The Stable Roommates Problem

The roommates problem is similar to the marriage problem, but instead of having n men and n women to marry off, we have an even number n of persons, all of the same sex, who must be paired off as roommates. Each person has a list rating the other n-1 persons in descending order of preference. A paring is stable if there are no two persons, not already roommates, who prefer each other to their actual roommates. A less fanciful model is a class with an even number of students who must be paired off into project teams of two persons.

This problem is somewhat more difficult than the stable marriage problem; for one thing, unlike the marriage problem, the roommates problem does not always have a stable solution. The first polynomial time algorithm for the stable roommates problem was given in, "An Efficient Algorithm for the 'Stable Roommates' Problem," *Journal of Algorithms 6*, 577-595 (1985), by Robert W. Irving, and it is the algorithm from this paper we will describe. (You can find this paper online at http://www.dcs.gla.ac.uk/~chrisu/public/Thesis/Thesis/Papers/An%20efficient%20algorithm%20for%20the%20%E2%80%9Cstable%20roommates%E2%80%9D%20problem.pdf)

The first phase of the algorithm is very like the stable marriage algorithm, with individuals making proposals to one another. I've modified the process a bit from Irving's paper, in a fashion that seems to me to lend itself better to functional programming.

When an individual x receives a proposal from and individual y, he provisionally accepts it, simultaneously crossing everyone he rates lower than y off his list. (In particular, he rejects any proposal he may be holding.) Furthermore, x notifies everyone he has crossed off, and they cross x off their lists. This makes the lists symmetric: a appears on b's list if and only if b appears on a's.

An individual x proposes to others in the order they appear on his preference list. The procedure in the previous ensures that each proposal will be provisionally accepted, but it may be rejected later if the accepter gets a better offer. In that case, x proposes to the first person remaining on his list.

Note that, just because a person provisionally accepts a proposal, he is not barred from proposing himself. People stop proposing when their proposals are accepted, not when they accept a proposal. Also note that, any time y holds a proposal from x, y is first on x's preference, and x is last on y's preference list.

This phase of the algorithm ends when either every person holds a proposal, or one person has been rejected by everyone. In the latter case, no stable matching is possible, so we can report failure and terminate the algorithm. On the other hand, if all the lists are of length one, then we have a stable matching, that matches each person with the single individual on his preference list.

If some of the reduced preference lists have length greater than one, we have to perform what Irving calls a "second-stage reduction," but I just call it a "rotation." This starts out by finding what Irving calls an "all-or-nothing cycle." This is a (cyclic) sequence of distinct persons a_0, a_1, ..., a_r, such that for k = 0, 1, ..., r-1, the second person in a_k's reduced preference list (call him b_{k+1},) is the first person in a_{k+1}'s list. Finally, the second person in a_r's reduced preference list is the first person in a_0's. We'll call this person b_0.

Now, we force each b_k to reject the proposal he holds, so that each a_k is now forced to propose to his second choice. Then we can prune the preference list, just as we did in the stage 1 reduction. We have the same criteria here. If somebody' list gets reduced to nothing, there is no stable matching; if all the reduced preference lists are of length one, the implied matching is stable. If neither condition holds, we do another stage two reduction. Since at least one list gets shorter at every step, the process must terminate. I'll discuss how to find such an "all-or-nothing cycle" later on.

Here is an example, taken from Irving's paper, with the modifications I've made. We have six individuals, with the preference lists shown.

1:	4	6	2	5	3
2:	6	3	5	1	4
3:	4	5	1	6	2
4:	2	6	5	1	3
5:	4	2	3	6	1
6:	5	1	4	2	3

According to rules, we get the following sequence of proposals and rejections:
1 proposes to 4; 4 holds 1; 4 crosses off 3; 3 crosses off 4
2 proposes to 6; 6 holds 2; 6 crosses off 3; 3 crosses off 6
3 proposes to 5 (3 crossed off 4 earlier); 5 holds 3; 5 crosses off 6 and 1; 6 and 1 cross off 5

At this point, the lists look like this:

1:	4	6	2		3
2:	6	3	5	1	4
3:		5	1		2
4:	2	6	5	1	
5:	4	2	3		
6:		1	4	2	

The process continues:
4 proposes to 2; 2 holds 4;
5 proposes to 4; 4 holds 5 and rejects 1; 1 crosses off 4
1 proposes to 6 (1 just crossed off 4); 6 holds 1 and crosses off 4 and 2; 2 and 4 cross off 6

Now the lists look like this:

1:		6	2		3
2:		3	5	1	4
3:		5	1		2
4:	2		5		
5:	4	2	3		
6:		1			

2 proposes to 3; 3 holds 2;
6 proposes to 1; 1 holds 6 and crosses off 2 and 3, who cross off 1

Now the lists look like this, with everyone holding a proposal from the last person on his list.

1:	6		
2:	3	5	4
3:	5	2	
4:	2	5	
5:	4	2	3
6:	1		

Since some lists have length greater than one, we must perform a rotation. We can take $a_0 = 3$, $a_1 = 4$, $b_0 = 5$, $b_1 = 2$. Note that the second person on a_0's list is b_1, and the second person on a_1's list is b_0. Also, the last person on b_0's list is a_0, and the last person on b_1's list is a_1. Each b_k holds a proposal from a_k, and we force him to reject it. That is, we force 2 to reject 4 and 5 to reject 3.

We now have these preference lists:

1:	6		
2:	3	5	
3:		2	
4:		5	
5:	4	2	
6:	1		

Now 2 and 5 hold no proposal, and 3 and 4 have not been accepted. So 3 proposes to 2; 2 accepts 3 and crosses off 5; 5 crosses off 2. Finally, 4's proposal to 5 is accepted and all lists have been reduced to singletons.

We have the matching: 1 ↔ 6, 2↔ 3, 4 ↔ 5.

Pruning the Preference Lists

Earlier I said, "When an individual x receives a proposal from and individual y, he provisionally accepts it, simultaneously crossing everyone he rates lower than y off his list. (In particular, he rejects any proposal he may be holding.) Furthermore, x notifies everyone he has crossed off, and they cross x off their lists." Of course, this is figurative; we don't have a message-passing facility.

Suppose we write a function crossOff so that `crossOff pref (x, y)` returns a modified pref function with x crossed off y's list and y crossed off x's Once we get a formula for the list of people to be crossed off x's list, we can just use a list comprehension to make a list of pairs (x, y) where y runs over the people x is crossing off.

Now it easy enough to write a recursive function to modify the pref function as we want, or we can use magic instead: look up the foldl function.

Finding an "All-or-Nothing Cycle"

Suppose that there are 10 people, numbered 0 through 9, and that when everyone is holding a proposal, the preference lists are

0: 4 9 2
1: 2 3 9
2: 0 9 3 1
3: 7 1 4 2 9 5
4: 9 3 0
5: 3 7
6: 8
7: 5 3
8: 6
9: 1 3 0 2 4

To find a cycle, we start with anyone whose preference list has more than one person, and we choose to start with 0. He is our first p. The first q that we take is p's second choice, or 9. Now Irving's paper indexes the p's and q's, but when working with lists, the indices are not so important as when working with arrays, so we'll just think of a list of p's and a list of q's. That is, we have ps = [0] and qs = [9]. The next p is 9's last choice, namely 4. Recall that 9 must hold 4's proposal. The next q is 4's second choice, that is, 3. In Haskell, it's easier, and much more efficient, to build up lists by adding elements at the front than the back, so we'll say that now ps = [4, 0] and qs = [3, 9]. Now 3's last choice is 5, and 5's second choice is 7, so we get ps = [5, 4, 0] and qs = [7, 3, 9]. Another step gives us ps = [3, 5, 4, 0], qs = [1, 7, 3, 9], and the next step gets us to ps = [9, 3, 5, 4, 0], qs = [3, 1, 7, 3, 9]. Now, the next p should be 3's last choice, 5, which

would make the next q 5's second choice, 7. However, 5 is already in the ps list, so we have found a cycle. Now we know that the ps cycle is [9, 3, 5], but it's easier to extract it as [5, 9, 3]. (Take everything that isn't 5, and then put 5 at the front of the list.) We know that the beginning of the qs list must be a cycle too, and it's the same length as the ps cycle, so the qs cycle is [3, 1, 7].

We want to zip the two cycles together and then cross each member of a pair off the other's preference list, using the crossOut function I described earlier. If we zip the two cycles, we get [(5, 3), (9, 1), (3, 7)]. Note that 3 holds a proposal from 5, 1 holds a proposal from 9, and 7 holds a proposal from 3, so this is just what we need. We zip ps and qs and use the crossOut function to modify the preference lists. Now the proposal process can start again, with the q's holding no proposals, and the p's not having a proposal accepted.

Note: As we generate the lists, the ps and qs are out of sync, since each q is holds a proposal from the preceding p. When we insert a p at the beginning of the cycle, however, the ps and qs are aligned correctly.

Project

Your program should prompt the user for an input file name. You can either automatically compute the name of the output file from the input file name, or prompt for an output file name. In my program, for example, I compute the name of the input file by dropping the extension from the input file name, and then appending ".out.txt".

There will be no errors in the input files I use to test your program, so you need not edit the input. The input file will have n lines, for some even number n, and each line will have contain n strings, where each string is the name of one of the individuals to be paired up. The second through the last names on the line constitute the preference list of the person listed first on the line, in descending order. So, if the line is

Alex Bob Carl David

then Alex's first choice is Bob, his second choice is Carl, and his third choice is David.

If a stable matching exists, your program should produce n lines of output, containing with two names, separated by a blank. The first names will be the same as the first names on the input lines, in the same order. The second name will be the name his "roommate." (Notice that if Alex and Bob are matched, then the pair will appear twice; once as "Alex Bob," and once as "Bob Alex.")

You may have noticed that by using the "do" construct it is possible to write imperative code in Haskell. Don't do that! Follow the model in the sample code I gave you for the stable marriage problem. That is, use sequential code only for I/O. Solve the problem with functional programming.

Advice

Don't just start coding. Plan what your functions will be. What are their arguments? What are their values? Debugging a functional program is different from debugging an imperative program. You do it more by thinking about what the code is supposed to do. Generally, you build a functional program by putting together rather small, simple functions, and you test these one by one to see that they produce the values you expect.

Try to avoid using the !! (indexing) operator. Not only is this highly inefficient for accessing the elements of a linked list, but it doesn't lend itself well to functional code.

You'll probably find it worthwhile to write a little python script to check your answers. The script should read the input and output files, and test whether the matching is stable. This is very easy. For each person x, with partner y and for each individual z that x prefers to y, check if z prefers y to z's partner. If so, there is an instability. If no instability is found, then the matching is stable.

```
for each x
    let y = partner of x
    for each z that y prefers to x
          let w = partner of z
          if z prefers y to w
                return unstable
return stable
```

Alternatively, you might write a Haskell function to test the answer.

Besides turning in your code, each team will make a brief presentation, consisting of a code walkthrough and a demonstration. I will supply test data files for the demonstration.

Appendix

It isn't necessary to understand why the algorithm works to do the project, and of course, if you're really interested you should read Irving's paper, but here's a sketch for those who are only casually interested.

The first part of the algorithm depends on the following lemma, which has a short, easy proof given in the paper.

Lemma: If at any time, in stage 1, x rejects y, then there is no stable matching in which x and y are paired.

This means that if anyone is rejected by everyone else, he has no possible partner in any stable matching, so no such matching exists. It's also easy to see that if every preference list has been reduced to a singleton, the implied matching will be stable. A consequence of the lemma is that, at the end of phase one, there is no stable matching in which a person can have a partner better than the first person on his list, or worse than the last person on his list.

All of this is straightforward, and not at all surprising, but the rotation procedure seems mysterious. Here is where Irving has the crucial insight. He shows that, in any stable matching, either each p is paired with corresponding q, or no p is paired with corresponding q. (Hence the name "all-or-nothing cycle.") Furthermore, he shows that there exists a stable matching where all are paired if and only if there exists a stable matching where none are paired. Therefore, while forcing the rejections eliminates certain solutions, it won't prevent us from finding a solution if one exists.

Of course, we have to prove that the fundamental lemma and it consequences, suitably modified, apply after the rotation, but this is straightforward.

Much of the paper is concerned with data structures for an efficient algorithm in an imperative programming language, using arrays, and has no application in this assignment.