Due: Wednesday Oct. 18th, 2017 at 10 am

Assignment Guidelines

- This assignment covers material in Module 04.
- Submission details:
  - Solutions to these questions must be placed in files `a04q1.py`, `a04q2.py`, and `a04q3.py`.
  - You must be using Python 3 or higher.
  - 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.
  - Be sure to review the Academic Integrity policy on the Assignments page
- Download the testing module from the course web page. Include `import check` in each solution file.
- Restrictions:
  - Do not import any modules other than `math` and `check`.
  - 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 or tips.
  - Tip: for this assignment, repetition can be implemented using recursion as loops are not allowed

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.

Fun with lists….

1. For all the parts of this question you must not sort nor reverse a list
   You must not use loops nor abstract list functions. You may use recursion though.
   All five parts should be submitted in one file and should include design recipe for each part as well.

   Part a) Write a Python function `create_odds` that consumes a natural number (`target`) and returns a list of all positive odd integers that are <= `target` in ascending order.
   For example:
   ```
   create_odds(8) => [1, 3, 5, 7]
   create_odds(0) => []
   ```

   Part b) Write a Python function `build_special_list` that consumes a natural number (`n`) and returns a list following the pattern
   `[[1], [1,2], … , [1,2,3,…,n]]`.
   For example:
   ```
   build_special_list(6) => [[1], [1, 2], [1, 2, 3], [1, 2, 3, 4], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5, 6]]
   build_special_list(0) => []
   build_special_list(1) => [[1]]
   ```

   Part c) Write a Python function `divisibles` that consumes a natural number `n` and returns a list of all divisibles of `n` (that <u>are less</u> than `n`) in ascending order.
   For example:
   ```
   divisibles(16) => [1, 2, 4, 8]
   divisibles(0) => []
   divisibles(1) => []
   divisibles(19) => [1]
   ```

   Part d) Write a Python function `update_list` that consumes a list of integers `nlst`, and two integers `val` and `newval`, and mutates `nlst` by changing all the occurrences of `val` in `nlst` to `newval`. The function also returns how many times the change occurred. You may assume that `val` and `newval` are different.
   For example:
   if `nl` is `[]` then `update_list(nl, 5, 10) => 0` and `nl` is unchanged
   if `nl` is `[3, 10, 5, 10, -4]` then `update_list(nl, 10, 7) => 2`
   and `nl` is mutated to `[3,7, 5, 7, -4]`

Part e) Write a Python function `mult_list` consumes two lists of integers called `m1` and `m2.` The function mutates `m1`, by multiplying each entry in `m2` to the corresponding entry in `m1.` Assume lists are of the same length. The function returns `None.`

Suppose `m1=[1, 2, 23, 104]` and `m2=[-3, 2, 0, 6],` calling `mult_list (m1,m2)` mutates `m1 to [-3, 4, 0, 624]` and returns `None.`

Q2+Q3 should be solved without using explicit recursion (which means you can't define recursive functions), however you should use abstract list functions (yes they include implicit recursion, because their behavior or implementation is recursive, but that is Okay)

To make it simple: you are not allowed to define a function that calls itself, or to define a function (f) that calls (g) and (g) calls (f) back

Let's tweet….

Note: You must not use explicit recursion for this question, nor loops. You may use abstract list functions though.

2. In Twitter, it is possible to search through tweets for specific tweeter. For the purpose of this question, we will represent a Twitter tweet (containing the tweet number, the tweeter name, and the body) as a single string with the following format: "#N:-@name:-Body". (You may assume that ":-" appears only twice in a tweet).
   For example, `"#1:-@DanClark:-The party was amazing"` corresponds to tweet #1, the tweeter name is `"DanClark"`, and the body is `"The party was amazing"`.
   You may assume that N>0, the tweeter and the body are non-empty.
   Write a Python function `search_tweets`, that consumes `tweets`, a list of strings in the format described above, and `tweeter,` a nonempty string, and returns a list of `Nat`, containing the tweet numbers of every string in `tweets` which the sender name is `tweeter`.

For example,

suppose `tweets` contains:

```
["#1:-@DanClark:-The party was amazing",
"#19:-@NatalyS:-Avoid 401 Toronto area at this time",
"#50:-@CBCNews:-How Canadian captain gave her team a
speech",
"#14:-@DanClark:-The food was good",
"#15:-@DaveLin:-Lucky you DanClark"]
```

then

```
search_tweets(tweets, "DanClark")  =>  [1, 14]
```

Note: The returned list of tweet numbers should be in the same order as the consumed `tweets`.

For old days' sake….

Note: You must not use explicit recursion for this question, nor loops. You may use abstract list functions though.

3. This question using the following data definition:

```
## A keycode is a list of length 2, [d, n] where
##   d is an Int from 0 - 9 representing a digit on a phone
##   keypad and
##   n is a positive Int, representing the number of times
##   the key has been pressed. The value of n will be less
##   than or equal to the number of symbols associated with
##   d on a phone.
```

On an inexpensive mobile phone, you can compose a text message using the keypad on the phone. Each digit is associated with a set of characters according to the following chart:

Digit on Phone	Associated Characters
0	space
1	.,?
2	abc
3	def
4	ghi
5	jkl
6	mno
7	pqrs
8	tuv
9	wxyz

Write a function called `compose_msg` that consumes `keypresses`, a list of `keycode` values for a phone described earlier, and returns a string that represents a text message. All of the letters in the message should appear in lower case.
For example:
`compose_msg([[6, 3], [0, 1], [5, 2]])` returns the string `"o k"`.

For an empty consumed list, the function should return an empty string.
Once again, remember that the second element of a `keycode` value is always a number less than or equal to the number of characters associated with its keypad digit.