Mapping Parks & Rec

I. Problem description

Suppose that you're writing a website for California's State and National Parks, and you've been asked to help tourists find their ideal park. To do so, you create a list of all the parks in California, along with all of the activities that visitors can do when they're at each park. For example, you might have this list of parks and their activities:

Park	**Activities**
Half Moon Bay	Biking, Boating, Camping, Dog Walking, Hiking, Kayaking, Swimming
Big Basin Redwoods	Biking, Camping, Dog Walking, Hiking
Andrew Molera	Boating, Camping, Hiking, Swimming,
Mount Diablo	Biking, Camping, Hiking, Riding
Yosemite	Biking, Climbing, Camping, Hiking, Kayaking, Riding
Castle Rock	Climbing, Camping, Hiking, Kayaking

If a customer gives you a list of activities, you can then recommend to them every park that has all of those activities. For example, given the activities "Kayaking," "Hiking," and "Camping," you could recommend Half Moon Bay, Yosemite, and Castle Rock. Given the activity "Biking," you could recommend Half Moon Bay, Big Basin Redwoods, Mount Diablo, and Yosemite. However, given the activities "Riding" and "Swimming," you could not recommend any parks at all.

Write a function

```
Set<string> parksMatching(Map<string, Set<string>>& parksMap, Vector<string>& requirements);
```

that accepts as input a **`Map<string, Set<string>>`** that has parks as keys that are associated with the activities you can do in a given park, along with a tourist's required activities (represented by a **`Vector<string>`**), and then returns a **`Set<string>`** holding all of the parks that match those activities.

Your function should also fulfill the following requirements:

- There might not be any parks that match the requirements, in which case your function should return an empty set.
- You can assume that all activities have the same capitalization, so you don't need to worry about case-sensitivity.
- If a client does not include any activities at all in their requirements, you should return a set containing all the parks, since it is true that all parks match zero requirements.

```
Set<string> parksMatching( Map<string, Set<string>>& parksMap, Vector<string>&
requirements) {
    /* TODO: Fill me in! */
    return {};
}
```

II. Solutions

SOLUTION 1

```
bool hasAllRequirements(const Set<string>& activities, const Vector<string>&
requirements) {
    for (string req: requirements) {
       if (!activities.contains(req)) {
         return false;
       }
    }
    return true;
}

Set<string> parksMatching(Map<string, Set<string>>& parksMap, Vector<string>&
requirements) {
    Set<string> result;
    for (string park: parksMap) {
       if (hasAllRequirements(parksMap[park], requirements)) {
          result += park;
       }
    }
    return result;
}
```

Solution 1 takes advantage of a helper function to help identify if a park has all of the needed requirements. It decomposes the logic nicely to avoid any confusion due to nested for loops, and the helper function passes in the large data structures by reference to avoid making copies. The use of const indicates that the helper will not be editing the data structures, but the keyword is not necessary for the problem to work correctly. Let n be the number of parks, m be the maximum number of activities in a park, and k be the number of requirements. The Big O of this solution is O(n*k*log(m)).

SOLUTION 2

```
Set<string> parksMatching(Map<string, Set<string>>& parksMap, Vector<string>&
requirements) {
    Set<string> result;
    for (string park: parksMap) {
        bool has_requirements = true;
        for (string req: requirements) {
           if (!parksMap[park].contains(req)) {
             has_requirements = false;
           }
        }
        if (has_requirements) {
           result += park;
        }
    }
    return result;
}
```

Solution 2 is very similar to Solution 1 except it does not use a helper function for determining if a park has all of the necessary requirements. A student might come up with the solution if they weren't as comfortable with helper functions or function return values. It has the same Big O as Solution 1.

SOLUTION 3

```
Set<string> parksMatching(Map<string, Set<string>>& parksMap, Vector<string>&
requirements) {
    Set<string> result;
    for (string park: parksMap) {
        int matchingActivities = 0;
        for (string activity: parksMap[park]) {
            for (string req: requirements) {
                if (req == activity) {
                    matchingActivities += 1;
                }
            }
        }
        if (matchingActivities == requirements.size()) {
           result += park;
        }
    }
    return result;
}
```

Solution 3 iterates over all of the park's activities instead of the requirements Vector and keeps a count of the number of matching activities to see if a park contains all of the requirements. For this solution to work, the requirements Vector must not have any duplicate elements. Using the same n, m, and k as above, this would have a Big O of O(n*m*k).

SOLUTION 4

```
Set<string> parksMatching(Map<string, Set<string>>& parksMap, Vector<string>&
requirements) {
    Set<string> activities;
    for (string req: requirements) {
        activities.add(req);
    }

    Set<string> result;
    for (string park: parksMap) {
        Set<string> leftoverActivities = activities - parksMap[park];
        if (leftoverActivites.isEmpty()) {
            result.add(park);
        }
    }
    return result;
}
```

Solution 4 takes advantage of set properties to solve the problem. It first converts the activities vector into a set and then uses set operations between the new requirements set and a park's activities set to see if a park contains all of the necessary activities. Students would need to be very comfortable with Sets and Set math to come up with this approach to the problem. Using the same n, m, and k as above, this Big O of this solution is O(k*n*m*log(k)). These last two factors are due to the fact that subtraction between the two sets will require going through each element in a park's activities list (O(m)) and then removing it from the Set of requirements (O(log k)).

III. Problem motivation

Concept coverage

This question is designed to get students playing around with different container and collection types (here, Vector, Map, and Set) and to work with nested containers. The problem requires an understanding of:

- How to iterate over each of these different data structures
- The ability to manipulate nested data structures (a set stored as values in a map)
- Passing data structures by reference vs. returning them

Keeping track of how everything is stored, in addition to the set you're building up, can be difficult. This problem is also significantly easier to solve if you write a helper function – the logic to handle everything when you don't have a separate helper to check if all the requirements are met is tricky.

Personal significance

From a personal perspective, I struggled with ADTs on the diagnostic exam and wanted to focus on Sets, Maps, and Vectors in my final project. I also hoped to get more practice with Big O, so I used the opportunity to develop my understanding of how using different data structures might affect the runtime of the function. It was interesting to think about how the different data structure input sizes might make one solution a better option than another, and I also thought about how if the problem had passed in the user's requirements as a Set instead of a Vector (making the first for loop in Solution 4 unnecessary), the last solution's runtime would have been O(n*m*log(k)).

I was also interested in real-world applications of these ADTs and wanted to incorporate my interest in the outdoors into the problem. Using this function would be a great way to help me plan my camping trips!

IV. Common misconceptions

Misconceptions or bugs could come in one of three main areas in this problem:

- Looping over all of the parks
- Checking for required activities for a given park
- Building up the resulting set of parks

Maps are unordered data structures, and a student might incorrectly try to use a for loop with a counter variable to index into the key-value items in the map. Students might also loop over the values (activity sets) instead of the keys (parks), which would not prevent them from being able to build up the resulting parks set.

When checking if a park has the given requirements, students might accidentally return true (or set a flag to true) as soon as the first required activity is found instead of after all of the required activities is found. Students may also attempt to exactly match the requirements to the park's activities (instead of just checking if they are a subset of the activities). Lastly, a student may forget to loop over the elements in requirements and may instead try to check if the entire requirements Vector is inside an activities Set (e.g. set.contains(requirements)).

When building up the set of parks, students might accidentally initialize the result Set inside the for loop over the parks, which would prevent it from being returned outside the loop. Alternatively, they may forget to add to the result Set when all activities match (for example, just returning the first park that matches) or forget to return the Set they build up.

Lastly, common bugs when dealing with these data structures include off-by-one errors or accidentally editing the data structures passed in by reference while iterating over them.