Matthew Morris
Professor Kontothanassis
11 Feb 2025
DS 210 HW 2

Worked with Liam Durcan on this assignment.

Report: What are the computation times for different *k* for each of the options? How do the times to compute Fibonacci numbers compare for large *k* between these two options? Are they roughly the same or are they very different? If they are different, what is the multiplicative difference?

K Value	Cargo Run (Time)	Cargo Run –Release (Time)
10 (fibonacci 9)	137µs	61µs
20 (fibonacci 19)	934µs	296µs
30 (fibonacci 29)	47.292ms	11.909ms
40 (fibonacci 39)	4.995778s	725.011ms
50 (fibonacci 49)	466.266559s	68.848674s

They are drastically different–a ~6.7x difference between Cargo run and Cargo run –release

The time difference for large k is huge–the function I used has a time complexity of O(2^N).

Report: Now conduct the following experiment. Replace the array entry type with u8 and adjust any other types accordingly so your program still compiles. Try running the modified code with both cargo run and cargo run --release. Are there any differences in the behavior of the program? If so, what are they?

With u8 and Cargo run –release, the numbers start looping after f(13), however with Cargo run, it doesn't work at all, instead giving "thread 'main' panicked at src/main.rs:6:20: attempt to add with overflow," and because it isn't in release, it doesn't work with overflow. It may be better to run the regular version over the –release version due to accuracy, because after f(13), the –release version becomes inaccurate.

Report: Explain why the situation described above is not happening, i.e., why the range of integers you use is sufficiently large. This kind of problem is known as *integer overflow*, i.e., you want to explain why integer overflow is not a problem in your code.

Integer overflow is not a problem because I account for every integer; if the number input is >255, thus if it is not in the range of u8, then it returns “not a good number,” but if it is less than that, it returns the respective value. I tested this with 1, 10, 255 and 256 and they all work as expected. I found out via doing the calculation myself (using the (n(n+1)/2)^2 equation) with 255, that the unsigned integer bit value must accommodate 1065369600–thus nothing below u32 would suffice. Therefore, I made sure that the u8 would convert to u32 in the function to make it run appropriately and not result in integer overflow.