QGCL CHALLENGE PROGRAM

JK BEATTIE

ABSTRACT. This document is a record of my analysis of the qGCL challenge program as shown by McIver in her lecture of 15 January 2025.

Thursday 1ˢᵗ May, 2025 at 08:26

1. INTRODUCTION

In her presentation at the annual LMS and BCS-FACS joint seminar, January 2025, McIver included "a challenge program". The screen grab below shows the program (Figure 1.1 on page 1).

The challenge program takes an argument N, a natural greater than zero, and returns a natural c selected uniformly at random from the range $0 \leq c < N$.

The program uses a single random choice primitive, which selects between two options at equal probability.

Screen grab from Probabilistic Datatypes, Annabelle McIver,
LMS BCS-FACS 15 January 2025

Algorithm 1 McIver's challenge program

```
    var c, v = 0, 1
    while(V<N or c>=N){
        {Inv x [v < N or C >= N]}
        □ (v <= N) → v = 2v ;  c = 2c [½] c = 2c+1
        □ (v > N)  → v,c=v-N,c-N
        {Inv}
    }
```

The program is efficient in that the random choices which are made in order to get to the result *c* in the desired range are all effective, none are wasted.

The code is shown in Algorithm 1, based on Dijkstra's guarded command language. Annotations are in braces. Random choice is indicated by [½].

The purpose of this paper is to analyse this algorithm, ideally providing loop invariant(s) as appropriate and proving correctness. The algorithm terminates for *N* a power of 2. The algorithm terminates almost-certainly for *N* not a power of 2. That is, the probability of that the algorithm will cycle through more loops reduces exponentially with each outer loop, and is zero in the limit. The analysis below explains the meaning of 'outer loop'.

This PDF and the implementation scripts are available on github.

01 May 2025 I have skimmed various papers and the textbook [McI06] to see if I can express the correctness proof formally. I can see that the technology given there would support a formal proof but I don't have the patience to work through the details.

2. Analysis

2.1. **Restructuring the loop.** Referring to Algorithm 1, the two guarded commands inside the while loop above govern two different loop constructs.

So, we convert to nested loops, first using repeat for the outer loop, Algorithm 2. This first try is incomplete, see the next refinement for better.

Now tidy-up to make two while loops and modify the initialisation so that the outer loop can be expressed correctly as a while, Algorithm 3.

Finally, unroll the outer loop once to simplify the initialisation steps, Algorithm 4. Note that the initialisation step now sets c=0, v=1 which means that the loop invariant is true. It is not true at the top of the outer loop in the previous version. The proposed invariant is defined at the end of the next section. Then the initial run of I(v,c) expands the initial state

Algorithm 2 Initial re-write, incomplete

```
var v=1
var c=0
repeat {
    ...
    while (v<N or c>=N){
        □ (v <= N ) -> v = 2v; c=2c [½] c=2c+1
    }
    □ (v > N ) → v = v - N ; c = c - N
} until (v>=N and c < N)
```

Algorithm 3 Re-written to be two nested while loops

```
var v=N+1
var c=N
while (c >= N){
    v = v - N ; c = c - N          # Note 1
    while (v<N){                    # Note 2
        v = 2v; c=2c [½] c=2c+1     # Note 3
    }
}
```

(1) Relative to the previous version Algorithm 2, we don't need the guard, (v > N)→, since v>c>=N
(2) Likewise, we don't need 'or (c >= N)', in part because it is false, since c<v and v<N but mostly because it controls the outer while loop, not the inner loop
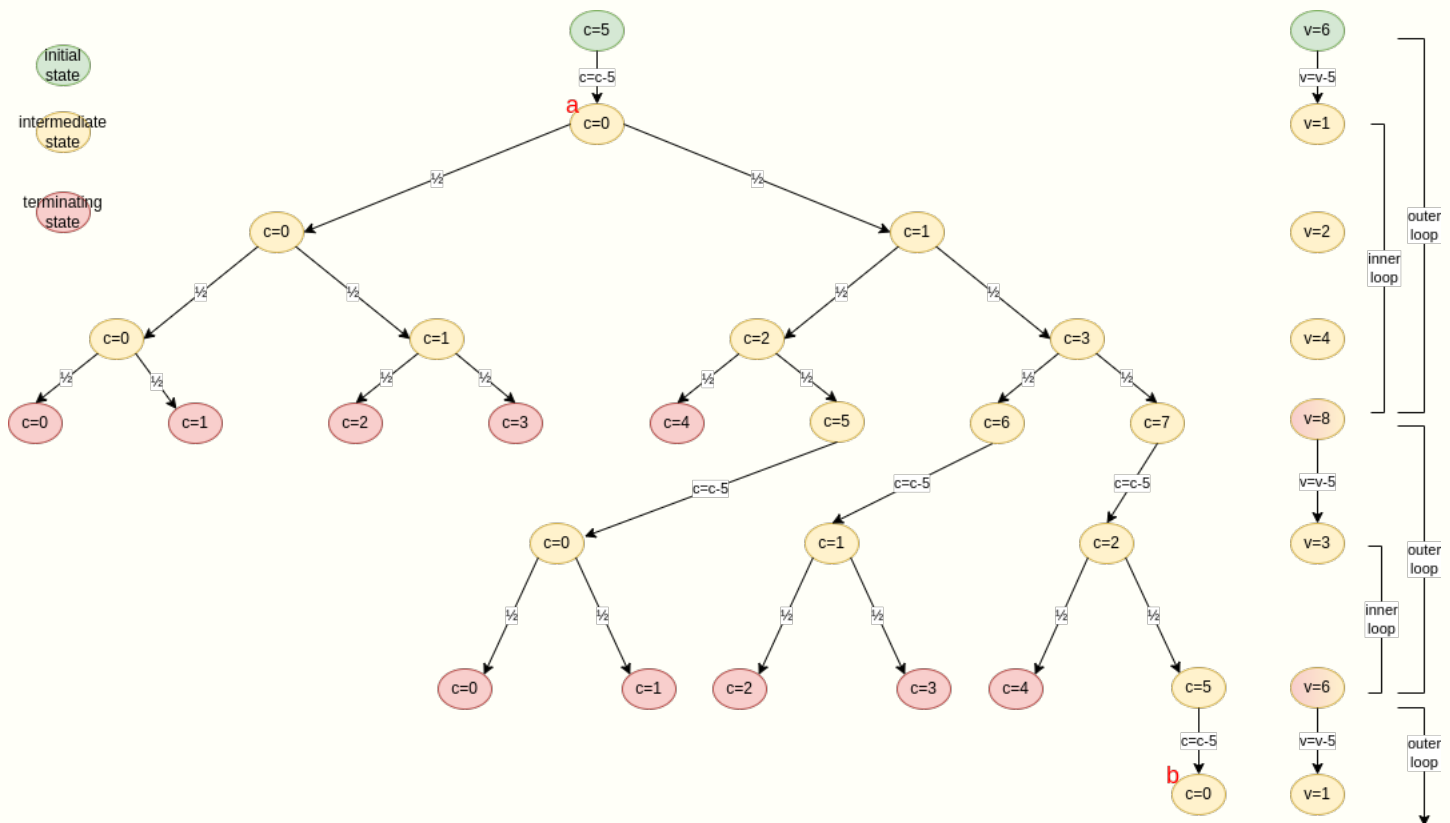(3) ½ is shorthand for $p : \sigma \to 1/2$, where σ represents the state. See [Kam19, p57 &seq] for the syntax & semantics.

Algorithm 4 Re-written to unroll the outer loop once

```
var v=1
var c=0
while (v<N){
    v = 2v; c=2c [½] c=2c+1
}
while (c >= N){
    v = v - N ; c = c - N
    while (v<N){
        v = 2v; c=2c [½] c=2c+1
    }
}
```

until $v \geq N$, and then `O(v,c)` runs if required, including daughter runs of `I(v,c)`.

Evolution of state space for N=5

Figure 2.1 on page 4 shows the evolution of the state space for N=5. The arrows show successive transitions of the two state variables, v, c. They transition in tandem, going down the diagram. At $v = 8$ the probability range has been doubled three times and is now larger than $N = 5$. Each possible value of c has equal probability of being chosen. Hence if the program has selected a value of c in the desired range $0 \le c < N$, then this value has an equal probability with any other value in the desired range.

If the program has selected a value in the complement of the desired range, then it reduces the state c, v such that $c < v \,\&\, v < N$ and then cycles to expand the range represented by v.

Finally, it is worth noting that the state space repeats indefinitely, but at a smaller scale in each repeat, and is thus a fractal. In Figure 2.1 on page 4 the state at a is repeated at b.

There is a relevant relationship with the binary expansion of $1/N$. For $N = 5$, $1/5 = \overline{0.0011}_2$, as shown in the long division in Figure 2.2 on page 5, which we discuss briefly, so as to connect it with the evolution of the state space.

Long division of 1/5 in binary

Bringing down a zero from the dividend means appending a zero to the right-hand end of the current trial value. Ignoring the place of the binary point, this works in the same way as multiplying by two. Zeroes are brought down until the trial value is bigger than N, which in the example means 1000_2 or 8. Then we subtract 101_2, leaving remainder 11_2. This is equivalent to the stage $v = 8$, Figure 2.1 on page 4. The desired range for c is $0 \leq c < 5$, size 5, and the complement is size 3. If $c \geq 5$, i.e. c is in the complement range, then the outer cycle loops. v is reduced to 3 giving a range which starts at zero. It is less than N, as it must be as it is the remainder $v\%N$. c is reduced so that $c < v$ is maintained.

Doubling v gives $6 > 5$ so 1 is appended to the quotient. In the state space, the inner loop terminates immediately and the outer loop tests for $c < N$ for termination.

The long division also shows the obvious repetition for the expansion of a rational in any base. This matches the fractal repeat of the state space.

Another interpretation for the binary expansion of $1/N$ is that the algorithm is is consuming increasing quantities of the probability mass: $1 = 5/8 + 5/16 + 5/128 + 5/256 + \dots$ That is, on the first cycle of the outer loop, the sub-distribution at termination has mass $5/8$, after the second, the sub-distribution has mass $5/16$ and so on.

Algorithm 5 Program components of the algorithm

```
var v,c,w,N
Init(v,c,w)(N) = {v=N+1;c=N;w=1}
I(v,c) = while (v < N) {
            v=2v;
            {c=2c [1/2] c=2c+1}
        }
O(v,c) = while (c >= N) {
            v=v-N; c=c-N;
            I(v,c);
            if (c>= N) {w=w*(v-N)/v} else {w=w*N/v}
        }
C(v,c) = Init(v,c,w)(0); I(v,c); O(v,c)
```

Or, it is using increasingly accurate binary approximations of the desired probability, $1/N$. In the example case, $1/5 = 1/8 + 1/16 + 1/128 + 1/256 +$ .... Where the algorithm terminates after the first cycle of the outer loop, the probability of the value c is $1/8$. If it terminates after the second, the probability is $1/16$. Visually, instead of dividing the unit interval into five equal slots, we divide it first into eight equal slots and assign five at $1/8$ each. We then subdivide the remaining three slots to make six and assign five at $1/16$ each. We then start again with the remaining $1/16$ slot and divide that into eight, repeating the process, but at one-sixteenth of the overall mass. Overall, the unit interval is subdivided into a fractal, assigning increasingly accurate approximations of one-fifth of the interval to each possible result.

Note that, when N is a power of 2, the algorithm definitely terminates after one cycle of the outer loop. It does not definitely terminate for N not a power of 2. Hence these two cases are different and must both be discussed.

2.2. **qGCL.** The programs in Algorithm 5 comprise the algorithm.

Definition 1. Define a version of *Unif* ([Kam19, p60]) where the total probability mass can be other than 1:

$$Unif(w,b) \quad = \quad \sum_{i=0}^{b-1} \frac{w}{b} |i\rangle \tag{2.1}$$

$$(v \in \mathbb{N}, w \in \mathbb{Q}, 0 < w \leq 1) \tag{2.2}$$

Definition 2. The following equations define the semantics of the algorithm, where $N : \mathbb{N}$ and $w \in [0,1] \cap \mathbb{Q}$ are auxilliary variables. N is an external parameter of the program, i.e. the input argument. w is the probability mass of the sub-distribution at termination [Kam19, definition 3.8].

$$Vars = \{v, c\} \tag{2.3}$$

$$Vals = \mathbb{N} \tag{2.4}$$

$$\sigma : Vars \rightarrow \mathbb{N} \tag{2.5}$$

$$F(\sigma, w, N) : [\![C(v, c)]\!]_\sigma = Unif(w, N) = \sum_{0 \leq i < N} \frac{w}{N} |\sigma[c \mapsto i]\rangle \tag{2.6}$$

$F(\sigma, w, N)$ is the predicate for the desired post-condition. It asserts of the sub-distribution at termination, $[\![C(v, c)]\!]_\sigma$, that c is distributed uniformly in the range $0 \leq c < N$. Note that this is a sub-distribution. The total weight is $w \leq 1$.

We want to prove

$$\{N > 1 \,\&\, (\nexists m.N = 2^m)\}\,\texttt{Init(v,c,w)(0); I(v,c); O(v,c)}\,\{F(\sigma, w', N)\}$$

where F is the assertion above. The special cases excluded by the precondition are discussed below.

The invariant for both loops is $Inv : c < v \,\&\, \mu(\sigma) = Unif(w, v)$. $\mu(\sigma)$ is the distribution of c at state σ. v is a definite variable and doesn't affect the distribution.

At termination, $[\![C(v, c)]\!]_\sigma = \mu(\sigma)$, [Kam19, definition 3.8]. At termination, $c < N \leq v$. Together with the invariant, it follows that $\mu(\sigma)$ is the subdistribution of $Unif(w', v)$ with $c < N$, i.e. $Unif(w', N)$.

2.3. **Special cases of N.**

N=1.

$$\{\texttt{N=1}\}\ \texttt{Init(v,c,w)}\ ;\ \texttt{O(v,c)}\ \{F(1,1,1)\}$$

is also true, since $\{\texttt{N=1}\}\texttt{I(1,0)}\{F(1,1,1)\}$ is true without executing the inner loop, as c is already $0 < 1$. However, this all follows different reasoning than the case(s) $N > 1$.

$$\begin{aligned} F(1) &= \sum_{0 \leq i < N} \frac{w}{N} |\sigma[c \mapsto i]\rangle \\ &= \sum_{0 \leq i < 1} \frac{w}{1} |\sigma[c \mapsto i]\rangle \\ &= |\sigma[c \mapsto 0]\rangle \end{aligned}$$

N a power of 2.

```
{N>1 & N is a power of 2}
Init(v,c,w} ; O(v,c)
{F(1,1,N)}
```

is also true, since `{N=2^m} Init(v,c,w} ; O(v,c) {F(1,1,N)}` requires only one cycle of the outer loop `O(v,c)` to be true. At termination $v = N$:

```
{N = 2^m}
Init(v,c,w)
v=v-N; c=c-N; I(v,c); w=w*N/v
{v=N & w=1 & 0<=c<N & F(w,N)}
```

3. ADDITIONAL

3.1. **Script implementations.**
The algorithm has been implemented as bash and python scripts, see the files `nested` in the bash and python folders in the github project.

3.2. **Generalisation.**
The algorithm can be generalised to other bases. That is, instead of random choice with probability $1/2$, we generalise to allow probability $1/B$ for $B \in \mathbb{N}$. This amounts to selecting a base B for the expansion of $1/N$.

See the nestedB script in the python folder.

One key difference is that the range $0 \leq c < v$ is expanded by multiplying by B, which can be larger than 2. Then $v > N$ may in fact be several multiples of N. Hence the code treats the range $[0..N..2N....mN..v - 1]$ by folding the complete sections of length N. Any $c < mN$ is taken modulo N and is a valid result: a selection in the desired range, of equal probability.

If N is a power of B or of one of the factors of B (consider $N = 5, B = 10$) then the algorithm will terminate as in the case of N a power 2 discussed above.

3.3. **Other formulations of the basic algorithm.**
The algorithm as presented by McIver is a single loop containing two guarded commands. The analysis above started by reconfiguring the code to be two loops, one nested inside the other. There are alternatives, two are mentioned below.

Make the outer loop into a guarded command within the inner.

```
E(v,c) = while (v < N) {   # E for expand
            v=2v;
            c=2c [1/2] c=2c+1
            S(v,c)
         }
S(v,c) = if(c>=N) {        # S for setup
            v=v-N; c=c-N;
```

```
        w=w*(v-N)/v
    }
Term(w) = {w=w*N/v}          # sub-distribution weight
C''(v,c) = Init(v,c,w)(0); E(v,c); Term(w)
```

This avoids unrolling the outer loop and allows correct initialisation of c, v. However, it means that the guard in S is tested on every loop of E, which often is unnecessary work.

Express the algorithm using coroutines. See the coroutine scripts in the bash and python folders. One benefit of the coroutine formulation is to make it easier to see how to modify the loops as shown in the alternatives above.

3.4. **Web pages.** See also stackoverflow which discusses manual algorithms corresponding to long division.

https://en.wikibooks.org/wiki/Fractals/Mathematics/binary

https://www.omnicalculator.com/math/binary-fraction

REFERENCES

[Kam19] Benjamin Lucien Kaminski. "Erweiterte wp-Kalküle für Proba-bilistische Programme". en. PhD thesis. RWTH Aachen University, 2019, p. 2019. DOI: 10.18154/RWTH-2019-01829. URL: https://publications.rwth-aachen.de/record/755408/files/755408.pdf (visited on 02/25/2025) (cited pp 3, 6, 7).

[McI06] Annabelle McIver. *Abstraction, Refinement and Proof for Probabilistic Systems*. Ed. by Charles C. Morgan. Monographs in computer science. Dordrecht: Springer-Verlag New York Inc, 2006. 397 pp. ISBN: 9780387401157 (cited p 2).