Information Final Report

Problem 1

1.

1_Root_Finder.py > ...

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

#Function
def f(x):
    return x ** 3 + np.log(x)

```

```python
#Function Plot
x = np.linspace(0.1, 2, 1000)
y = f(x)
plt.figure(figsize=(15, 5))
plt.subplot(131)
plt.plot(x, y, 'r-', label='f(x)')
plt.grid(True)
plt.legend()
plt.title('f(x) = x³ + ln(x)')
plt.xlabel('x')
plt.ylabel('f(x)')

# A marker for roots
root_nr, iter_nr, err_nr = newton_raphson(1.0)
plt.plot(root_nr, 0, 'ro', label='Newton Root', markersize=8)
```

Defined the function as f(x) and asked it to return the function you defined. Then used matplotlib's plot functions to plot the function. np.linspace makes sure that there are 1000 data plots in the given space and the plot function plots the graph. The rest are for labels and axes. I also marked out the root and used Newton-Raphson Method for this.

2.

1_Root_Finder.py > fx bisection_method

```python
#Bisection Method
def bisection_method(a, b, tol=1e-6, max_iter=100):
    iterations = []
    errors = []
    x_prev = a

    for i in range(max_iter):
        c = (a + b) / 2
        iterations.append(i)
        errors.append(abs(c - x_prev))
        x_prev = c
        if abs(f(c)) < tol:
            return c, iterations, errors
        if f(c) * f(a) < 0:
            b = c
        else:
            a = c
    return c, iterations, errors
```

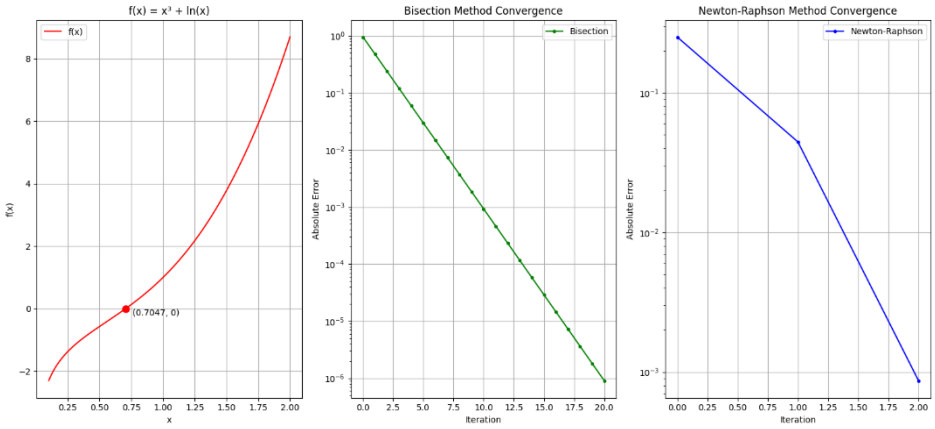
I defined the range at first which is a & b where the root lies in. Then I initiated a loop inside which I defined their midpoint, c. If the absolute value of f(c) is lower than the given tolerance, then the loop stops and returns the value of c, number of iterations and errors and each stage which have been defined as a list. Until that is achieved the functions run the Bisection Method Process where if the product of the left side function value and the midpoint function value is negative then the midpoint becomes the new right side point otherwise the opposite. When the product is negative it shows that a root lies in the interval that's why c becomes the new b. If it's positive, then the root lies in the other interval in which case c becomes a.

3.

```python
#Newton-Raphson Method
def newton_raphson(x0, tol=1e-6, max_iter=100):
    iterations = []
    errors = []
    x = x0
    x_prev = x0

    for i in range(max_iter):
        x = x - f(x) / df(x)
        iterations.append(i)
        errors.append(abs(x - x_prev))
        x_prev = x
        if abs(f(x)) < tol:
            return x, iterations, errors
    return x, iterations, errors
```

Just defined the basic idea of Newton Raphson Method, which comes from drawing a tangent near the root, and then we use basic trigonometry and the fact that tan(θ) is the gradient. In this I kept iterating until the absolute value of the function is less than the tolerance, in which case the root, iterations and their error each case will be returned.

4.



From the semiology graphs we can see Newton-Raphson Method gives us desired value much quicker part of which is that I picked a good starting position for the function. Both Methods have a time complexity of O(n) but in the case of Newton Raphson Method it can be reduced with a good guess.

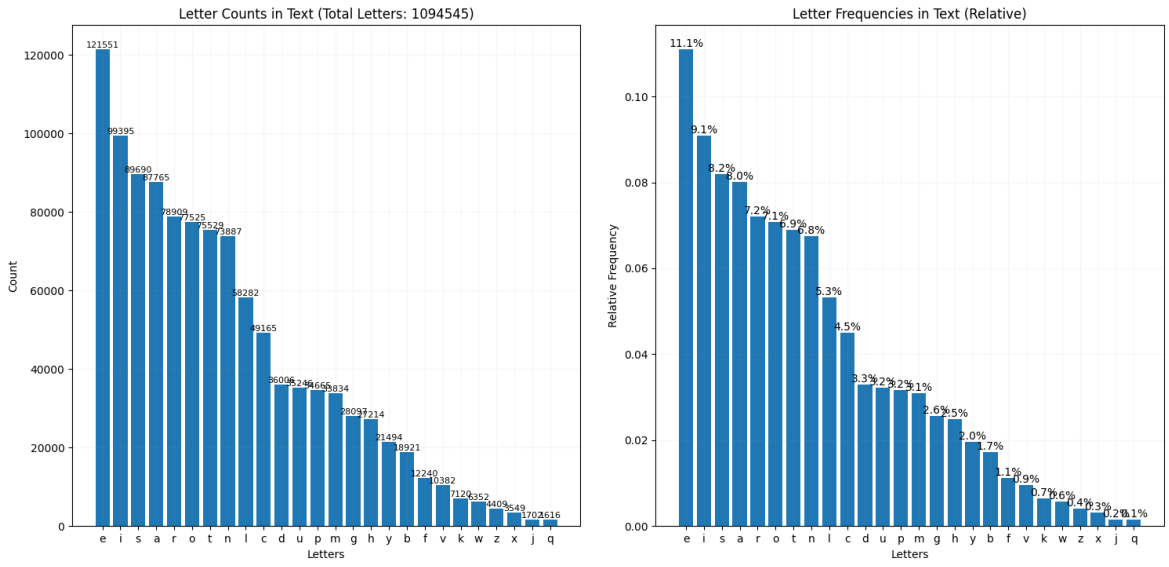
Problem 2

1.

```python
def analyze_text(filename):
    try:
        with open(filename, 'r', encoding='utf-8') as file:
            text = file.read().lower()
    except FileNotFoundError:
        print("File not found.")
    letters = [char for char in text if char.isalpha()]
    total_letters = len(letters)

    #Raw counts
    counter = Counter(letters)
    raw_counts = dict(sorted(counter.items(), key=lambda x: x[1], reverse=True))

    #Frequencies
    frequencies = {char: count/total_letters for char, count in counter.items()}
    sorted_frequencies = dict(sorted(frequencies.items(), key=lambda x: x[1], reverse=True))

    return raw_counts, sorted_frequencies, total_letters
```

First, I opened the file in UTF-8 Encoding, then converted all the characters to lowercase as the UTF-8 Encoding is different for Capital & Lowercase Letters (A-Z: 65-90 & a-z: 97-122 in decimal values). Then used isalpha function to check if the encoding lies in between 97-122 in which case this is a lowercase English alphabet. Used the counter function from the counter library to separate the letters and count them. In the raw count I converted all of them into separate lists using the lambda function. Did almost same thing for the frequency and then divided them by the total numbers of characters to get the frequency. Then sorted them in descending order using reverse.

2. Used the same methods as in Problem 1 to create histograms. Used plt.bar instead of plt.plot which is for histogram. Added the counts & frequencies which is just the height of the bars. Below is the histogram:



3.

```python
def calculate_entropy(frequencies):
    entropy = -sum(p * np.log2(p) for p in frequencies.values())
    return entropy
```

Used the formula you taught in class to get the entropy along with the formula to find the mean ie: $\sum xf(x)$, where p is the relative frequency or probability in other words.

4. The average entropy is just $log_2 p$ where p is the probability to a certain letter occurring which is 1/26. Also calculated the efficiency of this histogram which is just it's Entropy/Avg_Entropy. Results of 3 & 4:

```
Entropy per character: 4.1822 bits
Maximum possible entropy: 4.7004 bits
Entropy efficiency: 88.97%
```

Problem 3

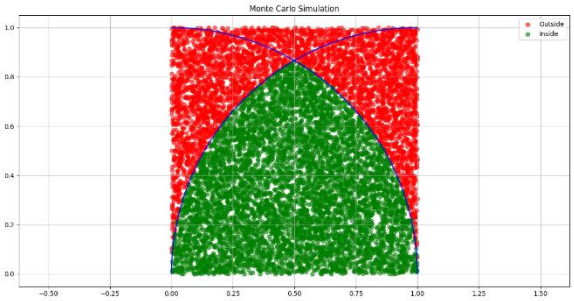
1.

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

def sampling(x, y):
    # Point must be below both quarter circles
    left_circle = (x-0) ** 2 + (y-0) ** 2 <= 1
    right_circle = (x-1) ** 2 + (y-0) ** 2 <= 1
    return left_circle and right_circle

def monte_carlo_estimation(n=10000):
    # Generate random points
    x = np.random.uniform(0, 1, n)
    y = np.random.uniform(0, 1, n)
    points_inside = sum(sampling(x[i], y[i]) for i in range(n))
    estimated_area = points_inside / n
    return estimated_area
```

```python
def visualize_simulation(n=10000):
    # Generate random points
    x = np.random.uniform(0, 1, n)
    y = np.random.uniform(0, 1, n)
    inside = [sampling(x[i], y[i]) for i in range(n)]
    plt.figure(figsize=(10, 10))
    plt.scatter(x[~np.array(inside)], y[~np.array(inside)],
                c='red', alpha=0.6, label='Outside')
    plt.scatter(x[np.array(inside)], y[np.array(inside)],
                c='green', alpha=0.6, label='Inside')
    theta = np.linspace(0, np.pi/2, 100)
    plt.plot(np.cos(theta), np.sin(theta), 'b-')
    plt.plot(1 + np.cos(theta+np.pi/2), np.sin(theta+np.pi/2), 'b-')
    plt.axis('equal')
    plt.grid(True)
    plt.legend()
    plt.title('Monte Carlo Simulation')
    plt.show()
```

In the sampling function I defined two circles centered at (0,0) & (1,0) the functions return true if and only if the given point lies inside both quarter circles. The function monte_carlo_estimation generates random points in my case I gave 10000 points in the range of (0,0) to (1,1. Then I use the sampling function to find which of the 10000 points are within the circles and then divide it by the total number of points to get the estimated area or also the probability. I also added a scatter plot to visualize the whole thing. Using the sampling function, I defined an array inside where all the satisfied coordinates are stored. Points which satisfy the condition get a green color and those which don't get a red color this is done by using the bitwise not operator (~). I drew two lines in polar coordinates to show the quarter circles. I also ran 5 simulations of the monte_carlo_estimation to get an even better result.

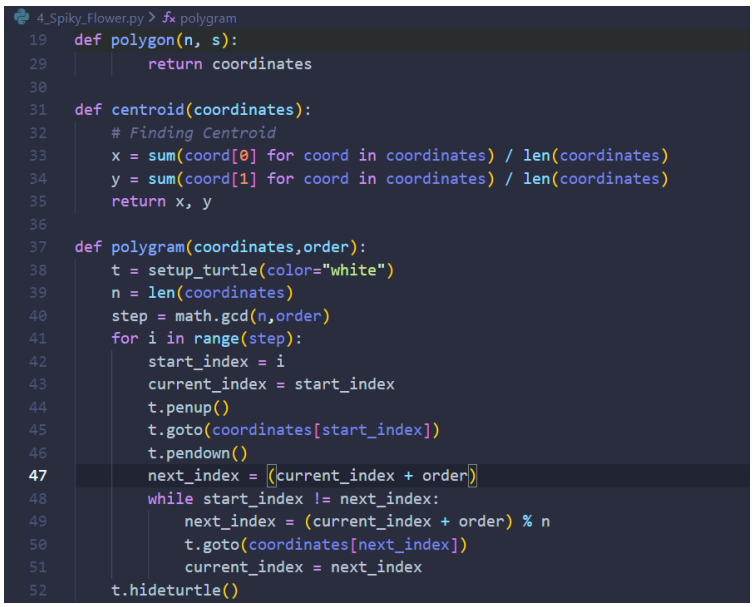
2. The area comes out to be $\frac{\pi}{3} - \frac{\sqrt{3}}{4}$. This can be easily found by joining the centers of the circles to their intersection point and forming an equilateral triangle.
   Results:




```
Monte Carlo Estimation Results:
Average estimated area: 0.614760
Exact area (π/3 - √3/4): 0.614185
Relative error: 0.0936%
```

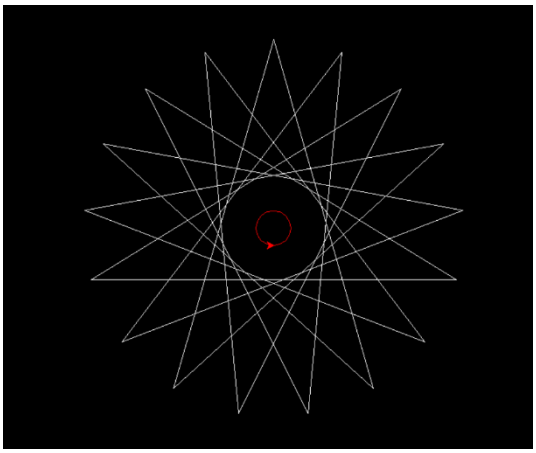
Problem 4

1. After some research into the figure I found that these figures are called polygrams. These are made by joining every k-th vertex of a polygon. The polygram has two terms which define it. These are n, the number of sides of the polygon and the order which is after how many vertices should the k-th vertex connect to. The order can range from 2 to a number less than n/2. In the question the given figure is a polygram of size 17 and order of 7. Some polygrams also have a degeneracy effect where the line connecting the vertices reaches the initial vertex before visiting all the vertices once. The given polygram doesn't have a degeneracy effect (as 17 and 7 are co-prime this fact will come into play later in the code). My code also solves the degeneracy issue of polygrams and thus can draw polygrams of any number of sides and order perfectly.

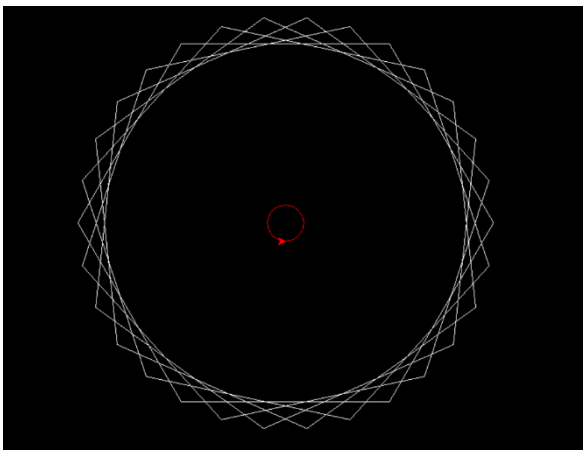


First, I did a basic polygon function where I collected all the coordinates in an array. The centroid function sums all of the x,y coordinates and divides it by the number of coordinates to find the centroid of the polygon. The main work is done inside the while function it draws a line from the current point to the next point which is found by adding the order number to the current point. To make sure it doesn't exceed the polygon size I added a modulus operator with it. After that the next point becomes the current point and, in this way, it continues until the next point becomes the starting point in which case the loop breaks. However, polygrams also has a degeneracy effect as mentioned before. The solve this I put the while loop inside the for loop which will change the starting position and run the while loop again. Now it turns out the number of times we need to iterate this while loop inside the for loop is the greatest common divisor of the size and order. It does make sense if we see an example like 10-gram with an order of 2. When we run the while loop only 5 points are touched (10/2 = 5) and so we need to run it once again from the next point to complete the polygram. Similarly for 30-gram with an order of 3 only 10 points are touched (30/3 = 10) and so we need to go the next starting point and run it once again which completes 20 points and now doing it once more completes all 30 points. I can't explain all the mathematical details, but we need to find the gcd of the size and order and then iterate the while loop that many times.

To draw the circle, we need to move the turtle to centroid and then move it r distance below ie: (x,y-r).
Also, when increasing the order and sides decreases the size of the circle we can fit and the whole space converges towards a singular point, which makes sense as increasing the sides makes the polygon more into a circle and increasing the order makes the next point near the opposite of the starting point and so the line goes very near the centroid.



Polygram with 17 sides and order of 7.
The one asked in the question.



Polygram with 30 sides and order of 5.

Problem 5

1.

```
def koch_curve(t, level, size):
    if level == 0:
        t.forward(size)
    else:
        segment = size / 3
        # Treat each segment as a smaller curve
        t.pendown()
        koch_curve(t, level-1, segment)
        t.left(60)
        koch_curve(t, level-1, segment)
        t.right(120)
        koch_curve(t, level-1, segment)
        t.left(60)
        koch_curve(t, level-1, segment)
```

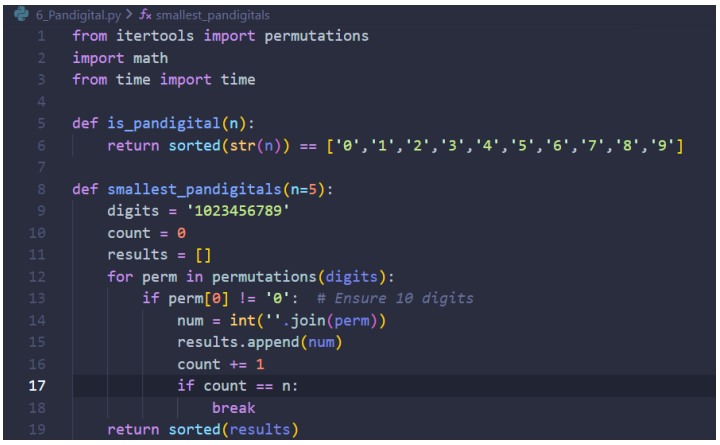
```
def koch_snowflake(t, level, size):
    #Center the snowflake
    t.penup()
    t.backward(size/2)
    t.left(90)
    t.forward(size/3)
    t.right(90)
    t.pendown()
    #3 Curves for snowflakes
    for i in range(3):
        koch_curve(t, level, size)
        t.right(120)
```

The function Koch_curve draws a straight line for level 0. For any other level n, at first the function divides the line into 3 equal parts. And then draws a Koch curve of level n-1 moves 60° anticlockwise draws another then moves 120° clockwise and then moves 60° anticlockwise or the initial alignment and then draw the final Koch curve. In this way the function will draw curves while decreasing the level by 1 every time until reaches level 0 that's when it will just draw a straight line.

2. I used the basic idea of drawing a triangle where every time after drawing a line the cursor will move 120° anticlockwise and continue for 3 times or until it reaches the initial coordinates, just instead of straight lines it'll draw Koch curves, and thus make a Koch Snowflake. I also centralized the snowflake, by moving the cursor (-size/2,size/3) which will then put the centroid of the koch snowflake at the center of the screen.

Problem 6

1.

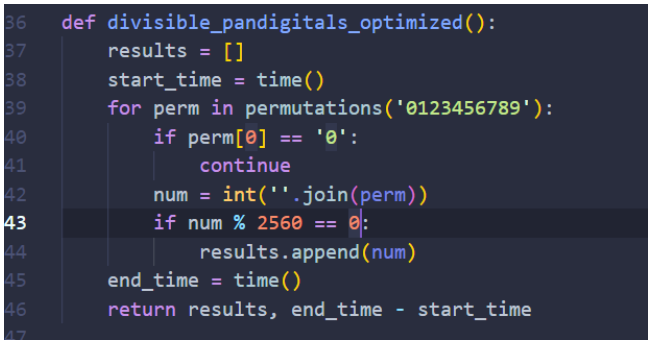


The is_pandigital returns True if it manages to sort the digits like the given order otherwise False.
For the finding the smallest pandigital I used the permutation function from itertools library to find the permutations of the digit 1023456789 and in this way return the first 5 permutations which is also the smallest 5 pandigitals.

2.

Checks through all the pandigitals using a for loop and then uses another for loop to check if the number is divisible by all the numbers from 1-10.

3.



Similar thing but this time instead of checking if the number is divisible by all the numbers from 1-10, it only checks if it is divisible by 2560. This works because 2560 is the least common multiple of the numbers 1-10.

Time Complexity: I checked the time needed to run both functions and this was the result:

```
2 & 3. Pandigital numbers divisible by 1-9:
1st force approach:
Time taken by 1st Method: 8.11 seconds

Optimized approach (using 2520):
Time taken by Optimized Method: 3.25 seconds
```

4.

```
47
48  def pandigital_squares():
49      results = []
50      start_time = time()
51      for perm in permutations('0123456789'):
52          if perm[0] == '0':
53              continue
54          num = int(''.join(perm))
55          root = int(math.isqrt(num))
56          if root * root == num:
57              results.append(num)
58              if len(results) == 2:  # Stop after finding two numbers
59                  break
60      end_time = time()
61      return results, end_time - start_time
```

Takes each number and find the square root of it and turn it into an integer. If the square of that number still equals to that pandigital then that pandigital is a square.

5.

```
63  def pandigital_squares_optimized():
64      results = []
65      start_time = time()
66      min_root = math.isqrt(1023456789)  # sqrt of smallest pandigital
67      max_root = math.isqrt(9876543210)  # sqrt of largest pandigital
68      for root in range(min_root, max_root + 1):
69          square = root * root
70          if is_pandigital(square):
71              results.append(square)
72      end_time = time()
73      return results, end_time - start_time
74
```

Instead of checking if the pandigitals are squares or not. I iterated through the roots and then checked whether their squares are pandigitals or not.

Time Complexity: There was a huge difference in the time complexity this time as we are iterating through a much smaller range.
Result:

```
4 & 5. Pandigital squares:
1st approach:
Time taken: 3.51 seconds

Optimized approach:
Time taken: 0.12 seconds
```