Assignment Guidelines:

- This assignment covers material in Module 9.
- Do not use recursion. All repetition must be performed using iteration (while and for loops only) and abstract list functions (map and filter).
- Submission details:
  - Solutions to these questions must be placed in files a08q1.py, a08q2.py, a08q3.py, and a08q4.py, respectively, and must be completed using Python 3.
  - Download the interface file from the course Web page to ensure that all function names are spelled correctly and each function has the correct number and order of parameters.
  - All solutions must be submitted to MarkUs. No solutions will be accepted through email, even if you are having issues with MarkUs.
  - Verify using MarkUs and your basic test results that your files were properly submitted and are readable on MarkUs.
  - For full style marks, your program must follow the Python section of the CS116 Style Guide.
  - Helper functions need design recipe elements but not examples and tests.
- Download the testing module from the course web page. Include import check in each solution file.
  - When a function produces a floating point value, you must use check.within for your testing. Unless told otherwise, you may use a tolerance of 0.00001 in your tests.
  - Test data for all questions will always meet the stated assumptions for consumed values.
- Restrictions:
  - Do not import any modules other than math and check.
  - Do not use Python constructs from later modules. Do not use any other Python functions not discussed in class or explicitly allowed elsewhere. See the allowable functions post on Piazza. You are always allowed to define your own helper functions, as long as they meet the assignment restrictions.
  - While you may use global constants in your solutions, do not use global variables for anything other than testing.
  - Read each question carefully for additional restrictions.
  - Do not mutate any passed parameters unless otherwise told.
  - The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.
1 Dictionary Inversion

Write a function `invert_dictionary` that consumes a dictionary \( d \) of the form \((\text{dictof Str (listof Int)})\) and returns an inverted dictionary of the form \((\text{dictof Int (listof Str)})\) in the following way. Each integer in \( d \)'s values' lists will become a key value in the new dictionary. The dictionary values associated with these integers are a list of all of the string keys in \( d \) whose values' lists contains the integer. These strings should be sorted in alphabetical order in the final lists. Do not mutate any passed parameters. You may assume that each list of integers does not contain a duplicate element.

Sample:

```
invert_dictionary({"a": [1,2,3], "b": [1,2], "z": [5]}) =>
{1: ["a", "b"], 3: ["a"], 5: ["z"], 2: ["a", "b"]}
```

Remember as always, the order of the key-value pairs of the returned dictionary is irrelevant.

2 Shopping Time!

Consider the following data definitions:

A Store is a \((\text{list Str (dictof Str Float)})\) consisting of a store's name and a dictionary mapping the names of the items they sell to the prices of those items.

requires: Each key is a string and the associated values are positive Floats.

A Shopping_List is a \((\text{listof Str})\)

A Shopping_Trip is a \((\text{dictof Str (listof Str)})\) where the keys are the names of stores and each key is mapped to a list of the names of the items to be bought at that store.

Given a Shopping_List, determine a Shopping_Trip that minimizes the cost of purchasing all the items in the Shopping_List at the given list of Stores. Concretely, you will complete the function `optimize_shopping(stores, goods)` which consumes stores, which is of type \((\text{listof Store})\), and goods, which is of type Shopping_List, and returns a minimum cost Shopping_Trip. Your function should also print the total cost of the trip (using the string constant provided in the interface file). Return an empty dictionary if there is an empty shopping list given and print out 0 as a float. Do not mutate any passed parameters. You may assume that each element of goods occurs in at least one store, that identical items sold at different stores have different costs and that goods does not have any repeated elements.

Sample:

```
stores = [
    ["Arnie's",{"apples": 2.32, "bananas": 0.77, "chickens": 4.50}],
    ["Darlene's",{"apples": 1.50, "bananas": 0.20, "chickens": 9.50}],
    ["Eddie's",{"grapefruits": 3.31, "bananas": 0.97, "beans": 2.50}]

shop_list = ["apples", "bananas", "grapefruits"]

optimize_shopping(stores, shop_list) =>
    {"Darlene's":["apples", "bananas"], "Eddie's": ["grapefruits"]}
```

and the following is printed to the screen using the given string:

Final total: $5.01.

Each store's list of items in the final returned dictionary should be in alphabetical order. Remember as always, the order of the key-value pairs of the returned dictionary is irrelevant.
3 Stocks Part 1

For the next two problems, you will be considering a stock class which you will create and modify and consider a problem dealing with share holders.

Some brief background. A stock is a part of a company that the public can purchase and own. A stock has a name and list of prices consisting of both past and present values stored in order from inception until present day. A trader is someone who can purchase stocks and sell stocks.

Complete the Stock class in the provided interface by adding the following methods:

1. update_prices which consumes a list of floating point numbers, returns nothing and adds this list at the end of the current list of prices in the stock. Do not mutate any passed parameters.

2. average which consumes a natural number n and returns the average stock price of the last n entries in the stock’s prices field. You may assume that the testing data for a Stock s will always be such that n <= len(s.prices). The average should be of type Float.

Also, complete the Trader class in the provided interface by adding the following methods:

1. buy_stocks which consumes a Stock s and a natural number n and then buys n shares of stock s (if the Trader has enough money to do so!) and returns True if successful and False otherwise. You should use the most recent value of the prices field, that is, the last element, to determining the purchasing price and update the Trader’s cash and stocks owned once the transaction has occurred. Note you may buy more of stock you already own and you either buy the requested number or none at all.

2. sell_stocks which consumes a Stock s and a natural number n and then sells n shares of stock s (if the Trader owns enough stock to sell!) and returns True if successful and False otherwise. You should use the most recent value of the prices field, that is, the last element, to determining the selling price. You should remove a stock from the dictionary in the stocks Owned field if you no longer hold any shares and you should not sell stocks unless you can sell exactly n stocks.

A reminder to test your class methods outside of the class itself. Use the samples below to guide your testing.

Sample. Consider the following commands:

apple = Stock("APPL", 3.0)
apple.update_prices([3.01, 3.5, 3.42, 3.64, 3.41, 4.2, 4.3, 5.21,
        5.13, 4.66, 5.21, 5.31, 5.5, 5.57, 4.56 ])
warren_buffet = Trader("Warren Buffet", {}, 1000000000.0)

Combining the above commands with those commands on the left below applied in succession, we get the results on the right:
print(apple)  
Stock: APPL  
Current Value: 4.56

print(apple.prices)  
[3.0, 3.01, 3.5, 3.42, 3.64, 3.41, 4.2, 4.3, 5.21, 5.13, 4.66, 5.21, 5.31, 5.5, 5.57, 4.56]

print("Average:", apple.average(3))  
Average: 5.21

warren_buffet.buy_stocks(apple, 100)  
Trader: Warren Buffet  
Stocks: ['APPL Shares: 100']  
Current Cash Value: 999999544.0

print(warren_buffet)  
Trader: Warren Buffet  
Stocks: ['APPL Shares: 10']  
Current Cash Value: 999999954.4

warren_buffet.sell_stocks(apple, 90)  
Trader: Warren Buffet  
Stocks: ['APPL Shares: 10']  
Current Cash Value: 999999954.4

print(warren_buffet)  
Trader: Warren Buffet  
Stocks: []  
Current Cash Value: 1000000014.4

apple.update_prices([5.8, 6.00])  
warren_buffet.sell_stocks(apple, 10)  
print(warren_buffet)  
Trader: Warren Buffet  
Stocks: []  
Current Cash Value: 1000000014.4

4 Stocks Part 2

Companies only make a finite number of shares available. Many companies will have an annual shareholder meeting consisting of all people who own portions of the company. At this meeting, shareholders gather to make decisions. We say that a collection of Traders has a majority if and only if they own strictly more than half of all available shares in a company. Such a collection would be able to make decisions for the company independent of what other people vote on.

Write a function

minimal_majority(company, traders)

which consumes company, which is of type (list Stock Int) where the second parameter consists of an integer of the number of available shares the company has and a list of Traders. The function then returns a list of trader names in alphabetical order that comprise a minimum number of Traders that could form a majority, that is, there should be no smaller collection of traders that form a majority and this collection of Traders should own strictly more than half of all available shares. You may assume that such a list of people is always unique if it exists. You should return an empty list if no such collection exists. Do not mutate any passed parameters. Hint: Including traders that own more stock first into your collection ensures you have a minimum such collection of these traders who can form a majority.

Sample:

apple = Stock("APPL",3.0)  
microsoft = Stock("MSFT",2.75)  
warren_buffet = Trader("Warren Buffet", {apple.symbol: 30, microsoft.  
    symbol: 100}, 1000000000.0)  
tim_cook = Trader("Tim Cook", {apple.symbol: 45}, 1000000000.0)  
paul_tudor_jones = Trader("Paul Tudor Jones", {apple.symbol: 20},  
    1000000000.0)  
minimal_majority([apple, 140],[warren_buffet, tim_cook,  
     paul_tudor_jones]) => ["Tim Cook", "Warren Buffet"]