CS 116 TUTORIAL

5

ACCUMULATIVE AND GENERATIVE RECURRENCE
REMINDER

- Assignment 05 due Wednesday, June 19th at 10:00AM

- Midterm is on June 24th at 7 PM
  - Midterm reference sheet will be posted on Piazza & course webpage.
RECURSION

Types:

• Structural recursion

• Accumulative recursion

• Generative recursion

We’ve been using this so far.

New ones that we’re learning.
DIFFERENCE BETWEEN THE THREE

• **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
  – Uses a *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*
def acc_fn(remaining, acc):
    if (base_case of remaining):
        return ... acc ...
    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 return the expected result
• Sometimes, you may need more than one accumulator.
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
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` cannot be used 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.
This is an example of generative recursion.

def gcd(m, n):
    if n == 0:
        return m
    else:
        return gcd(n, m%n)

Base Cases:
To end the recursion

Recursive Case:
Continues the function with a sub-problem which is closer to a base case.
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}"
- "a(bc)d(ef)g"

Write a function `is_balanced` that consumes a string $s$ and returns `True` if $s$ is balanced, `False` otherwise.
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 remaining skip value
  - Move ahead \( L[0] \) places in the list, and repeat 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,

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

\[
\text{skip\_value}([2,100,3,1,1,1]) \\
\Rightarrow 1 + \text{skip\_value}([3,1,1,1]) \\
\Rightarrow 1 + (1 + \text{skip\_value}([1])) \\
\Rightarrow 1 + (1 + (1 + \text{skip\_value}([]))) \\
\Rightarrow 1 + (1 + (1 + 0)) \\
\Rightarrow 3
\]
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 is a table:

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

**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:
\[
\begin{align*}
t_0 &= \begin{bmatrix} 1 \end{bmatrix} \\
t_1 &= \begin{bmatrix} 1, 2, 3 \end{bmatrix} \\
t_2 &= \begin{bmatrix} 1 \end{bmatrix}, \begin{bmatrix} 2 \end{bmatrix}, \begin{bmatrix} 3 \end{bmatrix} \\
t_3 &= \begin{bmatrix} 1, 2 \end{bmatrix}, \begin{bmatrix} 3, 4 \end{bmatrix}, \begin{bmatrix} 5, 6 \end{bmatrix}, \begin{bmatrix} 7, 8 \end{bmatrix}
\end{align*}
\]

Write a function `sum_columns` that consumes a Table `t`, and returns a list containing the columns sums for `t`.

For example,
\[
\begin{align*}
\text{sum_columns}(t0) &= [1] \\
\text{sum_columns}(t1) &= [1, 2, 3] \\
\text{sum_columns}(t2) &= [6] \\
\text{sum_columns}(t3) &= [16, 20]
\end{align*}
\]
Write a function `smaller` that consumes a nonempty string `s`, containing only numeric characters, and generates a new string 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:** Do not use `min`.

For example,

- `smaller("4325") => "2"`
- `smaller("1") => "1"`
- `smaller("2325") => "2"`
- `smaller("8668") => "6"`