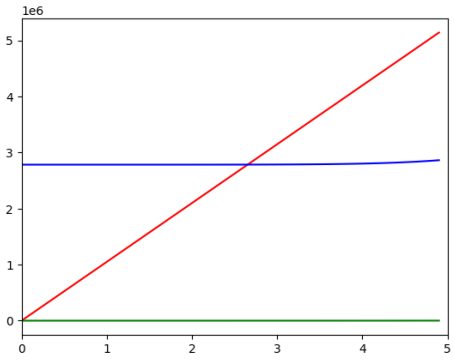
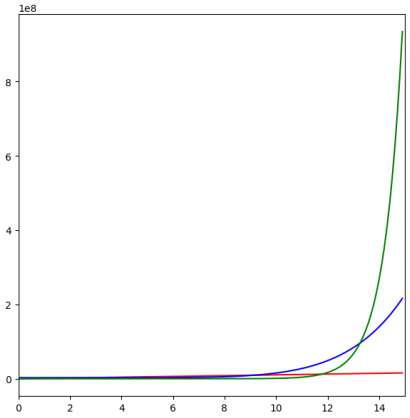
Assignment 4 Primo Marquez

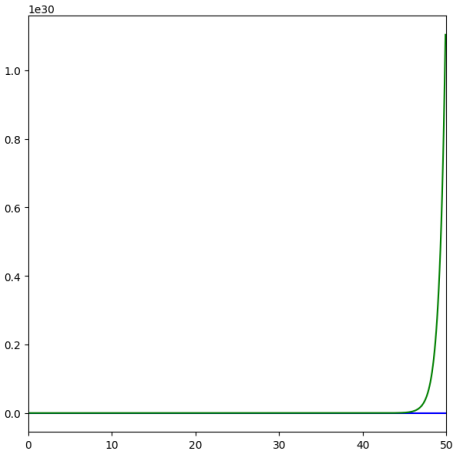
1. The first plot shows f1 initially has the highest rate of growth, whereas f2 and f3 have slower growth rates. F3 has the slowest growth rate.



The second plot shows the growth rate of f3 significantly increased, soaring over the other two functions. The growth rate of f2 also increased, overtaking f1's rate of growth.



The third plot illustrates how fast f3 grows relative to f2 and f1. F2 is barely visible in the plot, indicating its somewhat high growth rate relative to f3, and f1 is no longer visible. This shows how slow f1 grows.



2. $2^{(2n + 2.3)} = O(2^{(n)})$
   - For c = 4, $2^{4n} > 2^{(2n + 2.3)}$ when n = 2.

   $3^{(2n)} = O(3^{(n)})$
   - For c = 4, $3^{4n} > 3^{2n}$ when n = 1.

3. $f(n) = (4n)^{150} + (2n + 1024)^{400}$ vs. $g(n) = 20n^{300} + (n + 121)^{152}$
   f(n) != O(g(n))
   Reducing each function to its highest-order terms we have:
   $f(n) = n^{400}$ vs $g(n) = n^{300}$
   Because f(n) will always grow faster than g(n), f(n) != O(g(n))

   $f(n) = n^{1.4} * 2^{2n}$ vs. $g(n) = n^{100} + 3.99^{n}$
   f(n) = O(g(n))
   Reducing each function to its highest-order terms we have:
   $f(n) = 4^{n}$ vs $g(n) = 3.99^{n}$
   Both functions grow at the same rate, so there exists a c and an x_0 such that f(x) <= g(x)

   $f(n) = 2^{\log2(n)}$ vs. $g(n) = n^{1024}$
   f(n) = O(g(n))
   Reducing each function to its highest-order terms we have:
   $f(n) = 2n$ vs. $g(n) = n^{1024}$
   Because g(n) will always grow faster than f(x), f(n) = O(g(n))

4. The Big O of the pseudocode is **O(nlog(n))**
   There are two nested loops, where the outer loop runs n times and the inner loop runs dependent on the inner loop. Since the number of times the inner loop runs decreases as the outer loop iterates, we say that the while loop runs log(n) times. Therefore, the algorithm has a Big O of nlog(n).

5. The Big O of the pseudocode is **$O(n^3)$**
   There are two nested loops, where the outer loop runs n times and the inner loop runs dependent on the inner loop. The inner loop runs (i x n) times, which means as n increases, it iterates n^2 times. The algorithm runs $n(n^2)$ times, so we say the Big O is $O(n^3)$.

Python Code:

```
import math
import numpy as np
import matplotlib.pyplot as plt

msize = 5

t = np.arange(0, msize, 0.1)

# red dashes, blue squares and green triangles
# plt.plot(t, t, 'r--', t, t**3.5 - 2**10, 'bs', t, 100*t**2.1 + 50, 'g^')

plt.plot(t, (2**20) * t + 2, 'red', t, (t**7.1) + 2.1**20, 'blue', t,
(4**t) - 2.1**8, 'green')

plt.xlim(0, msize)
plt.rcParams["figure.figsize"] = (7,7)
plt.show()
```

```
import math
import numpy as np
import matplotlib.pyplot as plt

msize = 15

t = np.arange(0, msize, 0.1)

# red dashes, blue squares and green triangles
# plt.plot(t, t, 'r--', t, t**3.5 - 2**10, 'bs', t, 100*t**2.1 + 50, 'g^')

plt.plot(t, (2**20) * t + 2, 'red', t, (t**7.1) + 2.1**20, 'blue', t,
(4**t) - 2.1**8, 'green')

plt.xlim(0, msize)
plt.rcParams["figure.figsize"] = (7,7)
plt.show()
```

```
import math
import numpy as np
import matplotlib.pyplot as plt

msize = 50

t = np.arange(0, msize, 0.1)

# red dashes, blue squares and green triangles
# plt.plot(t, t, 'r--', t, t**3.5 - 2**10, 'bs', t, 100*t**2.1 + 50, 'g^')

plt.plot(t, (2**20) * t + 2, 'red', t, (t**7.1) + 2.1**20, 'blue', t,
(4**t) - 2.1**8, 'green')

plt.xlim(0, msize)
plt.rcParams["figure.figsize"] = (7,7)
plt.show()
```