A visual solver for fair division: Adding state-of-the-art-algorithms

Yongteng Lei

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2025

Abstract

The divisible fair division problem, also known as cake-cutting, has been an active research topic across several fields for nearly 70 years. Its core task is to allocate a divisible resource, often referred to as a cake, to n agents in a fair manner according to their preference values, typically based on the widely accepted fairness criterion of envy-freeness. The cases involving two or three agents have been elegantly solved by the Cut & Choose and the Selfridge-Conway Method, and these have been integrated into Fair Slice, a visual fair division tool primarily used for educational purposes. Despite decades of intensive research, the problem of fair division for four or more agents remains challenging. Recently, Hollender and Rubinstein [12] proposed an ε-envy-free fair division algorithm that fairly divides a cake into four pieces in the four-agent scenario within $O(\log^3(1/\varepsilon))$, using only three cuts, allowing agents to divide the cake within a small ε factor without envy. The goal of this project was to thoroughly explore the Hollender-Rubinstein algorithm, translate it from theory into practical code, integrate it into Fair Slice, and maintain its user-friendly and educational characteristics. This goal was ultimately achieved, though with some limitations, such as the need to further enhance the robustness of the implementation and the algorithm's reliance on users having specific preferences for successful operation. Additionally, the gap between implementation and theory has been discussed, and deeper research and more comprehensive testing are identified as directions for future work.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Yongteng Lei*)

Acknowledgements

I would like to give my sincerest gratitude to my supervisor, Aris Filos-Ratsikas. Two years ago, I read his paper *Fair Division of Indivisible Goods: Recent Progress and Open Questions*, and entered the world of Fair Division. It was a great honour to do this work under his supervision.

I am grateful for the book *Economics and Computation: An Introduction to Algorithmic Game Theory, Computational Social Choice, and Fair Division*, which provided me with extensive knowledge about Fair Division and helped me realise that I am not alone in my interest in this field. It is such a fascinating field. This book deepened my understanding and reinforced my passion for Fair Division.

Thanks to my friends, Yuwei Zhang, Rutong Li, Shuyuan Pang, they are my family.

Thank you to my dearest beloved brother from Pakistan - Dera Malik Nawaz Walasar, Asif Nawaz (Yes! he asked). We once spent an entire night working together on my project, and it was a memorable experience. We also shared time together during our ordinary days, supporting each other along the way.

My deepest gratitude to my mum, who was always there for me during my most difficult times. I am also profoundly thankful to my dad for his unwavering and unconditional support.

My heartfelt thanks to my dearest grandma.

Thanks to my own love for Fair Division, or I would say Fair Allocation, which has made these months of work enjoyable and fulfilling.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

"Justice is blind, and fairness requires anonymous rules of arbitration." [14] The development of fair division algorithms, which promote fairness and trust by removing bias and ensuring that the distribution process is both transparent and equitable, has been an important topic of research in the fields of mathematics, economics, politics, and society from diverse perspectives for over 70 years [14, 16].

Fair division algorithms have broad significant applications in real-world scenarios. Some practical fields are in real estate projects [27], power distribution [4], and land redistribution [21, 22]. In the international context, the Adjusted Winner algorithm, proposed by Brams and Taylor [7, 8], played an important role in the Panama dispute [18]. This marked the first instance that fair division algorithms demonstrated significant potential in real-world problem solving. Furthermore, the application of these algorithms in resolving divorce settlements [17], managing shift allocations [15], and facilitating equitable rent division [10, 24] highlights their practical relevance and effectiveness in addressing everyday life challenges. Therefore, it is a necessary topic to study the application of fair division algorithms.

After several decades of intensive research, scholars have made substantial progress in this field, yet they continue to encounter numerous challenges and uncertainties. In scenarios involving a small number of agents, these algorithms can efficiently and elegantly resolve disputes. For instance, the "Cut & Choose" protocol, a simple and effective method, ensures fairness in two-agent situations by having one agent cut and the other choose her preferred portion. Conversely, the Selfridge-Conway [7] method only requires several cuts to achieve a fair division, by ensuring each party receives at

least two pieces. However, when dealing with four or more agents, due to the complexity of the algorithm, the necessity for potential infinite queries [2], or the impracticality of results, the cake may be divided into million pieces [23], making such scenarios still disconnected from practical real-life applications or unrealistic.

Recently, Hollender and Rubinstein [12] proposed an implementable approximate envy-free framework with considerable complexity, featuring the advantageous property that with only $n-1$ cut points, agents can achieve a continuous allocation of cakes in $O(\log^3(\frac{1}{\varepsilon}))$, where ε is the factor of approximation. Moreover, thanks to the work of Ernst [11], undertaken as his MSc project at the School of Informatics, University of Edinburgh, during the academic year 2022/23, Fair Slice emerges as the first fully realised visual fair division tool. This tool effectively brings scenarios involving two and three agents to public attention, serving as a valuable educational resource. In addition, its well-designed, interactive educational pages enhance its utility as an effective presentation tool for educational purposes. Building on these inspiring work, the aim of this project is to explore fair allocation algorithms for scenarios involving four or more agents. Specifically, this project focuses on the work of Hollender and Rubinstein [12], converting it into practical code, and integrating it into Fair Slice. This extension enhances the functionality and responsibility of Fair Slice, enabling it to address more realistic scenarios and provide visual educational presentations for more complex situations.

1.2 Main Contributions

While Ernst [11]'s work primarily addresses interface design and user experience, this project focus more on the exploration and analysis of fair division algorithm proposed by Hollender and Rubinstein [12], particularly emphasising its potential implementation. As the four-agent fair division algorithm remains mysterious, it brings significant complexity and challenges to its implementation. Furthermore, integrating the algorithm with interactive educational pages is also a key focus of this work. The following are the main contributions of this project:

- The core codebase, originally written in TypeScript and implementing the Cut & Choose algorithm, has been refactored in Python to enhance maintainability and extensibility, laying a solid foundation for the future development of this project.
- The Hollender-Rubinstein algorithm has been thoroughly explored and analysed,

with its core concepts, implementation details, challenges, and solutions clearly highlighted. This work serves as a bridge between theory and practice, transforming the algorithm from a theoretical framework into practical, executable code.

- Furthermore, the Hollender-Rubinstein algorithm has been successfully integrated into the existing Fair Slice platform, extending its functionality to accommodate four-agent scenarios. An interactive course, consistent with the platform's interface, has also been developed, establishing Fair Slice as a valuable tool for educational presentations and demonstrations.

1.3 Dissertation Roadmap

The structure of this dissertation is as follows:

Chapter 2 lays the foundational background necessary for understanding fair division. It begins with the preliminaries of cake-cutting, including the problem definition, evaluation equations, and fairness criteria. This is followed by a historical overview of fair division, along with the challenges and opportunities encountered in four-agent scenarios. Subsequently, the chapter introduces the Hollender-Rubinstein algorithm, offering a broad overview to grasp the core concepts before proceeding to a detailed analysis. The chapter then concludes with a brief introduction to Fair Slice, outlining its core features, key highlights, and how our work can integrate with and extends the platform.

Chapter 3 presents the Hollender-Rubinstein algorithm in a more technical manner, transitioning from theory to practice and from problem identification to solution development. It offers a comprehensive perspective on the analysis and explanation of the algorithm, including the transplanting of the original evaluation functions from the TypeScript codebase, the rationale behind the modified evaluation functions, and the necessary adaptations required to align them with the Fair Slice platform.

Chapter 4 begins with a discussion of the performance of the implemented algorithm and the challenges it faces. It examines the impact of various factors on the algorithm from three perspectives: the influence of agent preferences on the satisfaction of invariant, the challenges posed by floating-point computation due to precision and tolerance issues, and the consequences of potential implementation errors and decisions. Subsequently, the results of the integration with Fair Slice are presented. Finally, the

chapter addresses the project's limitations and outlines future work.

Chapter 5 concludes the key information of the project and gives a review of the entire dissertation.

Chapter 2

Background

2.1 Preliminaries of Cake-Cutting

2.1.1 Problem Formal Definition

This project is devoted to investigating algorithms for the fair division of **divisible** resources among four agents, commonly referred to as cake-cutting. The cake is metaphorically represented by a real number unit interval, $X = [0,1]$, which is sufficient to represent any infinitely divisible resource that can be divided among n agents. The purpose of the task is to cut the cake X into n portions, $X_1, X_2, \cdots, X_n$, out of n agents, $a_1, a_2, \cdots, a_n$, and assign these portions to the agents in a fair way, for $1 \leq i < j \leq n$, each portion X_j satisfies $X_j \subseteq X$ and we have $X = \bigcup_{j=1}^{n} X_j$ and $X_i \cap X_j = \emptyset$. Ideally, only $n-1$ cuts would be needed to obtain continuous pieces of cake portions, which is a significant property we desire. However, this is not necessary. Each portion $X_i \subseteq X$ can be considered as a collection of disjoint, thus more than n pieces could be made.

2.1.2 Valuation Functions

Fair division algorithms barely know anything about agents and primarily operate based on their expressed preferences through valuation functions. For $1 \leq i \leq n$, the valuation function v_i maps agent a_i's preference for each piece of cake X to some real number in $[0,1]$. Each agent is evaluated independently, knowing only their own valuation for each piece of cake according to the corresponding function. Additionally, for any subset of $X = [0,1]$, which can be evaluated differently by the same agent, such a cake is considered *heterogeneous*. More specifically, when all agents a_i evaluate the entire cake identically, the cake is deemed *homogeneous*, a simplified cake-cutting problem.

Valuation functions, as a tool for representing agent preferences, are assumed to have the following four properties in this project:

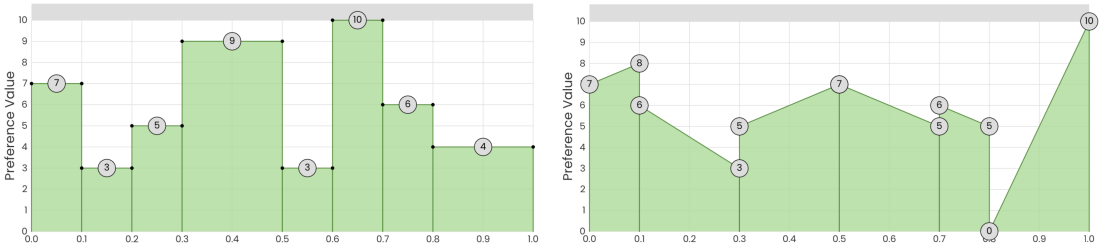
1. **Normalisation:** For $1 \leq i \leq n$, we have $v_i(X) = 1$ and $v_i(\emptyset) = 0$.
2. **Nonnegativity[19][1]:** For all portions $X_j \subseteq X$, we have $v_i(X_j) \geq 0$.
3. **Additivity:** For all portions $X_j, X_k \subseteq X$ with $X_j \cap X_k = \emptyset$, we have

$$v_i(X_j \cup X_k) = v_i(X_j) + v_i(X_k).$$

4. **Divisibility[2]:** For all portion $X_j \subseteq X$ and there is a sub-portion $X_k \subseteq X_j$, we can always find a real number α, such that

$$v_i(X_k) = \alpha \cdot v_i(X_j).$$

These properties hold for different forms of evaluation functions. Figure 2.1 shows the two forms of evaluation equations involved in this project, piecewise constant and piecewise linear. See Ernst's [11] work for more design details. In fact, piecewise constant functions can be considered as a special case of piecewise linear, where all values are fixed constants within the smallest granularity intervals of each evaluation. Therefore, for $1 \leq i \leq n$, the evaluation $v_i(X_j)$ of agent a_i for a portion $X_j = [\alpha, \beta]$, where $\alpha, \beta \in [0,1]$, is the sum of the areas of the rectangles covered by X_j, that is, $\sum_{\alpha}^{\beta}$ area of rectangles in $[\alpha, \beta]$. In contrast, piecewise linear valuation functions represent a more general case. Here, agent a_i evaluates $v_i(X_j)$ for the portion X_j as the sum of the areas of the right trapezoids within $[\alpha, \beta]$, we say, $\sum_{\alpha}^{\beta}$ area of the right trapezium.



(a) Piecewise constant valuation (b) Piecewise linear valuation

Figure 2.1: Two relevant evaluation function forms [11]

[1]Using Positivity: For all portions $X_j \subseteq X$, we have $v_i(X_j) > 0$, which seems more reasonable, as users typically do not evaluate an empty portion. However, when we allow users to assign 0 utility to a non-empty portion, this is where nonnegativity plays a role.

[2]Additivity and Divisibility imply that the cake can be cut arbitrarily without loss of value.

The two forms of the evaluation function described above, which are utilised in this project, demonstrate that while we can efficiently calculate an agent's preference for any portion, the more complex piecewise linear functions introduce greater computational complexity. This complexity arises from the intricacies in determining cut points and the non-linear changes within intervals. Thanks to Ernst's work [11], we now have a high-precision method for identifying cut points that meet the desired conditions. Based on his work, this method, using *decimal.Decimal* from the Python library to ensure precision up to 15 decimal places, will be enhanced and utilised throughout this project, with detailed information provided in the Appendix A.1.

2.1.3 Cake-Cutting Protocols

From the computational perspective, cake-cutting protocols provide algorithmic solutions to the two core questions of fair division: whether a division exists that satisfies a given fairness criterion, and how to achieve such a division. A cake-cutting protocol is a set of rules that instructs agents what to do next and describes an interactive process to be followed to divide a cake X among n agents [20]. Although the protocol does not know anything about the agents, including their evaluation equations, it asks the agent for the value of a particular piece of cake at a specific step and recommends the next action based on the response. Robertson and Webb [19] suggest that such interactions between the agent and the protocol can be categorised into two types of requests:

For $1 \leq i \leq n$, agents a_i and all possible portions $X_j \subseteq X = [0,1]$

1. **Evaluation** $eval_i(X_j)$**:** Requests agent a_i to return the value of X_j based on her evaluation function, that is, $v_i(X_j)$.

2. **Cut** $cut_i(X_j, x, \alpha)$**:** Given a portion X_j, a cut point $x \in X_j$, and a target value α, requests agent a_i to return another cut point $0 \leq x \leq y \leq 1$ such that $v_i([x,y]) = \alpha$, if possible[3].

This approach, known as the Robertson-Webb query model, will continue to serve as the core of the project, particularly when using binary search to find eligible cut points. Refer to Section 3.3 for implementation details.

[3]If no such cut point exists, the treatment may vary. In the released version of this project, the rightmost side of the portion is returned. During testing, the programme terminates.

2.1.4 Fairness Criteria

In the cake-cutting problem, various fairness criteria are discussed, with envy-freeness being the most well-known and directly relevant to this project. An envy-free division is a division where no agent believes that another agent has received a more valuable portion than their own, meaning no agent envies another. Formally, for every agent i receiving a portion X_i, where $1 \leq i, j \leq n$ and $i \neq j$, we have

$$v_i(X_i) \geq v_i(X_j).$$

While the concept of envy-freeness is intuitive, it represents a stringent fairness criterion. Despite decades of research, no general solution has been discovered for scenarios involving even more than three agents. Thus, a relaxation of envy-freeness, known as ε-envy-freeness, was introduced by Brânzei and Nisan [9]. This accepted approximation of fair division permits the cake-cutting problem to be addressed more efficiently in scenarios of greater complexity. A division is ε-envy-free if, ε is a small approximation factor, every agent i receiving a portion X_i , where $1 \leq i, j \leq n$ and $i \neq j$, satisfies

$$v_i(X_i) \geq v_i(X_j) - \varepsilon.$$

Notably, the problem for two and three agents has been elegantly addressed by two well-known cake-cutting protocols: Cut & Choose and the Selfridge-Conway Method. These methods have been integrated into Fair Slice by Ernst [11], providing an interactive visualisation that enhances user understanding of envy-free divisions. This work explores the promising algorithm recently proposed by Hollender and Rubinstein [12] for resulting continuous ε-envy-free fair division among four people. It translates this protocol into practical code that extends Fair Slice's functionality to four agents scenarios and utilises the same interactive educational interface to visually explain this advanced algorithm. The goal is to enable both practical application and educational demonstration use of the four agents cake-cutting problem in Fair Slice.

2.2 Overview of Divisible Fair Division Algorithms

The work of Steinhaus [26] and the collaborative research he conducted with his colleagues [13] in the 1940s is considered the foundation of the cake-cutting problem, which has since sparked decades of extensive research. In the early stages of the problem, the Cut & Choose method was proposed as an elegant solution for two-person

scenarios. Approximately twenty years later, in 1960s, the Selfridge-Conway Method [7, 19] offered a solution for three-agent scenarios, employing a straightforward mechanism built upon a complex underlying concept. Since then, research has concentrated on more complex scenarios involving more than three agents, an area that remains both challenging and actively pursued, with no comprehensive solution yet identified.

Although Steinhaus [25] has proven that an envy-free cake division always exists for any number of agents, the method to achieve such a division remains elusive, leading to the introduction of various constraints in researches from a computational perspective. In tackling the four agents scenario, an obvious idea is to extend the principles of the Selfridge-Conway Method to develop a suitable protocol for this case. Brams, Taylor, and Zwicker [5] responded this in 1997 by proposing a protocol that combines Austin's two-agent moving-knife protocol [1] with the ideas from the Selfridge-Conway Method. Although their protocol guarantees envy-freeness, it sacrifices finite boundedness, thereby limiting its practicality. Moreover, the finite envy-free cake-cutting algorithm designed by Brams and Taylor [6, 7] for n agents becomes impractical with a growing number of agents, as its complexity escalates, often requiring an intolerable amount of runtime, which makes it unsuitable for fast and practical resource allocation. Until 2016, a breakthrough finite bounded, envy-free cake-cutting protocol for four agents with a maximum 203 cuts, was proposed by Aziz and Mackenzie [3]. It implies that the number of decisions needed for envy-free division is predictable before the protocol is executed. This protocol was subsequently extended to n agents scenarios [2], which typically require $O\left(n^{n^{n^{n^n}}}\right)$ runtime. All of the above protocols suffer from two drawbacks in real-world applications, the potentially infinite dividing of the cake and the intolerable running time. Recent work by Hollender and Rubinstein [12] proposed a promising framework to address this problem by resulting continuous envy-free fair division within a certain time complexity.

2.3 Overview of Hollender-Rubinstein Algorithm

The Hollender-Rubinstein Algorithm produces a continuous ε-envy-free (ε-EF) fair division with a time complexity of $O(\log^3(\frac{1}{\varepsilon}))$. This ensures that the cake is divided into n pieces among n agents using $n-1$ cut points. Specifically, for $n=4$, the cut points are denoted as (l,m,r), where $0 \leq l \leq m \leq r \leq 1$. In addition to the assumptions outlined in Section 2.1.2, the operation of the algorithm relies on the following three additional considerations:

1. **Lipschitz-continuous with Lipschitz constant L:** For all points $0 \leq a' \leq a \leq b \leq b' \leq 1$ and the evaluation function v_i for agent i, the following holds:

$$|v_i(a,b) - v_i(a',b')| \leq L(|a-a'| + |b-b'|).$$

   Without loss of generality, we assume this function is 1-Lipschitz-coutinuous where $L = 1$, Therefore, we have

$$|v_i(a,b) - v_i(a',b')| \leq (|a-a'| + |b-b'|).$$

2. **Monotonicity:** The value $v_i(a,b)$ is at least $v_i(a',b')$, that is $v_i(a,b) \leq v_i(a',b')$, whenever the interval $[a,b]$ is the subset of the interval $[a',b']$.

3. **Hungriness:** The value $v_i(a,b)$ is strictly less than $v_i(a',b')$, that is $v_i(a,b) < v_i(a',b')$, whenever the interval $[a,b]$ is the true subset the interval $[a',b']$.

2.3.1 Core Idea

The core idea of the Hollender-Rubinstein Algorithm is to maintain an invariant throughout its execution, parameterised by $\alpha \in [0,1]$. The algorithm begins with an equipartition by Agent 1, and as α increases monotonically, it follows a continuous path that consistently holds the invariant. This path is guaranteed to eventually terminate, resulting in an ε-EF allocation. The desired invariant is one of the following two conditions:

- **Condition A:** Agent 1 is indifferent among its top three favourite pieces, and the remaining piece is (weakly) preferred by at least two of the other three agents.
- **Condition B:** Agent 1 is indifferent between its two favourite pieces, and each of the remaining two pieces is (weakly) preferred by at least two of the other three agents.

Equipartition refers to the division of a cake into four pieces, each with equal value from the perspective of Agent 1, denoted as $\alpha_4^=$. It has been proved that such a unique equipartition always exists and can be efficiently determined [12]. This equipartition satisfies Condition A, which serves as the initial step in the Hollender-Rubinstein Algorithm. The methodology for achieving equipartition is detailed in Section 3.4, while the formal definitions of Conditions A and B are provided in Section 3.5.

2.3.2 Algorithm

The Hollender-Rubinstein Algorithm is based on an intuitive idea and straightforward procedures. However, each of these steps involves significant computational effort and extensive searching behind the scenes, with the detailed implementations provided in Chapter 3. The workflow of the algorithm is shown in Figure 2.2 and the pseudocode is as follows.

Algorithm 1 Hollender-Rubinstein Algorithm

1: Compute the unique equipartition of the cake according to Agent 1.
2: **IF** this equipartition yields an envy-free division **THEN**
3: Return the allocation.
4: **ELSE**
5: Set $\underline{\alpha} := \alpha_4^{=}$ and $\bar{\alpha} := 1$.
6: **ENDIF**
7: **REPEAT**
8: Let $\alpha := (\underline{\alpha} + \bar{\alpha})/2$.
9: **IF** Condition A or B holds at value α **THEN**
10: Set $\underline{\alpha} := \alpha$.
11: **ELSE**
12: Set $\bar{\alpha} := \alpha$.
13: **ENDIF**
14: **UNTIL** $|\bar{\alpha} - \alpha| \leq \varepsilon^4/12$
15: Return a division of the cake that satisfies Condition A or B at value $\underline{\alpha}$.

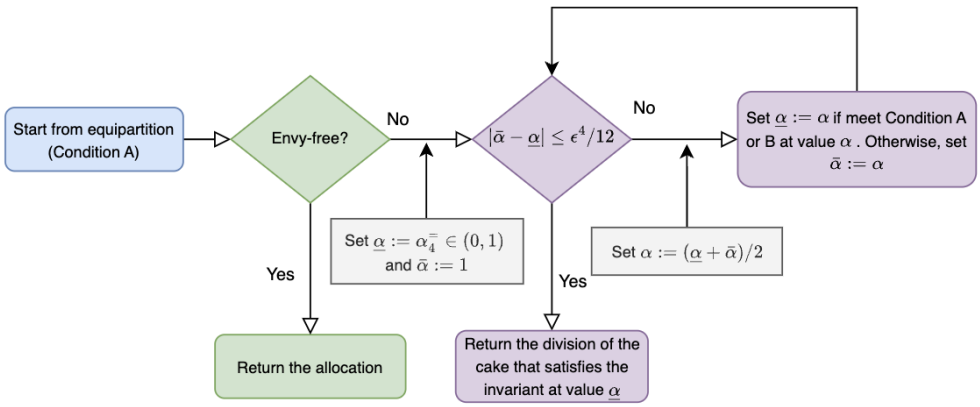
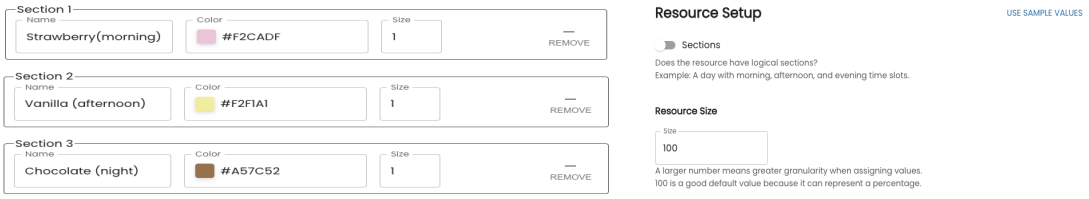


Figure 2.2: Algorithm Workflow

2.4 Fair Slice

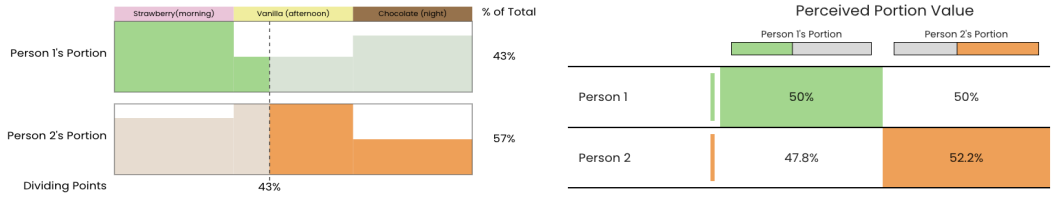
Fair Slice is a visual resource partitioning tool developed by Ernst [11] as his MSc project for the 2022/2023 academic year. As the first of its kind, it demonstrates how fair division algorithms can be applied to divisible resources, such as time and simplified land, optionally using a cake as a metaphor, see Figure 2.3.



(a) Resource Setup with Meaning (Three-Shift System) (b) Resource Setup without Meaning

Figure 2.3: Multi-Purpose Resource Setup

The project effectively integrates the Cut & Choose and Selfridge-Conway Method to support 2 to 3 participants scenarios with a highly engaging visual interface, allowing users to naturally and effectively express their preferences for resources, as discussed in Section 2.1.2. Additionally, Its clear explanations of algorithmic results, combined with its interactive course, make it one of the most effective tools for illustrating fair division concepts, while also providing practical references for real-world resource allocation scenarios. For more outstanding features and design concepts, please refer to the original work [11].



(a) Result explanation: Evaluation View (b) Result explanation: Perceived Portion View

Figure 2.4: Two of the Three Result Representations of the Algorithm.

The aim of this project is to extend the capabilities of Fair Slice to accommodate a four-agent scenario and to integrate the Hollender-Rubinstein Algorithm, thereby bringing the four-agent envy-free cake-cutting problem into the public attention. Additionally, the project aims to maintain the interactive educational and presentation interface that offers users an intuitive understanding of the algorithm's mechanics, supporting both practical applications and academic research.

Chapter 3

Code Implementation

3.1 Original Valuation Functions

Thanks to Ernst's [11] work, the original valuation functions v_i can be interpreted as the area of a triangle or right trapezium within the desired interval, and have been effectively implemented in TypeScript. One of the primary tasks of this project is to port the original TypeScript codebase to Python[1] (see Appendix B.1 for the corresponding Python code, get_value_for_interval). Function get_value_for_interval accepts a range from $[0,\infty]$, just as Fair Slice originally did. However, it is important to note that the original valuation functions have values in the range $[0,\infty]$, which conflicts with the 1-Lipschitz assumption. To address this issue, we normalise the original valuation functions, bringing their values into the range $[0,1]$ to align with the algorithm's requirements (see Appendix B.2).

```
def _v(segments: List[Segment], a: Decimal, b: Decimal, cake_size:
    Decimal) -> Decimal:
     whole_cake_value = get_value_for_interval(segments, to_decimal
         (0), cake_size)
     v = norm(get_value_for_interval(segments, a, b),
         whole_cake_value)
     return v
```

Listing 3.1: Original Valuation Functions

[1] As mentioned earlier, the algorithm relies heavily on floating-point computations, where small differences in precision are generally negligible. However, since we are consistently working with values in the range [0,1], particularly when computing modified evaluation functions, the complexity of expressions and integration of results can significantly impact the final outcome. Based on our experiments, we use Python's decimal.Decimal with 15 decimal places of precision to ensure accuracy while maintaining acceptable performance.

3.2 Preprocessing

The Hollender-Rubinstein Algorithm operates based on monotone 1-Lipschitz valuations. However, to ensure that the assumptions outlined in Section 2.3 are satisfied, it is necessary to perform on-the-fly adjustments during the algorithm's execution to transform the original valuation functions v_i into ε-strongly-hungry valuations. As a result, any ε-envy-free allocation derived from the modified valuation functions will correspond to a 12ε-envy-free allocation with respect to the original valuations. Thus, although this process is termed "preprocessing" these adjustments occur during the computation.

Based on Hollender and Rubinstein's work [12], we modify the original 1-Lipschitz valuation functions v_i to $v_i'(a,b) = \frac{v_i(a,b)}{2} + \varepsilon|b-a|$, making v_i' both 1-Lipschitz continuous and ε-strongly-hungry. Given that $v_i \in [0,1]$, it is obvious that $v_i' \in [0,1]$ as well.

```
def _v_prime(segments: List[Segment], epsilon: Decimal, a: Decimal,
    b: Decimal, cake_size: Decimal ) -> Decimal:
     v = _v(segments, a, b, cake_size) / 2 + epsilon * abs(b - a)
     assert 0 <= v <= 1, f"prime value should in [0, 1], got {v}"
     return v
```

Listing 3.2: Modified v_i' valuation functions

We then construct v'' by performing a piecewise linear interpolation of v' over the grid $\{0,\delta,2\delta,\ldots,1-\delta,1\}$[2]. More specifically, for any $0 \le a \le b \le 1$, consider consecutive multiples of δ, denoted $\underline{a}$ and $\bar{a}$, where $\underline{a}$ and $\bar{a}$ satisfy $\underline{a} \le a \le \bar{a}$, and similarly $\underline{b}$ and $\bar{b}$ satisfy $\underline{b} \le b \le \bar{b}$. Next, we modify the evaluation functions according to the conditions:

$$v_i'' = \begin{cases} \frac{(\bar{a}-a)-(b-\underline{b})}{\delta} \cdot v_i'(\underline{a},\underline{b}) + \frac{b-\underline{b}}{\delta} \cdot v_i'(\underline{a},\bar{b}) + \frac{a-\underline{a}}{\delta} \cdot v_i'(\bar{a},\underline{b}) & \text{if } \bar{a}-a \ge b-\underline{b} \\ \frac{(b-\underline{b})-(\bar{a}-a)}{\delta} \cdot v_i'(\bar{a},\bar{b}) + \frac{\bar{a}-a}{\delta} \cdot v_i'(\underline{a},\bar{b}) + \frac{\bar{b}-b}{\delta} \cdot v_i'(\bar{a},\underline{b}) & \text{if } \bar{a}-a \le b-\underline{b} \end{cases}$$

In this process, given an $a \in [0,1]$, finding its consecutive multiples in the δ grid is an important task. Take Listing 3.3 as an example to find $\underline{a}$. First, we validate the input a. Then, we handle specific boundary cases: for example, the function should return 0 when $a < \delta$ or $a = 0$, and $1-\delta$ when $a = 1$. Additionally, if a is already a multiple of δ, the function should return the next smaller multiple of δ rather than a itself. To account for the precision limitations of floating-point calculations, we introduce a tolerance that

[2] In this project, we let $\delta := \varepsilon$.

is at least as small as δ to detect this scenario. A similar function that computes $\bar{a}$ is provided Listing B.3 in Appendix B.3.

```
def underline(x, delta, tolerance=Decimal("1e-10")) -> Decimal:
    assert 0 <= x <= 1, f"got {x}, expect it between [0, 1]"
    if x < delta or x == 0:
        return to_decimal(0)

    if x == 1:
        return x - delta

    # Check if x is an exact multiple of delta,
    # considering floating point precision issues
    if abs(x % delta) < tolerance or abs(delta - (x % delta)) <
       tolerance:
        v = (x / delta - 1).to_integral_value(rounding="ROUND_FLOOR"
            ) * delta
    else:
        v = (x / delta).to_integral_value(rounding="ROUND_FLOOR") *
            delta

    return max(v, to_decimal(0))
```

Listing 3.3: Finding the largest multiple in the δ-grid that is less than or equal to x.

The complete code for calculating v'' is provided in Listing B.4. Since Fair Slice allows users to evaluate multiple segments, where the number of segments ranges from $[1, \infty]$, these segments must be treated as a whole cake. This requires a transformation from $[1, \infty]$ to $[0, 1]$, see Listing 3.4. The code for the reverse transformation is also provided in Listing B.5.

```
def scale_to_unit(a: Decimal, cake_size: Decimal) -> Decimal:
    assert (
        to_decimal(0) <= a <= to_decimal(cake_size)
    ), f"a must be greater than or equal to 0 and less than
       cake_size: {cake_size}, got {a}"

    if cake_size == 1:
        return to_decimal(a)
    elif cake_size == 0:
        return to_decimal(0)
    return to_decimal(a) / to_decimal(cake_size)
```

Listing 3.4: Coverting segments range from $[1, \infty]$ to $[0, 1]$.

Thus, we then use the transformed values to interpolate $v_i^{'}$ for computing $v_i^{''}$, while the original segment values are used to compute v_i and $v_i^{'}$. This approach maximises the utilisation of the original codebase with minimal modifications, ensuring the accuracy of the implementation.

```
def _v_double_prime(segments: List[Segment], delta: Decimal, a:
    Decimal, b: Decimal, cake_size: Decimal) -> Decimal:
     a_unit = to_decimal(scale_to_unit(a, cake_size))
     b_unit = to_decimal(scale_to_unit(b, cake_size))
     delta = to_decimal(delta)

     # Get the transformed grid points around a and b
     a_underline_unit = underline(a_unit, delta)
     a_overline_unit = overline(a_unit, delta)
     b_underline_unit = underline(b_unit, delta)
     b_overline_unit = overline(b_unit, delta)

     # Using original segments range to calculate v and v' values
     a_underline = scale_back_from_unit(a_underline_unit, cake_size)
     a_overline = scale_back_from_unit(a_overline_unit, cake_size)
     b_underline = scale_back_from_unit(b_underline_unit, cake_size)
     b_overline = scale_back_from_unit(b_overline_unit, cake_size)

     v_prime_a_under_b_over = _v_prime(segments, delta, a_underline,
         b_overline, cake_size)
     v_prime_a_over_b_under = _v_prime(segments, delta, a_overline,
         b_underline, cake_size)

     if a_overline_unit - a_unit >= b_unit - b_underline_unit:
         # Calculate v'' value using transformed values
         ...
```

Listing 3.5: Part of final modified $v_i^{''}$ valuation functions: using transformed segment values while retaining original segment values for calculating v_i and $v_i^{'}$.

Similar to the calculation of v_i, there is a wrapper function, get_double_prime_for_interval, designed to calculate the value of $v_i^{''}$ for any given interval and set of segments, see Listing B.6.

3.3 Binary Search

The efficiency of the Hollender-Rubinstein Algorithm lies in its innovative design, which guarantees a path that maintains the invariant and ultimately results in three cuts, producing an ε-EF division. Due to the monotonicity of the evaluation functions, the value of the cake strictly increases as a cut moves from left to right. This property enables the algorithm to leverage binary search to accelerate both the Evaluation and Cut requests, as the cake is inherently ordered from left to right. In the δ-grid, it jsut requires at most $\log(1/\varepsilon)$ steps to determine the interval where the cut point for a given value must lie.

The binary search utilises the classical approach of comparing the middle value to the target value within an internally ordered cake. This process involves determining which part of the set the target lies in and then recursively continuing the search within that subset, either to find the target value or to conclude that it does not exist[3]. The process is illustrated in Figure 3.1. Notably, the critical value is the v_i'' of a piece of cake, and calculating this value requires recording the original starting point of this piece. Additionally, due to the potential precision issues inherent in floating-point calculation, we consider the target value found when the critical searched value is sufficiently close to the target within a specified tolerance.

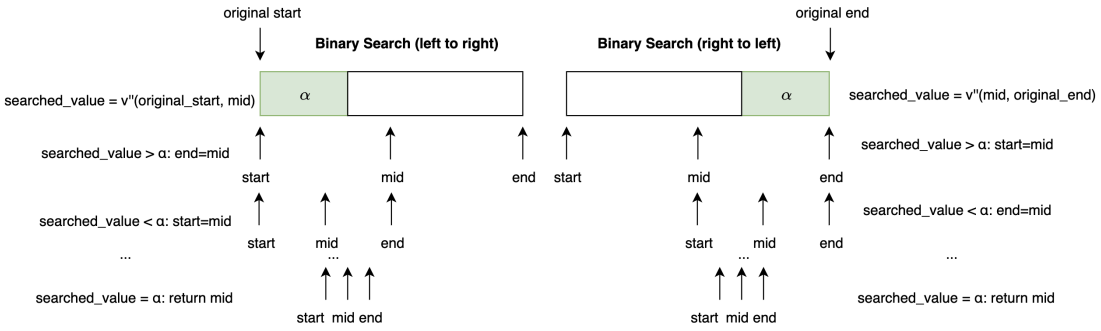


Figure 3.1: Binary Search

As illustrated in Figure 3.1, there are two binary search alternatives tailored for different scenarios. A notable example that demonstrates this is during the checking of condition A, where the locations of the two cuts, l and r, must be determined before identifying the location of m. The l is located using the left-to-right approach, while the r is identified through the right-to-left variation. Listing 3.6 shows the implementation of the right-to-left binary search. Another variant is provided in Listing B.7.

[3]In this case, we just return the whole cake or the end of a piece of cake.

```
def _binary_search_left_to_right(preference: List[Segment],
    cake_size: Decimal, epsilon: Decimal, start: Decimal, end:
    Decimal, target: Decimal, tolerance: Decimal = to_decimal(1e-10),
     max_iterations: int = 1000) -> Decimal:
     full_cake = get_double_prime_for_interval(segments=preference,
        epsilon=epsilon, start=start, end=end, cake_size=cake_size)
     if full_cake < target:
         return end

     original_start = start
     iteration = 0

     while end - start > tolerance and iteration < max_iterations:
         mid = to_decimal((start + end) / 2)
         searched_value = get_double_prime_for_interval(segments=
            preference, epsilon=epsilon, start=original_start, end=
            mid, cake_size=cake_size)

         if abs(searched_value - target) < tolerance:
             return mid

         if searched_value < target:
             start = mid
         else:
             end = mid

         iteration = iteration + 1
     return to_decimal((start + end) / 2)
```

Listing 3.6: Binary Search From Left to Right.

3.4 Equipartition

In the work of Hollender and Rubinstein [12], they theoretically proved that at most $O(\log^2(1/\varepsilon))$ evaluation requests are needed to find a unique equipartition that divides the cake into four equal pieces according to Agent 1's preference. In practice, leveraging the Evaluation request model alongside binary search, we can find this equipartition in $O(\log(1/\varepsilon))$. The approach is straightforward. Due to the normalisation property of the evaluation functions, Agent 1 consistently expects the value of each piece in the

equipartition to be 0.25. We then use binary search within predictably appropriate sets to locate the exact position of these desired cuts. For example, l must lie within $[0,1]$, m must lie within $[l,1]$, and r must lie within $[m,1]$[4]. The code is shown in Listing 3.7.

```
def equipartition(preference: List[Segment], cake_size: Decimal,
    epsilon: Decimal, start: Decimal, end: Decimal, tolerance:
    Decimal = to_decimal(1e-10)) -> List[Decimal]:

    total_v = get_double_prime_for_interval(segments=preference,
        epsilon=epsilon, start=start, end=end, cake_size=cake_size)
    segment_value = total_v / 4 # Expected to be 0.25

    # Finding cuts at 1/4, 1/2, and 3/4 of the cake
    first_cut = _binary_search_left_to_right(preference=preference,
        cake_size=cake_size, epsilon=epsilon, start=start, end=end,
        target=segment_value, tolerance=tolerance)

    second_cut = _binary_search_left_to_right(preference=preference,
         cake_size=cake_size, epsilon=epsilon, start=first_cut, end=
        end, target=segment_value, tolerance=tolerance)

    third_cut = _binary_search_left_to_right(preference=preference,
        cake_size=cake_size, epsilon=epsilon, start=second_cut, end=
        end, target=segment_value, tolerance=tolerance)

    return [first_cut, second_cut, third_cut]
```

Listing 3.7: Equipartition in Practice.

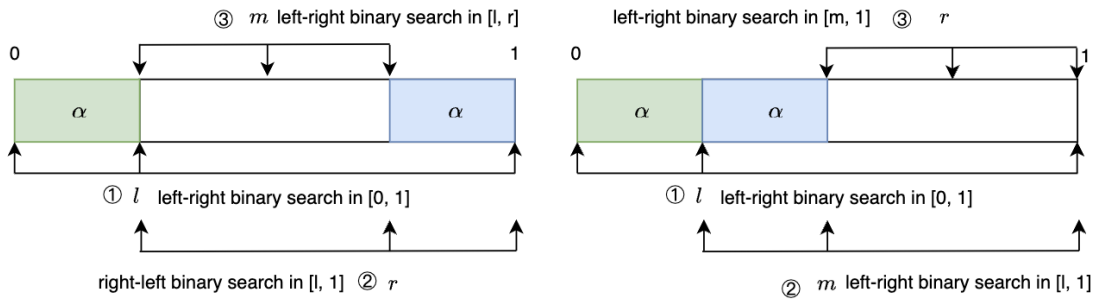


Figure 3.2: Different Methods Yield the Same (l,m,r) for the Unique Equipartition.

[4]There are alternative sequences for finding such an equipartition. For instance, one can use left-to-right binary search to find l and right-to-left binary search to find r, then determine m within the interval $[l,r]$. Ultimately, all methods yield the same unique (l,m,r) for the unique equipartition, if one exists.

3.5 Conditions Handling

Given a division (X_1, X_2, X_3, X_4) where $X = \bigcup_{j=1}^{4} X_j$, we now provide the formal definitions of Conditions A and B:

- **Condition A:** If the division (X_1, X_2, X_3, X_4) satisfies Condition A, then for any piece $k \in \{1,2,3,4\}$, it holds that $v_1(X_t) = \alpha$ for all $t \in \{1,2,3,4\} \setminus \{k\}$. Furthermore, there must exist two distinct agents $i, i' \in \{2,3,4\}$ such that: 1. $v_i(X_k) \geq \max_t v_i(P_t)$. 2. $v_{i'}(X_k) \geq \max_t v_{i'}(X_t)$.

- **Condition B:** If the division (X_1, X_2, X_3, X_4) satisfies Condition B, then for any agent $i \in \{2,3,4\}$ and two distinct pieces $k, k' \in \{1,2,3,4\}$, it holds that $v_i(X_t) = v_i(X_{t'})$ and $v_1(X_t) = \alpha$ for all $t \in \{1,2,3,4\} \setminus \{k, k'\}$. Furthermore, these should also be satisfied:

  1. $v_1(X_k) \leq \alpha$ and $v_1(X_{k'}) \leq \alpha$.
  2. $v_i(X_k) = v_i(X_{k'}) \geq \max_t v_i(X_t)$.
  3. there must exist an agent $i' \in \{2,3,4\} \setminus \{i\}$ such that $v_{i'}(X_k) \geq \max_t v_{i'}(X_t)$.
  4. there must exist an agent $i' \in \{2,3,4\} \setminus \{i\}$ such that $v_{i'}(X_{k'}) \geq \max_t v_{i'}(X_t)$.

Both conditions can be considered in a two-step process. For Condition A, we first identify all possible values of k and determine the corresponding cuts. We then verify whether at least two agents satisfy the additional condition. Similarly, for Condition B, we identify the relevant agents i, k and k', determine the corresponding cuts, and subsequently check if the additional conditions are met.

3.5.1 Condition A

Given an $\alpha \in [0,1]$, for each of the four possible values of k, we can find the exact and unique cuts in $O(\log(1/\varepsilon))$, ensuring that all other pieces of the cake, except for k, have an exact value of α. Consider the case where k is the leftmost piece of cake. We can perform a right-to-left binary search to find r such that $v_1(r,1) = \alpha$. Similarly, we then find m such that $v_1(m,r) = \alpha$, and finally l such that $v_1(l,m) = \alpha$. In this way, we determined the exact position of cuts (l,m,r), where each piece has a value of exactly α, except for k. Next, we check if there are at least two agents who prefer k. If this is the case, the division satisfies Condition A at value α. Otherwise, this division does not

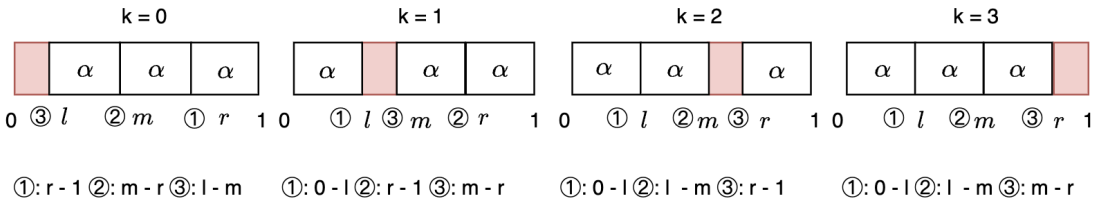


Figure 3.3: Solutions of All Possible identity of piece k.

satisfy Condition A at this point. The demonstrated case and the other three cases are shown in Figure 3.3.

As previously discussed, the process of handling Condition A begins with identifying k and determining the cuts. Once a suitable k and the corresponding cuts are found, we check whether there are at least two other agents prefer k. If not, it will try to identify another different k.

```
def check_condition_a(alpha: Decimal, preferences: Preferences,
    cake_size: int, epsilon: Decimal, tolerance: Decimal) -> Tuple[
    bool, Dict[str, Any]]:
     preference_a = preferences[0]

     # Find cuts and identify k
     for k in POSIBLE_K:
         results = _find_cuts_and_k_for_condition_a(k=k, alpha=alpha,
              preference=preference_a, cake_size=to_decimal(cake_size)
             , epsilon=epsilon, tolerance=tolerance)
         if len(results) == 0:
              continue

         cuts = results["cuts"]
         k = results["k"]
         ...
```

Listing 3.8: Condition A Handling Part 1

Using two types of binary searches in different orders is sufficient to address all four possible cases of k. A part of the implementation of finding cuts and k for Condition A is provided in Listing 3.9.

```
def _find_cuts_and_k_for_condition_a(k: int, alpha: Decimal,
    cake_size: Decimal, preference: List[Segment], epsilon: Decimal,
    tolerance: Decimal = to_decimal(1e-3)) -> Dict[str, Any]:
     if k == 0:
```

```
        r = _binary_search_right_to_left(preference=preference,
            cake_size=cake_size, epsilon=epsilon, start=start, end=
            end, target=alpha, tolerance=tolerance)
        m = _binary_search_right_to_left(preference=preference,
            cake_size=cake_size, epsilon=epsilon, start=start, end=r,
             target=alpha, tolerance=tolerance)
        l = _binary_search_right_to_left(preference=preference,
            cake_size=cake_size, epsilon=epsilon, start=start, end=m,
             target=alpha, tolerance=tolerance)
        remained_value = get_double_prime_for_interval(preference,
            epsilon, start=start, end=l, cake_size=to_decimal(
            cake_size))
        if remained_value <= alpha:
            return {"cuts": [l, m, r], "k": 0}
    if k == 1:
    ...
```

Listing 3.9: Find Cuts and K for Condition A

We then check whether at least two other agents prefer k. This is accomplished using the function _check_if_weakly_prefer_piece_k, which is literally a wrapper function that simply checks that if a particular agent's evaluation of the interval $[start, end]$ is at least as large as the highest evaluated value, excluding k, which in this context is α.

```
        ...
        start_k, end_k = get_range_by_cuts(cuts,k,to_decimal(
            cake_size))
        v_1 = get_double_prime_for_interval(segments=preference_a,
            epsilon=epsilon,start=start_k,end=end_k,cake_size=
            cake_size)
        weak_preference = [False for _ in range(len(preferences))]
        # check if at least two of the other agents weakly prefer k
        for i in range(1, len(preferences)):
            weak_preference[i] = _check_if_weakly_prefer_piece_k(
                preference=preferences[i],cake_size=to_decimal(
                cake_size),epsilon=epsilon,start=start_k,end=end_k,
                alpha=k_1)
        if sum(weak_preference) >= 2:
            return (True, {"cuts": cuts, "k": k})
        else:
            return (False, {})
```

Listing 3.10: Condition A Handling Part 2

3.5.2 Condition B

For Condition B, we first need to examine all agents $i \in \{2,3,4\}$ and identify all possible cases where Agent 1 does not prefer pieces k and $k^{'}$. There are six cases, which are characterised in Listing 3.11. Based on this, we can then identify the appropriate cuts. Finally, we verify whether there are any agents $i^{'} \in \{2,3,4\} \setminus \{i\}$ who are indifferent between pieces k and $k^{'}$. If such agents exist, then we state that Condition B is satisfied.

```
POSSIBLE_K_AND_K_PRIME_COMBINATION_ON_CONDITION_B = [
    (0, 1, [2, 3]), (0, 2, [1, 3]), (0, 3, [1, 2]),
    (1, 2, [0, 3]), (1, 3, [0, 2]), (2, 3, [0, 1]),
]
```

Listing 3.11: Possible Identity of k and $k^{'}$ On Condition B

In order to find the positions of the cuts corresponding to all possible combinations $(k,k^{'})$, we have three handlers. As shown in Listing 3.12.

```
CONDITION_B_HANDLERS = {
    (0, 1): _handle_adjacent, (1, 2): _handle_adjacent,
    (2, 3): _handle_adjacent, (0, 2): _handle_one_between,
    (1, 3): _handle_one_between, (0, 3): _handle_leftmost_rightmost,
}
```

Listing 3.12: Handlers for Handling Each Possible Case $(k,k^{'})$

We perform operations similar to Condition A when k and $k^{'}$ are adjacent. Supposing k and $k^{'}$ are the first and second pieces of the cake is an effective example to illustrate this process. We can use two right-to-left binary searches to find the exact position of cuts r and m successively to ensure that $v_1(r,1) = \alpha$ and $v_1(m,r) = \alpha$. Next, a general binary search is employed to divide the cake into two equal parts within an interval according to Agent i's preference, in this case, finding a cut l such that $v_i(0,l) = v_i(l,m)$. This binary search, with a complexity of $O(\log(1/\varepsilon))$, is detailed in Listing 3.13.

```
def _find_balanced_cut_for_adjacent(preference: List[Segment],
    cake_size: Decimal,epsilon: Decimal,left: Decimal,right: Decimal,
    tolerance: Decimal,max_iterations: int = 1000) -> Decimal:
     start, end, iteration = left, right, 0

     while end - start > tolerance and iteration < max_iterations:
         m = (start + end) / 2
         first_half_value = get_double_prime_for_interval(segments=
             preference,epsilon=epsilon,start=left,end=m,cake_size=
             cake_size)
```

```
        second_half_value = get_double_prime_for_interval(segments=
            preference,epsilon=epsilon,start=m,end=right,
             cake_size=cake_size)

        if abs(first_half_value - second_half_value) < tolerance:
             return m

        if first_half_value < second_half_value:
             start = m
        else:
             end = m
        iteration += 1
    return (start + end) / 2
```

Listing 3.13: Generic Binary Search For Dividing an Interval into Equal Two Parts

The full implementation of _handle_adjacent is presented in Listing B.11, while all corresponding solutions are illustrated in Figure 3.4.

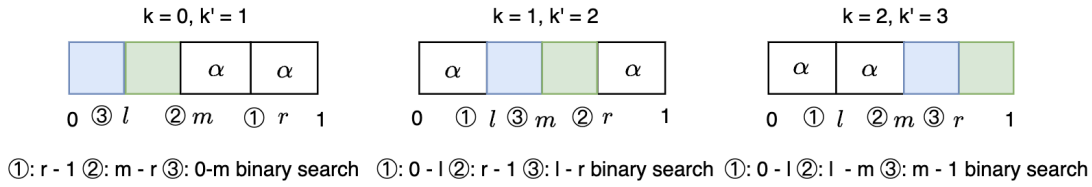


Figure 3.4: Solutions of All Possible identity of adjacent piece k and $k^{'}$.

When it comes to there is one piece between k and $k^{'}$, the process differs from the previous case where exact positions for two cuts could be determined before finding a third. Assuming that k is the first piece of the cake, and $k^{'}$ is the third. In this scenario, we can be certain that there is a cut r can be found such that $v_1(r,1) = \alpha$. Additionally, for every possible position of l, there exists a unique cut $m(l)$ such that $v_1(l,m(l)) = \alpha$. At this point, we are assured that the value of the other two pieces of cake, except for k and $k^{'}$, is at least α from the perspective of Agent 1. Finally, based on the fact that as l moves to the right, the interval $[m(l),r]$ becomes smaller, a point[5] will eventually be reached where $v_i(0,l) = v_i(m(l),r)$, which can be located using binary search. This

[5]Hollender and Rubinstein [12] suggest first identifying appropriate intervals for both of l and m, and then determining their final positions based on the information obtained. In practice, these suitable intervals are often extremely narrow. For example, a suitable interval for r might be $[0.74999999999999999999, 0.75000000000001]$, where such minimal differences are often negligible in float-point calculations. Additionally, it is common for the upper and lower bounds of the interval to converge to a single point. The function _handle_leftmost_rightmost employs this approach, which is computationally intensive and produces marginal gains.

process requires $O(\log^2(1/\varepsilon))$ evaluation requests, because for a given cut l, it takes $O(\log(1/\varepsilon))$ to find $m(l)$, and then l is adjusted to find the desired positions of l and m. The core algorithm for this scenario is presented in Listing 3.14, and the case where $(k,k') = (1,3)$ follows a similar approach to what has been previously discussed. That is, the position of l is fixed, and a dynamic $m(r)$ must be found to satisfy $v_1(m(r),r) = \alpha$. Subsequently, r is adjusted to finalise the positions of both m and r. The complete _handle_one_between implementation is provided in Listing B.12.

```
def _binary_search_case_0_2(preference_1: List[Segment],preference_i
    : List[Segment],epsilon: Decimal,l_start: Decimal,l_end: Decimal,
    alpha: Decimal,cake_size: Decimal,tolerance: Decimal,
    max_iterations: int = 1000) -> Tuple[Decimal, Decimal]:
     original_l_end = to_decimal(l_end)  # namely r
     iteration, m_for_l = 0, to_decimal(0)
     while l_end - l_start > tolerance and iteration<max_iterations:
         l = (l_start + l_end) / 2
         m_for_l = _find_m_given_l(l=l,r=original_l_end,alpha=alpha,
             preference_1=preference_1,cake_size=cake_size,epsilon=
             epsilon,tolerance=tolerance)

         searched_value = get_double_prime_for_interval(segments=
             preference_i,epsilon=epsilon,start=to_decimal(0),end=l,
             cake_size=to_decimal(cake_size))
         # Want v_i[(0, l)]= v_i[(m(l), r)]
         desired_value = get_double_prime_for_interval(segments=
             preference_i,epsilon=epsilon,start=m_for_l,end=
             original_l_end,cake_size=to_decimal(cake_size))
         if abs(searched_value - desired_value) <= tolerance:
              return l, m_for_l

         if searched_value < desired_value:
              l_start = l
         else:
              l_end = l
         iteration = iteration + 1
     return to_decimal((l_start + l_end) / 2), m_for_l
```

Listing 3.14: Binary Search Handle $(k,k') = (0,2)$

When $(k,k') = (0,3)$, none of the cuts are predetermined. For any given l, we can determine the unique cuts $m(l)$ and $r(l)$ such that $v_1(l,m(l)) = v_1(m(l),r(l)) = \alpha$. Subsequently, we adjust l to identify the possible interval where $v_i(0,l) = v_i(r(l),1)$

is satisfied. Analogously, the same procedure is applied to m and r. Ultimately, we determine the possible intervals for the cuts l, m, and r to precisely locate the exact positions of the cuts. It is important to note that after determining the possible intervals for the three cuts, we use the narrowest interval to identify the final cut positions, as defined in Listing 3. The function _handle_leftmost_rightmost is too cumbersome to include here due to space constraints. The full implementation can be found in the code repository[6].

```
def _find_best_cuts_by_range(lower_l: Decimal,upper_l: Decimal,
    lower_m: Decimal,upper_m: Decimal,lower_r: Decimal,upper_r:
    Decimal) -> List[Decimal]:
     return [upper_l, (lower_m + upper_m) / 2, lower_r]
```

Listing 3.15: Find Best Cuts by Range

3.6 Wrap Up

With all components of the algorithm in place, we can now proceed to finalise it[7]. Following the algorithm's flow outlined in Section 2.3.2, the first step is to determine whether an envy-free equipartition already exists. If an envy-free equipartition is found, then the desired ε-EF has been achieved. As Listing 3.16 shows. When a set of cuts (l,m,r) that appears promising for achieving an ε-EF division is identified we use the find_envy_free_allocation function to exhaustively check all possible envy-free allocations. For four agents, there are 24 possible allocations $(n! = 4! = 24)$. Once such an allocation is found, it is immediately returned.[8] The implementation of find_envy_free_allocation is provided in Listing B.8.

```
def alex_aviad(preferences: Preferences, cake_size: int, epsilon:
    Decimal, tolerance: Decimal) -> Dict[str, Any]:
     assert len(preferences) == 4, "Need 4 agents here"
     solution, steps = [], []
     # Find the equipartition by Agent1
     cuts = equipartition(preferences[0], to_decimal(cake_size),
         epsilon, to_decimal(0), to_decimal(cake_size), tolerance)
     solution = find_envy_free_allocation(cuts=cuts, num_agents=4,
         cake_size=to_decimal(cake_size), preferences=preferences,
         epsilon=epsilon)
```

[6] https://github.com/yongtenglei/treat_cake

[7] The Hollender-Rubinstein Algorithm is referred to as the alex_aviad algorithm in its implementation.

[8] If multiple allocations meet the criteria, the first one identified is returned.

```
    if solution is not None:
        return {"solution": solution, "steps": steps}
```

Listing 3.16: Hollender-Rubinstein Algorithm Part 1

If no envy-free equipartition is found, we proceed to the main loop of the algorithm. Prior to this, it is necessary to set the initial values for $\underline{\alpha}$ and $\bar{\alpha}$.

```
    ...
    # Set alpha_underline as the value of 1/4 cake
    alpha_underline = get_double_prime_for_interval(preferences[0],
        epsilon, to_decimal(0), cuts[0], to_decimal(cake_size))
    # Expected to be 1
    alpha_overline = get_double_prime_for_interval(preferences[0],
        epsilon, to_decimal(0), to_decimal(cake_size), to_decimal(
        cake_size))
    alpha = -1
```

Listing 3.17: Hollender-Rubinstein Algorithm Part 2

Next, we enter the main loop, which continues until $\bar{\alpha} - \underline{\alpha} \leq \varepsilon^4/12$ or until the loop has run 100 iterations[9]. It is important to note that Condition A is specifically satisfied within the interval $[\overline{\alpha_4}, \overline{\alpha_3}]$, while Condition B is met within the interval $[\overline{\alpha_4}, \overline{\alpha_2}]$. However, within a specific interval, both conditions can be simultaneously satisfied, at which point they are considered equivalent. In our implementation, we therefore omit the check for Condition B.

```
    ...
    condition_info = {
        "A": {"cuts": [], "k": -1, "alpha_underline": -1},
        "B": {"cuts": [], "k": -1, "k_prime": -1, "alpha_underline":
            -1},
    }
    meet_condition, counter = "", 0
    # Main loop
    while abs(alpha_overline - alpha_underline) > (epsilon**4 / 12)
        and counter <= 100:
        alpha = (alpha_underline + alpha_overline) / 2
        if to_decimal(0.25) <= alpha < Decimal("1") / Decimal("3"):
```

[9]The algorithm is generally expected to terminate once the exit condition 1 is met. The 100-iteration limit is imposed solely to ensure termination. According to our experiments, when ε is sufficiently small, even with this iteration limit, the algorithm may require an impractical amount of time to complete and result in undesirable outcome.

```
            meet_a, condition_a_info = check_condition_a(alpha=alpha
                , preferences=preferences, cake_size=to_decimal(
                cake_size), epsilon=epsilon, tolerance=tolerance)
            if meet_a:
                meet_condition = "A"
                condition_info["A"] = condition_a_info
                condition_info["A"]["alpha_underline"] =
                    alpha_underline
                alpha_underline = alpha
                counter += 1
                continue

        if to_decimal(0.25) <= alpha < to_decimal(0.5):
            meet_b, condition_b_info = check_condition_b(alpha=alpha
                , preferences=preferences, cake_size=to_decimal(
                cake_size), epsilon=epsilon, tolerance=tolerance)
            if meet_b:
                meet_condition = "B"
                condition_info["B"] = condition_b_info
                condition_info["B"]["alpha_underline"] =
                    alpha_underline
                alpha_underline = alpha
                counter += 1
                continue

        alpha_overline = alpha
        counter += 1
    ...
```

Listing 3.18: Hollender-Rubinstein Algorithm Part 3

After the main loop exits, we identify an ε-EF allocation using the final information that satisfies either Condition A or B at α, and then return the result.

```
    allocation = None
    if meet_condition == "A":
        allocation = find_allocation_on_condition_a(preferences=
            preferences, cake_size=to_decimal(cake_size), cuts=
            condition_info["A"]["cuts"], episilon=epsilon, k=
            condition_info["A"]["k"])
    elif meet_condition == "B":
        allocation = find_allocation_on_condition_b(cuts=
            condition_info["B"]["cuts"], cake_size=to_decimal(
```

```
            cake_size), episilon=epsilon, k=condition_info["B"]["k"],
             k_prime=condition_info["B"]["k_prime"], preferences=
            preferences)
    return {"solution": allocation, "steps": steps}
```

Listing 3.19: Hollender-Rubinstein Algorithm Part 4

Chapter 4

Discussion

4.1 Code Implementation Review

Overall, the Hollender-Rubinstein Algorithm has been successfully implemented. When all agents share the same evaluation functions, the algorithm can accurately and fairly allocate the cake among all agents, relying on effectively function of Equipartition and two types of binary searches. However, when entering the main loop, the algorithm becomes more sensitive to the inputs. For example, when the agents' evaluation functions lack certain distinctive features, it may be difficult to satisfy either of the two conditions, potentially leading to the failure of the algorithm to achieve a fair allocation. The expected path that consistently holds an invariant, such as Condition A - Condition B - Condition B - ... - Condition A, eventually leading to an ε-envy-free allocation, may only satisfy the invariant under specific conditions during the execution of algorithm. Thus, this implementation is not a universal algorithm applicable to all general evaluation functions. It is constrained by the agents' preferences, computation precision, and potential implementation errors.

4.1.1 Agents Preferences

Figure 4.1 illustrates a randomised preferences of four agents. Agent 1 evaluates the entire cake with an average value of 10, while the other agents evaluate the cake in the same way but with lower values. This will cause Condition A to fail consistently: If k is the rightmost piece, after we perform three left-to-right dichotomies, the last remaining piece k satisfies $v_1(r,1) \leq \alpha$. Since the evaluations of the other three agents are consistently lower than those of Agent 1 and are evenly distributed, we can never

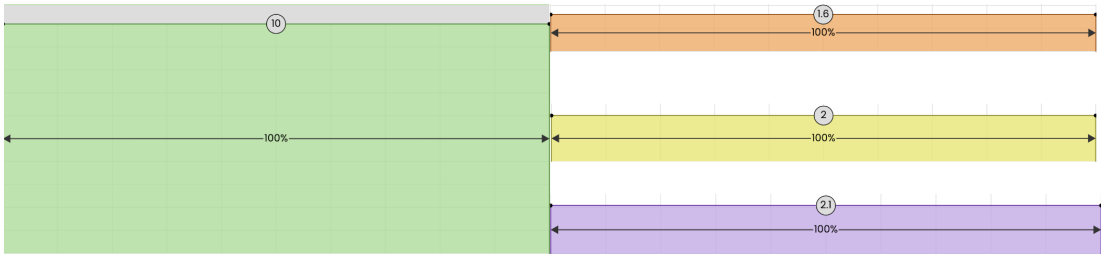


Figure 4.1: Agent Preferences Potentially Failing Conditions A and B.

find a cut r such that $v_i(r,1) \geq \alpha$, where $i \in \{2,3,4\}$. Such additional checks will also fail in the other three cases, analogously.

For Condition B, it can only be satisfied when $\alpha \in [\alpha_4^=, \alpha_2^=] = [0.25, 0.5]$. Taking the tightest case of $\alpha = 0.25$ as an example, half of the cake will be equally divided by Agent 1, and a cut point can be found to make the other half of the cake equal in value from Agent i's perspective. However, since the evaluations of the possible agents $i \in \{2,3,4\}$ are constant, we cannot satisfy the sub-condition 2 of Condition B, $v_i(X_k) = v_i(X_{k'}) \geq \max_t v_i(X_t)$, within a smaller or equal interval, due to the Monotonicity of the evaluation functions. This will cause the failure to meet the Condition B.

4.1.2 Computation Precision and Tolerance

The algorithm relies heavily on floating point calculations, as ε is expected to be a very small number and the evaluation functions have values in the range $[0,1]$, the algorithm uses the Python library decimal.Decimal with a precision of 15 floating point numbers to enhance the processing. Specifically, the precision plays an important role in the calculation of the modified evaluation functions v_i''. Modified evaluation functions v_i'' is the interpolation of v_i' on the ε-grid, and only enough precision can make such a small step movement meaningful. In addition, the conversion between the number of cake segments and the interval $[0,1]$, the normalisation and de-normalisation of evaluation functions, all depend on high precision conversion to prevent the accumulation of errors that could lead to unwanted results.

In high precision scenarios, comparisons between floating-point numbers can be challenging and require careful handling. Table 4.1 illustrates two cases with different tolerances: a more lenient tolerance is used when verifying the accuracy of the positions of cut points, assuming the program is functioning correctly, while a stricter tolerance is needed during binary search to ensure precision in finding cuts. However, not all cases could be handled correctly, leading to unintended deviations of the algorithm from the

	Left	Right	Pass
Verify cuts l and r: $v\_1(0,l) = \alpha = v\_1(r,1)$	0.25641369	0.26039671	Yes
Check searched_value = desired_value in binary search	0.25741369	0.25741453	No

Table 4.1: Two Scenarios Requiring Different Tolerance levels for accurate computation.

desired result.

4.1.3 Potential Implementation Errors and Decisions

The Hollender-Rubinstein Algorithm is based on an intuitive idea, yet it requires an extremely complex implementation. Although the code implementation was tested extensively, with a reported code coverage of 71% in Appendix C.1, there were still unanticipated boundary cases that were not tested. It is worth noting that while 85% of the code related to Condition A has been tested, only 16% of the code for Condition B was covered due to the intricate nature of applicable user inputs and the complexity of Condition B. However, the behaviour of the code was verified through logging and generally aligned with expectations.

There are two important decisions in the implementation that deviate from the description of Hollender and Rubinstein's[12] work. The first function is the equipartition. Hollender and Rubinstein [12] suggest that given any cut l, one should then search for the positions of cuts m and r such that $v_1(0,l) = v_1(l,m) = v_1(m,r)$. An additional evaluation request is then used to verify that if $v_i(r,1)$ has the same value. Next, adjust the position of l, if necessary. Then, we identified the intervals on the ε-grid that cut l must lie. Analogously, intervals where m and r are located can also be found. Finally, this information is used to determine the exact locations of (l,m,r). This implementation has a complexity of $O(\log^2(1/\varepsilon))$. In contrast, this project is based on the fact that, since the value of the entire cake is normalised to 1, each piece in the equipartition is worth 0.25. Based on this, we only need to use binary search to precisely and sequentially determine the cuts (l, m, r), satisfying $v_1(0,l) = v_1(l,m) = v_1(m,r) = 0.25$. The remaining cuts must then also satisfy $v_1(r,1) = 0.25$. In this way, we achieve equipartition with a complexity of $O(\log(1/\varepsilon))$. Experiments show that this implementation is complete. On the other hand, another function addresses the case where Condition B involves a piece between k and k'. As discussed in Section 3.5.2, the original work suggested finding the exact locations of the cuts after determining the interval in which all cuts lie on the ε-grid. However, our experiments have shown that such a interval is often

negligible. Therefore, in our implementation, we use the first suitable cut found directly as an approximate position. In the case of k and k' are leftmost and rightmost pieces, we adhere to the original implementation.

Consequently, potential implementation errors, variations in the implementation approach, the complexity of floating-point computation, and the nature of user inputs, all of these factors combined may lead to potential failure of the algorithm. Additionally, the complexity of the implementation and the difficulty in tracing the sources of errors make it challenging to find a solution within a limited time period.

4.2 Platform Integration

The aim of this project is to introduce the Hollender-Rubinstein Algorithm to the Fair Slice platform, expanding its functionality from two and three agents to a four-agent scenario, while maintaining its interactive and user-friendly features. As shown in Figure 4.2, users can easily add an agent and draw their preferences for all agents before seeking an allocation through the Hollender-Rubinstein Algorithm. Figure 4.3 presents a scenario where four individuals share the same evaluation function, resulting in an ε-envy-free division that relies solely on equipartition. Fair Slice displays the proportion of the total cake assigned to each agent, as well as each agent's preference for each piece, in two distinct and intuitive formats. Additionally, a visual representation of the allocation process is provided to enhance the user's understanding of how the division is achieved.

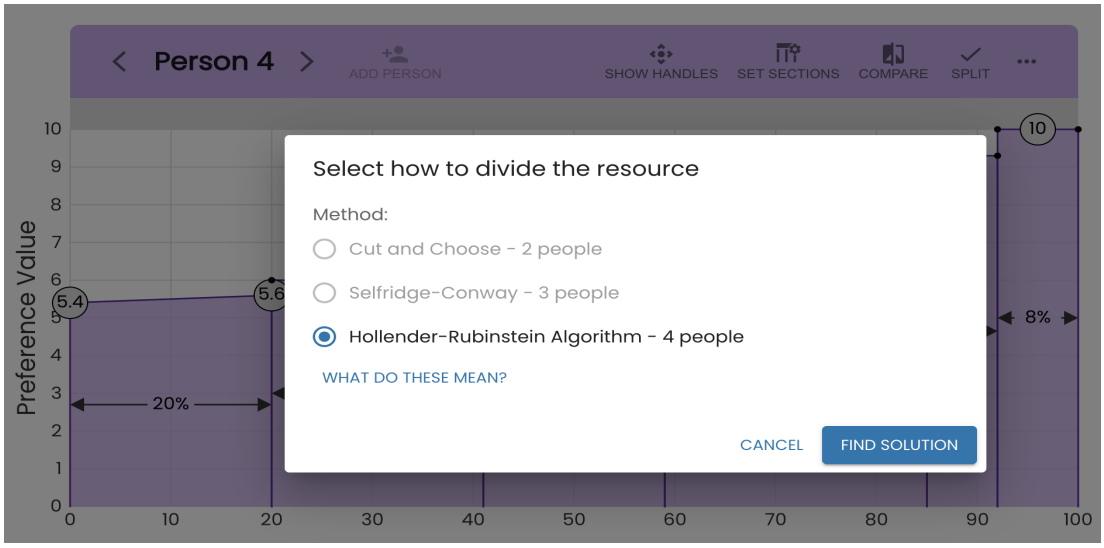


Figure 4.2: Users Can Easily Extend the Scenario to Include a Fourth agent.

For the general case, Figure 4.4 illustrates the experimentally selected preferences of agents with specific tendencies, as shown in Figure 4.4a, where Agent i has a clear preference for the ith piece and exhibits only limited interest in the other pieces. For instance, Agent 2 assigns the highest valuation only to the vicinity of the expected second piece of the cake, showing little interest in the remaining pieces. Such preferential valuations facilitate the smooth progression of the main loop. The final allocation is displayed in Figure 4.4b. As shown, this is not an envy-free allocation, as Agent 1 envies Agent 2's second piece, which is valued 30% for Agent 1 compared to the 26.4% allocated. Therefore, this represents an ε-envy-free allocation, where $\varepsilon = 1e^{-5}$ in the setting. Additionally, an interesting observation is that for Agent 3, the entire cake is worth 99.9%, whereas for the other agents, it is 100%, reflecting the minor error introduced by floating-point calculation.

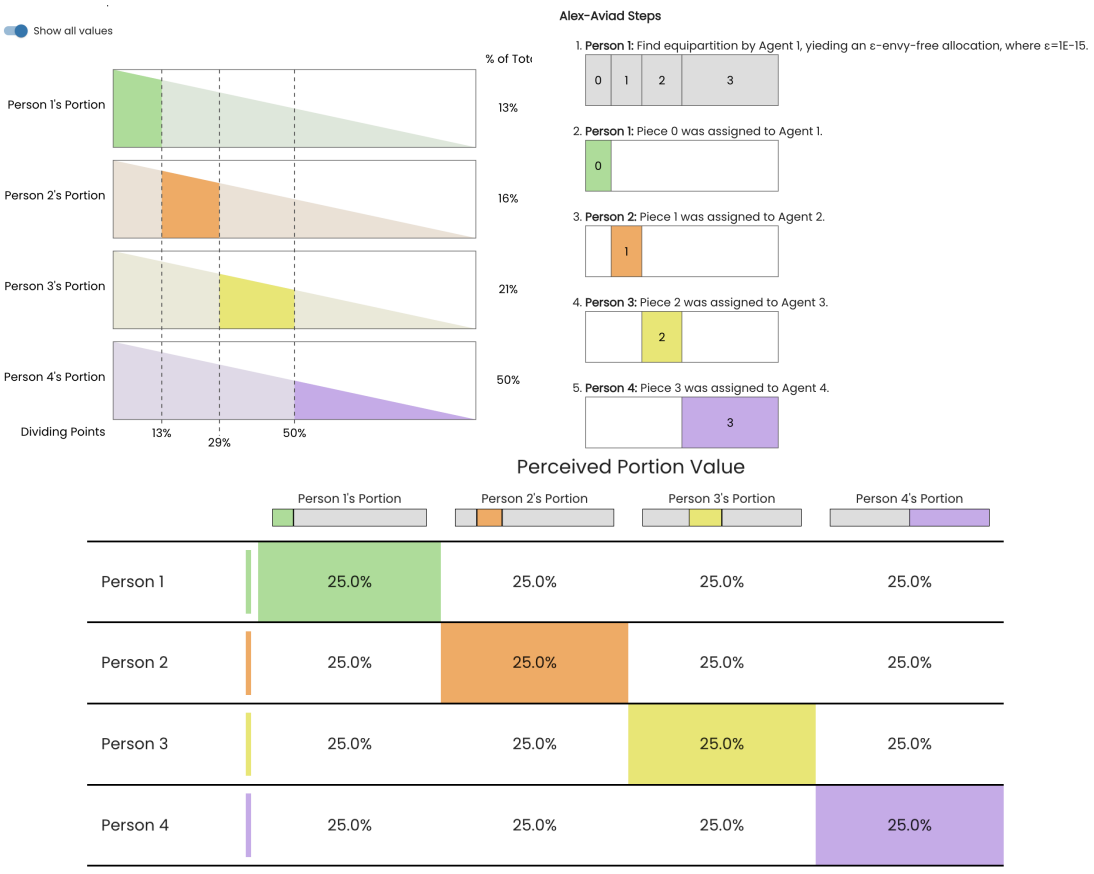
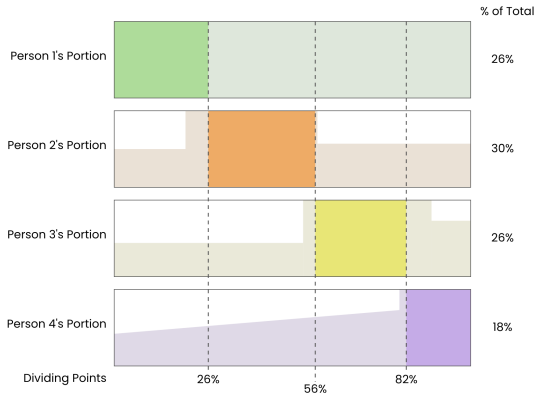
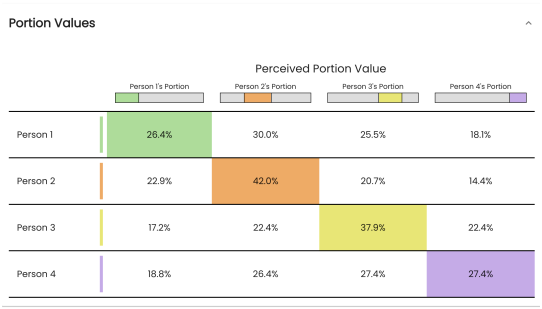


Figure 4.3: Example with the Same Evaluation Functions

Fair Slice is primarily an educational tool, with previous work effectively introducing concepts like fair division, Cut & Choose, and the Selfridge-Conway Method in an engaging and accessible manner. This work builds on that foundation, as illustrated



(a) Experimentally Selected Preferences



(b) Result of General Case Yielding ε-envy-free Allocation Where $\varepsilon = 1e^{-5}$

Figure 4.4: Example with Experimentally Selected Evaluation Functions

The Hollender-Rubinstein Algorithm

When it comes to fairly dividing things among four people, it gets tricky. It's no longer guaranteed that everyone will be completely envy-free, and finding such a division can be very difficult to compute. To address this challenge, the concept of ε-Envy-free was introduced, which improves the efficiency of finding fair divisions for three or even four people.

A division is considered ε-Envy-free if, by allowing a small amount of envy (denoted as ε), each person i who receives a portion X_i is satisfied that their share isn't worse than someone else's share X_j by more than ε. In other words, $v_i(X_i) \geq v_i(X_j) - \varepsilon$.

After decades of research, many methods for fairly dividing things among four people have been developed. However, two big challenges still haven't been solved: the need to split resources into countless tiny pieces and the impractical amount of time it takes to do the calculations.

Recent work by Hollender and Rubinstein proposed a promising framework to address this problem by resulting continuous envy-free fair division within a certain time complexity. This means the cake will be divided into 4 contiguous pieces, with the final result determined after $O(\log^3(1/\varepsilon))$ value queries.

This Algorithm we call it as **Hollender-Rubinstein Algorithm.**

Let's see it together!

Core Idea of Hollender-Rubinstein Algorithm

The core idea of the Hollender-Rubinstein Algorithm is to maintain an invariant throughout its execution, parameterised by $\alpha \in [0, 1]$. The algorithm begins with an equipartition by Agent 1, and as α increases monotonically, it follows a continuous path that consistently holds the invariant. This path is guaranteed to eventually terminate, resulting in an ε-EF allocation. The desired invariant is one of the following two conditions:

- **Condition A:** Agent 1 is indifferent among its top three favourite pieces, and the remaining piece is (weakly) preferred by at least two of the other three agents.
- **Condition B:** Agent 1 is indifferent between its two favourite pieces, and each of the remaining two pieces is (weakly) preferred by at least two of the other three agents.

Equipartition refers to the division of a cake into four pieces, each with equal value from the perspective of Agent 1, denoted as $\alpha_4^=$. It has been proved that such a unique equipartition always exists and can be efficiently determined. This equipartition satisfies Condition A, which serves as the initial step in the Hollender-Rubinstein Algorithm.

Let's begin by exploring how to achieve an ε-Envy-free allocation using Equipartition. This is the relatively straightforward case!

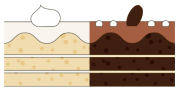
Figure 4.5: Interactive course: Introduction of Hollender-Rubinstein Algorithm

Equipartition of the Hollender-Rubinstein Algorithm

Let's see how to create an ε-envy-free outcome by Equipartition.

Here is the problem again:

Here we have two cakes again.



But we need to split it among **4** people this time, they treat these cake equally.

Aki likes both vanilla and chocolate

Since all four friends use the same evaluation equation, we can achieve a fair division through Equipartition. However, this division is *not envy-free*, as each friend may *envy* the others' portions. By introducing a small approximation factor ε, we can achieve an ε-envy-free allocation instead.

Figure 4.6: Interactive course: Equipartition of Hollender-Rubinstein Algorithm

in Figures 4.5 and 4.6, by introducing a new character, Derek, to demonstrate the elegance of fair division among four agents. The course page about Condition Handling is provided in Appendix C.2.

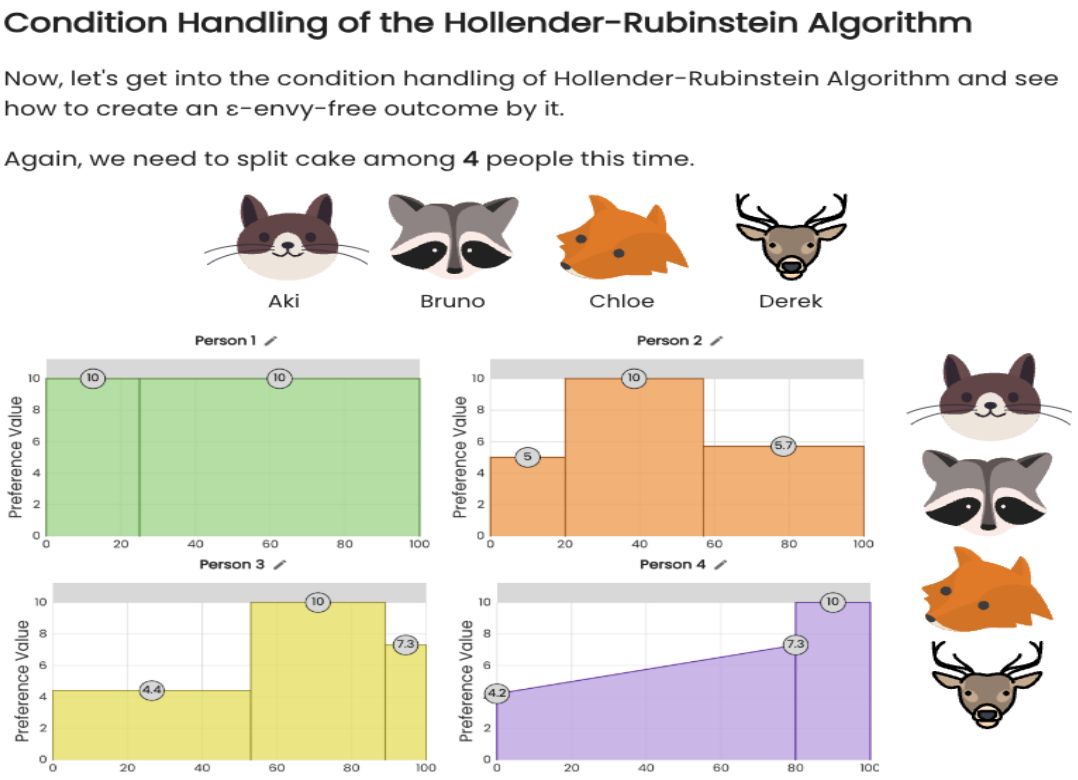


Figure 4.7: Condition Handling Course Part 1

4.3 Limitations

The limitations of this project can be viewed in two parts, the implementation of the code and its integration with Fair Slice. As previously discussed, although our algorithm can accurately and efficiently handle scenarios where four agents share the same evaluation functions, and the results depend solely on equipartition, issues may arise when different ε values, tolerances, or user inputs are introduced. This can cause the algorithm to stagnate, produce undesired results, or even fail to produce any results if neither Condition A nor Condition B is satisfied. For example, while an equipartition may yield an instantaneous result, the main loop could take several minutes to complete dozens of rounds with an uncertain outcome. The algorithm's reliance on critical floating-point calculations makes it vulnerable to inappropriate settings, leading to potential stagnation. The fact that the algorithm operates with specific ε values

and tolerances indicates that the implementation is not entirely robust. Additionally, the code related to Condition B has only 16% test coverage. Although its behaviour has been verified through logging, there remains a possibility of inaccuracies in the implementation.

For Fair Slice integration, the Hollender-Rubinstein Algorithm provides limited flexibility, as it does not allow users to set the value of ε or the tolerance. A strict tolerance improves the algorithm's accuracy, while a more lenient tolerance speeds up the response time. Additionally, the interactive educational page for Equipartition employs a local solution storage solution, which preloads sample evaluation functions and loads the solution locally, thus limits user interaction. This setup reduces the overall interactivity and engagement of the course. Furthermore, because Condition Handling is highly sensitive to input, we still provide users with sample evaluation functions and only offer textual descriptions of the algorithmic process and then display the results. The educational value of this approach could be further evaluated.

4.4 Future Work

Firstly, the robustness of the algorithm should be guaranteed. The algorithm should be able to dynamically adapt to any given value of ε and tolerance to ensure the smooth running of the programme and the accuracy of the results, which requires further refinement of the codebase. Additionally, the failure of the algorithm due to user inputs contradicts Hollender and Rubinstein's theory that, given any α, the algorithm will hold Condition A and Condition B until it terminates, yielding a desired ε-envy-free allocation. The conflict between theory and practice needs to be resolved with further research and more testing.

Furthermore, if the Hollender-Rubinstein Algorithm is not a generalised four-agent fair allocation solution. An alternative approach, based on piecewise-constant valuations for n agents, as suggested by the supervisor, Aris Filos-Ratsikas, could be integrated into Fair Slice to extend its functionality to n-agent scenarios, though it may reduce the flexibility in preference expression.

Once the Hollender-Rubinstein Algorithm is made more robust, users could be given the ability to input their own preferred parameters to explore custom fair division solutions. Additionally, the interactive educational interface for the Hollender-Rubinstein Algorithm should be enhanced to be more engaging and interactive, potentially by offering a playground environment for users.

Chapter 5

Conclusions

In summary, this project investigated the design of four-agent fair division in greater depth, with a particular focus on the work of Hollender and Rubinstein [12]. By thoroughly exploring and understanding their proposed theory, it was successfully translated into practical, executable code and seamlessly integrated into Fair Slice Ernst [11], a fair division visualisation tool. This integration extended the tool's functionality to four-agent scenarios, complemented by easy-to-understand, interactive courses that further aligned with its mission as an educational presentation tool.

In this project, we have presented a clear and intuitive overview of the newly integrated Hollender-Rubinstein Algorithm, highlighting the key aspects necessary for its comprehensive understanding. Chapter 2 provided the essential background on fair division within the context of divisible resources, commonly referred to as the cake-cutting problem. It emphasised the properties of evaluation functions, computational approaches, and introduced the most widely recognised fairness criterion, envy-freeness, along with its relaxation, ε-envy-freeness, which was adopted in this project. We then reviewed the history of divisible fair division, highlighting the achievements in four-agent scenarios and the challenges that remain. This set the stage for introducing the Hollender-Rubinstein Algorithm, a recently proposed and promising fair division method. In Chapter 3, we comprehensively detailed the theory and implementation of the Hollender-Rubinstein Algorithm, breaking it down into six key components to provide readers with a deeper insight into the theoretical foundations and implementation choices. The subsequent discussion evaluated existing implementations, identifying gaps between theoretical concepts and real-world applications, the challenges encountered during implementation, and the intensive deviations from the original work. Additionally, we presented the results of integrating the algorithm with

Fair Slice. Finally, the project's limitations were discussed, along with suggestions for potential future work.

As a final conclusion, it has been a great honour to write this dissertation, showcasing all the efforts made over the past two months. As demonstrated in this dissertation, it is dedicated to introducing the reader to fair division in a friendly manner and helping them understand the exciting and promising work of the Hollender-Rubinstein Algorithm. The work presented aligns closely with the goals of Fair Slice, which aims to bring the field of fair division to the attention of the general public and offer meaningful insights to scholars in this field. To fully realise this vision, it will require ongoing research and sustained effort.

Bibliography

[1] A Keith Austin. “Sharing a cake”. In: *The Mathematical Gazette* 66.437 (1982), pp. 212–215.

[2] Haris Aziz and Simon Mackenzie. “A discrete and bounded envy-free cake cutting protocol for any number of agents”. In: *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2016, pp. 416–427.

[3] Haris Aziz and Simon Mackenzie. “A discrete and bounded envy-free cake cutting protocol for four agents”. In: *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*. 2016, pp. 454–464.

[4] Dinesh Kumar Baghel, Vadim E. Levit, and Erel Segal-Halevi. *Fair Division Algorithms for Electricity Distribution*. 2022. arXiv: 2205.14531 [cs.GT]. URL: https://arxiv.org/abs/2205.14531.

[5] Steven Brams, Alan Taylor, and William Zwicker. “A moving-knife solution to the four-person envy-free cake-division problem”. In: *Proceedings of the american mathematical society* 125.2 (1997), pp. 547–554.

[6] Steven J Brams and Alan D Taylor. “An envy-free cake division protocol”. In: *The American Mathematical Monthly* 102.1 (1995), pp. 9–18.

[7] Steven J Brams and Alan D Taylor. *Fair Division: From cake-cutting to dispute resolution*. Cambridge University Press, 1996.

[8] Steven J Brams and Alan D Taylor. *The win–win solution: Guaranteeing fair shares to everybody*. WW Norton & Company, 2000.

[9] Simina Brânzei and Noam Nisan. “The query complexity of cake cutting”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 37905–37919.

[10] John Cloutier, Kathryn L Nyman, and Francis Edward Su. “Two-player envy-free multi-cake division”. In: *Mathematical Social Sciences* 59.1 (2010), pp. 26–37.

[11] Andy Ernst. *Fair Slice*. 2024. URL: https://fairslice.netlify.app/.

[12] Alexandros Hollender and Aviad Rubinstein. “Envy-free cake-cutting for four agents”. In: *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2023, pp. 113–122.

[13] Bronislaw Knaster. “Sur le problème du partage pragmatique de H. Steinhaus”. In: *Annales de la Societé Polonaise de Mathematique*. Vol. 19. 1946, pp. 228–230.

[14] Hervé Moulin. *Fair division and collective welfare*. MIT press, 2004.

[15] Kathryn Nyman, Francis Edward Su, and Shira Zerbib. “Fair division with multiple pieces”. In: *Discrete Applied Mathematics* 283 (2020), pp. 115–122. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2019.12.018.

[16] Ariel D. Procaccia. “Cake Cutting Algorithms”. In: *Handbook of Computational Social Choice*. Ed. by Felix Brandt et al. Cambridge University Press, 2016, pp. 311–330.

[17] Kirk Pruhs and Gerhard J Woeginger. “Divorcing made easy”. In: *Fun with Algorithms: 6th International Conference, FUN 2012, Venice, Italy, June 4-6, 2012. Proceedings 6*. Springer. 2012, pp. 305–314.

[18] Howard Raiffa. *The art and science of negotiation*. Harvard University Press, 1982.

[19] Jack Robertson and William Webb. *Cake-cutting algorithms: Be fair if you can*. AK Peters/CRC Press, 1998.

[20] Jörg Rothe et al. *Economics and computation*. Vol. 4. Springer, 2015.

[21] Erel Segal-Halevi et al. “Envy-free division of land”. In: *Mathematics of Operations Research* 45.3 (2020), pp. 896–922.

[22] Erel Segal-Halevi et al. “Fair and square: Cake-cutting in two dimensions”. In: *Journal of Mathematical Economics* 70 (2017), pp. 1–28. ISSN: 0304-4068. DOI: https://doi.org/10.1016/j.jmateco.2017.01.007.

[23] Forest W Simmons and Francis Edward Su. “Consensus-halving via theorems of Borsuk-Ulam and Tucker”. In: *Mathematical social sciences* 45.1 (2003), pp. 15–25.

[24] Splitwise. *Fairness Calculators – Rent Calculator*. https://www.splitwise.com/calculators/rent. (Visited on 07/23/2024).

[25] H. Steinhaus. “Sur la division pragmatique”. In: *Econometrica* 17 (1949), pp. 315–319. ISSN: 00129682, 14680262. URL: http://www.jstor.org/stable/1907319 (visited on 08/10/2024).

[26] Hugo Steinhaus. “The problem of fair division”. In: *Econometrica* 16 (1948), pp. 101–104.

[27] Francesco Tajani and Pierluigi Morano. “A model for the elaboration of fair divisional projects in inheritance disputes”. In: *Property Management* 36.2 (2018), pp. 186–202.

Appendix A

Complex Valuation Functions

A.1 Complexity of Piecewise Linear valuation functions

Given the two sides of the portion, the desired value (the area of the portion), and the slope, Ernst [11] obtains the following formula to determine the cut point:

$$\frac{-\text{Height} \pm \sqrt{\text{Height}^2 + 2 \cdot \text{Slope} \cdot \text{Area}}}{\text{Slope}}$$

While traditional solutions result large inaccuracy in small intervals, this optimised formula can produce the correct answer in 12 decimal places in such cases. The *decimal.Decimal* from the *decimal* library in Python will be used in this project to further improve it, and the precision is set to 15 decimal places. Through experimental experiments, this precision is balanced between computational efficiency and result accuracy.

A.2 More Complex Forms of Valuation Functions

More complex valuation functions, as illustrated in Figure A.1, are challenging to integrate into the UI, and the significance of representing such preferences still requires further investigation. However, these complex preferences can be approximated using smaller piecewise linear intervals.

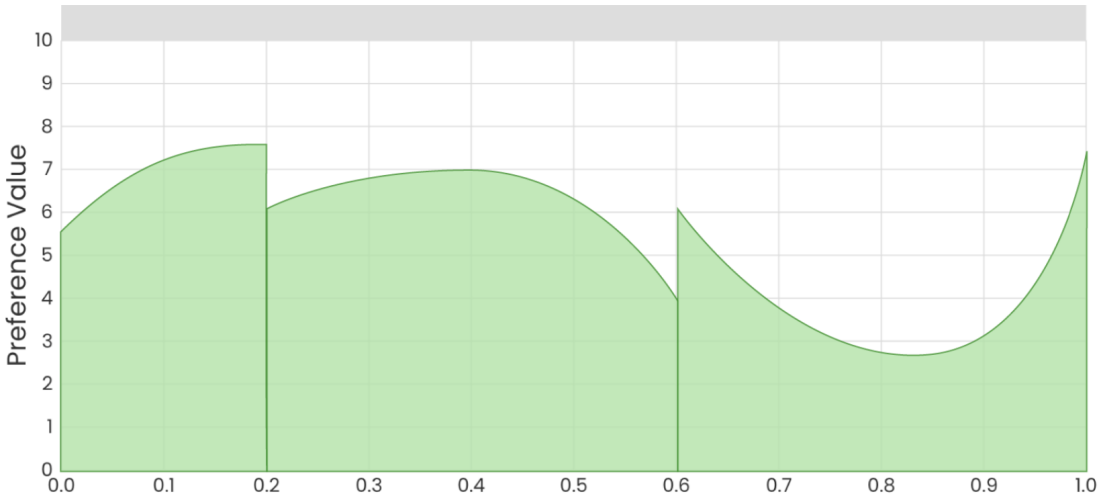


Figure A.1: Concave and convex valuation

Appendix B

Code Implementation

B.1 Original Evaluation Functions

```
def get_value_for_interval(
    segments: List[Segment], start: Decimal, end: Decimal
) -> Decimal:
    """
    Returns the total value of an interval,
    even if covers several segments or splits segments in half.
    """
    total = to_decimal(0)
    start = to_decimal(start)
    end = to_decimal(end)

    for seg in segments:
        if seg.end <= start or seg.start >= end:
            # this segment not relevant
            continue

        total += _measure_partial_segment(seg, start, end)
    return total


def _measure_partial_segment(seg: Segment, start: Decimal, end:
    Decimal) -> Decimal:
    """
    Measures the area of a segment
    Works with flat or sloped sections, whole numbers and decimals.
    """
```

```
    start = to_decimal(start)
    end = to_decimal(end)

    start_cap = max(start, to_decimal(seg.start))
    end_cap = min(end, to_decimal(seg.end))
    measuring_width = end_cap - start_cap

    if measuring_width <= 0:
        # Nothing to measure
        return to_decimal(0)
    if seg.start_value == seg.end_value:
        # Flat section
        return seg.start_value * measuring_width
    else:
        # Sloped section
        segment_width = seg.end - seg.start
        slope = (seg.end_value - seg.start_value) / segment_width
        start_val = seg.start_value + slope * (start_cap - seg.start
            )
        end_val = seg.end_value - slope * (seg.end - end_cap)
        avg_value = (start_val + end_val) / 2
        return measuring_width * avg_value
```

Listing B.1: Original Evaluation Functions

B.2 Normalisation and De-normalisation

```
def norm(v: Decimal, whole_cake_value: Decimal) -> Decimal:
    """
    Adjust value from infinity to [0, 1]
    """
    whole_cake_value = to_decimal(whole_cake_value)
    v = to_decimal(v)

    assert (
        to_decimal(0) <= v <= whole_cake_value
    ), f"v must be greater than or equal to 0 and less than whole
        cake value: {whole_cake_value}, got {v}"

    if whole_cake_value == 0:
        return to_decimal(0)
```

```
    return v / whole_cake_value


def de_norm(v: Decimal, whole_cake_value: Decimal) -> Decimal:
    """
    De-normalize a value from [0, 1] back to [0, whole_cake_value].
    """
    v = to_decimal(v)
    whole_cake_value = to_decimal(whole_cake_value)

    assert (
        Decimal(0) <= v <= Decimal(1)
    ), f"Normalized value must be between 0 and 1, got {v}"

    if whole_cake_value == 0:
        return to_decimal(0)

    return v * whole_cake_value
```

Listing B.2: Normalisation and De-normalisation

B.3 Final Modified Valuation Functions

```
def overline(x, delta, tolerance=Decimal("1e-10")) -> Decimal:
    assert 0 <= x <= 1, f"got {x}, expect it between [0, 1]"
    if x < delta or x == 0:
        return delta

    if x == 1:
        return x

    v = (x / delta).to_integral_value(rounding="ROUND_CEILING") *
       delta

    # If x is exactly a multiple of delta, step up to the next
       multiple
    # considering floating point precision issues
    if abs(x % delta) < tolerance or abs(delta - (x % delta)) <
       tolerance:
        v += delta
```

```
    return min(v, to_decimal(1))
```

Listing B.3: Finding the smallest multiple in the δ-grid that is greater than or equal to x.

```
def _v_double_prime(segments: List[Segment], delta: Decimal, a:
    Decimal, b: Decimal, cake_size: Decimal) -> Decimal:
    # Letting delta := epsilon, so,
    # any epsilon-envy-free allocation for (v_double_prime) is 5*
        epsilon-envy-free for (v_prime) for each agent.

    a_unit = to_decimal(scale_to_unit(a, cake_size))
    b_unit = to_decimal(scale_to_unit(b, cake_size))
    delta = to_decimal(delta)

    # Get the grid points around a and b
    a_underline_unit = underline(a_unit, delta)
    a_overline_unit = overline(a_unit, delta)
    b_underline_unit = underline(b_unit, delta)
    b_overline_unit = overline(b_unit, delta)
    assert a_underline_unit <= a_unit <= a_overline_unit, "Wrong
        grid points"
    assert b_underline_unit <= b_unit <= b_overline_unit, "Wrong
        grid points"

    a_underline = scale_back_from_unit(a_underline_unit, cake_size)
    a_overline = scale_back_from_unit(a_overline_unit, cake_size)
    b_underline = scale_back_from_unit(b_underline_unit, cake_size)
    b_overline = scale_back_from_unit(b_overline_unit, cake_size)

    v_prime_a_under_b_over = _v_prime(segments, delta, a_underline,
        b_overline, cake_size)
    v_prime_a_over_b_under = _v_prime(segments, delta, a_overline,
        b_underline, cake_size)

    if a_overline_unit - a_unit >= b_unit - b_underline_unit:
        v_prime_a_under_b_under = _v_prime(segments, delta,
            a_underline, b_underline, cake_size)
        v_double_prime = (
            (((a_overline_unit - a_unit) - (b_unit -
                b_underline_unit)) / delta)
            * v_prime_a_under_b_under
            + ((b_unit - b_underline_unit) / delta) *
```

```
                v_prime_a_under_b_over
            + ((a_unit - a_underline_unit) / delta) *
                v_prime_a_over_b_under
        )

        if (a_unit == 0 and (a_unit - a_underline_unit) / delta *
            v_prime_a_over_b_under == 0):
            # If start from 0, need to compensate the last term
            v_double_prime += v_prime_a_over_b_under

        return v_double_prime
    elif a_overline_unit - a_unit <= b_unit - b_underline_unit:
        v_prime_a_over_b_over = _v_prime(segments, delta, a_overline
            , b_overline, cake_size)
        v_double_prime = (
        (((b_unit - b_underline_unit) - (a_overline_unit - a_unit))
            / delta)
            * v_prime_a_over_b_over
            + ((a_overline_unit - a_unit) / delta) *
                v_prime_a_under_b_over
            + ((b_overline_unit - b_unit) / delta) *
                v_prime_a_over_b_under
        )

        return v_double_prime

    raise ValueError("Should not reach here")
```

Listing B.4: Final modified v_i'' valuation functions

```
def scale_back_from_unit(a: Decimal, cake_size: Decimal) -> Decimal:
    """
    Adjust cut point value from [0, 1] back to [0, cake_size]
    """
    assert (
        to_decimal(0) <= a <= to_decimal(1)
    ), f"a must be greater than or equal to 0 and less than 1, to
        transform back to [0, cake_size({cake_size})], got {a}, "

    if cake_size == 1:
        return to_decimal(a)
    elif cake_size == 0:
        return to_decimal(0)
```

```
    return to_decimal(a) * to_decimal(cake_size)
```

Listing B.5: Coverting segments range back from $[0, 1]$ to $[0, \infty]$.

```
def get_double_prime_for_interval(segments: List[Segment], epsilon:
    Decimal, start: Decimal, end: Decimal, cake_size: Decimal) ->
    Decimal:
    assert 0 <= start <= end, "start or end out of range"

    # Make sure using Decimal
    epsilon = to_decimal(epsilon)
    start = to_decimal(start)
    end = to_decimal(min(end, cake_size))
    cake_size = to_decimal(cake_size)

    tolerance = to_decimal("1e-10")

    # Only one segment
    if end <= 1:
        return _safe_double_prime(
            _v_double_prime(segments, epsilon, start, end, cake_size
                ),
            tolerance=tolerance,
        )

    # Multi-segments
    total = to_decimal(0)

    start_int = int(start)
    end_int = int(end)

    if start == to_decimal(start_int) and end == to_decimal(end_int)
        :
        return _safe_double_prime(
            _v_double_prime(segments, epsilon, start, end, cake_size
                ),
            tolerance=tolerance,
        )

    # Incomplete start segment
    if start != to_decimal(start_int):
        first_segment_end = to_decimal(min(end, to_decimal(start_int
            + 1)))
```

```
        total += _v_double_prime(segments, epsilon, start,
            first_segment_end, cake_size)
        start_int += 1

    # Complete middle segments
    for mid in range(start_int, end_int):
        mid_start = to_decimal(mid)
        mid_end = to_decimal(mid + 1)
        if mid_end > end:
            mid_end = end
        total += _v_double_prime(segments, epsilon, mid_start,
            mid_end, cake_size)

    # Incomplete end segment
    if end > to_decimal(end_int):
        last_segment_start = to_decimal(end_int)
        total += _v_double_prime(segments, epsilon,
            last_segment_start, end, cake_size)

    return _safe_double_prime(total, tolerance=tolerance)
```

Listing B.6: Wrapper function to calculate the value of v_i'' for any given interval

B.4 Binary Search

```
def _binary_search_right_to_left(preference: List[Segment],
    cake_size: Decimal, epsilon: Decimal, start: Decimal, end:
    Decimal, target: Decimal, tolerance: Decimal = to_decimal(1e-10),
     max_iterations: int = 1000) -> Decimal:
    full_cake = get_double_prime_for_interval(
        segments=preference,
        epsilon=epsilon,
        start=start,
        end=end,
        cake_size=cake_size,
    )
    if full_cake < target:
        return start

    original_end = end
    iteration = 0
```

```
    while end - start > tolerance and iteration < max_iterations:
        mid = to_decimal((start + end) / 2)
        searched_value = get_double_prime_for_interval(segments=
            preference, epsilon=epsilon, start=mid, end=original_end,
             cake_size=to_decimal(cake_size))

        if abs(searched_value - target) < tolerance:
            return mid

        if searched_value < target:
            end = mid
        else:
            start = mid

        iteration = iteration + 1
    return to_decimal((start + end) / 2)
```

Listing B.7: Binary Search From Right to Left.

B.5 Find Envy-Free Allocation

It relies on Listing B.9 and Listing B.10.

```
def find_envy_free_allocation(
    cuts: List[Decimal],
    num_agents: int,
    cake_size: Decimal,
    preferences: Preferences,
    epsilon: Decimal,
) -> List[AssignedSlice]:
    cake_size = to_decimal(cake_size)

    for allocation in generate_all_possible_allocations(cuts,
        num_agents):
        envy_free_allocation = []
        for agent_id, slices in enumerate(allocation):
            for slice_index in slices:
                if slice_index == 0:
                    start = 0
                else:
                    start = cuts[slice_index - 1]
```

```
                if slice_index == len(cuts):
                    end = cake_size
                else:
                    end = cuts[slice_index]
                unassigned_slice = cut_slice(
                    preferences=preferences,
                    cake_size=to_decimal(cake_size),
                    epsilon=epsilon,
                    start=to_decimal(start),
                    end=to_decimal(end),
                    id=slice_index,
                    note=None,
                )
                envy_free_allocation.append(unassigned_slice.assign(
                    agent_id))
        if check_if_envy_free(num_agents, envy_free_allocation):
            return envy_free_allocation
    return None
```

Listing B.8: Find Envy-Free Allocation.

```
def generate_all_possible_allocations(cuts: List[Decimal],
    num_agents: int):
    slices = list(range(len(cuts) + 1))
    assert len(slices) == num_agents

    for perm in permutations(slices, num_agents):
        allocation = [[] for _ in range(num_agents)]
        for i, slice_index in enumerate(perm):
            allocation[i % num_agents].append(slice_index)
        yield allocation
```

Listing B.9: Generate All Possible Allocation.

```
def cut_slice(
    preferences: Preferences,
    cake_size: Decimal,
    epsilon: Decimal,
    start: Decimal,
    end: Decimal,
    id: int,
    note=None,
) -> FrozenUnassignedSlice:
```

```
    if start > end:
        raise ValueError(
            f"Start cannot be before end. Start {start}, end {end},
               preferences {str(preferences)}"
        )

    values = [
        de_norm(
            v=get_double_prime_for_interval(
                segments, epsilon, start, end, cake_size=cake_size
            ),
            whole_cake_value=get_value_for_interval(
                segments,
                to_decimal(0),
                to_decimal(cake_size),
            ),
        )
        for segments in preferences
    ]

    return FrozenUnassignedSlice(start=start, end=end, values=values
       , id=id, note=note)
```

Listing B.10: Cut Slice.

B.6 Condition B

```
def _handle_adjacent(
    k: int,
    k_prime: int,
    alpha: Decimal,
    preference_1: List[Segment],
    preference_i: List[Segment],
    epsilon: Decimal,
    cake_size: Decimal,
    tolerance: Decimal,
) -> List[Decimal]:
    # if k, k' = (0, 1)
    #     0         1         2             3
    # [0, l] | [l, m] | [m, r] | [r, cake_size]
    #    (3       3)          2           1
```

```
    if k == 0 and k_prime == 1:
        r = _binary_search_right_to_left(
            preference=preference_1,
            cake_size=cake_size,
            epsilon=epsilon,
            start=to_decimal(0),
            end=cake_size,
            target=alpha,
            tolerance=tolerance,
        )

        m = _binary_search_right_to_left(
            preference=preference_1,
            cake_size=cake_size,
            epsilon=epsilon,
            start=to_decimal(0),
            end=r,
            target=alpha,
            tolerance=tolerance,
        )

        l = _find_balanced_cut_for_adjacent(
            preference=preference_i,
            cake_size=cake_size,
            epsilon=epsilon,
            left=to_decimal(0),
            right=m,
            tolerance=tolerance,
        )

        return [l, m, r]
    # if k, k' = (1, 2)
    #     0         1         2             3
    # [0, l] | [l, m] | [m, r] | [r, cake_size]
    #     1       (3          3)           2
    elif k == 1 and k_prime == 2:
        l = _binary_search_left_to_right(
            preference=preference_1,
            cake_size=cake_size,
            epsilon=epsilon,
            start=to_decimal(0),
            end=cake_size,
```

```
            target=alpha,
            tolerance=tolerance,
        )

        r = _binary_search_right_to_left(
            preference=preference_1,
            cake_size=cake_size,
            epsilon=epsilon,
            start=l,
            end=cake_size,
            target=alpha,
            tolerance=tolerance,
        )

        m = _find_balanced_cut_for_adjacent(
            preference=preference_i,
            cake_size=cake_size,
            epsilon=epsilon,
            left=l,
            right=r,
            tolerance=tolerance,
        )

        return [l, m, r]
    # if k, k' = (2, 3)
    #     0         1         2             3
    # [0, l] | [l, m] | [m, r] | [r, cake_size]
    #     1       2          (3           3)
    elif k == 2 and k_prime == 3:
        l = _binary_search_left_to_right(
            preference=preference_1,
            cake_size=cake_size,
            epsilon=epsilon,
            start=to_decimal(0),
            end=to_decimal(cake_size),
            target=alpha,
            tolerance=tolerance,
        )

        m = _binary_search_left_to_right(
            preference=preference_i,
            cake_size=cake_size,
```

```
            epsilon=epsilon,
            start=l,
            end=to_decimal(cake_size),
            target=alpha,
            tolerance=tolerance,
        )

        r = _find_balanced_cut_for_adjacent(
            preference=preference_i,
            cake_size=cake_size,
            epsilon=epsilon,
            left=m,
            right=to_decimal(cake_size),
            tolerance=tolerance,
        )

        return [l, m, r]
```

Listing B.11: Function to Handle Adjacent k and k' On Condition B

```
def _handle_one_between(
    k: int,
    k_prime: int,
    alpha: Decimal,
    preference_1: List[Segment],
    preference_i: List[Segment],
    epsilon: Decimal,
    cake_size: Decimal,
    tolerance: Decimal = to_decimal(1e-10),
) -> List[Decimal]:
    # if k, k' = (0, 2)
    #    0        1        2            3
    # [0, l] | [l, m] | [m, r] | [r, cake_size]
    #                                  1
    # give l, find m(l), where v_1([l, m(l)]) = alpha (second piece)
    # keep move l, making v_i([0, l]) = v_i([m(l), r])
    if k == 0 and k_prime == 2:
        r = _binary_search_right_to_left(
            preference=preference_1,
            cake_size=cake_size,
            epsilon=epsilon,
            start=to_decimal(0),
            end=cake_size,
```

```
            target=alpha,
            tolerance=tolerance,
        )
        l_start = to_decimal(0)
        l_end = r
        l, m = _binary_search_case_0_2(
            preference_1=preference_1,
            preference_i=preference_i,
            epsilon=epsilon,
            l_start=l_start,
            l_end=l_end,
            alpha=alpha,
            cake_size=cake_size,
            tolerance=tolerance,
        )
        return [l, m, r]
    # if k, k' = (1, 3)
    #    0        1        2            3
    # [0, l] | [l, m] | [m, r] | [r, cake_size]
    #    1
    # give r, find m(r), where v_1([m(r), r]) = alpha (third piece)
       = v_1([(0, l)])
    # keep move r, making v_i([l, m(r)]) = v_i([r, cake_size])
    elif k == 1 and k_prime == 3:
        l = _binary_search_left_to_right(
            preference=preference_1,
            cake_size=cake_size,
            epsilon=epsilon,
            start=to_decimal(0),
            end=cake_size,
            target=alpha,
            tolerance=tolerance,
        )

        r_start = to_decimal(l)
        r_end = to_decimal(cake_size)
        m, r = _binary_search_case_1_3(
            preference_1=preference_1,
            preference_i=preference_i,
            epsilon=epsilon,
            r_start=r_start,
            r_end=r_end,
```

```
            cake_size=to_decimal(cake_size),
            alpha=alpha,
            tolerance=tolerance,
        )
        return [l, m, r]
    else:
        raise ValueError(f"_handle_one_between: Invalid k and k', {k
            =}, {k_prime=}")
```

Listing B.12: Function to Handle There is One Poice between k and $k^{'}$ On Condition B

Appendix C

Discussion

C.1 Code Coverage Report

Coverage report: 71%

Files | Functions | Classes

coverage.py v7.6.1, created at 2024-08-22 19:07 +0100

File ▲	*statements*	*missing*	*excluded*	*coverage*
__init__.py	0	0	0	100%
algorithms/__init__.py	0	0	0	100%
algorithms/alex_aviad_condition/__init__.py	0	0	0	100%
algorithms/alex_aviad_condition/condition_a_test.py	16	0	0	100%
algorithms/alex_aviad_condition/condition_a.py	112	17	0	85%
algorithms/alex_aviad_condition/condition_b_helper.py	23	5	0	78%
algorithms/alex_aviad_condition/condition_b_test.py	75	0	0	100%
algorithms/alex_aviad_condition/condition_b.py	423	354	0	16%
algorithms/alex_aviad_hepler_test.py	170	0	0	100%
algorithms/alex_aviad_hepler.py	100	3	0	97%
algorithms/alex_aviad_result_helper.py	29	29	0	0%
algorithms/alex_aviad_test.py	83	30	0	64%
algorithms/alex_aviad.py	82	57	0	30%
algorithms/algorithm_test_utils_test.py	11	0	0	100%
algorithms/algorithm_test_utils.py	88	4	0	95%
algorithms/algorithm_types.py	27	0	0	100%
algorithms/cut_and_choose_test.py	57	0	0	100%
algorithms/cut_and_choose.py	28	1	0	96%
base_types.py	83	9	0	89%
cut.py	36	16	0	56%
main.py	34	34	0	0%
server/__init__.py	0	0	0	100%
type_helper_test.py	67	0	0	100%
type_helper.py	44	5	0	89%
utils.py	23	12	0	48%
valuation_test.py	201	0	0	100%
valuation.py	137	5	0	96%
values_test.py	8	0	0	100%
values.py	86	20	0	77%
Total	**2043**	**601**	**0**	**71%**

C.2 Interactive Course

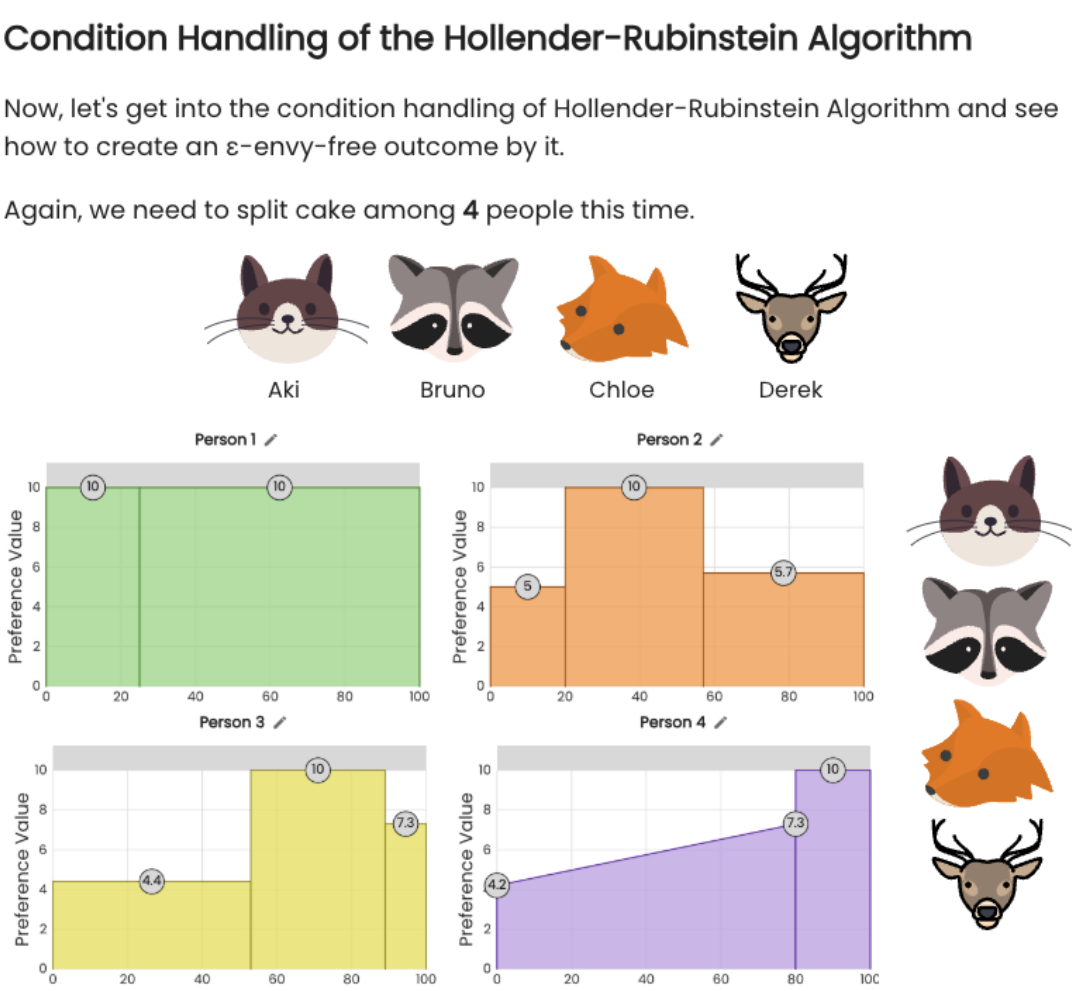


Figure C.2: Condition Handling Course Part 1

The cake is now split using the Hollender-Rubinstein Algorithm, but equipartition no longer works... We're going to go into the main loop, and what we're going to show is what the algorithm goes through with the settings ε=1e-5, and tolerance=1e-5. The algorithm goes through the process...

- Missed conditions at $\alpha = 0.625000000000025$, set $\bar{\alpha} = \alpha$
- Missed conditions at $\alpha = 0.437500000000037$, set $\bar{\alpha} = \alpha$
- Missed conditions at $\alpha = 0.343750000000043$, set $\bar{\alpha} = \alpha$
- Missed conditions at $\alpha = 0.296875000000046$, set $\bar{\alpha} = \alpha$
- Missed conditions at $\alpha = 0.273437500000048$, set $\bar{\alpha} = \alpha$
- Meet B at $\alpha = 0.261718750000048$, set $\underline{\alpha} = \alpha$
- Missed conditions at $\alpha = 0.267578125000048$, set $\bar{\alpha} = \alpha$
- Missed conditions at $\alpha = 0.264648437500048$, set $\bar{\alpha} = \alpha$
- Meet B at $\alpha = 0.263183593750048$, set $\underline{\alpha} = \alpha$
- Missed conditions at $\alpha = 0.263916015625048$, set $\bar{\alpha} = \alpha$
- Meet B at $\alpha = 0.263549804687548$, set $\underline{\alpha} = \alpha$
- Meet B at $\alpha = 0.263732910156298$, set $\underline{\alpha} = \alpha$
- Meet B at $\alpha = 0.263824462890673$, set $\underline{\alpha} = \alpha$
- Try to find a ε-EF allocation at $\underline{\alpha} = 0.263824462890673$

Finally, we get what we want, a ε-EF allocation! Even though we don't meet Condition once...

Figure C.3: Condition Handling Course Part 2

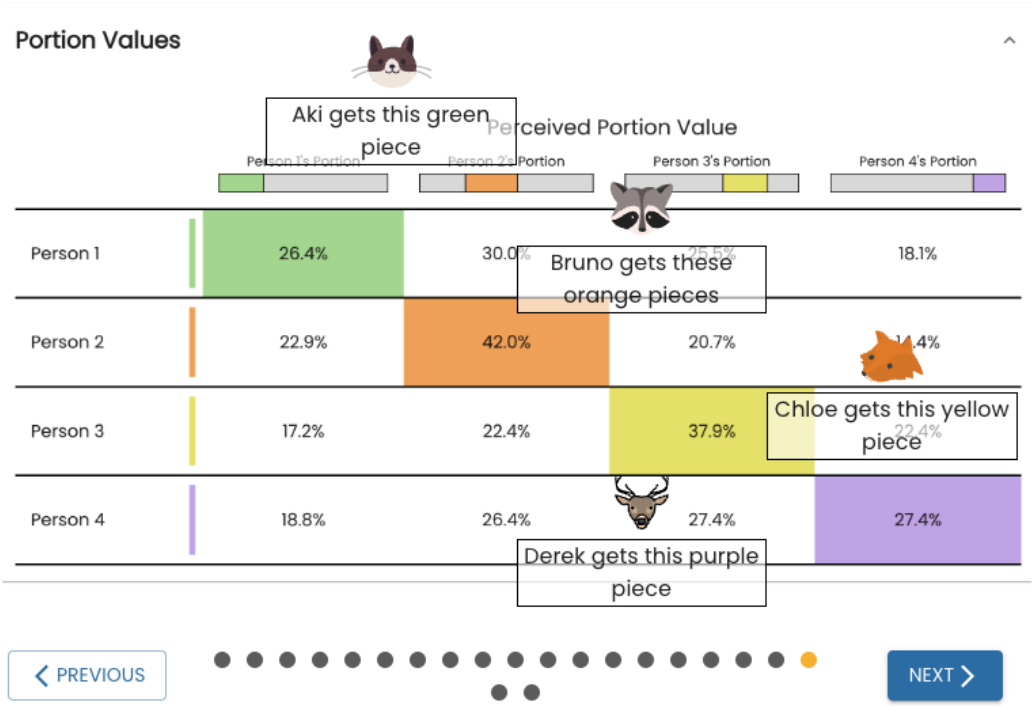


Figure C.4: Condition Handling Course Part 3