Merge Sort and Polynomial Multiplication with FFT

1. Recursive Merge Sort

Recursive merge sort is a sorting algorithm which uses divide and conquer technique to sort the given input. It uses recursion to recursively divide input and conquer them.
Below are the implementations of recursive merge sort and their testing functions.

```
def merge_recursive(x,y):
    if len(x) == 0:
        return y
    if len(y) == 0:
        return x
    if x[0] <= y[0]:
        return [x[0]] + merge_recursive(x[1:], y)
    else:
        return [y[0]] + merge_recursive(x, y[1:])

def merge_sort(a):
    n = len(a)
    if n > 1:
        return merge_iterative(merge_sort(a[:n//2]), merge_sort(a[n//2:]))
    else:
        return a
```

Below is an utility function for testing merge_sort

```
def test_merge(inp,out,sort_func):
    """
    Utility function for checking testing mergesort
    """
    aout = sort_func(inp)
    print("Passed" if out == aout else "Failed ", end="\n")
    print(f"Input : {inp} \nExpected O/P : {out} \nActual O/P : {aout} \n")

test_cases_input = [
                    [], [1], [2, 1], [13, 7, 5], [23, 7, 13, 5], [1, 2, 2, 1, 0, 0, 15, 15],
[135604, 1000000, 45, 78435, 456219832, 2, 546]
]

for case in test_cases_input:
    test_merge(case,sorted(case),merge_sort)
```

Testing merge sort :

```
Passed
Input : []
Expected O/P : []
Actual O/P : []

Passed
Input : [1]
Expected O/P : [1]
Actual O/P : [1]

Passed
Input : [2, 1]
Expected O/P : [1, 2]
Actual O/P : [1, 2]

Passed
Input : [13, 7, 5]
Expected O/P : [5, 7, 13]
Actual O/P : [5, 7, 13]

Passed
Input : [23, 7, 13, 5]
Expected O/P : [5, 7, 13, 23]
Actual O/P : [5, 7, 13, 23]

Passed
Input : [1, 2, 2, 1, 0, 0, 15, 15]
Expected O/P : [0, 0, 1, 1, 2, 2, 15, 15]
Actual O/P : [0, 0, 1, 1, 2, 2, 15, 15]

Passed
Input : [135604, 1000000, 45, 78435, 456219832, 2, 546]
Expected O/P : [2, 45, 546, 78435, 135604, 1000000, 456219832]
Actual O/P : [2, 45, 546, 78435, 135604, 1000000, 456219832]

All test cases passed
```

Runtime profiling

Now we are comparing the merge sort with insertion sort to check the empirical running time performance. Below is a result of in-place insertion sort and recursive merge sort. It clearly shows that insertion sort is faster than merge sort in my implementation. In my merge_recursively() function creates an array every time called and it has large overheads. So always inplace operations outperforms than assigning to a new memory block.

```
[7]   arr = list(range(10000,1,-1))
      print("--------Worst Case profiling----------")

      print("-----------Recursive Merge sort----------")
      %timeit merge_sort(arr)

      print("-------Insertion sort----------")
      %timeit insertion_sort(arr)
```

```
--------Worst Case profiling----------
-----------Recursive Merge sort----------
10 loops, best of 5: 34.7 ms per loop
-------Insertion sort----------
The slowest run took 5736.79 times longer than the fastest. This could mean that an intermediate result is being cached.
1 loop, best of 5: 2.84 ms per loop
```

When in-place merge sort is implemented to check with and check against with in-place insertion sort still insertion sort is faster. Although we know merge sort is faster

```
arr = list(range(100000,1,-1))
print("--------Worst Case profiling----------")

print("-----------Recursive In-place Merge sort----------")
%timeit mergeSort(arr,0,len(arr)-1)

arr = list(range(100000,1,-1))

print("-------In-place Insertion sort----------")
%timeit insertion_sort(arr)
```

```
--------Worst Case profiling----------
-----------Recursive Merge sort----------
The slowest run took 13054.04 times longer than the fastest.
1 loop, best of 5: 50.8 ms per loop
-------Insertion sort----------
The slowest run took 67337.07 times longer than the fastest.
1 loop, best of 5: 23.8 ms per loop
```

2. Polynomial Multiplication with FFT & IFFT

Polynomial multiplication is a trivial algebra problem which involves multiplying two polynomials. When done naively its running time is in the order of $O(n^2)$. We can reduce that to O(nlogn) with the help of Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (IFFT). Time complexity for FFT and IFFT is O(nlogn). We are transforming the coefficient representation to value representation with FFT; this is called *Evaluation*. Transforming back from value to coefficient representation is called *Interpolation*.

Below is the function *poly_mult(a,b)* for multiplying polynomials naively.

Naive polynomial multiplication

```
def poly_mult(a,b):
    n1 = len(a)
    n2 = len(b)
    a  += [0 for _ in range(n2-1)]
    b  += [0 for _ in range(n1-1)]
    c = [sum([a[i]*b[k-i] for i in range(k+1)]) for k in range(n1+n2-1)]
    return c
A = [1, 2, 3]
B = [2, 1, 4]
print("A * B = ", poly_mult(A, B))
```

Output : `A * B =  [2, 5, 12, 11, 12]`

FFT and IFFT Implementation

Below are the functions for converting to the fourier domain and back to the time domain.

```
def FFT(x):
    N = len(x)
    if N == 1:
        return x
    else:
        X_even = FFT(x[::2])
        X_odd = FFT(x[1::2])
        factor = \
          np.exp(-2j*np.pi*np.arange(N)/ N)

        X = np.concatenate(\
            [X_even+factor[:int(N/2)]*X_odd,
             X_even+factor[int(N/2):]*X_odd])
        return X
```

```
def IFFT1(x):
    N = len(x)
    if N == 1:
        return x
```

```
    else:
        X_even = IFFT1(x[::2])
        X_odd = IFFT1(x[1::2])
        factor = np.exp(2j*np.pi*np.arange(N)/ N)

        X = np.concatenate(\
            [X_even+factor[:int(N/2)]*X_odd,
             X_even+factor[int(N/2):]*X_odd])
        return X

def IFFT(x):
    n = len(x)
    return IFFT(x)/n
```

Below function *pad_arrays(A,B)* is a helper function to prepare the given polynomials for FFT.

```
A = [1, 2, 3]
B = [2, 1, 4]

def pad_arrays(A,B):
    n1 = len(A)
    n2 = len(B)
    A = A + [0 for _ in range(n2-1)]
    B = B + [0 for _ in range(n1-1)]
    pad_size_A = 2**ceil(np.log2(len(A))) - len(A)
    pad_size_B = 2**ceil(np.log2(len(B))) - len(B)
    A += [0 for _ in range(pad_size_A)]
    B += [0 for _ in range(pad_size_B)]
    return A,B

A,B = pad_arrays(A,B)
A = np.array(A)
B = np.array(B)
print(A,B,sep='\n')
```

Input :

```
A = [1 2 3 0 0 0 0 0]
B = [2 1 4 0 0 0 0 0]
```

Output :

```
A * B =  [2, 5, 12, 11, 12, 0, 0, 0]
```

Running time profiling

As we can see, the current implementation of polynomial multiplication with FFT and IFFT is worse than naive multiplication. But the same with numpy's implementations is way faster than anything.

```
from random import randint
N = 1000
A = [randint(0,10) for _ in range(N)]
B = [randint(0,10) for _ in range(N)]
%timeit -n 1 poly_mult(A,B)
```

```
The slowest run took 274.36 times longer than the fastest. This could
1 loop, best of 5: 284 ms per loop
```

```
A,B = pad_arrays(A,B)
A = np.array(A)
B = np.array(B)
%timeit -n 1 np.real(IFFT2(FFT(A)*FFT(B)))
%timeit -n 1 np.real(np.fft.ifft(np.fft.fft(A)*np.fft.fft(B)))
```

```
1 loop, best of 5: 2.97 s per loop
1 loop, best of 5: 6.09 ms per loop
```