STARK by Hand

This document offers a concrete example of the RISC Zero Proof System; you can find a PDF version on www.RISCZero.com or peek behind the formulas with the Google Sheet Version.

For questions, corrections, conversation, and collaboration, find us on Twitter or Discord.

When any code executes in the RISC Zero virtual machine, each step of that execution is recorded in an **Execution Trace**. We show a simplified example, computing 4 steps of a Fibonacci sequence modulo 97, using two user-specified inputs. In this sheet, we introduce the columns of a RISC Zero Execution Trace. In the following sheets, we will demonstrate a concrete example of how RISC Zero proves the validity of an Execution Trace without revealing any knowledge.

In this example, our trace consists of 6 columns. Each of the first three columns is a record of the internal state of a register at each clock cycle from initialization until termination. We call these **Data Columns**. The next three columns are **Control Columns,** which we use to mark initialization and termination points.

In the full RISC Zero protocol, the **Data Columns** hold the state of the RISC-V processor, including ISA registers, the program counter, and various microarchitecture details such as instruction decoding data, ALU registers, etc., while the **Control Columns** handle system initialization and shutdown, the initial program code to load into memory before execution, and other control signals that don't depend on the programs execution.

Input 1	24
Input 2	30

Asserted Output	28

Modulo	97

	Clock Cycle	Data Column 1	Data Column 2	Data Column 3	Control Column - Initialization	Control Column - Transition	Control Column - Termination
Execution Trace - Initialization	0	24	30	54	1	0	0
Execution Trace - Transition	1	30	54	84	0	1	0
Execution Trace - Transition	2	54	84	41	0	1	0
Execution Trace - Termination	3	84	41	28	0	1	1

Checking the Trace

In this sheet, we introduce a number of rule-checking cells in order to demonstrate the validity of the Execution In this example, we show six rules. In the full RISC-V implementation, we check over 100 rules in order to validate the execution trace.

Each rule check is written as the product of two terms, modulo 97. The first term equals zero when the rule holds. The second term equals zero when we don't want to enforce the rule.

Each rule checking column can be expressed as a multi-input, single-output polynomial, where the inputs are some combination of entries in the trace; we call these **Rule-Checking Polynomials**.

Input 1	24
Input 2	30

Asserted Output	28

Modulo	97

	Clock Cycle	Data Column 1	Data Column 2	Data Column 3	Control Column - Initialization	Control Column - Transition	Control Column - Termination	Does Fibonacci relation hold?	Does Initialization for Data Column 1 match User Input 1?	Does Initialization for Data Column 2 match User Input 2?	Does Termination value for Data Column 3 match Output?	Does entry i from Input Column 1 match entry i-1 from Input Column 2?	Does entry i from Input Column 2 match entry i-1 from Output Column?
Execution Trace - Initialization	0	24	30	54	1	0	0	0	0	0	0	0	0
Execution Trace - Transition	1	30	54	84	0	1	0	0	0	0	0	0	0
Execution Trace - Transition	2	54	84	41	0	1	0	0	0	0	0	0	0
Execution Trace - Termination	3	84	41	28	0	1	1	0	0	0	0	0	0

Adding Random Padding

Before encoding each column as a polynomial, we append random padding to the end of the Execution Trace, which allows for a zero-knowledge protocol. This random noise is generated by the host system's cryptographically secure pseudorandom number generator. We set the Control columns to 0 for these random noise rows, in order to turn off our rule checks.

Input 1	24
Input 2	30

Asserted Output	28

Modulo	97

	Clock Cycle	Data Column 1	Data Column 2	Data Column 3	Control Column - Initialization	Control Column - Transition	Control Column - Termination	Does Fibonacci relation hold?	Does Initialization for Data Column 1 match User Input 1?	Does Initialization for Data Column 2 match User Input 2?	Does Termination value for Data Column 3 match Output?	Does entry i from Input Column 1 match entry i-1 from Input Column 2?	Does entry i from Input Column 2 match entry i-1 from Output Column?
Execution Trace - Initialization	0	24	30	54	1	0	0	0	0	0	0	0	0
Execution Trace - Transition	1	30	54	84	0	1	0	0	0	0	0	0	0
Execution Trace - Transition	2	54	84	41	0	1	0	0	0	0	0	0	0
Execution Trace - Termination	3	84	41	28	0	1	1	0	0	0	0	0	0
Random Padding	4	78	20	74	0	0	0	0	0	0	0	0	0
Random Padding	5	15	30	42	0	0	0	0	0	0	0	0	0
Random Padding	6	29	89	54	0	0	0	0	0	0	0	0	0
Random Padding	7	50	2	91	0	0	0	0	0	0	0	0	0

Interpolating Trace Polynomials

Let's remove the rule-checking columns for a minute and turn our attention toward encoding our Trace data in terms of polynomials. Just as any two points allow you to draw a line, any 8 points allow you to determine a degree 7 polynomial. The standard techniques for constructing a polynomial through a given set of points are Lagrange Interpolation and Finite Fourier Transforms (FFTs). In our context, we use NTTs (Number Theoretic Transforms), which are essentially just a finite field equivalent of an FFT. Since we're working modulo 97, our polynomials have values and coefficients in the finite field F_97.

The polynomial encoding technique we use in RISC Zero is called Reed-Solomon encoding; RS codes are ubiquitous in the world of digital signal processing as a method of providing redundance for error checking and error correction purposes.

Reed-Solomon Codes are built using points of the form (a^0, x_0), (a^1, x_1), ... In this example, we use powers of 28 as the inputs (and the entries of the trace as the outputs).

We use this technique to write a **Trace Polynomial** for each column of the trace. Running an iNTT on the 8 entries from Data Column 1, we use iNTT(column, modulus) to generate a polynomial whose evaluations agree with the trace data. Then, we evaluate this polynomial over an **Expanded Domain** to construct the Reed Solomon encoding of the column.

In Python using sympy, intt([**24**, **30**, **54**, **84**, **78**, **15**, **29**, **50**], prime=97) returns [*94*, *68*, *41*, *69*, *25*, *72*, *85*, *55*].

The 8 entries of this iNTT input array are shown in **boldface** in Data Column 1 below; with each entry corresponding to a row of the Padded Trace.

We use the entries of the output array as the coefficients of the associated Trace Polynomial. In this case, d_1(x) = *94 + 68x + 41x^2 + 69x^3 + 25x^4 +72x^5 +85x^6 +55x^7* (modulo 97).

The key feature of d_1(x) is that for z=5^0, 5^12, 5^24, ... the evaluations d_1(z) agree with the values in Data Column 1: d_1(5^0) = 24, d_1(5^12)=30, d_1(5^24) = 54, etc.

We proceed similarly on each data column and each control column, generating one Trace Polynomial for each column of our Trace.

A quick note about F_97: every element of this field can be written as a power of 5. In other words, the elements of F_97 are 0, 5^0, 5^1, ..., and 5^95. Written in this form, we can view our Reed-Solomon inputs as every third power of 5: 5^0, 5^3, 5^6, etc.
For a brief introduction to finite fields as they relate to the RISC Zero proof system, see here.

Input 1	24
Input 2	30

Asserted Output	28

Modulo	97

	Reed Solomon Input (Exponent Notation in terms of 28)	**Reed Solomon Input** (Exponent Notation in terms of 5)	**Reed Solomon Input** (Simplified)	**Clock Cycle**	**Data Column 1**	**Data Column 2**	**Data Column 3**	**Control Column - Initialization**	**Control Column - Transition**	**Control Column - Termination**
Execution Trace - Initialization	28^0 mod 97	5^0 mod 97	1	0	**24**	30	54	1	0	0
RS Redundancy Row	28^1 mod 97	5^3 mod 97	28		27	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row	28^2 mod 97	5^6 mod 97	8		74	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row	28^3 mod 97	5^9 mod 97	30		77	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
Execution Trace - Transition			64	1	**30**	54	84	0	1	0
RS Redundancy Row			46		37	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row			27		62	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row			77		3	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
Execution Trace - Transition			22	2	**54**	84	41	0	1	0
RS Redundancy Row			34		42	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row			79		96	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row			78		69	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
Execution Trace - Termination			50	3	**84**	41	28	0	0	1
RS Redundancy Row			42		36	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row			12		26	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row			45		37	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
Random Padding			96	4	**78**	95	77	0	0	0
RS Redundancy Row			69		71	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row			89		24	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row			67		70	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
Random Padding			33	5	**15**	52	7	0	0	0
RS Redundancy Row			51		31	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row			70		38	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row			20		71	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
Random Padding			75	6	**29**	82	12	0	0	0
RS Redundancy Row			63		40	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row			18		54	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row			19		19	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
Random Padding			47	7	**50**	12	26	0	0	0
RS Redundancy Row			55		80	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row			85		87	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
RS Redundancy Row	28^31 mod 97	5^93 mod 97	52		18	For each column, compute iNTT(Trace Column, modulus) to fill in this data				
	28^32 mod 97	5^96 mod 97	1		24					

97

5 order 96

64 5^12 we have a trace of length 8

5^3

x0, x1, x2, x3, x4, ..., x7

(g^0, x0), (g^1, x1), ... (g^7, x7)

Interpolating Trace Polynomials (cont.)

This sheet shows our 6 trace polynomials, each evaluated at 32 points.

intt(TraceColumn, prime=97) returns the coefficients of the trace polynomial.

Column	Python Code (from sympy import intt)	Code Output (Coefficients of Trace Polynomials)							
d1	d1 = intt([24, 30, 54, 84, 78, 15, 29, 50], prime=97)	94	68	41	69	25	72	85	55
d2	d2 = intt([30, 54, 84, 41, 2, 77, 21, 36], prime=97)	31	31	0	87	76	66	6	24
d3	d3 = intt([54, 84, 41, 28, 71, 17, 92, 33], prime=97)	4	14	83	44	12	44	12	35
c1	c1 = intt([1, 0, 0, 0, 0, 0, 0, 0], prime=97)	85	85	85	85	85	85	85	85
c2	c2 = intt([0, 1, 1, 1, 0, 0, 0, 0], prime=97)	61	80	12	37	12	60	12	17
c3	c3 = intt([0, 0, 0, 1, 0, 0, 0, 0], prime=97)	85	89	27	18	12	8	70	79

Trace Polynomials
d1(x)=94+68x+41x^2+69x^3+25x^4+72x^5+85x^6+55x^7
d2(x)=31+31x+0x^2+87x^3+76x^4+66x^5+6x^6+24x^7
d3(x)=4+14x+83x^2+44x^3+12x^4+44x^5+12x^6+35x^7
c1(x)=85+85x+85x^2+85x^3+85x^4+85x^5+85x^6+85x^7
c2(x)=61+80x+12x^2+37x^3+12x^4+60x^5+12x^6+17x^7
c3(x)=85+89x+27x^2+18x^3+12x^4+8x^5+70x^6+79x^7

Input 1	24
Input 2	30

Asserted Output	28

Modulo	97

	Reed Solomon Input (Exponent Notation in terms of 28)	**Reed Solomon Input** (Exponent Notation in terms of 5)	**Reed Solomon Input** (Simplified)	**Clock Cycle**	**Data Column 1**	**Data Column 2**	**Data Column 3**	**Control Column - Initialization**	**Control Column - Transition**	**Control Column - Termination**
Execution Trace - Initialization	28^0 mod 97	5^0 mod 97	1	**0**	**24**	**30**	**54**	**1**	**0**	**0**
RS Redundancy Row	28^1 mod 97	5^3 mod 97	28		27	33	43	23	35	26
RS Redundancy Row	28^2 mod 97	5^6 mod 97	8		74	14	27	45	31	13
RS Redundancy Row	28^3 mod 97	5^9 mod 97	30		77	31	88	53	63	86
Execution Trace - Transition			64	**1**	**30**	**54**	**84**	**0**	**1**	**0**
RS Redundancy Row			46		37	76	67	72	32	46
RS Redundancy Row			27		62	85	34	83	63	87
RS Redundancy Row			77		3	84	63	70	30	80
Execution Trace - Transition			22	**2**	**54**	**84**	**41**	**0**	**1**	**0**
RS Redundancy Row			34		42	14	11	10	58	60
RS Redundancy Row			79		96	18	86	60	59	28
RS Redundancy Row			78		69	3	29	25	20	91
Execution Trace - Termination			50	**3**	**84**	**41**	**28**	**0**	**1**	**1**
RS Redundancy Row			42		36	15	54	53	8	23
RS Redundancy Row			12		26	16	31	11	91	45
RS Redundancy Row			45		37	77	1	68	51	53
Random Padding			96	**4**	**78**	**2**	**71**	**0**	**0**	**0**
RS Redundancy Row			69		71	90	82	2	38	72
RS Redundancy Row			89		24	15	14	62	57	83
RS Redundancy Row			67		70	66	75	13	66	70
Random Padding			33	**5**	**15**	**77**	**17**	**0**	**0**	**0**
RS Redundancy Row			51		31	8	76	26	65	10
RS Redundancy Row			70		38	20	67	13	36	60
RS Redundancy Row			20		71	19	14	86	9	25
Random Padding			75	**6**	**29**	**21**	**92**	**0**	**0**	**0**
RS Redundancy Row			63		40	43	42	46	81	53
RS Redundancy Row			18		54	72	72	87	86	11
RS Redundancy Row			19		19	92	90	80	70	68
Random Padding			47	**7**	**50**	**36**	**33**	**0**	**0**	**0**
RS Redundancy Row			55		80	66	45	60	74	2
RS Redundancy Row			85		87	8	89	28	65	62
RS Redundancy Row	28^31 mod 97		52		18	70	60	91	82	13
	28^32 mod 97		1							

Committing Trace Polynomials

The next step is for the Prover to commit the **Trace Polynomials** into a **Merkle Tree**. In order to maintain a Zero-Knowledge protocol, the Prover evaluates each Trace Polynomial over a "shifted evaluation domain." Specifically, we evaluate each d_i(x) at x=5, 5^4, 5^7, ..., 5^93.

Notice that because of our shifted evaluation domain, the yellow and blue cells in Data Columns 1, 2, and 3 no longer match the Inputs and Asserted Outputs. In fact, this shift in the evaluation domain disguises *all* the Trace Data. We only reveal information about the disguised trace, and the random padding we appended is sufficient to prevent an attacker from deducing any connection between the disguised trace and the actual trace.

Trace Polynomials
d1(x)=94+68x+41x^2+69x^3=25x^4+72x^5+85x^6+55x^7
d2(x)=31+31x+0x^2+87x^3=76x^4+66x^5+6x^6+24x^7
d3(x)=4+14x+83x^2+44x^3=12x^4+44x^5+12x^6+35x^7
c1(x)=85+85x+85x^2+85x^3=85x^4+85x^5+85x^6+85x^7
c2(x)=61+80x+12x^2+37x^3=12x^4+60x^5+12x^6+17x^7
c3(x)=85+89x+27x^2+18x^3=12x^4+8x^5+70x^6+79x^7

Input 1	24
Input 2	30
Modulo	97

Asserted Output	28

	Reed Solomon Input (Exponent Notation in terms of 28)	**Reed Solomon Input** (Exponent Notation in terms of 5)	**Reed Solomon Input** (Simplified)	**Clock Cycle**	**Data Column 1**	**Data Column 2**	**Data Column 3**	**Control Column - Initialization**	**Control Column - Transition**	**Control Column - Termination**
Disguised Execution Trace	5*28^0 mod 97	5^1 mod 97	5	**0**	**31**	**39**	**12**	**82**	**81**	**2**
Disguised RS Redundancy Row	5*28^1 mod 97	5^4 mod 97	43		15	36	11	18	72	32
Disguised RS Redundancy Row	5*28^2 mod 97	5^7 mod 97	40		96	65	6	32	73	65
Disguised RS Redundancy Row	5*28^3 mod 97	5^10 mod 97	53		79	41	88	68	10	22
Disguised Execution Trace			29	**1**	**69**	**35**	**49**	**81**	**64**	**32**
Disguised RS Redundancy Row			36		31	85	50	41	58	16
Disguised RS Redundancy Row			38		16	41	69	18	40	38
Disguised RS Redundancy Row			94		71	24	89	86	56	50
Disguised Execution Trace			13	**2**	**35**	**77**	**46**	**92**	**16**	**47**
Disguised RS Redundancy Row			73		10	40	9	59	83	24
Disguised RS Redundancy Row			7		53	54	58	14	20	67
Disguised RS Redundancy Row			2		28	7	95	44	92	35
Disguised Execution Trace			56	**3**	**67**	**81**	**80**	**43**	**61**	**82**
Disguised RS Redundancy Row			16		26	2	73	31	21	18
Disguised RS Redundancy Row			60		0	54	76	8	64	32
Disguised RS Redundancy Row			31		45	36	80	92	4	68
Disguised Random Padding			92	**4**	**91**	**59**	**35**	**10**	**22**	**81**
Disguised RS Redundancy Row			54		94	39	57	71	34	41
Disguised RS Redundancy Row			57		18	82	19	50	40	18
Disguised RS Redundancy Row			44		80	36	44	89	28	86
Disguised Random Padding			68	**5**	**54**	**12**	**61**	**2**	**48**	**92**
Disguised RS Redundancy Row			61		39	46	78	32	64	59
Disguised RS Redundancy Row			59		16	16	16	65	72	14
Disguised RS Redundancy Row			3		19	49	95	22	31	44
Disguised Random Padding			84	**6**	**57**	**23**	**47**	**32**	**55**	**43**
Disguised RS Redundancy Row			24		74	89	18	16	37	31
Disguised RS Redundancy Row			90		40	96	42	38	26	8
Disguised RS Redundancy Row			95		43	54	72	50	9	92
Disguised Random Padding			41	**7**	**57**	**19**	**90**	**47**	**44**	**10**
Disguised RS Redundancy Row			81		75	8	27	24	22	71
Disguised RS Redundancy Row			37		28	34	37	67	56	50
Disguised RS Redundancy Row	5*28^31 mod 97		66		96	1	51	35	64	89
	28^32 mod 97		5							

Introducing Constraint Polynomials

Now that we've encoded our Trace data into Trace polynomials, let's return to our original Reed-Solomon domain and add back in our Rule Checking cells.

Of course, we shouldn't expect these rule checks to evaluate to 0 in the redundancy rows, as they're not directly associated with the data from the trace.

Conveniently, by writing these rule checks in terms of our trace polynomials, we can convert our multi-input rule checking polynomials into single-input polynomials, which we call **Constraint Polynomials.**

Note that each Constraint Polynomials will evaluate to 0 at the RS input values that are associated with actual trace data.

Column	**Python Code** (from sympy import intt)	**Code Output** (Coefficients of Trace Polynomials)							
d1	d1 = intt([24, 30, 54, 84, 78, 15, 29, 50], prime=97)	94	68	41	69	25	72	85	55
d2	d2 = intt([30, 54, 84, 41, 2, 77, 21, 36], prime=97)	31	31	0	87	76	66	6	24
d3	d3 = intt([54, 84, 41, 28, 71, 17, 92, 33], prime=97)	4	14	83	44	12	44	12	35
c1	c1 = intt([1, 0, 0, 0, 0, 0, 0, 0], prime=97)	85	85	85	85	85	85	85	85
c2	c2 = intt([0, 1, 1, 1, 0, 0, 0, 0], prime=97)	61	80	12	37	12	60	12	17
c3	c3 = intt([0, 0, 0, 1, 0, 0, 0, 0], prime=97)	85	89	27	18	12	8	70	79

Input 1	24
Input 2	30

Asserted Output	28

Modulo	97

	Reed Solomon Input (Exponent Notation in terms of 28)	**Reed Solomon Input** (Exponent Notation in terms of 5)	**Reed Solomon Input** (Simplified)	**Clock Cycle**	**Data Column 1**	**Data Column 2**	**Data Column 3**	**Control Column - Initialization**	**Control Column - Transition**	**Control Column - Termination**	**Does Fibonacci relation hold?**	**Does Initialization for Data Column 1 match User Input 1?**	**Does Initialization for Data Column 2 match User Input 2?**	**Does Termination value for Data Column 3 match Output?**	**Does entry i from Input Column 1 match entry i-1 from Input Column 2?**	**Does entry i from Input Column 2 match entry i-1 from Output Column?**
Execution Trace - Initialization	28^0 mod 97	5^0 mod 97	1	**0**	**24**	**30**	**54**	**1**	**0**	**0**	0	0	0	0	0	0
RS Redundancy Row	28^1 mod 97	5^3 mod 97	28		27	33	43	23	35	26	27	69	69	2	90	65
RS Redundancy Row	28^2 mod 97	5^6 mod 97	8		74	14	27	45	31	13	3	19	56	84	9	3
RS Redundancy Row	28^3 mod 97	5^9 mod 97	30		77	31	88	53	63	86	34	93	53	19	53	16
Execution Trace - Transition			64	**1**	**30**	**54**	**84**	**0**	**1**	**0**	0	0	0	0	0	0
RS Redundancy Row			46		37	76	67	72	32	46	84	63	14	48	31	86
RS Redundancy Row			27		62	85	34	83	63	87	55	50	6	37	17	65
RS Redundancy Row			77		3	84	63	70	30	80	45	82	94	84	33	74
Execution Trace - Transition			22	**2**	**54**	**84**	**41**	**0**	**1**	**0**	0	0	0	0	0	0
RS Redundancy Row			34		42	14	11	10	58	60	60	83	34	47	65	30
RS Redundancy Row			79		96	18	86	60	59	28	55	52	56	72	67	26
RS Redundancy Row			78		69	3	29	25	20	91	69	58	4	91	88	61
Execution Trace - Termination			50	**3**	**84**	**41**	**28**	**0**	**1**	**1**	0	0	0	0	0	0
RS Redundancy Row			42		36	15	54	53	8	23	58	54	78	16	79	32
RS Redundancy Row			12		26	16	31	11	91	45	32	22	40	38	49	32
RS Redundancy Row			45		37	77	1	68	51	53	61	11	92	24	85	23
Random Padding			96	**4**	**78**	**2**	**71**	**0**	**0**	**0**	0	0	0	0	0	0
RS Redundancy Row			69		71	90	82	2	38	72	76	94	23	8	91	10
RS Redundancy Row			89		24	15	14	62	57	83	91	0	40	2	68	58
RS Redundancy Row			67		70	66	75	13	66	70	29	16	80	89	23	22
Random Padding			33	**5**	**15**	**77**	**17**	**0**	**0**	**0**	0	0	0	0	0	0
RS Redundancy Row			51		31	8	76	26	65	10	51	85	10	92	45	40
RS Redundancy Row			70		38	20	67	13	36	60	11	85	64	12	52	22
RS Redundancy Row			20		71	19	14	86	9	25	95	65	24	38	45	78
Random Padding			75	**6**	**29**	**21**	**92**	**0**	**0**	**0**	0	0	0	0	0	0
RS Redundancy Row			63		40	43	42	46	81	53	89	57	16	63	70	43
RS Redundancy Row			18		54	72	72	87	86	11	55	88	65	96	14	42
RS Redundancy Row			19		19	92	90	80	70	68	78	85	13	45	0	28
Random Padding			47	**7**	**50**	**36**	**33**	**0**	**0**	**0**	0	0	0	0	0	0
RS Redundancy Row			55		80	66	45	60	74	2	38	62	26	34	22	30
RS Redundancy Row			85		87	8	89	28	65	62	40	18	63	96	5	11
RS Redundancy Row	28^31 mod 97		52		18	70	60	91	82	13	30	36	51	28	43	9
	28^32 mod 97		1													

Mixing Constraint Polynomials

In this sheet, we add one new column, which Mixes our **Constraint Polynomials** into a single **Constraint Polynomial**.

After the Prover sends a Merkle root for each trace polynomial, the verifier responds with a **Constraint Mixing Parameter,** ***a***.

Letting c_i denote the constraint polynomials for i=0,1,2,3,4,5, we write C(x) = a^0 * c_0(x) + a^1 c_1(x) + ... + a^5 *c_5(x)

Note that if each c_i evaluates to 0 at some input *x*, then C will also evaluate to 0 for that input.

In this example, the degree of the Mixed Constraint Polynomial is equal to the degree of the Trace Polynomials, because the Rule-Checking involved is particularly simple. In more complicated examples, composing our Rule Checking Polynomials with our Trace Polynomials would yield "High Degree Constraint Polynomials." In that case, we'd add an extra step at this point to split our "High Degree Mixed Constraint Polynomial" into a few "Low Degree Mixed Constraint Polynomials."

Column	**Python Code** (from sympy import intt)	**Code Output** (Coefficients of Trace Polynomials)							
d1	d1 = intt([24, 30, 54, 84, 78, 15, 29, 50], pr	94	68	41	69	25	72	85	55
d2	d2 = intt([30, 54, 84, 41, 2, 77, 21, 36], prime=97)	31	31	0	87	76	66	6	24
d3	d3 = intt([54, 84, 41, 28, 71, 17, 92, 33], prime=97)	4	14	83	44	12	44	12	35
c1	c1 = intt([1, 0, 0, 0, 0, 0, 0, 0], prime=97)	85	85	85	85	85	85	85	85
c2	c2 = intt([0, 1, 1, 1, 0, 0, 0, 0], prime=97)	61	80	12	37	12	60	12	17
c3	c3 = intt([0, 0, 0, 1, 0, 0, 0, 0], prime=97)	85	89	27	18	12	8	70	79

Input 1	24
Input 2	30

Asserted Output	28
Constraint Mixing Parameter	3

Modulo	97

	Reed Solomon Input (Exponent Notation in terms of 5)	**Reed Solomon Input** (Simplified)	**Clock Cycle**	**Data Column 1**	**Data Column 2**	**Data Column 3**	**Control Column - Initialization**	**Control Column - Transition**	**Control Column - Termination**	**Mixed Constraint Polynomial**	**Does Fibonacci relation hold?**	**Does Initialization for Data Column 1 match User Input 1?**	**Does Initialization for Data Column 2 match User Input 2?**	**Does Termination value for Data Column 3 match Output?**	**Does entry i from Input Column 1 match entry i-1 from Input Column 2?**	**Does entry i from Input Column 2 match entry i-1 from Output Column?**
Execution Trace - Initialization	5^1 mod 97	5	**0**	**31**	**39**	**12**	**82**	**81**	**2**	52	33	89	59	65	2	40
RS Redundancy Row	5^4 mod 97	43		15	36	11	18	72	32	47	67	32	11	38	19	66
RS Redundancy Row	5^7 mod 97	40		96	65	6	32	73	65	45	34	73	53	25	64	7
RS Redundancy Row	5^10 mod 97	53		79	41	88	68	10	22	32	1	54	69	59	4	94
Execution Trace - Transition		29	**1**	**69**	**35**	**49**	**81**	**64**	**32**	86	62	56	17	90	77	17
RS Redundancy Row		36		31	85	50	41	58	16	77	73	93	24	61	1	24
RS Redundancy Row		38		16	41	69	18	40	38	95	85	50	4	6	77	42
RS Redundancy Row		94		71	24	89	86	56	50	62	12	65	66	43	31	5
Execution Trace - Transition		13	**2**	**35**	**77**	**46**	**92**	**16**	**47**	80	52	42	56	70	0	60
RS Redundancy Row		73		10	40	9	59	83	24	61	81	47	8	29	80	43
RS Redundancy Row		7		53	54	58	14	20	67	6	95	18	45	70	46	88
RS Redundancy Row		2		28	7	95	44	92	35	45	75	79	55	17	77	22
Execution Trace - Termination		56	**3**	**67**	**81**	**80**	**43**	**61**	**82**	27	59	6	59	93	69	1
RS Redundancy Row		16		26	2	73	31	21	18	54	46	62	5	34	94	47
RS Redundancy Row		60		0	54	76	8	64	32	73	57	2	95	81	36	35
RS Redundancy Row		31		45	36	80	92	4	68	23	30	89	67	44	55	55
Random Padding		92	**4**	**91**	**59**	**35**	**10**	**22**	**81**	79	3	88	96	82	26	23
RS Redundancy Row		54		94	39	57	71	34	41	63	59	23	57	25	24	8
RS Redundancy Row		57		18	82	19	50	40	18	43	79	88	78	32	15	46
RS Redundancy Row		44		80	36	44	89	28	86	44	31	37	49	18	68	29
Random Padding		68	**5**	**54**	**12**	**61**	**2**	**48**	**92**	16	66	60	61	29	51	60
RS Redundancy Row		61		39	46	78	32	64	59	65	79	92	27	40	0	72
RS Redundancy Row		59		16	16	16	65	72	14	52	9	62	60	26	1	75
RS Redundancy Row		3		19	49	95	22	31	44	18	0	84	30	38	55	58
Random Padding		84	**6**	**57**	**23**	**47**	**32**	**55**	**43**	4	75	86	67	41	50	44
RS Redundancy Row		24		74	89	18	16	37	31	18	42	24	71	78	66	19
RS Redundancy Row		90		40	96	42	38	26	8	68	22	26	83	15	42	43
RS Redundancy Row		95		43	54	72	50	9	92	7	8	77	36	71	43	19
Random Padding		41	**7**	**57**	**19**	**90**	**47**	**44**	**10**	4	56	96	65	38	41	29
RS Redundancy Row		81		75	8	27	24	22	71	22	44	60	54	26	80	71
RS Redundancy Row		37		28	34	37	67	56	50	62	40	74	74	62	72	37
RS Redundancy Row		66		96	1	51	35	64	89	57	82	95	52	10	69	15
		5														

The Core of the RISC Zero STARK

The Prover constructs the **Validity Polynomial** by dividing the **Constraint Polynomial** from the previous sheet by the publicly known **Zeros Polynomial**.

$V(x) = C(x) / Z(x)$

In our example, the **Zeros Polynomial** is $Z(x) = (x-1)*(x-47)*(x-75)*(x-33)*(x-96)*(x-50)*(x-22)*(x-64)$

Normally when we divide two low degree polynomials, we don't expect to get another low degree polynomial. But for an honest prover, it's not hard to see that V(x) will be lower degree than C(x), since the roots of Z(x) line up perfectly with roots of C(x) (see previous page)

The Prover evaluates V(x) over the "shifted evaluation domain" shown below, commits the values to a Merkle Tree, and sends the Merkle root to the verifier.

The construction of these polynomials is the core conceptual thrust of RISC Zero's Proof of Trace Validity. All of the information necessary to confirm the validity of the original Execution trace can be described in the following assertions about these polynomials:

(i) $V(x) = C(x) / Z(x)$ for all x

(ii) The degree of the Validity Polynomial and each Trace Polynomials are less than or equal to 7.

The FRI protocol is the technique we use for proving (ii). Those details are omitted from this simplified example.

In the original STARK protocol, the Verifier tests (i) at a number of test points; the soundness of the protocol depends on the number of tests. The DEEP-ALI technique allows us to achieve a high degree of soundness with a single test. The details of DEEP are described in the following sheet.

Input 1	24
Input 2	30

Asserted Output	28
Constraint Mixing Parameter	3

Modulo	97

	Reed Solomon Input (Exponent Notation in terms of 28)	**Reed Solomon Input** (Exponent Notation in terms of 5)	**Reed Solomon Input** (Simplified)	**Clock Cycle**	**Data Column 1**	**Data Column 2**	**Data Column 3**	**Control Column - Initialization**	**Control Column - Transition**	**Control Column - Termination**	**Mixed Constraint Polynomial**	**Zeros Polynomial, Z (x)**	**Validity Polynomial, V (x) = C(x) / Z(x)**
Execution Trace - Initialization	28^0 mod 97	5^0 mod 97	5	**0**	**31**	**39**	**12**	**82**	**81**	**2**	52	5	88
RS Redundancy Row	28^1 mod 97	5^3 mod 97	43		15	36	11	18	72	32	47	34	67
RS Redundancy Row	28^2 mod 97	5^6 mod 97	40		96	65	6	32	73	65	45	90	49
RS Redundancy Row	28^3 mod 97	5^9 mod 97	53		79	41	88	68	10	22	32	61	53
Execution Trace - Transition			29	**1**	**69**	**35**	**49**	**81**	**64**	**32**	86	5	56
RS Redundancy Row			36		31	85	50	41	58	16	77	34	85
RS Redundancy Row			38		16	41	69	18	40	38	95	90	28
RS Redundancy Row			94		71	24	89	86	56	50	62	61	36
Execution Trace - Transition			13	**2**	**35**	**77**	**46**	**92**	**16**	**47**	80	5	16
RS Redundancy Row			73		10	40	9	59	83	24	61	34	56
RS Redundancy Row			7		53	54	58	14	20	67	6	90	13
RS Redundancy Row			2		28	7	95	44	92	35	45	61	23
Execution Trace - Termination			56	**3**	**67**	**81**	**80**	**43**	**61**	**82**	27	5	83
RS Redundancy Row			16		26	2	73	31	21	18	54	34	13
RS Redundancy Row			60		0	54	76	8	64	32	73	90	45
RS Redundancy Row			31		45	36	80	92	4	68	23	61	29
Random Padding			92	**4**	**91**	**59**	**35**	**10**	**22**	**81**	79	5	74
RS Redundancy Row			54		94	39	57	71	34	41	63	34	96
RS Redundancy Row			57		18	82	19	50	40	18	43	90	77
RS Redundancy Row			44		80	36	44	89	28	86	44	61	85
Random Padding			68	**5**	**54**	**12**	**61**	**2**	**48**	**92**	16	5	42
RS Redundancy Row			61		39	46	78	32	64	59	65	34	39
RS Redundancy Row			59		16	16	16	65	72	14	52	90	48
RS Redundancy Row			3		19	49	95	22	31	44	18	61	48
Random Padding			84	**6**	**57**	**23**	**47**	**32**	**55**	**43**	4	5	59
RS Redundancy Row			24		74	89	18	16	37	31	18	34	69
RS Redundancy Row			90		40	96	42	38	26	8	68	90	18
RS Redundancy Row			95		43	54	72	50	9	92	7	61	51
Random Padding			41	**7**	**57**	**19**	**90**	**47**	**44**	**10**	4	5	59
RS Redundancy Row			81		75	8	27	24	22	71	22	34	52
RS Redundancy Row			37		28	34	37	67	56	50	62	90	5
RS Redundancy Row	28^31 mod 97		66		96	1	51	35	64	89	57	61	55
	28^32 mod 97		5										

Constructing the DEEP Polynomials

In this sheet, we use the Trace Polynomials and the Validity Polynomial(s) to construct the DEEP Polynomials. The DEEP polynomials allow the Verifier to test V(x) = C(x) / Z(x) outside the original Merkle tree commitments, which substantially improves the robustness of the Verifier's test.

With commitments of the trace polynomials and the validity polynomial in place, the Verifier picks a random field element z. We use z=93 in this example. The Verifier would like to be able to compute the Mixed constraint polynomial, C(93). The Prover sends V(z) and the **necessary evaluations** of the Trace Polynomials to allow the Verifer to compute C(93).

In order to enable the Verifier to make this computation, the Prover supplies the **necessary evaluations**: d_1(93), d_2(93), d_3(93), c_1(93), c_2(93), c_3(93), d_2(93*5^-12), d_3(93*5^-12). This 5^-12 is a pointer backwards 1 computational step and allows for checking the rules that span multiple clock-cycles.

These 8 points are called the **taps of the trace** at 93. These 8 points, together with the publicly known rule-checking functions, allow the Verifier to manually compute C(93) and therefore V(93).

The Prover also constructs the DEEP polynomials, interpolates each one, and sends the coefficients of each DEEP polynomial to the Verifier. The DEEP polynomials are defined as follows:

d`_1(x) = (d_1(x) - d_1(93)) / (x - 93)

d`_2(x) = (d_2(x) - dbar_2(x)) / ((x-93)(x-6)) where dbar_2(x) is constructed by interpolating (6, d_2(6)) and (93, d_2(93)

d`_3(x) = (d_3(x) - dbar_3(x)) / ((x-93)(x-6)) where dbar_3(x) is constructed by interpolating (6, d_3(6)) and (93, d_3(93)

c`_1(x) = (c_1(x) - c_1(93)) / (x - 93)

c`_2(x) = (c_2(x) - c_2(93)) / (x - 93)

c`_3(x) = (c_3(x) - c_3(93)) / (x - 93)

V`(x) = (V(x) - V(93)) / (x - 9 where the Prover computes V(93) by running iNTT(ValidityColumn) and then evaluating at 93.

intt([88,67,49,53,56,85,28,36,16,56,13,23,83,13,45,29,74,96,77,85,42,39,48,48,59,69,18,51,59,52,5,55], prime=97)

V coefficients [96, 83, 47, 58, 36, 60, 96, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Without the DEEP technique, the Prover would assert that d_1, d_2, d_3, c_1, c_2, c_3, and V were all low degree polynomials.

With the DEEP technique, the Prover argues instead that d`_1, d`_2, d`_3, c`_1, c`_2, c`_3, and V` are low degree polynomials

Input 1	24
Input 2	30

Modulo	97

Asserted Output	28
Constraint Mixing Parameter	3
Random Test Point	93

	Reed Solomon Input (Exponent Notation in terms of 28)	**Reed Solomon Input** (Exponent Notation in terms of 5)	**Reed Solomon Input** (Simplified)	**Clock Cycle**	**Data Column 1**	**DEEP Polynomial 1**	**Data Column 2**	**DEEP Polynomial 2**	**Data Column 3**	**DEEP Polynomial 3**	**Control Column - Initialization**	**DEEP Polynomial 4**	**Control Column - Transition**	**DEEP Polynomial 5**	**Control Column - Termination**	**DEEP Polynomial 6**	**Validity Polynomial, V (x) = C(x) / Z(x)**	**DEEP Validity Polynomial**
Disguised Execution Trace	5*28^0 mod 97	5^1 mod 97	5	0	31	50	39	19	12	84	82	47	81	4	2	95	88	53
Disguised RS Redundancy Row	5*28^1 mod 97	5^4 mod 97	43		15	34	36	74	11	63	18	84	72	79	32	89	67	84
Disguised RS Redundancy Row	5*28^2 mod 97	5^7 mod 97	40		96	58	65	82	6	89	32	68	73	80	65	87	49	32
Non-Merkle Points	5^2*28^2 mod 97	5^8 mod 97	**6**				**71**		**96**									
		5^9 mod 97																
Disguised RS Redundancy Row	5*28^3 mod 97	5^10 mod 97	53		79	70	41	13	88	93	68	31	10	13	22	63	53	52
Disguised Execution Trace		5^13 mod 97	29	1	69	53	35	48	49	75	81	51	64	77	32	18	56	37
Disguised RS Redundancy Row		5^16 mod 97	36		31	84	85	30	50	36	41	92	58	27	16	29	85	7
Disguised RS Redundancy Row	5*28^6 mod 97	5^19 mod 97	38		16	45	41	17	69	11	18	94	40	53	38	42	28	3
Non-Merkle Points	5^2*28^6 mod 97	5^20 mod 97	93		66		6		26		47		45		20		96	
		5^21 mod 97	77															
Disguised RS Redundancy Row		5^22 mod 97	94		71	5	24	58	89	80	86	39	56	11	50	30	36	37
Disguised Execution Trace			13	2	35	21	77	93	46	44	92	54	16	4	47	13	16	1
Disguised RS Redundancy Row			73		10	61	40	63	9	23	59	77	83	66	24	58	56	2
Disguised RS Redundancy Row			7		53	87	54	64	58	40	14	94	20	33	67	66	13	63
Disguised RS Redundancy Row			2		28	26	7	42	95	11	44	48	92	24	35	51	23	4
Disguised Execution Trace			56	3	67	76	81	45	80	80	43	84	61	52	82	56	83	79

(For readability, we've abbreviated the trace.)

The FRI Polynomial

After using the DEEP polynomials to check the relation between the Trace Polynomial, the Validity Polynomial, and the Zeros Polynomial at z=93, the only thing left for the Prover to do is to show that the DEEP polynomials are low-degree.

The FRI protocol provides a mechanism for the Verifier to confirm the low-degree-ness of polynomials, with very little computation required of the Verifier. In order to reduce this assertion of low-degree-ness to a single application of FRI, the Prover mixes the DEEP polynomials into a single FRI polynomial, using the DEEP Mixing parameter.

Letting c`_1, c`_2, c`_3, d`_1, d`_2, d`_3, and V` denote the DEEP polynomials, we mix the DEEP polynomials to construct the FRI polynomial,
f(x) = b^0 * c`_1(x) + b^1 c`_2(x) + ... + b^6 *V`(x)

To complete the argument, the Prover constructs a FRI proof that f(x) is a low degree polynomial.

With this process, the Prover has constructed a zero-knowledge argument of computational integrity that can be verified incredibly quickly.

Input 1	24
Input 2	30
Modulo	97

Asserted Output	28
Constraint Mixing Parameter	3
Random Test Point	93
DEEP Mixing Parameter	21

	Reed Solomon Input (Exponent Notation in terms of 28)	**Reed Solomon Input** (Exponent Notation in terms of 5)	**Reed Solomon Input (Simplified)**	**Clock Cycle**	**DEEP Polynomial 1**	**DEEP Polynomial 2**	**DEEP Polynomial 3**	**DEEP Polynomial 4**	**DEEP Polynomial 5**	**DEEP Polynomial 6**	**DEEP Validity Polynomial**	**FRI Polynomial**
Disguised Execution Trace	5*28^0 mod 97	5^1 mod 97	5	0	50	19	84	47	4	95	53	53
Disguised RS Redundancy Row	5*28^1 mod 97	5^4 mod 97	43		34	74	63	84	79	89	84	69
Disguised RS Redundancy Row	5*28^2 mod 97	5^7 mod 97	40		58	82	89	68	80	87	32	63
Disguised RS Redundancy Row	5*28^3 mod 97	5^10 mod 97	53		70	13	93	31	13	63	52	30
Disguised Execution Trace		5^13 mod 97	29	1	53	48	75	51	77	18	37	46
Disguised RS Redundancy Row		5^16 mod 97	36		84	30	36	92	27	29	7	13
Disguised RS Redundancy Row	5*28^6 mod 97	5^19 mod 97	38		45	17	11	94	53	42	3	60
Disguised RS Redundancy Row		5^22 mod 97	94		5	58	80	39	11	30	37	50
Disguised Execution Trace			13	2	21	93	44	54	4	13	1	38
Disguised RS Redundancy Row			73		61	63	23	77	66	58	2	3
Disguised RS Redundancy Row			7		87	64	40	94	33	66	63	95
Disguised RS Redundancy Row			2		26	42	11	48	24	51	4	23
Disguised Execution Trace			56	3	76	45	80	84	52	56	79	75
Disguised RS Redundancy Row			16		95	10	33	38	57	29	88	39
Disguised RS Redundancy Row			60		2	21	38	10	20	79	28	62
Disguised RS Redundancy Row			31		77	71	50	29	21	18	84	19
Disguised Random Padding			92	4	72	1	72	37	23	36	22	62
Disguised RS Redundancy Row			54		54	52	33	74	55	89	0	58
Disguised RS Redundancy Row			57		66	72	35	8	19	27	14	41
Disguised RS Redundancy Row			44		69	71	78	13	34	62	22	67
Disguised Random Padding			68	5	16	60	8	60	93	1	72	89
Disguised RS Redundancy Row			61		16	10	8	52	57	20	23	41
Disguised RS Redundancy Row			59		30	8	86	28	42	23	87	50
Disguised RS Redundancy Row			3		21	44	36	38	95	45	7	24
Disguised Random Padding			84	6	1	41	9	34	42	19	58	95
Disguised RS Redundancy Row			24		28	5	25	37	69	87	51	90
Disguised RS Redundancy Row			90		41	14	88	3	71	4	26	72
Disguised RS Redundancy Row			95		37	16	95	50	79	36	26	20
Disguised Random Padding			41	7	58	49	65	0	28	86	66	82
Disguised RS Redundancy Row			81		72	15	22	10	10	20	36	33
Disguised RS Redundancy Row			37		18	78	50	62	5	93	38	0
Disguised RS Redundancy Row	5*28^31 mod 97	5^93 mod 97	66		42	41	91	22	46	19	59	16
	5*28^32 mod 97	5^96 mod 97	5									

FRI Folding

Given a vector commitment, the FRI protocol proves that the commitment corresponds to evaluations of a low-degree polynomial. In this example, we use FRI to prove that the "FRI Polynomial" commitment (from the previous page) has degree at most 7.

The "FRI blow-up factor" here is 4, since the commitment for the FRI polynomial has 32 entries and a degree 7 polynomial has 8 coefficients. This blow-up factor is a consequence of the choice of "rate" for the Reed-Solomon expansion used earlier. A FRI blow-up factor of 4 corresponds to an RS code of rate 1/4.

FRI consists of a commit phase and a query phase. The commit phase consists of *r* rounds. In each round, the prover "folds" the previous commitment into a smaller commitment (both in terms of commitment size and polynomial degree).

Here, we show 3 rounds using a folding factor of 2: in each round, the Prover commits to a vector whose length is half that of the previous commitment.

The folding at each round is accomplished by first splitting the coefficients into an even part and an odd part and then mixing the two parts together using verifier-supplied randomness.

This page shows the coefficient form for the polynomials at each round; the commitments at each round are shown on the following page.

f_0	**19 + 56x + 34x^2 + 48x^3 + 43x^4 + 37x^5 + 10x^6 + 0x^7**
Even Part of f_0	19 + 34x + 43x^2 + 10x^3
Odd Part of f_0	56 + 48x + 37x^2 + 0x^3

Round 1 Randomness: 12

f_1 = [Even Part of f_0] + 12*[Odd Part of f_0]

f_1	**12 + 28x + 2x^2 + 10x^3**
Even Part of f_1	12 + 2x
Odd Part of f_1	28 + 10x

Round 2 Randomness: 32

f_2 = [Even Part of f_1] + 32*[Odd Part of f_1]

f_2	**35 + 31x**
Even Part of f_2	35 + x
Odd Part of f_2	31 + x

Round 3 Randomness 64

f_3 = [Even Part of f_2] + 64*[Odd Part of f_2]

f_3	**79**

FRI Queries

The Prover has committed to evaluations of f_0 at powers of 28, evaluations of f_1 at powers of 28^2, evaluations of f_2 at powers of 28^4, and evaluations of f_3 at powers of 28^8.

Since 28 has multiplicative order 32, this corresponds to evaluation domains of 32 elements, 16 elements, 8 elements, and 4 elements.

After these commitments are made, the Verifier makes a number of *queries*. The queries serve as a random challenge, testing the legitimacy of the Prover's commitments. Loosely speaking, with a blow-up factor of 4, a single query will catch a cheating Prover 3/4 of the time. In other words, a single query provides 2 bits of security. The RISC Zero zkVM uses 50 queries and a blow-up factor of 4, which amounts to 100 bits of security.

Note that the analysis above is a substantial simplification of the full security analysis; the precise security level is not exactly 100 bits. For a more thorough security analysis, see our [ZKP Whitepaper].

The key point about FRI folding is that it can be checked *locally*. For a single query, the Prover provides 2 evaluations from f_0, f_1, and f_2, and a single evaluation from f_3.

In particular, if the Verifier requests a query for *g,* the Prover sends evaluations for: f_0(±g), f_1(±g^2), f_2(±g^4), f_3(g^8)

The Verifier can check the evaluations are consistent from round-to-round. For example, f_1(g^2) can be expressed in terms of f_0(g), f_0(-g), and the randomness for that round.

For details, we refer readers to the markdown version of this explainer: https://www.risczero.com/docs/explainers/proof-system/stark-by-hand

	f_0	f_1	f_2	f_3
Evaluation Domain (Powers of 28)	**19 + 56x + 34x^2 + 48x^3 + 43x^4 + 37x^5 + 10x^6 + 0x^7**	**12 + 28x + 2x^2 + 10x^3**	**35 + 31x**	**79**
1	53	52	66	79
28	69			
8	63	52		
30	30			
64	46	20	79	
46	13			
27	60	12		
77	50			
22	38	18	38	79
34	3			
79	95	36		
78	23			
50	75	68	33	
42	39			
12	62	68		
45	19			
96	62	73	4	79
69	58			
89	41	34		
67	67			
33	89	92	88	
51	41			
70	50	18		
20	24			
75	95	2	32	79
63	90			
18	72	23		
19	20			
47	82	62	37	
55	33			
85	0	47		
52	16			