| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Program | | fib1 | | | fib2 | | | sum1 | | | sum2 | | | sum3 | | | sum4 | | |
| | | Input | 40.00 | 45.00 | 50.00 | 40.00 | 45.00 | 50.00 | 1,000,000.00 | 10,000,000.00 | 100,000,000.00 | 1,000,000.00 | 10,000,000.00 | 100,000,000.00 | 1,000,000.00 | 10,000,000.00 | 100,000,000.00 | 1,000,000.00 | 10,000,000.00 | 100,000,000.00 |
| Event | | | | | | | | | | | | | | | | | | | | |
| branch-instructions | | | 1,426,932,409.0( | 15,826,575,440.( | 175,510,190,644 | 10,509.00 | 10,515.00 | 10,522.00 | 2,961,128.00 | 30,205,556.00 | 300,735,133.00 | 452,928.00 | 9,982,992.00 | 100,000,960.00 | 2,974.00 | 2,981.00 | 0.00 | 459,686.00 | 8,617,073.00 | 99,803,744.00 |
| branch-misses | | | 129,268.00 | 1,413,197.00 | 16,057,475.00 | 829.00 | 826.00 | 823.00 | 924,481.00 | 10,109,044.00 | 99,702,586.00 | 812.00 | 705.00 | 741.00 | 537.00 | 497.00 | 514.00 | 839.00 | 1,586.00 | 1,060.00 |
| cpu-cycles | | | 2,536,875,704.0( | 28,102,521,428.( | 311,733,360,364 | 93,212.00 | 88,885.00 | 122,769.00 | 45,056,385.00 | 502,930,344.00 | 5,005,324,192.0( | 8,483,042.00 | 84,476,637.00 | 844,468,774.00 | 69,837.00 | 64,513.00 | 64,991.00 | 1,153,253.00 | 10,222,479.00 | 100,083,929.00 |
| instructions | | | 6,499,341,606.0( | 72,033,577,287.( | 798,936,676,568 | 39,069.00 | 39,147.00 | 39,226.00 | 18,805,461.00 | 190,012,974.00 | 1,900,490,898.0( | 7,011,949.00 | 69,955,054.00 | 699,265,946.00 | 11,946.00 | 11,997.00 | 12,052.00 | 5,038,734.00 | 50,038,791.00 | 500,250,648.00 |
| cache-misses | | | 125.00 | 4,279.00 | 21,891.00 | 72.00 | 482.00 | 1,471.00 | 394,810.00 | 7,141,867.00 | 74,888,340.00 | 1,256.00 | 397.00 | 376.00 | 89.00 | 275.00 | 274.00 | 1,392.00 | 1,014.00 | 1,674.00 |
| L1-dcache-load-misses | | | 10,535.00 | 39,288.00 | 669,876.00 | 658.00 | 550.00 | 533.00 | 732,847.00 | 7,613,774.00 | 75,850,613.00 | 578.00 | 538.00 | 503.00 | 527.00 | 422.00 | 429.00 | 757.00 | 547.00 | 683.00 |
| L1-dcache-loads | | | 1,923,009,912.0( | 21,336,971,032.( | 236,613,249,271 | 7,865.00 | 7,899.00 | 7,935.00 | 5,915,171.00 | 59,545,300.00 | 600,978,568.00 | 5,002,442.00 | 49,882,461.00 | 500,093,844.00 | 2,432.00 | 2,438.00 | 2,446.00 | 7,592.00 | 7,600.00 | 8,972.00 |
| L1-dcache-stores | | | 1,323,957,544.0( | 14,689,640,921.( | 162,925,115,409 | 2,169.00 | 2,191.00 | 2,213.00 | 3,962,559.00 | 40,165,264.00 | 400,195,866.00 | 2,001,576.00 | 19,953,465.00 | 199,997,124.00 | 1,573.00 | 1,577.00 | 1,581.00 | 2,007.00 | 2,011.00 | 2,361.00 |
| L1-icache-load-misses | | | 4,255.00 | 17,246.00 | 160,685.00 | 2,294.00 | 2,226.00 | 2,335.00 | 8,184.00 | 12,830.00 | 181,447.00 | 2,414.00 | 2,764.00 | 2,932.00 | 2,266.00 | 2,106.00 | 2,126.00 | 2,622.00 | 2,157.00 | 2,692.00 |
| branch-load-misses | | | 128,644.00 | 1,412,201.00 | 16,056,928.00 | 1,496.00 | 1,892.00 | 1,434.00 | 1,031,275.00 | 9,951,413.00 | 99,987,257.00 | 321.00 | 0.00 | 132.00 | 3,009.00 | 2,041.00 | 591.00 | 403.00 | 119.00 | 122.00 |
| branch-loads | | | 1,427,107,786.0( | 15,825,643,800.( | 175,514,948,391 | 29,763.00 | 30,921.00 | 34,517.00 | 3,009,538.00 | 30,213,846.00 | 300,964,619.00 | 1,209,530.00 | 10,062,811.00 | 99,990,185.00 | 22,718.00 | 26,734.00 | 19,527.00 | 1,349,949.00 | 10,379,979.00 | 100,535,907.00 |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | Some events weren't counted. Try disabling the NMI watchdog: | | | | No need, just don't run all at once | | | | | | | | | | |
| | | | | | | echo 0 > /proc/sys/kernel/nmi_watchdog | | | | | | | | | | | | | | |
| | | | | | | perf stat ... | | | | | | | | | | | | | | |
| | | | | | | echo 1 > /proc/sys/kernel/nmi_watchdog | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | manage locality in 3d array | | | | | | | | | | | | | | |
| | | | | | | matrix mulitplies, all of AI Locality example | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | use cache misses and l1dchache load misses as a proxy to answer 3 | | | | | | | | | | | | | | | | | | |

# Task 3

## 3.1 and 3.2:

| Part 3 | Input | | | | | |
|---|---|---|---|---|---|---|
| time ./. | 100000000 | | | | | |
| sum3 | real 0m0.001s | user 0m0.000s | sys 0m0.001s | | real is wall-clock time | |
| sum4 | real 0m0.031s | user 0m0.029s | sys 0m0.000s | | ranked from fastest down to slowest | |
| sum2 | real 0m0.235s | user 0m0.233s | sys 0m0.000s | | | |
| sum1 | real 0m3.458s | user 0m1.278s | sys 0m2.171s | | | |
| | | | | | | |
| | | | | | | |
| fib2 | real 0m0.003s | user 0m0.000s | sys 0m0.001s | | | |
| fib1 | real 1m25.188s | user 1m24.958s | sys 0m0.001s | | | |

## 3.3 and 3.4:

3.3 Big O runtime complexity determined by looking at the real time for each program for a sequence of inputs.
3.4 Program nature described based on the outputs of programs from part 2, especially call_graph, as well as Big O runtime.

### **Sum1**:

Big O runtime appears to be **O(n)** as runtime increases approximately proportional to input size. With smaller inputs the runtime of the sum part of the program is dominated by the rest of the program. Appears to be a **recursive** program based on call_graph. Memory complexity is **O(n)** based on call_graph.out since no memory is allocated for the input, so the memory complexity is determined by the recursive nature of the program, so each additional function call as n increases linearly will increase the memory linearly.

Here is a snippet of call_graph.out:

```
sum(100,...)
    sum(99,...)
     sum(98,...)
      sum(97,...)
       sum(96,...)
```

Here are the results of different inputs used:

```
Singularity> time ./sum1 100
5050

real    0m0.001s
user    0m0.000s
sys     0m0.001s
Singularity> time ./sum1 1000
500500

real    0m0.001s
user    0m0.001s
sys     0m0.000s
Singularity> time ./sum1 10000
50005000

real    0m0.001s
user    0m0.001s
sys     0m0.000s
Singularity> time ./sum1 100000
5000050000

real    0m0.005s
user    0m0.001s
sys     0m0.004s
Singularity> time ./sum1 1000000
500000500000

real    0m0.034s
user    0m0.010s
sys     0m0.023s
Singularity> time ./sum1 10000000
50000005000000

real    0m0.347s
```

user 0m0.123s
sys 0m0.223s
Singularity> time ./sum1 100000000
5000000050000000

real 0m4.794s
user 0m1.501s
sys 0m3.263s
Singularity> time ./sum1 1000000000
500000000500000000

real 0m39.258s
user 0m14.025s
sys 0m25.104s

## Sum2:

Big O runtime appears to be **O(1)** as runtime is pretty constant until the inputs get massive. It is **not an iterative or recursive program** based on call_graph. It just uses one call to a sum function that contains a mathematical formula to calculate the answer. Memory complexity is probably **O(1)** as memory is allocated a constant amount.

Here is a snippet of call_graph.out:
main(2,...)
 atoi(140721472721773,...)
  __strtol(140721472721773,...)
   ____strtol_l_internal(140721472721773,...)
  sum(100,...)
  __printf(4854052,...)

Here is a snippet of call_graph.out for memory complexity:
 __libc_malloc(1024,...)
  _int_malloc(7120896,...)
   malloc_consolidate(7120896,...)

Here are the results of different inputs used:

Singularity> time ./sum2 1000000000
500000000500000000

real 0m2.379s
user 0m2.344s
sys 0m0.015s

```
Singularity> time ./sum2 100000000
5000000050000000

real	0m0.239s
user	0m0.235s
sys 0m0.002s
Singularity> time ./sum2 10000000
50000005000000

real	0m0.024s
user	0m0.024s
sys 0m0.000s
Singularity> time ./sum2 1000000
500000500000

real	0m0.003s
user	0m0.003s
sys 0m0.000s
Singularity> time ./sum2 100000
5000050000

real	0m0.001s
user	0m0.001s
sys 0m0.000s
Singularity> time ./sum2 10000
50005000

real	0m0.001s
user	0m0.000s
sys 0m0.000s
Singularity> time ./sum2 1000
500500

real	0m0.001s
user	0m0.000s
sys 0m0.001s
Singularity> time ./sum2 100
5050

real	0m0.001s
user	0m0.001s
sys 0m0.000s
```

## **Sum3**:

Big O runtime appears to definitely be **O(1)** as runtime is pretty constant until the inputs get massive. It is **not an iterative or recursive program** based on call_graph. It just uses one call to a sum function that contains a mathematical formula to calculate the answer. Memory complexity is probably **O(1)** as memory is allocated a constant amount.
Here is a snippet of call_graph.out for program description:

```
main(2,...)
 atoi(140727603663725,...)
  __strtol(140727603663725,...)
   ____strtol_l_internal(140727603663725,...)
  sum(100,...)
  __printf(4854056,...)
```

Here is a snippet of call_graph.out for memory complexity:

```
        __libc_malloc(1024,...)
         _int_malloc(7120896,...)
          malloc_consolidate(7120896,...)
```

Here are the results of different inputs used:

```
Singularity> time ./sum3 100
5050

real	0m0.001s
user	0m0.000s
sys	0m0.000s
Singularity> time ./sum3 1000
500500

real	0m0.001s
user	0m0.000s
sys	0m0.000s
Singularity> time ./sum3 10000
50005000

real	0m0.001s
user	0m0.000s
sys	0m0.001s
Singularity> time ./sum3 100000
5000050000

real	0m0.001s
user	0m0.000s
sys	0m0.000s
```

```
Singularity> time ./sum3 1000000
500000500000

real	0m0.001s
user	0m0.000s
sys 0m0.000s
Singularity> time ./sum3 10000000
50000005000000

real	0m0.001s
user	0m0.000s
sys 0m0.000s
Singularity> time ./sum3 100000000
5000000050000000

real	0m0.001s
user	0m0.000s
sys 0m0.000s
Singularity> time ./sum3 1000000000
500000000500000000

real	0m0.001s
user	0m0.000s
sys 0m0.000s
Singularity> time ./sum3 10000000000
-
994142228124135936

real	0m0.001s
user	0m0.000s
sys 0m0.000s
Singularity> time ./sum3 100000000000
739026696784278528

real	0m0.001s
user	0m0.000s
sys 0m0.001s
```

## **Sum4**:

Big O runtime appears to definitely be **O(1)** as runtime is pretty constant until the inputs get quite large, while the real time jumps up when the input gets quite large, it generally stays constant and does not increase linearly with input size. It is **not an iterative or recursive program** based on call_graph, and the runtime complexity. It doesn't even make a call to a sum helper function that contains a mathematical formula to calculate the answer, and just computes

the result directly after parsing the input. Memory complexity is probably **O(1)** as memory is allocated a constant amount.
Here is call_graph.out for program description and memory complexity:

```
Singularity> cat call_graph.out
 main(2,...)
  strtoll(140734076060525,...)
   ____strtoll_l_internal(140734076060525,...)
   ___printf_chk(1,...)
    __vfprintf_internal(4981536,...)
     __strchrnul_avx2(4804612,...)
     _IO_file_xsputn(4981536,...)
     _itoa_word(5050,...)
     _IO_file_xsputn(4981536,...)
      _IO_new_file_overflow(4981536,...)
       _IO_doallocbuf(4981536,...)
        _IO_file_doallocate(4981536,...)
         _IO_file_stat(4981536,...)
          __fxstat(1,...)
          malloc(1024,...)
           _int_malloc(4982688,...)
          _IO_setb(4981536,...)
        _IO_do_write(4981536,...)
        _IO_default_xsputn(4981536,...)
         _IO_new_file_overflow(4981536,...)
         _IO_new_file_overflow(4981536,...)
         _IO_new_file_overflow(4981536,...)
         _IO_new_file_overflow(4981536,...)
       __strchrnul_avx2(4804615,...)
       _IO_file_xsputn(4981536,...)
        __mempcpy_avx_unaligned_erms(6872484,...)
        _IO_new_file_overflow(4981536,...)
         _IO_do_write(4981536,...)
          _IO_file_write(4981536,...)
           __libc_write(1,...)
     exit(0,...)
```

Here are the results of different inputs used:

```
Singularity> time ./sum4 100000000000
7390266967784278528

real    0m0.340s
user    0m0.339s
sys 0m0.000s
Singularity> time ./sum4 10000000000
```

```
994142228124135936

real	0m0.395s
user	0m0.393s
sys 0m0.000s
Singularity> time ./sum4 1000000000
500000000500000000

real	0m0.283s
user	0m0.277s
sys 0m0.002s
Singularity> time ./sum4 100000000
5000000050000000

real	0m0.029s
user	0m0.027s
sys 0m0.001s
Singularity> time ./sum4 10000000
50000005000000

real	0m0.003s
user	0m0.003s
sys 0m0.000s
Singularity> time ./sum4 1000000
500000500000

real	0m0.001s
user	0m0.001s
sys 0m0.000s
Singularity> time ./sum4 100000
5000050000

real	0m0.001s
user	0m0.000s
sys 0m0.000s
Singularity> time ./sum4 10000
50005000

real	0m0.001s
user	0m0.000s
sys 0m0.001s
Singularity> time ./sum4 1000
500500
```

```
real    0m0.001s
user    0m0.000s
sys 0m0.001s
Singularity> time ./sum4 100
5050

real    0m0.001s
user    0m0.001s
sys 0m0.000s
```

## Fib1:

Big O runtime appears to definitely be **O($2^n$)** as the runtime grows exponentially. It is **a recursive program** based on call_graph, and we can tell it is a naive recursive program since it recomputes the same fibonacci numbers, which is why Fib2 is so much faster. Memory complexity is **O(n)** as memory is allocated based on n by the naive recursive nature of this program.

Here is call_graph.out for program description and memory complexity:

```
Singularity> cat call_graph.out
 main(2,...)
  atoi(140724212626286,...)
   strtoll(140724212626286,...)
    ____strtol_l_internal(140724212626286,...)
   fib(10,...)
    fib(9,...)
     fib(8,...)
      fib(7,...)
       fib(6,...)
        fib(5,...)
         fib(4,...)
          fib(3,...)
           fib(2,...)
            fib(1,...)
            fib(0,...)
           fib(1,...)
          fib(2,...)
           fib(1,...)
           fib(0,...)
         fib(3,...)
          fib(2,...)
           fib(1,...)
           fib(0,...)
          fib(1,...)
```

```
  fib(4,...)
   fib(3,...)
    fib(2,...)
     fib(1,...)
     fib(0,...)
    fib(1,...)
   fib(2,...)
    fib(1,...)
    fib(0,...)
 fib(5,...)
  fib(4,...)
   fib(3,...)
    fib(2,...)
     fib(1,...)
     fib(0,...)
    fib(1,...)
   fib(2,...)
    fib(1,...)
    fib(0,...)
  fib(3,...)
   fib(2,...)
    fib(1,...)
    fib(0,...)
   fib(1,...)
fib(6,...)
 fib(5,...)
  fib(4,...)
   fib(3,...)
    fib(2,...)
     fib(1,...)
     fib(0,...)
    fib(1,...)
   fib(2,...)
    fib(1,...)
    fib(0,...)
  fib(3,...)
   fib(2,...)
    fib(1,...)
    fib(0,...)
   fib(1,...)
 fib(4,...)
  fib(3,...)
   fib(2,...)
    fib(1,...)
```

```
     fib(0,...)
    fib(1,...)
   fib(2,...)
    fib(1,...)
    fib(0,...)
fib(7,...)
 fib(6,...)
  fib(5,...)
   fib(4,...)
    fib(3,...)
     fib(2,...)
      fib(1,...)
      fib(0,...)
     fib(1,...)
    fib(2,...)
     fib(1,...)
     fib(0,...)
   fib(3,...)
    fib(2,...)
     fib(1,...)
     fib(0,...)
    fib(1,...)
  fib(4,...)
   fib(3,...)
    fib(2,...)
     fib(1,...)
     fib(0,...)
    fib(1,...)
   fib(2,...)
    fib(1,...)
    fib(0,...)
 fib(5,...)
  fib(4,...)
   fib(3,...)
    fib(2,...)
     fib(1,...)
     fib(0,...)
    fib(1,...)
   fib(2,...)
    fib(1,...)
    fib(0,...)
  fib(3,...)
   fib(2,...)
    fib(1,...)
```

```
     fib(0,...)
    fib(1,...)
fib(8,...)
 fib(7,...)
  fib(6,...)
   fib(5,...)
    fib(4,...)
     fib(3,...)
      fib(2,...)
       fib(1,...)
       fib(0,...)
      fib(1,...)
     fib(2,...)
      fib(1,...)
      fib(0,...)
    fib(3,...)
     fib(2,...)
      fib(1,...)
      fib(0,...)
     fib(1,...)
   fib(4,...)
    fib(3,...)
     fib(2,...)
      fib(1,...)
      fib(0,...)
     fib(1,...)
    fib(2,...)
     fib(1,...)
     fib(0,...)
  fib(5,...)
   fib(4,...)
    fib(3,...)
     fib(2,...)
      fib(1,...)
      fib(0,...)
     fib(1,...)
    fib(2,...)
     fib(1,...)
     fib(0,...)
   fib(3,...)
    fib(2,...)
     fib(1,...)
     fib(0,...)
    fib(1,...)
```

```
  fib(6,...)
   fib(5,...)
    fib(4,...)
     fib(3,...)
      fib(2,...)
       fib(1,...)
       fib(0,...)
      fib(1,...)
     fib(2,...)
      fib(1,...)
      fib(0,...)
    fib(3,...)
     fib(2,...)
      fib(1,...)
      fib(0,...)
     fib(1,...)
   fib(4,...)
    fib(3,...)
     fib(2,...)
      fib(1,...)
      fib(0,...)
     fib(1,...)
    fib(2,...)
     fib(1,...)
     fib(0,...)
printf(4804612,...)
 __vfprintf_internal(4981536,...)
  __strchrnul_avx2(4804612,...)
  _IO_file_xsputn(4981536,...)
  _itoa_word(55,...)
  _IO_file_xsputn(4981536,...)
   _IO_new_file_overflow(4981536,...)
    _IO_doallocbuf(4981536,...)
     _IO_file_doallocate(4981536,...)
      _IO_file_stat(4981536,...)
       __fxstat64(1,...)
       __malloc(1024,...)
        _int_malloc(4982688,...)
       _IO_setb(4981536,...)
     _IO_new_do_write(4981536,...)
     _IO_default_xsputn(4981536,...)
      _IO_new_file_overflow(4981536,...)
      _IO_new_file_overflow(4981536,...)
    __strchrnul_avx2(4804615,...)
```

```
      _IO_file_xsputn(4981536,...)
       __mempcpy_avx_unaligned_erms(24366498,...)
       _IO_new_file_overflow(4981536,...)
        _IO_new_do_write(4981536,...)
         _IO_new_file_write(4981536,...)
          __libc_write(1,...)
    exit(0,...)
COS375 pin tool call graph
```

Here are the results of different inputs used:

```
Singularity> time ./fib1 10
55

real    0m0.001s
user    0m0.000s
sys 0m0.000s
Singularity> time ./fib1 20
6765

real    0m0.001s
user    0m0.001s
sys 0m0.000s
Singularity> time ./fib1 30
832040

real    0m0.007s
user    0m0.006s
sys 0m0.000s
Singularity> time ./fib1 40
102334155

real    0m0.709s
user    0m0.705s
sys 0m0.002s
Singularity> time ./fib1 45
1134903170

real    0m7.878s
user    0m7.813s
sys 0m0.021s
Singularity> time ./fib1 50
12586269025

real    1m27.167s
```

user 1m26.713s
sys 0m0.105s

## Fib2:

Big O runtime appears to definitely be **O(1)** as the runtime stays constant even up to inputs of 1000. It is **not a recursive program** based on call_graph, and it is probably **not an iterative program** since no memory is allocated for any intermediate steps, like for variables in a for loop, or calls to helper functions. Memory complexity is **O(1)** as memory is allocated with malloc independent of n.

Here is a call_graph.out for program description and memory complexity:

```
 main(2,...)
  atoi(140721344160622,...)
   strtoll(140721344160622,...)
    ____strtol_l_internal(140721344160622,...)
   fib(10,...)
   printf(4804612,...)
    __vfprintf_internal(4981536,...)
     __strchrnul_avx2(4804612,...)
     _IO_file_xsputn(4981536,...)
     _itoa_word(55,...)
     _IO_file_xsputn(4981536,...)
      _IO_new_file_overflow(4981536,...)
       _IO_doallocbuf(4981536,...)
        _IO_file_doallocate(4981536,...)
         _IO_file_stat(4981536,...)
          __fxstat64(1,...)
          __malloc(1024,...)
           _int_malloc(4982688,...)
          _IO_setb(4981536,...)
        _IO_new_do_write(4981536,...)
        _IO_default_xsputn(4981536,...)
         _IO_new_file_overflow(4981536,...)
         _IO_new_file_overflow(4981536,...)
       __strchrnul_avx2(4804615,...)
       _IO_file_xsputn(4981536,...)
        __mempcpy_avx_unaligned_erms(29482402,...)
        _IO_new_file_overflow(4981536,...)
         _IO_new_do_write(4981536,...)
          _IO_new_file_write(4981536,...)
           __libc_write(1,...)
     exit(0,...)
```

Here are the results of different inputs used:

```
Singularity> time ./fib2 50
12586269025

real	0m0.001s
user	0m0.001s
sys 0m0.000s
Singularity> time ./fib2 45
1134903170

real	0m0.001s
user	0m0.000s
sys 0m0.001s
Singularity> time ./fib2 40
102334155

real	0m0.001s
user	0m0.001s
sys 0m0.000s
Singularity> time ./fib2 30
832040

real	0m0.001s
user	0m0.000s
sys 0m0.000s
Singularity> time ./fib2 20
6765

real	0m0.001s
user	0m0.001s
sys 0m0.000s
Singularity> time ./fib2 10
55

real	0m0.001s
user	0m0.000s
sys 0m0.001s

Singularity> time ./fib2 100
3736710778780434371

real	0m0.001s
user	0m0.000s
sys 0m0.000s
```

Singularity> time ./fib2 1000
817770325994397771

real 0m0.001s
user 0m0.000s
sys 0m0.000s

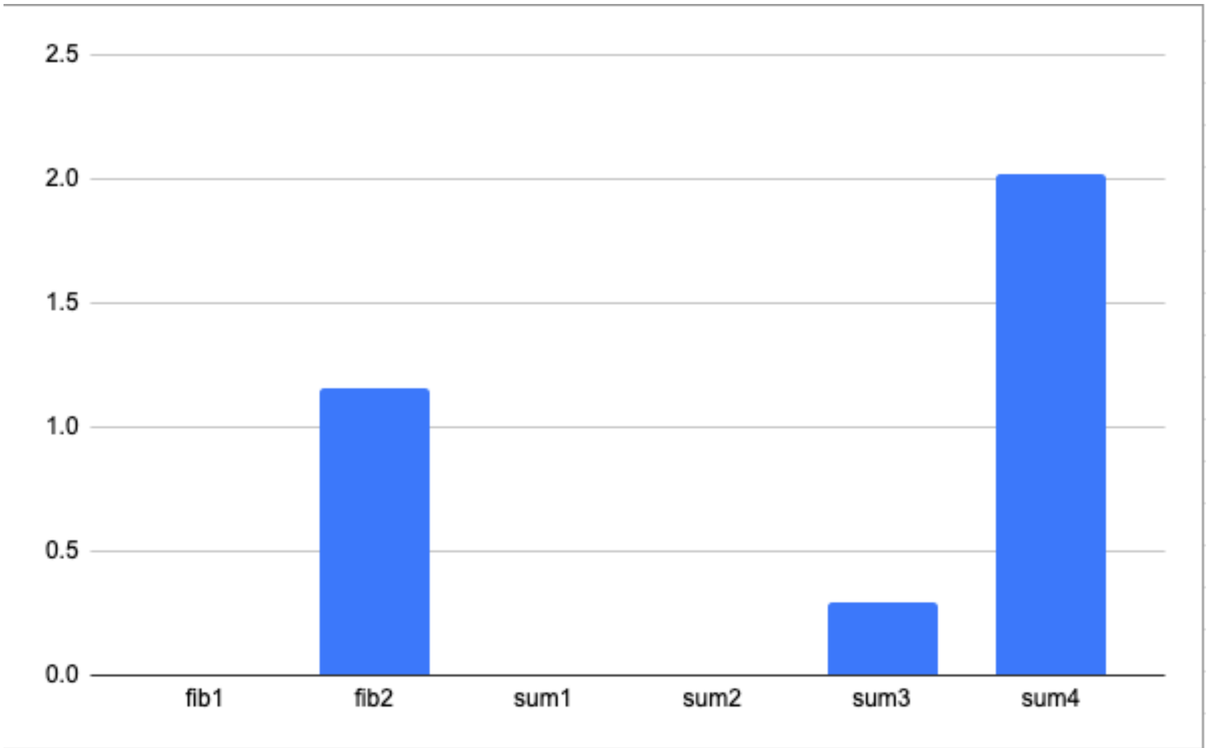
# 3.5:

The recursion used in Sum1 makes it far slower, but Sum 2, 3, and 4 have the same runtime complexity, yet take different amounts of time to complete. We can see from Part 1 data that sum3 exhibits very low L1-dcache-load misses, which would make it faster. Looking at data from 3.1, we can see sum3 was the fastest. Sum4 does have the most cache misses out of sum2, sum3, and sum4, but is the second fastest in 3.1. Yet, sum4 has significantly lower branch-misses and instructions than sum2. Having fewer branch-misses than sum2 could be the reason that sum4 was faster than sum2 in 3.1.

# 3.6:

I used mem_trace, and modified it to count the total number of memory accesses. I think my methodology is correct, but I could not compute all of the results in time. My program computing the memory accesses for fib1 would not complete in a reasonable amount of time, but considering the high number of cache misses, and that printing a line for every memory access using mem_trace made the program take longer than 10 times the time it was taking before, the memory must have had many access. I had the same issue with sum1 and sum2.

| 3.6 | cache misses | memory accesses | cache-miss rate |
|---|---|---|---|
| fib1 | 21,891.00 | NA | NA |
| fib2 | 1,471.00 | 1271 | 1.157356412 |
| sum1 | 74,888,340.00 | NA | NA |
| sum2 | 376.00 | NA | NA |
| sum3 | 274.00 | 927 | 0.2955771305 |
| sum4 | 1,674.00 | 827 | 2.024183797 |


2.5
2.0
1.5
1.0
0.5
0.0
fib1
fib2
sum1
sum2
sum3
sum4