MCS 275 Project Three : finding lists of stable pairs 
due Wednesday 19 March at 11AM

The goal of the project is to create a web interface for an algorithm to find a list of stable pairs with minimal rank.

The task of a software developer responsible for a team of six programmers – we call them Alice, Brian, Cindy, David, Eve, and Fred – is to form pairs of programmers. Each programmer provides her or his preferences with whom they would like to work. For example, the preference list for Alice is Eve, David, Cindy, Fred, Brian, listed in increasing order. Alice really likes to work with Eve, and can get along with David and Cindy, and even Fred, but she does not like Brian. To form \( k \) pairs, \( n = 2k \), the input to the program will be a list of \( n \) permutations of \( n - 1 \) numbers. Each permutation lists the preferences for collaboration in increasing order.

The pairs (Brian, Eve) and (Cindy, David) are not stable if the pairing (Brian, Cindy) and (Eve, David) is preferable to all four: if Brian likes Cindy better than Eve, Cindy prefers Brian over David, David likes Eve better than Cindy, and Eve prefers David over Brian. Likewise, if the pairing (Brian, David) and (Cindy, Eve) would be preferable to all four, then the pairs (Brian, Eve) and (Cindy, David) should then also not be generated.

We define the rank of a pair as the sum of their positions in the preference list. The rank of the pair (Brian, Eve) would be 0 + 1 if Eve is Brian’s first choice and Brian is Eve’s second choice. The rank of a list of pairs is the sum of the ranks of each pair. We want the list of stable pairs with minimal rank.

After prompting the user for the number of pairs, the user has the choice between entering lists of preferences or having the computer generate the permutations at random. The program will show the computed list of stable pairs along with their rank. At the end of the program, the list of stable pairs with minimal rank is printed.

Examples of runs with the program `stabpairs.py` are listed below.

```
$ python stabpairs.py
Give #pairs : 2
random preferences ? (y/n) n
list of preferences for 0 : [1, 2, 3]
list of preferences for 1 : [0, 2, 3]
list of preferences for 2 : [0, 1, 3]
list of preferences for 3 : [0, 1, 2]
Lists of preferences :
[[1, 2, 3], [0, 2, 3], [0, 1, 3], [0, 1, 2]]
Stable pairs with their rank :
[(0, 1), (2, 3)] rank = 4
[(0, 2), (1, 3)] rank = 4
[(0, 3), (1, 2)] rank = 4
Stable pairs with lowest rank :
[(0, 1), (2, 3)] rank = 4
```
Every pair is stable because there is a strictly linear order of preferences: the $i$-th person is preferred to the $(i+1)$-th. Entering this linear order will list all possible pairs. Note that $(0,1)$ and $(1,0)$ define the same pair.

For random choices of preferences, the backtracking algorithm starts the enumeration along the order of preferences. The current list of pairs with the lowest rank is kept. Pairs that lead to a rank higher than the current lowest rank are not generated.

$$\texttt{python stabpairs.py}$$

Give #pairs : 4
random preferences ? (y/n) \texttt{y}
Lists of preferences :
[[2, 5, 7, 6, 3, 1, 4], [3, 5, 7, 2, 6, 0, 4], [4, 6, 3, 1, 0, 7, 5],
 [1, 5, 7, 2, 4, 6, 0], [7, 2, 1, 3, 0, 5, 6], [3, 2, 1, 7, 4, 0, 6],
 [1, 0, 5, 7, 3, 2, 4], [3, 6, 0, 2, 4, 5, 1]]
Stable pairs with their rank :


Stable pairs with lowest rank :


The algorithm will not print all possible pairs, instead we will see a gradual decrease in rank. Even as the complexity of this problem might not allow for the program to terminate in a reasonable time, we will see a gradual improvement in the solutions.

The program will be used as a module and imported in the scripts which provide the web interface.

The web interface consists of two Python scripts. The first Python script will display a page to enter the number of pairs. This page must contain an input element for the user to enter this number and a submit button. Pressing the submit button activates the second script. If $n$ denotes the number of people, then the second script will bring up $n$ pages. The $i$-th page has $n$ input elements: the first input element is the name of the $i$-th person and the remaining $n-1$ input elements will be used to enter the order of preferences. Each page has a submit button, either to go to the next page to enter the preferences of the $(i+1)$-th person, or to call the enumeration of the stable pairs routine (in the stabpairs module).

Note that in a real web interface application, every person would login separately to the web server, submit login name and password, and then enter her or his order of preferences. In this project, the web interface is used by the same person who enters all data.
The web interface starts by running `web_pairs1.py` in a browser:

![Image of web_pairs1.py interface]

The submit launches the second script `web_pairs2.py`. Each time the user submits, a new page comes up, asking for the preferences of the next person. Notice that the same script `web_pairs.py` collects all input.

![Image of web_pairs2.py interface]

Notice that all information in the web interface is alphanumeric, although for testing purposes, you could abbreviate the names by single letters. One way to store the preferences is to work with a file, collecting the strings from the $n$ input elements on one page into one list that is appended as one line to the file.

At the last submit, the program converts the alphanumeric data on file into the lists of permutations. These lists of permutations is the input to the backtracking function of `stabpairs.py`. The output of the backtracking function is then converted back into alphanumeric format. To keep the web interface simple, only the final list of stable pairs with minimal rank is displayed.
The information displayed at the end summarizes the input: the names of the people and their preferences, before showing the list of stable pairs with minimal rank:

Some important points:

1. Since this project consists in two parts, you may collaborate in pairs.
   Both authors will receive the same number of points.

2. Representing preferences as lists of lists is convenient because of the `index()` method.

3. You may assume that all input entered by the user is correct.

4. Use a clear functional design to implement the backtracking algorithm. Provide appropriate documentation, i.e.: every function must have a documentation string.

5. Handing in an incomplete but working program is better than handing in a program that crashes or does not run at all.

6. The solution consists of three files: the first file is the module where the main program calls the backtracking algorithm. The function which implements the backtracking is called in the second web script.

7. The first line of your Python programs must be

   # MCS 275 Project Three by <Authors>

   where you replace the `<Authors>` by your names.

8. Email your solution to the project to `jan@math.uic.edu` before 11AM on Wednesday 19 March so the date of the email is proof of an on time submission. As a backup, bring also a printed version of your solution to class.

If you have questions or difficulties with the project, feel free to come to my office for help.