CS 116 TUTORIAL 5

ACCUMULATIVE AND GENERATIVE RECURSION
REMINDER

• Assignment 05 due next Wednesday, June 14th (at 10:00AM)
RECURSION

- Accumulative recursion
- Generative recursion
def acc_fn(remaining, acc):
    if (base_case of remaining):
        ...
        return acc
    if/elif ...
    else:
        ...
        return acc_fn(updated_remaining, updated_acc)

def fn(lst):
    return acc_fn(initial_remaining, initial_acc)

• Accumulators “keep track” of something so that you can quickly produce the expected result
• You may need more than one accumulator.
I. Develop an accumulatively recursive function `list_to_num` that consumes a **nonempty list**, `digits`, of integers between 0 and 9, and returns the number corresponding to `digits`.

For example,

- `list_to_num([9, 0, 8])` => 908
- `list_to_num([8, 6])` => 86
- `list_to_num([0, 6, 0])` => 60
2. Write an accumulatively recursive function `count_max` that consumes a nonempty list of integers `alon` and returns the number of times the largest integer in `alon` appears.

**Note:** `max` is not allowed to use in this question.

For example,

```
count_max([1, 3, 5, 4, 2, 3, 3, 3, 5]) => 2
```

since the largest element of the list, 5, appears twice. Your function should pass through the list only once.
STRUCTURAL V.S. GENERATIVE

• **Structural Recursion:**
  – Break problems into smaller problems using the recursive definition of our data.
  – recursive subproblem is always **one step closer** to a base case
  – Use recursive template

• **Generative Recursion:**
  – We break our problems into smaller subproblems **inspired by the problem itself.**
  – No standard template

• **Accumulative Recursion:**
  – Recursion with an **accumulator**
  – *When using accumulative recursion, our accumulative helper functions can use either structural or generative recursion, depending on the technique we use.*
This is an example of generative recursion.

def recursion(lst):
    if lst == []:
        return True
    elif lst[0] == lst[-1]:
        return recursion(lst[1:-1])
    else:
        return False
3. We define a string $s$ to be “balanced” based on the following criteria:

- Any string not containing brackets (any of ( ) { }) is considered balanced.
- If $s$ is a balanced string then $(s)$, and $\{s\}$ are also balanced strings.
- If $s$ is a balanced string and $t$ is a string that does not contain any brackets, then $s+t$ is a balanced string and so is $t+s$.

  - For example, “(Wo)rd” is balanced as well as “rd{Wo}”.

Balanced strings:

- “”
- “(hi)”
- “wrong{}thing”
- “cool(brackets)”
- “{{()}}”

Unbalanced strings:

- “NO)PE”
- “{no}”

Write a function `is_balanced` that consumes a string $s$ and returns `True` if $s$ is balanced, `False` otherwise.
4. Given a list $L$ of positive integers, the skip-value of a list is the number of steps to reach the end of the list, using the values in the list:

- If $L$ is empty, the skip value is $0$.
- If $L$ is nonempty:
  - Add 1 to the current skip value.
  - Move ahead $L[0]$ places in the list, and repeat from the process with the remainder of the list from that place.

Write a function `skip_value` to calculate the skip value of the list $L$.

For example,

```python
skip_value([]) => 0
skip_value([1,1,1]) => 3
skip_value([2,100,1]) => 2
```
For example,

\[
\text{skip\_value}([1,1,1])
\Rightarrow 1+\text{skip\_value}([1,1])
\Rightarrow 1+(1+\text{skip\_value}([1]))
\Rightarrow 1+(1+(1+\text{skip\_value}([])))
\Rightarrow 1+(1+(1+0))
\Rightarrow 3
\]

\[
\text{skip\_value}([2,100,3,1,1,1,1])
\Rightarrow 1+\text{skip\_value}([3,1,1,1,1])
\Rightarrow 1+(1+\text{skip\_value}([1]))
\Rightarrow 1+(1+(1+\text{skip\_value}([])))
\Rightarrow 1+(1+(1+0))
\Rightarrow 3
\]
5. Consider a new type: `Table`. A `Table` is a `(listof (listof Int))`, which is nonempty, and in which each list corresponds to a row of a `Table`. It is assumed that each row is nonempty and each row has the same number of entries as every other row.

For example, the following are tables:

\[
t3 = [[1,2],[3,4],[5,6],[7,8]]
\]

Sum of each column is calculated as following:

Sum of column 1 of \(t3\) = 1+3+5+7 = 16

Sum of column 2 of \(t3\) = 2+4+6+8 = 20
For example, the following are tables:

\[
\begin{align*}
t0 &= \begin{bmatrix} 1 \end{bmatrix} \\
t1 &= \begin{bmatrix} 1, 2, 3 \end{bmatrix} \\
t2 &= \begin{bmatrix} 1, 2 \end{bmatrix}, \begin{bmatrix} 3, 4 \end{bmatrix}, \begin{bmatrix} 5, 6 \end{bmatrix}, \begin{bmatrix} 7, 8 \end{bmatrix}
\end{align*}
\]

Write a function `sum_columns` that consumes a Table \( t \), and produces a list containing the columns sums for \( t \).

For example,

\[
\begin{align*}
\text{sum_columns}(t0) &\Rightarrow [1] \\
\text{sum_columns}(t1) &\Rightarrow [1, 2, 3] \\
\text{sum_columns}(t2) &\Rightarrow [6] \\
\text{sum_columns}(t3) &\Rightarrow [16, 20]
\end{align*}
\]
6. Write a function `smaller` that consumes a nonempty string called `s`, containing only numeric characters, by repeatedly removing the larger of the first and last characters in `s`. If the first and the last number are the same, remove the last one.

For example, starting from "5284", compare "5" and "4", and recurse on "284", which will compare "2" and "4", and recurse on "28". Comparing "2" and "8", leads to recursing on "2", which is the answer (since it is a string of length 1).

**NOTE:** `min` is not allowed to use.

For example,

```plaintext
smaller("4325") => "2"
smaller("1") => "1"
smaller("2325") => "2"
```