CS 61A Summer 2026

Recursion, Tree Recursion

Discussion 3: July 2, 2026

Q1: Skip Factorial

Define the base case for the `skip_factorial` function, which returns the product of **every other positive** integer, starting with `n`.

```
def skip_factorial(n):
    """Return the product of positive integers n * (n - 2) * (n - 4) * ...

    >>> skip_factorial(5) # 5 * 3 * 1
    15
    >>> skip_factorial(8) # 8 * 6 * 4 * 2
    384
    """
    if n <= 2:
        return n
    else:
        return n * skip_factorial(n - 2)
```

If `n` is even, then the base case will be 2. If `n` is odd, then the base case will be 1. Try to write a condition that handles both possibilities.

Q2: Swipe

Implement `swipe`, which prints the digits of argument `n`, one per line, **first backward then forward**. The left-most digit is printed only once. **Do not use** `while` or `for` or `str`. (Use recursion, of course!)

```
def swipe(n):
    """Print the digits of n, one per line, first backward then forward.

    >>> swipe(283)
    3
    8
    2
    8
    3
    """
    if n < 10:
        print(n)
    else:
        print(n % 10)
        swipe(n // 10)
        print(n % 10)
```

First `print` the first line of the output, then make a recursive call, then `print` the last line of the output.

Q3: Is Prime

Implement `is_prime` that takes an integer `n` greater than 1. It returns `True` if `n` is a prime number and `False` otherwise. Try following the approach below, but implement it recursively without using a `while` (or `for`) loop.

You will need to define another "helper" function (a function that exists just to help implement this one). Does it matter whether you define it within `is_prime` or as a separate function in the global frame? Try to define it to take as few arguments as possible.

```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(9)
    False
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    def check_all(i):
        "Check whether no number from i up to n evenly divides n."
        if i == n:      # could be replaced with i > (n ** 0.5)
            return True
        elif n % i == 0:
            return False
        return check_all(i + 1)
    return check_all(2)
```

Define an inner function that checks whether some integer between `i` and `n` evenly divides `n`. Then you can call it starting with `i=2`:

```
def is_prime(n):
    def f(i):
        if i == n:
            return ____
        elif ____:
            return ____
        else:
            return f(____)
    return f(2)
```

Q4: Recursive Hailstone

Recall the `hailstone` function from Homework 1.

- First, pick a positive integer `n` as the start. If `n` is even, divide it by 2.
- If `n` is odd, multiply it by 3 and add 1. Repeat this process until `n` is 1.
- Complete this recursive version of `hailstone` that prints out the values of the sequence and returns the number of steps.

```
def hailstone(n):
    """Print out the hailstone sequence starting at n,
    and return the number of elements in the sequence.
    >>> a = hailstone(10)
    10
    5
    16
    8
    4
    2
    1
    >>> a
    7
    >>> b = hailstone(1)
    1
    >>> b
    1
    """
    print(n)
    if n % 2 == 0:
        return even(n)
    else:
        return odd(n)

def even(n):
    return 1 + hailstone(n // 2)

def odd(n):
    if n == 1:
        return 1
    else:
        return 1 + hailstone(3 * n + 1)
```

An even number is never a base case, so `even` always makes a recursive call to `hailstone` and returns one more than the length of the rest of the hailstone sequence.

An odd number might be 1 (the base case) or greater than one (the recursive case). Only the recursive case should call `hailstone`.

Q5: Maximum Subsequence

A *subsequence* of a number is a series of digits from the number (not necessarily contiguous). For example, 12345 has subsequences like 123, 234, 124, 245, etc. Your task is to find the **largest subsequence** that is under a specified length.

> **Hint:** To add a digit (`d`) to an existing number (`n`), calculate: `n * 10 + d`. For instance, to add the digit `8` to `15` to get `158`, compute `15 * 10 + 8`.

```
def max_subseq(n, t):
    """
    Return the maximum subsequence of length at most t that can be found in the given
    number n.
    For example, for n = 2012 and t = 2, we have that the subsequences are
        2
        0
        1
        2
        20
        21
        22
        01
        02
        12
    and of these, the maximum number is 22, so our answer is 22.

    >>> max_subseq(2012, 2)
    22
    >>> max_subseq(20125, 3)
    225
    >>> max_subseq(20125, 5)
    20125
    >>> max_subseq(20125, 6) # note that 20125 == 020125
    20125
    >>> max_subseq(12345, 3)
    345
    >>> max_subseq(12345, 0) # 0 is of length 0
    0
    >>> max_subseq(12345, 1)
    5
    """
    if n == 0 or t == 0:
        return 0
    with_last = max_subseq(n // 10, t - 1) * 10 + n % 10
    without_last = max_subseq(n // 10, t)
    return max(with_last, without_last)
```

Q6: Sevens

The Game of Sevens: Players in a circle count up from 1 in the clockwise direction. (The starting player says 1, the player to their left says 2, etc.) If a number is divisible by 7 or contains a 7 (or both), switch directions. Numbers must be said on the beat at 60 beats per minute. If someone says a number when it's not their turn or someone misses the beat on their turn, the game ends.

For example, 5 people would count to 20 like this:

```
Player 1 says 1
Player 2 says 2
Player 3 says 3
Player 4 says 4
Player 5 says 5
Player 1 says 6  # All the way around the circle
Player 2 says 7  # Switch to counterclockwise
Player 1 says 8
Player 5 says 9  # Back around the circle counterclockwise
Player 4 says 10
Player 3 says 11
Player 2 says 12
Player 1 says 13
Player 5 says 14 # Switch back to clockwise
Player 1 says 15
Player 2 says 16
Player 3 says 17 # Switch back to counterclockwise
Player 2 says 18
Player 1 says 19
Player 5 says 20
```

Implement `sevens` which takes a positive integer `n` and a number of players `k`. It returns which of the `k` players says `n`. You may call `has_seven`.

An effective approach to this problem is to simulate the game, stopping on turn `n`. The implementation must keep track of the following:

- the final number `n`
- the current number `i`
- the player `who` will say `i`
- and the current `direction` that determines the next player (either increasing or decreasing).

It works well to use integers to represent all of these, with `direction` switching between `1` (increase) and `-1` (decreasing).

```
def sevens(n, k):
    """Return the (clockwise) position of who says n among k players.

    >>> sevens(2, 5)
    2
    >>> sevens(6, 5)
    1
    >>> sevens(7, 5)
    2
    >>> sevens(8, 5)
    1
    >>> sevens(9, 5)
    5
    >>> sevens(18, 5)
    2
    """
    def f(i, who, direction):
        if i == n:
            return who
        if i % 7 == 0 or has_seven(i):
            direction = -direction
        who = who + direction
        if who > k:
            who = 1
        if who < 1:
            who = k
        return f(i + 1, who, direction)
    return f(1, 1, 1)

def has_seven(n):
    if n == 0:
        return False
    elif n % 10 == 7:
        return True
    else:
        return has_seven(n // 10)
```

First check if `i` is a multiple of 7 or contains a 7, and if so, switch directions. Then, add the direction to `who` and ensure that `who` has not become smaller than 1 or greater than `k`.