Daniel Lung
DS210 HW6 Report

1c.
"cargo run"

- For numbers **1-20**, the time it took to compute ranged from 1µs to 252µs, increasing at varying intervals as the input number increased.
- For numbers **21-30**, the time it took to compute ranged from 406µs to 21.341ms, increasing at varying intervals as the input number increased.
- For numbers **31-40**, the time it took to compute ranged from 32.269ms to 2.44996s, increasing at varying intervals as the input number increased. The increase is starting to be extremely noticeable and calculations take longer.
- For numbers **41-47** (terminated at 47 because the calculation was taking too long), the time it took to compute ranged from 3.93241s to 70.084953s, increasing at varying intervals as the input number increased. The increase is e extremely noticeable and calculations take longer than a minute after 47.

"cargo run –release"

- For numbers **1-20**, the time it took to compute ranged from 0ns to 36µs, increasing at varying intervals as the input number increased. There is a noticeable increase in speed compared to "cargo run"
- For numbers **21-30**, the time it took to compute ranged from 59µs to 3.81ms, increasing at varying intervals as the input number increased. There is a noticeable increase in speed compared to "cargo run"
- For numbers **31-40**, the time it took to compute ranged from 5.693ms to 288.28ms, increasing at varying intervals as the input number increased. The increase is getting larger, but it takes a fraction of the time it takes for "cargo run".
- For numbers **41-50**, the time it took to compute ranged from 469.153ms to 36.926453s, increasing at varying intervals as the input number increased. The increase is extremely noticeable and calculations take longer. However, the time was not so bad that I needed to terminate the program. Calculating the value for 50 took half the time it took for "cargo run" to calculate for 47.

Overall, "cargo run –release" takes a fraction of the time it takes for "cargo run" to run the same operations. When it comes to calculating the smaller numbers, both do a pretty good job, but there is a clear difference as numbers start getting bigger. I would say that "cargo run –release" feels at least 2-4 times faster than "cargo run" on average.

2b.

u8:
“cargo run” gives this output:

```
thread 'main' panicked at src/main.rs:13:26:
attempt to add with overflow
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

“cargo run –release” gives this output:

```
The Fibonacci number of each index in the array is: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 121, 98, 219, 61, 24,
85, 109, 194, 47, 241, 32, 17, 49, 66, 115, 181, 40, 221, 5, 226, 231, 201, 176, 121, 41, 162, 203, 109, 56, 165, 221, 130, 95
, 225, 64, 33, 97, 130, 227, 101, 72, 173, 245, 162, 151, 57, 208, 9, 217, 226, 187, 157, 88, 245, 77, 66, 143, 209, 96, 49, 1
45, 194, 83, 21, 104, 125, 229, 98, 71, 169, 240, 153, 137, 34, 171, 205, 120, 69, 189, 2, 191, 193, 128, 65, 193, 2, 195]
```

“cargo run” doesn’t work and says that it panicked. This means that there was an integer overflow while attempting to run this program. (u8) can only represent values from 0 to 255, and if the Fibonacci number exceeds this range, it will overflow. The program doesn’t run for “cargo run”, but runs for “cargo run –release”. However, the output by “cargo run –release” is incorrect because u8 only represents values for 0-255. Due to this, the correct values are not able to be represented, leading to the values being incorrect,

U128
Same output for “cargo run” and “cargo run –release”

```
The Fibonacci number of each index in the array is: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 15
97, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040, 1346269, 2178309, 352457
8, 5702887, 9227465, 14930352, 24157817, 39088169, 63245986, 102334155, 165580141, 267914296, 433494437, 701408733, 11349
03170, 1836311903, 2971215073, 4807526976, 7778742049, 12586269025, 20365011074, 32951280099, 53316291173, 86267571272, 1
39583862445, 225851433717, 365435296162, 591286729879, 956722026041, 1548008755920, 2504730781961, 4052739537881, 6557470
319842, 10610209857723, 17167680177565, 27777890035288, 44945570212853, 72723460248141, 117669030460994, 190392490709135,
 308061521170129, 498454011879264, 806515533049393, 1304969544928657, 2111485077978050, 3416454622906707, 552793970088475
7, 8944394323791464, 14472334024676221, 23416728348467685, 37889062373143906, 61305790721611591, 99194853094755497, 16050
0643816367088, 259695496911122585, 420196140727489673, 679891637638612258, 1100087778366101931, 1779979416004714189, 2880
067194370816120, 4660046610375530309, 7540113804746346429, 12200160415121876738, 19740274219868223167, 319404346349900999
05, 5168070885485832307 2, 8362114348984842297 7, 13530185234470674604 9, 21892299583455516902 6, 35422484817926191507 5]
```

There is no difference in between the two, they both give the same, correct output this time. However, looking at the time taken to run the program, “cargo run” took 0.81s while “cargo run –version” took 0.15s This time there is no overflow because u128 is represents numbers from 0 to 2^128, which is a very large number, so there is no issue with representing the correct numbers now.

3.

In this program, we don’t encounter any overflow issues because u32 is sufficient for displaying the result of the calculations that would mostly be used in this program. The results would not be big enough to cause it to overflow. 2^32 is still a pretty significant number. u8 would not be big enough because it only goes up to 255, but u32 is much bigger. “cargo run” and “cargo run –release” are both able to run the code with no issues and no differences.