```
%matplotlib inline
import matplotlib
import seaborn as sns
sns.set()
matplotlib.rcParams['figure.dpi'] = 144
```

```
%matplotlib inline
import matplotlib
import seaborn as sns
matplotlib.rcParams['savefig.dpi'] = 144
```

```
from static_grader import grader
```

Program Flow exercises

The objective of these exercises is to develop your ability to use iteration and conditional logic to build reusable functions. We will be extending our `get_primes` example from the Program Flow notebook for testing whether much larger numbers are prime. Large primes are useful for encryption. It is too slow to test every possible factor of a large number to determine if it is prime, so we will take a different approach.

Exercise 1: `mersenne_numbers`

A Mersenne number is any number that can be written as $2^p - 1$ for some p. For example, 3 is a Mersenne number ($2^2 - 1$) as is 31 ($2^5 - 1$). We will see later on that it is easy to test if Mersenne numbers are prime.

Write a function that accepts an exponent p and returns the corresponding Mersenne number.

```
def mersenne_number(p):
    return 2 ** p - 1
```

Mersenne numbers can only be prime if their exponent, p, is prime. Make a list of the Mersenne numbers for all primes p between 3 and 65 (there should be 17 of them).

Hint: It may be useful to modify the `is_prime` and `get_primes` functions from the Program Flow notebook for use in this problem.

```
# we can make a list like this
my_list = [0, 1, 2]
print(my_list)
```

```
[0, 1, 2]
```

```
# we can also make an empty list and add items to it
another_list = []
print(another_list)

for item in my_list:
    another_list.append(item)

print(another_list)
```

```
[]
[0, 1, 2]
```

```
def is_prime(number):
    if number < 2:
        return False

    for i in range(2, number):
        if number % i == 0:
            return False

    return True

def get_primes(n_start, n_end):
    return [x for x in range(n_start, n_end + 1) if is_prime(x)]
```

The next cell shows a dummy solution, a list of 17 sevens. Alter the next cell to make use of the functions you've defined above to create the appropriate list of Mersenne numbers.

```
mersennes = [mersenne_number(x) for x in get_primes(3, 65)]
```

```
grader.score.ip__mersenne_numbers(mersennes)
```

```
==================
Your score:  1.0
==================
```

Exercise 2: `lucas_lehmer`

We can test if a Mersenne number is prime using the Lucas-Lehmer test. First let's write a function that generates the sequence used in the test. Given a Mersenne number with exponent p, the sequence can be defined as

$$n_0 = 4$$

$$n_i = (n_{i-1}^2 - 2) mod (2^p - 1)$$

Write a function that accepts the exponent p of a Mersenne number and returns the Lucas-Lehmer sequence up to $i = p - 2$ (inclusive). Remember that the modulo operation is implemented in Python as `%`.

```
def lucas_lehmer(p):
    n = [4]

    limit = p - 2
    mersenne = mersenne_number(p)

    for i in range(1, limit + 1):
        n.append((n[i - 1] ** 2 - 2) % mersenne)


    return n
```

Use your function to calculate the Lucas-Lehmer series for $p = 17$ and pass the result to the grader.

```
ll_result = lucas_lehmer(17)

grader.score.ip__lucas_lehmer(ll_result)
```

```
==================
Your score:  1.0
==================
```

Exercise 3: `mersenne_primes`

For a given Mersenne number with exponent p, the number is prime if the Lucas-Lehmer series is 0 at position $p - 2$. Write a function that tests if a Mersenne number with exponent p is prime. Test if the Mersenne numbers with prime p between 3 and 65 (i.e. 3, 5, 7, ..., 61) are prime. Your final answer should be a list of tuples consisting of `(Mersenne exponent, 0)` (or `1`) for each Mersenne number you test, where `0` and `1` are replacements for `False` and `True` respectively.

HINT: The `zip` function is useful for combining two lists into a list of tuples

```
def ll_prime(p):
    ll = lucas_lehmer(p)
    return not bool(ll[-1])
```

```
mersenne_primes = [(x, int(ll_prime(x))) for x in get_primes(3, 65)]

grader.score.ip__mersenne_primes(mersenne_primes)
```

```
==================
Your score:  1.0
==================
```

Exercise 4: Optimize `is_prime`

You might have noticed that the primality check `is_prime` we developed before is somewhat slow for large numbers. This is because we are doing a ton of extra work checking every possible factor of the tested number. We will use two optimizations to make a `is_prime_fast` function.

The first optimization takes advantage of the fact that two is the only even prime. Thus we can check if a number is even and as long as its greater than 2, we know that it is not prime.

Our second optimization takes advantage of the fact that when checking factors, we only need to check odd factors up to the square root of a number. Consider a number n decomposed into factors $n = ab$. There are two cases, either n is prime and without loss of generality, $a = n, b = 1$ or n is not prime and $a, b \neq n, 1$. In this case, if $a > \sqrt{n}$, then $b < \sqrt{n}$. So we only need to check all possible values of b and we get the values of a for free! This means that even the simple method of checking factors will increase in complexity as a square root compared to the size of the number instead of linearly.

Lets write the function to do this and check the speed! `is_prime_fast` will take a number and return whether or not it is prime.

You will see the functions followed by a cell with an `assert` statement. These cells should run and produce no output, if they produce an error, then your function needs to be modified. Do not modify the assert statements, they are exactly as they should be!

```
import math
def is_prime_fast(number):
    if number < 2:
        return False

    root = round(math.sqrt(number))
    for i in range(2, root + 1):
        if number % i == 0:
            return False
```

```
    return True
```

Run the following cell to make sure it finds the same primes as the original function.

```
for n in range(10000):
    assert is_prime(n) == is_prime_fast(n)
```

Now lets check the timing, here we will use the `%%timeit` magic which will time the execution of a particular cell.

```
%%timeit
is_prime(67867967)
```

```
5.04 s ± 108 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
is_prime_fast(67867967)
```

```
603 µs ± 16.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Now return a function which will find all prime numbers up to and including n. Submit this function to the grader.

```
def get_primes_fast(n):
    return [x for x in range(n + 1) if is_prime_fast(x)]
```

```
grader.score.ip__is_prime_fast(get_primes_fast)
```

```
==================
Your score:  1.0
==================
```

Exercise 5: sieve

In this problem we will develop an even faster method which is known as the Sieve of Eratosthenes (although it will be more expensive in terms of memory). The Sieve of Eratosthenes is an example of dynamic programming, where the general idea is to not redo computations we have already done (read more about it here). We will break this sieve down into several small functions.

Our submission will be a list of all prime numbers less than 2000.

The method works as follows (see here for more details)

1. Generate a list of all numbers between 0 and N; mark the numbers 0 and 1 to be not prime
2. Starting with $p = 2$ (the first prime) mark all numbers of the form np where $n > 1$ and $np <= N$ to be not prime (they can't be prime since they are multiples of 2!)
3. Find the smallest number greater than p which is not marked and set that equal to p, then go back to step 2. Stop if there is no unmarked number greater than p and less than $N + 1$

We will break this up into a few functions, our general strategy will be to use a Python `list` as our container although we could use other data structures. The index of this list will represent numbers.

We have implemented a `sieve` function which will find all the prime numbers up to n. You will need to implement the functions which it calls. They are as follows

- `list_true` Make a list of true values of length $n + 1$ where the first two values are false (this corresponds with step 1 of the algorithm above)
- `mark_false` takes a list of booleans and a number p. Mark all elements $2p, 3p, \ldots n$ false (this corresponds with step 2 of the algorithm above)
- `find_next` Find the smallest `True` element in a list which is greater than some p (has index greater than p (this corresponds with step 3 of the algorithm above)
- `prime_from_list` Return indices of True values

Remember that python lists are zero indexed. We have provided assertions below to help you assess whether your functions are functioning properly.

```
def list_true(n):
    return [False] * (2) + [True] * (n - 1)
```

```
assert len(list_true(20)) == 21
assert list_true(20)[0] is False
assert list_true(20)[1] is False
```

Now we want to write a function which takes a list of elements and a number p and marks elements false which are in the range $2p, 3p \ldots N$.

```
def mark_false(bool_list, p):
    limit = ((len(bool_list) -1)  // p) + 1
    for i in range(2, limit):
        bool_list[i*p] = False
    return bool_list
```

```
assert mark_false(list_true(6), 2) == [False, False, True, True, False, True, False]
```

Now lets write a `find_next` function which returns the smallest element in a list which is not false and is greater than p.

```
def find_next(bool_list, p):
    for i in range(p + 1, len(bool_list)):
        if bool_list[i]:
            return i
```

```
assert find_next([True, True, True, True], 2) == 3
assert find_next([True, True, True, False], 2) is None
```

Now given a list of `True` and `False`, return the index of the true values.

```
def prime_from_list(bool_list):
    return [i for i, x in enumerate(bool_list) if x]
```

```
assert prime_from_list([False, False, True, True, False]) ==  [2, 3]
```

```
def sieve(n):
    bool_list = list_true(n)
    p = 2
    while p is not None:
        bool_list = mark_false(bool_list, p)
        p = find_next(bool_list, p)
    return prime_from_list(bool_list)
```

```
assert sieve(1000) == get_primes(0, 1000)
```

```
%%timeit
sieve(1000)
```

```
403 µs ± 6.18 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
%%timeit
get_primes(0, 1000)
```

```
5.07 ms ± 154 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
grader.score.ip__eratosthenes(sieve)
```

```
==================
Your score:  1.0
==================
```