Given a biased coin with probability of p of heads, for what values of p is it possible to provide an algorithm to produce an unbiased bit from exactly N throws of the coin? We will use the example N = 3 as posed to develop the approach. Also replace heads and tails by 1 and 0.
With N throws there are 2^N possible outcomes. To guarantee an output after N throws, an algorithm must assign an output to each of the 2^N possible outcomes, O:i in [0..(2^N)-1] --> [0,1]. There is a probability of each outcome P(i), and the requirement of fairness means that sum P(i)*O(i) = 1/2. In the example N = 3 there are 8 possible outcomes, 000, 100, etc, and we must assign a result to each outcome such that the sum of the outcomes weighted by the probability of each outcome is 1/2. For example, we might assign the output 1 if either 000 or 010 occurs, otherwise 0. The probability of the former is (1-p)^3, and the probability of the latter is (1-p)^2*p1. Therefore a solution is possible if (1-p)^3+(1-p)^2*p = (1-p)^3+(1-p)^2*p = 1/2. This simplifies to (1-p)^2 = 1/2, so we solve for this and find p1 = sqrt(2) = .7071. In this case there is a .353 chance of 000 and .146 chance of 010, so we will output a 1 50% of the time.

Define a mapping algorithm as a mapping of a set of outcomes to an output. The preceding algorithm is A({000,010}) which produces the polynomial listed above. Since there are 2^N possible outcomes, and we can independently assign an outcome to each of the 2^N outcomes, there appear to be 2^(2^N) possible algorithms and seemingly the same number of roots. For N=3 this amounts to 256 possibilities.

We can reduce this number, both to get a tighter bound on the number of possible answers, and for computational efficiency. First, note that many algorithms produce the same polynomial, because the probability of an outcome depends only on the number of 1’s and 0’s, not their order. For example, 001, 010, and 100 are all equally likely, so algorithms containing any one of them are identical. Similarly for 011, 101, and 110. Therefore we can replace the exact patterns with a count of the number of 1’s in an outcome. The pattern 000 has 0 1’s, and 010 has 1 1. We can represent an algorithm as a vector [n0, n1,..] where the n[i] are the number of patterns with i 1’s that will produce a 1. The above algorithm {111, 010} can be represented as [0,0,1,1].

It can be seen that the number n[i] must be between 0 and combin (N, i) inclusive, since combin (N,i) is the number of possible outcomes that contain exactly i 1’s. Therefore the number of unique algorithms reduces to product (i = 0 to N of (combin (N, i) + 1)). For N = 3 the values of combin are 1,3,3,1 so the number of possible algorithms is 2 * 4 * 4 * 2 = 64. This dramatically reduces the number of polynomials that need to be tested for large N, as seen in the table below:

	2^2^N	combin
2	16	12
3	256	64
4	65536	700
5	4294967296	17424
6	1.8446744E+019	1053696
7	3.4028236E+038	160579584

N larger than 4 is certainly intractable with the exponential approach, but up to 6 or maybe 7 is possible with the combin.

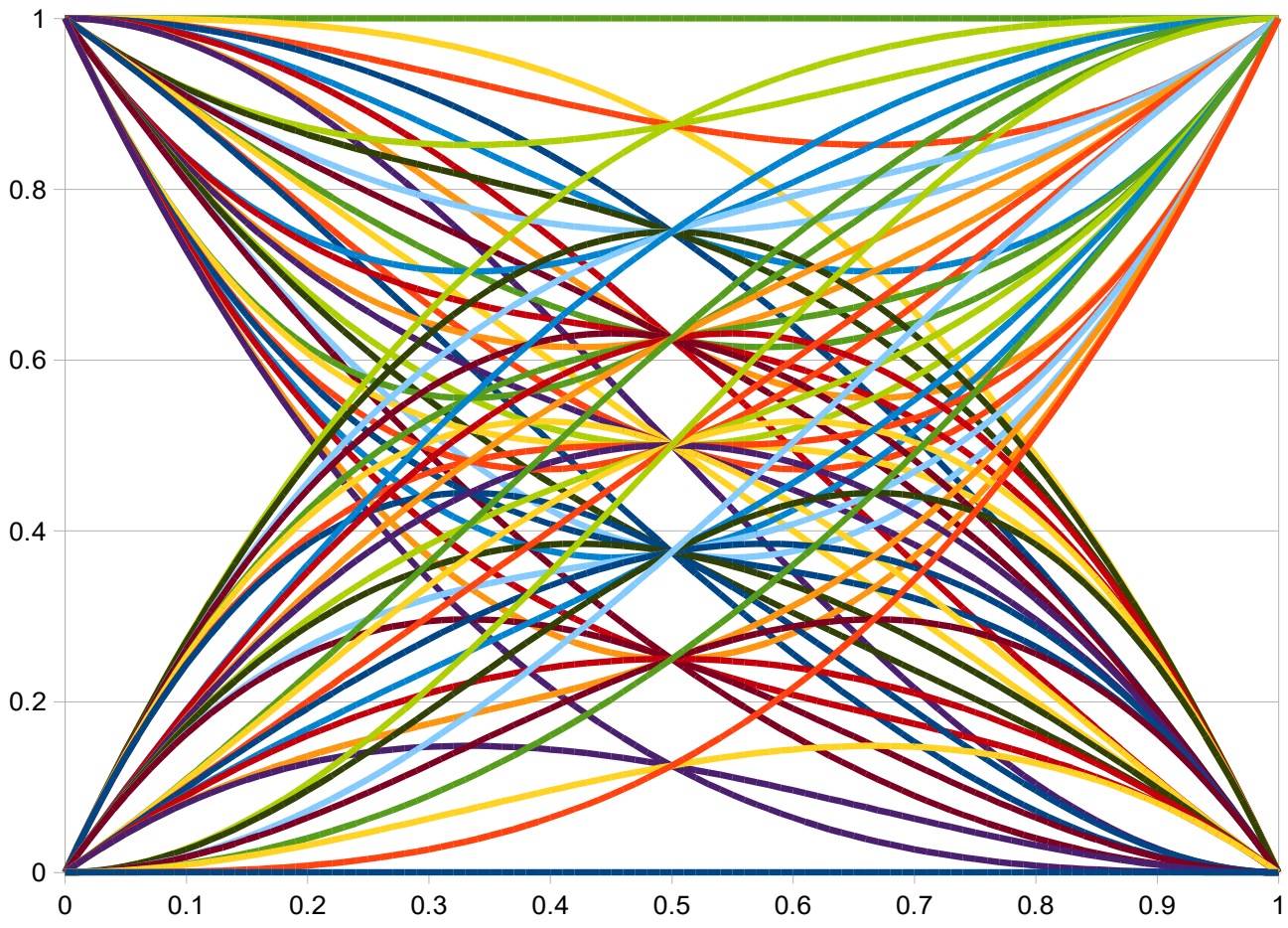
Also note that the combin is the number of polynomials, and since higher order polynomials can have more than one distinct root, there are possibly more solutions than the combin number.

This still overestimates the number of solutions for a couple of reasons.

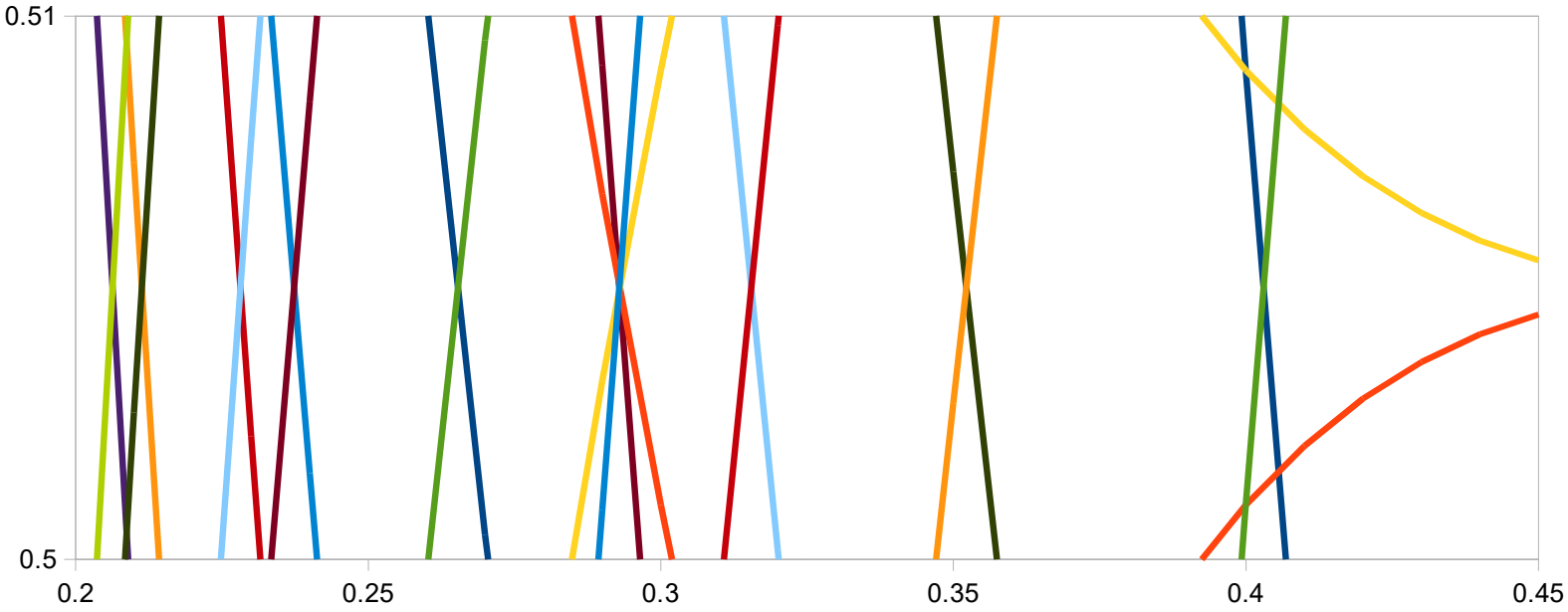
First, some of the algorithms have no valid values of p. Consider A({}), the empty set. This has the polynomial 0, and there is no solution to the equation 0 = 1/2. This corresponds to the fact that the algorithm always produces a 0 output.

Other polynomials can also fail to have a valid p. For example A({010}) corresponds to the polynomial (1-p)^2 * p, which has a maximum value of 0.148, so has no value of p that produces a 0.5 probability. Finally, if A(s) has a value of 0.5 for some p, then A(~s) = 1 – A(s) = 0.5, as well for the same value of p. Here, ~s means the complement of the set s, so corresponds to the algorithm that produces the exact opposite output as s. Therefore there are always at least 2 algorithms that work for the same value(s) of p. This reduces our potential 64 to 32, less the ones that have no solution.

At this stage we need to explicitly find the solutions for these polynomials. A pretty plot of all the curves is below. This shows the probability of outputting a 1 for all values of p, for each algorithm.



Each line that crosses y=0.5 produces an unbiased output for the value of p corresponding to that location on the x axis. As described above, there is always a pair of lines that intersect at y=0.5. There is also symmetry about the value x=0.5. To explore a bit more we zoom in:



Now we can see 9 distinct values in the range 0..0.5. There is also a root at 0.5, and another 9 roots mirrored about x=0.5.

To determine these precisely we simply write a program that tests every polynomial and finds the roots:

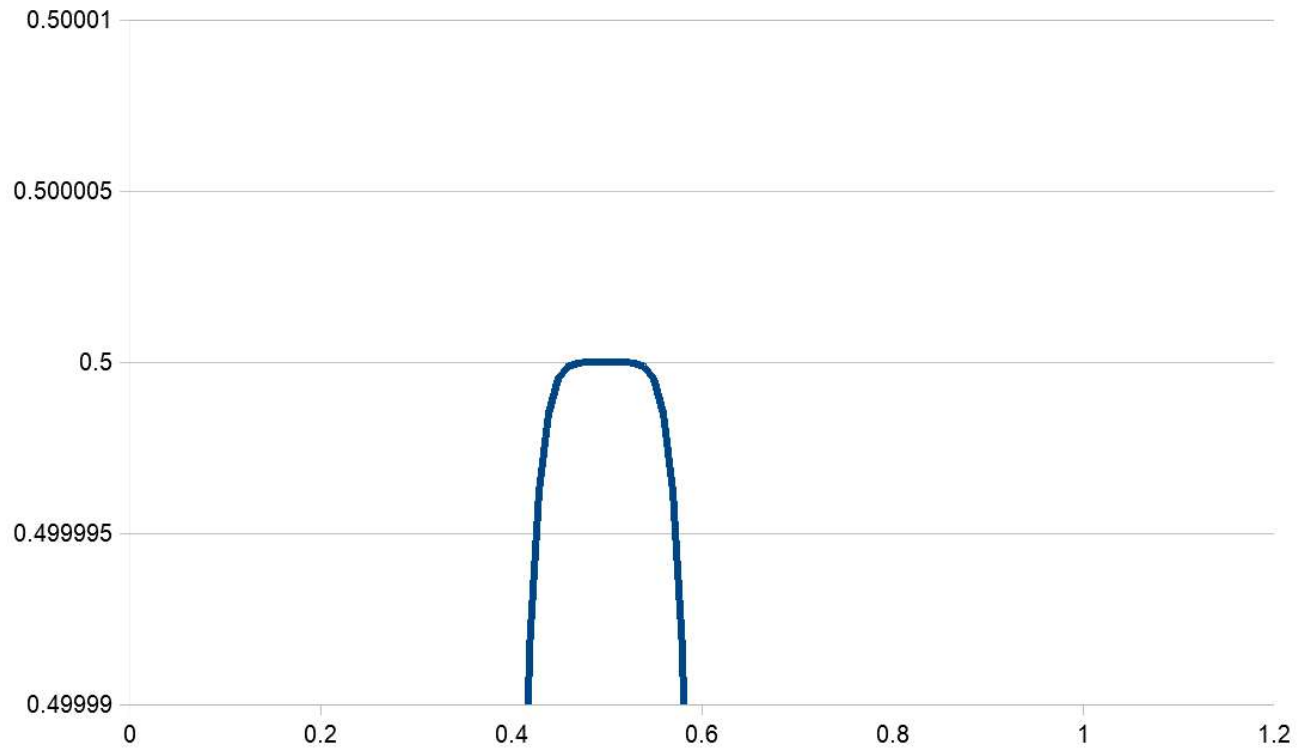
root 0.206299
root 0.211325
root 0.228155
root 0.237286
root 0.265302
root 0.292893
root 0.315449
root 0.352201
root 0.403032

root 0.5
root 0.596968
root 0.647799
root 0.684551
root 0.707107
root 0.734698
root 0.762714
root 0.771845
root 0.788675
root 0.793701

There are the 19 roots as seen on the graph.

Note the extreme values are 2^(-1/3) and 1-2^(-1/3) corresponding to the extremes where a string of 3 1's or 3'0s has a 0.5 probabilty.

Another note is that even if you don't have the exact p that produces an unbiased result, for values near 0.5, the error is reduced by the N * error ^ N. For example in the N=3 example, the best algorithm near 0.5 is 4*(p-0.5)^3 so the error is reduced.



I use Newton-Raphson to solve the polynomials. It can be seen that some polynomials are tangent to y=0.5, indicating a compound root, and Newton-Raphson converges badly for these. Here is an example for N=6, [0,6,0,20,0,6,0]. This seems to be a 6^{th} order root at x=0.5 so converges horribly. Note the zoomed in scale on Y.

The easiest solution was to use a multiprecision library and insist on a convergence of Y to within 1e-100. Using this we can get the following number of roots for N=4,5,6:

N = 4 217
N = 5 8635
N = 6 421733
N = 7 107958091

Note that number of roots for N=7 exceeds the combin number / 2, which means that some of the polynomials have multiple roots.

This table compares the exponential to the combin to the actual number of roots found.

	2^(2^N)	combin	actual	Actual / combin
2	16	12	3	0.25
3	256	64	19	0.296875
4	65536	700	271	0.387147
5	4294967296	17424	8635	0.495580
6	1.8446744073E+019	1053696	421733	0.400241
7	3.4028236692E+038	160579584	107958091	0.672302