MinSeok Cho

Prof. Chator

CDS DS210 B1

31 Oct 2024

HW6 Report

```rust
fn read_1st(file_name: &str) -> HashMap<String, HashSet<String>> {
    let mut line_read: HashMap<String, HashSet<String>> = HashMap::new();
    let file = File::open(file_name).expect("Check the file location.");
    let buf_reader = std::io::BufReader::new(file).lines();
    for line in buf_reader {
        let line_string = line.expect("Read Error");
        if let Some((name, items)) = line_string.split_once(": ") {
            let name = name.trim().to_string().to_lowercase();
            let items: HashSet<String> = items.split(", ")
            .map(|item| item.trim().to_lowercase()).collect();
        //.map transforms each item in an iterator
        //|item|: one argument which represents each item in the iterator
        //.trim(): removes whitespace
        //.collect(): gathers all the results from map and combines
        //into a new collection type
        line_read.insert(name, items);
        }
    }
    line_read
}
```

I made three different functions that will read and classify the text files so that we can utilize those when we call the functions for each question (1-a, 1-b, 2). Three functions look similar, except the standard of splitting the items. The first text file was divided by comma with space, while the rest of the files were divided into only comma. I made "**line_read**" variable which will store the future collected keys and values (depending on the text file I will call). The "**file**" will call the file to open the file that I will call from the file_name. "**buf_reader**" is a variable that will read the file's context by line and store it to the buffer. This was to enhance the speed of the reading each line of the text file. Then, with the read lines and stores lines in **buf_reader**, I looped each line with **line** and give the line a new name **line_string**. In the line_string, the format will look like big categories and the items in the big categories, split by ": ". After the line_string was divided into these two section, I gave name part to eliminate white spaces and turn that into string, and make it all lowercase so that we can conveniently check for future reference and cross checking the recipes that are liked from the person. For items, as it is styled as HashSet, I splited the inside

items by “, “ or “,” depending on how the text file is formatted. Then, the .map helps us to find certain values inside the HashSet in this case and converts the value to trim whitespaces, turn it do lowercase, and collect. After we done data cleaning and separation, we now have to insert those cleaned data to our initial line_read empty HashMap to store the information.

```rust
fn like_recipe(
    person: &str,
    recipe: &str,
    categories: &HashMap<String, HashSet<String>>,
    people: &HashMap<String, HashSet<String>>,
    recipes: &HashMap<String, HashSet<String>>,
) -> bool {
    //let mut count = 0;
    //when look at recipe,
    //if more than half ingredients is in the person's ingredient list,
    //the person likes it.
    let person = person.to_lowercase();
    let recipe = recipe.to_lowercase();
    if let (Some(person_subject), Some(favorite_ingredients)) = (people.get(&person), recipes.get(&recipe)) {
        let favorite_counts = favorite_ingredients.iter()
        .filter(|each_ingredient| {
            categories.iter()
                .any(|(category, ingredients)| person_subject.contains(&category.to_lowercase()) && ingredients.contains(&each_ingredient.to_lowercase()))
        })
            //favorite_ingredients.contains(&(each_ingredient as String)))
        .count();
        return favorite_counts >= favorite_ingredients.len()/2;
        //let mut favorite_counts = favorite_ingredients.len() / 2;
        //let match_count =
    }
    //let match_count =
    //let mut number_of_ingredients =
    false
}
```

The function “like_recipe” is here to commit 1-a). I called **person** who will be the subject of this function to check whether he or she likes the recipe, **recipe** that will be checked, and categories, people, and recipes that are gathered from the collection function above. As we have to check whether he or she likes the recipe, I made the function to return boolean (true or false). As the **person** and **recipe** will be typed as actual name which may contain uppercase, I had to turn those to lowercase. Next, check whether the person is in the people and recipe is in the recipes and these matches with some value of **person_subject** and **favorite_ingredients**. Next, I will assign new variable favorite_counts by iterating favorite_ingredients, filtering the each segment by assigning each_ingredient which will only count the matched categories that the person liked. Then it will iterate the categories and see if the category meets (category, ingredients) conditions. And it will count how many ingredients meet the criteria of the code. Next, it will return true or false depending on the number of the ingredients that he or she liked mentioned in the recipe is more than half of the ingredients he or she liked. Else, when these whole (which will mean the person or recipe were not found) are not met, we will return false.

```rust
fn what_recipes(
    person: &str,
    categories: &HashMap<String, HashSet<String>>,
    people: &HashMap<String, HashSet<String>>,
    recipes: &HashMap<String, HashSet<String>>,
) -> Vec<String> {
    let mut liked_lists = Vec::new();
    for recipe in recipes.keys() {
        if like_recipe(person, recipe, categories, people, recipes) {
        //as this will return true or false,
            liked_lists.push(recipe.to_string());
        }
    }
    liked_lists
}
```

This is a function for 1-b). I made this to return as vectors of string, and made variable **liked_lists** inside to store the recipes that the person likes. In order to get the keys only from the recipes so that we can figure out whether he or she likes the recipe, we have to call like_recipe again which was made above. If he or she likes it, that recipe will turn into string and be pushed into the vector liked_lists.

```rust
//2): if input name is "popular recipes", produce a
//list of the five most popular recipes (=liked by most people)
//in case
fn five_most(input: &str,
    categories: &HashMap<String, HashSet<String>>,
    people: &HashMap<String, HashSet<String>>,
    recipes: &HashMap<String, HashSet<String>>,
) -> Vec<String> {
    if input != "popular recipes" {
        return Vec::new();
    }
    let mut each_recipe_likes: HashMap<String, usize> = HashMap::new();
        //get ready for each recipe's liked number
    for person in people.keys() {
        for recipe in recipes.keys() {
            if like_recipe(person, recipe, categories, people, recipes) {
                //this shows the person actually likes the recipe or not
                *each_recipe_likes.entry(recipe.to_string()).or_insert(0) += 1;
            }
        }
    }
    let mut recipes_final: Vec<_> = each_recipe_likes.into_iter().collect();
    recipes_final.sort_by(|a, b| {
        b.1.cmp(&a.1).then_with(|| a.0.cmp(&b.0))
    });
    recipes_final.into_iter().take(5).map(|(recipe_name, _)| recipe_name).collect()
}
```

This **five_most** is the function to solve question 2. As we have to run this code when the input is "popular recipes," we need to ignore something else. As we are returning the vector, we will return the empty vector if this is false. If it matches, we proceed to making HashMap to store each recipe's liked from the people. We made it by **each_recipe_likes**, and I looped person and recipe from the people and recipes using .keys() to only refer key from each HashMap. With the condition matching with the like_recipe from the above, we will enter the recipe and give them a value increasing from 0 if the new recipe is added to the HashMap. Next, in order to iterate through the each_recipes_likes that were pushed during person and recipe iteration, now it's time to sort the HashMap and HashSet by descending or lexiological order. In this case, it will sort the numbers that prioritize the numbers that comes first. For example, when we see 10 and 2, we will prioritize 10 first as we see 1 first. After sorting the

recipes, we will take five most liked recipes, by checking the format is matching with |(recipe_name, _)|. We will collect this and return as a result.

This is the result when I put "Halle Vaughn" as a name of like_recipe and what_recipes function. And the last five list with vector types will be the result when I put “popular recipes” in the function.