HardCode: How To Turn Good Programmers Into Great Programmers

Mathew Migliore

Computer Science Honours Thesis

Supervised by Dr. Michael Miljanovic

University of Ontario Institute of Technology

April 21, 2023

Abstract

Algorithmic concepts are often confusing and difficult to grasp [25]. Students who do not take the necessary time to learn and understand these concepts will be at a stark deficit while career searching compared to their peers who have. In university, many achieve their grades through memorization of content instead of mastery of it [15]. The lack of content mastery may lead to new graduates who struggle with algorithmic concepts, thus having a less-than-desired outcome during technical software interviews. This thesis investigates solving these issues through HardCode: an educational video game aimed to assist users in learning and mastering algorithmic concepts needed to succeed during technical software interviews and throughout their careers [6].

Contents

List of Figures

1 Introduction

1.1 Problem Statement

Technical software interviews represent a significant challenge for people seeking employment in the software industry [17]. Preparation for these interviews and amidst them may pose a challenging and often disheartening experience. Therefore, a tool is needed to help users gain experience with the concepts necessary to succeed in their interviews while ensuring they feel comfortable within the interview environment.

As the job market for software developers quickly becomes over-saturated, there needs to be a way to help users prepare for these interviews, become better at algorithms, and stand out while also having an enjoyable, motivated experience in the learning environment. Thus, this thesis explores the following research:

How can game-based education improve a programmer's knowledge of algorithm creation?

1.2 Motivation

It is important to have serious educational video games that help users master algorithmic concepts, learn code quality, and award users for the efficiency of their code rather than the speed at which it is developed. Therefore, having educational video games with these ideologies at their core will be a necessary progression in the educational video game space targeted at teaching advanced programming concepts.

The true motivation and the reason behind choosing to create this game and to do this thesis began when I bridged from Durham College to Ontario Tech University. At Durham College, no courses prepared you for technical software interviews. So after my bridge, one of the first questions I asked my new Ontario Tech peers was, "How do I become better at programming? How do I prepare

for my career?". While half of my peers encouraged learning algorithms, the other half stated that developing a game would be beneficial; thus, I combined both ideas and began creating an educational video game about algorithms. The goal instilled within HardCode is to materialize the path I took while learning these algorithmic concepts; to teach the users as I have taught myself.

1.3 Contributions

This thesis contributes to the area of algorithm education in computer science through the creation of HardCode. HardCode is an educational video game designed with the intention of helping users learn, as well as master, algorithmic concepts. HardCode is a serious educational video game developed with the purpose of shedding light on the advantages of game-based education, which can aid in enriching the learning experiences of users. Furthermore, this study serves the purpose of laying the foundation for future research exploring the effectiveness of game-based education in preparing users, including computer science graduates, for technical software interviews. Thus, the goal of HardCode is to follow the ideologies mentioned above and to better prepare its users for technical software interviews.

2 Background

2.1 Introduction

Educational video games have become an increasingly popular method to enhance learning experiences across various fields, including computer programming [18, 19, 16]. The potential to increase intrinsic motivation through various factors, such as endogenous fantasy, has been drawn as a significant advantage of utilizing video games in education [11, 8]. Endogenous fantasy is the integration of game mechanics and learning content, where the game narrative and challenges directly relate to the learning objectives, thereby leading to better engagement and learning outcomes for students [11, 19].

In computer science education, various games have been developed to teach fundamental programming concepts [18, 16]. However, there is a lack of games targeting advanced programming concepts, such as algorithm efficiency and code quality[18], which are crucial for success in technical software interviews and subsequent careers [24, 14]. Furthermore, the effectiveness of using playability heuristics in educational video games has been investigated to ensure that future games can provide a more positive user experience and help users accomplish their learning objectives more effectively [13].

With the increase in demand for skilled computer scientists [24], being less knowledgeable about algorithmic concepts may be detrimental during one's career search. Therefore, while pursuing a career in software, it is essential to understand these concepts; however, it is recognized that they are often challenging to grasp and abstract, making them difficult to learn and master. A potential solution for helping people with this problem is the gamification of education [7, 23], which involves game-like elements to engage users in the hopes of adding motivation.

While gamification has shown clear promise in other aspects of education [21, 26], there is still limited research on its effectiveness in helping programmers master programming concepts to achieve better success during technical

software interviews [18, 7]. To better understand the potential benefits of using game-based learning tools, this thesis will examine various aspects of educational video games, such as narrative [19, 20], playability heuristics [13], and endogenous fantasy [11, 8]. Additionally, this study will explore the use of mastery learning assessments in the context of game-based learning [15]. Combining these elements aims to develop a comprehensive understanding of how educational games can effectively improve students' learning experience and outcomes grappling with advanced programming concepts. Thus, by exploring game-based learning tools, this study aims to enhance the learning experience and results for students struggling with complex algorithmic concepts.

2.2 Past and Related Works

Many educational video games are designed to teach computer programming, and although there are still limitations in using them as learning tools for all scenarios, the future may be promising [18, 16]. Dr. Miljanovic and Dr. Bradbury, researchers and professors at The University of Ontario Institute of Technology, conducted a review of serious games for programming education and found them effective learning tools [18]. Dr. Malliarakis also discussed the use of educational games in teaching computer programming, stating that these games appeared to successfully fulfill the educational goals in relation to the group of concepts they set out to teach [16]. Dr. Qurat-ul-Ain reviewed technological tools in teaching and learning computer science and found that, though educational technology in education is still in its evolutionary stages, there is much promise in its inclusion in learning environments [24]. Dr. Habgood discovered that the endogenous integration of learning content and game mechanics led to higher levels of intrinsic motivation and more effective learning, supporting the use of video games in education [11]. The narrative approach has also been explored, with Dr. Naul and Dr. Liu demonstrating that using narrative elements within the context of programming can positively impact student engagement

and understanding [19]. The effectiveness of using playability heuristics in educational video games has been investigated to ensure that games can provide a more positive user experience and accomplish their learning objectives more effectively [13]. Dr. Ibrahim conducted a review on playability heuristics in evaluating educational video games and found that it can help identify areas for improvement and contribute to the overall success of these games [13]. However, there is a lack of games targeting advanced programming concepts, such as algorithm efficiency and code quality [18], and even less so are games designed to master them. This section will expand upon these issues and their potential solutions and speak more about educational video games and websites.

2.3 Educational Content

2.3.1 Educational Games

This section will be solely about the utilization of educational video games to teach programming concepts, including algorithms.

Code Combat [3]

Code Combat is about programming the game as it is happening. It is conceptually a different idea than HardCode because of this, as well as being aimed at teaching new programmers and children. However, it is still an excellent game to study due to its success.

Robocode [2]

Robocode is similar to Code Combat in that you must program the actions and movements as the game progresses: "where the goal is to develop a robot battle tank to battle against other tanks in Java." However, this game does not focus on the efficiency of the user's code but instead simply on its development. It also focuses on teaching multiple units of learning, and the learning objectives are in accordance with learning object-oriented programming (e.g., inheritance,

polymorphism) with the end goal of having the user build a robot ready for combat.

CodinGame [4]

CodinGame is a series of several mini-games with no storyline linking them together. A study was completed on students' perceptions of learning programming with CodinGame, in which some of the perceptions were as follows:

- "P70: I had trouble understanding the game story and the problem."
- "P12: The games in CodinGame have different independent stories."
- "P36: It took me too long to understand the purpose of the game."

The above testimonials show the need for a cohesive storyline—a bigger picture to motivate the users to continue to learn.

2.3.2 Educational Websites

This section will solely reference specific websites and their utilization of gamification elements to help motivate and teach their users programming concepts, including algorithm creation. The websites expanded upon below are of major use in assisting users with their success during technical software interviews due to their focus on algorithmic concepts.

LeetCode [5]

One cannot simply mention the topic of learning algorithms and algorithmic concepts to succeed in technical software interviews without noting LeetCode. LeetCode uses gamification as part of its website, including badges (submissions) and daily check-in coins. However, it is not a video game; the user has no storyline to follow. Regardless, this leads us to the advantage: there is no path

if you do not want one; it is free learning at your own pace. Courses can be too structured for some learning styles, following precisely what is given in the allotted amount of time, but with LeetCode, the user can create their own path entirely.

AlgoExpert [6]

"Acing the coding interview." AlgoExpert aligns its goals with HardCode by directly trying to help its users prepare for technical software interviews. However, while AlgoExpert includes gamification elements, these elements are packaged in a website. In contrast, HardCode aims to envelope educational ideas and algorithmic concepts into a video game that includes a storyline and fantasy to create intrinsic motivation.

2.4 Understanding Immersive Gameplay

2.4.1 Heuristics in creating effective educational video games

This section will touch on the heuristics needed to create an engaging educational video game.

Comparison of learning in games vs. fun

Educational video games must find a balance between two realms: i) education and playfulness and ii) a game that is easy to learn but challenging to master [13]. First, given that the scope of this thesis has the target audience of individuals familiar with programming, users should be able to understand what they are meant to accomplish generally, thus quickly picking up on the game. Secondly, since the primary goal of HardCode is to help users master algorithmic concepts, the game is designed with the notion of "Easy to use, difficult to master."

Immersion in Educational Video Games

Four narratives correlate with engagement in educational video games [20]:

1. Distributed narrative
2. Endogenous fantasy and intrinsic integration
3. Empathetic characters and virtual agents
4. Adaptive and responsive storytelling

Expanding Upon The Four Narratives

1. *Distributed narrative*

The distributed narrative is a type of storytelling approach that scatters the narrative elements throughout the game environment, allowing the user to discover and piece together the story at whichever pace they opt to. This type of narrative can create a sense of exploration and curiosity, encouraging users to engage actively with the game world. Furthermore, in an educational context, distributed narratives can help learners construct their understanding of a topic by discovering relevant information or solving puzzles, leading to deeper comprehension and retention of the material.

2. *Endogenous fantasy and intrinsic integration*

Endogenous fantasy refers to a narrative that deeply intertwines the game mechanics and learning objectives, integrating the fantasy world and the educational content. The intent is for the game's story and characters to serve as entertainment for the user and as a way to facilitate the learning process. When users are immersed in a well-designed endogenous fantasy storyline, users have significantly better learning outcomes compared to an exogenous fantasy, as the educational content becomes an integral part of the game experience rather than an extrinsic add-on.

3. *Empathetic characters and virtual agents*

Using empathetic characters and virtual agents in educational video games can create a strong emotional connection between the user and the game world. These characters can serve as role models, mentors, or companions, guiding the user through the learning process and providing support, feedback, or encouragement. These characters can enhance the user's motivation to learn and deepen their engagement with the educational content by fostering empathy and emotional investment.

4. *Adaptive and responsive storytelling*

Adaptive and responsive storytelling refers to a narrative structure that changes and evolves based on the user's choices, actions, and progress within the game. This approach can create a more personalized and meaningful learning experience as the story responds to the user's needs, preferences, and learning style. In educational video games, adaptive storytelling can offer different paths, challenges, and feedback depending on the user's performance, ensuring that the game remains engaging and relevant to the users' needs. In addition, this can lead to increased motivation, self-efficacy, and a sense of agency in the learning process.

2.5 Interviews

The algorithm questions used to test candidates in technical software interviews are not static, but although the questions change, the underlying concepts to solve these questions do not.

2.5.1 Data Structures, Algorithms, And Concepts

The below is from Cracking The Coding Interview [17]:

Data Structures	**Algorithms**	**Concepts**
Linked Lists	Breadth First Search	Bit Manipulation
Binary Trees	Depth First Search	Singleton Design Pattern
Tries	Binary Search	Factory Design Pattern
Stacks	Merge Sort	Memory (Stack vs Heap)
Queues	Quick Sort	Recursion
Vectors / ArrayLists	Tree Insert / Find / etc	Big-O Time
Hash Tables		

This table shows that the questions in technical software interviews are combinations and permutations of a finite list of concepts. Thus, when a person has a firm grasp of the above concepts, they will better understand exactly where each is best used. Therefore, they can more easily and efficiently solve the given questions.

2.6 Background Conclusion

In conclusion, the primary objective of this thesis is two-fold; it is to investigate the potential of gamification while also diving in to determine how educational video games can assist programmers in mastering advanced programming concepts to improve their performance in technical software interviews. The background literature reveals the potential of game-based learning tools in computer programming education. However, there is a need for further exploration of immersive educational video games that target advanced algorithmic concepts and their effectiveness in preparing computer science graduates for technical software interviews and their subsequent careers.

3 HardCode Design and Gameplay

Figure 1: HardCode Logo

3.1 Overview

HardCode is an educational video game targeted toward users who know the fundamentals of how to program but wish to deepen their conceptual knowledge of algorithms. The "Hard" difficulty studied in this thesis will be of significant use to assist users in preparing for technical software interviews. HardCode will teach users several methods to optimize algorithms efficiencies and why certain methods for writing some specific algorithms end up being strictly better than others. Theoretical concepts, such as Big-O, a concept used to calculate the efficiency of algorithms mathematically, will be taught throughout the game.

3.2 Learning Objectives

As previously referred to in 1.2 (Motivation) section of this paper, the users' learning objectives will reflect my own; how I taught myself how to become better at programming. The following learning objectives are designed to help programmers deepen their knowledge of algorithmic concepts and prepare for technical software interviews. Upon a successful playthrough of HardCode, users will be able to:

1. Understand various concepts to recognize the advantages of certain methods for writing specific algorithms.

2. Gain a grasp of Big-O notation to evaluate and compare the time complexity of algorithms, enabling informed decision-making when selecting optimal solutions.

3. Apply best practices to improve code quality and code efficiency.

4. Master the skills essential for tackling technical software interviews.

5. Build confidence through a simulated interview environment, empowering users to excel in real-world technical software interviews and distinguish themselves in the competitive job market.

These learning objectives will be achieved through a game-based educational experience that incorporates the following:

1. An extensive collection algorithm challenges that expose users to the defined concepts and their advantages.
2. Engaging examples exercises that familiarize users with Big-O notation and its applications in evaluating algorithm efficiency.
3. Feedback achieved through HardCode's time complexity algorithm.
4. A progression of increasingly difficult programming tasks that encourage users to apply and refine their problem-solving and critical-thinking skills.
5. A motivating and immersive learning environment that fosters the learning of algorithmic concepts and simulates the experience of technical software interviews.

3.3 Gameplay

HardCode is a single-player educational video game, a top-down 2D RPG in which users will walk around the cyberpunk-themed cities, talk to NPCs, and from them, accept "Jobs/Contracts" (algorithm challenges). These jobs will be of varying difficulties and will increase or decrease in difficulty depending on the user's current performance. Their current performance is based on the difference in their overall average efficiency to the expected. Once a user has completed enough jobs and gained enough experience, they will be allowed to face an in-game boss battle. This boss will give the user a timed algorithm that the user will need to complete at or below the desired efficiency, which will hopefully emulate a real-life technical software interview.

Although HardCode's "Hard" difficulty is intended for users who are studying for technical software interviews, the true scope of this difficulty is for users of all ages with prerequisite programming knowledge and a desire to become better at programming. There will be two more difficulties outside this thesis's

scope which will be future work, but they will be aimed at teaching the less advanced concepts and assuming the user has limited programming knowledge.

3.3.1 How The Four Narratives Were Incorporated Into HardCode

1. ***Distributed narrative***

HardCode's game-play style aligns with the distributed narrative by allowing players to play at their own pace, explore the world, and choose their desired algorithms. In the future, this narrative will continually be integrated into my game.

2. ***Endogenous fantasy and intrinsic integration***

The storyline of HardCode, which will be depicted in detail further below, will give a brief outline of how these factors will be integrated into the game's learning, thus motivating the user.

3. ***Empathetic characters and virtual agents***

This is accomplished in the tutorial scene at the beginning of the HardCode. An NPC is in the same position as you, unknowingly sent down by your boss into a wasteland. The NPC is angry at you; she assumes you are the person she was supposed to meet, but she has been waiting days and is unhappy with how long you've taken. They then figure out that you're just another victim and decide they must find their way out. You, the player, will meet that NPC later in the game.

4. ***Adaptive and responsive storytelling***

HardCode will directly integrate an adaptive learning environment by adjusting the difficulty of the algorithms given to the users depending on their past performances. Integrating helper NPCs using advanced language learning models is also worth consideration due to having the ability to tailor answers to virtually any question the users may have. They would be there to help users get on track but for an in-game cost.

Storyline: Free the Programmers

HardCode's overarching storyline delves into a dystopian future where the monopoly of software creation, leading to vast wealth gaps, brings upon a necessary and covert movement to overthrow the people in power: the Aristocrats. There are four Aristocrats, one per city (Math, Game, Data, Main), each with unimaginable wealth gained through the creation and distribution of software. With this wealth, they slowly purchased, corrupted, and gained control of the government, thus achieving their goal: to monopolize software creation. You are unknowingly hired to illegally undercut and work against these Aristocrats as part of a much bigger movement devised by your boss. This movement will set in place the dominos to dismantle the Aristocratic oligarchy and Free the Programmers! And you, the player, happens to be the one to make them all tip.

3.4 Game Design

The goal of every video game, and likewise every educator, is to emphasize engagement [10, 9]. For this reason, HardCode's genre will be Role Playing. Keeping track of users' statistics and allowing progression via tasks and leveling up will help create a more enjoyable and appealing learning environment. This, combined with an immersive storyline, nostalgic music and UI, and likable characters, will make HardCode deeply engaging, thus easier to learn from [26].

HardCode's idea began with the need to practice programming and algorithmic concepts, a means to push me beyond the limits of my knowledge. It was, and still is, for me to learn. My first idea was to have a game similar to Pokemon, but instead of Pokemon that you would stumble upon in the grass, it would be algorithm challenges. Users would go around and "battle" these Pokemon, gain experience, then learn over time. I imagined this game becoming extraordinarily frustrating once I thought about a user simply trying to get from one city to another while having to spend up to hours on multiple algorithm battles since

they would be randomly occurring.

From this original idea, I kept the concept of Gym Battles. These Gym Battles would be timed algorithms in which the user must complete an algorithm at or below a desired efficiency. With the new idea, the users would be able to walk around cities and have the ability to accept contracts of their choosing. In my mind, if the user were to be able to choose the algorithm they would like to accomplish rather than one randomly getting assigned, they would take their time and be much more methodical while completing the algorithm rather than wanting to get it done solely to move on.

Once the user has gained enough experience and completed enough algorithms, they will be called upon to face the Gym Battle. This Gym Battle would be timed and require the user to make the algorithm at the given efficiency.

None of this would work without the capability to execute software from inside a game; thus, this was my first task to accomplish. Python was the first language to become functional. Still, the user needed to install the Python executable on their computers and have the system environment variables set up correctly for code to execute in the game. The game's installation process would be much more tedious, given they would have to follow many steps; thus, I discarded the use of "real" Python and instead embedded IronPython through C#, restricting the use of external Python libraries.

3.5 Calculating Efficiency

One of the biggest hurdles during the creation of HardCode was developing an accurate and consistent way to calculate the efficiency of a user's algorithm. The first implementation for estimating efficiencies was through the use of literal time. The user's algorithm and the solution algorithm were fed random test cases. A timer would begin before the invocation of their algorithm, then stop once the function is finished executing. The same would occur with the solution algorithm. These timed values would be compared with the solution algorithms. Thus, since the solution algorithm's time complexity is known, it would provide a general estimate of the user's algorithm's efficiency. However, there was an issue: an algorithm with several separate for-loops, which technically has a time complexity of $O(n)$, was calculated to run "slower" than an algorithm with two nested for-loops, $O(n^2)$. This is because, on a regular desktop computer, it is unreasonable to use test cases with a magnitude large enough to negate these types of discrepancies completely. Another inaccuracy with this method was when using the same test case and code as input, different times and time complexities were returned. This was because one cannot assume the user's machine is static with no tasks running in the background. This method felt like it went around the problem instead of tackling it head-on; thus, a new idea was required.

In this section, there is a short outline of the methodology used in the time complexity algorithm currently implemented in HardCode.

Calculating Time Complexity

Problem statement

Creating an algorithm that calculates and returns the time complexity of arbitrary input algorithms (in Python). The reason for the creation of this algorithm is to fulfill two necessities: to help the user with feedback and to give HardCode the ability to rate users' competency.

Methodology

This algorithm, in some ways, works similarly to a compiler's lexical analyzer and parser in that it breaks down input code, removes any fluff, finds tokens, formats it in a way that makes it ready to execute, and places it inside an overarching time complexity algorithm. When the overarching time complexity algorithm is executed, the users' embedded code gets tracked. The number of steps it takes to come to completion is used to calculate its time complexity.

Step 0: Inside HardCode

We must begin preceding step 1 because it depends on external factors of HardCode. In HardCode, the process to submit an algorithm goes as follows:

1. Users must input test cases and compare the output of their algorithm against the back-ends. Once their returns match, they may choose to submit for further testing.

2. Once submitted, HardCode validates the users' algorithm by feeding up to one hundred randomly generated test cases, testing it as above for matches. If their algorithm is proven valid, the longest-timed test case is returned.

3. The time complexity algorithm takes two arguments: the longest-timed test case and the users' code. It then returns their code's time complexity.

Step 0.1: Test Case Generation

To create randomly generated test cases, an algorithm had to be designed that accepts bounds, or "blueprints," that outline valid input for a given algorithm, then returns test cases within those bounds. The following documentation outlines the test case generation algorithm along with the Regex used to parse the bounds:

```
Generates test cases according to specifications defined in Regex.

For integers:

- There are two possible input methods:
    The range of integer values. Example: 1..10
    Using an operator and an integer. Example: >10, <10, =10

Let NUM..NUM be RANGE.
Let {>, <, =} be COMPARISON_OPERATOR.
Let {+, -, *, /} be OPERATOR.
```

Type	Inputs	
Integers	Syntax	val: (RANGE | COMPARISON_OPERATOR NUM)
	Description	The number of test cases to generate. Usage: **val:>10, <10, =10, 1..10**
	Example	**int{val:1..100 amt:5}** : Generates 5 test cases of integers in the range 1 to 100
Arrays	Syntax	1d: (RANGE | COMPARISON_OPERATOR NUM) val: (RANGE | COMPARISON_OPERATOR NUM)
	Description	Determines the size of the array. Usage: **1d:=10, 1d:>5, 1d:<5, 1d:5..10**
	Example	**int{1d:=10 val:2..1000 inc:*10 order:desc}** : Generates an array of size 10 with values ranging from 2 to 1000, increasing by a factor of 10, then sorted descending
Matrices	Syntax	2d: (RANGE | COMPARISON_OPERATOR NUM) 1d: (RANGE | COMPARISON_OPERATOR NUM) 2d_nbyn: (RANGE | COMPARISON_OPERATOR NUM) val: (RANGE | COMPARISON_OPERATOR NUM)
	Description	Determines the size of the matrix. Usage: **2d:=3 1d:=1, 2d_nbyn:=3**
	Example	**int{2d:=3 1d:=2 val:<-100}** : Generates a 3x2 matrix of integers with values less than -100 **int{2d_nbyn:=5 val:10..20 order:asc}** : Generates a 5x5 matrix in range of 10 to 20 in ascending order
Functions	Syntax	inc: OPERATOR NUM order: ORDER amt: NUM
	Description	Increments the integer values. Usage: **inc:+5, inc:-5, inc:*5, inc:/5** Orders the values ascending or descending. Usage **order:asc, order:desc** The amount of total test cases. Usage: **amt:5**
	Example	**int{val:1..100 inc:+2 order:desc amt:5}** : Generates 5 test cases in range 1 to 100, incrementing by 2, then sorted descending

Step 0.2: Regex Design

The Regex created to handle parsing these inputs (reformatted for slightly better readability) is available in the Appendix, 5.1 Test Case Generation Using Regex.

Step 1: Formatting and removing fluff

This step removes any white space, print statements, comments, and et cetera so that we are left with the core of the program.

Step 2: Creating an overarching algorithm

This step will be broken down into sub-steps to better outline the methods used while solving time complexities. Below are issues that arose during the development and the chosen solutions.

Step 2.1: Recursive function finder

In Python, a function may contain more functions nested inside of it. A recursive function finder was developed to handle these cases. Thus, each of the below steps maintains its own scope for the function it is inside. This allows the time complexity of each nested function to be calculated as well.

Step 2.2: Inside a function

Once the above recursive algorithm finds a function, it follows these steps:

1. Place a static integer array named *functionName*IterationsInside on the first level of indentation and set it to increment itself by 1.
2. Iterate through the code of the function, trying to find (a) loops, (b) function calls, and (c) nested functions. If the algorithm finds any of those three, it has separate actions:
   (a) Loops (while, for): Add another element to the static array, placing it at the indentation level of the inside of the loop.

(b) Function calls:

i. If an external function is being called but not recursively, it begins a tree of calls: *functionInside* → *functionOutside*. These will become connected once the algorithm has come to completion, i.e., if *functionOne* → *functionTwo* and *functionTwo* → *functionThree*, then you must know *functionOne* → *functionTwo* → *functionThree*.

ii. If a function is called but has the same name as the current function it is inside, i.e., a recursive call, no action is taken. This decision will be explained further in a later step.

(c) Nested function: Recursively call this algorithm to begin this process again.

Below is a basic example of this algorithm finding loops inside of a function and adding an integer array, incremented, inside each loop:

```
# Original input/argument code before overarching
# time complexity algorithm:

def example(y):
    for x in range(y):
        for z in range(y):
            ...
    for i in range(y):
        ...

# After overarching time complexity algorithm formats
# the user's input algorithm:

def example(y):
    iteratorInsideExample[0]+=1
    for x in range(y):
        iteratorInsideExample[1]+=1
        for z in range(y):
            iteratorInsideExample[2]+=1
            ...
    for i in range(y):
        iteratorInsideExample[3]+=1
```

...

Step 2.3: Internal function calls to external functions

As mentioned above, a tree of all called functions is created, and if a function calls another, its max() iteration value will be multiplied down the chain until it reaches the root function. For recursive functions, the algorithm will view them as an ordinary loop and take its time complexity as usual:

```
# I.e., these two functions have the same time complexity
def factoralRecursive(x):
    if (x == 1):
        return x
    return x * factoralRecursive(x-1)

def factoralWhile(x):
    y = 1
    while (x > 1):
        y *= x
        x-=1
    return y
```

Although one of these functions is recursive, both are treated as the same function regarding their time complexity.

Step 3: Executing the overarching algorithm

The above is a small example. Once it has run, and since we know the input test cases' value (from the generated test cases in step 0), we may compare it to the greatest iteration value amongst the *iteratorInsideExample* array.

Step 3.1: Finding the time complexity

Let us say that the input is $y = 5$. Once we take the max() of *iteratorInsideExample*, you will find the maximum is 25. So compared to the original input value, the relation ends up being 5^2, precisely what we wanted!

Now we may use a function that takes the input value as the length of the input and the maximum value in the array and do something like this:

```
def timeComplexityAsString(length, maxValue):
    returnValue = ""
    val = maxValue
    while (maxValue > math.log(val)):
        if (maxValue >= length^2):
            maxValue //= length^2
            returnValue += "n^2"
        elif (maxValue >= length):
            maxValue //= length
            returnValue += "n"
        elif (maxValue >= math.sqrt(length)):
            maxValue //= math.sqrt(length)
            returnValue += "sqrt(n)"
        elif (maxValue >= math.log(length)):
            maxValue //= math.log(length)
            returnValue += "log(n)"
    return f"O({returnValue})"
```

This function would run and return $O(n^2)$. Therefore, we have found the time complexity of this simple algorithm.

Conclusion

Of course, from a theoretical point of view, it is not possible for this program to find the time complexity for all algorithms due to the ramifications of it being a subset of the halting problem, thus undecidable. Although, while thinking deeply about undecidability, one could argue that the aforementioned impossibility pertains only to the implications of creating an omniscient program (i.e., solving twin primes conjecture). However, if a scope is given to the program and acknowledging that it cannot be omniscient, the belief is that accurately computing time complexities is achievable.

Future work

Implementing machine learning AI into the test case algorithm instead of randomly choosing test cases. I.e., intentionally finding the worse test case, then using the time complexity algorithm. This will yield more accurate and reliable results.

Handling more complex relations between the input length and the maximum iteration value. I.e., if $x = 5$ and $y = 25$, you cannot simply assume that $y = x^2$. This may be solved through symbolic regression for discovering parametric equations [28].

3.6 UI Design

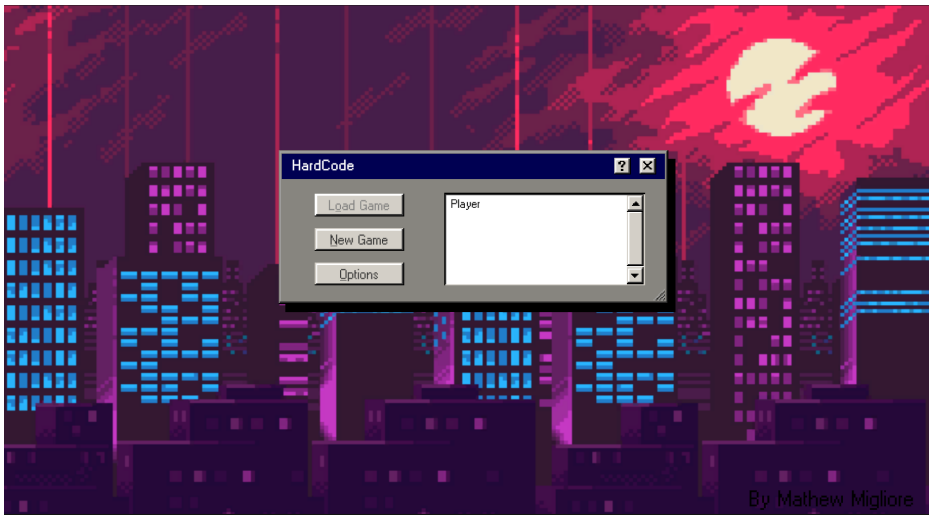

Figure 2: Load Screen

HardCode's UI is intended to look like Windows 98 but in dark mode. This is the style of UI the user will be given at the beginning of the game because the user is "poor." However, the user will be able to upgrade their UI with in-game money to make it more modern.

3.6.1 Abacus

The Abacus is where users write their code, read their emails, and look at their statistics. HardCode was built around the Abacus. Months were spent solely on its development; no other part of HardCode had any significance without its functionality. When thinking about it, there is so much packed inside the Abacus that sometimes it feels like its own OS, like its own separate project, and like it has no logical reason to be part of any video game whatsoever. The user can tap into a whole other application: execute code, run test cases, and get their algorithm's time complexity just by pressing E. I wish to express the magnitude of effort applied, knowledge gained, enjoyment, and frustrations I had while developing this piece of software, but, alas, writing this thesis is the best I can do. The enjoyment of seeing such a large piece of software slowly pieced together like a puzzle is a magnificent sight. Of course, there were frustrations at every turn, but if you view it as a whole rather than individual pieces, you begin to enjoy the struggle.

The Abacus has seven tabs for the user to explore, each with different uses. The users will be able to slowly build upon the Abacus's functionality, making it closer to an IDE rather than a regular text editor. Some upgrades will include auto-tab, coloured variables, and auto-complete. Below, we will go through each of these tabs, briefly explaining their uses and functionality.

Jobs Window

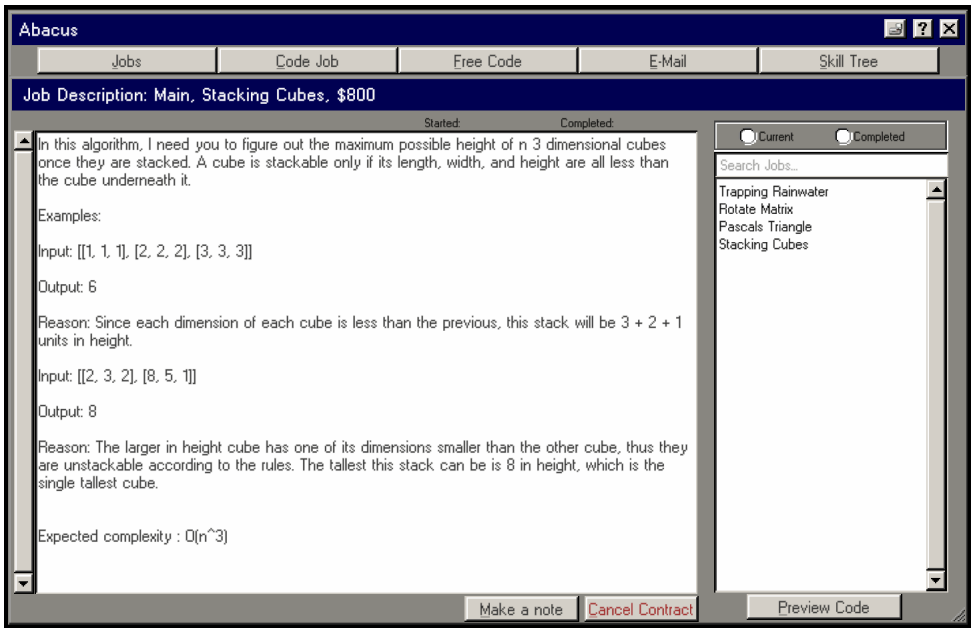

Figure 3: Jobs Window

The Jobs window is where users can view the description of their accepted and completed jobs/contracts, take notes, and can load the code into the Code Job window. The list to the right has two options: Current and Completed, which sorts the jobs based on if they have been completed or not. When a user clicks on a job in the list, it loads the title and all of its data, ready for the user to begin programming.

Code Jobs Window

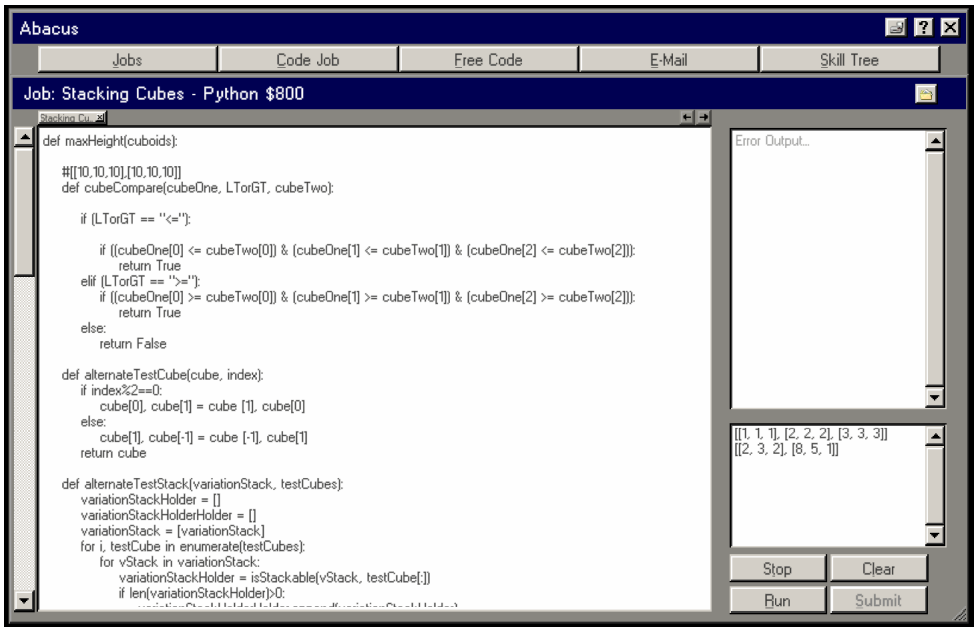

Figure 4: Code Jobs Window

The Code Jobs window is where the user writes their code. Their code can be loaded from the Jobs window or the folder button at the top right of the Code Jobs window.

There are two UI text inputs inside this window: the large text box in the center where the user writes their algorithms and the smaller text box at the bottom right where the user writes their test cases.

This tab has four buttons: the Stop, Clear, Run, and Submit buttons. The Run and Submit buttons are inactive until the user loads the code into the editor. The Submit button remains inactive until the user runs passing test cases they have entered. Once the Submit button is pressed, the user will either receive an email showing the acceptance and the user's code's time complexity or an Email saying that a test case failed, in which the test case gets automatically placed into the test case box in the user's Abacus.

Free Code Window

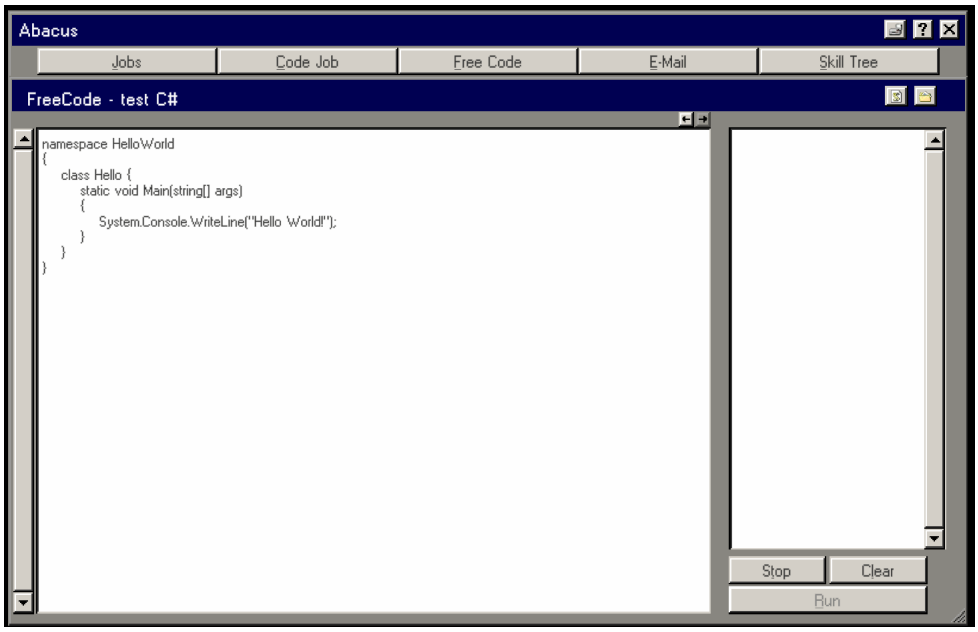

Figure 5: Free Code Window

The Free Code window is where the user may write code that is not correlated with the jobs they have. They are not required to use test cases. They can create classes and, thus, whole applications inside the free code area. This will likely be the window where the user cracks the codes their boss sends them.

Email Window

Figure 6: Email Window

The Email window is where the user receives notifications about any instructions, if any contracts were completed, and the details. In addition, there is a read and unread tab; once the user clicks an unread Email, it goes into the read tab. An idea that will be implemented in the future as part of the storyline: the user will begin receiving secret codes from their boss that they are required to crack using their masterful algorithm knowledge. One of the codes your boss sends you to crack will say "Meet me in this location at this time; tell no one." It was meant for you to crack this code...

Skill Tree Window

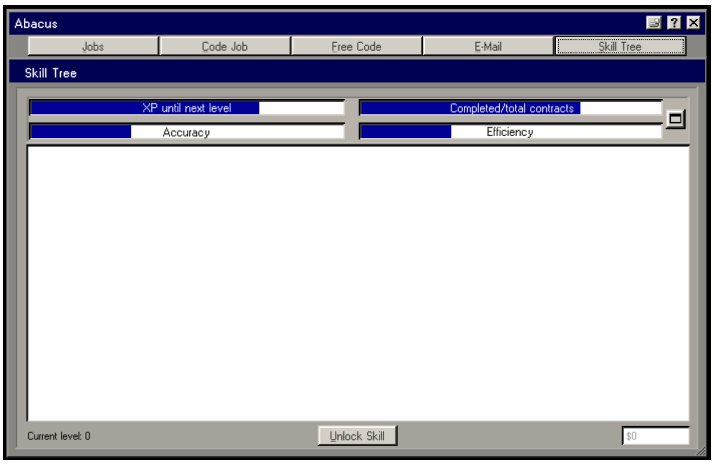

Figure 7: Skill Tree Window

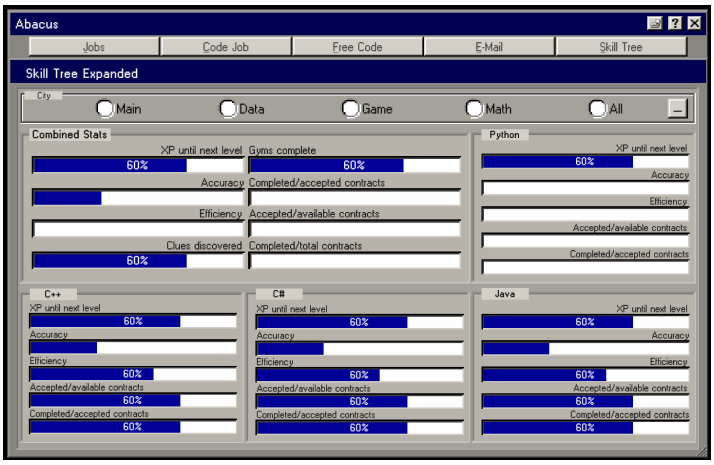

Figure 8: Statistics Window

The Skill Tree window is a dual window. It shows the user's statistics along with the skill tree. The skill tree is where users can unlock abilities that may help them throughout the game. There is a full-screen button beside the compact version of the statistics, which opens a more detailed view showing various categories by language and city.

Backup Save System Window

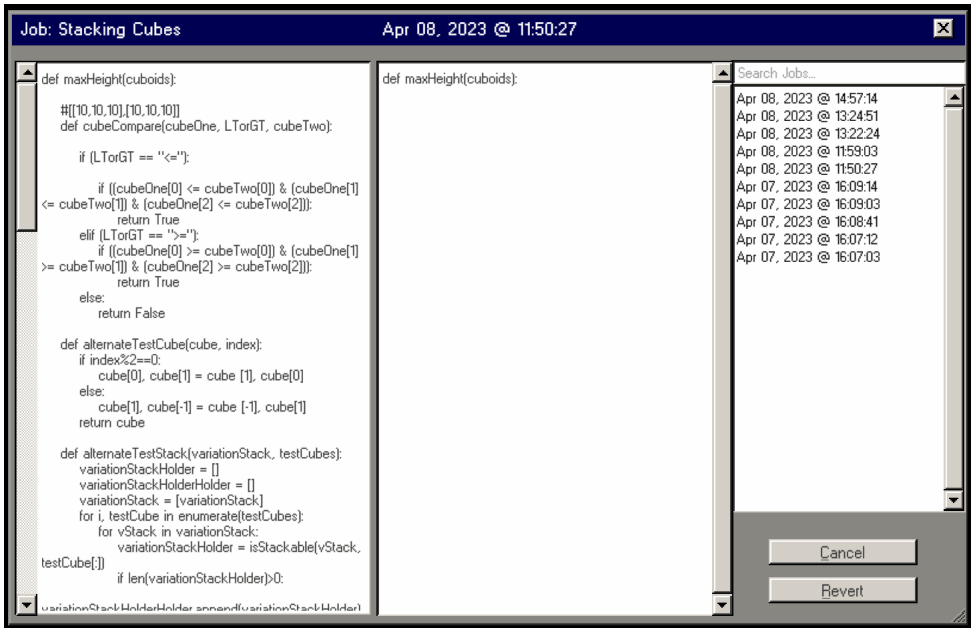

Figure 9: Backup Save System Window

The Backup Save System window shows all of the user's current and completed jobs, and once the user clicks on one, it shows backups that have been saved for those jobs. The user can revert to past pieces of code through this window. The Abacus backs up code whenever the user clicks "Run" in the Code Jobs window.

Help window

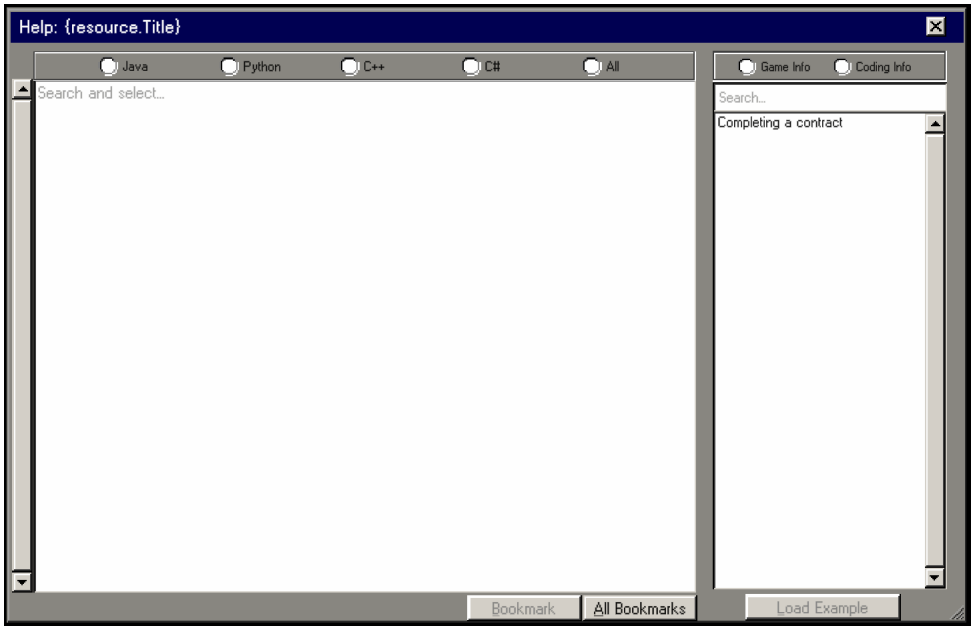

Figure 10: Help Window

The Help window shows resources, information about the game, tips on using the Abacus, and coding and algorithmic tips showing specific techniques and giving examples of their inputs and output. There are two tabs on the right side with the search area: "Game Info" and "Coding Info." Through "Game Info", the users can search for details and information regarding any of HardCode's in-game mechanics; through "Coding Info," the users can find resources on algorithmic concepts.

3.7 World Design

In the usual sense, there are no levels in HardCode. The users are free to roam around the city, speak to NPCs, and choose their own contracts. The only time total freedom is not applicable is while the user is in a mini-boss battle: the contract is given to the user without any prior knowledge of what it may be; this is to emulate a real-life technical software interview. Below, the seven areas currently implemented in HardCode will be discussed.

3.7.1 Opening Scene

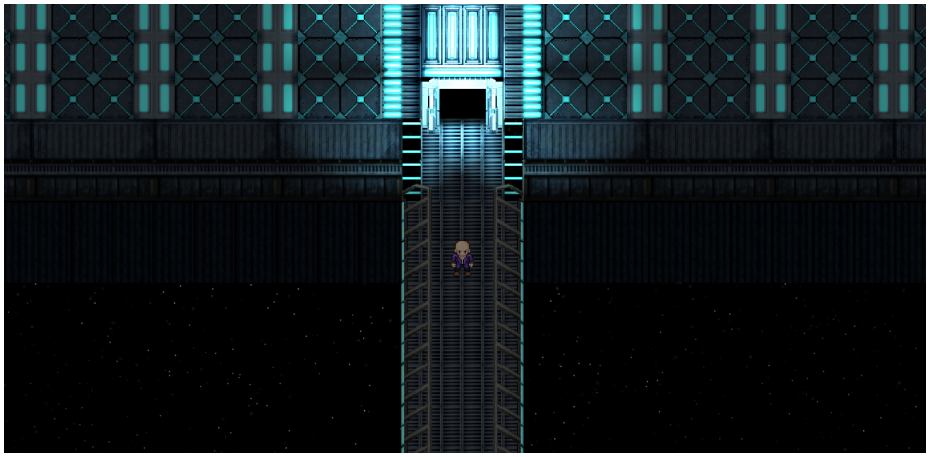

Figure 11: Opening Scene

This is the first place users see after they begin a new game. It fades in from black, just teleported to the entranceway of a massive spaceship in a secret and secluded area of deep space, the stars moving slowly in the background and the lights of the spaceship flashing bright; the music is a big textured synth bass.

3.7.2 Introduction

Figure 12: Introduction Scene

Once the user enters the spaceship, they are greeted by Avacyn, their new boss. She tells the user how magnificent of an opportunity this is for them and how wealthy they will be in the future. Next, she gives the user an option of four languages to choose from and tells them to choose whichever one they feel the most comfortable with. This is another play on Pokemon, choosing your first Pokemon at the beginning of the game, although the player will be able to upgrade and purchase new languages as the game progresses and in future work. The four languages that will be included are C#, C++, Java, and Python, all of which the user will have to learn and use. This adds to the real-life applicability of what the user gets out of the game, as they will become comfortable with some of the most prominent languages in software.

3.7.3 Grounded

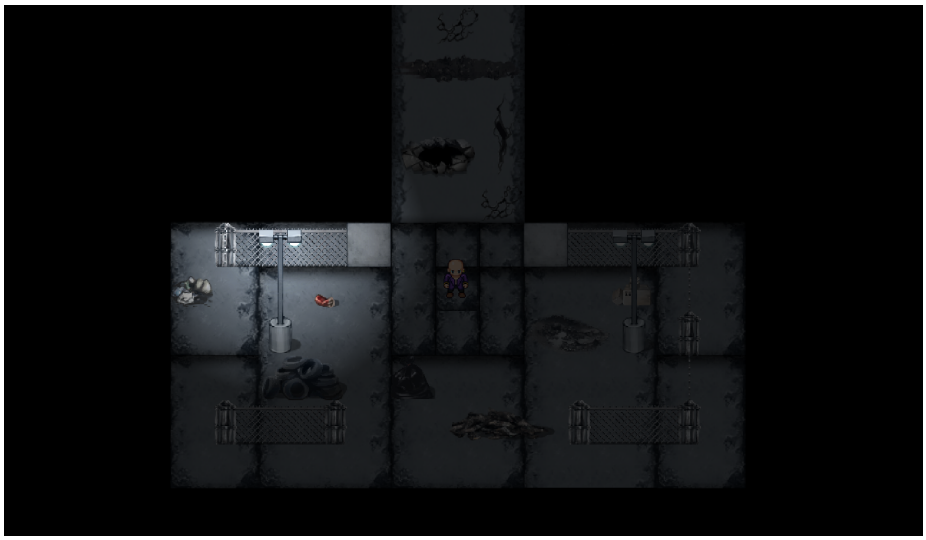

Figure 13: Grounded

The user is then teleported down to a wasteland, dark and dreary, full of garbage, and unmaintained roads. The user is brought back to reality, grounded. This is the beginning of the feelings of being lied to by their boss. The user has yet to learn about the storyline; thus, the contrast between the scenes may come as a shock to them. As they continue to walk forwards, there is a house with graffiti on its walls and lights on. That is the "Safehouse" and is also when the tutorial begins.

3.7.4 Main City

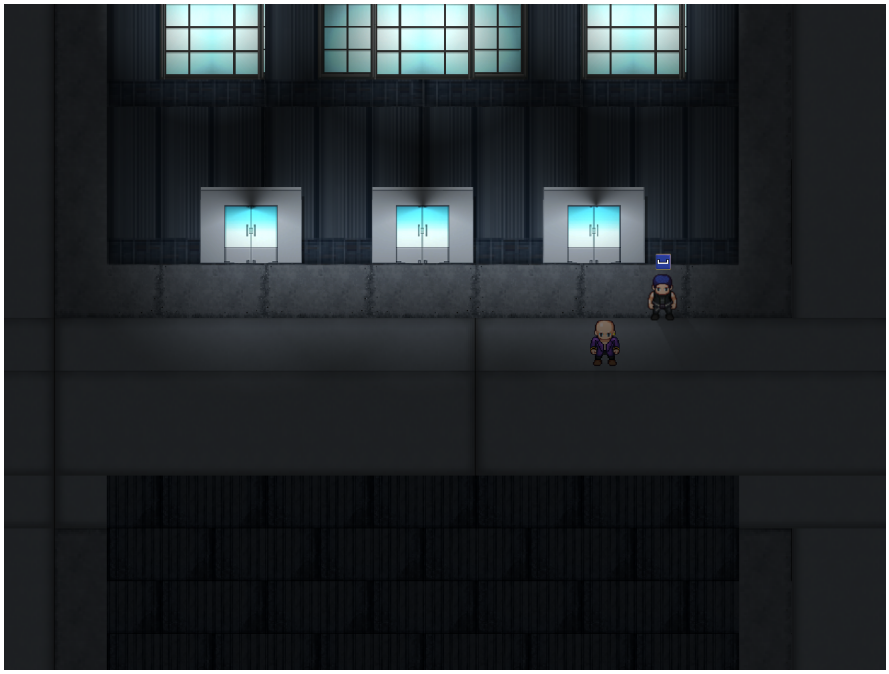

Figure 14: Main City with NPC

The user is brought to the Main City, where they are told to accept contracts and begin making money, given some more information by the NPC giving the tutorial, then are set off on their own. In Figure 14, the user has walked close enough to the NPC that it stops and offers an option to press space to interact. Once the user sees the difference in wealth and cleanliness between the Main City and the previous locations, they should gather a better idea of the society's state in the game and where they lie. On the next page is a broader view of the Main City and what will be added in future work.

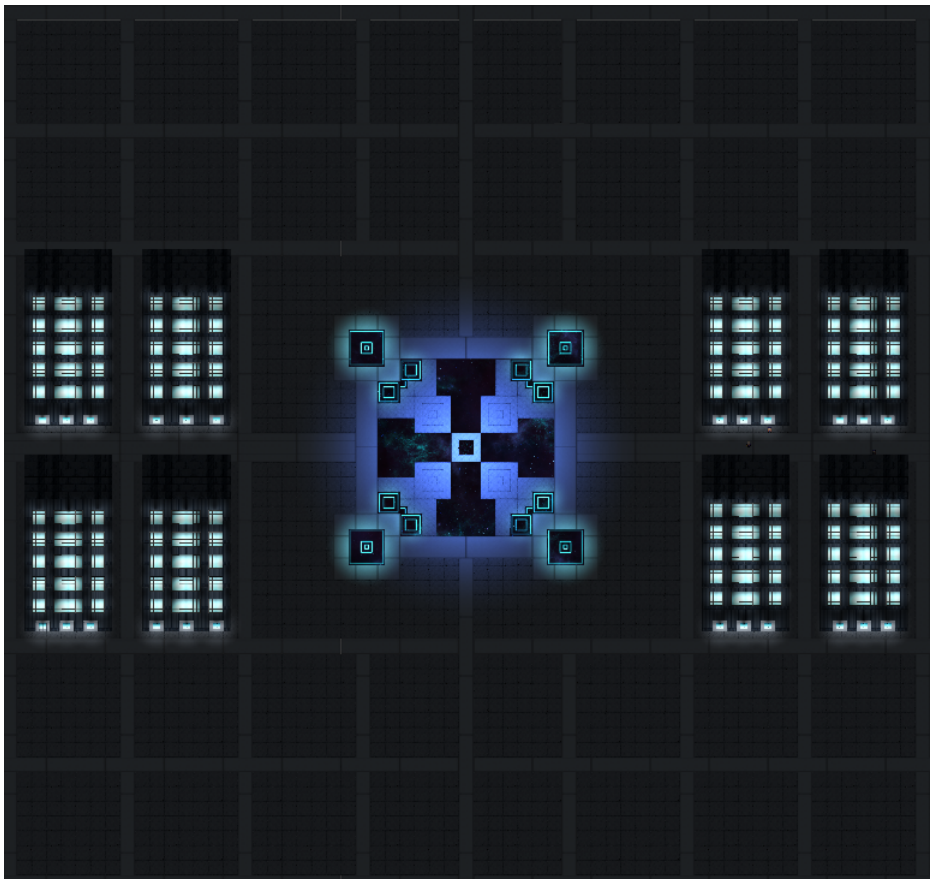

Figure 15: Main City True Size

The true scale of the Main City is quite large; each of the plots/squares viewed in this picture will be filled with buildings that the user can enter, each with different NPCs and challenges the users can accept. There will also be mini-game activities in some of the buildings; I would like to incorporate a casino into the Main City, an arcade into the Game City, board games and chess-style games into the Math City, and (as a joke) a data collection center into the Data City where people enjoy going and writing surveys, giving data. The first three would have classics, card games, vintage arcade games, and board games.

3.7.5 Safehouse

Figure 16: Safehouse

This is the Safehouse the user enters from the "Grounded" location. Once the user enters the Safehouse, they are met with a distressed NPC, then the tutorial begins. The lit-up server at the back of the room is the secret entrance-way to The Underground, which will be discussed later in this section. The dirty, unkemptness of the Safehouse will hopefully solidify the idea of the user being lied to by the boss, Avacyn.

3.7.6 The Underground

Figure 17: The Underground

If the user enters the secret server entrance shown in Figure 16, they will be brought to The Underground. This is a slum where all the undercutting people go as a place of safety. It is also where the Boss Battle is located. The secret entrance opens only if the user is above a certain level and has been explicitly invited. Winning against this Boss will be the users' first triumph of many, good or bad; it is an accomplishment that will make a name for themselves in The Underground.

3.7.7 Boss Battle

Figure 18: Boss Battle

Once the user completes enough contracts, they will be invited to face Boss Kristjan through an Email. The user will then need to complete his algorithm challenge in the given time and under the specified efficiency, which is the attempt to emulate a real-life technical software interview. Kristjan will be in disbelief when the user beats him; he will pay the user, then will ask them kindly to leave.

3.8 HardCode Design Conclusion

The goal throughout the creation of HardCode was to make an enjoyable, rich atmosphere where the user could get lost for hours, days, and weeks on end. An educational video game is successful in its creation if it incorporates the aspects of education without hindering the game's playability, where the users progress and learn without feeling forced to, only to look back and see all they have accomplished. The goal is to achieve a desire in users to play HardCode, for them to want to hear the music, be in the atmosphere, the cities, and interact with the NPCs, work on their jobs, learn as much as they can, then become masters at algorithms.

4 Conclusion

4.1 Summary

In conclusion, this thesis began the deep dive into the possibilities of game-based education by creating HardCode, an educational video game designed to help users learn and master algorithmic concepts. HardCode aims to begin filling the literature gap, focusing on game-based learning effectiveness in the context of algorithm education. Furthermore, this study will contribute to expanding knowledge on game-based learning in computer science education by investigating how game-based education can create a fun and motivating learning environment.

The efforts of this research will offer valuable information on the potential benefits of using game-based learning tools like HardCode to enhance the learning experience and outcomes for students struggling with complex algorithmic concepts. In addition, the hope is for this study, however small, to act as a basis for future research on the efficacy of game-based education in readying computer science graduates for technical software interviews and their subsequent career success.

4.2 Limitations

The context behind the limitations: *The limitations of this study were time and resources. This is an undergraduate thesis; thus, I could not focus all my efforts on this study or development of HardCode.*

There are many limitations of HardCode at its current level of development:

- It is not a web-based game. Users must download an exe to play on their local machine. Although I have purchased the domain of HardCode.ca, this may change in the near future.

- It currently only works on Windows machines, limiting the demographics of possible learners.
- It is only playable for users who know how to program.
- There are far from enough in-game learning resources. Therefore, resources for in-game, HardCode-related information and general algorithmic concepts will be added.
- Many more algorithm challenges need to be implemented.
- The storyline needs to be implemented.

The game is playable as a demo of its functionality rather than a fully fleshed-out learning resource. Nevertheless, it has been an intensely challenging and rewarding solo project which will need years more development time to complete, and the goal is completion.

4.3 Future Work

Future research could explore various game design elements and user demographics to refine and further broaden the comprehension of how game-based education can best help learners in the computer science field. Additionally, longitudinal studies could be carried out to investigate the long-lasting effects of gamification on learners' algorithmic understanding, skill development, and career paths.

5 Appendix

5.1 Test Case Generation Using Regex

```
int\{
    (?:2d
        (?<group_matrix>
            (?<group_nbyn>(?:_nbyn:(?:
                (?<group_nbyn_operator>
                (?<nbyn_dim_comparison_operator><|>|=)
                (?<nbyn_dim_value>\d+))
                |
                (?<group_nbyn_range>
                (?<nbyn_dim_range_first_value>[+\-]?\d+)\.\.
                (?<nbyn_dim_range_second_value>[+\-]?\d+))))?
            )?
            |
            :(?<group_nbym>
                (?<group_nbym_operator>
                (?<nbym_dim_comparison_operator><|>|=)
                (?<nbym_dim_value>\d+))
                |
                (?<group_nbym_range>
                (?<nbym_dim_range_first_value>[+\-]?\d+)\.\.
                (?<nbym_dim_range_second_value>[+\-]?\d+))
            )
        )?
    )?
    (?:1d:
        (?<group_array>
        (?<group_array_operator>
```

```
        (?<array_dim_comparison_operator><|>|=)
        (?<array_dim_value>\d+))
        |
        (?<group_array_range>
        (?<array_dim_range_fist_value>[+\-]?\d+)\.\.
        (?<array_dim_range_second_value>[+\-]?\d+))
        )?
    )?
    val:
    (?<group_integer>
        (?<group_integer_operator>
        (?<integer_comparison_operator>>|<|=)
        (?<integer_value>[+\-]?\d+))
        |
        (?<group_integer_range>
        (?<integer_range_first_value>[+\-]?\d+)\.\.
        (?<integer_range_second_value>[+\-]?\d+))
    )
    (?: inc:(?<group_increment>
        (?<increment_operator>[+\-\*\/])
        (?<increment_value>\d+))
    )?
    (?: order:
        (?<order>asc|desc)
    )?
    (?: amt:
        (?<quantity_of_test_cases>\d+)
    )?
\}
```

References

[1] Codefight. https://codefights.com/. Retrieved October 2022.

[2] Robocode. https://robocode.sourceforge.io/, 2001. Retrieved October 2022.

[3] Codecombat. https://codecombat.com/, 2013. Retrieved October 2022.

[4] Codeingame. https://www.codingame.com/start, 2013. Retrieved October 2022.

[5] Leetcode. https://leetcode.com/, 2015. Retrieved October 2022.

[6] Algoexpert. https://www.algoexpert.io/product, September 2017. Retrieved October 2022.

[7] C. Carpio. Gamification of software engineering education: An exploration, March 2022. Retrieved September 2022.

[8] B. Choi and Y. Baek. Rethinking fantasy as a contributor to intrinsic motivation in digital gameplay. ScholorWorks, January 2013. Retrieved September 2022.

[9] Nina Dorfner and Rana Zakerzadeh. Academic games as a form of increasing student engagement in remote teaching. 2021. doi: 10.1007/s43683-021-00048-x. URL https://link.springer.com/article/10.1007/s43683-021-00048-x.

[10] Gallup. Focus on student engagement for better academic outcomes. Gallup.com, 2021. Retrieved October 2022.

[11] M. P. J. Habgood, S. E. Ainsworth, and S. Benford. Endogenous fantasy and learning in digital games. *Simulation & Gaming*, 2005.

[12] C. Haynes-Magyar and N. Haynes-Magyar. Codespec: A computer programming practice environment. In *Proceedings of the 2022 ACM Conference on International Computing Education Research*. ACM, August 2022. Retrieved October 2022.

[13] A. Ibrahim, M. Mahfuri, N. Abdallah, and H. Alkhazaleh. Using playability heuristics to evaluate player experience in educational video games. ResearchGate, January 2020. Retrieved September 2022.

[14] V. KarlStoffová. Educational computer games in programming teaching and learning. 2019. Retrieved October 2022.

[15] M. Lineberry, Y. S. Park, D. A. Cook, and R. Yudkowsky. Making the case for mastery learning assessments. AAMC, November 2015. Retrieved October 2022.

[16] C. Malliarakis, M. Satratzemi, and S. Xinogalos. Educational games for teaching computer programming. In *2014 6th International Conference on Games and Virtual Worlds for Serious Applications (VS-Games)*. IEEE, June 2014. Retrieved September 2022.

[17] G. L. McDowell. Cracking the coding interview: 150 programming questions and solutions. 2016.

[18] M. A. Miljanovic and J. S. Bradbury. *A Review of Serious Games for Programming*. Springer, 2018. Retrieved July 2022.

[19] E. Naul and M. Liu. Narrative in educational video games. *Journal of Educational Computing Research*, July 2019. Retrieved September 2022.

[20] Emily Naul and Min Liu. Why story matters: A review of narrative in serious games. *TechTrends*, July 2019. Retrieved October 2022.

[21] M. A. F. Randi and H. F. de Carvalho. Learning through role-playing games: an approach for active learning and teaching. *Revista Brasileira de Educação Médica*, February 2013. Retrieved October 2022.

[22] K. Royle and R. Clarke. Making the case for computer games as a learning environment. January 2003. Retrieved September 2022.

[23] J. Swacha, R. Queirós, J. C. Paiva, J. P. Leal, S. Kosta, and R. Montella. A roadmap to gamify programming education. *AALBORG UNIVERSITET*, 2020. Retrieved October 2022.

[24] Qurat ul Ain, F. Shahid, M. Aleem, M. A. Islam, M. A. Iqbal, and M. M. Yousaf. A review of technological tools in teaching and learning computer science. *EURASIA Journal of Mathematics, Science and Technology Education*, April 2019. Retrieved September 2022.

[25] C. Watson and F. W. Li. Failure rates in introductory programming revisited. *Journal of Information Technology Education: Research*, 2013.

[26] A. C. Wilkinson and S. M. Brussow. Engaged learning: a pathway to better teaching. *South African Journal of Higher Education*, January 2010. Retrieved October 2022.

[27] M. Wynn and D. Bouchard. Hello code: An active programming video game. ProQuest, October 2018. Retrieved October 2022.

[28] Michael Zhang, Samuel Kim, Peter Y. Lu, and Marin Soljačić. Deep learning and symbolic regression for discovering parametric equations, July 2022.

[29] S. I. Zinovieva. The use of online coding platforms as additional distance tools in programming education. *Journal of Physics: Conference Series*, 2021. Retrieved October 2022.

[30] E. Y. İnce. Students' perceptions on learning programming with codingame. *International Journal of Technology in Teaching and Learning*, 2021. Retrieved 2022.