I. Question b 2

In 11/2 slides p.55, we use AspectJ to make the Fibonacci program much faster.

In Python, the decorators proposed by PEP 318 can do similar thing as what the around advice does. Please first write a normal Fibonacci program in Python, use decorators to make it fast, and then simply compare their performance (with and w/o decorators)

II. Tested Fibonacci ($F_n = F_{n-1} + F_{n-2}, F_1 = 1, F_0 = 0$)

a) Fibonacci using recursion 1

```
def fib(n):

    if n == 0: return 0
    if n == 1: return 1
    return fib(n-2) + fib(n-1)
```

c) Fibonacci using recursion 2

```
def fib3(n):

    if n < 2:
        return n
    return fib3(n-2) + fib3(n-1)
```

b) Fibonacci (a) using decorators

```
class memoize:

  def __init__(self, function):
    self.function = function
    self.store = {}

  def __call__(self, *args):
    key = (args)
    if not key in self.store:
      self.store[key] = self.function(*args)
    return self.store[key]

@memoize
def fib2(n):

    if n == 0: return 0
    if n == 1: return 1
    return fib2(n-2) + fib2(n-1)
```

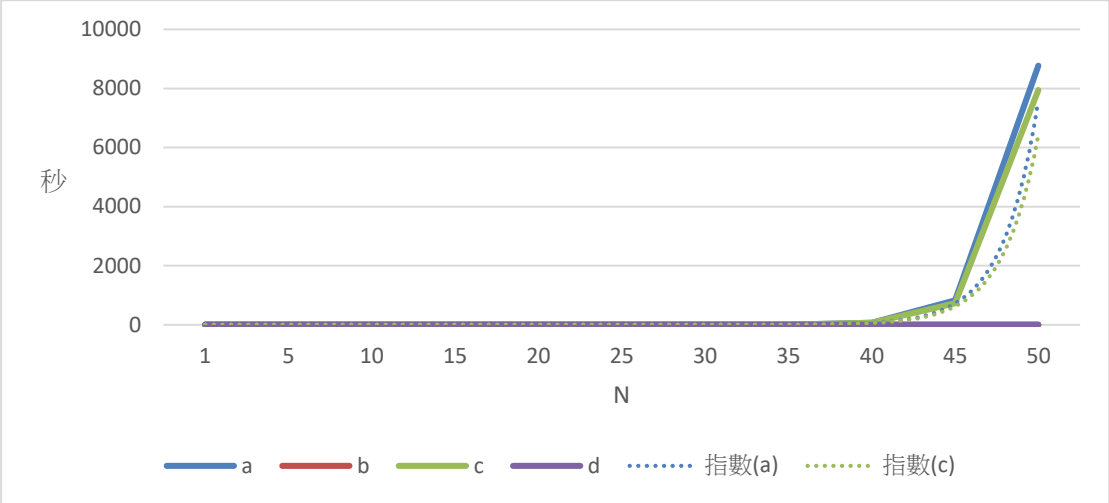
d) Fibonacci (c) using decorators

```
class memoize:

  def __init__(self, function):
    self.function = function
    self.store = {}

  def __call__(self, *args):
    key = (args)
    if not key in self.store:
      self.store[key] = self.function(*args)
    return self.store[key]

@memoize
def fib4(n):

    if n < 2:
        return n
    return fib4(n-2) + fib4(n-1)
```

III. Tested Results (In seconds)

N	(a)	(b)	(c)	(d)
1	0.0000007685	0.0000034584	0.0000011528	0.0000042269
5	0.0000046112	0.0000134494	0.0000042269	0.0000111438
10	0.0000407324	0.0000222875	0.0000368897	0.0000207505
15	0.0004276899	0.0000299729	0.0003915687	0.0000622514
20	0.0046200496	0.0000376582	0.0042811256	0.0000361212
25	0.0516839952	0.0000526447	0.0477148949	0.0000557188
30	0.5628095036	0.0000576401	0.5263063850	0.0000572559
35	6.5207049159	0.0000745479	5.9227837269	0.0000714738
40	**70.666974465**	**0.0000749322**	**64.960184495**	**0.0000864602**
45	830.53280511	0.0000903029	725.95829006	0.0000879973
50	8770.8164254	0.0000956826	7950.8082228	0.0000937613

IV. Graph (b and d nearly overlay each other at bottom line)



V. Code Explanation

```
timer_start = timeit.default_timer()
fib(x)
timer_end = timeit.default_timer()
print(f'{timer_end - timer_start:.10f}')
timer_start = timeit.default_timer()
fib2(x)
timer_end = timeit.default_timer()
print(f'{timer_end - timer_start:.10f}')
timer_start = timeit.default_timer()
fib3(x)
timer_end = timeit.default_timer()
print(f'{timer_end - timer_start:.10f}')
timer_start = timeit.default_timer()
fib4(x)
timer_end = timeit.default_timer()
print(f'{timer_end - timer_start:.10f}')
```

Using timeit.default_timer() to calculate the performance of each function.

```
class memoize:

  def __init__(self, function):
    self.function = function
    self.store = {}

  def __call__(self, *args):
    key = (args)
    if not key in self.store:
      self.store[key] = self.function(*args)
    return self.store[key]
```

Class memoize:

First link fib with memoize(@memoize). Then initialize function and store. When called, if the key is not stored, store the key. Else return the value stored with the key

VI. Conclusion

What I did in the experiment is: compare recursive Fibonacci with and without decorators, a memorize technique. I chose to write two kinds of Fibonacci, one as (a) shows and the other as (c) shows.

To begin with, we can take a close look at the results of (a) and (b). It is undoubtedly that (b) is greatly faster than (a). When at N = 40, (a) already used up to approximately 70 seconds whereas (b) used up not even close to 1 second. As for (c) and (d), it is almost the same. We can see (c) used approximately 65 seconds while (d) used 0.00008 seconds when N = 40.

In conclusion, it is obvious that Fibonacci using decorators is faster since it stores the results already calculated so that it can use those results directly later on instead of re-calculate and waste more time.