Matthew Morris
Professor Kontothanassis
DS210
26 March 2025

DS210 HW 6 Journal

Sources: (see last page)

Question 1:

Terminal Output:

```
name:
*input* Halle Vaughn
recipe:
*input* recipe0
halle vaughn does NOT like the recipe 'recipe0'.
```

```
use std::collections::{HashMap, HashSet};
use std::fs;
use std::io;
```

Importing hashmap, hashset, because the datasets are essential to make the .txt files manipulatable.
Importing filesystem (fs), in order to pull data from the .txt files
Importing io, which allows for input in the terminal to ask for recipes and names

```
fn strings_from_data(
  name: &str,
  recipe: &str,
  people_categories_data: &str,
  categories_ingredients_data: &str,
  recipes_data: &str,
) -> String {
  let mut output = String::new();

```

Intializing the variables required in the query–specifically name, and recipe which are set as part of the io query, as well as the recipes, category ingredients, and people categories which are used throughout as well as refereanble immutable strings. I also want this to output into “output” as a new string, which I can print out.

```
{
 let mut people_categories = HashMap::new();
 for line in people_categories_data.lines() {
    if let Some((person, categories)) = line.split_once(": ") {
        let category_set: HashSet<_> = categories
            .split(',')
            .map(str::trim)
            .map(|s| s.to_lowercase())
            .collect();
        people_categories.insert(person.to_lowercase(), category_set);
    }
 }
```

Created a hashmap for the people categories in order to track what each category people like. This reads the people_categories_data, so that later, I am able to map individual ingredients to categories, thus connecting people to their favorite ingredients. It does this through creating a hashset for each category, as well as lowercase mapping each category (which is for consistency). From there, it inserts the lowercase person (which is treated as such for consistency with the hashmap so that the entered in name is the same as in the .txt file).

```
let mut category_ingredients = HashMap::new();
 let mut ingredient_to_category = HashMap::new();
 for line in categories_ingredients_data.lines() {
    if let Some((category, ingredients)) = line.split_once(": ") {
        let ingredient_set: HashSet<_> = ingredients
            .split(',')
            .map(str::trim)
            .map(|s| s.to_lowercase())
            .collect();
        for ingredient in &ingredient_set {
            ingredient_to_category.insert(ingredient.clone(), category.to_lowercase());
        }
        category_ingredients.insert(category.to_lowercase(), ingredient_set);
    }
 }
```

This function allows us to map ingredients to categories, reading the format so that we can transitively associate the ingredients with the people. It iterates through all lines in the data, splitting categories based on the colon, which is used as a delimiter in the .txt file, because of the key value format. and then maps/trims each individual ingredient. From there, it clones all ingredients and turns them lowercase (also for consistency).

This is also inserted into the category_ingredients hashmap.

```
let mut recipe_ingredients = HashMap::new();
for line in recipes_data.lines() {
    if let Some((recipe_name, ingredients)) = line.split_once(": ") {
        let ingredient_list: Vec<_> = ingredients
            .split(',')
            .map(str::trim)
            .map(|s| s.to_lowercase())
            .collect();
        recipe_ingredients.insert(recipe_name.to_lowercase(), ingredient_list);
    }
}
```

This function allows us to read the recipes.txt file, doing something similar to the previous two functions. From there, it generates an ingredient list based on each recipe. We get a hashmap in the form <string, vec<string>> which is useful because it allows us to construct a hashmap of the recipes before we perform any actions upon the hashmaps.

```
fn main() {
    let mut name = String::new();
    println!("name:");
    io::stdin()
        .read_line(&mut name)
        .expect("Failed to read name");
    let name = name.trim().to_lowercase();
```

This is code for the query of the name, turning the name into lowercase for consistency. It initializes a new mutable string "name," and has a result "failed to read name" if the name doesn't work, so that the rust program does not panic.

```
    let mut recipe = String::new();
    println!("recipe:");
    io::stdin()
```

```
        .read_line(&mut recipe)
        .expect("Failed to read recipe");
    let recipe = recipe.trim().to_lowercase();
```

This is virtually the same code, except for recipes.

```
    let people_categories_path = "../hw_06_new/people_categories.txt";
    let categories_ingredients_path = "../hw_06_new/categories_ingredients.txt";
    let recipes_path = "../hw_06_new/recipes.txt";

```

This is a file path for the text files based on my file system. This works exclusively with my folder setup. I noticed that you need to go back only 1 folder if you are running the file, but 2 if you are using cargo test. This goes back to the ds210hw6 folder, and then navigates from there into where the .txt files live.

```
let people_categories_data = fs::read_to_string(people_categories_path).expect("Failed to read people categories file");
    let categories_ingredients_data = fs::read_to_string(categories_ingredients_path).expect("Failed to read categories ingredients file");
    let recipes_data = fs::read_to_string(recipes_path).expect("Failed to read recipes file");
```

This is the code I used to read the .txt filed and save them as variables and to access them. It also has an “expect” which I used as a troubleshooting method to see if the files were mapped properly.

```
    strings_from_data(
        &name,
        &recipe,
        &people_categories_data,
        &categories_ingredients_data,
        &recipes_data,
    );
}
```

These are the strings that I set, which are not mutable because these are the fixed in place variables taken from a combination of the data I took from the .txt files and the queries, which I don’t want to change.

```
    if !people_categories.contains_key(name) {
        println!("Person '{}' not found.", name);
        return;
    }
```

This is checking if the person exists or not.

```
    if !recipe_ingredients.contains_key(recipe) {
        println!("Recipe '{}' not found.", recipe);
        return;
    }
```

This checks if the recipe ingredients exist

```
    let user_categories = match people_categories.get(name) {
        Some(c) => c,
        None => return,
    };
```

This obtains the person's preferred categories using a match function.

```
    let ingredients = match recipe_ingredients.get(recipe) {
        Some(i) => i,
        None => return,
    };
```

This obtains the ingredients in the recipe.

```
    let total = ingredients.len();
    let liked = ingredients
        .iter()
        .filter(|&ingredient| {
            if let Some(category) = ingredient_to_category.get(ingredient) {
                user_categories.contains(category)
            } else {
                false
            }
        })
        .count();
```

This function matches ingredients to their respective category.

```
    let threshold = (0.6 * total as f64).ceil() as usize;
    if liked >= threshold {
        println!("{} likes the recipe '{}'.", name, recipe);
    } else {
        println!("{} does NOT like the recipe '{}'.", name, recipe);
    }
}
```

From there, it tests whether the person likes the recipe or not.

```
let result = strings_from_data(
    &name,
    &recipe,
    &people_categories_data,
    &categories_ingredients_data,
    &recipes_data,
);
print!("{}", result);
```

This prints the result, whatever it may be.

Question 2:

Terminal Output:

```
name:
*input* popular recipes
recipe:
*input* popular recipes
Top 3 popular recipes:
recipe963 (liked by 318 people)
recipe874 (liked by 314 people)
recipe533 (liked by 313 people)
```

```
if name == "popular recipes" {
```

This function does a variety of things. Firstly, if the name input for the query is equal to “popular recipes” (which is a string) it triggers the function.

```
    let mut recipe_scores: Vec<(&String, usize)> = recipe_ingredients
```

It then creates a mutable vector of pairs (&string, usize) -> the recipe name and the # of people who like the recipe.

```
        .iter()
        .map(|(recipe_name, ingredients)| {
            let mut count = 0;
            for (_person, categories) in &people_categories {
                let liked = ingredients
                    .iter()
                    .filter(|&ingredient| {
```

From there, it checks how many people like it. It then loops through every person and their liked categories. For each recipe, it then counts how many of its ingredients fall into a category that the person likes.

```
                        if let Some(category) = ingredient_to_category.get(ingredient) {
                            categories.contains(category)
                        } else {
                            false
                        }
                    })
```

```
                .count();
            let threshold = (0.6 * ingredients.len() as f64).ceil() as usize;
            if liked >= threshold {
                count += 1;
            }
        }
        (recipe_name, count)
    })
```

I was initially going to merge all repeating ingredients (sort all the ingredients into individual categories and eliminate duplicates), but I realized that would hurt the 60% count because some ingredients would be underrepresented. From there, it uses ingredient_to_category mapping to check if a person's preferences match.

```
    .collect();

recipe_scores.sort_by(|a, b| {
    b.1.cmp(&a.1)
        .then_with(|| a.0.to_lowercase().cmp(&b.0.to_lowercase()))
});
```

It then creates a threshold at 60% of the ingredients, using that as the ceiling (ceil()). Then, it ups the count by one, which is used in the output. This also sorts by alphabetical orer.

The final result for each recipe is a tuple, which we want to output the recipe name, and the number of people who like it. From there, I implemented logic to sort the number of people who like each recipe, and if they tie, then it sorts alphabetically.

```
    println!("Top 3 popular recipes:");
    for (recipe, count) in recipe_scores.iter().take(3) {
        println!("{} (liked by {} people)", recipe, count);
    }
    return;
}
```

This part takes the top 3 recipes by score, and prints the recipe name and how many people like it.

Tests

```
#[cfg(test)]
mod tests {
  use super::*;
```

Initializing tests

```
  #[test]
  fn test_unknown_person() {
  let people_categories_data = include_str!("../../hw_06_new/people_categories.txt");
  let categories_ingredients_data = include_str!("../../hw_06_new/categories_ingredients.txt");
  let recipes_data = include_str!("../../hw_06_new/recipes.txt");
```

Creates file mapping for each of the .txt files

```
    strings_from_data(
        "unknown_person",
        "recipe0",
        people_categories_data,
        categories_ingredients_data,
        recipes_data,
    );
  }
```

The strings essential as a part of the query created in the first part of the code. I set the input as “unknown_person” and “recipe0” as my tests to see what happens and if the rust code panics based an invalid input for the person.

```
#[cfg(test)]
mod tests {
  use super::*;

  #[test]
  fn test_popular_recipes() {
    let people_categories_data = include_str!("../../hw_06_new/people_categories.txt");
    let categories_ingredients_data = include_str!("../../hw_06_new/categories_ingredients.txt");
    let recipes_data = include_str!("../../hw_06_new/recipes.txt");

    let result = strings_from_data(
      "popular recipes",
      "",
      people_categories_data,
      categories_ingredients_data,
      recipes_data,
    );

    assert!(result.contains("Top 3 popular recipes:"));
  }
```

This tests to see if the result contains the following “Top 3 popular recipes:”, which is known assuming that the function works properly and is able to pass through and calculate top 3 recipes. While less accurate than using asserteq! And directly being able to compare against the real answer, this is still helpful.

```
  #[test]
  fn test_person_likes_recipe() {
    let people_categories_data = include_str!("../../hw_06_new/people_categories.txt");
    let categories_ingredients_data = include_str!("../../hw_06_new/categories_ingredients.txt");
    let recipes_data = include_str!("../../hw_06_new/recipes.txt");
    strings_from_data(
      "person0",
      "recipe0",
      people_categories_data,
      categories_ingredients_data,
      recipes_data,
    );
```

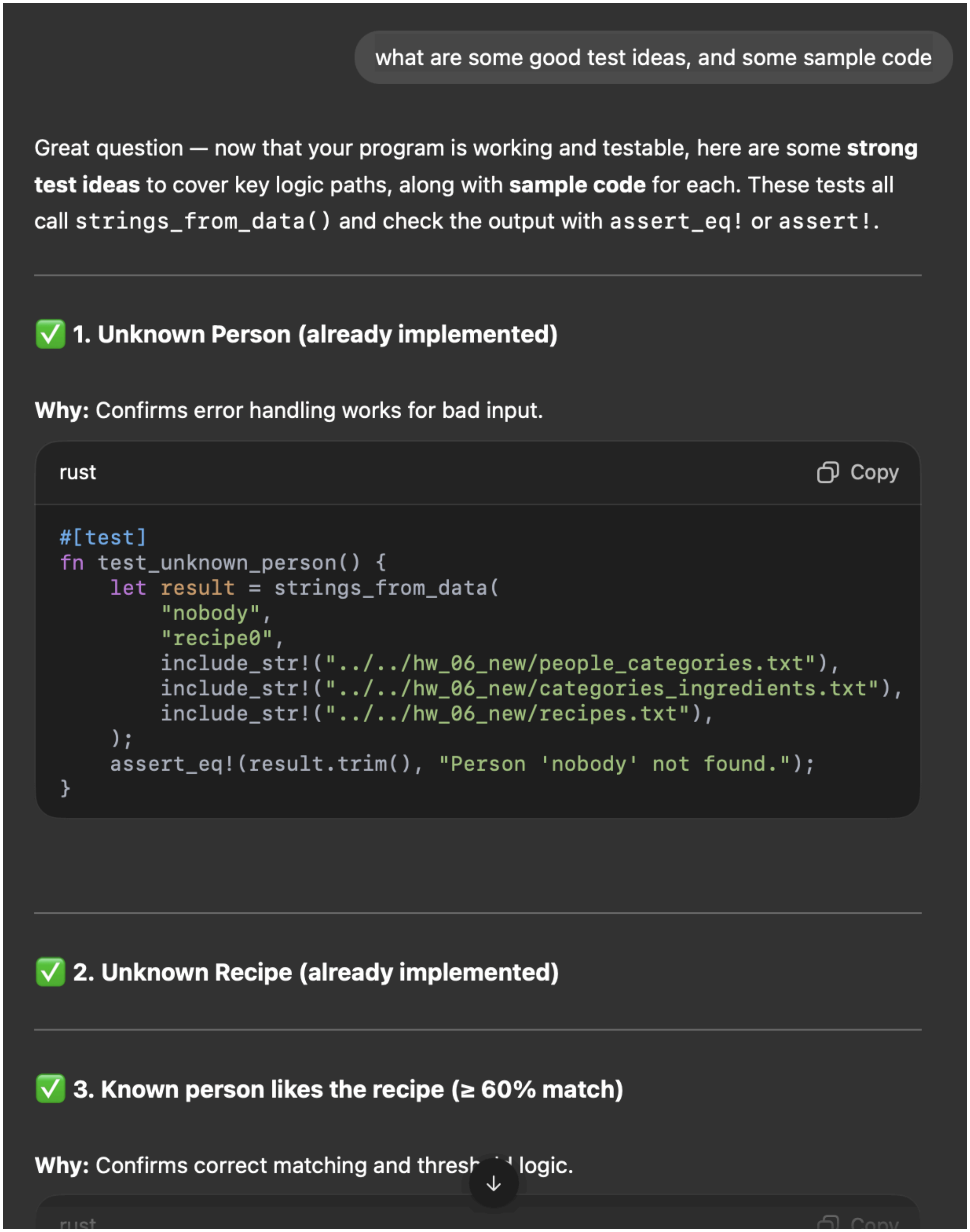
```
}
```

This function allows for a test if person0 (which is not a valid input for the function) works, and to see if the function will panic. I use include_str! To show the mapping to the txt files. Note that it is two ../.. Because we want to go back two files from /src and /question1 and then to navigate into the folder where the .txt files live.

Sources:

https://stackoverflow.com/questions/59515668/how-does-the-iteratorcollect-function-work,

ChatGPT



what are some good test ideas, and some sample code

Great question — now that your program is working and testable, here are some **strong test ideas** to cover key logic paths, along with **sample code** for each. These tests all call `strings_from_data()` and check the output with `assert_eq!` or `assert!`.

✅ 1. Unknown Person (already implemented)

Why: Confirms error handling works for bad input.

rust

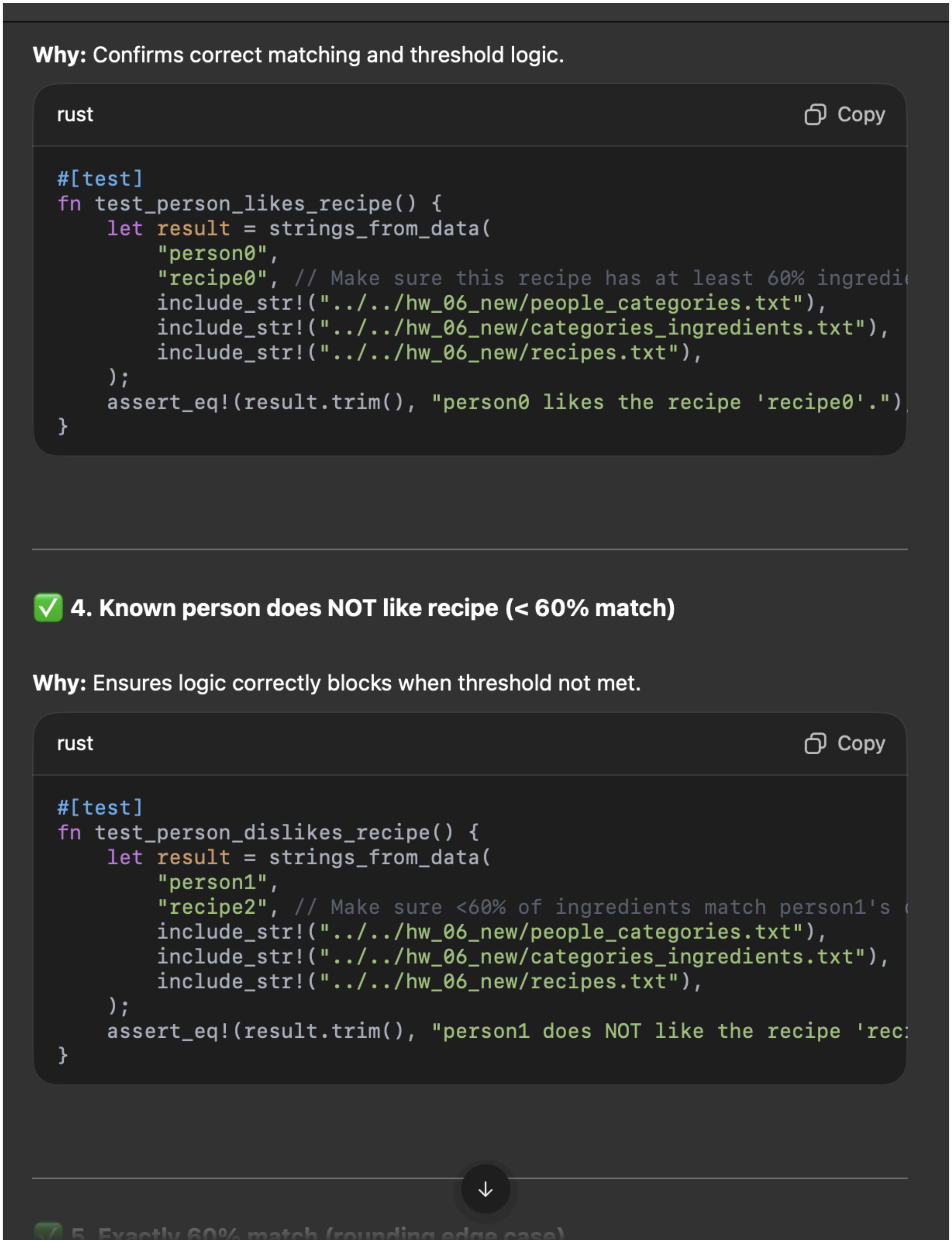
```rust
#[test]
fn test_unknown_person() {
    let result = strings_from_data(
        "nobody",
        "recipe0",
        include_str!("../../hw_06_new/people_categories.txt"),
        include_str!("../../hw_06_new/categories_ingredients.txt"),
        include_str!("../../hw_06_new/recipes.txt"),
    );
    assert_eq!(result.trim(), "Person 'nobody' not found.");
}
```

✅ 2. Unknown Recipe (already implemented)

✅ 3. Known person likes the recipe (≥ 60% match)

Why: Confirms correct matching and thresh l logic.

rust



Why: Confirms correct matching and threshold logic.

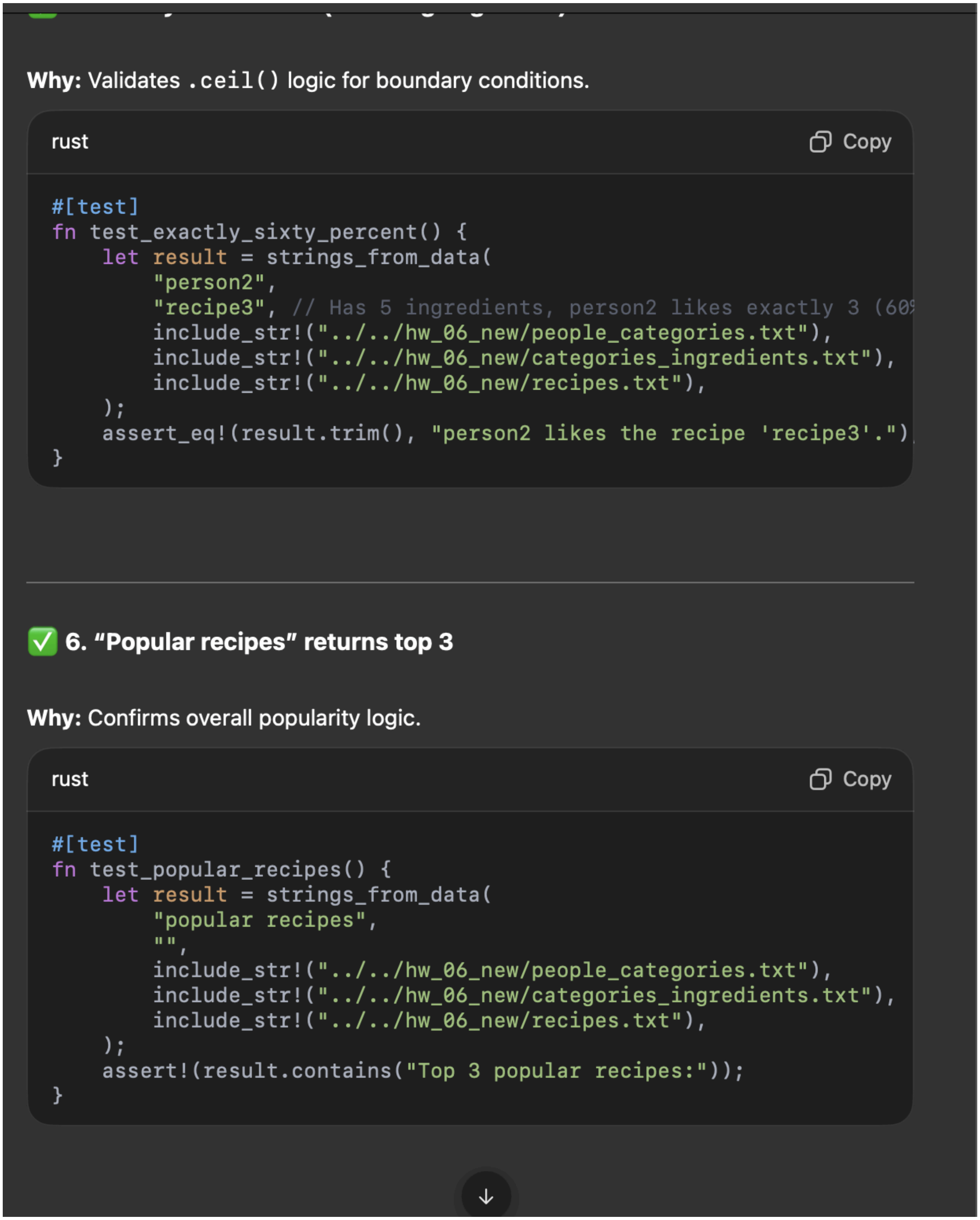
```rust
#[test]
fn test_person_likes_recipe() {
    let result = strings_from_data(
        "person0",
        "recipe0", // Make sure this recipe has at least 60% ingredi
        include_str!("../../hw_06_new/people_categories.txt"),
        include_str!("../../hw_06_new/categories_ingredients.txt"),
        include_str!("../../hw_06_new/recipes.txt"),
    );
    assert_eq!(result.trim(), "person0 likes the recipe 'recipe0'.")
}
```

✅ 4. Known person does NOT like recipe (< 60% match)

Why: Ensures logic correctly blocks when threshold not met.

```rust
#[test]
fn test_person_dislikes_recipe() {
    let result = strings_from_data(
        "person1",
        "recipe2", // Make sure <60% of ingredients match person1's
        include_str!("../../hw_06_new/people_categories.txt"),
        include_str!("../../hw_06_new/categories_ingredients.txt"),
        include_str!("../../hw_06_new/recipes.txt"),
    );
    assert_eq!(result.trim(), "person1 does NOT like the recipe 'rec
}
```

✅ 5. Exactly 60% match (rounding edge case)



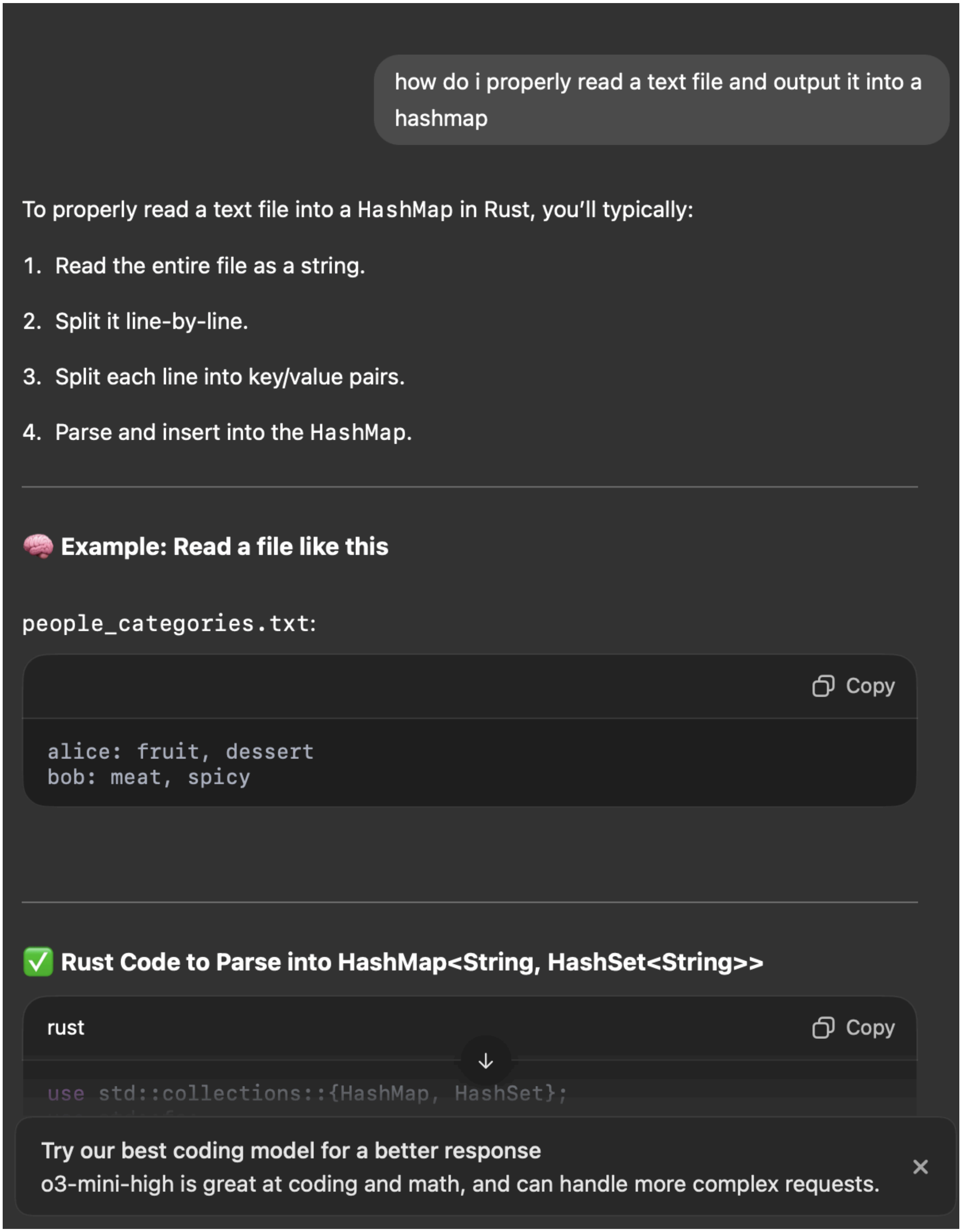
Why: Validates `.ceil()` logic for boundary conditions.

```rust
#[test]
fn test_exactly_sixty_percent() {
    let result = strings_from_data(
        "person2",
        "recipe3", // Has 5 ingredients, person2 likes exactly 3 (60
        include_str!("../../hw_06_new/people_categories.txt"),
        include_str!("../../hw_06_new/categories_ingredients.txt"),
        include_str!("../../hw_06_new/recipes.txt"),
    );
    assert_eq!(result.trim(), "person2 likes the recipe 'recipe3'.")
}
```

✅ 6. “Popular recipes” returns top 3

Why: Confirms overall popularity logic.

```rust
#[test]
fn test_popular_recipes() {
    let result = strings_from_data(
        "popular recipes",
        "",
        include_str!("../../hw_06_new/people_categories.txt"),
        include_str!("../../hw_06_new/categories_ingredients.txt"),
        include_str!("../../hw_06_new/recipes.txt"),
    );
    assert!(result.contains("Top 3 popular recipes:"));
}
```



how do i properly read a text file and output it into a hashmap

To properly read a text file into a `HashMap` in Rust, you'll typically:

1. Read the entire file as a string.
2. Split it line-by-line.
3. Split each line into key/value pairs.
4. Parse and insert into the `HashMap`.

🧠 Example: Read a file like this

`people_categories.txt`:

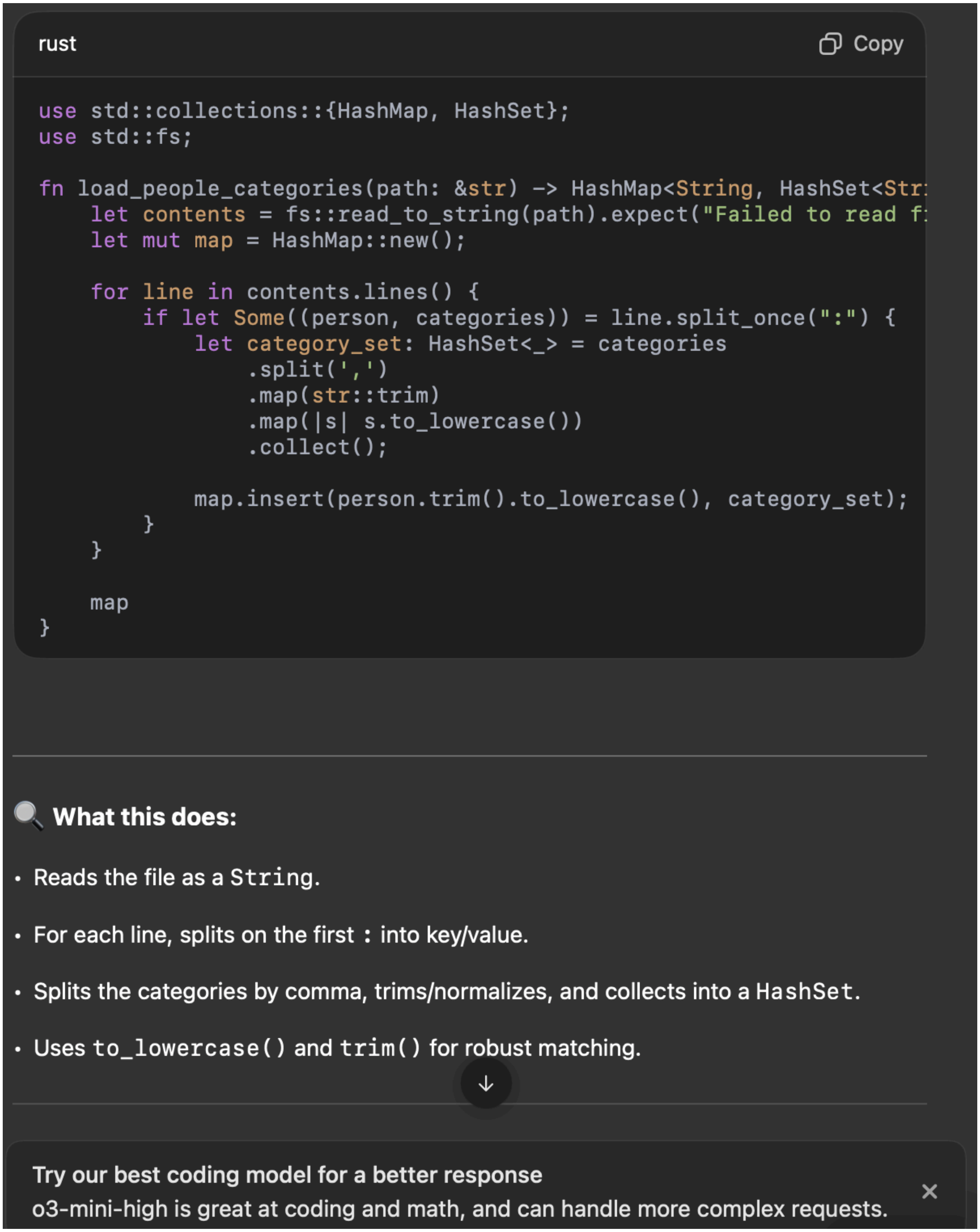
```
alice: fruit, dessert
bob: meat, spicy
```

✅ Rust Code to Parse into HashMap<String, HashSet<String>>

```rust
use std::collections::{HashMap, HashSet};
```

Try our best coding model for a better response

o3-mini-high is great at coding and math, and can handle more complex requests.

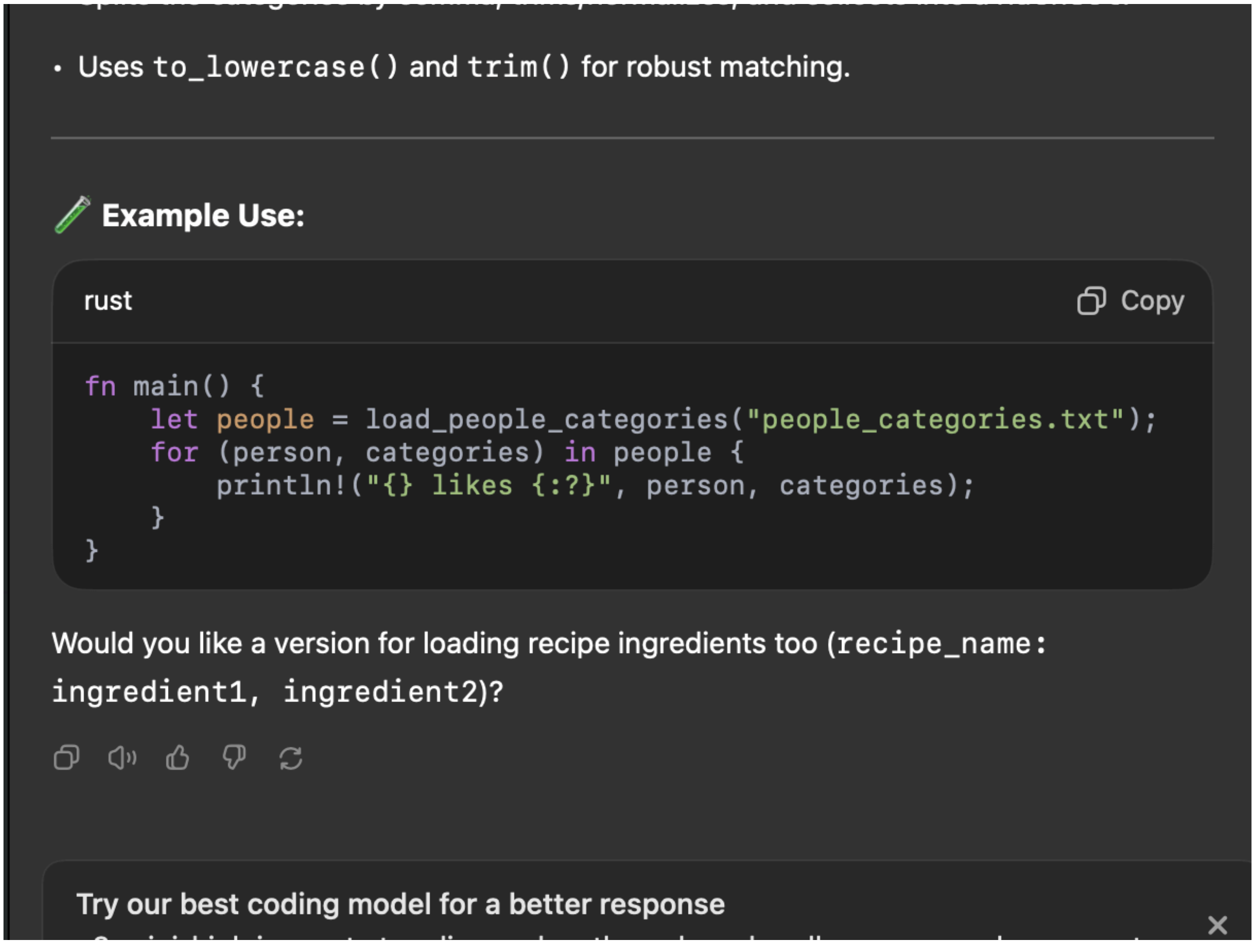


```rust
use std::collections::{HashMap, HashSet};
use std::fs;

fn load_people_categories(path: &str) -> HashMap<String, HashSet<Str
    let contents = fs::read_to_string(path).expect("Failed to read f
    let mut map = HashMap::new();

    for line in contents.lines() {
        if let Some((person, categories)) = line.split_once(":") {
            let category_set: HashSet<_> = categories
                .split(',')
                .map(str::trim)
                .map(|s| s.to_lowercase())
                .collect();

            map.insert(person.trim().to_lowercase(), category_set);
        }
    }

    map
}
```

🔍 What this does:

- Reads the file as a `String`.
- For each line, splits on the first `:` into key/value.
- Splits the categories by comma, trims/normalizes, and collects into a `HashSet`.
- Uses `to_lowercase()` and `trim()` for robust matching.

Try our best coding model for a better response
o3-mini-high is great at coding and math, and can handle more complex requests.



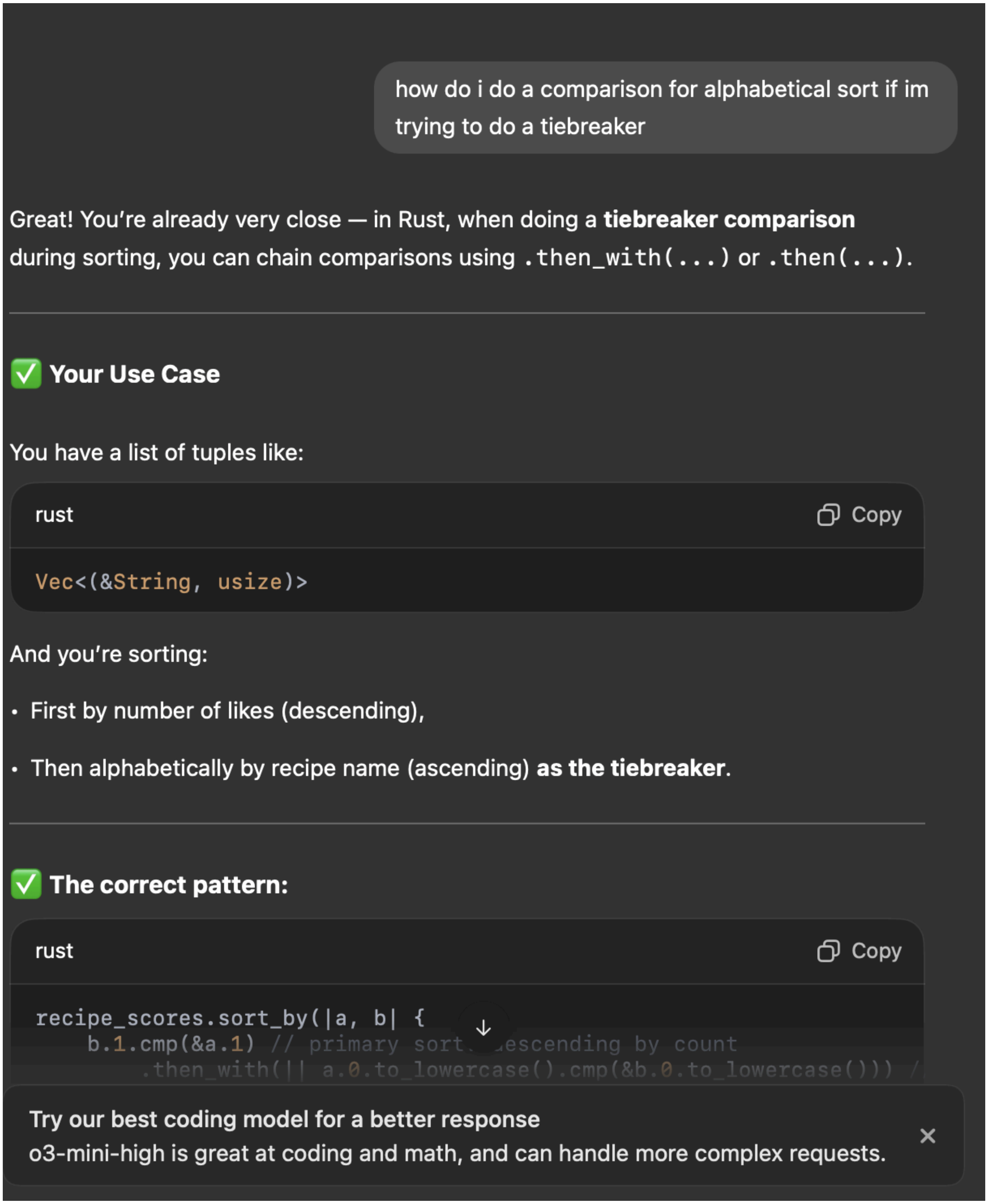
- Uses `to_lowercase()` and `trim()` for robust matching.

🧪 Example Use:

```rust
fn main() {
    let people = load_people_categories("people_categories.txt");
    for (person, categories) in people {
        println!("{} likes {:?}", person, categories);
    }
}
```

Would you like a version for loading recipe ingredients too (`recipe_name: ingredient1, ingredient2`)?

Try our best coding model for a better response



how do i do a comparison for alphabetical sort if im trying to do a tiebreaker

Great! You're already very close — in Rust, when doing a **tiebreaker comparison** during sorting, you can chain comparisons using `.then_with(...)` or `.then(...)`.

✅ Your Use Case

You have a list of tuples like:

```rust
Vec<(&String, usize)>
```

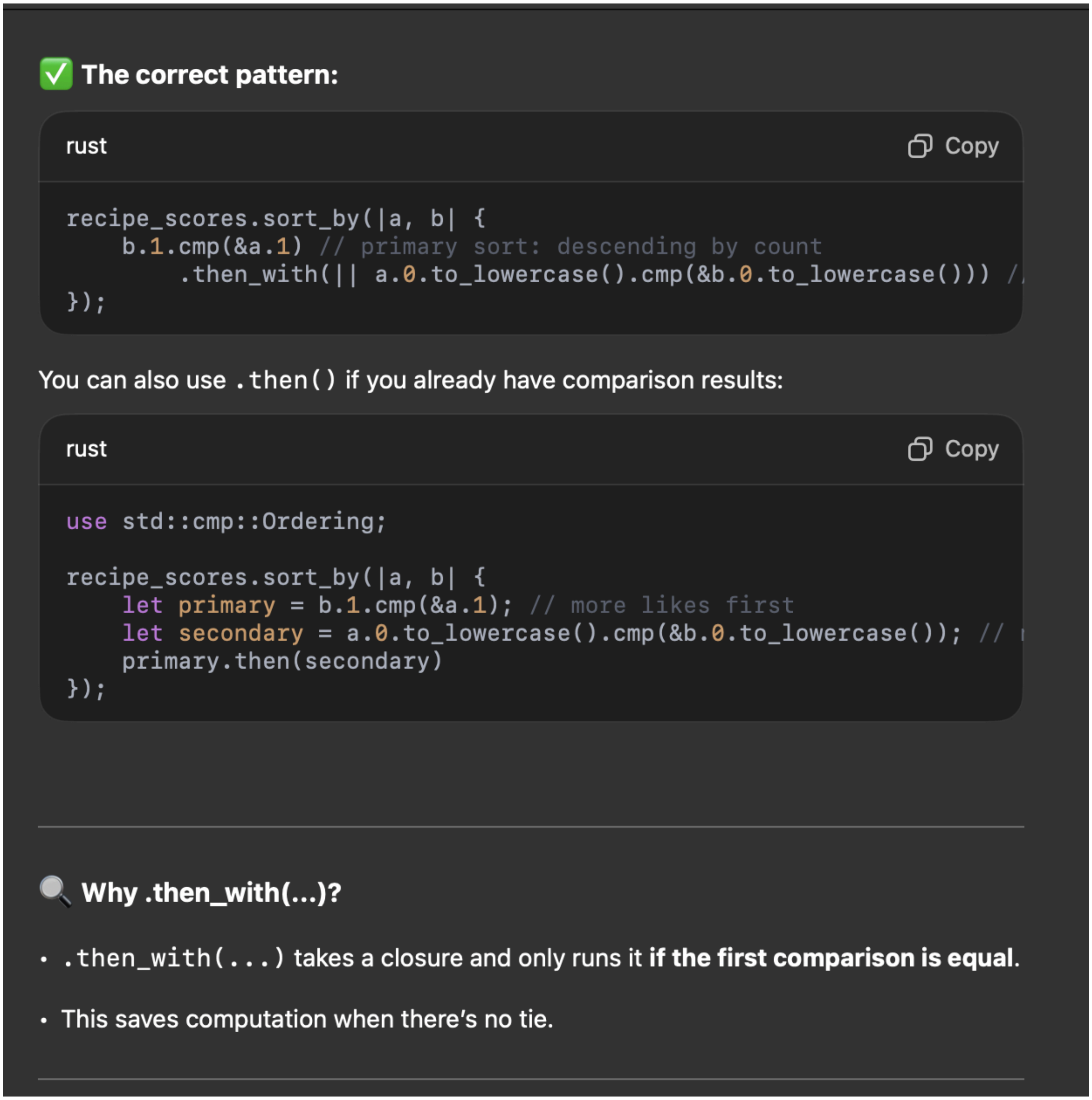
And you're sorting:

- First by number of likes (descending),
- Then alphabetically by recipe name (ascending) **as the tiebreaker**.

✅ The correct pattern:

```rust
recipe_scores.sort_by(|a, b| {
    b.1.cmp(&a.1) // primary sort.[illegible]escending by count
        .then_with(|| a.0.to_lowercase().cmp(&b.0.to_lowercase())) /
```

Try our best coding model for a better response
o3-mini-high is great at coding and math, and can handle more complex requests.



✅ The correct pattern:

```rust
recipe_scores.sort_by(|a, b| {
    b.1.cmp(&a.1) // primary sort: descending by count
        .then_with(|| a.0.to_lowercase().cmp(&b.0.to_lowercase())) /
});
```

You can also use `.then()` if you already have comparison results:

```rust
use std::cmp::Ordering;

recipe_scores.sort_by(|a, b| {
    let primary = b.1.cmp(&a.1); // more likes first
    let secondary = a.0.to_lowercase().cmp(&b.0.to_lowercase()); //
    primary.then(secondary)
});
```

🔍 Why .then_with(...)?

- `.then_with(...)` takes a closure and only runs it **if the first comparison is equal**.
- This saves computation when there's no tie.
