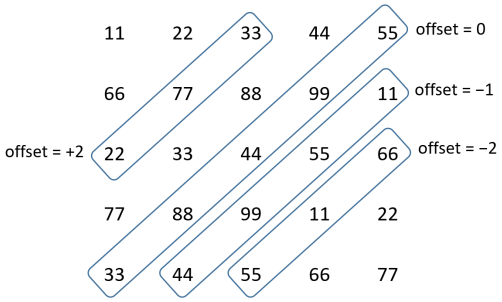
NAME _____ **ICA-4**

NetID _____ @email.arizona.edu

Total time: 50 minutes

Problem 1

Write a function **`sum_diag_UR_LL`**`(grid, offset)` that takes as arguments a grid of numbers and an offset and returns the result of summing the numbers on a specified diagonal of `grid`. This function considers diagonals running from the upper-right of the grid to the lower-left (hence the 'UR_LL' in the function name). The offset is used to select which diagonal to sum, as shown on the right. The grid is represented as a list of lists, as in Problem 1. Your code can assume that the input is in fact a grid, but should be able to handle grids of any size.



Problem 2

Assume that `slist` is a list of strings. Create a dictionary `d` where the strings in `slist` are the keys and the corresponding values are 0, if the string is unique in `slist`, and 1 otherwise. For example, if `slist` is the following,

```
[ "this", "and", "that", "and", "that", "is", "it"]
```

then the dictionary would be

```
{'this': 0, 'and': 1, 'that': 1, 'is': 0, 'it': 0}
```

Problem 3

Consider a file whose contents are a series of comma separated values that represent a student. The first value is the student name and the remaining values are pairs of values representing the classes and grades of the student, separated by colons. A sample input file, `students.txt`, is shown below:

```
#name,class1:grade1,class2:grade2,…, classn:graden
sue,csc110:A,csc120:A,csc210:B
javier,ece175:B,csc120:A
```

Write a function `process_input(fname)` that takes an argument `fname` which is the name of a file containing comma separated values as described above. The function reads the file and creates a dictionary. The keys of the dictionary are the student names. The corresponding value of each key is a list of lists, where each list is a class name and grade.

For the sample input file above, `student.txt`, the call `process_input("student.txt")` would return the following dictionary:

```
{ 'sue':    [['csc110', 'A'], ['csc120', 'A'], ['csc210', 'B']],
  'javier': [['ece175', 'B'], ['csc120', 'A']]
}
```

Assume that the file exists, has only one header line starting with #, and that the file contains all expected values. (That is, no error checking is needed.) Do NOT import the csv library.

Problem 4

a) Suppost that you have a list of lists, where each inner list represents a triangle. Each inner list consists of a name of a rectangle followed by its width and height. A sample list is shown below:

```
[['u1', 3, 5], ['rec1', 80, 20], ['rec2', 7, 2], ['z', 10, 3]]
```

Write a function `create_rdict(rlist)` that takes a list of lists as described above and creates a dictionary where each key is the name of a rectangle and the corresponding value is a list of the rectangle's width, height, and area. For the list of lists above, `create_rdict(rlist)` would return the dictionary below:

```
{ 'u1': [3, 5, 15], 'rec1': [80, 20, 1600],
  'rec2': [7, 2, 14], 'z': [10, 3, 30] }
```

b) Write a function `print_greatest_area(rdict)` that takes a dictionary of rectangles as described above in (a) and prints the rectangle with the largest area. Using the dictionary described in (a) above, the function would print the following:

```
rec1: [80, 20, 1600]
```

If there are ties, print *all* of the rectangles with ties in any order, one per line.

Problem 5

In the English language, some combinations of adjacent letters are more common than others. For example, 'h' often follows 't' (as in 'th'), but rarely does 'x' follow 't' (as in 'tx'). Knowing how often a given letter follows other letters is useful in many contexts.[1]

For this problem, you will write a function name `pair_frequencies(word_list)` where `word_list` is a list of strings representing valid English words. Your function will examine `word_list` and print all 2-character sequences of letters along with a count of how many times each pairing occurs. (The function does not return anything.) For example, suppose `word_list` has the following contents:

```
["banana", "bends", "i", "mend", "sandy"]
```

This list of words contains the following 2-character pairs: "ba", "an", "na", "an", "na" from banana; "be", "en", "nd", "ds" from bends; "me", "en", "nd" from mend; and "sa", "an", "nd", "dy" from sandy. (Note that "i" is only one character long, so it contains no pairs.) The call of `pair_frequencies(words_list)` would print the following output:

```
ba : 1
an : 3
na : 2
be : 1
en : 2
nd : 3
ds : 1
me : 1
sa : 1
dy : 1
```

Notice that any pairings that occur more than once in the same word should be counted as separate occurrences. For example, "an" and "na" each occur twice in "banana".[2]

[1] In cryptography, we use this data to crack substitution ciphers (codes where each letter has been replaced by a different letter, for example: A -> M, B->T, etc.) by identifying which possible decoding substitutions produce plausible letter combinations and which produce nonsense.

[2] Adapted from CodeStepbyStep.