**Assignment Guidelines:**

- This assignment covers material up to Module 5.
- Submission details:
  - Solutions to these questions must be placed in files `a06q1.py`, `a06q2.py`, `a06q3.py`, and `a06q4.py`, respectively, and must be completed using Python 3.
  - Download the interface file from the course Web page to ensure that all function names are spelled correctly and each function has the correct number and order of parameters.
  - All solutions must be submitted to MarkUs. No solutions will be accepted through email, even if you are having issues with MarkUs.
  - Verify using MarkUs and your basic test results that your files were properly submitted and are readable on MarkUs.
  - For full style marks, your program must follow the Python section of the CS116 Style Guide.
  - Be sure to review the Academic Integrity policy on the Assignments page
  - Natural numbers in this course begin at 0.
  - Required functions need all design recipe elements. Functions you define (eg. helper functions) need all design recipe elements except for examples and tests.
- Download the testing module from the course web page. Include `import check` in each solution file.
  - When a function produces a floating point value, you *must* use `check.within` for your testing. Unless told otherwise, you may use a tolerance of $0.00001$ in your tests.
  - Test data for all questions will always meet the stated assumptions for consumed values.
- Restrictions:
  - Do not import any modules other than `math` and `check`.
  - You are always allowed to define your own helper/wrapper functions, as long as they meet the assignment restrictions. Do **not** use Python constructs from later modules (e.g. loops (`for` or `while` or others, `zip`, `sorted`, anything with `set`s). For this assignment, abstract list functions will **not** be allowed. Use only the functions and methods as follows:
    - `abs`, `len`, `max` and `min`
    - Any method or constant in the `math` module
    - Type casting including `int()`, `str()`, `float()`, `bool()`, `list()`
    - The command `type()`
    - Any basic arithmetic operation (including `+, -, *, /, //, %, **`)
    - String or list slicing and indexing as well as string or list operations using the operators above
    - Any string or list methods.
    - `input` and `print` as well as the formatting parameter `end` and method `format`. Note that all prompts must match exactly in order to obtain marks so ensure that you do not alter these prompts.
  - Do **not** mutate any passed parameters unless instructions dictate otherwise. You may mutate lists you have created however.
  - While you may use global *constants* in your solutions, **do not** use global *variables* for anything other than testing.
  - Read each question carefully for additional restrictions.
  - **The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.**

## 1 Fibonacci Revisited

Recall the definition of the Fibonacci numbers:

$$f_n = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ f_{n-1} + f_{n-2} & \text{if } n > 1 \end{cases}$$

With what we have done in module 5, we still cannot compute values like the 1000000th Fibonacci number. However, with a clever observation, we can make this a reality. With the Principle of Mathematical Induction, one can verify that for any integer $k \geq 0$,

$$f_{2k} = f_k(2f_{k+1} - f_k) \qquad f_{2k+1} = f_k^2 + f_{k+1}^2$$

and now, one can compute large Fibonacci numbers by using these smaller values instead of computing every fibonacci number before $n$. As a tiny example, we can compute $f_9$ by using:

$$f_9 = f_4^2 + f_5^2 = (f_2(2f_3 - f_2))^2 + (f_2^2 + f_3^2)^2 = (1(2(2) - 1))^2 + ((1)^2 + (2)^2)^2 = 3^2 + 5^2 = 34$$

and using small values of $f_k$ (say values when $k \leq 3$) as known values for the Fibonacci numbers. Write a function

`large_fibonacci(n)`

which consumes a natural number `n` and computes the $n$th Fibonacci number using the above identity.

Sample:

```
large_fibonacci(10000) => large_number
```

where `large_number` is defined in your starter code. As a reminder, abstract list functions are **not** allowed on this assignment.

**Hint**: You must store the values of $f_k$ and $f_{k+1}$ in variables before attempting to compute either $f_{2k}$ or $f_{2k+1}$. Recomputing these values will cause your code to time out. Testing for this code will be hard but you might want to use the code given in class to at least test that your code gives correct values up to the first say 1000 Fibonacci numbers. Your code should not take longer than a few second to compute the millionth fibonacci number.

## 2 Flatten

Write a function

`flatten(L)`

that flattens a `(listof Any)` so that it contains no sublists but all of the elements are in a single list in their relative left to right order in the list.

Sample:

```
flatten([[1,2,3],[4,5]]) => [1,2,3,4,5]
flatten([1,2,3,4,5]) => [1,2,3,4,5]
flatten([[1,2,3],[4,5,[6], None, []]]) => [1,2,3,4,5,6,None]
```

**Hint**: The command `type()` may be of use.

## 3 Slime Numbers

We say that a natural number is a **slime number** if and only if there is some way to slice the number so that each slice is a prime number. For example, the number 70137 is a slime number since you can slice it into 701 then 3 and then 7. Note you could also show this is a slime number by using the slices 701 and 37. We can even use the slices 7, then 013 (which we interpret at 13) and the slice 7. In fact, even the slices 7013 and 7 will also show us this number is a slime number. Remember we just need one such slicing. Note that this generalizes the notion of a prime number since every prime

number is itself a slime number by using the empty slice and just returning the number itself. Further, notice how the number itself does not need to be prime (note that 70137 is a slime number but is not prime as it is divisible by 3).

Write a function

`is_slime(num)`

that returns `True` if and only if the number is a slime number and `false` otherwise. For simplicity, we will never test a slime number that requires a prime number of more than 4 digits.

```
is_slime(70137) => True
```

## 4 $k$-Near Palindrome

Recall that a palindrome is a string that is equal to the string when reversed. A palindrome is called a $k$**-near palindrome** if and only if there are $k$ characters that when removed from the string, the resulting string becomes a palindrome. For example, the string `"abcda"` is a $2$-near palindrome because removing say `"bc"` leaves us with `"ada"` which is a palindrome. Write a function

`near_palindrome(s, k)`

that returns `True` if and only if the string `s` is a $k$-near palindrome and `False` otherwise.

```
near_palindrome("abcda", 2) => True
```