I experimented with a few different ideas: (1) Collatz1 simply calculates the Collatz sequence length for every number under the limit separately and returns the number with the highest (this is pure brute force). (2) Collatz2 does the same, except it remembers the sequence length of every number less than the present starting number, so that the total can be calculated as soon as the Collatz path intersects with anything we’ve already figured out (many versions of this have already been posted). (3) Collatz3 does what Collatz2 does, using the knowledge gained in previous iterations, except that it uses a stack to remember its path, and then it records all Collatz numbers (under some limit) along the Collatz path it reaches, not just the very first one. Finally, (4) Collatz4 uses a priority queue and calculates the Collatz number backwards starting with one.

Results: Collatz2 and Collatz3, unsurprisingly, significantly outperformed Collatz1. Collatz2 (the fastest) could handle limit=1 billion in under a minute on my machine. With respect to Collatz2 and 3, it’s possible to remember more than we strictly need to remember (say, 1..2n rather than 1…n) on the theory it might ultimately be faster to use that “excess” info. However, Collatz2/3 ultimately performed fastest when the working limit was the same as the problem limit. Collatz3 was slower than Collatz2, probably because it wasted a lot of time unpacking stacks of numbers way over the limit, though still much faster than Collatz1. I tried to make Collatz2 even faster by directly computing available stuff like C(q\*2^p) is C(q)+p where q is odd, but it didn’t make a huge difference. Collatz4 was the slowest of all – I guess that idea just didn’t pan out, probably because it wastes too much time figuring out unnecessary Collatz numbers. I experimented with a few different priority functions, but none of them got close to Collatz2/3 performance.

Collatz1 code

Collatz2 code

Collatz3 code

Collatz4 code