CS-UY 1134: Homework 1
Summer 2019

Due June 21, 11:55 PM

Programming Part (60 Points)

1. (10 points) The *harmonic numbers* are the partial sums of the *harmonic series*. Specifically, the nth harmonic number is defined as:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$

   (see `https://en.wikipedia.org/wiki/Harmonic_number` for more details on the harmonic numbers)

   (a) Write a short function `def harmonic(n)` that takes a number n and returns the nth harmonic number.

   (b) Now, write the function `def harmonicV2(n)` that does the same thing as above (returns the nth harmonic number), but using a list comprehension and the built-in `sum` method.

2. (10 points) In this problem, we will write two different algorithms for finding all the prime numbers up to a given number n.

   In each implementation, the algorithm should return a list of all the primes up to (and possibly including) n.

   (a) The first approach will use the prime testing algorithm from lecture. You should iterate through all numbers from 1 up to (and possibly including) n. For each number, test if it is prime using the method from lecture, and if it is, add it to the list of primes. At the end, return the final list.

   Use the third (and most efficient) version of the prime testing algorithm, which when testing if a number k is prime only tests factors up to $\sqrt{k}$.

   Implement this as a method called `find_primes(n)`.

   Example: Calling `find_primes(17)` should return the list `[2, 3, 5, 7, 11, 13, 17]`.

   (b) The second approach will implement the *Sieve of Eratosthenes*. You can read about the Sieve of Eratosthenes at `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`.

   The general approach you will use is as follows. Create a list (say, `prime_lst`) of size $n + 1$, consisting of all `True` values. When we are finished running the seive algorithm, `prime_lst[i]` will be `True` if i is prime, and `False` otherwise.

   The seive will work roughly as follows. Start at index $i = 2$. Then, set `prime_lst[2*i]`, `prime_lst[3*i]`, `prime_lst[4*i]`, etc., to `False`. Then increment i until `prime_lst[i]` is `True`, and do the same thing again: set `prime_lst[2*i]`, `prime_lst[3*i]`, etc., to `False`. Continue until i reaches the end of the list.

   (If this is a bit confusing to you, I recommend reading the Wikipedia page on the Seive of Eratosthenes.)

Once this is done, you will have a list `prime_lst` for which `prime_lst[i]` is `True` if and only if i is prime. You should then convert this into a single list consisting of the prime numbers. (Try using a list comprehension for this!)

As an example, when calling `prime_seive(17)`, after running the seive portion, `prime_lst` will be `True` for the indices 2, 3, 5, 7, 11, 13, and 17, and `False` for all other indices. The function should then construct and return the list `[2, 3, 5, 7, 11, 13, 17]`.

Implement this as: `def prime_sieve(n)`.

3. (10 points) The *Leonardo Numbers* are a sequence of numbers similar to the Fibonacci numbers. They are defined as follows:

$$L_n = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ L_{n-1} + L_{n-2} + 1 & \text{if } n > 1 \end{cases}$$

So, $L_0 = L_1 = 1$, and $L_n = L_{n-1} + L_{n-2} + 1$ for $n > 1$.

Write the function: `def leonardo(n)`, which creates a *generator*, that when iterated over will yield the first n numbers in the Leonardo sequence (that is, the numbers L_0 up to and including L_{n-1}). For example, the following code:

```
for val in leonardo(7):
    print(val, end=' ')
```

should print: `1 1 3 5 9 15 25`. Note again that this should be a *generator*: do not simply return a list!

4. (20 points) In this problem, we will modify the dynamic array implementation `MyList` from class.

   (a) Implement the `__repr__` to return a string representation of the list in the same format as Python's built-in list type. For example, if `mylst` is a `MyList` object containing the elements 1,2,3, then calling `str(mylst)` should return `"[1, 2, 3]"`.

   (b) Implement the `__add__` method, so that `mylst1 + mylst2` will return a new `MyList` object representing the concatenation of the two lists. For example, if `mylst1` contains `[1, 2, 3]` and `mylst2` contains `[4, 5, 6]`, then `mylst1 + mylst2` should return a new `MyList` object containing `[1, 2, 3, 4, 5, 6]`.

   Note: Your implementation should run in *linear time.* This means that if `mylst1` has n_1 elements and `mylst2` has n_2 elements, computing `mylst1 + mylst2` should run in time $\Theta(n_1 + n_2)$.

   (c) Implement the `__mul__` method so that expressions of the form `mylst * n`, where `n` is a positive integer, will return a new `MyList` object that contains n copies of the elements of `mylst` (this is the same behavior as Python's built-in list type).

   Note: Your implementation should run in *linear time*, proportional to the length of the new list. This means if `mylst` has m elements, `mylst * n` should run in time $\Theta(m \cdot n)$.

   (d) After implementing the `__mul__` method, you should notice that syntax like `mylst * 7` works, returning a new `MyList` object. However, writing `7 * mylst` won't work. Implement the `__rmul__` method so that the syntax `7 * mylst` works — it should do the same thing as `mylst * 7`. See Section 2.3.2 in the textbook for more details about `__mul__` and `__rmul__`, and why implementing `__rmul__` is necessary to allow writing `7 * mylst`.

5. (10 points) The *greatest common divisor* (gcd) of two integers a and b is the largest integer that evenly divides both a and b.

Euclid's Algorithm is a way of finding the greatest common divisor, using the following property: The greatest common divisor of a and b is the same as the greatest common divisor of b and $a \,\%\, b$; in symbols, $\gcd(a, b) = \gcd(b, a \,\%\, b)$.

Note that the greatest common divisor of any number, a, and 0 is a itself. In symbols: $\gcd(a, 0) = a$.

Write a **recursive** function: `def gcd(a, b)`, which returns the *greatest common divisor* of two numbers a and b using *Euclid's algorithm* as described above. Note that while it is possible to implement Euclid's algorithm without recursion, in order to get more practice with recursion, you should implement the algorithm using recursion. To be clear: You **must** use recursion for full credit.

Written Part (40 Points)

1. (10 points) For each of the following code snippets, give the worst-case running time in asymptotic (Big-Oh or Big-Theta) notation. Make sure your answer is *tight*: For example, if a code snippet runs in time $\Theta(n^4)$, saying it runs in time $O(n^5)$ is technically correct, but it is not *tight*; you would need to answer $\Theta(n^4)$ or $O(n^4)$ for full credit.

   (a)
   ```
   total = 0
   for i in range(n):
       total += i
   for j in range(1,n+1):
       total += j
   ```

   (b)
   ```
   total = 0
   for i in range(n):
       total += i
       for j in range(i,n):
           total += j
   ```

   (c)
   ```
   total = 0
   while n > 1:
       total += n % 2
       n = n // 2
   ```

2. (10 points) Use the definitions of Big-Oh (O) and Big-Theta (Θ) to show the following:

   (a) $3n^4 + 8n^3 - 3n = \Theta(n^4)$

   (b) $\sqrt{17n^2 + 4n - 7} = \Theta(n)$

   (c) Let $f(n)$, $g(n)$, and $h(n)$ be functions. Show that if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

3. (10 points) In problem 2(a) of the programming part, you implemented a method `find_primes(n)` for finding all primes up to n by using the prime testing algorithm from class (the third version that tests if k is prime by checking for factors up to $\sqrt{k}$).

   Analyze the running time of `find_primes` in asymptotic (Big-Oh) notation. If necessary, you may use the fact that $\sum_{k=1}^{n} \sqrt{k} = \sqrt{1} + \sqrt{2} + \cdots + \sqrt{n} = \Theta(n\sqrt{n})$. Briefly explain your answer.

4. (10 points) Consider the following two approaches for reversing a list:

```
def reverse1(lst):
    rev_lst = []
    for i in range(len(lst)):
        rev_lst.insert(0, lst[i])
    return rev_lst
```

```
def reverse2(lst):
    rev_lst = []
    for i in range(len(lst)-1, -1, -1):
        rev_lst.append(lst[i])
    return rev_lst
```

1. If `lst` is a list of n integers, what is the worst case running time of `reverse1(lst)`? Explain your answer.

2. If `lst` is a list of n integers, what is the worst case running time of `reverse2(lst)`? Explain your answer.