The Sorting Game: A New Way to Teach Computer Science in Outreach and Museum Settings

by

Adam M. Gleitman

S.B., Computer Science and Engineering
Massachusetts Institute of Technology, 2013

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering at the Massachusetts Institute of Technology

September 2014

© Adam M. Gleitman 2014. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Signature redacted

Author: .................................................................

Department of Electrical Engineering and Computer Science

August 13, 2014

Signature redacted

Certified by: ..................

James W. Bales
Instructor and Associate Director, MIT Edgerton Center

Signature redacted

Thesis Supervisor

Accepted by: ........

Prof. Albert R. Meyer
Chairman, Masters of Engineering Committee
The Sorting Game: A New Way to Teach Computer Science in Outreach and Museum Settings

by

Adam M. Gleitman

Submitted to the Department of Electrical Engineering and Computer Science on August 13, 2014 in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering

ABSTRACT

The Sorting Game is a game that teaches people about sorting algorithms in an engaging way. It consists of eight blocks, each with a secret number, and the user must arrange the blocks in numerical order. The user does not know any of the numbers, but may compare any two and figure out which block has a higher number. This is analogous to the comparison model of sorting. A secondary goal is to find the correct order in as few comparisons as possible. The Sorting Game can therefore lead to discussions about algorithmic thinking and other topics in computer science. This game, when completed, can be used as an interactive exhibit at a museum or as an outreach module in a classroom.

Thesis Supervisor: James W. Bales
Title: Instructor and Associate Director, MIT Edgerton Center
## Contents

1 Introduction  .................................................. 7  
   1.1 Motivation .................................................. 7  
   1.2 Design Goals ............................................... 9  
   1.3 Organization ............................................... 10  

2 Initial Prototype ................................................. 11  
   2.1 The Game: A General Overview .......................... 11  
   2.2 Identifying the Blocks .................................... 12  
   2.3 Juggling UARTs ............................................ 14  
   2.4 Tying Everything Together ............................... 16  
   2.5 Improvements Identified in the Initial Prototype .... 17  

3 Version 2: Hardening Things Up ................................. 19  
   3.1 Structural Changes From Prototype ...................... 19  
   3.2 A User-Friendly Display .................................. 20  
   3.3 The LEDs .................................................. 22  
   3.4 Improvements Identified in Version 2 ................... 24  

4 Version 3: Making It Smarter and Flashier ..................... 27  
   4.1 Physical and Structural Changes ......................... 27  
   4.2 Improving the Electronics ............................... 29  
      4.2.1 Commands .......................................... 29  
      4.2.2 Reading RFID Tags ................................. 31  
      4.2.3 LED Banks .......................................... 32  
      4.2.4 The Hardware Controller ......................... 33  
   4.3 LEDs: A Backup Plan ....................................... 33  
      4.3.1 LED Strips ......................................... 35  
      4.3.2 Seven-Segment Display Modules ................... 36  
      4.3.3 Performance ........................................ 38  
   4.4 The Main Application ....................................... 39  
      4.4.1 Activities .......................................... 39  
      4.4.2 Program Flow ...................................... 40
4.5 Manipulating Luck ................................................................. 41
  4.5.1 Hard Mode: An Overview ................................................. 42
  4.5.2 Theoretical Discussion ...................................................... 43
  4.5.3 Talking About Worst-Case Performance ............................. 44
4.6 Visualizing What We Know .................................................. 45
  4.6.1 Maintaining the Data Structure ....................................... 46
  4.6.2 Laying Out the Graph ..................................................... 48
  4.6.3 Smooth Transitions .......................................................... 50

5 Reflection and Evaluation ..................................................... 53
  5.1 Review of Design Goals ..................................................... 53
  5.2 Future Work .................................................................. 54
  5.3 Conclusion .................................................................... 55
1 Introduction

1.1 Motivation

Computer science is one of the fastest-growing career fields. U.S. News chose Software Developer as
the best job of 2014 due to factors such as good job security, numerous employment opportunities, and
competitive salaries. [1] Unfortunately, despite its importance, computer science is often overlooked in
public schools, thereby depriving students of the opportunity to learn about it, whether they are interested in
turning computer science into a career or just learning how to program for fun.

Fortunately, students can make use of other external resources to learn about computer science concepts
on their own time, including community-driven websites such as StackOverflow, official tutorials such as
the one provided in Python’s documentation, and general-purposes knowledge bases like Wikipedia. While
these resources are good for some self-learning, they can only go so far. For example, the Python docu-
mentation mentions many built-in data structures, such as lists, sets, and dictionaries, but does not go into
much detail regarding the advantages and disadvantages of each of these structures or in which contexts each
one should be used. Other sites like Wikipedia occasionally go into much more detail than an entry-level
programmer would need or want, and this can be intimidating. Some of them may even be too nervous to
create an account on StackOverflow just to ask a “stupid” question like, “How or why does a hash table do
such a better job than an array-backed list at lookups?”

Granted, it may seem ridiculous to teach a student who just learned Hello World how to design and
analyze optimal algorithms for various problems. After all, new programmers generally will not be working
with such large inputs that they need to worry about whether or not their algorithm’s running time is opti-
mal. Furthermore, some ideas behind optimizing algorithms are not at all obvious. For example, the naïve
$\Theta(n^2)$ algorithm for multiplying two $n$-digit numbers is obvious, but not optimal. Karatsuba’s algorithm, as
discussed in [2], runs in $\Theta(n^{\log_2 3})$ time, and while it is easy for a high school algebra student to verify that the algorithm itself is correct (and perhaps, with some guidance, figure out the break-even point where Karatsuba's algorithm is as efficient as the naive algorithm after taking constant factors into account), coming up with the algorithm is not very easy. Other FFT-based algorithms that can achieve $o(n^{1+\epsilon})$ running time, like those discussed in [3] and [4], are even more mysterious to a novice programmer, partially because these algorithms require undergraduate-level mathematics at a minimum to fully understand.

However, this does not mean that people cannot appreciate the fact that some algorithms are better than others or that certain problems can even be solved algorithmically. A simple demonstration can show that for extremely large numbers, an FFT-based multiplication algorithm will easily blow the naive multiplication algorithm out of the water. I believe that people can appreciate visualizations of these efficient algorithms without being programmers, just like how people can appreciate a musical masterpiece despite having never played an instrument or learned any formal music theory. Even better, when given proper guidance and a set of rules, one can easily compose a simple piece that sounds good. The same holds for computer science concepts; given a proper computational model, one can come up with an algorithm, data structure, etc. that is best suited for any particular problem.

This is the driving idea behind my thesis. My goal for this thesis is to create a prototype of an interactive exhibit about sorting algorithms that can be used in museum settings and classroom outreach programs. I have chosen to name it “The Sorting Game.” I have developed this prototype under the supervision of the MIT Edgerton Center. They have been involved in educational outreach programs for K-12 students, and they are starting to work with local museums to develop and improve exhibits. Furthermore, the Edgerton Center is very good at getting kids excited about mechanical engineering and electrical engineering, but they do not focus much on computer science because most of their staff members are not as familiar with computer science as with other disciplines. Therefore my project will prove to be a valuable asset to the Edgerton Center and its outreach efforts.

I have chosen the problem of sorting because is a simple yet important problem in the field of computer science. Most people are familiar with what it means to sort a list of items, but the first idea that usually comes to mind for a sorting algorithm is a quadratic-time algorithm, even though it is by no means the most efficient. This provides a perfect opportunity to show people how choosing the right algorithm can make a big difference.
1.2 Design Goals

For my thesis I mainly focused on developing a museum-grade exhibit, although I did think about how things would translate into classroom-style modules and what changes I would have to make. While working on this prototype, I had several design goals in mind, including, but not limited to:

1. **The Sorting Game should be fun to use.** We want users to enjoy the learning process. Turning sorting into a game definitely helps, but there are other ways to make the experience even more enjoyable.

2. **The Sorting Game should give the users a clear idea of why the problem of sorting is relevant.** This includes emphasizing the fact that there are multiple sorting algorithms, and that some are better than others.

3. **The Sorting Game should be easy to use.** The interface should be easy enough that a user can jump in without too much instruction. This is especially needed for museum exhibits, because there is less likely to be direct supervision on the museum floor than in a classroom where an outreach program is taking place.

4. **The Sorting Game should be easy to build and mass-produce.** The parts we use should come from familiar and reliable sources. Custom parts like mechanical enclosures or printed circuit boards are considered acceptable for this criterion, as they can be made with standard equipment or ordered from a third party.

5. **The Sorting Game should be resilient to abuse.** This needs to be true both in terms of physical abuse and unusual use cases in the underlying software. This is especially needed for museum exhibits, because users may attempt to misuse or break the exhibit in ways one would never expect.

6. **The Sorting Game should be relatively inexpensive.** This is especially needed for outreach modules, which are more likely to be sold to a wider range of users. Museum-grade exhibits could be more expensive than their classroom counterparts due to stricter requirements of resilience and because museums typically have higher budgets for these things than schools, but this doesn’t mean that museum versions should be unnecessarily pricey.

7. **The Sorting Game should be easy to repair or replace if a part malfunctions.** Teachers and museum employees should be able to operate a program, replace a part, reprogram a microcontroller, etc. if need be in order to fix any part of the module or exhibit if it breaks. This can be accomplished by providing all the basic tools needed, such as programmers, scripts, and instructions, to get a piece
of equipment up and running again. For custom-fabricated parts, they should be easy to make and inexpensive in case they need to get replaced. This is also helpful in a museum setting where a guest can “accidentally” walk away with one of the manipulatives, as a museum employee can grab a replacement from a large stash of spares.

1.3 Organization

This thesis is organized to show the different iterations of design that The Sorting Game has gone through. Sections 2, 3, and 4 discuss the first, second, and third versions, respectively. Of these three, we spend most of the time discussing the third version, as this is the latest version and is the most developed in its design. Section 5 reflects upon these versions, how well they meet the above design goals, and future work.
2 Initial Prototype

The initial development of The Sorting Game began in the spring of 2013 as my 6.UAP project. This section discusses the workings of this initial prototype and how it was received.

2.1 The Game: A General Overview

The Sorting Game consists of eight blocks that are distinguishable by certain physical features such as color, shape, or texture. When the user presses a button marked “Start,” a computer assigns a secret, randomly chosen number to each block. The user must then arrange the blocks in order from smallest number to largest number. To deduce the correct order, the user may take any two blocks and compare their values, only being told which block has the larger value (or if the blocks’ values are equal). Once the user believes that the blocks are in the correct order, he or she presses a button marked “Check,” which reveals the numerical value of each block and whether or not the user was able to sort them correctly. Figure 1 shows an initial sketch of the design of The Sorting Game.

![Initial Sketch of The Sorting Game](image)

Figure 1: An initial sketch of what I wanted The Sorting Game to look like, from a top-down perspective.

The user receives feedback on a display controlled by an appropriate piece of hardware. The display shows the results of previous comparisons, the total number of comparisons done so far, the numerical values of the blocks at the end of the game, and the current locations of the blocks as a way to provide real-time feedback. By doing this I allow the user to feel as though he or she is inside a computer, seeing what a computer sees and understanding its computational limits. The ability to compare two numbers and
only receive information on which one is bigger is analogous to sending these two numbers $a$ and $b$ through an arithmetic logic unit in a CPU and only receiving a few bits that indicate whether $a < b$, $a = b$, or $a > b$, which is essentially the model of computation used in comparison sorts.

2.2 Identifying the Blocks

In order to provide real-time feedback regarding the locations of the blocks, The Sorting Game needs a way of identifying the blocks and distinguishing them from each other. I have considered several different methods for doing this task.

Similar museum exhibits use reed switches, which close in the presence of a magnetic field, to detect certain patterns of magnets. If the blocks are made asymmetric, like the shape in Figure 2, then $n$ reed switches per block can account for $2^n - 1$ different blocks (not $2^n$ because we also need to detect when a slot is empty), assuming that each switch can act independently. If you want to allow for error detection/correction you need a few more reed switches. While this is a relatively inexpensive, tried-and-true approach, such a method would be difficult to implement quickly due to the amount of hardware needed and digital logic required to interpret them. Nevertheless, this is still a viable solution for the final product, both in the context of a museum-grade exhibit and a classroom outreach module.

![Figure 2: An asymmetric block. This shape is used in another exhibit at the Museum of Science that requires distinguishing between two classes of blocks.](image)

One recommendation that I received from Ed Moriarty, a technical instructor at the Edgerton Center, my freshman advisor, and my UAP advisor, was to use optical sensing. He suggested that the blocks have markings or infrared LEDs on the underside and to use a reflectance sensor or phototransistor to identify the blocks. While this could work, I was hesitant to implement this for two reasons. Firstly, at the time I was working on this project another student who is also working to develop an exhibit with the Museum of Science was experiencing some frustrations with getting an infrared LED and phototransistor pair to work
properly using reflectance. Although he was attempting to make his design work well even in suboptimal lighting conditions, and I could probably have more control over lighting, I wanted to avoid calibration problems wherever possible. Secondly, putting infrared LEDs or phototransistors inside the blocks would require the blocks to be powered by a battery. To keep things simple, I wanted the blocks to be completely passive, so this approach is not ideal.

Another suggestion I received was to use color sensing. Color sensors are relatively cheap (an entire sensor circuit, with a PCB, could easily cost less than $5.00 when produced in bulk), which would make this convenient for both museum exhibits and classroom outreach modules. However, they are sensitive to lighting conditions and false positives. As an experiment, I placed different colors of opaque acrylic squares over a TCS3200 color sensor and four bright white LEDs and measured which wavelengths of light got reflected back to the sensor. Under relatively dark lighting conditions, I was able to distinguish between different colors very reliably. However, when I placed a transparent acrylic square in between the opaque squares and the sensor circuit, the readings became less clear-cut. Furthermore, such a sensor may not be able to distinguish a block from another colored object, such as a the sleeve of a long-sleeved shirt.

Originally I planned to use a wide-angle video camera capturing an aerial view of the surface to read the tops of the blocks or in a cavity underneath the surface to read the bottoms of the blocks through a clear material. The tops or bottoms of the blocks would then be marked with distinct patterns, such as QR codes, that the camera can read. From this, we can deduce the locations of the blocks. Another exhibit currently on display at the Museum of Science uses this technology, so it should be fairly reliable. However, this would make the exhibit less compact.

Instead, I chose to opt for RFID technology, as suggested by Peter Moriarty, son of Ed Moriarty and an employee at the Museum of Science. Each block will have an embedded RFID tag that will allow a reader to uniquely identify it by a six-byte address, such as 65 00 12 A2 54 81. The main disadvantage of this is that the readers I am using, the ID-12 from ID Innovations, are somewhat pricey (SparkFun sells ten for approximately $270), but they are extremely reliable. Another minor issue is that the RFID readers have 2 mm pin spacing, but this can be easily solved with a $9.99 breakout board that SparkFun also sells. This part converts the 2 mm pin spacing to the more common 0.1" spacing. (As of July 2014 this specific part has been retired from SparkFun's catalog, but a SparkFun employee has confirmed that they are currently designing another revision of this breakout board.) The RFID tags themselves are relatively cheap, approximately $2.19 each when purchased in bulk from Digikey as of July 3, 2014.

Another advantage of RFID is that the RFID readers I am using have a very simple 9600-baud serial interface that reports addresses. The only downside to this is that in the UART configuration, the reader
does not report the removal of tags; it only reports a new tag when it is introduced to the reader. The reader can report the same address twice in a row if a tag is placed near the reader, removed, and then placed near the reader again. We solve this problem by resetting the RFID reader periodically. This way, the reader will continuously report the same tag as long as it is present, and it will stop reporting that tag once it is removed.

Experiments have shown that we need to hold the reset signal in place for 150 milliseconds to reset the module properly, and we have to wait another 150 milliseconds to ensure that we get a proper reading. The time spent doing computations is negligible compared to these values, which means we can get a reading every 300 milliseconds, a sample rate of approximately 3.3 Hz. This has been shown to be good enough for decent real-time feedback.

The Sorting Game contains ten RFID readers: one for each of the eight main slots, and two for the comparison slots. The RFID readers, attached to the aforementioned breakout boards, are seated on ten pieces of perfboard. Each piece of perfboard has a four-conductor ribbon cable attached to it: two for the 5 V power supply, one for the serial output, and one for the reset line.

2.3 Juggling UARTs

We now need a way to read all ten UARTs at once. Originally, I planned on using a single Arduino Mega to handle all ten UARTs. Unfortunately, an Arduino Mega only has four hardware UARTs, and the included SoftwareSerial library is incapable of listening to multiple serial ports at the same time. [5] Therefore, doing this would require implementing a serial library from scratch or using external logic. The former option would be rather difficult to implement, so I tried to implement the latter option.

I attempted to use three of the four hardware UARTs, leaving the fourth one for debugging purposes. I split the ten RFID readers into three groups. The transmit lines of the readers in a group were connected to the inputs of a large AND gate, and the output was connected to an RX line on the Arduino Mega. To ensure that no crosstalk would occur, I maintained the invariant that among all readers in a particular group, at most one could be kept active (the others must be held in reset mode). This was based on the assumption that while an RFID reader is being reset, its serial output is being held high. Therefore, the output of the AND gate would be low if and only if the active reader’s UART line was transmitting a low signal.

However, this ended up not being the case, and I was not able to get this to work. One thing that made this difficult was that the datasheet for the ID-12 did not go into details about how the data lines would behave under conditions when you would normally not read from them (like when you’re resetting the device). I probably would have had more success had I built a more complex logic circuit, but I did not do this at the time.
In the end, I decided to synthesize ten UART controllers on an FPGA. Specifically, I synthesized them on an Atlys board produced by Digilent. This specific implementation would obviously not be the best solution from a cost perspective, because an Atlys board normally goes for $419. However, I had this board lying around, and I decided that it would be perfect for this project. Also, there are cheaper FPGA boards available, such as the Papilio One, which starts at $37.99. (These prices are accurate as of July 3, 2014.)

On the other end, an Arduino communicates with these UART controllers through its own serial interface. If the Arduino sends a digit character \( c \) from 0 to 9, the FPGA will pull the serial data from reader \( c \) and send the stream back to the Arduino. If there is no tag present, the FPGA will send 0 0 0 0 0 0 0 0 0 0 0 0, which we assume is not the address of an actual tag. Figure 3 shows a block diagram of the resulting system.

![Figure 3: A block diagram of the exhibit's internal workings. The components within the dashed box were implemented on an FPGA.](image)

One caveat of this design is that the FPGA runs on 3.3 V logic, while the RFID readers and Arduino Mega run on 5 V logic. Luckily, it is very easy to solve these problems. The 5 V signals from the RFID readers and Arduino Mega can be passed through a voltage divider to make them 3.3 V signals for the FPGA. The 3.3 V output signals from the FPGA can be directly hooked up to the 5 V inputs on the Arduino Mega. This works because the value of \( V_{IH} \) for the digital inputs on the ATmega2560, the microcontroller
on the Arduino Mega, is 60% of VCC, or 3.0 V. [6] Technically this doesn’t guarantee that everything will work; what we really need is for the FPGA’s $V_{OH}$ to be greater than the ATmega2560’s $V_{IH}$. I have not been able to find any information on the FPGA’s $V_{OH}$, but this solution has not given me any problems related to logic level violations.

The only problem I came across was that the RFID tag address that the FPGA picked up would be prefixed by an extra nibble 2, and the last nibble would be dropped. For example, \texttt{65 00 12 A2 54 81} would turn into \texttt{26 50 01 2A 25 48}. The reason for this stems from the lack of a robust parser in the UART-to-Parallel modules. After being reset, the RFID reader’s serial line is pushed low for a very brief period of time. If our reader picks up on this, then it will receive an extra throwaway byte. However, the protocol of the RFID readers in serial mode is to send out a byte \texttt{0x02} (the ASCII code meaning “start of text”) before sending out the address of the tag we read as a 12-byte human-readable string. Our system is translating this into a digit 2 (which has ASCII code \texttt{0x32}). When designing an interpreter that translated bytes into nibbles, I decided to make invalid characters (i.e., those that are not hexadecimal digits) have undefined behavior because I did not expect that we would ever have to read them.

This issue, however, was easily solved in software. All the tags used in this version had addresses that start with \texttt{65 00}, so we could easily tell when the extra 2 nibble managed to sneak in. The only pitfall with this solution is that two tags whose addresses are identical except for the last nibble could not be distinguished if the extra nibble sneaks in. This could be solved by expanding the sizes of each bus by one nibble to be 52 bits instead of 48 bits, but this was not necessary because no two tags had such similar addresses.

### 2.4 Tying Everything Together

Now that we can read ten serial ports somewhat reliably, tying everything together to make a functional prototype is relatively straightforward. From here, it just becomes a matter of programming the Arduino to continuously poll the serial ports, reset them periodically, and read the status of the “Start,” “Compare,” and “Check” buttons.

For this prototype, these three buttons are on a solderless breadboard, the “blocks” are just the RFID tags with shapes drawn on them, and a serial terminal provided feedback to the user. The feedback was rather minimal and cryptic. The system would periodically print out 10 characters indicating which tags were in which locations, and when the user did a comparison, the system would print out one of three strings: "L < R", "L = R", or "L > R". From a user experience point of view, this prototype was barely functional without me explaining how it worked. At the time I did not worry about making this prototype
look good; I just wanted something that was functional.

However, the hacked-togetherness of this design led to many problems. The position of the buttons was dictated by whatever made the wiring convenient, so they were in an awkward place for the user. Therefore I ended up controlling the buttons from the side. Also, I occasionally got confused by the cryptic nature of the results, which led to me making mistakes. This caused the user to get the answer wrong.

Despite these shortcomings, volunteers still enjoyed playing The Sorting Game. The most memorable attempt involved three elementary school students visiting the Edgerton Center on a Saturday. They worked as a team to figure out which comparisons to do next and what information they could deduce from the comparisons they already did. They even used a whiteboard in the room to write down what information they had. In the end, they were able to put the blocks in the correct order.

2.5 Improvements Identified in the Initial Prototype

This prototype gave me a lot of ideas regarding how The Sorting Game can be improved. Firstly, we need a better way of telling the user which block has the bigger value. There are two ways this can be done. The first way is to use a gigantic less-than, greater-than, or equal-to sign in between the two comparison slots, lighting up the appropriate sign when the user performs a comparison. The second way is to put indicator LEDs around or underneath the two comparison slots and to light up the greater of the two blocks. Putting the LEDs underneath the slots would provide for a very cool effect if the blocks are thin enough and have the right amount of translucence to diffuse the light, but would require more careful planning and a small enough RFID tag. Putting the LEDs around the comparison slots instead would provide for a slightly less impressive visual effect, but would be easier to implement.

A second idea that I received was to make the device work on its own, without a monitor. This would involve including real-time feedback mechanisms on the physical device itself. This would include putting LEDs around or underneath the eight main slots (similarly to what was discussed in the previous paragraph) and placing seven-segment displays underneath each of the eight main slots that show the secret numbers once they are revealed, among other things. Another added bonus about this suggestion is that it can make The Sorting Game look more flashy and exciting. If the user gets the correct order, The Sorting Game can flash in celebration, similar to how a slot machine lights up and goes crazy when somebody hits the jackpot. However, I do not want to remove the monitor entirely, because it allows other people to watch (and possibly try to make suggestions) as somebody plays this game. The monitor also enforces the connection between the physical environment of blocks and buttons and the virtual environment of the computer.

A third idea is to show the last few comparisons that a user has done. This makes the game slightly easier,
as a user can look back, take what he or she knows, and draw conclusions. One may argue that showing too many previous comparisons (or showing any at all, for that matter) messes with the computational model, as using the last few comparisons to figure out the next move can take a substantial amount of time for a computer to do. However, showing a history of comparisons serves as a stepping stone for people to develop their own algorithm or to realize why certain algorithms are inefficient. For example, suppose the user is performing a selection sort on four numbers \(A, B, C, D\) by trying to find the minimum element first. The results of the first three comparisons are \(A > B\), \(B > C\), and \(C < D\), so \(C\) is the minimum element. When the user repeats this process with the elements \(A, B, D\), he or she might try to compare \(A\) and \(B\) first, but he or she can realize that he or she already knows that \(A > B\) due to the first comparison and can just proceed to comparing \(B\) and \(D\). Further prompting leads to the conclusion that when sorting \(n\) elements using a naïve selection sort, one may need to compare \(A\) and \(B\) \(\Theta(n)\) times in the worst case.

Finally, another idea that I had was to include a “workspace” that users could use when playing the game. After somebody played a round, I asked them about their strategy and how they would explicitly communicate it to a computer. Typically most users did something along the lines of an insertion sort or a selection sort. I then performed a quicksort or merge sort, which often ended up requiring significantly fewer comparisons. Then I had the user walk through one of these sorting algorithms, and they become amazed at how much faster the new algorithm is. However, having some extra room to place blocks that fell on either side of the pivot (for quicksort) or to make piles of sorted blocks to merge together (for merge sort) would have been very convenient.
3 Version 2: Hardening Things Up

To implement the previously discussed changes, I made a new version of The Sorting Game that expanded on the original prototype. The electronics are mostly the same; there are still ten RFID readers with parsers synthesized on an FPGA. The main difference here is in physical appearance.

3.1 Structural Changes From Prototype

For this second version, I designed a physical enclosure made out of laser-cut birch plywood. A diagram of the enclosure can be seen in Figure 4, and a picture of the enclosure can be seen in Figure 5. The enclosure is 29 inches long, 16 inches deep, and 1.75 inches high, so it can fit on a medium-sized table. One RFID reader is located underneath each slot and is fastened with Velcro to the bottom of the enclosure.

There are several notable changes to the design. Firstly, the eight main slots have been moved further away from the user and the two comparison slots have been moved towards the front. I chose to do this because I want to place more emphasis on the comparisons than on the final answer. Secondly, there is an LED adjacent to each slot, which serve as indicator lights. Thirdly, the buttons have been attached to the enclosure and labeled. Finally, there is now a workspace that players can use while playing the game.
The blocks are now pucks that are 3" in diameter and ½" thick that are made of three ⅛" plywood pieces glued together. The tag is embedded inside the puck, as seen in Figure 6. The shape for each puck is drawn on top. I was recommended to use round pucks for this iteration because one does not need to worry about the angle of rotation of the puck when placing it into the slot. Squares only have fourfold rotational symmetry, while circles have rotational symmetry with respect to any angle.

This was a big improvement over just using RFID tags. The blocks were much easier to manipulate since they had some physical substance to them. It was also very easy to slide the blocks into the slots because it was always easy to see which way the user had to slide the blocks in order to seat them in the slots. Also, as mentioned before, the fact that circles have infinite rotational symmetry means that users need to worry about one fewer degree of freedom when trying to place the blocks down.

3.2 A User-Friendly Display

For this version I also implemented a more user-friendly display for real-time feedback. The controller for the display is a Gameduino, an Arduino shield that facilitates the creation of retro-style games on an Arduino. The shield has a video controller synthesized on an FPGA that can output an $800 \times 600$ 72 Hz signal to a VGA monitor (the actual image is a $400 \times 300$ image scaled up by a factor of two). The Gameduino
RFID tag: 50 mm diameter, 1 mm thick
2.2” inner diameter

Figure 6: An exploded view of one of the pucks.

communicates with the Arduino using the ATmega2560’s built-in SPI controller. The only (minor) issue is that the Gameduino, like many other Arduino shields, assumes that the MOSI, MISO, and SCK pins are broken out on Arduino digital I/O pins 11, 12, and 13, respectively. While this is true for the Arduino Uno, it is not true for the Arduino Mega, where these pins are 51, 50, and 52, respectively. This problem can be easily solved by adding three jumper wires that connect pins 51, 50, and 52 to pins 11, 12, and 13, respectively, letting pins 50 through 52 control the Gameduino while leaving pins 11 through 13 as digital inputs so they do not interfere with SPI operations. We do not need to worry about the slave select pin because the Gameduino is the only SPI-enabled device we have hooked up to this circuit.

An example of the Gameduino’s video output can be seen in Figure 7. The Gameduino shows which shapes are in which slots. If an unknown RFID tag is placed in one of the slots, the slot on the screen will show a question mark. When the numbers are revealed, they are shown below each of the eight main slots. The total number of comparisons is shown in large text in the middle of the screen, and a history of the 28 most recent comparisons. (This number was determined based on how many lines could fit. The fact that 28 also happens to be \(\binom{8}{2}\), the number of unordered pairs of distinct values chosen from a universe of eight elements, is just a coincidence, but it does work out quite nicely.)
Figure 7: A rendition of the Gameduino's video output. Normally, the shapes are colored, the background is black, and the text is white, but I have opted for a slightly different coloring scheme for the sake of printability.

3.3 The LEDs

This version also includes ten LEDs, one along each slot. These LEDs help make The Sorting Game work as a standalone device (i.e., without a monitor), although some LED-based feedback mechanisms were less effective than others.

The LEDs near the comparison slots are used to indicate the results of comparisons. When the “Compare” button is pressed, the LED next to the slot with the larger element lights up. If the two blocks being compared have equal values, both LEDs will light up. The LEDs turn off as soon as one of the blocks is removed from the comparison slots.

The LEDs near the eight main slots serve two purposes. First, they serve as an indicator for how many comparisons have been done. This is done as a binary counter. The LED on the rightmost slot serves as the least significant bit, and the LED on the leftmost slot serves as the most significant bit. I wanted these LEDs to all be writable with one command, so I hooked them up to pins 22 through 29 on the Arduino Mega, which are all attached to PORTA on the ATmega2560. However, due to an unfortunate wiring decision, the order of the bits on the number to be displayed on the LEDs did not line up with the order of the bits on PORTA. Instead, bits 0, 1, 2, 3, 4, 5, 6, and 7 on the LEDs lined up with bits 6, 7, 4, 5, 2, 3, 0, and 1 on PORTA, respectively. Luckily, this problem turns out to be very easy and elegant to solve. We can write this
permutation $\pi$ as the composition of four cycles of length 2:

$$\pi = (0 \ 6) \ (1 \ 7) \ (2 \ 4) \ (3 \ 5)$$

Since each cycle has length 2, $\pi \circ \pi$ will map every element to itself. Alternatively, we can say that $\pi$ is equivalent to its inverse, $\pi^{-1}$. In other words, transforming values from a standard unsigned 8-bit integer to a value that will display the LEDs as intended is exactly the same as doing the reverse. Therefore this transformation can be described by a single 256-element lookup table that can be used to go in both directions. This could have been implemented as a sequence of bit tricks, but due to the simplicity of the AVR instruction set it would be faster to do a single memory lookup. The ATmega2560 has plenty of SRAM and flash memory space, so resource usage is not a big concern here.

The second purpose is to give the user some sense of satisfaction in the case of an almost-correct sort. For example, if the result of a user’s sort is $[10, 34, 37, 65, 71, 93, 51, 98]$, this can be considered “almost correct;” removing the 51 yields a sorted list of seven numbers. Finding the appropriate list of numbers at the end is the longest non-decreasing subsequence problem, a very well-known problem. By lighting up the elements that are in a longest non-decreasing subsequence, we can at least give users who were close somewhat of a consolation if they don’t get the correct answer and acknowledge partial progress for the user who decides to press the “Check” button after every single comparison.

The two comparison LEDs were intuitive. I did have to say that the LED that turns on corresponds to the bigger number and not the smaller number (there was no indication otherwise), but people got the hang of it very quickly. I found the comparison LEDs to be a better way of identifying the bigger number than using a less-than or greater-than sign, because it’s too easy to get confused when trying to work quickly. When two blocks ended up having the same number (causing both LEDs to turn on), users would be surprised for a brief moment. They would quickly realize that both lights on means that the two values are equal, but then they would wonder which one should go first. While The Sorting Game considers both options to be correct, this can lead to interesting discussions about what it means for a sort to be stable and why this might be a desirable property in certain cases.

The LEDs by the main slots, however, were rather confusing. After the first comparison was done, only the rightmost LED among the eight main slot LEDs would turn on. Some people thought this meant that the larger of the two blocks that were compared, whose corresponding LED was also on, should go in the rightmost position. I often had to clarify that this was not the case, and that it was only a binary counter showing how many comparisons have been done so far. While this was a neat curiosity, this feature turned
out to be highly impractical, and I would have been better off with two seven-segment displays showing the number of comparisons done.

3.4 Improvements Identified in Version 2

There were several big improvements in this iteration. However, there were also other issues that I saw.

First of all, while embedding the tags inside circular pucks made The Sorting Game more usable, there were other issues that came about from using circles. The main issue is that using circular pucks could imply that the user is supposed to rotate them like a dial. This is not the message that I want to get across; I want it to be obvious that the blocks are supposed to be picked up and moved around.

Another issue that I did not expect at all was confusion about the number displayed. Several users thought that this number represented the difference between the two blocks that were compared. Some users did not even recognize the fact that this number started at 0. One possible explanation for this is that the number showing how many comparisons were done was relatively large, and the 8×8 images typically associated with 8-bit fonts are better suited for smaller font sizes.

Thirdly, the buttons I used were rather flimsy and prone to accidentally getting pressed. This was frustrating when users were almost done completing a sort and they accidentally hit the “Reset” button, rendering all their hard work useless. A similar issue also happened with the “Compare” button, in which a single press would occasionally get registered as two separate pushes, incrementing the comparison counter by two instead of by one. I attribute this error to the buttons not being properly debounced.

At first I wanted to use arcade-style buttons on this version, but their total height, approximately 2.5 inches, was too large to fit in the enclosure I made (the height of the enclosure was limited by the thickness of a strip of pine that was used around the perimeter of the board and the height of the RFID readers when mounted on their perfboard stands). I therefore opted for whatever panel-mount pushbuttons I could find lying around the lab for the sake of rapid prototyping.

Fourthly, the internal wiring was not very robust. I used solderless breadboards and individual wires to connect the RFID modules to the FPGA and the FPGA to the Arduino Mega. If somebody was reaching inside the board to make repairs, one wrong move could dislodge one or more wires, and there was no clear indication as to where the wires should be put back. To make matters worse, the main power switch was on the FPGA board, and I was unable to position the FPGA board such that the switch was easily accessible from the outside. Therefore to turn the device on one needed to remove the top and reach into the machine. This increases the chances of a wire getting dislodged.

Fifthly, I found that my implementation of the entire system was not very robust in the long run. Due
to a more permanent appearance of this version of The Sorting Game, I was able to keep it running for extended periods of time. However, as it ran, the system began to malfunction. There were two ways in which this happened. The first way is that the reported positions of the blocks would get “shifted.” In other words, the block in slot 1 would appear in slot 2 on the display, the block in slot 2 would appear in slot 3 on the display, and so on. Another failure mode was that the system would no longer be able to distinguish different tags. The Sorting Game would still be able to recognize the presence of a tag, but the tag would appear as “unknown.” I have not been able to figure out the exact reason why these failure modes happen, but I would guess that the different parts of the system are going out of sync with each other. This problem can easily be solved by resetting the Arduino Mega, but this is not a very desirable solution, as it requires The Sorting Game to be under continuous maintenance.

Finally, this version of The Sorting Game was not truly able to run without an external display due to the confusing interface that the LEDs provided (mostly the ones near the main slots). There are five main functions that the display does, in order of importance:

1. Show the result of the latest comparison.
2. Show the total number of comparisons done.
3. Show the secret numbers associated with each block once the user presses the “Check” button.
4. Show the real-time locations of the blocks.
5. Show a history of the last $N$ comparisons, where $N$ is a conveniently chosen number.

If we only consider what the board does (i.e., if we ignore the display), this version of The Sorting Game does not do a very good job at being a standalone learning device. Out of the above five items, the only one that we can say works almost perfectly is item 1 (the only issue is that we do not specify that the illuminated LED indicates the larger number and not the smaller one). Item 2 is handled in a non-intuitive way. Item 3 is only partially handled in lighting up the longest non-decreasing subsequence, and as was pointed out before, this is not at all obvious. We do not even attempt to do items 4 and 5. While this version is a step in the right direction, it is far from complete. The next version does a better job at accomplishing these tasks.
4 Version 3: Making It Smarter and Flashier

For the third version of The Sorting Game, I thought carefully about how I originally envisioned this project before I started working on it. While I want this game to be educational, I want to also make it exciting and visually appealing, because this is what will draw users in. This train of thought motivated most of the changes that I made in this version, although there were other problems that I also resolved.

The changes can be summed up with the following four points:

1. The slots and blocks are now squares with rounded corners.
2. The hardware is more modular and more robust.
3. There are now multicolored LEDs and seven-segment displays next to each slot.
4. The display is now a computer application, which also serves as the master controller.

There are two versions of this system. The second is a slight modification of the first after I realized that several things did not work as planned.

4.1 Physical and Structural Changes

The enclosure has only gone through minor changes. It still has the same dimensions, but the slots are now 2.55" squares with rounded corners. They are slightly closer together to accommodate newly added LEDs. In addition to the LEDs, there are also seven-segment displays next to each slot. The enclosure also contains some internal support structures to maintain rigidity. A diagram of the first version of the new enclosure can be seen in Figure 8.

The RFID modules are now attached to custom-designed printed circuit boards, which are attached to the bottom of the enclosure with machine screws and hex nuts. The main hardware controller is now on another custom-designed printed circuit board, which is also attached to the enclosure with screws and nuts. Each of the RFID board connects to the main board through a length of ribbon cable.

The eight blocks are now 2½" × 2½" × ¾" rectangular prisms. Each block's corresponding shape is formed by a pocket of negative space. On the underside of each block is a 2 mm deep circular pocket where the RFID tag can sit. Every edge except for the four outermost edges has an ¼" diameter fillet to keep them smooth (the other four edges have a ¼" diameter fillet to make them concentric with the fillets on the inner rim). The blocks are made of ABS plastic and were made using a 3D printer. Originally the blocks
were going to be made of polypropylene and fabricated on a CNC mill, but 3D-printing the blocks ended up being cheaper and easier. A picture of one of these blocks can be seen in Figure 9.

The buttons are also more robust. For this version I have chosen to use three colored illuminated push-buttons from Adafruit. These buttons are much more structurally sound than the previous buttons; the actuators are 13 mm diameter cylinders instead of 6 mm disks attached to 3 mm shafts. This makes the buttons less likely to break or otherwise behave unexpectedly, which is very important when it comes to designing museum exhibits. The Museum of Science often uses buttons similar to those found on arcade and ticket-redemption games. These buttons are extremely resilient to abuse, but they are relatively expensive (around $5.00 each). The buttons from Adafruit that I used on this version are substantially cheaper (at most
$1.95 each depending on the color), and yet still robust and visually appealing.

Finally, turning the machine on is much easier. There is a 5 V, 4 A power supply inside the enclosure, with a standard power socket easily accessible from the side of the enclosure. There is also a power switch on the side of the enclosure and a power LED indicator on the front side of the enclosure.

4.2 Improving the Electronics

For this version I also designed and built custom printed circuit boards to work with each of the different slots independently. Each slot has its own dedicated ATtiny4313 microcontroller that interprets readings from an RFID reader. They also contain a local bank of LEDs and seven-segment displays, but in the end these were scrapped in favor of making them separate from the slot modules. Nevertheless, we will still discuss how they were intended to work for the sake of completeness.

A picture of an assembled slot module PCB can be seen in Figure 10. The schematic for the slot module can be seen in Figure 11, and the pinout for the ribbon cable can be seen in Figure 12.

![Figure 10: A picture of one of the ten slot modules PCBs.](image)

4.2.1 Commands

This microcontroller receives commands from a hardware controller via the hardware UART. All ten slot modules share the same command receive line, that is, the hardware controller will always send commands to every slot module at the same time. Therefore we divide commands into three categories: one-slot commands, all-slot commands, and controller commands. We will discuss controller commands later, as
Figure 11: A schematic of the slot module. The 0.1 μF decoupling capacitors, ISP header for the ATtiny4313, and unused lines on the RFID reader have been left out for the sake of brevity.

Figure 12: The pinout for the ten-pin ribbon connector. When the “TAG OK” pin is high, pins T3:0 indicate which known tag is present. When it is low, they indicate one of several error codes, such as “not ready to read” or “unknown tag.”
they are not relevant for the slot modules.

One-slot commands are intended for a single recipient and will indicate which slot module should re-
cieve them. Every slot module has a unique 4-bit address that is assigned in firmware. If a slot module
receives a one-slot command intended for a different address, it will ignore that command. All-slot com-
mands, on the other hand, are intended for all slot modules. A slot module will always execute an all-slot
command when it receives one.

There are five different types of one-slot commands:

- Set a particular LED to be a particular color. We allow for 6 bits of color resolution: two bits for each
  channel.

- Set the seven-segment displays to light up particular segments indicated by a 16-bit mask.

- Set the seven-segment displays to show a particular hexadecimal number. The advantage of using this
  command over the previous command is that this command only requires two bytes to be sent, while
  the previous one requires three.

- Set all LEDs to either be a particular color or off, depending on a 16-bit mask. The LEDs are changed
  simultaneously.

- Set all LEDs to either be the last color received or off. By “last color” we mean the color corresponding
to the most recent color command.

There is only one all-slot command: to register a new RFID tag address. We can store up to 16 different
addresses. While this may not seem necessary, one big advantage about having a safety factor of 2 is that if
any block gets stolen, damaged, or otherwise made unusable in a museum setting, an employee can switch
out a new block of the same color and shape while the exhibit is still running, and the code (or even just a
configuration file) can be changed after hours. I have not implemented this feature into this version because
the blocks are much less likely to disappear in a more controlled lab setting. However, this is still a useful
feature that can be implemented in the future.

4.2.2 Reading RFID Tags

The slot modules still make use of the RFID readers, and they still communicate using a 9600-baud UART.
Since the hardware UART on the ATtiny4313 is already being used to receive commands, we will make use
of two interrupt service routines: one triggered by a pin change and one triggered by an internal timer.
The RFID reader’s serial output is attached to pin D2 on the ATtiny4313, which we can attach to an interrupt service routine that triggers on a falling edge. We start by listening for this falling edge, as this corresponds to the start bit that UARTs send. When it occurs, we temporarily disable the falling-edge interrupt and enable a timer interrupt that will trigger 9600 times per second. We can configure the timer such that the first trigger will occur \( \frac{1}{19200} \) of a second after the falling edge, allowing us to manually poll pin D2 in the middle of the transmission of each bit, thereby reducing the chances of getting an erroneous reading. After retrieving the entire byte, we disable the timer interrupt and re-enable the falling-edge interrupt.

Once we receive a full address, or after 150 milliseconds pass without receiving any serial data, we reset the RFID reader for 150 milliseconds and then reactivate it. This is the same strategy we have used in previous versions to detect when the tag has been removed. The only difference is that this time, each RFID reader can be reset independently. This saves us time because once a reader reports that a tag is present, we do not have to wait for the other readers to confirm that they do not have tags nearby. In practice this saves us around 25 milliseconds of waiting time due to the delay between reactivating the RFID reader and being able to get a response from it. Nevertheless, this is an easy way to reduce the response time of the RFID readers.

There is one additional thing we do to optimize the slot modules for speed. When checking the received address against the other addresses we have registered, we compare the bytes in reverse order. With the set of eight tags I used to develop this prototype, the first three bytes of their addresses are all the same. However, each tag has a different last byte. We still need to check all six bytes before we can confirm a match, but checking the bytes in reverse order allows us to rule out incorrect matches as quickly as possible.

### 4.2.3 LED Banks

Each slot module contains 16 RGB LEDs arranged along the outside of a 2.6" square and two common-anode seven-segment displays at the bottom of the PCB. The LEDs have each of the red, green, and blue anodes and cathodes on six separate pins. The anodes are controlled by four 2N2907 PNP transistors: one for the red anodes, one for the green anodes, one for the blue anodes, and one for the anodes of the seven-segment displays. The cathodes are controlled by two TLC5916 ICs, which are daisy-chainable constant-current LED drivers that can control 8 LEDs each. The 64 cathodes among all the LEDs and displays are split into 16 groups; each group contains the cathodes of the three elements on a single LED and one of the segments on the displays. Each group is then connected to one of the \( \text{OUT}_n \) ports on the TLC5916s.

The LEDs and seven-segment displays are all handled by the same painting routine. We break up painting the LEDs in 12 frames, three frames for each channel as distinguished by the PNP transistors. For
the red, green, and blue channels, each LED can be on for anywhere between zero and three frames. This allows for four different duty cycles for each channel: 0, \( \frac{1}{12} \), \( \frac{2}{12} \), and \( \frac{3}{12} \). The seven-segment display channel does not have this feature because allowing for varying brightness is not as important for seven-segment displays.

4.2.4 The Hardware Controller

The hardware controller is another custom-designed printed circuit board. It contains an ATmega328p, which effectively makes the hardware controller an Arduino Uno. The slot modules attach to the hardware controller through ribbon cables, and the signals in them go into a chain of seven 74HC165 parallel-to-serial shift registers. Each slot module sends back five signals, so 50 of the 56 available inputs are used by the slot modules. Three additional inputs are used for the three pushbuttons. The other three inputs are unused for now and are tied to ground. The order of the signals on the shift register chain was chosen to make routing the hardware controller PCB as easy as possible. Breaking up a 56-bit value into meaningful information is handled by the main application.

Every 50 milliseconds, the hardware controller polls the shift registers and sends out their contents as a 14-digit hexadecimal number through its UART. This signal passes through an FTDI chip that relays these values to a USB port. At the same time, the hardware controller also listens for commands sent by a master computer. The hardware controller will pass along any one-slot and all-slot commands it receives through a software-controlled UART line, but it will absorb and act on controller commands. This version only has a single controller command: to change the status of the LEDs inside the pushbuttons.

4.3 LEDs: A Backup Plan

When I tried putting the entire system together, I was unable to get everything to work properly. An early test in which the LEDs would light up in a specific pattern when a tag was introduced showed that the slot modules on their own worked perfectly. However, once I attached them to the hardware controller, they did not seem to take commands or read tags properly when I tried to use the LED banks. Even if I sent a command to only turn on one LED, they would still light up erratically, and tags would occasionally disappear for several steps at a time. If I did not touch the LEDs or seven-segment displays, though, the RFID readers worked perfectly. I have not been able to figure out why this was happening, but it would not be acceptable for this version.

I solved this issue by considering substitutes that could be added “on-the-fly” and would also be decoupled from the RFID readers. From this point, I chose to ignore everything related to the LED banks
on the slot modules, and in their place I added a strip of individually addressable RGB LEDs and multiple
I²C-controllable seven-segment display modules. A drawing of the second enclosure design for the third
iteration can be seen in Figure 13, and a picture of it can be seen in Figure 14.

Figure 13: An overhead view of the third version of the enclosure.

Figure 14: A picture of the second enclosure for the third iteration of the system.
4.3.1 LED Strips

The LED strips consist of RGB LEDs with embedded WS2812 chips. These individually addressable LEDs are more commonly known as NeoPixels. NeoPixels are extremely well-documented and can be bought from many different suppliers, making them a very promising candidate for making this version even more visually appealing. In total, we use a 1 meter strip of 60 NeoPixels. By setting the slots to be 8.33 cm (approximately 3.28") apart, we can place exactly five NeoPixels underneath each slot. In total, there is one strip of 40 NeoPixels under the eight main slots and four strips of five NeoPixels each surrounding the comparison slots.

Adafruit provides an in-depth tutorial and an Arduino library for NeoPixels. [7] This should, in theory, make it very easy to connect a string of them to an Arduino and get them working in a few minutes. However, adding them to an already busy system is much more difficult. NeoPixels require very precise timing in order to function properly. Adafruit’s NeoPixel library handles this by disabling interrupts while sending data to the NeoPixels and embedding assembly code into the source code files to ensure precise timing. However, the amount of time needed to send all the data to the NeoPixels is rather significant. The WS2812 chip receives bits at a rate of 800 kHz, or 1.25 µs per bit. The chips also expect to receive a 24-bit color value, which means we need 24 bits per pixel. Finally, we need to wait at least 50 µs to allow the newly received colors to “latch.” Therefore the total amount of time we need to send one “frame” is:

\[
\frac{1.25 \text{ µs}}{\text{bit}} \times \frac{24 \text{ bits}}{\text{pixel}} \times 60 \text{ pixels} + 50 \text{ µs} = 1850 \text{ µs}
\]

This is a very long time to go without interrupts. By comparison, the UART that receives commands runs at 57600 bits per second, which corresponds to 5760 characters per second at maximum speed (8 bits per character, plus one start bit and one stop bit). This is equivalent to approximately 174 µs per character. In the amount of time we spend refreshing the NeoPixel strip, we could have received 10 characters. Since the ATmega328p uses an interrupt service routine to notify the CPU when a new character is received on the UART, we would not have any way of knowing if we received any commands during this time.

I chose to solve this by adding a second ATmega328p dedicated to controlling the NeoPixel strip. This microcontroller listens in on the commands the first one receives and executes NeoPixel-specific ones. With this comes the addition of several new controller commands:

- Set a particular NeoPixel to be one of 256 preset colors.
- Shift the colors of all NeoPixels one pixel to the left or right. The last color can either be dropped or
brought back to the beginning.

- Set whether or not the NeoPixels should be painted. This can be useful for flashing a particular pattern without having to turn off all NeoPixels and then reload the exact same pattern.

- Turn all NeoPixels off, discarding any previously held pattern.

- Commit the most recent changes made to the NeoPixel strip. This is what actually causes the next frame of NeoPixels to be painted.

By having a “commit” command, we can control exactly when the NeoPixels get updated. If we ensure that this is the last command we send on any particular step, and if there are at least 1850 μs (plus any overhead needed to prepare for writing to the NeoPixels) available, we should not run into any problems.

4.3.2 Seven-Segment Display Modules

The new seven-segment display modules are four-digit displays controlled by HT16K33 driver ICs. These ICs are I²C-controlled LED matrix drivers that can drive a 16×8 LED matrix. Since the pins in a multi-digit seven-segment displays are usually arranged in a similar fashion to an LED matrix, this driver is ideal for our purposes.

One issue is that we need to control 10 displays, but these drivers can only take on one of eight different I²C addresses from 0×70 to 0×77 inclusive. Therefore we cannot put all 10 display modules on the same bus. One way to solve this would be to create a software-controlled I²C bus that handles the last two displays. However, this would be rather difficult to implement. In order to ensure proper timing, we would need to either disable interrupts or make another interrupt-based system. The former idea is not at all feasible, since the Arduino library by default runs I²C with a speed of 100 kilobits per second, which means that updating the displays while staying consistent with the hardware I²C bus would take several milliseconds, a very long time to go without interrupts. The latter idea would be very time-consuming to implement and also runs the risk of using a timer that another Arduino library may want to use.

One would think that this is not a problem because we have two microcontrollers, so we can just use the second microcontroller’s hardware I²C bus. However, I did not want to do this because this microcontroller is already being used to control the NeoPixel strip. Since doing this already requires very precise timing, we would be better off keeping this microcontroller dedicated to performing a single task.

Instead, we implement a hardware hack that takes advantage of the fact that we only need one-way write-only access to use these displays. [8] describes a way to control nine of these displays using a single
hardware I²C bus and a 74HC138 3-to-8 line decoder with inverted outputs. However, this technique can be expanded to control up to 64 displays, as shown in Figure 15. The I²C data line is connected to all devices. The I²C clock line is fed into the inverted enable inputs of the line decoder, and a “line select” bus indicates which of the outputs on the line decoder will propagate that signal. The line select pins (pins 1 through 3 on the 74HC138) are connected to a microcontroller that also serves as the I²C master. By setting these three pins to a particular value, one of the outputs (Y0 through Y7) will respond to changes in SCK, while the other outputs will stay high. This effectively creates eight I²C buses where slaves can only receive data. Since the clock signals on the unused buses remain high, the devices on these buses will ignore data that is not intended for them.

Figure 15: A schematic for a circuit that can independently control 64 I²C slaves whose addresses are limited to eight different values. This will only work if the slaves do not need to send data back to the master.

However, our circuit does not need to be nearly as complex, since we only need to control 10 displays. This requires only two buses, so a 1-to-2 line decoder will suffice. We can either use a small portion of a larger decoder, or we can make our own. Let “bus 0” and “bus 1” be the names of the two buses, and let C, S, C₀, and C₁ represent the original SCK, the line select (S = 1 means we are selecting bus 1), the clock
line for bus 0, and the clock line for bus 1, respectively. Since an I²C clock is high during an idle state, $C_0 = 0$ if and only if $S = 0$ and $C = 0$, and $C_1 = 0$ if and only if $S = 1$ and $C = 0$.

These lead to the following formulas:

$$C_0 = S + C$$
$$C_1 = \overline{S} + C$$

These can be rewritten as follows using De Morgan's laws:

$$C_0 = \overline{S \cdot C}$$
$$C_1 = \overline{S + C}$$

Since $\overline{X}$ is equivalent to $\overline{X \cdot \overline{X}}$ for all $X$, we can generate $C_0$ and $C_1$ from $C$ and $S$ using four NAND gates, as seen in Figure 16. Therefore we only need a single 74HC00 IC.

![Logic circuit](image.png)

Figure 16: A logic circuit that controls the clock signals of two one-way I²C buses using the original clock signal and a line select signal.

### 4.3.3 Performance

This new system works very well. One thing I noticed as I was implementing this was that the library I was using to control the seven-segment display modules was particularly wasteful. This is because the IC driving the display is capable of controlling four times as many digits, as the LEDs in a four-digit seven-segment display are arranged in a $4 \times 8$ LED matrix configuration, and the library I am using sends data for the entire $16 \times 8$ matrix. This means that we are wasting 96 bits of information, and at 100 kilobits per second, we are wasting at least 960 μs of time. If we want to change all ten displays, we would end up wasting 9600 μs, almost 20% of the time we have available between steps. This is a problem that can be solved in a later version.
4.4 The Main Application

For this version I decided to use a computer as the master controller. Using a computer application as opposed to, say, a Gameduino, makes developing a nice-looking GUI much easier, and it also allows for more sophisticated graphics. Specifically, I created a Python application using PySide, a library that provides Qt bindings for Python. I have chosen this over the built-in Tkinter module because Tkinter lacks certain desirable advanced drawing features such as Bézier curves and antialiasing.

4.4.1 Activities

The framework of the application consists of multiple activities, which represent different things the user can do (similar to the notion of activities on Android). These activities are organized on an activity stack, which keeps track of what the user has done. When the activity stack is empty, the application terminates. Each activity this application uses is a subclass of Activity, an abstract class that I wrote for this application.

The Activity class has five methods:

Activity.parse_data_packet(data) sets certain specifications for an activity. For example, in the main game activity (where users actually try to perform a sort), one such specification is an object that specifies how the blocks’ secret numbers are selected (with or without repetition, for example). data is a dictionary whose specifications are activity-dependent. As long as the activities are constructed correctly and this method is overridden correctly, the keys and values can be anything that Python allows.

Activity.start() is called when an activity starts. This is different from parse_data_packet because parse_data_packet is only intended to be called once, while start could be called multiple times.

Activity.step(inp, hardware) performs a step through this activity. This method is called whenever the hardware controller reports the 56-bit value from the shift registers. inp is a 2-element tuple whose elements are the 10 RFID reader results and the statuses of the three buttons. hardware is a handle that allows the activity to send one-slot commands to the slot modules in order to control the LEDs. This method returns a tuple (stay_alive, new_activity). The first element is a Boolean value that indicates whether or not this activity should keep running. If this value is false, the activity will get popped off the activity stack as soon as possible. The second element is either a new activity to add to the activity stack (and will therefore be acted upon until it is finished), or None if we do not wish to create a new activity.

Activity.end() is called once an activity ends. After this method is called, we should never refer
to this instance again.

`Activity.paint(qp, width, height)` draws the activity onto a window. `qp` is an instance of `PySide.QtGui.QPainter` and is responsible for doing the actual painting. `width` and `height` are the width and height of the window in pixels, allowing us to draw items that scale properly with resolution. The implementations of this method in various subclasses of `Activity` assume that the window’s resolution has a 4:3 ratio, so the actual rendering may look weird if this condition does not hold.

For this version I have implemented two different subclasses of `Activity`: `MainMenuActivity` and `GameActivity`. It is possible to add more activities later on to add new features to this application, but these two activities provide the core functionality of this game. The `MainMenuActivity` allows the user to select different game modes. I originally wanted to include other non-game modes like tutorials on different sorting algorithms or a race mode in which users can race two algorithms against each other, but I was unable to do this due to lack of time. The `GameActivity` is where the main sorting game takes place. The GUI for this activity is similar to the GUI provided by the Gameduino in the second version, but this one has several stylistic changes.

### 4.4.2 Program Flow

When the application is started, we start a thread that listens for serial data. Whenever we receive a line that contains 14 hexadecimal digits, we parse that value into the statuses of the RFID readers and buttons based on which signals are connected to which inputs of the shift registers. These statuses are wrapped into Python objects and passed to the activity on top of the activity stack through the `step` method. Since the hardware controller sends this information out every 50 milliseconds, the program will execute approximately 20 steps per second.

This is what happens during every step:

1. Parse the 14-digit hexadecimal number into RFID reader and button statuses.

2. Call the top activity’s `step` method, passing the RFID reader and button statuses in as an input.

3. Draw the current state of the activity onto the window using the `paint` method.

4. If the `step` method indicated that this activity should end, call its `end` method and pop it off the stack.

5. If the `step` method indicated that a new activity should be spawned, push this new activity onto the top of the stack and call its `start` method.
4.5 Manipulating Luck

When people play The Sorting Game, they implement a variety of strategies. If I see them heading in a specific direction, I will recommend a similar but better sorting algorithm. For example, selection sort can lead to quicksort, and insertion sort can be improved from comparing new blocks against every other block to doing a binary search, which can lead to discussions about sorting by constructing a self-balancing binary search tree. Usually I would introduce quicksort as an \( \Theta(n \log n) \) algorithm (even though this is only with high probability and not guaranteed) due to its simplicity. However, I noticed that the number of comparisons needed to complete a quicksort can vary significantly. The probability distributions for how many comparisons are needed for an 8-element quicksort and insertion sort are shown in Figure 17.

![Figure 17: The probability distributions for how many comparisons are needed for 8-element quicksort and insertion sort.](image)

Based on this information, it would appear that quicksort is only slightly better than insertion sort. The mean and standard deviation for the number of comparisons needed to perform an 8-element quicksort are \( \mu \approx 16.921 \) and \( \sigma \approx 2.862 \), while for an 8-element insertion sort these numbers are \( \mu = 17.5 \) and \( \sigma \approx 3.329 \). This is not a very significant improvement. If an insertion sort and quicksort are performed independently, the quicksort will finish first 52.58% of the time, the insertion sort will finish first 38.62% of the time, and both will use the same number of comparisons 8.8% of the time.

However, quicksort will almost always do at least as well as selection sort, which always requires 28 comparisons to complete if the algorithm below is followed strictly:
**Selection-Sort(L)**

1. for $i = 1$ to $(|L| - 1)$
2. $\text{min-index} = i$
3. for $j = i + 1$ to $|L|$
4. if $L[j] < L[\text{min-index}]$
5. $\text{min-index} = j$
6. swap $L[i]$ and $L[\text{min-index}]$

This leads to an important topic in algorithms: worst-case analysis. An insertion sort on $n$ items could only require $n - 1$ comparisons if the user is extremely lucky. This luck factor also contributes to the problem that quicksort sometimes performs worse than insertion sort, even though quicksort is the better algorithm when you are sorting many items.

### 4.5.1 Hard Mode: An Overview

I therefore implemented another optional game mode, which I call “hard mode.” In the standard version of The Sorting Game, each block is assigned a secret number, and we stick with that arrangement of numbers until the end of the game. In hard mode, the computer is allowed to manipulate the numbers to make sorting as difficult as possible for the player, as long as the arrangement of numbers never contradicts a previous result. For example, if we say that $A < B$, then any future selection of numbers for this round will always satisfy $A < B$. We make this as difficult as possible for the player by choosing the more likely comparison result at every stage.

More precisely, suppose we are sorting $n$ numbers. In the beginning we choose $n$ distinct numbers, but we do not yet assign them to particular blocks. Instead we keep track of a candidate list of permutations of the integers 0 through $n - 1$. Each permutation therefore corresponds to a way those $n$ numbers could be assigned to the blocks. For example, if $\pi$ is a particular permutation in the candidate list and $\pi(0) = 3$, then this permutation assigns block 0 the number with rank 3 (where rank 0 is the lowest and rank $n - 1$ is the highest). The candidate list originally starts with $n!$ elements. Whenever we do a comparison, say, with item $A$ and item $B$, we split the candidate list $L$ into two lists $L_1$ and $L_2$. $L_1$ contains all permutations $\pi$ for which $\pi(A) < \pi(B)$, and $L_2$ contains all permutations $\pi$ for which $\pi(A) > \pi(B)$. If $|L_1| > |L_2|$, then it is more likely for item $A$ to be smaller than item $B$, so we report that item $A$ is smaller. Conversely, if $|L_1| < |L_2|$, we report that item $A$ is bigger. If $|L_1| = |L_2|$, we choose the result at random. Finally, the candidate list now becomes one of $L_1$ or $L_2$, whichever one has more elements.

In order for the user to get the correct answer in this mode, he or she must do a sequence of comparisons that reduces the number of candidate permutations to 1. If the user tries to check an answer when $|L| > 1$, the computer will choose a permutation that does not correspond to the user’s guess. After we
choose this permutation, though, we are obviously forced to stick with it if the user chooses to do any more experimentation with the blocks before starting a new game since the numbers have already been revealed.

To go from a permutation $\pi$ and a sorted list of candidate numbers $S$ to the corresponding assignment of numbers, we compute $\pi^{-1}S$. We need to invert the permutation because if $\pi(0) = 3$ as in our previous example, $\pi S$ would put the smallest candidate number at position 3, but we want the opposite. This representation makes doing computations more efficient because in a single round we work with $\pi$ many times, but we only need to compute one permutation inversion: the one that corresponds to the correct answer.

This mode always ensures that the player has "bad luck" when it comes to making comparisons on otherwise unknown pairs of items. In this mode, quicksort and insertion sort will always take $n(n-1)/2$ comparisons to complete. However, other deterministic algorithms like merge sort will still be very efficient. Therefore a very good way to take advantage of this mode’s educational value would be for a user to do insertion sort on hard mode and then a merge sort.

4.5.2 Theoretical Discussion

The functionality of hard mode is closely related to the problem of counting linear extensions of a strict partial order. A partial order is a binary relation $\prec$ on a set $S$ that is irreflexive ($a \not\prec a$ for all $a \in S$) and transitive (for all $a, b, c \in S$, if $a \prec b$ and $b \prec c$ then $a \prec c$), but it is not necessarily true that any two elements of $S$ are comparable. A total order also requires that the relation be trichotomous, that is, for all $a, b \in S$, either $a \prec b$, $b \prec a$, or $a = b$. A linear extension of a partial order (or partially ordered set, also known as a poset) is a total order that is compatible with said partial order. Technically, these are all strict orders, but if we forbid two different elements from having the same value (which we will assume for this section), all our partial orders automatically become strict.

At any point during a round of The Sorting Game, the user’s knowledge can be represented by a partial order on the set of blocks. The goal is to deduce the correct linear extension of this partial order by making comparisons. In the standard mode of play, the correct linear extension is fixed from the beginning, but in hard mode the keep track of all possible linear extensions and eliminates incorrect ones as slowly as possible.

The problem of counting linear extensions, which I will refer to as LIN-EXT-COUNT, takes in a finite poset $P = (S, \prec)$ and returns the number of distinct linear extensions of $P$, which we will denote as $N(P)$. When a user who is playing hard mode makes a comparison, say, with $a$ and $b$, we need to see which result is more likely, or in other words, whether or not $N(P \cup (a \prec b))$ is greater than $N(P \cup (b \prec a))$, where $P \cup (a \prec b)$ denotes the poset formed by adding $a \prec b$ and other transitivity-implied relations to $P$. Alternatively, we can word this as computing $\Pr(a \prec b \mid P)$, the probability that $a \prec b$ in a randomly
chosen linear extension of \( P \) assuming that all possible linear extensions of \( P \) are equally likely. This is equal to \( \frac{N(P \cup \{a < b\})}{N(P)} \).

Unfortunately, \( \text{LIN-EXT-COUNT} \) is known to be \#P-complete. This can be proven by a reduction from \( 3\text{-SAT-COUNT} \), which computes the number of variable assignments that make a 3-SAT statement evaluate to true. [9] Therefore a polynomial-time algorithm to count the number of linear extensions of a poset is unlikely to exist (such an algorithm would imply that \( P = \text{NP} \)). While technically not \#P-complete because it is not a counting problem, computing \( \Pr(a \prec b \mid P) \) is just as hard as \( \text{LIN-EXT-COUNT} \) because one can show that these two problems are polynomially equivalent to each other. While there is a fully polynomial randomized approximation scheme for estimating \( N(P) \) (and consequently \( \Pr(a \prec b \mid P) \)), it is not very practical for our purposes. One such algorithm estimates \( N(P) \) to within a factor of \( 1 + \epsilon \) with probability at least \( 1 - \delta \) in \( O(n^3 \log n \log^2 N(P)\epsilon^{-2} \log(1/\delta)) \) time, where \( n \) is the number of elements in the poset. [10] However, in our case, \( n \) is small enough that we can just brute-force through all possible permutations and get an exact answer without much more overhead. Furthermore, although randomized algorithms are usually good, using them would not completely remove the luck factor from The Sorting Game.

However, we do not need to know the exact value of \( \Pr(a \prec b \mid P) \); we only need to know whether or not \( \Pr(a \prec b \mid P) > \frac{1}{2} \). I will call this problem \( \text{LIN-EXT-HALF} \). Clearly this problem is at least as easy as computing \( \Pr(a \prec b \mid P) \), but I do not know if this is still hard. It might be possible to use an oracle for \( \text{LIN-EXT-HALF} \) on modified versions of \( P \) to effectively do a binary search to figure out the exact value of \( \Pr(a \prec b \mid P) \). Since this probability will always be a fraction whose denominator is at most \( n! \), a binary search would require \( O(\log(n!)) = O(n \log n) \) calls. However, I have not been able to find any positive or negative results. Without an efficient algorithm for this, true hard mode would be impractical for larger versions of The Sorting Game that use more blocks, although we can get rather close with an efficient randomized algorithm.

4.5.3 Talking About Worst-Case Performance

The addition of hard mode naturally leads to discussion about worst-case performance for a particular algorithm. For example, quicksort and insertion sort can take up to 28 comparisons for 8 elements, while merge sort only requires 17 comparisons in the worst case. One could then wonder what the “best” algorithm is in terms of fewest number of comparisons in the worst case. (Here we are talking in absolute terms, not in asymptotic terms.)

An obvious lower bound for the worst-case number of comparisons to sort \( n \) items is \( \lceil \log_2 n! \rceil \) comparisons. Since there are only two possible outcomes for a comparison, one result is guaranteed to contain at
least half of all possible permutations by the Pigeonhole principle. Therefore the number of comparisons after \( k \) comparisons will always be at least \( n!/2^k \). For \( n = 8 \), this lower bound is \( \lceil \log_2 40320 \rceil = 16 \).

Merge sort falls short of this goal by one comparison. However, it is possible to sort eight items using at most 16 comparisons using an algorithm known as the Ford-Johnson Algorithm, which I will abbreviate as FJA. [11] Before encountering this paper I came up with a very similar algorithm that uses FJA as a subroutine to sort a five-element list (which I developed independently) in 7 comparisons and to then place the remaining three elements into the list using binary search, with each element requiring at most 3 comparisons. This yields a total of \( 7 + 3 \times 3 = 16 \) comparisons.

This can lead to another question: Is it always possible to sort \( n \) elements in \( \lceil \log_2 n! \rceil \) comparisons? The answer to this question is no. FJA achieves this lower bound for \( n \leq 11 \) and \( n = 20, 21 \), but it fails to do so when \( 12 \leq n \leq 19 \) and for infinitely many values of \( n \geq 22 \). Ford and Johnson conjectured in 1959 that FJA is optimal in the number of comparisons needed to sort \( n \) elements. Since then it has been discovered that FJA is optimal for all \( n \leq 15 \) and for \( n = 22 \), and that FJA is not optimal for \( n = 47 \). [12] However, it is still unknown whether or not FJA is optimal for \( n = 16 \). [13] FJA can sort a 16-element list in 46 comparisons, but the information-theoretic lower bound is \( \lceil \log_2 16! \rceil = 45 \). Somebody who has mastered The Sorting Game could be inspired to study this problem and perhaps discover some interesting results.

### 4.6 Visualizing What We Know

The idea of hard mode forces people to think about what makes a “good” comparison. To illustrate this we would need a visual representation of what the user knows. A perfect candidate for this is a Hasse diagram, a graphical representation of a finite poset \( P = (S, \prec) \). A Hasse diagram is essentially a directed acyclic graph (also known as a DAG) \( G = (V, E) \), where \( V \) consists of one vertex for every element of \( S \) and \( E \) consists of exactly the edges \( (x, y) \) such that \( y \) covers \( x \), that is, \( x \prec y \) and there does not exist a third value \( z \) such that \( x \prec z \prec y \). We will refer to this as the covering invariant.

One particularly useful thing a Hasse diagram can tell us is whether or not two elements are comparable. Specifically, it is known that \( x \prec y \) if and only if the corresponding DAG contains a path from \( x \) to \( y \).

The implementation of this idea can be split up into three subproblems: maintaining the data structure itself, laying out the graph, and making visually appealing transitions from one graph to another. I did not have time to fully implement this idea into the software, but I have set up part of a framework that allows for this.
4.6.1 Maintaining the Data Structure

The first problem we need to solve is maintaining the data structure and its covering invariant. Consider the following sequence of actions with $V = \{a, b, c\}$:

1. Compare $a$ with $b$ and discover that $a \prec b$. Then we add $(a, b)$ to $E$.
2. Compare $a$ with $c$ and discover that $a \prec c$. Then we add $(a, c)$ to $E$.
3. Compare $b$ with $c$ and discover that $b \prec c$. Then we add $(b, c)$ to $E$ and remove $(a, c)$ from $E$.

We must remove edge $(a, c)$ from the graph in step 3 because $c$ no longer covers $a$, as we now know that $a \prec b \prec c$. We will call such an edge redundant. An edge can be redundant whether or not it is actually part of the graph.

To maintain this data structure, we need four methods:

1. **ADD-EDGE**($G$, $(u, v)$), which adds edge $(u, v)$ to $G$ without regard for the covering invariant.
2. **REMOVE-EDGE**($G$, $(u, v)$), which removes edge $(u, v)$ from $G$ without regard for the covering invariant.
3. **IS-REDUNDANT**($G$, $(u, v)$), which checks whether or not $(u, v)$ is a redundant edge in $G$.
4. **UPDATE**($G$, $a$, $b$), which updates the graph based on discovering that $a \prec b$. This makes appropriate calls to the other three methods.

We claim that an edge $(u, v)$ is redundant in a graph $G$ if and only if $G$ contains a path from $u$ to $v$ that does not include the (possibly nonexistent) edge $(u, v)$. This is very easy to show. For the forward direction, if $(u, v)$ is redundant, then there exists a path from $u$ to $v$ with some intermediate node $w$. We may break this up into two paths: one from $u$ to $w$ and another from $w$ to $v$. The existence of these two paths indicate that $u \prec w$ and $w \prec v$, which implies that $v$ does not cover $u$. For the reverse direction, if there exists a value $w$ such that $u \prec w \prec v$ (where $u, v, w$ are all distinct), then there exists a path from $u$ to $w$ and another path from $w$ to $v$. Therefore there must exist a path from $u$ to $v$ that passes through $w$, which implies that $(u, v)$ is redundant.

We have chosen to represent the edges of the graph $G = (V, E)$ using a collection of $|V|$ sets backed by hash tables. Each set $S_u$ contains the vertices $v$ such that $(u, v) \in E$. Therefore **ADD-EDGE** and **REMOVE-EDGE** consist of one table lookup to find the appropriate set and one insertion/deletion from a hash table, so these methods run in $O(1)$ time with high probability.
Using the fact that we proved above, we implement IS-REDUNDANT with a depth-first search among
the graph to find a non-direct path from \( u \) to \( v \). This takes \( O(E) \) time.

We can implement UPDATE naively. We add the corresponding edge to the graph, check every other edge
for redundancy, and remove those that are redundant. We can check each edge for redundancy independently.
Suppose \((u_1, v_1)\) and \((u_2, v_2)\) are both redundant edges that are actually in the graph. We claim that even
if we remove \((u_1, v_1)\), \((u_2, v_2)\) will still be redundant. If the redundancy-proving path from \( u_2 \) to \( v_2 \) does
not pass through \( u_1 \) or \( v_1 \), we’re done. If it does pass through both \( u_1 \) and \( v_1 \), note that since \((u_1, v_1) \in E\),
there must be at least two paths from \( u_1 \) to \( v_1 \) in the graph, so removing \((u_1, v_1)\) will not disconnect any two
vertices from each other. Therefore \((u_2, v_2)\) will still be redundant. This requires one call to ADD-EDGE
and \( O(E) \) calls to IS-REDUNDANT (and possibly REMOVE-EDGE). Therefore this naïve implementation
runs in \( O(E^2) \) time.

However, we can improve this by coming up with a necessary and sufficient condition for what makes
an edge \((u', v')\) redundant when a different edge \((u, v)\) is added. This condition is that the addition of
\((u, v)\) must complete a path from \( u' \) to \( v' \), or equivalently, \( u' \leq u \) and \( v \leq v' \). We can check for this by
first assembling two sets \( U' \) and \( V' \), which consist of the non-strict predecessors of \( u \) and the non-strict
successors of \( v \) respectively by performing a depth-first search. Then for every edge \((u', v')\) except for the
one we just added, we check if \( u' \in U' \) and \( v' \in V' \). If so, this edge is redundant and can be removed.
This redundancy check requires \( O(E) \) setup time and \( O(1) \) time per query with high probability. Therefore
UPDATE can be done in \( O(E) \) setup time with high probability.

One issue is that it is possible for \( E \) to contain \( \Theta(V^2) \) elements. This is possible by partitioning \( V \) into
two subsets \( V_1 \) and \( V_2 \) of size \( \frac{|V|}{2} \) and adding edges \((v_1, v_2)\) for each \( v_1 \in V_1 \) and \( v_2 \in V_2 \), giving a grand
total of \( \frac{|V|^2}{4} \) edges. This setup corresponds to knowing which elements are in the lower half of the sorted list
and which elements are in the upper half, but nothing else. Since this graph only contains paths of length at
most 1, no edge can be redundant.

However, this may not be that big of a deal. For this version of The Sorting Game, \(|V| = 8\), which is not
incredibly large. If we wanted to do graph representations for sorting algorithms with a larger input, these
algorithms may not even require \( \Theta(V^2) \) edges. Also, the number of edges is also bounded by the number of
comparisons done. Each unique call to UPDATE adds exactly one edge to the graph, and possibly removes
edges as well. This would have to be investigated further if we want this feature in the final version.
4.6.2 Laying Out the Graph

The next part of the problem involves laying out the graph. There exist numerous solutions for drawing directed graphs. I chose to use the one discussed in [14]. This algorithm has also been implemented into an open-source tool called Graphviz, which allows for many different configurations and constraints. The algorithm takes in a directed acyclic graph $G = (V, E)$ and outputs the coordinates for each node and B-spline control points for the edges. It is broken up into four steps: RANK, ORDERING, POSITION, and MAKE-SPLINES. RANK is responsible for assigning each node an integer rank such that for every edge $(u, v)$, the rank of $v$ is greater than the rank of $u$, thereby ensuring that all edges will be pointing in the same general direction. ORDERING arranges nodes with the same rank in a particular order to attempt to minimize the number of edge crossings in the resulting graph. POSITION determines the exact coordinates for each node to make the graph look visually appealing. MAKE-SPLINES determines the control points for the B-splines that make up each edge.

The biggest insight I have on making the layout algorithm work for The Sorting Game involves RANK.

For our purposes, RANK is responsible for assigning each node $v$ an integer rank $\lambda(v)$ such that for every edge $(u, v)$, $\lambda(v) - \lambda(u) \geq 1$. This allows us to formulate the problem of rank assignments as an integer linear program:

$$\begin{align*}
\text{minimize} & \quad \sum_{(u,v) \in E} (\lambda(v) - \lambda(u)) \\
\text{subject to} & \quad \lambda(v) - \lambda(u) \geq 1 \text{ for all } (u, v) \in E
\end{align*}$$

Graphviz's solution is a bit more flexible, allowing for features such as applying different weights to certain edges or forcing a particular edge to be longer, say, requiring $\lambda(v) - \lambda(u) \geq 2$ for some edge $(u, v)$. However, we do not need these features for our purposes.

Although solving integer linear programs in general is NP-complete, the constraint matrix for this problem is totally unimodular, so we can solve this particular ILP by solving the corresponding LP-relaxation. Alternatively, this can be solved by converting this problem into an equivalent min-cost flow or circulation problem.

As long as the graph is maintained properly, this linear program is always feasible. Every partial order has at least one linear extension, so we can choose one of them and assign each node a distinct rank between 1 and $|V|$ inclusive corresponding to its position in the linear extension. Technically this might not work if $\delta(e) \neq 1$ for some $e \in E$, but we can just multiply all ranks by $\max_e \delta(e)$ to solve this.
However, the number of solutions to this linear program is always infinite. Given a feasible solution \( \lambda \), we can generate infinitely many more feasible solutions by adding a constant \( c \) to each rank because we only care about rank differences. More generally, if the graph consists of \( k > 1 \) disjoint connected components, each connected component can move independently of each other. In other words, if we label each connected component with an integer between 1 and \( k \) inclusive, let \( \kappa(v) \) denote which component \( v \) belongs to, and let \( c_1, \ldots, c_k \) be any \( k \) integers, then any feasible solution \( \lambda \) can be transformed into another one by adding \( c_{\kappa(v)} \) to the rank of \( v \) for each \( v \). The most extreme case of this is when \( E \) is empty, in which case there are no restrictions at all on what the ranks of each vertex can be, although one would think that the “best” solution should set all ranks to be the same.

We can solve this issue by doing two things. Firstly, we two additional imaginary blocks whose values are known to be \( +\infty \) and \( -\infty \). Therefore, the graph starts out as a single connected component with \( 2n \) edges and \( n + 2 \) vertices instead of \( n \) individual vertices, where \( n \) is the number of real blocks being used. Therefore the graph will always consist of a single connected component, as removing redundant edges does not affect connectivity. Secondly, we add the constraint \( \lambda(u_{-\infty}) = 0 \), where \( u_{-\infty} \) is the node corresponding to the \( -\infty \) block. These two fixes make this linear program report the same answers more consistently.

There is one other issue with this approach, however. We will illustrate this issue with an example. Consider a knowledge state for a sort on seven elements \( x_1, \ldots, x_7 \) where we know that \( x_1 < \ldots < x_5 \) and \( x_6 < x_7 \), but nothing else. Then any of the three graphs in Figure 18 can be used as an optimal solution. However, the graph in the middle seems to be the best choice, as the nodes in the resulting graph are more evenly spaced out.

One way to encourage keeping nodes roughly equally spaced out is to use a quadratic program instead of a linear program:

\[
\begin{align*}
\text{minimize} & \quad \sum_{(u,v) \in E} (\lambda(v) - \lambda(u))^2 \\
\text{subject to} & \quad \lambda(v) - \lambda(u) \geq 1 \text{ for all } (u,v) \in E \\
& \quad \lambda(u_{-\infty}) = 0
\end{align*}
\]
This quadratic program allows for fractional rankings. It is rather easy to adapt the rest of the algorithm to allow for this.

In general, quadratic programming is NP-hard, as it is possible to construct a quadratic program that can solve the subset sum problem. However, in the specific case that the optimization function is convex, quadratic programming is feasible in polynomial time. In other words, suppose the function we are looking to minimize or maximize is written in the form $f(x) = \frac{1}{2}x^T Qx + c^T x$, where $Q$ is a symmetric matrix. If $Q$ is positive-semidefinite, then the quadratic program is solvable in polynomial time.

In the case of the above quadratic program, $Q$ can be written as a sum of $|E|$ matrices $Q^{(1)}, \ldots, Q^{(|E|)}$, corresponding to the edges $(u_1, v_1), \ldots, (u_1|E|, v_1|E|)$ in $E$. Each $Q^{(i)}$ contains exactly four nonzero elements: $q_{u_i v_i}^{(i)} = q_{v_i u_i}^{(i)} = 2$ and $q_{u_i v_i}^{(i)} = q_{v_i u_i}^{(i)} = -2$. It follows that the eigenvalues of $Q^{(i)}$ are 4 with multiplicity 1 and 0 with multiplicity $n - 1$. Since all eigenvalues are nonnegative, $Q^{(i)}$ is positive-semidefinite.

Since the set of positive-semidefinite matrices is closed under addition and nonnegative scaling (which is easy to prove using the fact that a positive-semidefinite matrix $A$ satisfies $x^T Ax \geq 0$ for all $x$), $Q$ must be positive-semidefinite as well.

I have not yet tried this approach, but it seems to be a valid alternative. There may be ways to formulate this particular quadratic program as another problem that is more specific but has much faster algorithms for solving. Nevertheless, our work here shows that this problem is indeed feasible.

### 4.6.3 Smooth Transitions

Another thing that needs to be considered is making sure transitions from one knowledge state to another look good. The algorithm we currently use draws each graph independently, but in the ideal situation the output of one graph should depend on the layout of the previous graph. Otherwise, large pieces of the graph can move around in unexpected ways. For example, if the graph would consist of multiple disjoint connected components had we not added $v_+, v_-$, then any two of these components could switch places in the final graph for no particular reason. This is especially worrisome since [14] prefers heuristic approaches and finding locally minimal solutions over demanding global minima (especially for ORDERING, where finding the global minimum is NP-hard), which can lead to different solutions depending on initial conditions. I did not have time to think about this problem, as solving this might require reworking large portions of the graph-drawing algorithm.

A separate problem we have to deal with is making sure that each edge moves smoothly from one knowledge state to the next. I did not have time to implement a solution, but I have looked briefly into possible approaches for solving this problem. According to the source code of Graphviz ([17]), the B-
splines that MAKE-SPLINES comes up with appear to be collections of cubic Bézier curves. This is supported by how Graphviz creates PostScript files of the graphs it renders. The code for this can be found in /graphviz/plugin/core/gvrender_core_ps.c. Specifically, the function of interest is psgen_bezier, which can be found in lines 326–355 in commit 7e5612e, the most recent commit as of August 12, 2014 that changed this file. The control points are translated into \( \frac{k-1}{3} \) cubic Bézier curves, and there are no precautions taken for cases where \( k \neq 1 \pmod{3} \). Therefore we seem to only be working with cubic Bézier curves.

A Bézier curve of degree \( n \) is a parametric curve \( \mathbf{B}(t) \) defined by \( n+1 \) control points \( P_{0,0}, P_{0,1}, \ldots, P_{0,n} \) and a parameter \( t \) that ranges from 0 to 1 as follows: For \( i = 1, 2, \ldots, n \) and \( j = 0, \ldots, n - i \), define \( P_{i,j} = (1-t)P_{i-1,j} + tP_{i-1,j+1} \). Then \( \mathbf{B}(t) = P_{n,0} \). This method of evaluation is known as de Casteljau's algorithm. [18] Alternatively, we can define \( \mathbf{B}(t) \) explicitly:

\[
\mathbf{B}(t) = \sum_{i=0}^{n} b_{i,n}(t)P_{0,i}
\]

where \( b_{i,n}(t) \) is the Bernstein basis polynomial defined as:

\[
b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}
\]

If corresponding edges in two consecutive states both consist of one curve, then moving the corresponding control points linearly should generate a nice effect. If we replace each control point \( P_{0,t} \) with a linear function of another parameter \( s \) that ranges from 0 to 1, corresponding to the old and new positions of the control points respectively, the explicit definition of the Bézier curve becomes:

\[
\mathbf{B}(t, s) = \sum_{i=0}^{n} b_{i,n}(t) \left( (1-s)P_{0,i}^{(\text{old})} + sP_{0,i}^{(\text{new})} \right)
\]

which is linear in \( s \).

If corresponding edges consist of different numbers of curves, we need to split them into smaller curves. This subdivision is very easy; de Casteljau proved that if the \( P_{i,j} \)'s are evaluated for a particular value of \( t \), then \( P_{0,0}, P_{1,0}, \ldots, P_{n,0} \) are the control points for the left side of the Bézier curve, and \( P_{0,n}, P_{1,n-1}, \ldots, P_{n,0} \) are the control points for the right side of the Bézier curve. [18] Figuring out where we should divide the curves is the hard part, as we would need to create a mathematical model for judging aesthetics. This should still be doable, since [14] comes up with such a model, but doing so would require further experimentation.
If corresponding edges have the same number of curves, we can get the same problem. If the curves vary wildly in length and shape, doing a simple control-point-to-control-point linear transition might look strange. Therefore we may need to use the aforementioned subdivision trick.

In the end I did not have time to implement this feature into The Sorting Game. However, if I did, there would be certain precautions that I would have to take. We only have 50 milliseconds per step, and the graph layout and transition algorithms could take longer. Therefore the tasks of doing these layouts would have to get delegated to another thread to prevent the program from locking up. This is easy to handle, but it is still very important.
5 Reflection and Evaluation

Overall, I consider this project to have been a success. Although I was unable to create what I would consider a final product that is ready to be sent to classrooms and museums, I would say that this product has seen substantial progress through its three iterations of design.

5.1 Review of Design Goals

Let us now review the design goals discussed in section 1.2.

1. *The Sorting Game should be fun to use.* I would consider this to be mostly successful. When people have played The Sorting Game, I would often see other people saying things like, “Compare the heart with the triangle!” or, “You already did that!” These interactions show that The Sorting Game can get many people involved, either as spectators or as advisors. These interactions, in my opinion, contribute significantly to the enjoyment of this product.

   The biggest improvement I could make in this respect would be to provide a more satisfying form of feedback when the user wins. In version 1 this was a line printed to a serial terminal. In version 2 this was eight LEDs turning on, giving a very cryptic message. In version 3 this can be accomplished very easily using the LED strips, although the only thing that prevented this from happening was time.

2. *The Sorting Game should give the users a clear idea of why the problem of sorting is relevant.* I feel that these three versions of The Sorting Game, as standalone products, do not accomplish this goal. This is because the relevance of sorting is better suited as supplemental information. In a museum setting, this might appear on a plaque near the exhibit. However, I do have other ideas for features that help accomplish this goal that I did not have time to implement in this version.

3. *The Sorting Game should be easy to use.* I would consider this to be moderately successful. The only issue is that people would need to be told the rules of the game beforehand, as without this guidance users in the museum would most likely just play with the blocks and buttons. One observation that a Museum of Science exhibit designed pointed out to me is that most people don’t even read the instructions for an exhibit, so the user would have no idea what to do unless they are explicitly and directly told so.

   One way to solve this would be to create a non-invasive tutorial. This could include showing that the blocks are getting assigned random numbers (not what the numbers are, obviously), prompting users...
to make comparisons, and getting them to put the blocks back in the eight main slots when they think they have the correct order.

4. The Sorting Game should be easy to build and mass-produce. I would consider this to be very successful. All parts used in the construction of this game can be bought from readily available sources. Due to hardware hacks that I needed to do to make different features in the third version work, though, there are still minor adjustments that need to be made. However, these should be relatively easy fixes if I wanted to remake version 3.

5. The Sorting Game should be resilient to abuse. The software and hardware for version 3 are solid, but I still think there are improvements to be made to the mechanical enclosure. The enclosure for version 3 is made out of dovetailed and glued pieces of 1/8" birch plywood. It may be better to make the wood thicker or to use a different material that is more robust.

6. The Sorting Game should be relatively inexpensive. As a museum exhibit, this product is rather cheap. Some exhibits at the Museum of Science make use of touch-screen monitors, which can cost several thousand dollars. The most expensive parts of The Sorting Game are the RFID readers, which only cost a few hundred dollars.

One obvious improvement that can be made would be to reduce the size of the slot module PCBs. Originally these were meant to house the LEDs, but since these did not work, and most of the circuitry on the board was dedicated to controlling the LEDs, the size and cost of each board can be greatly reduced.

As a classroom module that teachers can buy, this product is very expensive, so I would need to look into alternative technologies.

7. The Sorting Game should be easy to repair or replace if a part malfunctions. I have mixed feelings on this goal. For most use cases, we have met this goal, since it is very easy to 3D print another block. However, if something inside the machine breaks, then it is rather difficult to take the entire machine apart and fix it. Also, due to large amount of ribbon cable used, the inside of the enclosure is rather messy.

5.2 Future Work

There are several other features and improvements that I want to implement in future versions. I have mentioned some of these in passing in previous sections, but I want to outline the key ones here.
Firstly, I want to consider an alternative to RFID readers. While these readers are very robust, they are still somewhat expensive and would not be suitable in a classroom setting. A cheaper alternative might be to use magnetic sensors underneath each slot that detect magnets on the blocks. We would have to deal with cases such as rotated blocks and blocks that are not actually in the slots, but these should not be too difficult to handle.

Secondly, I want to add two additional modes to enhance the educational value of The Sorting Game. The first of these modes would walk the user through different sorting algorithms. The second would automatically run two different sorting algorithms on a larger dataset, allowing the user to see how certain algorithms are better than others.

Thirdly, I want to take the graph drawing algorithms discussed in 4.6 and make them part of the application itself if I choose to keep it in the final version. For the sake of prototyping, I had the application make a separate system call to Graphviz and interpret the standard output from that command. Ideally I would want to implement the graph drawing algorithm and other related functions natively into the application.

Fourthly, I believe the software could be improved if we made it asynchronous. Right now, any code we want the main controller to run must either complete in under 50 milliseconds or get scheduled to another thread. This also puts a limit on how much we can change the LED strips and seven-segment displays, as it takes time to send commands. We should only need to do updates when something relevant happens, such as when a block is moved. I think this would make developing the software much easier.

Finally, I want to make this unit functional on its own. While having an external computer application being the master makes The Sorting Game computationally powerful, it also makes The Sorting Game a bit difficult to set up. One way to do this might be to use a Raspberry Pi. There are already many libraries that allow people to interact with GPIO pins, hardware UARTs, and I²C buses, and a Raspberry Pi allows for many different peripherals that one would use in a computer, such as audio/video output, USB ports, and hardware interface devices like mice and keyboards. Including this would substantially reduce the cost of this game. While other computers can cost hundreds of dollars, a Raspberry Pi can be bought for less than $40. In addition to lowering the cost, using a Raspberry Pi allows the main controller to communicate with the rest of the system directly. This would save us a lot of time.

5.3 Conclusion

I believe The Sorting Game does a wonderful job in teaching people about computer science. The success of The Sorting Game can lead to other interactive exhibits or modules that teach about other computer science concepts. These products can serve to fill a gap in the spectrum of STEM outreach. Getting people excited
about other branches of science or engineering is much easier because the results are physical. A notable example of this is the “aurora bearealis” (a pun on aurora borealis), a popular outreach project that the Edgerton Center likes to use. It consists of three colored bright LEDs that refract inside an acrylic shape (the first shape ever used was a bear) to create interesting lighting effects. From start to finish, a person who has never built anything before and knows nothing about electronics can create one of these and understand the theory behind it in about an hour.

Getting people excited about computer science is a bit more difficult, as the problems and results tend to be more abstract. The Sorting Game takes one of these abstract problems and presents it in a tangible way. Also, the results are still just as portable. The Sorting Game can easily be implemented as a web game or a mobile app, allowing users to take it with them wherever they go.

I plan on continuing to work on The Sorting Game to bring it to a point where it could be mass-produced and sold to museums and classrooms. From what I have seen, The Sorting Game has the potential to be a huge hit, and it is both fun and educational. By making computer science fun and accessible to the general public, we can inspire a new generation of computer scientists. Somebody could end up developing a breakthrough in computer security, solving P vs. NP, or creating the next billion-dollar website, and it all will have started from one thing that sparked their curiosity: The Sorting Game.
References


