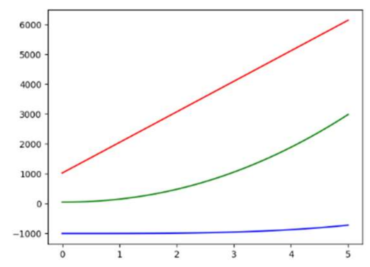
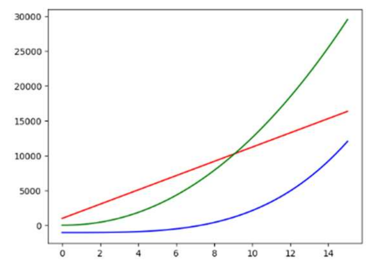
TASK 1

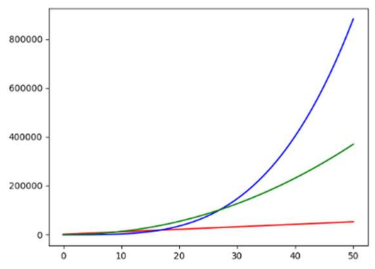
$f1(n) = 2^{10}(n) + 2^{10}$	RED	Linear
$f2(n) = n^{3.5} - 1000$	BLUE	Slight Power
$f3(n) = 100n^{2.1} + 50$	GREEN	Largest Power



Here, the overall result for this smaller domain is that the linear function (red) reaches the highest value, the slight power function (green) reaches the middle values, and the largest power function (blue) has the lowest values.



Here, things get a bit more interesting. The linear function (red) no longer reaches the largest values, but the slight power function (green) does. The largest power function (blue) is still lagging behind both.



Now, we see what we truly expected. The largest power function (blue) takes the lead with the larger domain, with the slight power function (green) getting the middle values, and the linear function (red) containing the smallest of values.

What is interesting here is that for a different amount of steps in a function, there is more time complexity with varying functions. Of these three graphs, for functions with less steps, the highly exponential function is best, and for functions with more steps, the linear function is best.

CODE

```
"""plots.py"""

import matplotlib.pyplot as plt
import numpy as np

# TASK 1
# Regular Plots
def f1(n):
    """first function"""
    return ((2 ** 10) *n) + (2 ** 10)
def f2(n):
    """second function"""
    return (n ** 3.5) - 1000
def f3(n):
    """third function"""
    return (100 * (n ** 2.1)) + 50

def plot(domain):
    """plots the functions given a domain max value"""
    x = np.linspace(0, domain)
    plt.plot(x, f1(x), label = "f1", color = 'red')
    plt.plot(x, f2(x), label = "f2", color = 'blue')
    plt.plot(x, f3(x), label = "f3", color = 'green')
    plt.show()

plot(5)
plot(15)
plot(50)
```

TASK 2

ASYMPTOTIC NOTATION

Is $2^{(n+1.3)} = O(2^n)$?

$f(n) = 2^{(n+1.3)}$, $f(n) \leq c * g(n)$

$g(n) = O(2^n)$

$f(n) / g(n) \leq c$

$2^{(n+1.3)} / 2^n = 2^{1.3} \leq c$

Since $2^{n+1.3} = 2^n * 2^{1.3}$, and the big-O notation (time complexity) simply worries about the largest factor and not any constants, the time complexity of $2^{n+1.3}$ is $O(2^n)$.

Is $3^{2n} = O(3^n)$?

$f(n) = 3^{2n} = 9^{n)}$, $f(n) \leq c * g(n)$

$g(n) = O(3^n)$

$f(n) / g(n) \leq c$

$9^n / 3^n = 3^n \leq c$

Since $3^{2n} = 9^n$, and the big-O notation (time complexity) simply worries about the largest factor, when comparing 9^n with 3^n, there is a factor of 3^n, which is a function (not a constant), this making 3^{2n} not $O(3^n)$.

TASK 3

PAIRS OF FUNCTIONS

$f(n) = (4n)^{150} + (2n + 1024)^{400}$, $g(n) = 20n^{400} + (n + 1024)^{200}$

Is $(4n)^{150} + (2n + 1024)^{400} = O(20n^{400} + (n + 1024)^{200})$?

Simplify f(n): $(4^{150}n^{150}) + (2^{400}n^{400} + \ldots)$

Simplify for time complexity: $n^{150} + n^{400} \to n^{400}$

Simplify O(g(n)): $O((20n^{400}) + (n^{200} + \ldots))$

Simplify for time complexity: $n^{400} + n^{200} \to n^{400}$

Is $n^{400} = n^{400}$?

Yes. Since the dominant term for both functions is n^{400}, they are equivalent time complexity functions.

$f(n) = 4^n n^{1.4}$, $g(n) = 3.99^n n^{200}$

Is $4^n n^{1.4} = O(3.99^n n^{200})$? Is $4^n n^{1.4} = 3.99^n n^{200}$?

No. Because f(n) and g(n) have different power growths (f having 1.4 and g having 200), they are not equivalent.

$f(n) = 2^{\log(n)}$, $g(n) = n^{1024}$

Is $2^{\log(n)} = O(n^{1024})$?

Take the log of both sides: $\log(n)*\log(2) = 1024*\log(n)$

Simplify: $\log(n) = \log(n)$

Is $\log(n) = \log(n)$?

Yes. Since time complexity does not care about the constants and just focuses on the dominant term, with both having time complexity log(n), they are equivalent time complexity functions.

TASK 4 (1)

ANALYSIS OF ALGORITHMS

ALGORITHM 1:

```
STEPS:                                            O(Step):
i = 1                                             1
while i < n do                                    n
    A[i] = i                                      1
    i = i + 1                                     1
end while                                         1
for j <- 1 to n do                                n
    i = j                                         1
    while i <= n do                               log(n)
        A[i] = i                                  1
        i = i + j                                 1
    end while                                     1
end for                                           1
```

To get the net time complexity of the function, we must individually add up time complexities of each step.

Summing up the O(Step)s, we get 9 + n + n * log(n).

Since time complexity only cares about the largest term, this simplifies down to <u>**nlog(n)**</u>.

TASK 4 (2)

ANALYSIS OF ALGORITHMS

ALGORITHM 2:

```
STEPS:                                            O(Step):

x = 0                                             1

for i <- 0 to n do                                n

    for j <- 0 to i*n do                          $n^2$

        x = x + 10                                1

    end for                                       1

end for                                           1
```

To get the net time complexity of the function, we must individually add up time complexities of each step.

Summing up the O(Step)s, we get $4 + n * n^2$.

Since time complexity only cares about the largest term, this simplifies down to **$\underline{n^3}$**.