資料結構

HW1

學號 : B103012002

姓名 : 林凡皓

I. Programming environment :

這次作業我使用 Visual Studio Code 做為開發環境和 Python 作為開發語言。要在 Visual Studio Code 中安裝 Python 的開發環境需先到 Python 的官方網站下載 Python，並在 Visual Studio Code 中安裝好 Python 的延伸模組。由於本次作業在資料視覺化的部分有式用到 Python 套件 matplotlib，因此需要到 cmd 輸入 pip install matplotlib。

II. Design of my program :

- __init__ :

  此函數的主要功能為接收一個 coefficient list 並將此 list 儲存到 self._coeff 中，以及根據此 list 計算出多項式的最高次方。

  關於 self._coeff 的部分，由於 coefficient list 的前幾個位元如果為 0 將會被忽略，因此我使用 while loop 來將前面所有 0 移除後再將 coefficient list assign 給 self._coeff。

  關於最高次方的計算，我是透過計算 self._coeff 的長度減一得到。之所以要減一是因為多項式的常數項為 x^0。

- __add__ :

  此函數的主要功能為將兩個多項式相加。

  由於兩個多項式的最高次方不一定相同，因此我將多項式相加想成兩個部分，第一為次方小於等於較小次方的部分，第二為次方大於較小次方的部分。

  對於第一部份來說，相加後的結果為兩個多項是直接相加。

  對於第二部分來說，相加後的結果其實就是次方數較高的多項式。

  要注意的是這樣的思維還需要加上一些 list index 的操作，這是因為 coefficient list 的規則為最高次方最靠左。

- __sub__ :

  此函數的主要功能為將兩個多項式相減。

  解題想法和__add__相同，差別只有在次方小於等於較小次方的部分，原本的兩多項式相加要改為相減，以及要再額外判斷是較大的多項式減較小的多項式，還是較小的多項式減較大的多項式。如果為後者，在次方大於較小次方的部分，相減後的結果會是較大的多項式加上一個負號。

- __mul__ :

  此函數的主要功能為將兩個多項式相乘。

  本題可以利用巢狀 for loop 迭代 self._coeff 和 other._coeff。這邊的 for loop 需要對 enumerate(self._coeff)和 enumerate(other._coeff)迭代，因為會需要同時取得元素的 index 和數值(self._coeff 的 index 會用 i 表示，other._coeff 的 index 會用 j 表示)。

  相乘後的結果的第 i+j 元素的值會是 self._coeff 第 i 個元素的值乘上 other._coeff 第 j 個元素的值。

- __neg__ :

  此函數的主要功能為將一個多項式取負號。

  本題可以先創建一個空的 list，之後利用 for loop 迭代 self._coeff 的每個元素，並將其加負號後 append 到剛才創建的空的 list 中。

III. Time complexity analysis & benchmarking :

- __init__ :

  1. Big O notation :

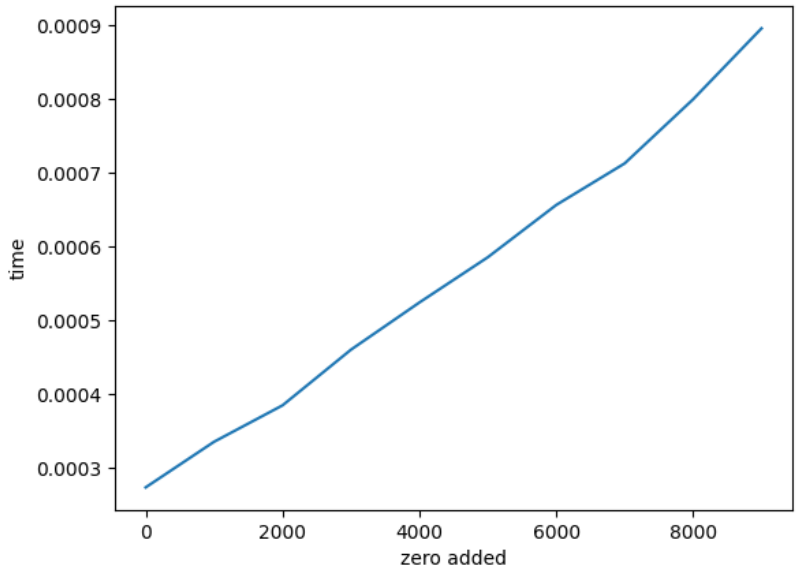
```
def __init__(self, coefficients):
    """
    Initialize the Polynomial instance.

    Args:
        coefficients (list): A list of coefficients, starting with the
                             coefficient of the highest degree term.
    """
    self._coeff = coefficients                  # Big O : O(1)
    while self._coeff and self._coeff[0] == 0:  # Big O : O(n)
        self._coeff = self._coeff[1:]
    self._degree = len(self._coeff) - 1         # Big O : O(1)
```

  Big O 分析結果為 O(n)。

  2. timeit 驗證 :

  __init__中最壞情況為 coefficients 中的最前面有許多 0，如此一來程式執行時會循環 while 迴圈很多次，因此我設計的實驗就是在 coefficients 最前面不段加入 0。驗證結果與視覺化結果如下 :

```
Benchmark for __init__
n              Polynomial()
0                   0.00027
1000                0.00034
2000                0.00039
3000                0.00046
4000                0.00052
5000                0.00059
6000                0.00066
7000                0.00071
8000                0.00080
9000                0.00090
```



由上圖可以看出來，執行時間隨著 0 的加入約為線性成長關係。

- __add__ :

1. Big O notation :

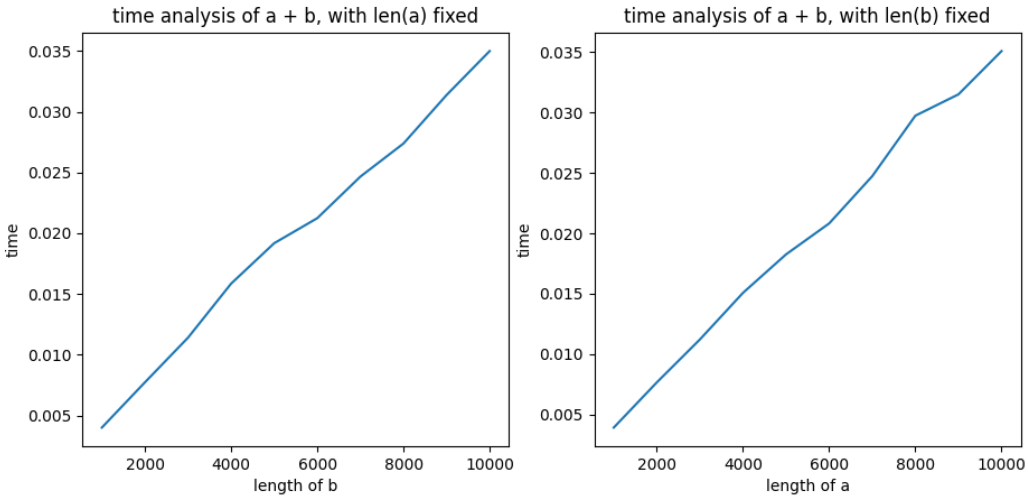
```python
def __add__(self, other):
    """
    Add two polynomials.

    Args:
        other (Polynomial): Another Polynomial instance to add.

    Returns:
        Polynomial: The sum of the two polynomials.
    """

    diff = abs(len(self._coeff) - len(other._coeff))                # Big O : O(1)

    if len(self._coeff) < len(other._coeff):
        result = [0] * len(other._coeff)                            # Big O : O(k)
        result[:diff] = other._coeff[:diff]                         # Big O : O(k)
        for i in range(len(self._coeff)):                           # Big O : O(n)
            result[diff+i] = self._coeff[i] + other._coeff[diff+i]
    else:
        result = [0] * len(self._coeff)                             # Big O : O(n)
        result[:diff] = self._coeff[:diff]                          # Big O : O(n)
        for i in range(len(other._coeff)):                          # Big O : O(k)
            result[diff+i] = self._coeff[diff+i] + other._coeff[i]

    return Polynomial(result)
```

Big O 分析結果為 O(n+k)。其中 n 為 self._coeff 的長度，k 為 other._coeff 的長度。

2. timeit 驗證 :

__add__時間複雜度主要會和 self._coeff 和 other._coeff 的長度相關，因此我選擇固定其中一個長度(a 的長度)，並持續增加另外一個長度(b 的長度)來驗證 Big O 分析結果，驗證結果如下 :

```
Benchmark for __add__
n                   a + b
fixed length of a
1000              0.00399
2000              0.00772
3000              0.01140
4000              0.01585
5000              0.01918
6000              0.02124
7000              0.02466
8000              0.02739
9000              0.03138
10000             0.03500
************************************************
fixed length of b
1000              0.00392
2000              0.00766
3000              0.01122
4000              0.01508
5000              0.01826
6000              0.02083
7000              0.02474
8000              0.02974
9000              0.03151
10000             0.03509
```



由上圖可以看出，不管是固定 a 或 b 的長度，隨著另外一個長度增加，執行時間約為線性成長，因此 Big O notation 應該為 O(n+k)。

- __sub__ :

1. Big O notation :

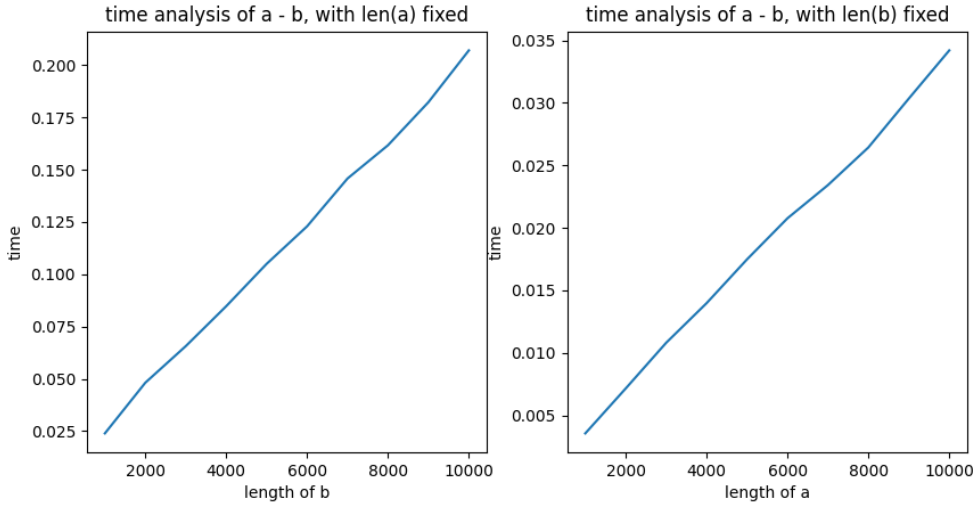
```python
def __sub__(self, other):
    """
    Subtract one polynomial from another.

    Args:
        other (Polynomial): Another Polynomial instance to subtract.

    Returns:
        Polynomial: The result of the subtraction.
    """

    diff = abs(len(self._coeff) - len(other._coeff))                # Big O : O(1)

    if len(self._coeff) < len(other._coeff):
        result = [0] * len(other._coeff)                            # Big O : O(k)
        result[:diff] = [-num for num in other._coeff[:diff]]       # Big O : O(k)
        for i in range(len(self._coeff)):                           # Big O : O(n)
            result[diff+i] = self._coeff[i] - other._coeff[diff+i]
    else:
        result = [0] * len(self._coeff)                             # Big O : O(n)
        result[:diff] = self._coeff[:diff]                          # Big O : O(n)
        for i in range(len(other._coeff)):                          # Big O : O(k)
            result[diff+i] = self._coeff[diff+i] - other._coeff[i]
    return Polynomial(result)
```

Big O 分析結果為 O(n+k)。其中 n 為 self._coeff 的長度，k 為 other._coeff 的長度。

2. Timeit 驗證 ：

__sub__時間複雜度主要會和 self._coeff 和 other._coeff 的長度相關，因此我選擇固定其中一個長度，並持續增加另外一個長度來驗證 Big O 分析結果，驗證結果如下 ：

```
Benchmark for __sub__
n                   a - b
fixed length of a
1000              0.02395
2000              0.04818
3000              0.06567
4000              0.08475
5000              0.10501
6000              0.12295
7000              0.14581
8000              0.16170
9000              0.18229
10000             0.20706
**********************************************
fixed length of b
1000              0.00357
2000              0.00716
3000              0.01081
4000              0.01398
5000              0.01748
6000              0.02077
7000              0.02342
8000              0.02643
9000              0.03034
10000             0.03419
```



由上圖可以看出，不管是固定 a 或 b 的長度，隨著另外一個長度增加，執行時間約為線性成長，因此 Big O notation 應該為 O(n+k)。

- __mul__ :

1. Big O notation :

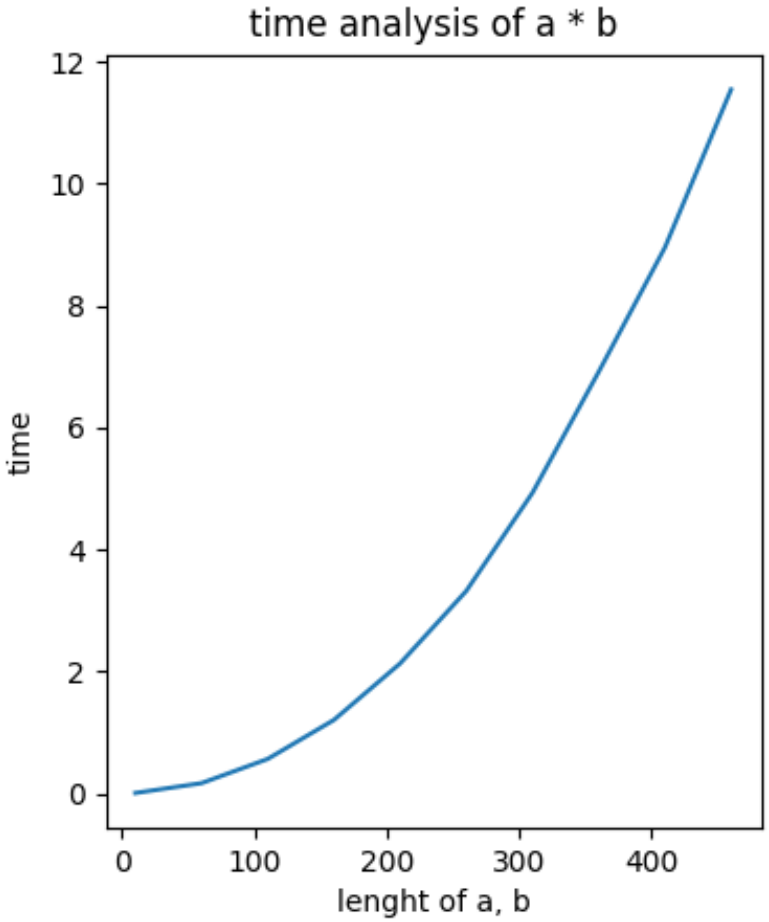
```
def __mul__(self, other):
    """
    Multiply two polynomials.

    Args:
        other (Polynomial): Another Polynomial instance to multiply with.

    Returns:
        Polynomial: The product of the two polynomials.
    """

    result = [0]*(self._degree + other._degree + 1)      # Big O : O(n+k)

    # print (s_coeff, o_coeff)
    for i,self_coeff in enumerate(self._coeff):          # Big O : O(n)
        for j,other_coeff in enumerate(other._coeff):    # Big O : O(k)
            result[i+j] += (self_coeff * other_coeff)
            # print(i,j)
    return Polynomial(result)
```

Big O 分析結果為 O(nk)。其中 n 為 self._coeff 的長度，k 為 other._coeff 的長度。

2. Timeit 驗證 ：

__add__時間複雜度主要會和 self._coeff 和 other._coeff 的長度相乘相關，因此我選擇持續增加 self._coeff 和 other._coeff 的長度來驗證 Big O 分析結果，驗證結果如下 ：

```
Benchmark for __mul__
n, k                 a * b
10                 0.00422
60                 0.16537
110                0.56263
160                1.20259
210                2.12880
260                3.31876
310                4.92932
360                6.91289
410                8.95241
460               11.54459
```



由上圖可以看出，隨著 a, b 長度增加，執行時間約為 x^2 成長，因此 Big O notation 應該為 O(nk)。

- __neg__ :

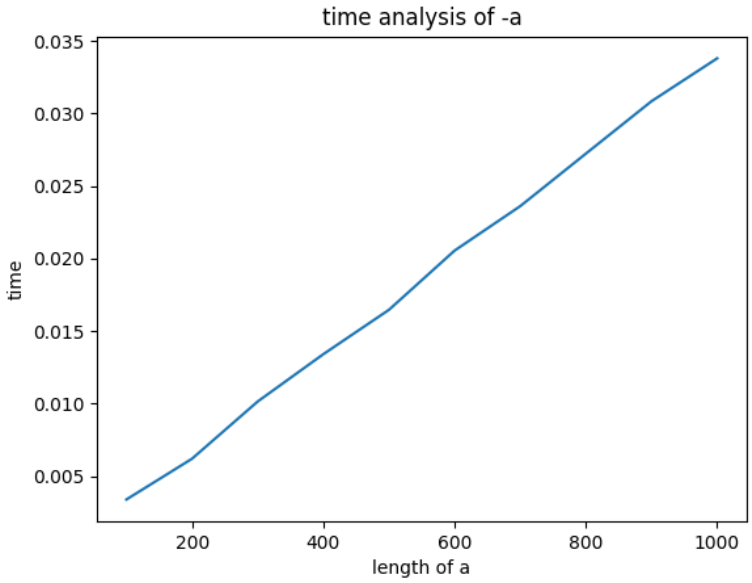
1. Big O notation :

```python
def __neg__(self):
    """
    Negate the polynomial.

    Returns:
        Polynomial: A new Polynomial instance representing the negation of the current polynomial.
    """

    num = len(self._coeff)              # Big O : O(1)
    result = []                         # Big O : O(1)
    for i in range(num):                # Big O : O(n)
        result.append(-self._coeff[i])

    return Polynomial(result)
```

Big O 分析結果為 O(n)。

2. Timeit 驗證 :

```
Benchmark for __neg__
n                    -a
100             0.00340
200             0.00621
300             0.01015
400             0.01341
500             0.01647
600             0.02055
700             0.02361
800             0.02723
900             0.03084
1000            0.03379
```



由上圖可以看出來，執行時間隨著 a 的長度變長約為線性成長關係。

IV. 總結與心得 ：

這次作業主要用的資料結構為 list。

在本次作業中，我花了需多時間在熟悉 Python class 的語法和使用。我在做作業時有遇到一個問題就是假設我在__neg__函數中不是透過創建一個新的 list，而是直接對 self._coeff 做修改並將修改的結果 return，那我只要呼叫這個函數，例如 x5 = - x2，那 x2 的 self._coeff 也會被修改掉。所以不可以直接對 self._coeff 做修改(除非本來就是要在呼叫函數時修改 self._coeff)，而是應該要創建一個新的 list 並對這個新的 list 做修改。

除此之外，本次作業也讓我對 list 的操作更加熟悉，特別是利用 for loop 迭代一個 list，以及 list index operation 的部分，這都是在這次作業中大量使用到的技巧。

最後就是時間分析的部分，上課時聽教授講說 timeit 就是要先創間一個虛擬的執行環境，然後你的主程式為__main__，因此會需要用 form __main__ import 會用到參數。當時聽到只覺得這是什麼，好複雜，但是經過這次時作後發覺到這其實沒有那麼難理解，而且對於創建一個虛擬環境以及 form __main__ import 這兩個部分都有更深的認識。另外我也自己嘗試是用 matplotlib 來將我分析的結果視覺化，在這過程中我除了多了解到 matplotlib 一些基礎的使用方式之外，也透過視覺化的幫助讓我更加清楚我的 code 的時間複雜度。

V. Reference :

[1] OpenAI. (2023). ChatGPT (Mar 14 version) [Large language model]. https://chat.openai.com/

[2] “matplotlib.pyplot” https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html

[3] Vivian Lo (2019) “[演算法]Big O and Time Complexity”. https://medium.com/@yunyubee/%E6%BC%94%E7%AE%97%E6%B3%95-big-o-and-time-complexity-65f2dfafe9d1