DATA-Lab2

Question 1 & 2.

There is no score assigned to those 2, so I wrote them on paper for better illustration.

Question 3.

The algorithms are implemented in ***3_insertion_sort_merge_sort.py***

Question 4.

To have an overview of the runtime, I ran 1501 test cases.

For each test case, ***"numpy.random"*** to create a random input array B of n (0≤n<1000). Each item B[i] in B has a value up to 10**7: *0≤B[i]≤10**4.* I acknowledge that the greater the integer, the more it costs to compare with each other. In our cases, we do not care about the value of integer, so I choose max value of integer is 10000, which is nearly 7 times greater than the maximum number of integers in sequence (n=1500). This ensures that the randomized array is well distribute

To test the precision of my functions, I check if the sorted arrays from ***insertionSort()*** and ***mergeSort()*** match with one from Python ***sorted()*** function. If there is any failed test case, the input of that test case is printed in the sdout.

"time.time()" in Python is used to track the cpu time at the start and end of each function. The cpu time needed for each function is the difference of two points of time (end - start).

If ***mergeSort()*** runs faster than ***insertionSort()*** for more than 100 consecutive values of n, the range is printed out. However, other cases are still counted in the total number of test that MergeSort is faster than InsertionSort.

As you can see from some attempts below, when n ≤ 100, there is no interval of length greater than 100 that ***mergeSort()*** runs faster, and the percentage of those cases are smaller than 10% , implying randomness. The larger n is, the higher the percentage and the more consistent that ***mergeSort()*** is faster. It is around 92% when n≤1500. I also tried a case with n=10**4, maxValue=10**7; the percentage goes up to 99%.

Result from overall test:

```
Attempt no.1
        For all n <= 100, percentage of cases that mergeSort() is faster than insertionSort() is 0.99%
        For all n <= 500, percentage of cases that mergeSort() is faster than insertionSort() is 77.05%
        For all n <= 1000, percentage of cases that mergeSort() is faster than insertionSort() is 88.51%
        For all n <= 1500, percentage of cases that mergeSort() is faster than insertionSort() is 92.34%
        Percentage of cases passed: 100.00%
        From n = 135 to 1500, mergeSort() runs consistently faster than insertionSort()
Attempt no.2
        For all n <= 100, percentage of cases that mergeSort() is faster than insertionSort() is 1.98%
        For all n <= 500, percentage of cases that mergeSort() is faster than insertionSort() is 77.84%
        For all n <= 1000, percentage of cases that mergeSort() is faster than insertionSort() is 88.91%
        For all n <= 1500, percentage of cases that mergeSort() is faster than insertionSort() is 92.60%
        Percentage of cases passed: 100.00%
        From n = 171 to 1500, mergeSort() runs consistently faster than insertionSort()
Attempt no.3
        For all n <= 100, percentage of cases that mergeSort() is faster than insertionSort() is 4.95%
```

```
        For all n <= 500, percentage of cases that mergeSort() is faster than insertionSort() is 78.24%
        For all n <= 1000, percentage of cases that mergeSort() is faster than insertionSort() is 89.11%
        For all n <= 1500, percentage of cases that mergeSort() is faster than insertionSort() is 92.74%
        Percentage of cases passed: 100.00%
        From n = 123 to 1500, mergeSort() runs consistently faster than insertionSort()
Attempt no.4
        For all n <= 100, percentage of cases that mergeSort() is faster than insertionSort() is 1.98%
        For all n <= 500, percentage of cases that mergeSort() is faster than insertionSort() is 77.25%
        For all n <= 1000, percentage of cases that mergeSort() is faster than insertionSort() is 88.61%
        For all n <= 1500, percentage of cases that mergeSort() is faster than insertionSort() is 92.41%
        Percentage of cases passed: 100.00%
        From n = 127 to 1500, mergeSort() runs consistently faster than insertionSort()
Attempt no.5
        For all n <= 100, percentage of cases that mergeSort() is faster than insertionSort() is 1.98%
        For all n <= 500, percentage of cases that mergeSort() is faster than insertionSort() is 76.25%
        For all n <= 1000, percentage of cases that mergeSort() is faster than insertionSort() is 88.11%
        For all n <= 1500, percentage of cases that mergeSort() is faster than insertionSort() is 92.07%
        Percentage of cases passed: 100.00%
        From n = 186 to 1500, mergeSort() runs consistently faster than insertionSort()
[135, 171, 123, 127, 186, 118, 176, 148, 125, 149, 190, 163, 152, 152, 145]
Approximately n = 150, mergeSort is the faster algorithm
```

Result for case of n=10**4, maxValue=10**7:

```
For all n <= 100, percentage of cases that mergeSort() is faster than insertionSort() is 1.98%
For all n <= 500, percentage of cases that mergeSort() is faster than insertionSort() is 76.45%
For all n <= 1000, percentage of cases that mergeSort() is faster than insertionSort() is 88.21%
For all n <= 1500, percentage of cases that mergeSort() is faster than insertionSort() is 92.14%
For all n <= 2000, percentage of cases that mergeSort() is faster than insertionSort() is 94.10%
For all n <= 2500, percentage of cases that mergeSort() is faster than insertionSort() is 95.28%
For all n <= 3000, percentage of cases that mergeSort() is faster than insertionSort() is 96.07%
For all n <= 3500, percentage of cases that mergeSort() is faster than insertionSort() is 96.63%
For all n <= 4000, percentage of cases that mergeSort() is faster than insertionSort() is 97.05%
For all n <= 4500, percentage of cases that mergeSort() is faster than insertionSort() is 97.38%
For all n <= 5000, percentage of cases that mergeSort() is faster than insertionSort() is 97.64%
For all n <= 5500, percentage of cases that mergeSort() is faster than insertionSort() is 97.85%
For all n <= 6000, percentage of cases that mergeSort() is faster than insertionSort() is 98.03%
For all n <= 6500, percentage of cases that mergeSort() is faster than insertionSort() is 98.18%
For all n <= 7000, percentage of cases that mergeSort() is faster than insertionSort() is 98.31%
For all n <= 7500, percentage of cases that mergeSort() is faster than insertionSort() is 98.43%
For all n <= 8000, percentage of cases that mergeSort() is faster than insertionSort() is 98.53%
For all n <= 8500, percentage of cases that mergeSort() is faster than insertionSort() is 98.61%
For all n <= 9000, percentage of cases that mergeSort() is faster than insertionSort() is 98.69%
For all n <= 9500, percentage of cases that mergeSort() is faster than insertionSort() is 98.76%
Percentage of cases passed: 100.00%
From n = 158 to 9999, mergeSort() runs consistently faster than insertionSort()
```

Question 5.

To estimate the minimum value **n_min** that ***mergeSort()*** starts running faster, I run test with n=1501, and integer in range [0,10**4] fifteen times (the first 5 attempts are printed above). **n_min** is the average of n that ***mergeSort()*** starts running faster consistently (more than 100 consecutive values of n).

Based on the simulations, approximately **n = 150**, mergeSort is the faster algorithm.

Question 6.

The algorithms are implemented in ***6_Strassen_square_matrices***

I implemented Strassen's algorithm in two function ***Strassen()*** and ***Strassen_use_numpy_matrices()*** with an assumption that n, the number of row/col in the matrix, is the power of 2.

For ***Strassen(),*** I store the matrix in Python list. The problem of this is the process of splitting a matrix ***split_matrices()*** is no longer O(1). It is now O(n^2), when using list comprehension.

Instead, for ***Strassen_use_numpy_matrices(),*** I use ***numpy.array*** to store the matrix. Now by using indices, the process of splitting ***split_matrices_use_numpy()*** runs in O(1). The difference in speed can be tested with ***test_runtime_split().*** As you can see in the box below, for a random matrix of size (10,10), ***split_matrices_use_numpy()*** ran around 1.3 times faster.

To show you my attempt to solve the problem, I still implemented the algorithm with Python list with function to split a matrix, calculate sum and difference of 2 matrices and product with Strassen's algorithm. When using numpy.array, I use their ***numpy.add*** and ***numpy.substract*** instead.

To test the accuracy of ***Strassen()*** and ***Strassen_use_numpy_matrices()***, I compare results again one from ***np.dot()***. The runtime is not concerned in this test, therefore, for each case, I create 2 random matrices of size n (n=2**i, 0≤i≤4). Each number in the matrix has value smaller than 10. As you can see in the result box, all test cases passed.

Result from test_runtime_split(10) and multiple_test_cases(100)

```
CPU time of split_matrices(): 4.696846008300781e-05
CPU time of split_matrices_use_numpy(): 3.0994415283203125e-06
Percentage of test cases passed: 100.00%
```