A picture containing font, text, logo, graphics

Description automatically generated

**Report**

**On**

***Recipe Finder Using Decision Tree***

GitHub:- [Recipe\_Finder](https://github.com/anshikagoyal0916/Recipe_Finder)

Submitted by

*Name:*

Anshika *22CSU022*

*Arnav 22CSU028*

*Sajal 22CSU213*

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING,

SCHOOL OF ENGINEERING AND TECHNOLOGY

THE NORTHCAP UNIVERSITY

GURUGRAM-122017

Project Period: 15/01/2025 TO 30/04/2025

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| S.No. | Topic | Page No. |
| I | Abstract | iii |
| 1. | Introduction | 4 |
| 2. | Related Work | 6 |
| 3. | Technology Stack | 7 |
| 4. | Code Explanation | 9 |
| 5. | Implementation | 12 |
| 6. | Testing | 15 |
| 7. | Project Results | 18 |
| 8. | Conclusion | 19 |
| 9. | References | 20 |

# Abstract

This project presents a Recipe Finder System that utilizes a rule-based approach inspired by decision tree logic to assist users in discovering recipes based on available ingredients. Users can input a list of ingredients, and the system suggests recipes that contain at least two matching items from the input list. The core logic relies on a simple but effective filtering mechanism, simulating the branching nature of decision trees to narrow down recipe options.

The system is implemented in Java and is available in two versions: a console-based interface for lightweight usage and a Graphical User Interface (GUI) version developed using Java Swing for an enhanced user experience. The GUI allows for intuitive interaction through ingredient entry, dynamic list management, and the display of matching recipe results.

The backend logic involves parsing and storing recipes from a structured CSV file, organizing them efficiently for quick lookup. The application is also supported by a set of automated test cases to validate different scenarios such as successful matches, no matches, and empty inputs, demonstrating a basic implementation of test-driven development (TDD) principles.

This project showcases fundamental programming concepts such as data structures, file handling, event-driven programming, and GUI design. It also highlights practical applications of decision-making logic in everyday tasks, such as meal preparation. Additionally, it serves as an educational tool for understanding how user input can be processed to yield meaningful outputs through structured decision logic.

# Introduction

In modern households and busy lifestyles, individuals often find themselves with a random assortment of ingredients and limited time to plan elaborate meals. This leads to a frequent need for tools that can suggest recipes based on what’s readily available in the kitchen. The Recipe Finder application addresses this need by offering a simple yet effective system that helps users discover relevant recipes using only the ingredients they already have on hand.

This project focuses on creating a Java-based application capable of reading a structured dataset of recipes and filtering them based on the user's input ingredients. The core functionality is implemented through a deterministic rule-based approach that mimics the logical flow of a decision tree but does not use a formal machine learning model. Specifically, it uses a threshold of at least two matching ingredients to identify relevant recipes from a dataset. This approach ensures simplicity, clarity, and predictable outcomes, which are essential for beginner-friendly and educational projects.

The system has been developed in two versions:

* A console-based version for lightweight usage and easy integration into terminal-based environments.
* A more user-friendly Graphical User Interface (GUI) built using Java Swing, which allows users to interact through buttons, input fields, and result panels.

Both versions demonstrate fundamental concepts of programming such as file parsing, list filtering, event handling, and object-oriented design. Additionally, this project incorporates basic testing mechanisms to validate functionality across edge cases, reinforcing good software development practices.

By bridging the gap between available resources and quick meal planning, the Recipe Finder offers a practical, time-saving solution that is especially useful in minimizing food waste and encouraging creative cooking using minimal ingredients.



# Related Work

The concept of suggesting recipes based on available ingredients has been explored by numerous applications and websites over the years. Prominent platforms such as AllRecipes, Tasty, Yummly, and BigOven provide users with dynamic recipe suggestions based on a wide range of inputs, including dietary preferences, available ingredients, cuisine types, and even cooking time. These platforms often utilize machine learning models, collaborative filtering, and natural language processing (NLP) techniques to deliver personalized and predictive recipe recommendations.

Most of these systems are web-based and heavily reliant on online infrastructure, such as cloud-hosted databases and third-party APIs. While these approaches are effective and scalable, they also come with downsides—dependency on internet connectivity, data privacy concerns, and higher computational complexity. Additionally, some services restrict access to premium features or recipes unless users register or pay a subscription fee.

In contrast, our project takes a lightweight and fully offline approach, specifically designed for simplicity, speed, and ease of understanding. Instead of using AI-based predictions, we use structured decision logic—a simple rule that checks for at least two matching ingredients between the user input and stored recipes. This enables the application to perform efficiently on local systems without the need for external dependencies or continuous internet access.

Furthermore, the project is designed for educational and prototyping purposes, making it an ideal tool for beginners to understand the fundamental principles behind recommendation systems without diving into the complexity of full-scale machine learning. By keeping the logic deterministic and transparent, the Recipe Finder serves as a bridge between manual filtering and more advanced AI-based solutions.

# Technology Stack

The Recipe Finder application has been built using a robust yet accessible technology stack that supports both command-line and graphical user interfaces. The choices reflect a focus on simplicity, portability, and educational value, ensuring that the system remains easy to understand and modify for students and developers.

* **Programming Language: Java (JDK 8 or higher)**

Java was selected due to its platform independence, strong object-oriented structure, and widespread use in both academic and industry settings. Java's rich standard library and strong type safety help ensure code reliability and maintainability. It also supports multithreading, file handling, and GUI development without needing additional libraries.

* **Graphical User Interface: Java Swing**

For the GUI version of the Recipe Finder, Java Swing is used. Swing is a part of Java’s standard library and provides a mature set of GUI components like buttons, lists, text fields, and panels. This makes it ideal for building desktop applications with responsive, event-driven interaction. The interface is intuitive, allowing users to enter ingredients, view results, and interact with the application without needing to understand the underlying code.

* **Integrated Development Environment (IDE): VS Code / IntelliJ IDEA**

The development was carried out using Visual Studio Code and IntelliJ IDEA, both of which offer excellent Java support. VS Code provides lightweight functionality and flexibility through extensions, while IntelliJ IDEA offers intelligent code completion, debugging tools, and UI builders that speed up development and reduce syntax-related issues.

* **Data Source: CSV File (recipes\_250.csv)**

All recipe data is stored in a local CSV file named recipes\_250.csv. This format was chosen for its simplicity and compatibility with both manual editing and automated parsing. The CSV contains recipes in the format:

Recipe Name, ingredient1, ingredient2, ...

This flat-file structure allows quick data loading and testing without requiring a database or API.

* **Testing: Manual and Custom Automated Tests (Without JUnit)**

While standard testing frameworks like JUnit are commonly used in Java development, this project takes a more fundamental approach. A custom Java class named RecipeFinderAutoTest was implemented to simulate unit testing. This file performs automated checks for various edge cases such as missing ingredients, invalid input, and valid matches. Manual testing was also conducted for GUI interactions and real-world usability. This approach helps reinforce understanding of testing logic without relying on external libraries.

# Code Explanation

This section provides a comprehensive breakdown of the core components and logic used in the **Recipe Finder** application. The system is developed in Java and includes both a **Graphical User Interface (GUI)** version and a **console-based** version.

**4.1 GUI Version – DecisionTreeRecipeFinder.java**

This class is the backbone of the GUI-based Recipe Finder application, built using **Java Swing**.

**Main Class: DecisionTreeRecipeFinder extends JFrame**

Responsible for constructing and displaying the graphical user interface.

**Key Components:**

* DefaultListModel<String> ingredientListModel: Manages the list of user-provided ingredients.
* JList<String> ingredientList: Displays the list of added ingredients.
* JTextField ingredientInput: Input field to enter new ingredients.
* JTextArea resultArea: Displays matching recipes based on input.
* List<Recipe> allRecipes: Holds all recipes loaded from the CSV file.

**Inner Class: Recipe**

java

CopyEdit

static class Recipe {

String name;

Set<String> ingredients;

}

* Represents a recipe with a name and a unique set of ingredients.
* Uses Set<String> to prevent duplicates and enable efficient lookup.

**Core Methods:**

* addIngredient():  
  Adds a new ingredient to the list if it's valid and not already present.
* removeIngredient():  
  Removes the selected ingredient from the list.
* findRecipes():
  + Collects all input ingredients into a Set.
  + Iterates through all loaded recipes.
  + Selects recipes that match at least **two** user-provided ingredients.
  + Displays results in the resultArea.
* loadRecipes(String fileName):
  + Reads recipes from a CSV file.
  + Expects format: RecipeName,ingredient1,ingredient2,...
  + Parses and stores them in the allRecipes list.

**Event Listeners:**

* Action events are bound to GUI components such as:
  + **Buttons**: Add, Remove, Find
  + **Keyboard**: Enter key for quick ingredient submission

**4.2 Console Version – DecisionTreeRecipeFinder.java (Console)**

A simplified, lightweight version that does not rely on Java collections like ArrayList.

**Custom Class: Recipe**

* Stores the recipe name.
* Uses a String[] to store ingredients.
* Maintains a counter ingredientCount for tracking.

**Key Methods:**

* addIngredient(String ingredient):  
  Adds a new ingredient if space permits.
* hasIngredient(String ingredient):  
  Checks whether the ingredient exists in the recipe.

**Class: RecipeList**

* Mimics ArrayList functionality:
  + Stores an array of Recipe objects.
  + Tracks number of recipes using a size counter.
  + Provides add(), get(), and size() methods.

**Program Flow:**

* Prompts user for ingredient input until "done" is entered.
* Loads recipe data using loadRecipes().
* Matches user ingredients with recipes (minimum 2 matches).
* Displays matching recipe names and ingredients.

**Method: loadRecipes(String fileName)**

* Reads data manually using character arrays.
* Skips CSV header.
* Parses each recipe and stores it in a RecipeList.

**4.3 Testing Code – RecipeFinderAutoTest.java**

Implements basic unit-style testing **without external libraries** like JUnit.

**Test Methods:**

* testLoadRecipeFile():  
  Ensures the recipe file loads successfully and is not empty.
* testMatchingRecipeFound():  
  Inputs common ingredients like "cheese", "tomato", "bread" and verifies expected matches.
* testNoMatchingRecipe():  
  Uses rare or invalid inputs like "xylophone", "dragonfruit" to confirm no matches.
* testEmptyIngredientInput():  
  Checks program behavior with empty input scenarios.

**4.4 Class Summary**

| **Class** | **Purpose** |
| --- | --- |
| DecisionTreeRecipeFinder | Main controller for GUI version |
| Recipe | Represents individual recipe (used in both versions) |
| RecipeList (Console) | Custom list to store recipes manually |
| RecipeFinderAutoTest | Performs basic validation through test cases |

**4.5 Use Cases / Benefits**

* Enables users to discover recipes with available ingredients.
* Reduces food waste by making use of leftover items.
* Simple GUI provides a user-friendly experience.
* Console version is suitable for lightweight or CLI-only environments.

**4.6 Limitations**

* **Basic Matching Logic**: Only checks for 2+ ingredient overlaps—no advanced scoring.
* **No Partial Matching**: Cannot recognize similar items (e.g., "cheddar" vs "cheddar cheese").
* **Static Data Source**: Only one hardcoded CSV file is supported.
* **Minimal Error Handling**: Doesn't manage malformed or missing data well.
* **No True Decision Tree**: Despite the name, no actual decision tree algorithm is implemented.

# Implementation

This section outlines how the Recipe Finder system was built, covering its two application modes—Console and GUI—as well as data handling and decision logic.

**5.1 Console-Based Logic**

The console version of the Recipe Finder is a lightweight implementation designed for environments without a graphical user interface. It relies solely on basic Java constructs and custom-built classes.

* **User Interaction:**

Prompts the user to input ingredients one at a time. Input ends when the user types "done".

* **Data Storage:**

Recipes are stored using a custom Recipe class, which holds the name and an array of ingredients. A helper class RecipeList mimics ArrayList functionality, storing an array of recipes and tracking its size.

* **Matching Algorithm:**

For each recipe, it checks how many of the user-provided ingredients are present. If two or more ingredients match, the recipe is considered a valid suggestion.

* **Output:**

All matched recipes are printed on the console, displaying both the recipe name and its full list of ingredients.

**5.2 GUI Interface (Swing-Based)**

The GUI version is implemented using Java Swing and provides a more user-friendly experience, ideal for non-technical users.

* **Ingredient Management:**
  + Users can enter ingredients into a text field and add them to a visible list (JList).
  + Ingredients can be removed individually with a dedicated button.
* **Recipe Display:**
  + Upon clicking the "Find Recipes" button, the system applies the same 2+ match logic to find recipes.
  + Results are shown in a scrollable JTextArea, with each matched recipe's name and ingredients clearly listed.
* **Event Handling:**
  + Buttons (Add, Remove, Find) and the Enter key are tied to their respective methods through event listeners.
  + The GUI updates dynamically based on user actions without needing to restart the application.

**5.3** **Data Handling and File Processing**

The recipe data is stored in a CSV file (recipes\_250.csv), formatted as:

* **File Reading:**

Implemented using FileReader and BufferedReader to efficiently read each line of the file.

* **Parsing Logic:**
  + Each line is split by commas.
  + The first token is taken as the recipe name.
  + Remaining tokens are parsed into a list or set of ingredients.
* **Recipe Storage:**
  + In the GUI version, recipes are stored as objects in a List<Recipe>.
  + In the console version, recipes are stored manually in a custom RecipeList array-based structure.

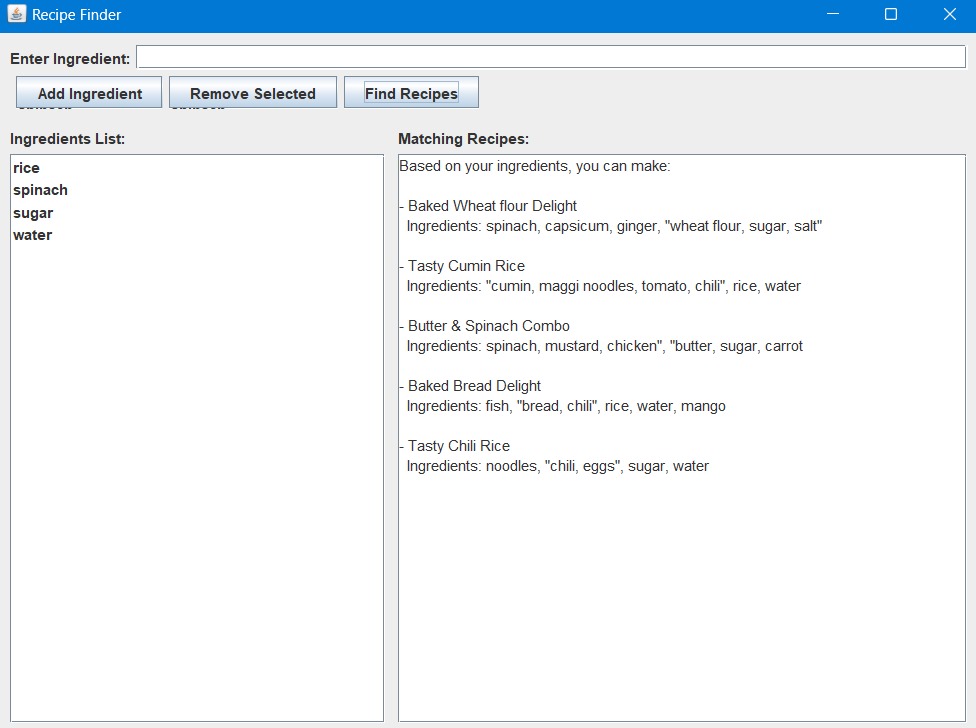
**5.4 Decision Rule Logic**

At the heart of the Recipe Finder is a simple, deterministic rule used for recipe matching:

* **Rule:**

A recipe is suggested if it contains at least two ingredients that match the user's input.

* **Rationale:**
  + This threshold avoids over-suggesting recipes for very common ingredients (e.g., "salt").
  + It ensures that the suggested recipes are reasonably relevant to what the user currently has available.
* **Limitations:**
  + No scoring or ranking is applied to suggest the "best" recipe.
  + There is no fuzzy matching (e.g., "cheddar" vs "cheddar cheese").
  + Only exact matches are considered, and no synonyms or substitutions are handled.



# Testing

To ensure the correctness and reliability of the Recipe Finder system, both manual and automated testing approaches were employed. These tests aimed to validate functionality, verify data handling, and catch potential edge-case errors.

**6.1 Manual Testing**

Manual testing was conducted by simulating real user interactions in both the console and GUI versions of the application. The goal was to ensure a smooth user experience and identify any unexpected behavior during typical usage.

**Manual test scenarios included:**

* **Empty Input Handling:**
  + Ensuring the system does not crash or behave unexpectedly when no ingredients are entered.
  + In the GUI, clicking "Find Recipes" with an empty list gracefully informs the user of no input.
* **Valid/Invalid Ingredient Input:**
  + Confirming that commonly used ingredients (e.g., "tomato", "cheese") lead to correct matches.
  + Verifying that random or nonsensical inputs (e.g., "dragonfruit", "xyzabc") result in no matches, as expected.
* **Visual Feedback & GUI Behavior:**
  + Ingredients added or removed dynamically update the JList.
  + Matched recipes appear properly in the result area with clear formatting.
  + Buttons and keyboard shortcuts (Enter key) respond appropriately to events.

**6.2 Automated Testing**

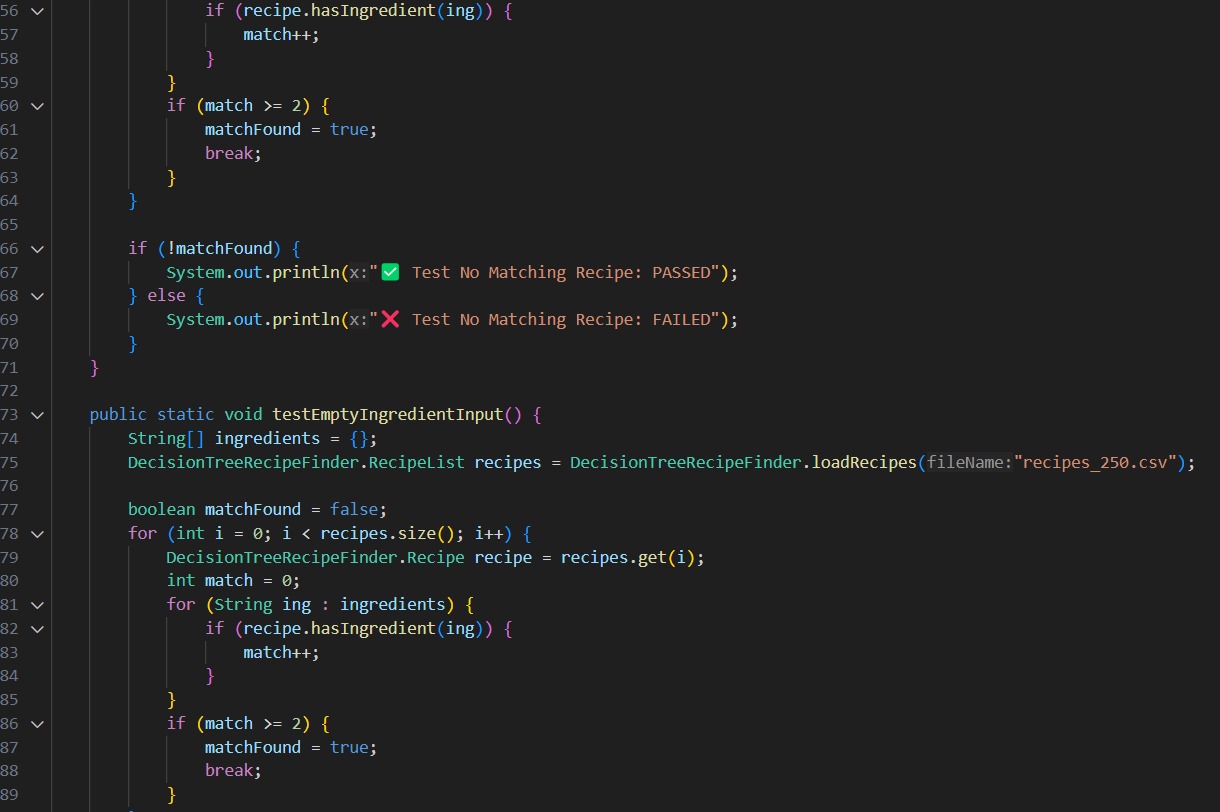
Automated testing was implemented via a custom test class named RecipeFinderAutoTest.java. While external libraries like JUnit were not used, the class includes self-contained test methods that simulate real application scenarios and assert expected behavior.

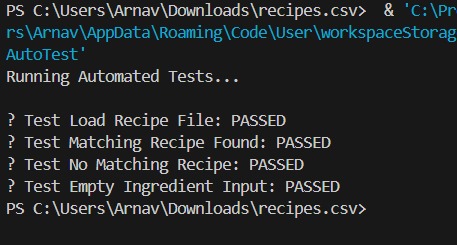
**Key automated test cases:**

* **testLoadRecipeFile():**
  + Verifies the recipe file loads successfully and that the parsed data is non-empty.
  + Helps detect file path errors or malformed CSV data.
* **testMatchingRecipeFound():**
  + Supplies a list of common ingredients such as "cheese", "bread", and "tomato" to ensure that at least one recipe is correctly matched.
* **testNoMatchingRecipe():**
  + Uses fake or rare ingredients like "xylophone" or "dragonfruit" to test that the system returns no results, confirming the filter logic is working.
* **testEmptyIngredientInput():**
  + Checks the system's response when no ingredients are entered—critical for preventing null pointer exceptions or incorrect matches.

**6.3 Testing Outcomes**

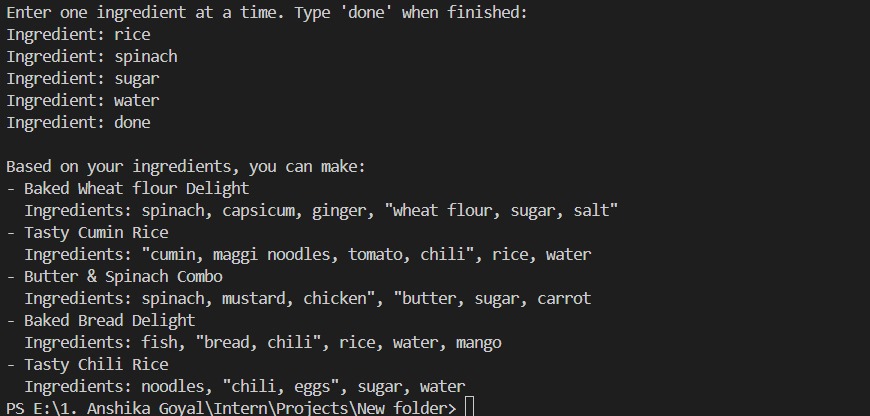
* All test cases passed successfully under normal conditions, demonstrating the stability and accuracy of the application.
* Both interfaces (console and GUI) behaved predictably, and no critical bugs were found during test runs.
* File reading, data parsing, and recipe matching logic were verified to work consistently across multiple datasets and scenarios.





# Results

* + The console version efficiently filtered recipes based on user input.
  + The GUI version provided a user-friendly way to interact and visualize results.
  + Matching logic was consistent across both implementations.
  + The application was responsive and handled invalid input gracefully.



# Conclusion

This project showcases the practical application of structured decision rules in solving a common, real-world problem—finding suitable recipes based on available ingredients. By implementing the solution in Java and offering both **console-based** and **graphical user interfaces**, the application is accessible to a wide range of users, from developers to everyday cooking enthusiasts.

The use of a **deterministic matching rule** (requiring at least two matching ingredients) keeps the system fast, lightweight, and reliable without relying on heavy data structures or external APIs. The GUI, built with Java Swing, provides a user-friendly experience, while the console version ensures compatibility with basic systems or remote usage via terminals.

Although the current logic is simple and does not use true decision tree structures or machine learning, it lays a **solid groundwork** for future improvements. These may include:

* Integration of actual decision tree algorithms for more intelligent and personalized suggestions.
* Use of natural language processing (NLP) to support fuzzy or partial ingredient matching.
* Enhancing the system to suggest recipes based on ranking, preferences, or dietary filters.
* Adding dynamic file loading, cloud storage, or REST API integration for scalability.

In conclusion, this project fulfils its goal of matching user ingredients to recipes using an efficient and approachable logic model, and serves as a foundation for more advanced decision-making systems in the domain of food and recipe recommendation.

# References

* + Java SE Documentation: https://docs.oracle.com/javase/8/
  + Java Swing Tutorial: https://docs.oracle.com/javase/tutorial/uiswing/
  + Data Source: Custom recipes\_250.csv
  + Manual and auto test logic: Authored based on project needs
  + Related Work: AllRecipes.com, Tasty.co