National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2022/2023

Tutorial 7
List Processing & Data Structures

1. The function **`accumulate_n(op, init, sequences)`** is similar to **`accumulate(op, init, seq)`** except that it takes as its third argument a *sequence of* ***sequences of equal length***. It applies the accumulation function **`op`** to combine all the first elements of the sequences, all the second elements of the sequences, and so on, and returns a sequence of the results. For instance, if **`s`** is a sequence containing four sequences, **`[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]`**, then the value of **`accumulate_n(lambda x,y:x+y, 0, s)`** should be the sequence **`[22, 26, 30]`**. Fill in the missing expressions `<T1>` and `<T2>` in the following definition of **`accumulate_n`**:

```
def accumulate(op, init, seq):
    if not seq:
        return init
    else:
        return op(seq[0], accumulate(op, init, seq[1:]))

def accumulate_n(op, init, sequences):
    if (not sequences) or (not sequences[0]):
        return type(sequences)()
    else:
        return ( [accumulate(op, init, <T1>)]
               + accumulate_n(op, init, <T2>) )
```

Note: `accumulate_n` uses both `accumulate` and `accumulate_n`.

2. Recall that in Tutorial 6, a matrix can be represented in Python by a list of lists (nested lists). For example, `m = [[ 1, 2, 3], [4, 5, 6], [7, 8, 9]]` represents the following 3×3 matrix:

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

   (a) Re-define the function **`col_sum`** using **`accumulate_n`**. The function should still take in a matrix and return a list, where the ***i***-th element in the list is the sum of the elements in the ***i***-th column of the matrix, .

   (b) Re-define the function **`row_sum`** using **`accumulate_n`**. The function should still take in a matrix and return a list, where the ***i***-th element is the sum of the elements in the ***i***-th row of the matrix, .

Hint: `transpose` may come in handy here.

3. Ignoring punctuation, an English sentence is a collection of words, and each word is a collection of letters. Suppose we would like to (for the fun of it) create a list representation of an English sentence. We could represent each word as a list of letters (for example, the word 'cat' would be represented as **`['c', 'a', 't']`**). A sentence would then be a list of such word representations. Some examples :

```
"CS1010S Rocks"
[['C', 'S', '1', '0', '1', '0', 'S'], ['R', 'o', 'c', 'k', 's']]

"Python is cool"
[['P', 'y', 't', 'h', 'o', 'n'], ['i', 's'], ['c', 'o', 'o', 'l']]
```

   a) Write a function **`count_sentence`** that takes a sentence representation (as described above) and returns a series list with two elements: the number of words in the sentence, and the number of letters in the sentence.

   Assume that spaces count as 1 letter per space, and that there is exactly 1 space between each word (but none at the start or end of the sentence).

   What is the order of growth in time and space (in terms of the number of letters in the sentence) of the function that you wrote?

   Example execution:

```
>>> count_sentence([['I'], ['l', 'i', 'k', 'e'], ['p', 'i', 'e']])
[3, 10]
```

   b) Write a function **`letter_count`** that takes a sentence and returns a list of lists, where you have one list for each distinct letter in the sentence and each list has two elements.

   The first element of the list pair is the letter and the second element is the count for the letter. The order of the list pairs does not matter.

   What is the order of growth in time and space (in terms of the number of letters in the sentence) of the function that you wrote?

   Example execution:

```
>>> letter_count([['s', 'h', 'e'], ['l', 'i', 'k', 'e', 's'],
                  ['p', 'i', 'e', 's']])
[['s', 3], ['h', 1], ['e', 3], ['l', 1], ['i', 2], ['k', 1], ['p', 1]]
```

   c) Write a function **`most_frequent_letters`** that takes a sentence representation and returns a list of letters that occur most frequently in the given sentence.

   The order of the letters does not matter. If only one such letter exists, then return a list with one element. If the sentence is empty, return an empty list.

   What is the order of growth in time and space (in terms of the number of letters in the sentence) of the function that you wrote?

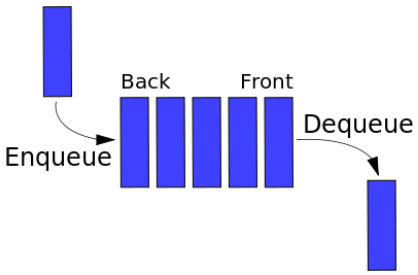
   Example execution:

```
>>> most_frequent_letters([['s', 'h', 'e'], ['l', 'i', 'k', 'e', 's'],
                           ['p', 'i', 'e', 's']])
['e', 's']
```

4. Recall that in lecture, you are taught that **stack** is a Last-In-First-Out (LIFO) data structure. There is another well known First-In-First-Out (FIFO) data structure that is called a **queue**.

   In this question, you will implement a **queue** data structure. In a queue, items are removed in the same order as they are added. Refer to the diagram below for an illustration of this. We actually discussed queues in Recitation 5, but because we were then working without mutable data structures, each queue operation returned a **new** queue object.

   In this example, you will implement a **mutable** queue, i.e. the operators will directly modify the specified queue object instead of returning a new object like in Recitation 5.

   For this question, you are required to define the following functions:

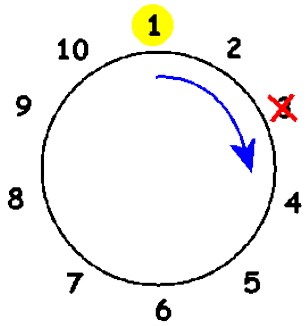
   ```
   make_queue(): returns a new empty queue
   enqueue(q, item): adds item to queue q
   dequeue(q): removes the first item from the queue q, and returns it
   size(q): returns the size of the queue q
   ```

   For example:

   ```
   >>> q = make_queue()
   >>> enqueue(q, 1)
   >>> enqueue(q, 5)
   >>> size(q)
   2
   >>> dequeue(q)
   1
   ```

5. Suppose now you are invited to play a game called Pass-the-Bomb. In this game, a group of `n` players will form a circle and the first player will be given a bomb. At each turn, the player with the bomb will pass it to the next player in the clockwise direction. After `m` turns, the bomb will explode and the player will be removed from the game. The next player in line will be given another bomb and the game will continue until only one player is left.

   The simulation below shows the second iteration of a game with `n=10` players and a `m=2`-time bomb. The bomb exploded at the 3rd player so play restarts with the bomb given to 4th player. The bomb will then explode at the 6th player.



Write a function **`who_wins`** that will take an integer `m` and a list of `players` and return the last **`m-1`** players in the game. For example:

```
>>> who_wins(3, ['val', 'hel', 'jam', 'jin', 'tze', 'eli', 'zha', 'lic'])
['eli','zha']
>>> who_wins(2, ['poo', 'ste', 'sim', 'nic', 'luo', 'ibr', 'sie', 'zhu'])
['sie']
```

Hint: You can use a queue to help you solve this question.