (If by chance both are even or both odd, we might declare a tie or something, but likely it won't happen due to game length parity).

- **Computer Strategy:** This game is reminiscent of a Nim-like game where moves are limited. Actually, ignoring the parity win condition, if it were a normal game where last move wins, it's like a take-away game where you can take 1-3 tokens, which usually has a simple winning strategy (positions that are multiples of 4 are losing in that normal mode). However, because the win is based on parity of moves, the strategy is a bit different. It's like a misère version of the 1-3 game combined with parity.

    - We can attempt to compute winning positions: We can define a state by (position of chip, whose turn, parity of moves so far perhaps). This could be solved via dynamic programming/backtracking to decide what moves ensure your parity advantage. However, given time, I might implement a simpler heuristic: the computer can try to ensure the total number of moves left (including its own) results in it winning. For example, if the chip is very close to end, the computer can foresee how many moves remain.

    - At least, I can implement a basic strategy: if a move leads to immediate win (i.e., the game ends and computer's moves are even), do that. If a move leads to a situation that is known to be losing, avoid it. For a quick implementation, I might not formalize it fully but test small scenarios manually and encode something. Alternatively, since N is small (say 20), I could brute force minimax with states (which is feasible since branching factor 3 and depth at most ~ N/1 = N moves).

    - Given our timeline, a simpler approach: ensure the computer doesn't obviously let the human win. Perhaps always moving such that the remaining distance mod 4 is not 0 might be a naive trick (since if it leaves a multiple of 4, the other player might force parity). But with parity win, not sure if mod4 logic holds.

    - At minimum, the computer will make a legal move. If we can't do a deep strategy, we might make it randomly choose 1-3 unless that would finish the game. The educational value doesn't rely on a

super strong AI; even playing against a moderate one, the child still has to think parity.

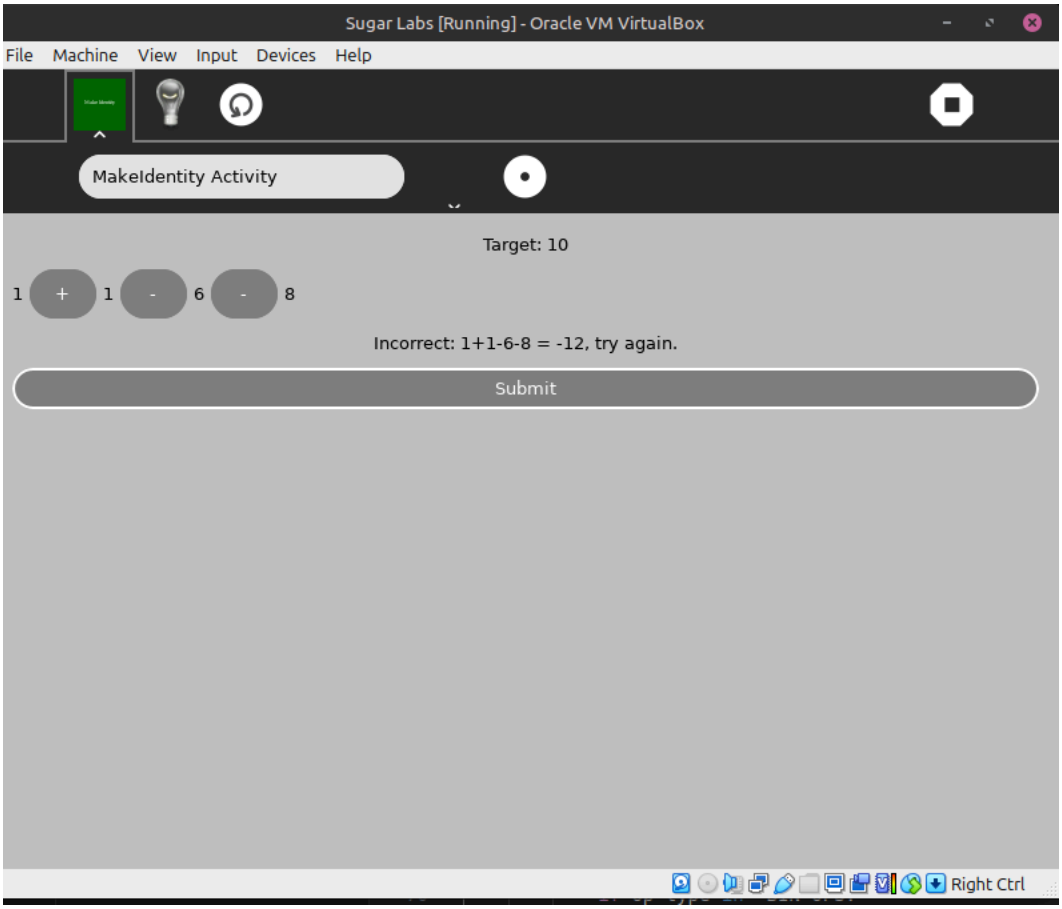
- **User Interface:**
    - Show the track and chip. Possibly label the chip's current cell number to know how far from the end.
    - Provide buttons: "Move 1", "Move 2", "Move 3". Only enable those that are possible (for instance, if at cell 2 (second from left), only a move of 1 or 2 is allowed to finish the game).
    - Indicate turns: maybe highlight the player's side when it's their turn and grey out the move buttons during the computer's turn while it "thinks" (which will be a very short delay).
    - Count moves: we can display a counter of moves for each side during play, or at least at the end present the counts.
    - Option for new game after finish.

Tools Required: Basic drawing for the track (could just be a horizontal row of labels or icons). I might use a simple icon for the chip (perhaps a small image or colored circle). The rest is standard GUI widgets (buttons for moves). The logic is straightforward arithmetic. If I do decide to implement a smarter AI through search, that might involve a recursive function or loop to evaluate states, still just Python logic.

Educational Impact: Odd Scoring is a great way to make the abstract concept of even vs odd tangible and strategic.

7. Make an Identity – *Arithmetic Puzzle (Target Equation)*

Prototype:- GitHub



Description & Educational Goal: "Make an Identity" is a puzzle where you are given a sequence of digits and a target number, and you must insert arithmetic operations (and possibly parentheses) between the digits to form an expression that evaluates to the target cut-the-knot.org. Essentially, it transforms a sequence of digits into a valid equation equalling the target. For example, given digits `8 5 6 1 4` and target 10, one solution is `8 * 5 - 6 * (1 + 4) = 10` cut-the-knot.org. Each puzzle has a set of digits (which must remain in the given order) and the player can choose to put plus, minus, multiply, or divide signs between them (or concatenate digits together by placing no operator, forming multi-digit numbers) and add at most a pair of parentheses if allowed, to reach the target number. This activity's educational goal is to provide **arithmetic practice and algebraic thinking**. It's like solving a math puzzle where the child is effectively doing mental math and exploring how different operations interact. It reinforces the concept that you can represent the same number in many ways, and it gets students comfortable with the order of operations and grouping (parentheses).

Implementation Plan & Features: The activity will operate in a puzzle format with potentially multiple levels of difficulty:

- I will prepare a set of puzzles with predetermined digits and target (taken from known examples or generated randomly with a known solution). For instance, an easy puzzle might be `Digits: 1 1 6 8, Target: 10` (like the example in the cut-the-knot text, solution `1 + 1 + 16 - 8 = 10`. Harder puzzles might have more digits or allow more operations.

- The UI will display the sequence of digits in order, maybe spaced out with gaps between them where an operator can be placed. The target number will be shown separately (e.g., "[10] : 8 5 6 1 4" as in the example format cut-the-knot.org, meaning target 10 and digits 8 5 6 1 4).

- The user can insert an operation symbol in each gap between digits. Possibly, we also allow no symbol which implies concatenation of digits (for example, between 1 and 6 if no symbol, that makes "16").

- We should allow at most one pair of parentheses or maybe multiple, depending on difficulty. The original description said "sometimes a pair of parentheses" – to manage complexity, I might implement a mode where parentheses are either not allowed (all operations left to right but treat * and / with usual precedence) for simpler puzzles, and a mode where one set of parentheses can be placed around a part of the expression.

- UI approach: The player could click on the gap between two digits to cycle through options: [blank (meaning concat) / + / − / × / ÷ / (maybe an open parenthesis at this point?)] – parentheses are trickier because they span across the expression. Alternatively, we can have a more explicit control: allow the user to tap an "insert (" before a certain digit and an "insert)" after a certain digit. To keep it simpler, maybe we restrict to one pair that covers some prefix of the expression.

- To start, I may implement without parentheses, focusing on placing operations, which is already plenty challenging. Then consider parentheses if time allows.

- There will be a "Check" or "Submit" button to evaluate the formed expression and compare it to the target.

- Internally, after the player places all desired operations, we construct the expression string (taking into account concatenation where no operator was placed, and standard precedence for * and /) and evaluate it. We must be careful to follow arithmetic rules: typically multiplication and division have higher precedence than addition and subtraction, so the evaluation isn't purely left-to-right unless we add parentheses accordingly.

- If the expression evaluates to the target, and uses only the given digits in order, the puzzle is solved – we show a success message ("Correct! 8*5 - 6*(1+4) = 10" perhaps echoing their solution).

- If it doesn't match the target, we can either just say "Not correct, try again" or provide partial feedback. Partial feedback is hard (because many combos to try), but maybe we can highlight if they overshot or undershot the target, or just encourage another attempt.

- We can include a "Hint" button that gives a subtle hint like "Try using multiplication" or "Try concatenating some digits" if the player is stuck. Or more directly, maybe offer to fill one operator correctly.

- We might include a "Solve" button for the teacher or for learning, which reveals one possible solution (since often these puzzles have multiple solutions, we can have a precomputed one). But typically, puzzle

games encourage the player to find it themselves, maybe only revealing the answer if requested.

- **Multiple puzzles / progression:** There could be a menu or next button to go to the next puzzle after solving one. We might categorize: puzzles with 4 digits and plus/minus only (easy), puzzles with 5-6 digits and all operations (medium), puzzles that also allow a pair of parentheses (hard). The activity could present them in increasing difficulty.

- It would be nice to allow users to input their own puzzle (digits + target) for a friend to solve, but that's extra; likely we stick to fixed puzzles for now.

Features:

- **Interactive equation builder:** The main feature is the interactive placement of operations between digits. We'll make it user-friendly by perhaps letting them tap through options, or drag an operator symbol into place.

- **Expression validation:** The app will need to ensure the expression is valid (e.g., not starting with an operator, not ending with an operator, no division by zero – although division by zero could occur if, say, they put a ÷ before a 0 digit, so we need to catch that).

- **Automatic formatting:** If no operator is placed, digits combine. We should display combined digits as a single number once chosen, so the player can see the effect (like if they remove the operator between 1 and 4, it should visually become "14" instead of "1 4" separate).

- **Feedback messages:** Provide clear messages for correct/incorrect attempts. Possibly count how many tries they took (just for interest, not necessarily scoring).

- **Reset puzzle:** Let them clear all operators to start fresh if needed.

Tools Required: Standard Python string handling and arithmetic. Possibly use the Python `eval` to evaluate the expression string (again carefully, ensuring it's safe – since the content is only digits and ops, it should be). Alternatively, write a small evaluator that goes through the digits and operations (this might be straightforward given the small scale). For UI, labels for digits and clickable areas for operations can be implemented using Sugar's toolkit. We might represent each gap as a small button that cycles through choices when clicked.

Educational Impact: Make an Identity puzzles are excellent for building strong arithmetic skills and introducing algebraic thinking like **Order of Operations, Mental Math and Estimation.**

8. Number Detective (AI-based) – *Sequence Pattern Learning Game*

Description & Educational Goal: Number Detective is an innovative game that combines math sequences with a dash of machine learning. The concept is that the computer (the "AI detective") tries to figure out the pattern in a sequence of numbers and predict the next number. The child either confirms the AI's guess if correct or corrects it if wrong. Over time, the AI "learns" from these corrections. The educational goals are two-fold:

1. Teach pattern recognition and sequence prediction skills to the child. By entering their own sequences or seeing provided sequences, children exercise understanding of how sequences can be formed (arithmetic progression, geometric progression, simple functional patterns like squares, factorial, or even non-math patterns like alternating).

2. Introduce the basics of how AI/machine learning works: the AI improves through feedback. It demystifies AI by showing that the computer doesn't always know the answer and needs to be taught, thereby highlighting the role of data/training in AI.

Implementation Plan & Features:

- **User Input/Interaction:** The game will prompt the user to enter a sequence of numbers. This could be done by typing numbers separated by commas or spaces into an input field. Alternatively, the game can present a random incomplete sequence and ask the user to guess the next (but since the idea specifically says user inputs and AI predicts, we'll go with the user-driven approach).

- Once a sequence is provided (for example: "2, 4, 6, 8"), the AI will output its predicted next number (for "2,4,6,8" it might guess "10" assuming it sees a pattern of +2). This prediction is shown on screen like "AI guesses the next number is 10."

- The user then either confirms that the guess is correct or says it's wrong and provides the correct next number. In our example, if the sequence was indeed evens, 10 is correct, the user would confirm and maybe move on to another sequence. If the AI had guessed wrong, or if the user had something else in mind (say the sequence was actually "powers of 2 truncated: 2,4,6,8" and maybe next was 10 which is still right actually in that case if pattern unclear – better example: sequence "2,4,6,9" which doesn't have a simple linear pattern, AI might guess 11, user says no the next is 12).

- **Learning Mechanism:** The key part is how the AI learns from the correction. We'll implement a simple rule-based learning:

    - Initially, the AI will have a set of basic patterns it can check: e.g., arithmetic progression (constant difference), geometric progression (constant ratio), maybe quadratic sequence (constant second difference), or alternating pattern (two sequences interleaved), etc. It will apply these simple rules to try to predict. If one fits perfectly with the given numbers, it uses that to extrapolate. If multiple fit, it might choose one or say it's unsure (but to keep it simple, maybe check in order: first try arithmetic, then geometric, etc.).

    - If the AI's guess is wrong and the user provides the correct answer, we treat that as a new piece of knowledge. One approach: store the entire sequence (with the correct next) as a known pattern case. Next time the AI sees the same initial sequence it could recall the answer. But sequences might not repeat exactly.

    - To actually “learn” a pattern, we might try to deduce what the pattern was based on the correct next term. For example, if the user’s sequence was 2,4,6,9 and they said the next is 13 (just hypothetical), that sequence doesn’t match a simple arithmetic or geometric. But maybe the user had a rule like +2, +2, +3, +4,... which is not obvious. The AI at our scope might not truly infer that complex rule just from one sequence.

    - A more feasible approach: incorporate a simple machine learning model – for instance, treat this as a sequence prediction problem and train on the fly. But with only one sequence example given at a time, training a general model is hard.

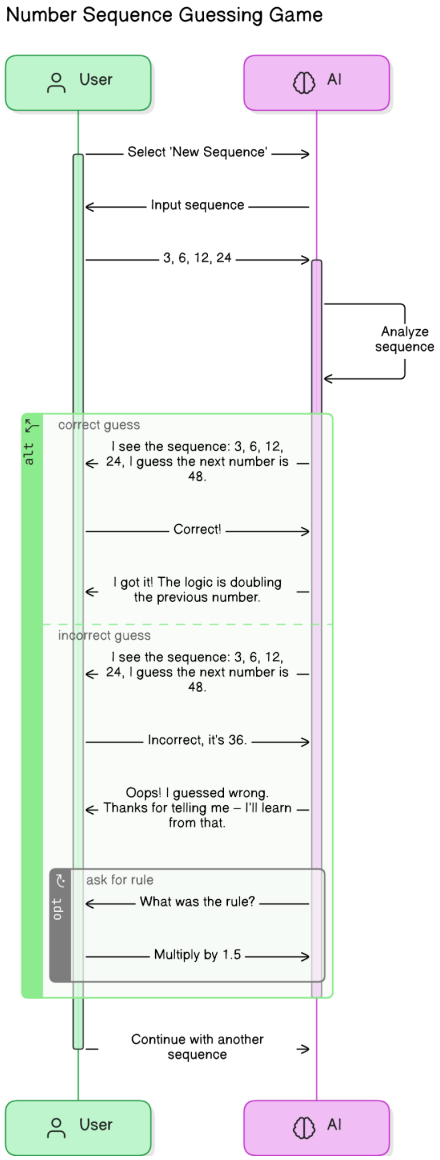
    - Instead, perhaps after a wrong guess, the AI can present a message like “I’ll remember this pattern.” We add the sequence and answer to a small database. Later, if the user inputs a sequence that matches one from the database (or a prefix of one), the AI can recall the learned continuation.

    - The improvement over time might be subtle: if a user tends to test the AI on the same few types of sequences, it will eventually get them right by memory.

    - Another angle is interactive learning: the game could allow the user to input a longer sequence (not just up to the missing last, but maybe they can keep extending a sequence, correcting the AI multiple times, thereby training it gradually on that pattern).

- Perhaps a simpler storyline approach: The AI could start very naive and have a limited set of pattern recognitions (like only arithmetic). The user is encouraged to “train” it by giving different sequences. Over sessions, it accumulates knowledge (persist in Sugar’s Journal or activity data). This way, children see that initially the AI fails often, but as they correct it, next time those sequences or similar ones it might get right, giving a sense of progress.

- **User Experience Flow:** Possibly:



- We might include some preset sequences to try (like a "Try one of these sequences" button) in case the user doesn't know what to input. For example, have a small list of sample patterns like [1,2,3,...], [2,4,6,8,...], [1,1,2,3,5,...] (Fibonacci), [1,2,4,8,...], [1,3,6,10,...] (triangular numbers), etc. The child can click one and see how the AI does. This can be a guided way to showcase the AI's learning – maybe initially it fails on Fibonacci, and after the user teaches it, it gets it next time.

- **AI Internals and Algorithms:** As suggested, we can use **rule-based logic plus a simple learning component**:

    - Rule-based: check if the sequence is arithmetic (compute difference between all consecutive terms, if all equal then predict last + diff). Check geometric (ratio common and integer maybe, predict last * ratio). Check if it's increasing by an increasing amount (like 2nd difference constant – then it's a polynomial of degree 2, we could extrapolate using second difference if it's consistent). If none match, perhaps default guess could be last difference again (like assume next

difference same as previous).

  - These simple guesses will sometimes be right for common sequences.

  - Machine learning aspect: Possibly use a minimal machine learning approach like treating it as regression or classification. For instance, use a decision tree that takes as input some features of the sequence and outputs next number. But designing features for a general sequence might be complicated. Alternatively, use k-Nearest Neighbors: treat each known sequence in memory as a point in some space (like the sequence itself), and if a new sequence is similar (maybe identical first few terms) to a known one, guess similarly. Essentially, memory-based.

  - Since the project idea specifically mentions machine learning, I might demonstrate something like a decision tree which is trained on the sequences that the user has corrected. But each sequence is one training sample (mapping from first N numbers to N+1th number). Over time, if the user provides enough examples, a pattern might emerge. This might be overkill; a simpler approach with stored examples might suffice for the scope of usage by one child (they're not going to systematically train it on thousands of sequences).

  - I will mention that the AI could be implemented with either a custom simple ML or using a library like scikit-learn to create a decision tree regressor or classifier. However, using such a library might be heavy and not necessary; we can illustrate the concept with minimal infrastructure.

- **Persistence:** The game should save the "learned" sequences or model to the Sugar Journal or activity data, so that if the child comes back later, the AI remembers what it learned previously. This emphasizes the idea of learning over time.

Features:

- **Free-form input:** user can input any sequence, which makes it open-ended and fun (they can try to stump the AI).

- **AI guess display:** clearly show what the AI predicts, maybe along with a thinking message like "Hmm... I think the next is X" to personify the AI a bit.

- **Feedback input:** a way for user to say correct or enter correct answer. Possibly two buttons: "Correct" or "No, actually it is: [input]".

- **AI improvement feedback:** after a correction, maybe display something like "I'll remember that pattern!" or even show that now if you give the same sequence again it will answer correctly (we could test it immediately: AI could ask, "Let me try again: sequence was ..., next number is ...").

- **Multiple rounds:** allow user to do multiple sequences in one session and track how many the AI got right vs wrong (just for information).

- **Reset learning:** perhaps a button to reset the AI's memory to see the effect of starting fresh or for a new user.

Tools Required: Just Python logic for sequence analysis. Possibly data structures to store learned patterns. If using a library for ML, scikit-learn's decision tree or a simple neural network could be considered, but likely not needed due to environment constraints and complexity. Instead, a simple decision tree can be coded because sequence patterns are small. But given time, memory-based might be enough: store sequences as dictionary keys or so. The UI will require text input fields (Sugar has entry widgets) and labels for output, plus some buttons.

Educational Impact: Number Detective is quite innovative in that the child is interacting with an AI as both a user and a teacher.

9. Sorting Hat AI – *Interactive Classification Game*

Description & Educational Goal: Sorting Hat AI is a game where the child teaches an AI to classify objects into categories, reminiscent of how one might train a model to distinguish between classes. The name is inspired by the "Sorting Hat" from Harry Potter (which decides categories for students) – here the AI plays the role of the sorting hat that tries to classify items, and the child is both a participant and a trainer. The game can involve various types of items, such as pictures of animals or shapes, or just numbers, and the categories could be up to the user or predefined. For example, the child might classify animals into "mammals" vs "reptiles" or shapes into "has straight edges" vs "no straight edges". The AI will observe these labeled examples and then attempt to classify new, unlabeled items. If it makes a mistake, the child corrects it, thereby further training the AI. Educationally, this activity demonstrates how classification works in AI (specifically supervised learning: learning from examples) and gives insight into attributes and grouping. It also teaches the child to observe features of objects and decide criteria for grouping, which is a fundamental cognitive skill in math and science (think sorting shapes, grouping numbers, taxonomy in biology, etc.).

Implementation Plan & Features:

- **Datasets/Modes:** We will include a couple of domains of items to classify:

    - **Shapes:** e.g., a collection of simple geometric shapes (circle, triangle, square, pentagon, etc., possibly with different colors or sizes for extra variety). Categories could be shape-based (triangles vs not triangles, or polygons vs non-polygons) or color-based or something the user chooses.

    - **Animals:** perhaps cartoon icons or names of animals with known attributes (bird, fish, cat, lizard, etc.). Categories could be mammals vs non-mammals, or land vs water, etc.

    - **Numbers:** simply numbers (maybe 1 to 20 or so). Categories could be even vs odd, prime vs composite, etc.

    - We might let the user choose which dataset to play with, or randomly rotate among them to show versatility of AI.

- **User Labeling Phase:** At the start, the game will present some items and ask the user to label them into two groups (to simplify, we can assume binary classification at first). The user essentially is creating the training set. For example, in shapes mode, we show a shape and ask the user to assign it a label from two options (the options might be user-defined or preset; maybe we allow the user to type category names, e.g. “Category A” and “Category B” or select a property like “shape with curves” vs “shape with only straight lines” if guided).

    - We should make it flexible. Perhaps initially, we ask the user “Pick how you want to classify these shapes. For example, you could sort by ‘Has 3 sides’ vs ‘Doesn’t have 3 sides’. It’s up to you!” The child might then start labeling, effectively deciding criteria in their head.

    - Alternatively, to ensure there is a learnable pattern, we might provide a scenario: “Let’s train the AI to know which animals are mammals and which are not. Please label these animals.”

    - Maybe best is to provide default categories for simplicity (so the AI can have pre-coded features relative to that category), but allow free labeling if the user wants to be creative (with the understanding that if categories are completely random, the AI might not find any pattern).

- **Feature extraction:** Under the hood, for the chosen domain, we need to represent each item with features that an algorithm can use:

    - For shapes: features could be number of sides, number of curves, color (if relevant), etc.

    - For animals: features could be has fur, has feathers, lays eggs, lives in water, etc. We might internally have a small table for each animal. For example, “Cat: {fur: yes, feathers: no, aquatic: no, legs:4, vertebrate: yes, etc.}”, “Eagle: {fur: no, feathers: yes, aquatic: no, legs:2, vertebrate: yes, etc.}”, etc. These attributes allow distinguishing categories like mammal vs bird vs fish, etc.

    - For numbers: features can be numeric properties: even/odd, prime/composite, >10 or <=10, etc. But since number classification might often be straightforward (if user picks even vs odd, there’s one clear feature mod2; if prime vs composite, that’s also definable).

- **Training the AI (Algorithm):** We will use a simple **Decision Tree or k-Nearest Neighbors (k-NN)** algorithm as suggested github.com:

    - **Decision Tree approach:** As the user labels examples, we can build a decision tree that attempts to split the data by features to classify it. For example, a decision tree could learn a rule like “If an

animal has fur, classify as Mammal; if not, classify as Not Mammal" from a few examples.

- We could implement a simple ID3 algorithm for a binary tree given the small scale, or even brute-force find a single feature that best separates the provided labels (which is essentially one level decision stump, might suffice if one feature perfectly correlates).
- **k-NN approach:** We can store all labeled examples (with their feature vectors). When the AI needs to classify a new item, it looks at the k (say k=3) nearest neighbors in feature space among the labeled ones and does a majority vote on their labels. "Distance" can be simple (like Hamming distance on boolean features or difference on numeric).
- k-NN is simpler to implement incremental (just store data) and can handle arbitrary patterns given enough data, but might misclassify if not enough close examples.
- Decision tree, once built, can generalize from few examples (especially if one feature is a clear divider).
- Possibly, we can start with k-NN for simplicity of coding and because it handles whatever weird category the user chooses as long as features support it. Or do a combination: if dataset small, the difference is minor.

- **Gameplay Loop:**
  - **Labeling Phase:** The user labels a certain number of items (maybe 5-10) to train the AI. We should show one item at a time with an interface to choose Category A or B.
  - **Classification Phase:** Now the AI tries to classify a new item that the user has not labeled yet. We present the new item and the AI outputs which category it thinks it belongs to. E.g., "AI sees a Duck and predicts: Category B (Not mammal)."
  - If the AI is correct (according to the user's intended classification), the user can confirm. If the AI is wrong, the user should correct it by providing the right label. That new example (item with correct label) is then added to the training set, and the AI model updates (the decision tree is rebuilt or the k-NN dataset grows).
  - We continue this way with more items. Essentially, every time the AI might guess until all items are classified or until a certain number of rounds.
  - We can then perhaps show a summary: the AI accuracy improved, or eventually it classifies all remaining items correctly.
  - Optionally, allow the user to start a new classification round with a different category or dataset.

- **User Interface:**
    - Should display the item (if it's an image of shape or animal, show the image; if it's a number, show the number).
    - In labeling phase: Show two category labels (either user-defined text or default ones) as buttons to assign to that item.
    - In guessing phase: Display the item and a statement like "AI's guess: [CategoryName]" perhaps with an icon or highlight on that category. Then provide buttons: "Correct" or "Wrong, it should be [Other Category]".
    - Also perhaps show count of how many it got right so far or how many items are left.
    - If category names are user-defined, use those; if not, generic A/B or preset like "Mammal/Not Mammal" depending on scenario.
- **Example Walkthrough (for clarity):** Suppose we choose the animals dataset, focusing on "Mammals vs Others".
    - The activity might come with that scenario ready. The user is shown "Cat" – they click "Mammal". Shown "Dog" – click "Mammal". Shown "Snake" – click "Not Mammal". Shown "Parrot" – click "Not Mammal". Now we have 4 training examples.
    - Now the AI classification phase begins: show "Elephant" – AI looks at features (Elephant has fur? no hair but maybe considered mammal by other features like warm-blooded – our features might include "gives birth to young" etc if we made it complex; but even simpler, maybe just "has fur or not" which Elephant ironically might not have obvious fur but is mammal. Let's assume our features included something like "animal type: mammal/bird/reptile" behind scenes so that Elephant would still classify as mammal). The AI likely guesses "Mammal" – user says Correct.
    - Next "Whale" – maybe features show aquatic, no fur but mammal class, AI might guess Not Mammal if it relies on "has fur". If it guesses wrong, user corrects as Mammal. Now that data point (whale) helps the model perhaps consider other features or at least memorize whale is mammal.
    - Next "Lizard" – AI guesses Not Mammal (hopefully correct), user confirms.
    - And so on. Over time, the AI might make fewer mistakes as it has more examples (especially if we implemented a decent algorithm).
    - At the end, the user sees that the AI learned their classification scheme.

Tools Required: If using decision tree or k-NN, we might implement from scratch due to simplicity. For decision tree, we might use information gain to pick the best feature splits if we want, but with only binary classification and a small feature set, even greedy works. For k-NN, just need a way to compute distance. Data representation as simple lists or dicts of features.

If including images (for shapes/animals), we will need those image assets packaged with the activity. If not, we can use textual description like “Cat”, “Dog” labels or simple SVG drawings for shapes. Possibly simpler: start with shapes and numbers which we can draw dynamically (shapes can be drawn with Python turtle or Cairo easily: draw a triangle, square, etc.). For animals, might need clipart which might be heavy; maybe just use text (though an image would be more engaging for kids).

But given this is a proposal, we can say we’ll include a small set of images or icons for clarity and appeal.

Educational Impact: Sorting Hat AI has several educational benefits like **Introduction to AI classification, Analytical thinking about attributes, Understanding of categories and sets.**

In summary, Sorting Hat AI offers a playful environment for understanding classification – a key concept in math (sets, logic) and computer science – while also touching on real-world AI principles. It’s interactive and open-ended, which can adapt to different education levels: younger kids might do simple visual categories, older kids might attempt more complex ones (like sorting numbers by prime vs composite, which requires them to think about what makes a number prime – an indirect way to learn that concept).

Delivering all these activities will involve careful design to ensure they are intuitive and fun. Each activity’s description, rules, and interface will be made child-friendly (with clear instructions likely accessible via Sugar’s help mechanism or an introductory screen). I will also incorporate feedback from educators and Sugar Labs mentors during development to fine-tune the educational aspects. By the end of the project, the deliverables will be eight well-crafted Sugar activities that can be easily installed and used in Sugar, significantly expanding the platform’s offerings in mathematics and computational thinking.

Proposed Timeline:

The project is planned for the GSoC 2025 timeline (350-hour project). Below is a week-by-week breakdown, including the community bonding period, planning, development milestones for each activity, testing phases, integration of AI components, and buffer time for polishing and documentation. This timeline is tentative and may be adjusted in consultation with mentors, but it demonstrates a clear plan for steady progress. I will use an iterative approach: implement basic versions early, then refine and add features, ensuring that by mid-term evaluations a significant portion is functional, and leaving adequate time for testing and improvements towards the end.

Timeline	Activity	Key Tasks / Deliverables
Community Bonding *(May 8 – May 28)*	**Community Engagement & Setup**	☑ ~~- Set up development environment (Sugar desktop, sugar-toolkit, libraries)~~ ☑ ~~- Familiarize with existing Sugar activities~~ - Finalize activity specifications with mentor input - Create rough UI sketches - Collect/create assets and plan test strategy
Week 1 *(May 29 – June 4)*	**Four Color Map Game – Design & Prototype**	- Develop basic UI layout - Implement region coloring with simple sample map - Add color palette and check logic - Manual testing and mentor feedback
Week 2 *(June 5 – June 11)*	**Four Color Map Game – Completion & Polishing**	- Extend functionality with multiple maps/randomization - Enhance UI (reset, hints, victory message) - Write user instructions and unit tests
Week 3 *(June 12 – June 18)*	**Broken Calculator & Fifteen Puzzle – Development**	☑ ~~- **Broken Calculator:** Create basic UI, target generation, equation evaluation~~ ☑ ~~- **Fifteen Puzzle:** Design grid UI, implement tile moves and shuffling~~ ☑ ~~- Basic prototypes for both activities~~
Week 4 *(June 19 – June 25)*	**Soma Cubes – 3D Puzzle Implementation**	- Design UI and define data structures for pieces - Implement rotation and placement logic - Develop core engine with unit tests
Week 5 *(June 26 – July 2)*	**Soma Cubes – UI Integration & Testing**	- Integrate UI with underlying logic - Enable interactive controls (drag/click) - Comprehensive user testing and debugging
Week 6 *(July 3 – July 9)*	**Euclid's Game – Implementation & Mid-term Prep**	☑ ~~- Build turn-based UI with initial AI strategy~~ ☑ ~~- Implement game-end logic and move tracking~~ - Prepare mid-term report and demo

Week 7 *(July 10 – July 16)*	**Odd Scoring – Implementation & Testing**	- Create UI for two-player game with linear track - Implement game loop with move buttons and AI strategy - Testing various scenarios and refining gameplay
Week 8 *(July 17 – July 23)*	**Make an Identity – Implementation**	☑ ~~- Develop UI for digit sequence puzzle~~ ☑ ~~- Implement expression evaluation logic~~ - Add extra features (parentheses support, hints, multiple puzzles)
Week 9 *(July 24 – July 30)*	**Number Detective – AI Integration (Part 1)**	- Create UI for number sequence input - Implement initial AI predictor and feedback loop - Incorporate rule-based logic and basic memory mapping
Week 10 *(July 31 – Aug 6)*	**Number Detective – AI Improvement & Testing**	- Enhance AI logic (improved rules or simple ML integration) - Implement persistent learning from user feedback - Thorough testing with diverse sequences
Week 11 *(Aug 7 – Aug 13)*	**Sorting Hat AI – Implementation (Part 1)**	- Set up UI for labeling (start with shapes) - Implement k-NN classifier with initial training - Collect training data and enable feedback for incremental learning
Week 12 *(Aug 14 – Aug 21)*	**Sorting Hat AI – Completion & Overall Polish**	- Integrate additional datasets (animals, numbers) - Refine AI algorithm and ensure UI consistency - Finalize user instructions and complete comprehensive testing
Final Week *(Aug 22 – Aug 28)*	**Submissions and Wrap-up**	- Finalize code, documentation, and demo videos - Merge code into Sugar Labs repositories - Conduct final testing, incorporate mentor feedback, and submit final evaluation

Availability

I am committed to contributing approximately **40 to 50 hours per week** to the project.. I will ensure consistent participation and focused work throughout the program to achieve all project goals and deadlines effectively.The GSoC timeline aligns well with my university's **summer break**, giving me ample uninterrupted time to focus on

the project. Even after the break ends, I will have **no exams or in-person classes**, and all academic commitments will be **asynchronous**, allowing me to continue contributing actively.

My typical working hours are from **10:00 IST (4:30 UTC) to 1:00 IST (19:30 UTC)**, during which I will be fully available and responsive. In case of any unforeseen circumstances requiring time off, I will promptly inform my mentors in advance.

Post GSoC Plans

As for myself (Ravjot Singh), after GSoC I plan to remain an active contributor to Sugar Labs. Future work can include me helping to mentor new contributors who might pick up these or similar projects. I could assist in code reviews or in improving these activities as I get feedback. Perhaps I could present these projects in community meetings or write a blog post about how they were implemented to attract more contributors to educational game development.

Conclusion

This project represents more than just the development of eight math games—it is an opportunity to empower children through interactive learning, foster curiosity, and build foundational problem-solving skills using open source technology. With a strong background in Python, educational tool development, and extensive contributions to organizations like Sugar Labs and the Linux Foundation, I am well-prepared to take on this challenge.

My past experiences contributing to educational platforms, working on scalable open source systems, and mentoring within community-driven projects have instilled in me a sense of responsibility and purpose. I view this project as a meaningful extension of that journey—one where creativity, logic, and community intersect to create tools that truly benefit learners.

Through structured development, thoughtful design, AI integration, and continuous testing, this project will deliver polished, purposeful, and engaging math activities. It will enrich Sugar Labs' ecosystem, align with its mission of learning through exploration, and leave a lasting impact on the global learning community.

I am deeply motivated to bring this vision to life and look forward to collaborating with mentors and contributors to make this project a valuable asset for Sugar Labs and a joyful learning experience for children around the world.