4C: Doctors without Orders

Assignment by Keith Schwarz. Updates by Ashley Taylor.

This problem is more practice with Recursive Backtracking.

This is a pair assignment. You may work in a pair or alone. Find a partner in section or use the course message board. If you work as a pair, comment both members' names atop every code file. Only one of you should submit the program; do not turn in two copies. Submit using the Paperless system linked on the class web site.

Links:

Starter Code

We provide a ZIP archive with a starter project that you should download and open with Qt Creator. You will edit and turn in only the following file:

- **DoctorsWithoutOrders.cpp**, the C++ code for your implementation.

The ZIP contains other files/libraries; do not modify these. Your code must work with the other files unmodified.

Turn in your DoctorsWithoutOrders files here.

Program Description:

Imagine you live in the land of Recursia, which now faces a crisis! No one has told the Recursian doctors which patients to see – they're Doctors Without Orders! As Minister of Health, it's time to, once again, help the Recursians with their medical needs.

Consider the following two structs, which represent doctors and patients, respectively:

```
struct Doctor {
   string name;
   int hoursFree;
};
```

```
struct Patient {
   string name;
   int hoursNeeded;
};
```

Each doctor has a number of hours that they're capable of working in a day, and each patient has a number of hours that they need to be seen for. Your task is to write a function:

```
bool canAllPatientsBeSeen(Vector<Doctor>& doctors,
                          Vector<Patient>& patients,
                          Map<string, Set<string>>& schedule);
```

that takes as input a list of available doctors, a list of available patients, then returns whether it's possible to schedule all the patients so that each one is seen by a doctor for the appropriate amount of time. Each patient must be seen by a single doctor, so, for example, a patient who needs five hours of time can't be seen by five doctors for one hour each. If it is possible to schedule everyone, the function should fill in the final schedule parameter by associating each doctor's name (as a key) with the set of the names of patients she should see (the value). **With the exception of the schedule, all other parameters should be unchanged at the end of your function**.

For example, suppose we have these doctors and these patients:

- Doctor Thomas: 10 Hours Free
- Doctor Taussig: 8 Hours Free
- Doctor Sacks: 8 Hours Free
- Doctor Ofri: 8 Hours Free

- Patient Lacks: 2 Hours Needed
- Patient Gage: 3 Hours Needed
- Patient Molaison: 4 Hours Needed
- Patient Writebol: 3 Hours Needed
- Patient St. Martin: 1 Hour Needed
- Patient Washkansky: 6 Hours Needed
- Patient Sandoval: 8 Hours Needed
- Patient Giese: 6 Hours Needed

In this case, everyone can be seen:

- Doctor Thomas (10 hours free) sees Patients Molaison, Gage, and Writebol (10 hours total)
- Doctor Taussig (8 hours free) sees Patients Lacks and Washkansky (8 hours total)

- Doctor Sacks (8 hours free) sees Patients Giese and St. Martin (7 hours total)
- Doctor Ofri (8 hours free) sees Patient Sandoval (8 hours total)

However, minor changes to the patient requirements can completely invalidate this schedule. For example, if Patient Lacks needed to be seen for three hours rather than two, then there wouldn't be a way to schedule all the patients so that they can be seen. On the other hand, if Patient Washkansky needed to be seen for seven hours instead of six, then there would indeed a way to schedule everyone. (Do you see how?)

Implementation Hints:

This problem is all about recursive backtracking. Think about what decision you might make at each point in time. How do you commit to the choice for that decision and then undo it if it doesn't work? And, as always, feel free to introduce as many helper functions as you'd like. You may even want to make the primary function a wrapper around some other recursive function.

Some notes on this problem:

- There are two ways to think of this problem: are doctors being assigned patients, or are patients being assigned doctors? Think through which one is easier to represent, and if you're having trouble with your approach, try switching to the other.
- You can assume that schedule is empty when the function is called.
- If your function returns false, the final contents of the schedule don't matter (though we suspect your code will probably leave it blank).
- If you find you're "fighting" your code – that an operation that seems simple is actually taking a lot of lines to accomplish – it might mean that you need to change up your data structures.
- You can assume no two doctors have the same name and no two patients have the same name.
- You may find it easier to solve this problem first by simply getting the return value right and ignoring the schedule parameter. Once you're sure that your code is always producing the right answer, update it so that you actually fill in the schedule. Doing so shouldn't require too much code, and it's way easier to add this in at the end than it is to debug the whole thing all at once.
- If there's a doctor who doesn't end up seeing any patients, you can either include the doctor's name as a key in the schedule associated with an empty set of patients or leave the doctor out entirely, whichever you'd prefer.

Testing Details:

We have provided several different test cases for you to try. To test your code, simply input the test file name (e.g. **ComplexNo.dwo**) into the testing code. Our starter code will test both the boolean return value and, in the event that a schedule is possible, the validity of the schedule.

Style Details:

Before getting started, you should read our **Style Guide** for information about expected coding style. You are expected to follow the Style Guide on all homework code. The following are some points of emphasis and style constraints specific to this problem:

Backtracking: You should cleanly represent your choices in parameters and reset or undo those choices when backtracking is required. Any wrapper functions you write should have the minimum number of extra parameters.

Data Structures: You should choose the correct data structure(s) for solving this problem.

Possible Extra Features:

Though your solution to this assignment must match all of the specifications mentioned previously, it is allowed and encouraged for you to add extra features to your program if you'd like to go beyond the basic assignment. Note that our motivation for allowing extra features is to encourage your creativity, not inflate your grade; so if any points are given for your extra features, they will be minimal. The purpose is to explore the assignment domain further, not to disrupt the class grading curve. Make sure to see the notes below about how to separate your extra code and how to turn it in properly.

Here are some example ideas for extra features that you could add to your program:

- **Doctor Specialties**: What happens if some of the doctors have various specialties (ophthalmology, physiatry, cardiology, neurology, etc.) and each patient needs to be seen only by a specialist of that sort?
- **Fair Distribution**: What happens if you want to distribute the load in a way that's as "fair" as possible, in the sense that the busiest and least busy doctors have roughly the same hourly load?
- **Maximize Scheduling**: What happens if you can't see everyone, but you want to see as many people as possible?

Indicating that you have done extra features: If you complete any extra features, then in the comment heading on the top of your program, please list all extra features that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can look at their code easily).

Submitting a program with extra features: Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. You should submit two versions of each modified program file: for example, a first one named **DoctorsWithoutOrders.cpp** without any extra features added (or with all necessary features disabled or commented out), and a second one named **DoctorsWithoutOrders-extra.cpp** with the extra features enabled. Please distinguish them by explaining which is which in the comment header. Our turnin system saves every submission you make, so if you make multiple submissions we will be able to view all of them; your previously submitted files will not be lost or overwritten.