CS674A Assignment

Vishal Kumar

Roll-231110058

The Python code in the Assignment.py file is implementing a fast convolution algorithm(negative wrapped convolution) using the Number Theoretic Transform (NTT) and comparing the results with a regular polynomial convolution.

Here's a summary of what the code does:

- ➔ The code defines a few constants, including n (the size of the input arrays/polynomials), q (a prime number) and gamma.
- ➔ It defines helper functions for bit-reversal, generating NTT twiddle factors, and performing the Cooley-Tukey NTT and Gentleman-Sande Inverse NTT operations.
- ➔ In the main function, it generates NTT twiddle factors and initializes two random arrays a and b of size n, representing polynomial coefficients. It also creates a polynomial f, which is used to perform polynomial division to obtain the convolution result.
- ➔ The code calculates the convolution using the regular polynomial multiplication and division operations, resulting in the polynomial p.
- ➔ It then performs the Cooley-Tukey NTT on arrays a and b, element-wise multiplication of the NTT-transformed arrays to get the convolution result c, and finally applies the Gentleman-Sande Inverse NTT to obtain the inverse transform.
- ➔ The code prints the NTT-convolved result c and checks if it is equal to the polynomial-convolved result p.

The code essentially demonstrates how to perform polynomial convolution using the NTT and validates that the NTT-based convolution matches the result obtained using traditional polynomial multiplication and division.

Time Complexity

Traditional Approach: O(n^2)

```python
a = np.random.randint(0, q, n)
b = np.random.randint(0, q, n)

p = np.remainder(np.polydiv(np.polymul(a[::-1], b[::-1]), f)[1], q).astype(int)[::-1]

print("Convolution result (p):", p)
```

- ➔ First of all, we are generating random arrays a and b of size n each using *np.random.randint(0, q, n)* takes O(n) time.
- ➔ Now, we are performing polynomial multiplication using *np.polymul(a[::-1], b[::-1])* which takes O(n^2) time because it involves multiplying each term of the first polynomial by each term of the second polynomial.
- ➔ Then, we are doing polynomial division using *np.polydiv* essentially has a time complexity of O(n) because it involves dividing the two polynomials of degree n and yields both quotient and remainder polynomials.
- ➔ Now, the modulo operation *np.remainder* applied to the result of the polynomial division which will take O(n) time, as it iterates through the coefficients.
- ➔ Then, we are converting the result to an integer array using *astype(int)* is a constant-time operation and can be considered O(1).
- ➔ Finally, we are reversing the result array using *[::-1]* which is also a constant-time operation and can be considered O(1).

Overall, the dominant time complexity for this code segment is the polynomial multiplication, which is **O(n^2)**. The other operations, such as array generation, polynomial division, and modification, have linear or constant time complexities relative to the polynomial multiplication, so they do not significantly impact the overall time complexity.

Fast Approach: O(n*log(n))

The code implements a fast polynomial multiplication algorithm based on the Number Theoretic Transform (NTT) using the Cooley-Tukey and Gentleman-Sande algorithms.

To discuss the time complexity in detail, let's break down the main components and their time complexities:

1) Twiddle Factor Generation:

```
def generate_twiddleFactor(n, gamma, q):
    alpha = 1
    tmp = []
    tFactor = []
    inv_tFactor = []

    for x in range(0, n):
        tmp.append(alpha)
        alpha = alpha * gamma % q

    positions = generate_positions(n)

    for x in range(0, n):
        val = tmp[positions[x]]
        inv_val = pow(val, -1, q)
        tFactor.append(val)
        inv_tFactor.append(inv_val)

    return tFactor, inv_tFactor
```

- ➔ The generate_twiddleFactor() function generates twiddle factors for the NTT.
- ➔ It computes the n twiddle factors using a loop with O(n) complexity.

Therefore, the time complexity is O(n) for generating twiddle factors.

2) Cooley-Tukey NTT:

```python
def cooley_tukey_ntt(a, tFactor, q):
    n = len(a)
    t = n
    m = 1

    while m < n:
        t = t // 2

        for i in range(0, m):
            j1 = 2 * i * t
            j2 = j1 + t - 1
            S = tFactor[m + i]

            for j in range(j1, j2 + 1):
                U = a[j]
                V = a[j + t] * S
                a[j] = (U + V) % q
                a[j + t] = (U - V) % q

        m = 2 * m
```

➔ The *cooley_tukey_ntt()* function performs the Cooley-Tukey NTT on two input arrays/polynomials a and b.

➔ It consists of two nested loops. The outer loop runs log2(n) times, where n is the size of the input array, and each iteration halves the value of t until it reaches 1.

➔ The inner loop performs arithmetic operations on the input arrays, and it runs n times for each outer loop iteration.

Therefore, the overall time complexity of Cooley-Tukey NTT is O(n * log(n)), where n is the size of the input arrays.

3) Gentleman-Sande Inverse NTT:

```python
def gentleman_sande_inv_ntt(a, inv_tFactor, q):
    n = len(a)
    t = 1
    m = n

    while m > 1:
        j1 = 0
        h = m // 2

        for i in range(0, h):
            j2 = j1 + t - 1
            S = inv_tFactor[h + i]

            for j in range(j1, j2 + 1):
                U = a[j]
                V = a[j + t]
                a[j] = (U + V) % q
                a[j + t] = (U - V) * S % q

            j1 = j1 + 2 * t

        t = 2 * t
        m = m // 2

    n_inv = pow(n, -1, q)
    for i in range(0, n):
        a[i] = a[i] * n_inv % q
```

- ➔ The *gentleman_sande_inv_ntt()* function performs the inverse NTT using the Gentleman-Sande algorithm.
- ➔ Similar to Cooley-Tukey, it consists of two nested loops, with the outer loop running log2(n) times, and each iteration halves the value of m until it reaches 1.
- ➔ The inner loop performs arithmetic operations on the input array and runs n times for each outer loop iteration.

Therefore, the overall time complexity of Gentleman-Sande Inverse NTT is O(n * log(n)), where n is the size of the input array.

4) Polynomial Multiplication:

```python
    c = np.multiply(a, b)
```

The polynomial multiplication is performed using NumPy's *np.multiply()* function, which has a time complexity of O(n) since it computes the element-wise product of two arrays.

5) Overall Time Complexity:

- ➔ The overall time complexity is dominated by the Cooley-Tukey NTT and Gentleman-Sande Inverse NTT, both of which have a time complexity of O(n * log(n)).
- ➔ The generation of twiddle factors and polynomial multiplication contribute additional O(n) time complexity.

OUTPUT:

```
main.py

    c = np.multiply(a, b)

    gentleman_sande_inv_ntt(c, inv_tFactor, q)

    print(c)
    if np.array_equal(c, p):
        print("Your code is CORRECT. Both the Traditional and Fast Approaches are giving the same output.")
    else:
        print("Your code is INCORRECT. There is a Mismatch between the Traditional and Fast approaches.")


if __name__ == "__main__":
    main()

```

```
  494  8403  9485  8485  9536 11915  8537  3451  7325  9328 10918  1653
10817  7507  8032  8044 10121  9319  9566  3775  6473  9022 10736  4833
 7491  1985  7363  2338  1886 10440 11835 11501  2383 11443  3372  7211
 1532 10251  2595  7336  4271  1645  7895 10665 10416  1330  8301 11899
 9101   691 10920 11750  6969   460  4192 10663  5984  9450  9223  8287
 7808  4895  5182  6322  1152  5548  8856  5666  8382  2659 11115  5997
 7243  2637  9712 10376  8325  3083  7735  3969  7761  6494  1496  3669
 2280 10626  4502  5156   316 10620  8789  7226  6153  2117   131  5191
 9037  5420  3980  5676  1290  4024  3369  5424 11884 10765    33  6272
11356  8654  5417  3594   852  8846  5879  5032 10866   439 11565 11382
 9389  6611  6704  5003  4252  3709  8192  2280  8229  6055 11755  1583
 4222  8638  4334  7066  4528  2901  5019  2716]
Your code is CORRECT. Both the Traditional and Fast Approaches are giving the same output.


...Program finished with exit code 0
Press ENTER to exit console.
```

Warning: The Python code is running without issues on online compilers and macOS, but it encounters overflow errors on Windows computers.

In summary, the provided code has an overall time complexity of **O(n * log(n))** due to the Cooley-Tukey and Gentleman-Sande NTT algorithms, with some additional O(n) complexity for twiddle factor generation and polynomial operations.

Thank you