Programming Assignment 1

Analyzing the Fibonacci Sequence

Daniel Throop

January 2021

1 Description of Code

I begin my code by initializing the necessary data structures to record my averages that the script outputs. I decided to use a module called 'xlsxwriter' in order to automatically filter the output of the code for both the recursive and non-recursive version of the Fibonacci sequence directly to Excel. I also import the necessary modules to record the time of the trials and rounding of certain results (Listing 1). From here, I define both 'rec_fibonnaci(n)' and 'good_fibonnaci(n)' the later implying the non-recursive implementation (Listing 2). Finally, I run twenty four trials for both fibonacci implementations taking great care to not effect their run-time using the xlsxwriter module (I confirmed this to be true). The trials are automatically formatted into excel to be further processed (Listing 3).

2 Listings of Code

```
1 import time
2 import math
3 import xlsxwriter
4
5 # intialize workbooks
6 outWorkbook = xlsxwriter.Workbook("out.xlsx")
7 recFibSheet = outWorkbook.add_worksheet()
8 goodFibSheet = outWorkbook.add_worksheet()
```

```
9
10 # writing headers
11 recFibSheet.write(0, 0, "rec_fib(n)")
12 goodFibSheet.write(0, 0, "good_fib(n)")
13
14 # intialzing of both vertical and horizontal colums in excel
15 for i in range(1, 25):
16   recFibSheet.write(0, i, i)
17   recFibSheet.write(i, 0, i)
18   goodFibSheet.write(0, i, i)
19   goodFibSheet.write(i, 0, i)
20
21 # intializes dictionary to gather durations of fib() runs
22 def initDict(n):
23   dummy = {}
24   for i in range(1, n):
25     dummy[i] = []
26   return dummy
27
28 durations = initDict(25)
29 averages = initDict(25)
```

Listing 1: **Importing of modules and initializing of data structures**

```
1
2  # recursive fibonacci
3  def rec_fibonnaci(i):
4    if i < 2:
5      return i
6    else:
7      return rec_fibonnaci(i-1)+rec_fibonnaci(i-2)
8
9  # non-recursive fibonacci
10 def good_fibonnaci(i):
11   pre = 0
12   curr = 1
13   if i == 0:
14     return 0
15   elif i == 1:
16     return 1
17   else:
18     for _ in range(i - 1):
19       curr, pre = curr+pre, curr
20     return curr
```

Listing 2: **Fibonacci Definitions**

```
# recursive fibonacci trials - 24 total
for trial in range(1, 25):

  # recording time durations (nsecs)
  for i in range(1, 25):
    total = 0
    print(i)
    for j  in range(20):
      start_time = time.perf_counter_ns()
      for _ in range(50):
        x = rec_fibonnaci(i)
      duration = (time.perf_counter_ns() - start_time)/50
      print("{:12.10f}".format(duration))
      durations[i].append(duration)

    # storing the average of the 20 runs from fib(n)

    # sum the 20 runs of fib(n)
    for j in range(len(durations[1])):
      total = total + durations[i][j]
    # take the average of the total
    total = total/20.0
    # store away average to be put in excel
    averages[i] = round(total, 4 - (int(math.floor(math.
   log10(abs(total)))) - 1))
```

```
25      print("\n")
26
27    # write averages to excel sheet
28    for row in range(1, len(averages) + 1):
29      recFibSheet.write(row, trial, str(averages[row]))
30
31    # clear and re-intialize data-structures
32    durations.clear()
33    averages.clear()
34    durations = initDict(25)
35    averages = initDict(25)
36
37  # non-recursive fibonacci trials - 24 total
38  for trial in range(1, 25):
39    # recording time durations (nsecs)
40    for i in range(1, 25):
41      total = 0
42      print(i)
43      for j  in range(20):
44        start_time = time.perf_counter_ns()
45        for _ in range(50):
46          x = good_fibonnaci(i)
47        duration = (time.perf_counter_ns() - start_time)/50
48        print("{:12.10f}".format(duration))
49        durations[i].append(duration)
```

```
50
51      # storing the average of the 20 runs from fib(n)
52
53      # sum the 20 runs of fib(n)
54      for j in range(len(durations[1])):
55        total = total + durations[i][j]
56      # take the average of the total
57      total = total/20.0
58      # store away average to be put in excel
59      averages[i] = round(total, 4 - (int(math.floor(math.
        log10(abs(total)))) - 1))
60      print("\n")
61
62    # write averages to excel sheet
63    for row in range(1, len(averages) + 1):
64      goodFibSheet.write(row, trial, str(averages[row]))
65
66    # clear and re-intialize data-structures
67    durations.clear()
68    averages.clear()
69    durations = initDict(25)
70    averages = initDict(25)
71
72  # closing workbook
```

```
outWorkbook.close()
```

Listing 3: **Running of Trials**

3 Results and Graphical Analysis

Value	Recrusive Fibonacci (nsec)	Good Fibonacci (nsec)
1	202.24 ±29.186	177.933 ±21.093
2	581.602 ±61.903	444.191 ±57.626
3	898.314 ±107.369	469.368 ±46.011
4	1107.635 ±58.598	499.427 ±22.341
5	1700.789 ±27.111	527.592 ±13.154
6	2785.763 ±51.268	577.686 ±19.912
7	4466.812 ±38.321	607.537 ±8.715
8	7236.813 ±68.84	643.667 ±24.056
9	11757.179 ±121.752	675.559 ±13.281
10	19723.467 ±322.535	711.708 ±11.857
11	31382.675 ±325.889	755.263 ±55.015
12	50497.538 ±265.645	778.88 ±20.511
13	81514.471 ±627.053	819.609 ±36.841
14	131859.875 ±3576.055	857.669 ±16.944
15	211982.875 ±1902.805	904.582 ±14.95
16	340712.125 ±2986.543	966.654 ±9.11
17	548667 ±3816.186	1017.213 ±15.619
18	907435.292 ±77989.629	1088.437 ±45.495
19	1428850.833 ±10429.325	1131.612 ±45.443
20	2286828.333 ±7766.445	1180.881 ±35.057
21	3679702.5 ±15697.394	1231.952 ±65.472
22	5927434.167 ±13145.598	1260.149 ±25.698
23	9793993.75 ±936934.923	1301.822 ±8.631
24	15826258.333 ±1476100	1389.601 ±117.67

Table 1: Trial results in Python 3.9

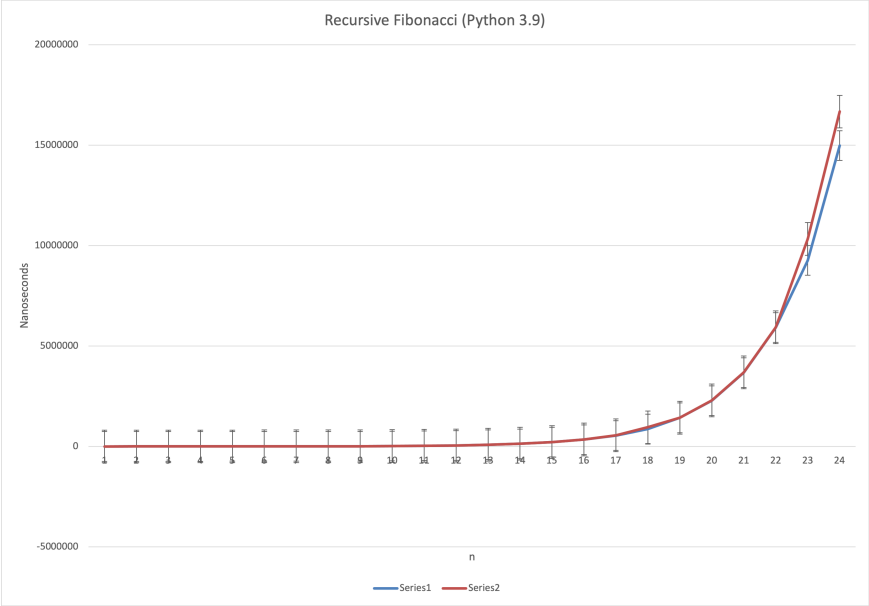

Figure 1: Recursive Fibonacci Trials

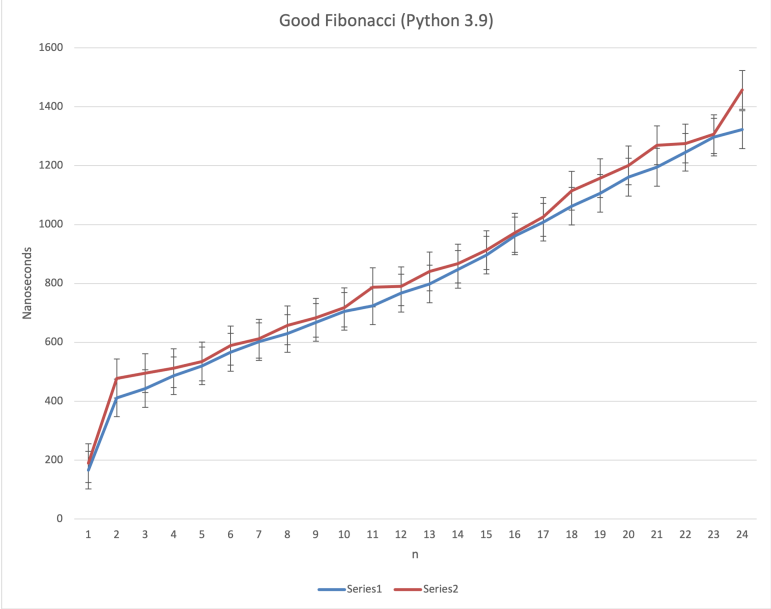

Figure 2: Non-recursive Fibonacci Trials

4 Discussion of Results

These results confirm our initial assumption of recursive function calling - in the case of the Fibonacci sequence it gets much slower upon larger values of "n". As we can see from the table and the graph, the recursive version grows exponentially slower in terms of nanoseconds. This clearly makes sense taking into account the number of recursive calls being made each function call (see Homework 1 for recurrence relation). Running these trials on a Macbook Pro 2020 with a 1.4 GHz Quad-Core Intel Core i5 and 16 GB 2133 MHz LPDDR memory, the upper limit of reasonable computation seems to be at about "n = 24" for the recursive version.

In terms of the non-recursive version or "good-fibonacci", we see a much different result and our assumptions confirmed. Storing the results of the function calls in temporary variables leads to a more linear result (see Figure 2). We could easily have computed to a much larger "n" but decided on twenty four trials for the sake of scope and comparison. At "n = 24" we see a computation time of roughly 1389.601 nanoseconds (see Table 1). A massive improvement from roughly 15,800,000 nanoseconds at "n = 24" for the recursive version.

For error, I am not necessarily sure what to attribute the large deviations from my averages. Besides cold cache misses to begin, I'm assuming these deviations are rather normal for the python compiler. The deviations grow proportionally with larger results. In future assignment's I plan to look for ways

to minimize these errors.