Project 3: Working with MapReduce

DS 730

Overview

In this project, you will be working with input, output, Python and the MapReduce framework. You will be writing multiple mappers and reducers to solve a few different problems. None of the code you write will be multithreaded but the code that you create for this project will eventually be plugged into Hadoop and it will automatically parallelize your program for you. In short, you will be writing rather simple code that works on 1 machine right now. Eventually we will be using tools to help us parallelize the code so that we can solve our problems much faster.

As said in lecture, the map function is **stateless**. The intermediate output the mapper produces must only depend on the current input value. Similarly, the output the reducer produces must only depend on a particular key. There may be multiple intermediate pairs with the same key and this is quite alright. We will be guaranteed to have all intermediate pairs with the same key end up on the same reducer. However, you can't assume that pairs with different keys will end up on the same reducer and therefore cannot remember any different keys that happened in the past. Doing it this way allows us to run map in parallel on many machines and also run reduce in parallel on many machines. Because the Hadoop tool will eventually sort our data after mapping, it is ok to sort the data by key in the map function for this project.

You are not allowed to use the information from the first input line to affect the output for the second input line.

Example:

Think back to the word count problem. Let's say the word "`cat`" appeared two times in a row. You'll recall our intermediate values were **(cat, 1)** and **(cat, 1)**. It would be *wrong* for the mapper to output **(cat, 2)** because *the second time we see `cat`, we would be remembering that we just saw cat and this is not stateless*. We are relying on the fact that `cat` was just seen in order to output **(cat, 2)**. This is not how MapReduce works.

Therefore, we must output **(cat, 1)** and **(cat, 1)**. If that isn't clear, think of it this way: what happens if your input file is split right between the two `cat` s and 1 `cat` is sent to one processor and the other cat is sent to another processor. There is no communication between the processors, so the second one would have no idea what the first one is producing. Therefore, the mappers would each produce a **(cat, 1)** pair.

The bottom line is this: you should not rely on previously read in information to affect your current output. If you are unsure, ask me and I'll tell you.

Project Tasks

You must write a mapper file and a reducer file to solve each of the following problems. Each mapper and reducer must be in its own file for each problem. You should end up with **3 mapper files** and **3 reducer files** for this project. For all of these problems, you must read in from the command line. Refer to the **`sys.stdin.readline()`** function for reading in from the command line. The MapReduce example in the slides goes over a couple of examples. In order to output to the command line, use the **`print(…)`** function (again, see slides for examples).

I am assuming that many of you are familiar with Python from the course prerequisites. You should have done some basic input and output, and that is all you will be doing in this project. You will read in from the command line and output to the command line. However, typing in a ton of data at the command line is a bit tedious. Instead, you'll want to redirect your input from a file and then redirect your output to a file. Since the virtual lab is in Windows, I will demonstrate how to do this on the command line on a Windows machine. If you are using something else, look for "input redirection" and you'll find what you need.

Let's say your mapper program for number 1 is called **mapper.py** and you want to read in from a file called **input.txt.** You want to output your intermediate (key, value) pairs to a file called **intermediate.txt**. To do this in Windows, you would do the following:

```
python mapper.py < input.txt > intermediate.txt
```

The **<** operator is a redirect operator. It is telling Windows, I want to send everything from **input.txt** to this program. Similarly, the **>** is redirecting your print statements to a file called **intermediate.txt** instead of printing it to the command line. Your intermediate (key, value) pairs are stored in a file called intermediate.txt.

Let's assume your reducer is called **reducer.py** and you want to output your answer to **output.txt**. You would do the following:

```
python reducer.py < intermediate.txt > output.txt
```

The reason we are splitting them this way is because eventually we will be using Hadoop's streaming feature. If you write your code in this fashion, you'll be able to use it with Hadoop without having to change a thing. It is also a good way to reinforce that the mapping phase and reducer phase are completely separate phases.

Problem 1: Words with same number of vowels

Read in your data from the command line. The problem will be similar to the problem we worked on in lecture with a small twist. Instead of printing out how many times a word appears in the file, you want to print out how many words have the same number of vowels. For this problem, only the number of vowels matter. The actual vowel is not important. The output will be the number of vowels (let's call it **x**), followed by a colon, followed by the number of words that had **x** vowels. The output is sorted by the number of vowels.

Example:

Let's say the file contained the following words:

`hello how are you doing today hello`

The output would be:

```
1 : 1
2 : 4
3 : 2
```

Words are separated by spaces, tabs and new lines only. As an example, "half-time" would be considered 1 word.

Using the example above, since 'how' contains 1 vowel, the first output is **`1 : 1`** (i.e. exactly 1 vowel occurred in exactly 1 word).

'hello', 'are', 'doing' and 'hello' contained 2 vowels each. This is why we ended up with **`2 : 4`**. A word appearing more than once will be counted more than once.

Finally, 'today' and 'you' contained 3 vowels. For this problem, we will assume 'y' is a vowel. The final output line was **`3 : 2`**.

You must output your answer to the terminal window. The above output example is sorted but the output need not be sorted in the end.

Problem 2: Words with exact same vowels

The second problem will be a slight modification of the first one. This problem will read in from the command line and output the number of words that contained the exact same vowels.

Example:

'hello' and 'pole' both contain exactly 1 e and exactly 1 o. The order of the vowels does not matter.

Imagine the input contains:

`hello moose pole`

We would end up with the following output:

```
eo : 2
eoo : 1
```

We will also assume ‘y’ is a vowel for this problem.

You must output your answer to the terminal window. The format should be as seen above: the vowels in alphabetical order, followed by a colon, then followed by the number of words that used those vowels. The actual keys needn’t be in alphabetical order. In other words, the key ‘aeo’ could occur after the key ‘oee.’

Problem 3: Clothing coordination

The third problem will be dealing with a problem that recommender sites deal with. Imagine an online store that sells clothing. The website has a list of items that pair well together. Let’s say shirt number 1 is usually bought with pants number 6 or shoes number 10. If there is only 1 item in the shopping cart, it is easy to recommend those pants or those shoes. However, let’s say a user wants to buy shirt number 1 and shirt number 2. Is there a list of items that pairs well with both 1 and 2? This is the motivation behind this problem: a recommender system based on multiple input.

Read in from the command line. The file will be constructed in the following fashion with the following list for 1 particular item *appearing on **one** line* (*disregard the line break in this document*):

> **ClothingID : ClothingIDOne ClothingIDTwo ClothingIDThree… !**
> **NotClothingOneID NotClothingTwoID NotClothingThreeID…**

Each item list will be on its own line (see example below). All of the items that pair well with **ClothingID** are listed before the exclamation point. All of the items that don’t pair well with ClothingID are listed after the exclamation point. All IDs will be integer values to keep things simple. If clothing item 1 pairs well with clothing item 2, then clothing item 2 also pairs well with clothing item 1. You can imagine each clothing item has a unique ID associated with it. The goal is this: produce all combinations of clothing items and the clothing items that they all pair well with.

The following pages walk you through example input and output.

As a starting point, consider the example input and starting output shown below. The next few pages go into more detail about how to interpret the output.

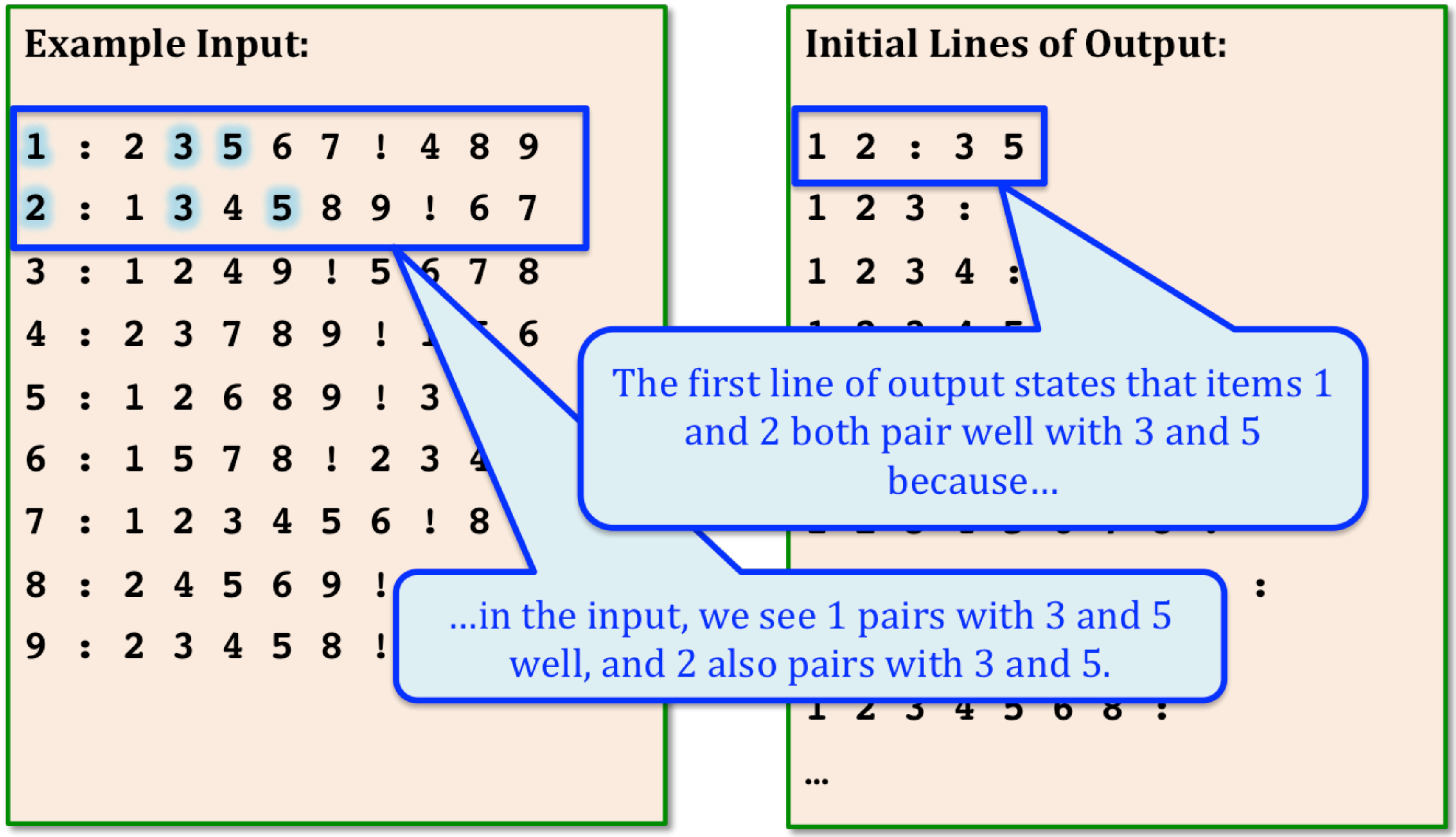
Example Input:

```
1 : 2 3 5 6 7 ! 4 8 9
2 : 1 3 4 5 8 9 ! 6 7
3 : 1 2 4 9 ! 5 6 7 8
4 : 2 3 7 8 9 ! 1 5 6
5 : 1 2 6 8 9 ! 3 4 7
6 : 1 5 7 8 ! 2 3 4 9
7 : 1 2 3 4 5 6 ! 8 9
8 : 2 4 5 6 9 ! 1 3 7
9 : 2 3 4 5 8 ! 1 6 7
```

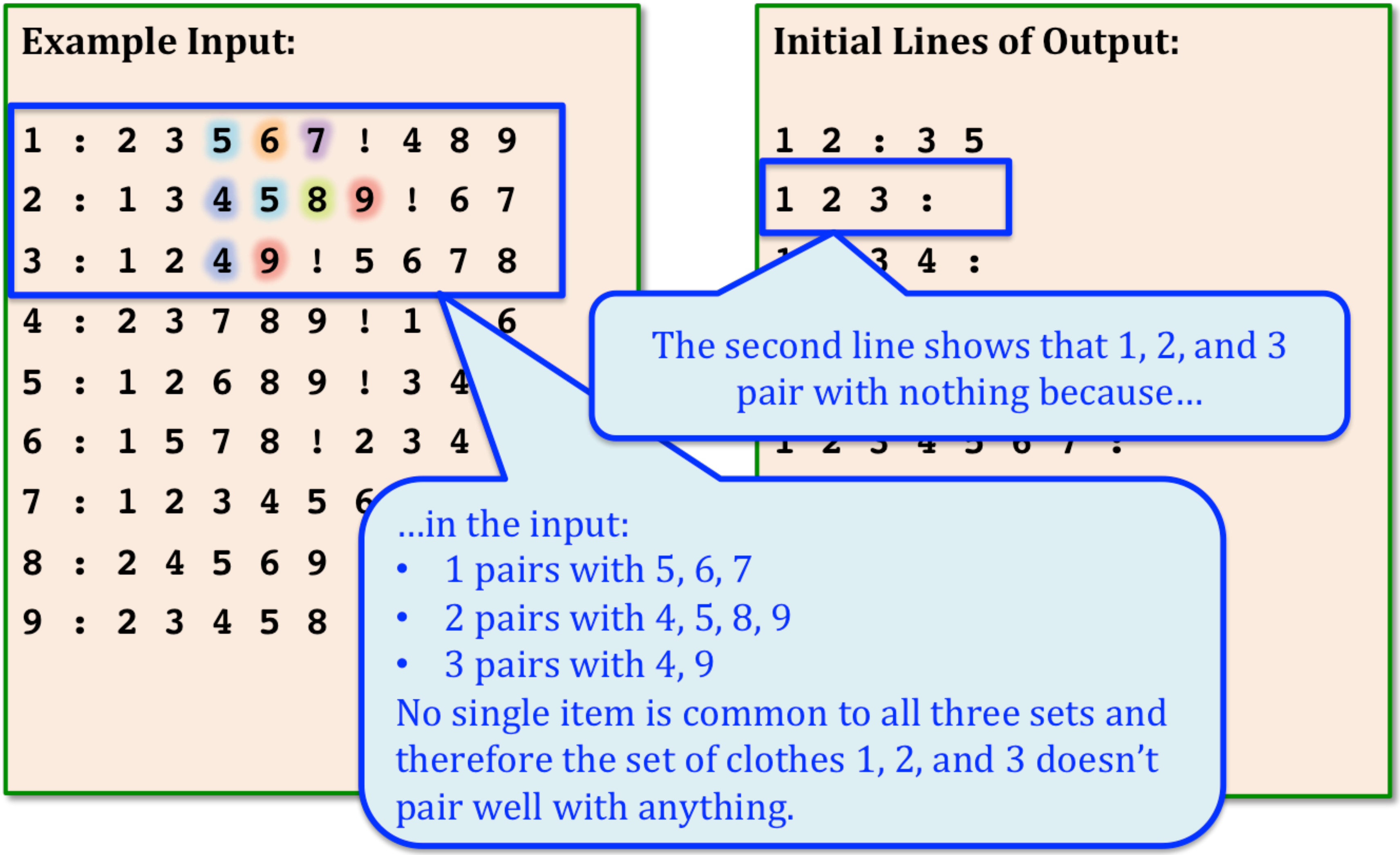
Initial Lines of Output:

```
1 2 : 3 5
1 2 3 :
1 2 3 4 :
1 2 3 4 5 :
1 2 3 4 5 6 :
1 2 3 4 5 6 7 :
1 2 3 4 5 6 7 8 :
1 2 3 4 5 6 7 8 9 :
1 2 3 4 5 6 7 9 :
1 2 3 4 5 6 8 :
...
```

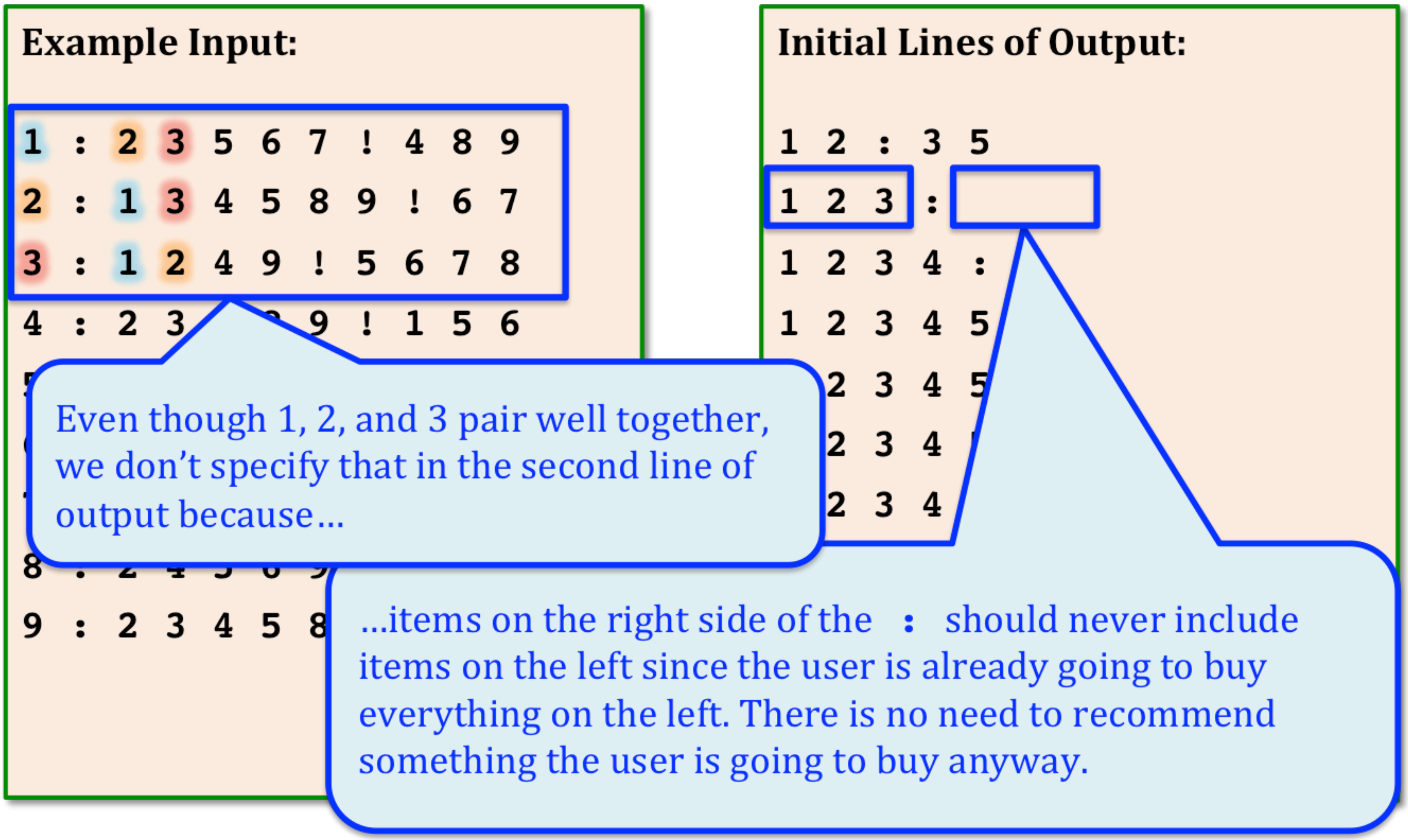
Consider the first output line:



Now examine the second output line:



Notice that we don't have to specify items the shopper is going to buy together anyway:



Since the full output is quite long, I will not be posting it here. Below, I've posted the next several output lines where there is actually a pairing:

Example Input:

```
1 : 2 3 5 6 7 ! 4 8 9
2 : 1 3 4 5 8 9 ! 6 7
3 : 1 2 4 9 ! 5 6 7 8
4 : 2 3 7 8 9 ! 1 5 6
5 : 1 2 6 8 9 ! 3 4 7
6 : 1 5 7 8 ! 2 3 4 9
7 : 1 2 3 4 5 6 ! 8 9
8 : 2 4 5 6 9 ! 1 3 7
9 : 2 3 4 5 8 ! 1 6 7
```

Selected Output of Pairings:

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

Despite the fact that my example shows the output sorted, *your output may appear in any order*. In other words, **1 2 3** does not have to appear before **1 4**. However, *the numbers in each row should all be in sorted order*.

Complete Example

Input:

```
1 : 2 3 5 ! 4
2 : 1 3 4 5 !
3 : 1 2 4 ! 5
4 : 2 3 ! 1 5
5 : 1 2 ! 3 4
```

Output:

```
1 2 : 3 5
1 2 3 :
1 2 3 4 :
1 2 3 4 5 :
1 2 3 5 :
1 2 4 : 3
1 2 4 5 :
1 2 5 :
1 3 : 2
1 3 4 : 2
1 3 4 5 : 2
1 3 5 : 2
1 4 : 2 3
1 4 5 : 2
1 5 : 2
2 3 : 1 4
2 3 4 :
2 3 4 5 :
2 3 5 : 1
2 4 : 3
2 4 5 :
2 5 : 1
3 4 : 2
3 4 5 : 2
3 5 : 1 2
4 5 : 2
```

You should print your output to the command line. The left hand side should be a list of the clothing IDs separated by a space. This is followed by a colon. The right hand side contains a sorted list of clothing IDs that pair well with everything on the left hand side. These are also separated by a space.

Submitting Your Work

When you are finished, submit *a .zip file* to the **Project 3 dropbox** that includes:

- Your code