MinSeok Cho

Prof. Chator

CDS DS210 B1

31 Oct 2024

Works Cited by AI and Resources for HW6

Q:
I think I have to filter out some values and cross-check with other hashmap's value. How can I do with iteration?

Answer:
If you need to filter values from a `HashMap` and cross-check them with values in another `HashMap`, you can use `.iter()` and Rust's `.filter()` method with closures to set the filtering conditions. Then you can perform your cross-checking within the filter or use other methods like `.any()` to search within the other `HashMap`. Here's an example:

Suppose we have two `HashMap`s, `map1` and `map2`, and we want to filter `map1` to find values that meet a certain condition and also exist in `map2`:

```rust
use std::collections::HashMap;

fn main() {
    // Sample HashMaps
    let mut map1 = HashMap::new();
    map1.insert("Alice", 25);
    map1.insert("Bob", 30);
    map1.insert("Charlie", 35);

    let mut map2 = HashMap::new();
    map2.insert("David", 30);
    map2.insert("Eve", 35);
    map2.insert("Frank", 40);

    // Filter map1 values and cross-check with values in map2
    let filtered_values: Vec<_> = map1
        .iter()
        .filter(|&(_, &value)| map2.values().any(|&v| v == value))  // Check if `value` exists in `map2`
        .map(|(key, value)| (key.clone(), *value))
        .collect();

    for (key, value) in filtered_values {
        println!("Key: {}, Value: {}", key, value);
    }
}
```

Explanation:

1. `.iter()` is used to iterate over all key-value pairs in `map1`.
2. `.filter(|&(_, &value)| ...)` is used to filter entries based on a condition. In this example, it checks if `map2` contains a value equal to the current `value` from `map1`.
3. `.any(|&v| v == value)` iterates over `map2` values to see if any of them match `value`.
4. `.map(|(key, value)| ...)` converts the filtered items into the desired format, here as tuples of `key` and `value`.
5. Finally, `.collect()` gathers the filtered entries into a `Vec` or other collection as needed.

This approach allows you to efficiently filter and cross-check values from two `HashMap`s. Let me know if you have a specific filter condition in mind!

Q:

```
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.
    if let (Some(person_subject), Some(favorite_ingredients)) = (people.get(person), recipes.get(recipe)) {
        let favorite_counts = favorite_ingredients.iter()
        .filter(|each_ingredient| 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
}
```

What's wrong here? I am solving 1-a).

Answer:
In your `like_recipe` function, there are a few issues in the logic of checking whether the person likes the recipe:

1. **Incorrect Filtering Condition**:
   In the line `.filter(|each_ingredient| favorite_ingredients.contains(&(each_ingredient as String)))`, the `favorite_ingredients` should not be checked against itself. Instead, you want to check if each ingredient in `favorite_ingredients` belongs to a category liked by the person.

2. **Accessing the Person’s Liked Categories**:
   You should compare each ingredient in the recipe to see if it matches any of the categories the person likes (`person_subject`), rather than trying to find it directly in `favorite_ingredients`.

3. **Returning the Boolean**:
   You are correctly calculating if the person likes the recipe by comparing `favorite_counts` to half the length of `favorite_ingredients`. However, you'll want to clarify the condition so it works with references to `HashSet` items.

Here’s a revised version that corrects the issues:

```rust
fn like_recipe(
    person: &str,
    recipe: &str,
    categories: &HashMap<String, HashSet<String>>,
    people: &HashMap<String, HashSet<String>>,
    recipes: &HashMap<String, HashSet<String>>,
) -> bool {
    if let (Some(person_subject), Some(recipe_ingredients)) = (people.get(person), recipes.get(recipe)) {

        // Count how many ingredients in the recipe belong to categories the person likes
        let match_count = recipe_ingredients.iter()
            .filter(|ingredient| {
                categories.iter()
                    .any(|(category, ingredients)| person_subject.contains(category) && ingredients.contains(*ingredient))
            })
            .count();

        // If at least half of the ingredients match, return true
        return match_count >= recipe_ingredients.len() / 2;
    }

    false
}
```

Explanation of Changes

1. **Inner Filter Condition**:
   - We use `.any()` to check if the current `ingredient` is part of any category that `person_subject` likes.

2. **Counting Matches**:
   - `match_count` represents the count of ingredients that match categories the person likes.

3. **Final Comparison**:
   - We compare `match_count` to `recipe_ingredients.len() / 2` to decide if the recipe has enough matching ingredients for the person to like it.

Q:
In the last line of the what_recipes, I instead wrote return liked_recipes.push(recipe.to_string())
But this was not working. Why this is not working?

Answer:
The line `return liked_recipes.push(recipe.to_string())` does not work because of how the `push` method and the `return` statement operate in Rust.

Key Points

1. **Return Value of `push`**:

   - The `push` method on a vector (`Vec`) does not return the new vector. Instead, it returns `()`, which is the unit type in Rust. This means it indicates that the method executed successfully but does not produce a value that can be used as a return value.
   - In other words, `liked_recipes.push(recipe.to_string())` adds the `recipe` to the `liked_recipes` vector but does not return the vector itself.

2. **Correct Use of `return`**:
   - The `return` statement is used to exit a function and return a value to the caller. If you want to return the `liked_recipes` vector itself, you should do it like this:
     ```rust
     return liked_recipes; // Correct way to return the vector
     ```
   - This line indicates that the function should end and that the current value of `liked_recipes` is what should be returned.

Example of Correct Usage

Here’s how the `what_recipes` function should look to return the vector correctly:

```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_recipes = Vec::new();

    for recipe in recipes.keys() {
        if like_recipe(person, recipe, categories, people, recipes) {
            liked_recipes.push(recipe.to_string());
        }
    }

    liked_recipes // Simply returning the vector
}
```

Conclusion

Using `return liked_recipes.push(recipe.to_string())` won't work because you're attempting to return the result of a method that returns `()`, not the vector itself. Always ensure that the value you're returning is the one you intend to use as the function's output.

Q: I want to make the function only when the input is "popular recipes." How can I reflect this on the function?

Answer:

To ensure the function only executes when the input is "popular recipes", you can add a condition at the beginning of the function. This condition will check the input string and proceed with the logic only if it matches "popular recipes". If the input doesn’t match, you can return an empty vector or an error, depending on how you want to handle it.

Explanation:

1. Check the Input: if input != "popular recipes" checks if the input is not "popular recipes". If it isn't, the function returns an empty Vec<String> and does nothing else.

2. Proceed Only if Input Matches: If input is exactly "popular recipes", the function continues to calculate the most popular recipes, counting likes, sorting, and returning the top five as planned.

This approach ensures the function only processes the data when requested specifically for "popular recipes".

Question:
```
205  |              *each_recipe_likes.insert(recipe.to_string()).or_insert(0) += 1;
     |                                 ^^^^^^-------------------- an argument of type usize is missing
```

Why can' I use .insert() here?

Answer:
The issue here is that `insert` in Rust's `HashMap` does not work exactly like `entry().or_insert()` when you want to update an existing value. The `insert` method only inserts a new key-value pair or updates an existing one, but it doesn't provide direct access to the mutable reference you need for incrementing.

To increment the count for a recipe in your `HashMap<String, usize>`, you can use `entry(recipe.to_string()).or_insert(0)` instead, which provides a mutable reference to the existing or new value associated with the key.

Here's how to fix the line:

```rust
*each_recipe_likes.entry(recipe.to_string()).or_insert(0) += 1;
```

Explanation

- `entry(recipe.to_string()).or_insert(0)` will insert `0` if the `recipe` is not yet in `each_recipe_likes` and return a mutable reference to the value.
- The `*` dereferences the mutable reference, allowing you to add `1` to the value.

This change should resolve the error!

Q:
Do we need <_> for Vec?

Answer:
The `<_>` part in `Vec<_>` is a shorthand for Rust to infer the type of the elements in the vector based on the context. In most cases, you can omit it and just write `Vec`, as Rust can infer the type on its own. However, if the compiler can't determine the type (for instance, in more complex expressions or in cases where the type information is unclear), specifying `Vec<_>` helps guide the compiler.

In your example, you could write:

```rust
let mut sorted_recipes: Vec<_> = each_recipe_likes.into_iter().collect();
```

Or, if you prefer, simply:

```rust
let mut sorted_recipes = each_recipe_likes.into_iter().collect::<Vec<_>>();
```

Both are functionally equivalent and let Rust infer the type. The `<_>` isn't necessary but can sometimes make the code clearer to readers by showing that we expect a vector.

Q:
How do we sort the items in the descending order or lexicographic order in the Hashmap we made already?

Answer:
In Rust, sorting a `HashMap` can be a bit indirect, as `HashMap` itself doesn't maintain any order for its keys or values. However, we can still sort its entries by converting them to a sortable collection, such as a vector. Once we have a sorted vector, we can perform sorting in either descending or lexicographic order.

Given the `HashMap<String, HashSet<String>>` structure you're working with, here's how you could sort the items:

1. **Sort by Keys in Lexicographic Order:**

   To sort by keys in lexicographic (alphabetical) order, you can convert the entries to a vector, sort it, and then collect it back if necessary.

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

   let mut map: HashMap<String, HashSet<String>> = HashMap::new();
   // populate map with data

   let mut sorted_by_key: Vec<_> = map.iter().collect();
   sorted_by_key.sort_by_key(|&(key, _)| key.clone());

   for (key, value) in sorted_by_key {
       println!("{}: {:?}", key, value);
   }
   ```

2. **Sort by Keys in Descending Order:**

   To sort by keys in descending order, you can use `.sort_by_key` with `.reverse()` or `.sort_by` with `Ordering::Greater`:

   ```rust
   let mut sorted_by_key_desc: Vec<_> = map.iter().collect();
   sorted_by_key_desc.sort_by(|a, b| b.0.cmp(a.0)); // b.0.cmp(a.0) for descending

   for (key, value) in sorted_by_key_desc {
       println!("{}: {:?}", key, value);
   }
   ```

3. **Sort by the Number of Items in `HashSet` in Descending Order:**

If you want to sort by the size of each `HashSet`, you can sort based on `value.len()`:

```rust
let mut sorted_by_value_size: Vec<_> = map.iter().collect();
sorted_by_value_size.sort_by(|a, b| b.1.len().cmp(&a.1.len()));

for (key, value) in sorted_by_value_size {
    println!("{}: {:?} ({} items)", key, value, value.len());
}
```

Each of these examples will output sorted key-value pairs. You can adapt the sorting criteria and direction to suit specific needs, such as lexicographic or by other properties in either ascending or descending order.