Scheduling with Restrictions

I. Problem Description

It's a busy week of the quarter, and on top of doing your assignments, you also have various other small tasks to complete. You can estimate slots of time you will have open to work on all your todos, and the amount of time that each task will take, but some of your tasks need to be completed before others (for example, you need to finish the rough draft of your paper before you can start the final draft). The question is, what are your choices in how to order your tasks? As a small example, let's consider the following three tasks:

Name	**Time (minutes)**
"wash dishes"	5
"brush hair"	5
"shower"	10

If there were no restrictions on time or whether you wanted to complete one task before another, there would be 3! = 6 possible orderings. However, let's suppose that you would like to complete "brush hair" and "shower" before "wash dishes." This brings down the number of possible orderings to only 2, as "wash dishes" must come last and the other two tasks can be done in any order. We can represent restrictions like this as a map of 'prerequisites,' with the keys being the tasks and their values a set of the tasks that must be completed before the key.

Name	**Prerequisites**
"wash dishes"	"brush hair", "shower"
"brush hair"	<none>
"shower"	<none>

Another restriction could be that you only have a time slot of 10 minutes to do tasks (but no prerequisite restrictions). In that case, there are 3 possibilities: "wash dishes" and "brush hair" can be ordered in two ways, "shower" only one way. If we add the restriction that you want to do "brush hair" before "wash dishes," there are only 2 possibilities.

We can represent a task with an **`Activity`** struct that has two fields: **`string name`**, the name of the task, and **`int time`**, the number of minutes the task takes. The **`Activity`** struct has a custom output you should use that prints an **`Activity`** in the format "<name>, <time> min" (example: "wash dishes, 5 min").

Write a function
`int schedule(Vector<Activity>& activities, Map<string, Set<Activity>>& prerequisites, int time);`

that accepts as input a **`Vector<Activity>`** that contains all the activities that you are considering doing, a **`Map<string, Set<Activity>>`** that has names of the activities as keys that are associated with a set of the activities that must be completed before the given activity, and an **`int time`** that is the maximum number of minutes you have to do activities. The function should return an **`int`** that is the number of possible orderings of activities and print out all of these orderings.

Example output (using the case with 2 possibilities from above):

```
brush hair, 5 min
wash dishes, 5 min

shower, 10 min
```

Specifications:

- All of the orderings that are counted and printed should be 'maximized': either all activities in the input **`Vector<Activity>& activities`** have been used, or adding any activity that has not been used yet causes the total amount of time of all the activities to be greater than the input **`int time`**. The order in which the orderings are printed does not matter.
- If an activity does not have any 'prerequisites,' its value in the input **`Map<string, Set<Activity>>& prerequisites`** is an empty set.
- If the input **`Vector<Activity>& activities`** is empty or there are no possible orderings with the given restrictions, the function should return 0 and not print anything.

```
int schedule(Vector<Activity>& activities, Map<string, Set<Activity>>&
prerequisites, int time) {
   /* TODO */
   return 0;
}
```

II. Solutions

Note: Function names have been modified to differentiate between the solutions. See project files for inline comments.

Solution 1

```
int scheduleRecursive(Vector<Activity>& soFar, Vector<Activity>& activities,
Map<string, Set<Activity>>& prerequisites, int remaining) {
   bool done = true;

   for (Activity a : activities) {
       if (a.time <= remaining) {
           done = false;
       }
   }
```

```
   if (done || activities.isEmpty()) {
       if (soFar.isEmpty()) {
           return 0;
       }

       for (Activity a : soFar) {
           cout << a << endl;
       }
       cout << endl;
       return 1;
   }

   int result = 0;

   for (int i = 0; i < activities.size(); i++) {
       Activity curr = activities[i];
       bool allowed = true;
       if (curr.time > remaining) {
           allowed = false;
       }
       else {
           for (Activity a : prerequisites[curr.name]) {
               if (!soFar.contains(a)) {
                   allowed = false;
               }
           }
       }

       if (allowed) {
           soFar.add(curr);
           activities.remove(i);
           remaining -= curr.time;
           result += scheduleRecursive(soFar, activities, prerequisites,
remaining);
           soFar.remove(soFar.size() - 1);
           activities.insert(i, curr);
           remaining += curr.time;
       }
   }
   return result;
}

int scheduleRecursive(Vector<Activity>& activities, Map<string, Set<Activity>>&
prerequisites, int time) {
   Vector<Activity> soFar;
   return scheduleRecursive(soFar, activities, prerequisites, time);
}
```

Solution 1 takes a recursive backtracking approach to check all possible permutations. The base case is if either of the two conditions for a 'maximized' ordering is satisfied: adding any additional activity will exceed the time limit or all activities have been used. The recursive case checks each remaining activity to see if it can be added, which also is according to two conditions: the time limit will not be exceeded and the prerequisites have been completed already. The helper function overloads the given function prototype with an additional input so that it can use a **`Vector<Activity>& soFar`** to keep track of the current ordering. Data structures are passed by reference to avoid copying.

This is a typical choose, explore, unchoose implementation with much of the complexity of the code due to checking conditions. Let n be the number of activities and k be the maximum number of prerequisites that any activity has. The Big O of this solution is **O(n! * n^2 * k)**. Each instance of the function has a runtime of O(n^2 * k), and the function is called about n! times.

Solution 2

```
int scheduleIterative(Vector<Activity>& activities, Map<string, Set<Activity>>&
prerequisites, int time) {
   struct inputAndSoFar {
       Vector<Activity> input;
       Vector<Activity> soFar;
       int remaining;
   };

   int result = 0;

   Stack<inputAndSoFar> stack;
   Vector<Activity> soFar;
   stack.push({activities, soFar, time});

   while (!stack.isEmpty()) {
       inputAndSoFar partial = stack.pop();
       Vector<Activity> input = partial.input;
       Vector<Activity> soFar = partial.soFar;
       int remaining = partial.remaining;

       bool done = true;

       for (Activity a : input) {
           if (a.time <= remaining) {
               done = false;
           }
       }

       if (done || input.isEmpty()) {
           if (!soFar.isEmpty()) {
               for (Activity a : soFar) {
```

```
                    cout << a << endl;
                }
                cout << endl;
                result++;
            }
        }
        else {
            for (int i = 0; i < input.size(); i++) {
                Activity curr = input[i];
                bool allowed = true;
                if (curr.time > remaining) {
                    allowed = false;
                }
                else {
                    for (Activity a : prerequisites[curr.name]) {
                        if (!soFar.contains(a)) {
                            allowed = false;
                        }
                    }
                }

                if (allowed) {
                    soFar.add(curr);
                    input.remove(i);
                    remaining -= curr.time;
                    stack.push({input, soFar, remaining});
                    soFar.remove(soFar.size() - 1);
                    input.insert(i, curr);
                    remaining += curr.time;
                }
            }
        }
    }
    return result;
}
```

Solution 2 is very similar to Solution 1, except it takes an iterative approach by using a loop and stack to mimic recursion. An **`inputAndSoFar`** struct with three fields (**`Vector<Activity> input`**, **`Vector<Activity> soFar`**, and **`int remaining`**) is created to keep track of the same parameters that the recursive solution does. The counter **`int result`** is incremented throughout the loop instead of being returned by smaller cases. Otherwise, the logic is the same as in Solution 1. Accordingly, the Big O of this solution is also **O(n! * n^2 * k)**, using the same n and k as above. Students would need to have a thorough understanding of recursion in order to identify the correct modifications required to translate the recursive solution into this iterative solution.

Although the logic is similar, the iterative solution is not limited by the maximum call stack size of the operating system, so it could potentially handle more activities than the recursive solution. However,

because each pathway to be explored is stored in the stack, this solution takes up more memory. For a reasonable number of activities, the recursive solution may be preferred.

I used the test cases below to check that my solutions worked correctly. My strategy for testing was to first check all edge cases and combinations of restrictions using a small number of activities and then check more standard test cases with a larger number of activities.

```
STUDENT_TEST("Small tests and edge cases"){
   Activity a1 = {"wash dishes", 5};
   Activity a2 = {"brush hair", 5};
   Activity a3 = {"shower", 10};

   cout << "No restrictions" << endl;
   Vector<Activity> activities = {a1, a2, a3};
   Map<string, Set<Activity>> prerequisites;
   int time = 50;
   EXPECT_EQUAL(schedule(activities, prerequisites, time), 6);

   cout << "Prereq restriction" << endl;
   activities = {a1, a2, a3};
   prerequisites[a1.name] = {a2, a3};
   time = 50;
   EXPECT_EQUAL(schedule(activities, prerequisites, time), 2);

   cout << "Impossible prereqs" << endl;
   activities = {a1, a2, a3};
   prerequisites[a1.name] = {a2};
   prerequisites[a2.name] = {a3};
   prerequisites[a3.name] = {a1};
   time = 100;
   EXPECT_EQUAL(schedule(activities, prerequisites, time), 0);

   cout << "Time restriction" << endl;
   activities = {a1, a2, a3};
   prerequisites.clear();
   time = 10;
   EXPECT_EQUAL(schedule(activities, prerequisites, time), 3);

   cout << "Prereq and time restriction" << endl;
   activities = {a1, a2, a3};
   prerequisites.clear();
   prerequisites[a1.name] = {a2, a3};
   time = 18;
   EXPECT_EQUAL(schedule(activities, prerequisites, time), 2);

   cout << "Prereq and time restriction" << endl;
   activities = {a1, a2, a3};
   prerequisites.clear();
```

```
    prerequisites[a1.name] = {a2};
    time = 10;
    EXPECT_EQUAL(schedule(activities, prerequisites, time), 2);

    cout << "No time" << endl;
    activities = {a1, a2, a3};
    prerequisites.clear();
    time = 2;
    EXPECT_EQUAL(schedule(activities, prerequisites, time), 0);
}

STUDENT_TEST("No activities case and larger tests"){
    cout << "No activities" << endl;
    Vector<Activity> activities;
    Map<string, Set<Activity>> prerequisites;
    int time = 50;
    EXPECT_EQUAL(scheduleIterative(activities, prerequisites, time), 0);

    Activity so1 = {"sock1", 1};
    Activity sh1 = {"shoe1", 1};
    Activity so2 = {"sock2", 1};
    Activity sh2 = {"shoe2", 1};
    Activity so3 = {"sock3", 1};
    Activity sh3 = {"shoe3", 1};
    activities = {so1, sh1, so2, sh2, so3, sh3};
    prerequisites[sh1.name] = {so1};
    prerequisites[sh2.name] = {so2};
    prerequisites[sh3.name] = {so3};
    time = 100;
    EXPECT_EQUAL(schedule(activities, prerequisites, time), 90);

    activities = {so1, sh1, so2, sh2, so3, sh3};
    time = 2;
    EXPECT_EQUAL(schedule(activities, prerequisites, time), 9);

    activities = {so1, sh1, so2, sh2, so3, sh3};
    time = 4;
    EXPECT_EQUAL(schedule(activities, prerequisites, time), 54);
}
```

III. Problem Motivation

Conceptual

This problem allows students to practice recursive backtracking and in particular how to keep track of information across function calls, since it is necessary to print out all the chosen activities in the base

case and return the sum of the number of possibilities from each path explored in the recursive case. Due to the restrictions that orderings must adhere to, students need to think carefully about the conditions they need to include in both the base case and recursive case.

Also, students must understand how to work with different data structures (Vectors, Maps, and Sets), using concepts such as iterating through Vectors and Sets and accessing values from a Map. Since the data structures are passed by reference, it is important to be aware of how adding and removing values in a path affect the rest of the recursion. Students should know how to access the fields of a struct as well.

Personal

Before taking this class, I was already aware of the basics of how recursion works, but I had not seen the concept of recursive backtracking before and it was not immediately obvious to me how exactly the choose, explore, unchoose approach worked. Once I had some practice with it, I appreciated how powerful it is and thought about possible uses in combinatorics (in fact, the larger test cases were inspired by a combinatorics problem about putting on socks and shoes).

I actually started the project by thinking about what kind of problem I would like to solve rather than what concept I wanted to use, and I happened to come across the topic of scheduling in a discrete math textbook. I remembered that when doing problems in that topic, it was tedious to manually list all the possibilities of how to order tasks, which seemed like something I could write a function to do instead. We had not covered a recursive backtracking problem before similar to the one I designed, in which it is necessary to check for 'prerequisites,' so I thought it would be interesting to implement. I was also curious as to how recursive functions could be written iteratively, which is how I chose to implement the second solution. It was a learning experience to see that recursion could be done with a loop and stack.

A possible extension to this problem could be to keep track of multiple people doing tasks at once, which is actually more true to the scheduling problems I saw in that math textbook. I decided to keep it simpler for my project, as having multiple people would make it much more complicated to keep track of timing, since the tasks each person has completed so far could take different total amounts of time. However, such a program would be useful for planning a group endeavor, like building a house.

IV. Common Misconceptions

In order to successfully implement a solution to the problem, students should have a mastery of recursive backtracking and basic techniques with data structures (see *III. Problem Motivation* for further details).

The most likely area in which bugs could occur is implementing the conditions of the base case and recursive case of the function. It is recommended that the student clearly understand what the conditions are before beginning to code, as this will make debugging much less arduous if at first the results of test cases do not match the expected number of orderings. The following are some ways in which the recursion could fail to work as expected:

- A form of arm's length recursion is necessary in the base case. Students may think that they can allow the recursion to go one step "too far" and check for a negative remaining time, in which case the function should print out all activities so far except for the last one added. The issue with allowing the remaining time to become negative is that some orderings will be repeated in

the output, as it is possible that multiple activities are chosen and explored at the end of a single, possible ordering.
- Students may forget to check whether the Vector of activities chosen so far is empty in the base case and return 1 in the recursive solution or add 1 in the iterative solution when instead 0 should be returned/added.
- In the recursive case, students may forget to check if an activity being considered will not exceed the remaining time, since a negative remaining time should be avoided as stated above and they could be focusing only on the 'prerequisites' aspect of the problem.
- Since recursive backtracking uses a choose, explore, unchoose method, students must remember to restore the values in the data structures and remaining time to their previous state after the explore step. This is necessary even in the iterative solution, as these steps are contained in a for loop for a given Vector of activities chosen so far.

There are several ways in which the recursion could fail to account for some requirement of the problem, and these bugs do not appear under certain conditions, which demonstrates the need for comprehensive testing. Larger test cases do not guarantee correctness, so it is important to carefully consider the ways in which 'prerequisites' and time limits can have a combined effect on the number of orderings.

Since the problem requires that the function print out all possible orderings, a for loop is needed to iterate through the remaining activities that have not been chosen yet. Students may try to use a for each loop to do this, which will not work because the Vector of remaining activities is modified inside the loop. Creating a copy of the Vector and passing in the modified copy is not recommended as this defeats the purpose of passing in the data structures by reference.

When accessing values from the map of 'prerequisites,' students should be aware that the keys are strings containing the activity names, and not the activities themselves, so it is necessary to access the name field of an activity to use as a key for the map.

Overall, errors are most likely to occur with the logic of the recursion, in most cases returning a larger number of orderings than expected because some have been counted multiple times. Debugging and reexamining the code are essential.