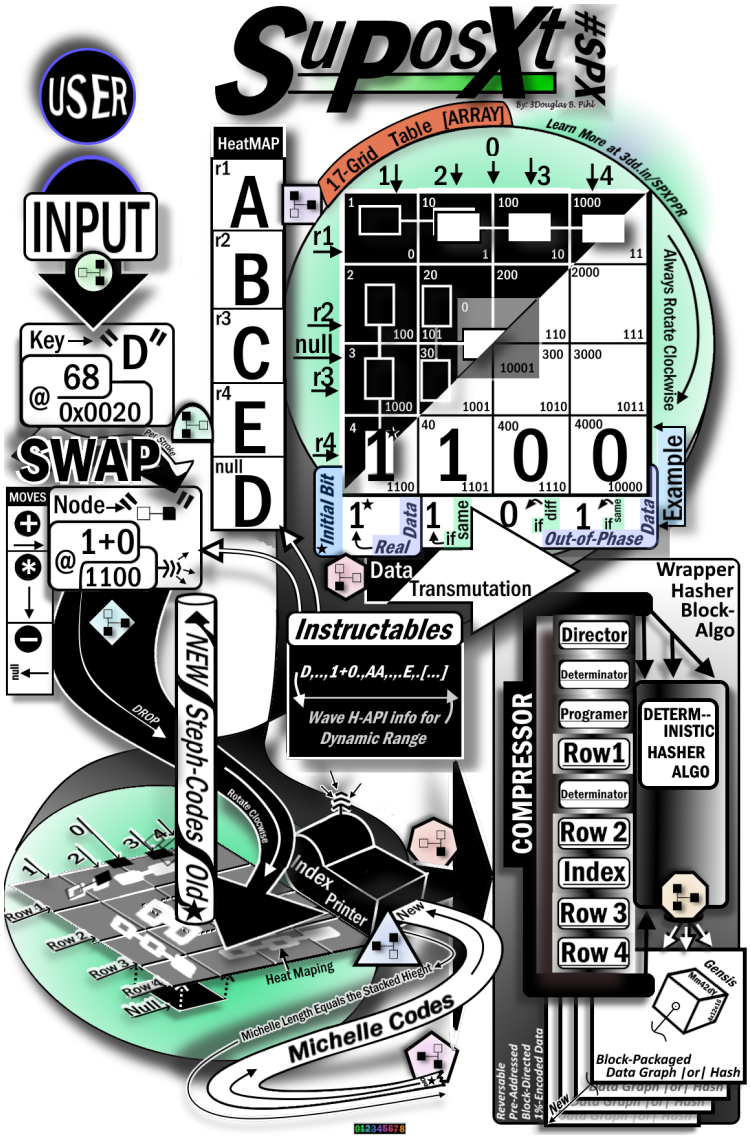
# Super Positioned Text (SuPosXt)

## Quantum Manipulated Dynamic Deterministic Compression to Superposition Text with Verifying Data




*The SPX & QEC compression concept originated from an idea to superposition data based on how the Tanner family communicates to one-another in the hit 1990's sitcom, "**Full House**"*

Can we deterministically but dynamically compress data using quantum mechanics and faux-binary strings that could be reversible while hiding APIs or verifying data? Compressing data dynamically may not be challenging but ensuring there is a way back to where we started may not be so easy without a key. There are keys all around us as relationships between naturally ordered phenomenons. Using the natural length of a language's alphabet, for example, gives us little information but in context gives us the 1st of many natural keys. If we stay in reflection to the alphabet's length we can generate new arrays based on a small data set alongside some rules to ensure we all have order in the chaos. Using the 13 smallest patterns ignoring all single-repeating patterns over lengths of 2, we can then mine up to 32 unique determinable patterns. As long as the order of the 13 base patterns are the same we should come up to the same 32 unique patterns because of the rules they follow for generation. This process can be used for any base reference array and 13 base patterns. Changing from the normalization can be embedded as we token-swap data out during compression, allowing it to be found later as long as our end output showcases at least 3 tokens from the base reference array, that should be enough information to convey what alphabet is used. Being able to quantum-manipulate data to compress purposely inflated data even when looking at just the binary of the text that was compressed to the end result should be plausible.Using quantum mechanics inline with binary or trinary data can allow for dynamic compression and data mixing without knowing any information other than what's in the string along with standardized procedures. Procedures involved can be used in a combination of multiple processes, all of which can work stand-alone, but in conjunction become an environment to perform quantum manipulations. Quantum-manipulated data can be smart data or self-directing data, which is data that contains the way it is to be used by being able to wrap extra data with the base data, like APIs.

Using quantum mechanics inline with binary or trinary data can allow for dynamic compression and data mixing without knowing any information other than what's in the string along with standardized procedures. Procedures involved can be used in a combination of multiple processes, all of which can work stand-alone, but in conjunction become an environment to perform quantum manipulations. Quantum-manipulated data can be smart data or self-directing data, which is data that contains the way it is to be used by being able to wrap extra data with the base data, like APIs.

Because everything is deterministic, we'll use mathematics & old-fashioned work to add entropy, which will inflate our output but will also help reduce the ability to crack or break a Super Positioned Text SuPosXt [***sue – pose – it]*** or *SPX*) hash while giving the keys to check for accuracy during detangling. We will first look at data mining so we can build

patterns for token swapping later. Then we will prepare the data for graphing based on complexities (see ***Figure B3*** in ***Appendix B*** for a visual explanation). Shift the data to a non-future state before performing checks to prepare for our first set of token swaps from the built patterns to remove added data & graphing boundaries. Finally we'll perform quantum entanglements to finally compress the graph into a type of hash.

If all goes to plan, we'll be able to perform various manipulations to the data that should be reproducible, similar to how quantum states are reproducible, making the resulting quantum manipulated data technically reversible.

## *Materials & Methods Used*

I used Notepad++ (text IDE) [1], Opera (Web Browser) [2], JavaScript (no dependencies, AKA vanilla JavaScript) [3], a custom font (Node) [4], basic arrays and loop functions as materials but the methods should be usable across any coding format. Arrays are used as a way to group the data. Node the End-2-End Encryption Font (file name: DNDregular.otf) [4] is used as a system to form the faux-binary strings.

Methods used are as follows:

1. Any reference array without a first-appearance token (IE: first letter of the alphabet is a first-appearance token) will need a "0" in the Zeros place of the array and a "1" in the Ones place of the array, all other arrays (except for mixing arrays) will have the "0" in the Zeros place. This is the 1/0 Rule;

2. All reference arrays, mixing arrays & identifiers combined must reflect the length of the initial reference array. Full Reference Array [sum of characters] = SUM(all_non_full_reference_arrays + mixing arrays + identifiers); (Equation 0)

3. Find Palindromes before Reversed (non-constructive manipulation of data);

4. Find Palindromes PLUS it's Reversed before doubling the Palindrome PLUS it's Reversed (constructive manipulation of data);

5. Using graph weight to create an index (going down adds based on row position; see Figure B2 in Appendix B for a visual explanation);

6. Mixing Arrays have no need for the 1/0 Rule;

7. Mixing Arrays are the only arrays that uses the 1/0 slots and are the only ones capable of using the 1st reference array token which would be a null-pointer;

8. We are utilizing sequenced out-of-place data to showcase a token swap;

9. We are using the "Full House Talk Model [5]" that is based on how the Tanner family members spoke to each other & their closest friends in the hit 1990s TV sitcom, "Full House" [6]. The Full House Talk Model mixes the input and the entire process can be done live but is best as a series of look-up tables. I used the Full House Talk Model look-up tables.

## *Data-Mining Token Patterns*

Using tokens to represent sections of data in a controlled manner can allow us to take the 13 smallest patterns in binary to create a predetermined sequence of binary patterns. We can use this same technique to obtain predetermined sequences from any form of data structure, for example, binary, trinary, CRC-encodings & secure-hashes. If our 13 smallest patterns are also the most commonly seen patterns for non-binary or non-trinary strings, then we can assimilate this base key from any data type & structure.

We need to build or have ready 3 alphabet reference arrays; they do not have to be alphabet characters, but they do need to be a set limiter of the max allowed length (or upper-limit) to swap for a token. For example, the base SPX system uses the American English alphabet, so the upper-limit is 26 characters to be swapped per token. The first alphabet array will need to be in one case, either capital or lower, and contain the full limit so it can be a "Full Reference Array." The second alphabet array will need to be 13 less than the max but in the opposite case of the first alphabet array, so this array will be the "$2^{cd}$ Reference Array." Then finally, the third alphabet array is only 5 characters long, also in the opposite case of the first alphabet array to be our "$3^{rd}$ Reference Array." Normally we will add "0" & "1" to the far left of all arrays. However, We do not add the "1" in the "ones" position for full reference array & $2^{cd}$ reference array; they will only get "0" added to the furthest left side of the array. Any and all exceptions to the 1/0 rule (placing "0" & "1" values in the "null" & "ones" places of the arrays) will be mentioned as they appear. We do not want to use the first character in the alphabet reference array, for that is reserved for single-character token swaps (see ***Figure 1*** for a visual explanation). Because the third alphabet reference array has no "a" or first alphabet character in it's array, it gets "0" in the "null" position & "1" in the "ones" position (full 1/0). Below are the example reference arrays:

| Array Title | Reference Arrays |
|---|---|
| Full Reference Array | ["0", "A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"] □ |
| $2^{cd}$ Reference Array | ["0", "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n"] □ |

| | |
|---|---|
| 3rd Reference Array | ["0", "1", "o", "p", "q", "r", "s"] □ |

**Example 1.** □

The remaining alphabet characters missing in the opposite case of the first alphabet array are used as token type identifiers. From our example, the token type identifiers are “t”, “u”, “v”, “w”, “x”, “y”, “z”. We’ll assign what these identify later. The alphabet reference arrays and type identifiers that share the same reference don’t have to be an alphabet, but these arrays and identifiers can be anything, including any pattern(s) that may be seen in the array. The important part is the length of the first array is the full length of the last 2 arrays & their token type identifiers, while everything in all the arrays and identifiers should be equivalent but different in case. We are using lower case and upper case as natural states for the used language and any character that only is in one of these cases should not be used because they cannot appear in the full reference array, because we need two states minimum but the system can support three or more states as long as separately all states equally reflect the full reference array. We can explain the case relationship by formula, where “Arr#” is the numbered array and [IDs] are the group of identifiers. These formulas are seen after the following figure:

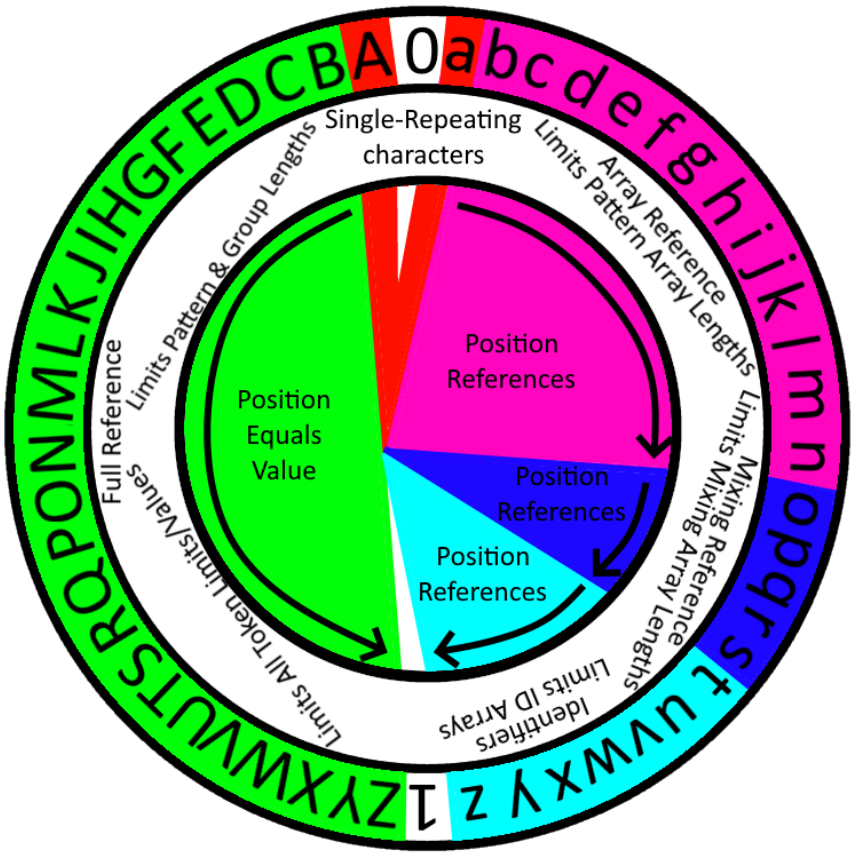



**Figure 1.** Array Reflection Wheel. This showcases the relationship each array has to the full reference array. (Green) The Green side is the full reference array, starting at the null-position or “Zero” ending at the “One”, this section provides the total length allowed while each position in the array is represented by the object in that

position, in this case a character seen in the left side of the Array Reflection Wheel. (Red) The Red sectional bits are the first alphabet character in both lower & upper cases, these are used for single repeating characters. (Pink/Purple) The Pink/Purple section is the 2$^{cd}$ reference array and is used to limit the pattern array lengths. (Blue) The Blue section is the 3$^{rd}$ reference array which is used for limiting the mixing array lengths. (Teal) The Teal section are the identifiers, and this section ends at 1 with the Green side showcasing that the reference is finished.

Arr1 = Arr2 + Arr3 + [IDs], (Equation 1) is a non-specific example of the array relationships with each of the arrays being the following: Full (Arr1), 2cd (Arr2), 3rd (Arr3) reference arrays & the identifiers (IDs). This is the general formula that figure 1 showcases. Arr0 = [a→z]26, (Equation 2) is the American English alphabet opposite full reference array example, displayed as the null-array. [A->Z]26 = [a->n]14 + [o->s]5 + [t→z]7, (Equation 3) is the American English alphabet array example showcasing the full reference array (Arr1), the 2cd reference array (Arr2), the 3rd reference array (Arr3), and the identifiers (IDs). Each array shows what characters are started with and ends with each array by placing them in brackets separated by an arrow or double greater-than sign as well as how many objects are in each array at the right side of the brackets.

A good example of a non-alphabetic full reference array is using primes but raising a single prime to the next prime value. If you need to get an opposite case of this, you simply raise all of these by the prime before the initial prime in the first slot. Only to show by example primes used as a full reference array is as follows:

- Arr1 = [P7→P89]20
- Arr0 = [P7^5→P89^5]20
- [P7→P89]20 = [Arr1^7]7 + [Arr1^3]5 + [Arr1^2]8

This notation uses “P” for prime, points to which prime it begins at, which it ends at, and how many primes are in the array. The opposite case of the 1st reference array is “Arr0” in notation form, which is the entire first array raised by the previous prime seen in the first slot. So Arr0 is Arr1 raised by 5. We can use the notation seen by Equation 3 as a simple way to describe any changed or modified reference arrays. Any notation used to identify a change can be placed within the programming slots for the decoder to know of any special changes if the decoder doesn’t allow for direct influence uploading of one’s patterns and array formats. This part on primes as objects used for the reference arrays was used to showcase how the object in the array is less meaningful then the way the arrays are related to one another. Each is unique to the other even if they are alike or seemingly similar.

Now that we have a limit set with the 1st reference array, we can build the patterns. Looking at the smallest patterns, we need to remove any single-length patterns, for they would inflate anything we apply them to. Starting at the length of 2 or L2, we will see “00”, “11”, “01”, “10”, “100”, “011”, “101”, “010”, “1001”, “0110”, “10100”, “01011”, “001101”. We purposely do not use any repeating single-character patterns over L2 (also written as “L3+”) for we will use a specific pattern replacement procedure for those. The procedure we are going to use for making the larger sequences will be a similar method for single-character replacement tokens, just with more steps, which will be covered later.

If you are using an alphabet reference or any full reference over 26 (not counting the null position), then there is room to add to any set of arrays. Just ensure any new array added gets an identifier added as well. We are using the relation of identifiers to full reference for determining what the starting bits may be. For example: If you needed to change the starting bits, you would place that information within the programming slots (as mentioned earlier for reference changes). You would have to use notations or Wave-Data APIs [7] to inform the decoder of the patterns if the decoder doesn’t allow for direct uploading of your patterns and forms.

When building patterns, we never want to accept a new pattern that is a duplicate of a previous pattern in any array. We will generate patterns and then check if we have those patterns already; if not, we’ll use them, and if so, we won’t. We will use our 13 patterns from smallest to largest so we have the best chances for the smallest considerations of patterns possible because we are only going to build up to 8 patterns per array. We will never use the null position of any of these arrays; most of these will not use the ones position either, and we will not go past 9 in position. This prevents double-digit overflow in the later processes as well as prevents duplicate identifiers. Each pattern array has a rule for its generated patterns, which are the following:

10. Pattern Palindromes [1st Generated-Pattern Array]

For Example: “010010” is a pattern palindrome, specifically a palindrome with the reference pattern “010” included. “010” is the 7th pattern in our pattern array.

11. Pattern Reversed [2cd Generated-Pattern Array]

For Example: “00101” is a pattern reversed, specifically a reversal of “10100” the 10th pattern in our pattern array.

12. Pattern Palindromes PLUS it’s Reversed [3rd Generated-Pattern Array]

For Example: “011110011110” is a pattern palindrome plus it’s reversed self, specifically a reversal of the palindrome “011110”, which is “011” & “110” added together which that sum is from our 1st Generated-Pattern Palindrome array.

13. Pattern Reversed-Doubled PLUS it’s Palindrome-Reversed [4th Generated-Pattern Array]

For Example: “010010010010” is a pattern reversed-doubled plus it’s palindrome self, specifically “010” was used to make first “010010” then it’s palindrome “010010” to be added in generated order for “010010010010”.

*The used patterns for the proof will come from this example data. This is also the same referenced data from the previous list (list numbers: 10-13).*

| **Array Pattern Starter** | |
|---|---|
| 13 Smallest Binary Patterns (used for bottom arrays) | “00”, “11”, “01”, “10”, “100”, “011”, “101”, “010”, “1001”, “0110”, “10100”, “01011”, “001101” □ |

| **Array Title** | **Reference Arrays** |
|---|---|
| 1st Generated-Pattern Array | [“0”, “1”, “001101101100”, “1010000101”, “0101111010”, “10011001”, “01100110”, “101101”, “011110”, “010010”] □ |
| 2cd Generated-Pattern Array | [“0”, “1”, “110001101001101110”, “0010101101011001”, “101100101100”, “1101011010”, “0010100101”, “101100”, “11010”, “00101”] □ |
| 3rd Generated-Pattern Array | [“0”, “1”, “100110011001100110011001”, “011001100110011001100110”, “001101101100001101101100”, “101000010110100000101”, “010111101001011110 10”, “101101101101101101”, “010010010010010010”, “011110011110”] □ |
| 4th Generated-Pattern Array | [“0”, “1”, “101100101100001101001101”, “110101101001011010 11”, “001010010110100101 00”, “1001100110011001”, “0110011001100110”, “110110011011”, “101101101101”, “010010010010”] □ |

**Example 2.** □

***Theorem 1.*** *Utilizing only the number of character available in the American English Alphabet mathematically adjusted and the 13 smallest binary patterns, a series of unique patterns can be generated and formatted in such a way that a computer system could later find the same patterns with the same arguments.*

***Proof of Theorem 1. Example 1 & Example 2*** *both showcase the determination of difference with the characters of the American English Alphabet for creating separate arrays as well as showcasing the building of the patterns. The pattern arguments were explained in the list before* ***Example 2****. The*

*arguments are the rules to build.* ***Figure 1*** *does showcase the relationship of how a single argument, "use the American English Alphabet" can be used to form multiple arrays to separate data we mined in* ***Example 2.***

We have taken the single array of 13 patterns and generated up to 32 unique patterns. Now we order the generated patterns in each array from largest (furthest left side) to smallest (furthest right side), within the array, before we add "0" & "1" to the furthest left side of each array (using up the null & ones position of each array). The arrays seen in Example 2 were already set to this formation. We want to always replace the largest possible pattern first, then work our way down to the smallest patterns last while also not using "0" & "1" both in position & in value, aka ignoring the 1/0.

So far, we have forced generated a number of unique strings that should always end up the same for as long as the initial 13 key patterns are in the same order each time. For as long as the generated patterns are always largest to smallest, we will always get the largest possible compression per data-set in our soon-to-be hash. Next, we will need to generate mixing tokens to hide the differences between data sets.

The next set of arrays is going to be specific to your data-formation but the examples given will still apply to the same examples as before. We have an index which could also be an API string. However, you need to set up 3 arrays each with at least 5 tokens inside. All three of these should always have the same number of tokens inside each plus 1/0. These tokens need to be your index keys or API key-bits, even if you have to split them up some. For example, here's our 3 index key arrays, which are as follows:

14. ["0", "1", "0.001", "-500.1", "-505.11", "-330.1", "220.1"]
15. ["0", "1", "770", "50.1", "55.01", "60.01", "0.3"]
16. ["0", "1", "0.2", "0.4", "0.5", "1100", "111"]

The index key arrays start off with the most complex (containing the most number of unique non-alphanumeric characters) at longer lengths, then work down to the least complex at the smaller lengths, & finally ending with no-complex tokens (alphanumeric only). The index key arrays of non-1/0 tokens can only be as long as the number of non-1/0 tokens present in the 3rd reference array.

We need 2 more arrays, one to specifically handle our data-set breaks & another to handle unique edge cases. Again, which tokens/strings you need in your case may change, but here are the base examples in line with the previous examples so far:

17. ["===;", "==;1;0;", "==;0;1;", "==;2;", "=;1;0;", "=;0;1;", "=;2;", ";2;", ";1;", ";0;"]
18. ["==", "=", ";1", ";0", "1;", "0;", ";.", ";", ".", "-"]

The data-break array attempts to remove all data breaks from most generic-complex (most number of non-alphanumeric characters at the longest lengths) to the least generic-complex (least number of non-alphanumeric characters at the shorter lengths), always putting non-alphanumeric tokens before alphanumeric containing tokens. The unique edge case array is also setup in the same manner as the data-break array. These arrays are similar to the reference arrays where we may use the “1” in position & in value, but these ones also may use “0” in reference but not in value to ensure we can use all the slots of the array. This will give us a unique token, “null-soft”. Null-soft a null-pointer token, meaning we will see a “0” before the token identifier. These are the only tokens possible to have a null-soft token.

Now we know what we are wanting to replace or use as token bases, we may need to search for first duplicates & sequences of these patterns in specific ways then work our way down to just these arrays to always get the largest possible compression. For the sequences we follow the same rules as we did for mining the patterns.

## *Binary VS Faux-Binary States*

The majority of our data will need to be either in a binary or faux-binary state. Trinary does count as a binary state, but your system will need to be handled more specifically if trinary is used; otherwise, this will go over using a binary state or a faux-binary state for your data.

The advantage of simply using the binary state of your data is that it is going to have the smallest possible compression, and oftentimes you won’t need an index at all (if all binary data-set lengths are equal, sections of 8 or sections of 4 for example). The index data-set can be, like all parts of the datasets, removed or swapped out for similar but different data-sets. The tokens used for compressing that data-set may need to be adjusted, or the data in the data-set will need to be modified to work as an index data-set. The advantage of using a faux-binary state is simply privacy. A faux-binary state may reduce overall binary characters seen, but the true point of faux-binary is privacy in what you send, because faux-binary states don’t have to contain the actual data but enough data to rebuild to the original state, so we can send small portions of rebuildable data.

A faux-binary state is taking your data and modifying it to appear like a binary state. The End-to-End Encryption font, Node, is designed to take ASCII characters and turn them into a physically encrypted faux-binary output. Based on the complexity states of simple dot-node connection configurations, Node takes a character and generates a formula with binary pointers to then create a heat map to direct how to align this representative shape of the node-connection configuration to a 17-box grid (see Figure B3 in Appendix B for a visual explanation). Each heat map comes with an index value to allow for decoding through reconstruction of the graph at a later time, we use a configuration look up table for seeing what the predetermined Node heat mapping is. Next is the example Node configuration look up table.

**Table 1.**

| Node Configuration Look Up Table | | |
|---|---|---|
| **Character** | **Heat Mapping** | **Index** |
| SPACE | D0 | 0 |
| A | A0A0D1 | 0.001 |
| B | A0A1 | 1100 |
| C | D1D1 | 0.1 |
| D | A1A0 | 1100 |
| E | A0A0 | 1100 |
| F | A1A1 | 1100 |
| G | D0D1 | 0.3 |
| H | D1D0 | 0.2 |
| I | A1B0 | 70 |
| J | A0B1 | 70 |
| K | A1B1 | 70 |
| L | A0B0 | 70 |
| M | A1B0A1B0 | 770 |
| N | A0B1A0B1 | 770 |
| O | A1B1A1B1 | 770 |
| P | A1B1A1B0 | 770 |
| Q | A0B0A0B0 | 770 |
| R | A1B10B1 | 770 |
| S | A1B0A0B1 | 770 |
| T | A0B1A1B0 | 770 |
| U | A1B0A1B1 | 770 |
| V | A0B1A1B1 | 770 |
| W | A0B0D1 | 50.1 |
| X | A0B0D0 | 50.1 |

| | | |
|---|---|---|
| Y | D1A0B0 | -500.1 |
| Z | D0A0B0 | -500.1 |
| a | A0A1A0 | 111 |
| b | A1A1A0 | 111 |
| c | A1A0A1 | 111 |
| d | A0A0A1 | 111 |
| e | A1A1A1 | 111 |
| f | A0A0A0 | 111 |
| g | A1A0A0 | 111 |
| h | A0A0A1 | 111 |
| i | A1B0C1 | 6 |
| j | A0B0C1 | 6 |
| k | A1B1C1 | 6 |
| l | A1B0C1 | 6 |
| m | A1B0D1A1B0 | 55.01 |
| n | A0B1D1A0B1 | 55.01 |
| o | A1B1D0A1B1 | 55.01 |
| p | A1B1D0A1B0 | 55.01 |
| q | A0B0D0A0B0 | 55.01 |
| r | A1B1D0A0B1 | 55.01 |
| s | A1B0D1A0B1 | 55.01 |
| t | A0B1D1A1B0 | 55.01 |
| u | A1B0D1A1B1 | 55.01 |
| v | A0B1D1A1B1 | 55.01 |
| w | A0B0D1A0B0 | 55.01 |
| x | A0B1D0A1B0 | 55.01 |
| y | A0B1D1A0B0 | 55.01 |
| z | A1B0D0A0B1 | 55.01 |
| 0 | D0A1B1 | -500.1 |
| 1 | A1B1D0 | 50.1 |
| 2 | A0B1D1 | 50.1 |
| 3 | D1A1B0 | -500.1 |
| 4 | D0A1B1 | 50.1 |
| 5 | D1A0B1 | -500.1 |
| 6 | A0B1D0 | 50.1 |
| 7 | D0A0B1 | -500.1 |
| 8 | A1B1D1 | 50.1 |
| 9 | D1A1B1 | -500.1 |
| ! | A0A0D1 | 220.1 |
| ? | D1A0A0 | -330.1 |
| “ | A1A1D0 | 220.1 |
| ‘ | A1A0D0 | 220.1 |

| | | |
|---|---|---|
| < | D0A1A1 | -330.1 |
| > | D0A0A1 | -330.1 |
| ^ | D1A1A1 | -330.1 |
| _ | D0A1A1 | -330.1 |
| [ | D1A0A1 | -330.1 |
| ] | D1A1A0 | -330.1 |
| # | A1A1D1 | 220.1 |
| & | A0A0D0 | 220.1 |
| * | A0A1D1 | 220.1 |
| ( | A0A1D1 | 220.1 |
| ) | A1A0D1 | 220.1 |
| $ | A1B0C1 | 6 |
| % | A0B1C0 | 6 |
| : | A0B1C1 | 6 |
| ; | A1B1C0 | 6 |
| @ | D0D1 | 0.4 |
| \| | D1D0 | 0.5 |
| { | A1B1D1A0B0 | 55.01 |
| } | A0B0D1A1B1 | 55.01 |
| / | A1B0D0A1B1 | 55.01 |
| \ | A0B1D0A1B1 | 55.01 |
| . | A1B1D1A1B0 | 55.01 |
| ` | A1B1D1A0B1 | 55.01 |
| + | A1BD1A0B0 | 55.01 |
| , | A0B0D1A1B0 | 55.01 |
| - | A0B0D1A0B0 | 55.01 |
| = | A1B1D1A1B1 | 55.01 |
| ~ | A0B0D0A1B0 | 55.01 |

In each row you will see a character, centered is its heat map and to the right is the index code. Within the heat map you will see up to 4 characters “A”, “B”, “C”, “D”; these are the rows of the grid, while the numbers are the node on/off positions. The exception to this is D; D actually refers to the aligning box (the 17th box in the grid) or under-table row. This is done specifically to confuse others. The entire system is based on mapping these “nodes” to a grid, but in that mapping you never need to know the last row, as long as you have the rest of the information. So we take advantage of this situation and purposely ignore and never use the real 4th row in the grid in reference. Node is designed around this concept so anytime a Node heat map says, “D”, it will refer to the 17th grid box. When we rebuild the pattern based on that grid, it will still work for anything going to the fourth row of the grid; it must either pass through the center or start/end on a different row. Even if a 3 or 2 straight was on

the 4th row on its own, we would see its index code and would be able to work backwards what is missing using the checks to find the blank space that is missing data. We may also have data residue that should be some of the missing bits from the mapping.

If you only use binary data, then you will need to divide the binary string into groups of 4 or groups with lengths of 4 (aka L4). In each L4 group: the 1st position is data-rail A, 2cd position is data-rail B, 3rd position is data-rail C, 4th position is data-rail D. The index is simply the length of that binary data set. For example, if your binary set is a full length of 8 (L8), its index would be "0.8". However, if your binary set is a full length of 16 (L16), its index would be "1.6", so you could only have a max limit of L99, which would be an index of "9.9". Just like the Node Configuration table, each input should be handled separately, but to save space in the index field, this extended binary consideration of indexing could be used.

## *Dimensional Shifting*

Once our data is set up in the graph where the raw data bits are in 4 or fewer rows, we can begin shifting them. After we shift the bits, we'll run everything through a specially modified formula to obtain a parity check output or action bit. This action bit can be compounded and used to help ensure no data was changed between sends.

To shift the data we have, we first look at the first bit in a row and record that bit. Then we compare this (previous) bit to the next bit; if they are the same, we'll record a "1," but if they are different, we will record a "0." If you are using a trinary system, you will use a "2" if the previous bit is more than the next bit, while a "0" would represent the previous bit being less than the next bit & "1" would mean both bits are equal. We will then compare the new previous bit to the new next bit and continue until the entire row has been compared and recorded. Do this for all binary-like state rows you have.

Shifting binary-like data allows you to only have 1 bit per row of real data being transferred. We're never actually sending the data but an "overflowed vector" of the data, while in this case we are not overflowing a buffer but recording the difference between two bits as overflowed data. To reverse this shift, we start at the first bit, record it, then compare the now (previous) bit to the next bit and work accordingly in the manner as before. There are no identifiers for what form of state is used, so a quick check down the string of each row should be down for "2" and if none is found, a binary or a binary-like state should be assumed.

We will take the end result of the data shifting and toss each bit through a Modified Futurama-Theorem (MFT) formula, adding the previous output of the MFT (if there was one) beforehand and recording the last bit of output from the MFT. This last output bit should be our action bit. We can perform a traditional hamming code parity check at this point if we want to verify ourselves. As of writing this, the current SPX demo does use traditional hamming code parity check to verify itself alongside the MFT parity check. All parity checks are recorded in some form in their designated data-sets (see Equation 19 for more information on the designated data-sets).

The Modified Futurama-Theorem is Ceiling( Trunc(input_SUM * 3.14) + Absolute(((current_position * 18) + input_length) - 1) + MOD) % 2 □ (Equation 4). This is the JavaScript implication of the MFT. The MFT uses the input sum, (ABSOLUTE((full_action_counter * max_loop_length) + input_length) – 1) (Equation 5) multiplied by PI (3.14) to get a radius based on that value to then keep it at it's own length and keep the output at mod2 or nothing higher than the digit "1". This will always shift the data into a form equal to its length, and in this case we are only dealing with a length of 2, and our end-modified allowed output is also capped at 2. This keeps all of our data in that binary formation. If we used trinary, we would need to change the end modifier number, mod2, to a "3" for mod3.

The MFT can also be adjusted, so for example, when we look to see if the total of 1's seen in the previous parity bits as a whole is even and if they are, then we use a "+2" adjuster instead of "+ MOD", otherwise we use a "+1" adjuster instead of "+ MOD" and this will force the output to always be properly formed and without time-based adjustments. To make time-based adjustments, change the "+ MOD" to a value that is exponentially higher than the "+ MOD" adjuster. If we want to go 1-step in the "future", we will use "MOD^2". The 2-steps would be "MOD^3". Where "MOD^1" is the assumed natural starter position for the "+ MOD" adjuster. Please note that time-adjusting to the future/past is mathematical guessing based on the formula presented, so the real-future or past parity output may be different, but this could be used to guess a possible parity without the extra data involved.

***Theorem 2.*** *Taking the absolute sum of the byproduct of multiplying the total actions taken by the max loop length then add the actual input length then subtracting 1 by Pi [3.14] before adding a modular change based on the time value wanting be presented to then find the modulus of the end data structure we need will allow a basic parity check to see if its equal to its modulus or not. The same data results in the same output but should change if it's MFT output is paired with the data by 1 or, if possible by modulus, doubled.*

***Proof of Theorem 2. Equation 4*** *is the Modified Futurama-Theorem that is being used to find the following concept. Writing through zero to one over and over until we have the equivalent length to what was inputted plus one on a paper of a single row. Staple the first digit on the end of the rod, then wrap the remaining paper around the rod so that only one digit side by side is on any one of the two sides of this rod. Looking at the longest side add the number of ones along the row and the stapled-starting digit (if the stapled-starting digit is a one) and if the total sum of seen ones is even, give a zero output; if odd, give an output of anything other than zero. This works on generic loop pairing, which is how the Futurama-Theorem is able to determine the number of loops to get two pairs back where they started under specific conditions, the MFT does this by using the type of required output (base of digit type allowed) as the loop identifier and we only are getting a result that shows the difference. Having the modulus set to two allows us to show "even" or "odd", without the value of the data used getting in the way.*

## *Spherical-Parallelogram Check*

Why trust multiple parity checks when you can verify a parity check. One of the key bits of information we want to superposition in one of our data-sets is a "Magic Number", which is a derivative of amalgamated data based on wrapping a parallelogram around a sphere where the parallelogram doesn't overlap but does touch when wrapped around the sphere without gap between both objects. We build a parallelogram and sphere based on the length of our input, turning this into a form of check based on what we should obtain after the detangling process. But we also base parts of the parallelogram's & sphere's data points off the number of internal loops that need to be made for the completion of rebuilding the graph as well as the input sum. In similar to how we did to modify the Futurama-Theorem to get the MFT. For example, here are the formulas used, we will only be using JavaScript Math notations, [keyword clarification] & connected_plain-text_explanations.

These define starting variables:

z = Input_Sum = (ABSOLUTE((full_action_counter * max_loop) + Input_length) – 1) (Equation 6)

y = Current_Micro-Loop_Position (Equation 7)

x = Input_Length (or Number_of_Seen_Inputs) (Equation 8)

[ANGLE] a = (((y * 18) + x) -1) (Equation 9) *18 is our loop limit.*

These find the parts of the parallelogram:

A | C = Round((trunc[z * PI]) + ((a) % 2)^((PI + z) / (a))) (Equation 10)

B | D = Round((180-(trunc[z & PI]) + ((a) % 2)^((PI + z) / (a))) (Equation 11)

[ANGLE] b = Round(a / sin([round-up{trunc[z * PI]} + ((a) % 2)^((PI + z) / (a))])) (Equation 12)

[HEIGHT] h = Round(a * (a / sin([round-up]{trunc[z * PI] + ((a % 2)^((PI + z) / a))*sin([round-up]{(trunc[z * PI] + a % 2)^((PI + z) / a)})}))) (Equation 13) *If output is negative, drop the negative sign for h.*

P = Round((a * 2) + (a / sin([round-up]{trunc[z * PI] + ((a % 2)^((PI + z) / (a * 2)))}))) (Equation 14)

AREA = Round((1/2)*(a /(sin([round-up]{trunc[z * PI] + ((a % 2)^((PI + z) / ((a * (a * (a /sin([round-up]{trunc[z * PI] + ((a % 2)^((PI + z) / ((a * (sin([round-up]{a}])))))})))))))))))) (Equation 15)

These find the parts of the sphere:

[SPHERE] VOLUME = Round((4/3) * Pi * POW(trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / ( a * (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / a)})))))) *{drop any negative sign at this point}* / 2, 3)) (Equation 16)

[SPHERE] RADIUS = Round(a * (a / (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / (a * (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / (a))}))))))}))))$^{\{drop\ any\ negative\ sign\ at\ this\ point\}}$ / 2
(Equation 17)

This defines the "Magic Number":

MAGIC NUMBER = Round(Round(Round(((4/3) * Pi) * POW(Round(a * (a / sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / (a * (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / (a))}))))))}))){drop any negative sign at this point} / 2, 3)) % ((1/2) * (a / (sin[round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / (a * (a * a)) / (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / a))}))))}))))) / Round(a * (a / (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi +z) / (a * (sin([round-up]{trunc[z * Pi] + ((a % 2) ^ ((Pi + z) / a))}))))))}))){drop any negative sign at this point} / 2)) □
(Equation 18)

***Theorem 3.*** *By having the parallelogram and sphere with values based on the input data, a 3-Dimensional mapping would show the parallelogram wrapped around the sphere to touch itself but not overlap itself without gap between the parallelogram and sphere. Finding the Magic Number during this alignment from the data used to map the objects gives a unique output based on the data.*

***Proof of Theorem 3. Equations 6-17*** *show how to use the data given to create the different values needed including to define all the parts of the parallelogram & sphere, even though not all parts are later needed. Equation 18 defines the Magic Number and because at specific points any negative sign is dropped and the value remained is used, it doesn't matter in which way the maths takes us (positive or negative) because both will be used as a single output (positive sign). This allows the natural relationship between positive and negative numbers to have impact in parts of the equation we need that cancellation but after defining subsections of the equation, we just want the end value for the next steps. The dual used relationships of the positive and negative signs allows for a unique output even if two same lengths inputs are used without having the same data.*

## *Pattern-Based Token Swaps*

We already built the patterns for swapping; now to go over how the entanglement tokens work. These will be the base steps to create all the Pattern-Based tokens for swapping. We will also go over the specific methods to look for sequences of patterns to swap for as well.

The tokens we are going to use will be NUMBER (+) IDENTIFIER (+) REPEAT_ID. This is the generic setup for all pattern-based tokens because it will entangle a small amount of information into the token. For example, the identifier tells us which unique pattern array we are using, and what's in that array can be anything. The same goes for number & repeat ID; anything can be used to represent these values. Starting off by simply looking for the predetermined pregenerated tokens (the first 4 generated arrays): The number we are looking for is the position of the token in the array; the identifier is one of the remaining lower-case characters (t-z); the repeat ID is the number of times we saw this pattern repeated in a row as equal to it's position in the full reference array.

The example identifiers are as follows:

| Identifier | Identified |
|---|---|
| w | 1st Generated-Pattern Array |
| x | 2cd Generated-Pattern Array |
| y | 3rd Generated-Pattern Array |
| z | 4th Generated-Pattern Array |
| t | 1st Index Keys Array |
| u | 2cd Index Keys Array |
| v | 3rd Index Keys Array |
| a | Data-Break Array |
| A | Unique Edge Cases Array |

**Example 3.** □

The way we chose these identifiers is the last 3 reference tokens in the full reference array, but in the opposite case, is always for the first 3 generated-pattern arrays, while the first 3 available tokens after the 3rd reference array are always used for the index key arrays, and the first token of reference for both cases is always double used, once for the data break and unique edge cases but also for single token replacements (although single token replacements only use the case used in the full reference array).

The generated-pattern tokens are pretty straightforward; the number is the array position for the token to put back, the letter immediately after the number is the identifier, and if there is an opposite case letter after the identifier, then that is the number of times to loop the token for a full replacement. Determining the repeating single-character tokens is again fairly straightforward. We will only look for repeating "0" and "1". So we will place a single bit of the repeating single-character bits that is being swapped for plus the capital-case

character from the full reference array, which represents the number of times the single bit was seen in a row. So if “000000” was replaced with “0F,” we’d replace the “F” for “00000” to make “000000” during reversal. We will only replace L3+ repeating single-character strings (or only strings with lengths of 3 or more), because “00” and “0B” are the same length, while “0” turned to “0A” only inflates, so that’s pointless extra work.

The Index Keys tokens are less straightforward. Whereas the generated-pattern tokens are simply what’s already been swapped, these index key tokens are first mixed with combinations of one another & then a simple token swap as before.

The goal is again to find the largest possible swaps first. At some point we need a way to determine edge cases, similar to before. So, we will introduce single tokens to be used as modifiers for these last token sets  This “7th array” is composed of the most commonly seen single-character tokens in the index data set that isn’t the 1/0 set. Your 7th array may differ, but you will want to make sure no alphanumerical character is aligned by value to its position, so if your 7th array token is “3”, make sure it’s not in the 3rd position of the array.

Now we will list the complexities used to create the patterns in order to do them in during this phase of token swapping. And they are as follows (MIXING_ARRAY_REFERENCE (+) IDENTIFIER(S) (+) REPEAT_ID):

19. LOWtuvCAP: the initial lower-case character position equals the position for the token from the “t”, “u”, “v” arrays to swap back in the order shown (t-u-v) and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).

20. LOW7tCAP: the initial lower-case character position equals the position for the token from the next two arrays; the “7th array”, [“0”, “1”, “7”, “6”, “5”, “3”, “2”], plus the 1st Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).

21. LOW7uCAP: the initial lower-case character position equals the position for the token from the next two arrays; the “7th array”, [“0”, “1”, “7”, “6”, “5”, “3”, “2”], plus the 2cd Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).

22. LOW7vCAP: the initial lower-case character position equals the position for the token from the next two arrays; the “7th array”, [“0”, “1”, “7”, “6”, “5”, “3”, “2”], plus the 3rd Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).

23. LOWLOWtCAP: the initial 2 lower-case characters positions equals the positions for the next two tokens, with the 1st lower-case character for the 1st token & the 2cd for the 2cd token for replacement from the 1st Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).

24. LOWLOWuCAP: the initial 2 lower-case characters positions equals the positions for the next two tokens, with the 1st lower-case character for the 1st token & the 2cd for the 2cd token for replacement from the 2cd Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).

25. LOWLOWvCAP: the initial 2 lower-case characters positions equals the positions for the next two tokens, with the 1st lower-case character for the 1st token & the 2cd for the 2cd token for replacement from the 3rd Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).

26. LOWtCAP: the initial lower-case character position equals the position for the token from the next array, the 1st Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).

27. LOWuCAP: the initial lower-case character position equals the position for the token from the next array, the 2cd Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).

28. LOWvCAP: the initial lower-case character position equals the position for the token from the next array, the 3rd Index Keys Array and finally the capital-case character position is equal to the number of times seen (if seen only once, no capital is needed).

The data-break tokens are very straight forward, NUM (+) “a”. The number is the position of the token on the data-break array, while the “a” is the identifier, so we know where to look for the token. We won’t need to look for anything else with data-breaks, but in the last array we’ll need to mix things up a bit more.

The unique edge case tokens are going to use 3, then 2, then 1 sets of tokens in a row. Similar to how we had the LOWtuvCAP, we’ll combine various tokens in the same array, the edge cases array, instead of different arrays (t-u-v). Next is a list of the following complexities to find the edge cases, which is as follows:

29. LOWLOWLOWCAP: Each lower-case character represents the location of the token to be placed here from the array, the Unique Edge Cases Array, while the capital-case character position is equal to the number of times seen (if seen only once, capital is needed).

30. LOWLOWCAP: Each lower-case character represents the location of the token to be placed here from the array, the Unique Edge Cases Array, while the capital-case character position is equal to the number of times seen (if seen only once, capital is needed).

31. LOWACAP: The lower-case character represents the location of the token to be placed here from the next array, the Unique Edge Cases Array; while the capital-case character position is equal to the number of times seen (if seen only once, DO NOT REPLACE).

32. LOW7CAP: The lower-case character represents the location of the token to be placed here from the “7th Array”, [“0”, “1”, “7”, “6”, “5”, “3”, “2”]; while the capital-case character position is equal to the number of times seen (if seen only once, DO NOT REPLACE). This will increase the count by 1 if only two 7th array tokens are in a row, L3+ |or| “No Low7C” for best results

33. NUMA: The number represents the location of the token to be placed here from the Unique Edge Cases Array (NO REPEATS CONSIDERED).

## *Quantum Entangled Compression*

Now that we have much of the unique data entangled, swapped and removed, we should have determinable data with possible duplicate strings made within the full string. Taking advantage of any data set, hash or string that has duplicate-character output, possible could have quantum entangled tokens to help compress the string. Quantum Entangled Compression (QEC) will be very similar in method to the tokens being generated for swapping from earlier but will relay on what's in the string, the length of the string & the length of our full reference array.

Let's start at finding the tokens we can reference for replacement, aka replacement tokens. Finding the replacement tokens is a matter of sectioning the string into smaller parts, even if smaller parts overlap one another, just as long as one character is different from the overlapping parts around it. This is a form of "Proof-of-Work [8]" commonly seen in machine-verifying databases & machine-verifying web-technologies.

We will start by looking at the head or front of the string (furthest left side). We will begin by recording what we see to the limit of the shortest of either the full reference array (minus the "0") or half of the total string length. This may seem like a strange limitation, but if we consider that we are going to reference one part of the string to replace another part, then we only need to look at half the total string length, but if that half of the string length is greater than the limit of our full reference array, then the length of the full reference array is all the length we need. So now we need to record all that we see moving over only 1 position each time, until we see a blank space in our recording string. When we get the point of having a blank space, we can stop & begin limiting how much we see by 1. So, for example, if we were looking at the full reference length of 26 and when we get to the end, so it begins to have empty space, we just return to the start of the full string and look at the 25 length strings, then when seeing blanks reappear we return again to the beginning of the full string and reduce by one again, over and over. So we reduce by one after every pass through the string until we are looking for & through the L4 bits. This will ensure we have all possible tokens viewable, and they go from biggest to smallest (in length reference). We don't look at L3 or less bits,

because that would be extra work and would inflate our end hash instead of compressing. So we can have a single array with all these “tokens” to swap out. Remember not to accept duplicate tokens just as before.

Now, we have to look for these tokens. To do this, we are going to search the string for our tokens (one at a time) and if the token search finds results over 1, it replaces the seen tokens after the first viewed one. This ensures the original token is still there for reference and the duplicates are swapped with a QEC token. We just have to do this for all the tokens placed in the array.

A Quantum Entangled Compression (QEC) token is a specially formed token that uses our full reference array and some minor logic. The only difference between a pattern-based token and a QEC token is that the QEC token relies on logical considerations instead of referencing an array. So the QEC tokens only need the string it’s compressing. The QEC tokens are 2-3 slots long, with each slot handling a different pointer; this is identical to the pattern-based tokens (see Figure 2 for a visual explanation).

The first slot will be the full reference token that represents the location on the string. Because we can have a string larger than our upper limit, we naturally divide the string into a soft upper limit. In our case, the soft upper limit is the shortest of either the length of the full reference array or half the length of the string presented. So, as we perform the first slot considerations, we will loop through the groups but still assign the token-object found in the full reference array. For example, if we have 2 groups and both groups have a token-swap object seen at the 14th position, both will have an “N” in their 1st QEC token slot. There may be more than one formatted object in the 1st slot, so if you use characters in slot 1, you can use numbers in slot 1 before the character to show the group number. If you have over 9 groups, use double singles to showcase the groups as sections per 9. For example, if your total number of groups was 18, you could have 2 sets of 9 groups, IE., 99[…]. To explain again, if your total number of groups was 22, you could have 3 sets of groups, IE., 994[…]. This works because the highest digit(s) is always before the lowest digit, so we can know they are acting as a single bit in correlation to the character bit behind it. Please Note that using group number string-bits can inflate the end result so it is wiser to design your decoder and encoder to naturally look for groups without group numbers but this can be more difficult to reverse.

The second slot will be the full reference array but in the opposite case. So if you used uppercase in slot 1, you’ll use lower-case in slot 2. This means we only need one full reference array to build the other reference arrays but also this opposite case array. The object in the

2cd slot simply refers to the length of the token-object that was swapped out. So the first slot says, “Here’s where the token begins,” while the second slot says, “Here’s how long the token is”.

The optional third slot will be the full reference array again, but this time it is used as a “Repeat ID.”If we can repeat the token-object in a row and find that in the string after the first time we saw the token-object, we can replace all the extra portions of that loop by referencing the original smallest pattern and then saying, “How many times to repeat that pattern,” for the full return. This is also the only way we can replace part of a token object. So if we find a token-object within one of our token-objects, this replacement method would lower the length of that full reference that other tokens are pointing to. And this form of replacement has to be done last to keep confusion out of the way. Which means when detangling, you’ll need to consider these first or at least check for these collisions first.

If we always encode the 2-slot QEC tokens before the 3-slot QEC tokens, we should have reversible results, just like if we have 9 before the connecting bits, we can have multi-part strings working as a single bit. However, If we encode the 3-slot QEC tokens before the 2-slot QEC tokens or if we order our multi-part strings with the lowest value before the 9s, we may not be able to reverse the end result, similar to a secure hash but more like a "semi-secure-hash." QEC tokens are unique if they stick to the rules for them even though we are picking and choosing the natural entanglements we can see. Using this 3-slot token replacement method, we ensure that no matter the pattern, the process is the same, and reversal is up to the decoder & encoder.

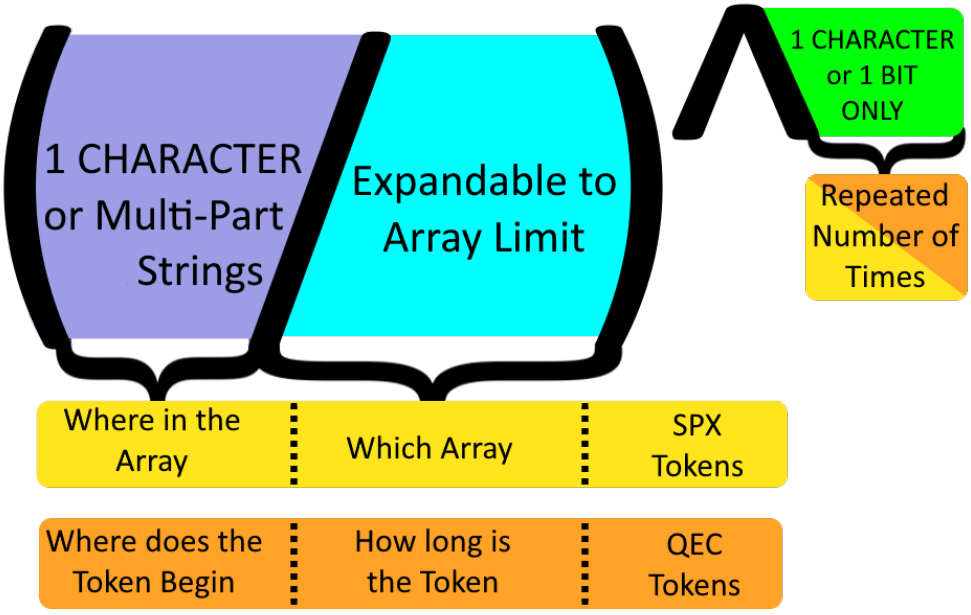



**Figure 2.** Token Map. The figure shows a mapping of swapping tokens and the difference between SPX Tokens & the QEC Tokens. QEC Tokens always rely on characters or the full reference array in two cases/states while the Non-QEC (or pattern-based) Tokens may begin alphanumerically then jump to full reference array in it's natural state, this is majorly how we tell the difference between both token sets. □ None of the mathematical symbols shown in this image are used in the tokens. The mathematical symbols are used in the imagery to explain how they work with each other.

Before showcasing the proofs through example, let's consider the rules to use quantum entanglement for compressing data.

Starting with the token used to replace sections within a string or data-set should use the same or similar but reflective objects when compared to the objects in the other slots, rather token-objects or alphanumeric-objects. The token used for swapping should be a 2-slot or 3-slot token.

The first token slot should always be a pointer, either alphanumerical or uniquely identifiable, and be the smallest possible length for its format. The second slot should always be the identifier, and no identifier should also be in the hot-swappable arrays (2cd Generated-Pattern array; 3rd Generated-Pattern array; 7th Array; Index Arrays; Mixing Arrays). The first token of the full reference array in both possible cases, or all possible cases (for non-alphabetic token-objects), should be used for L3+ repeating single characters. The only time the second slot is used for pointers is with multi-tagged tokens; in the example,

LOWLOWLOWCAP & LOWLOWCAP tokens for the unique edge cases use the second slot for additional pointers to the same array. The second slot is the main expandable slot to allow multiple pointers, in examples: LOW7tCAP, LOW7uCAP, LOW7vCAP, LOWtuvCAP, LOWLOWLOWCAP & LOWLOWCAP expand the second slot to give additional context without inflating per character (in most cases). All expanded second slot considerations should be done third-to-last in compression but third-to-do during detangling. The optional third slot should always be the same case as the full reference array case. The optional third slot is to inform of repeating needed to get back the full token that was swapped out.

The tokens that are the replaced sections within a string or data-set should be dynamically chosen or based on another section previously in the string. Any changes to these rules should be included in some form within the programming slots. Determiner slots are considered overflows. When adding non-binary or non-trinary data to the programming or determiner slots, use WaveDataAPIs [7], or use an in-house solution.

## *Setting the Data-Break*

We need to make sure there is some order to the chaos so we have an Application Binary Interface (ABI) to ensure we all can have a similar data structures even if we don't all use all the sections of the ABI.

```
Datagraph {

    Graphhash [
        previous pointer (base64 |or| base91) |or| radius (base64) if no previous
pointer
    ];
    Determiner slot 1 [
        anything
    ];
    Determiner slot 2 [
        anything
```

];

Programming slot 1 [

magic number (numerical; decimal preferred)

“x”

micro-loop point (numerical; decimal preferred)

“e” (normal encoding) |or| “f” (compressed encoding) |or| “g” (graphed encoding)

input length (numerical; decimal preferred)

];

Determiner slot 3 [

anything

];

Row 1[

anything,

anything,

anything

];

Determiner slot 4 [

anything

];

Row 2 [

anything,

anything,

anything

];

Programming slot 2 [

index

```
];
Row 3 [
        anything,
        anything,
        anything
                "."
                        Row 0 [
                                anything
                        ]
];
}
```

(Equation 19)

Equation 19 is the example ABI.

Utilizing the example table and information used so far we will transmute "Hello World" into a SuPosXt (SPX) hash.

| Data Name | Data/Output | Misc Data | Data Set Title |
|---|---|---|---|
| *Input (raw)* | Hello World | Input String | Input Information |
| *Length* | 11 | Input Length | |
| *Sum* | ((1.5*18)+11)-1=**37** | (ABSOLUTE((Full_Action_Counter * Max_Loop_length) + Input_Length) – 1) | |
| *Loop Point* | 1[x18] | (Round-Up[Previous_BIT + Length / 18]) | Loop Information |
| | *There was no previous bit | | |
| *Micro-Loop* | 2 | (Length % 9) | |
| *Full Action* Counter | 1.5 | ((Loop Point + Micro-Loop Point)/2) | |
| *Node Heat map* | [D1D0]0.2[A1A1A1]111[A0B0C0]6[A0B0C0]6[A1B1D0A1B1]55.01[D0]0[A0B0D1]50.1[A1B1D0A1B1]55.01[A1B1D0A0B1]55. | Node Output vis-a-vis Node (the EE2E font), "DNDregular.otf" & the "Full House Talk Model" | Node Output |

| | | | |
|---|---|---|---|
| | 01[A0B0C0]6[A0A1A1]111 | | |
| *Data Rail A* | 1110011011100011 | Faux-Binary State | |
| *Data Rail B* | 001101110 | | |
| *Data Rail C* | 000 | | |
| *Data Rail D* | 1000100 | | |
| *Index* | 0.21116655.01050.155.0155.016111 | | |
| *Full Grid* | 110011011100011;001101110;000;1000100;0.21116655.01050.155.0155.016111; | | |
| *Transmuting Data Rail A* | 1110011011100011 → **1110101001101101** | *before → after | Shifting or "Transmuting" Data |
| *Transmuting Data Rail B* | 001101110 → **010100110** | *before → after | |
| *Transmuting Data Rail C* | 000 → **011** | *before → after | |
| *Transmuting Data Rail D* | 1000100 → **1011001** | * before → after | |
| *Transmuted Index* | **0.21116655.01050.155.0155.016111** | *after | |
| *Transmuted Full Grid* | **1110101001101101;010100110;011;1011001;0.21116655.01050.155.0155.016111;** | *after | |
| *A*\|*D* | 0 | Parallelogram (Alignment Check A) | |
| *B*\|*C* | 180 | | |
| *a* | 37 | | |
| *b* | 685 | | |
| *h* | 1369 | | |
| *P* | 1444 | | |
| *Area* | 468768 | | |
| *Volume* | 1342730154 | Sphere (Alignment Check B) | |
| *Radius* | 684.5 | | |
| *Magic Number* | 101 | Magic Number Check | |
| *Chain Weight* | (((((11 * 3.14) + 37) % 2) ^ (3.14 + 11))) / 37 = **13** | Round-Up(ABSOLUTE(ABSOLUTE(CEILING(TRUNC(Input_Length * PI) + Input_Sum) % 2)^ABSOLUTE(PI + Input_Length))) / Input_Sum) | Chain Weight |

| **SPX Tokens** | **Array Tokens that were Swapped Out** |
|---|---|

| | | | |
|---|---|---|---|
| 0a | ===; | These were used for general compression & removing extra data | User Pattern-Based Tokens |
| 8a | ;1; | | |
| 7x | 11010 | | |
| 7w | 101101 | | |
| 8a | ;1; | | |
| 9a | ;0; | | |
| ov | 0.2 | | |
| sv | 111 | | |
| c7B | 66 | | |
| qu | 55.01 | | |
| pu | 50.1 | | |
| quB | 55.0155.01 | | |
| sv | 111 | | |
| 3A | ;0 | | |
| 8A | . | | |
| **QEC Token** | **Reference Token that was Swapped Out** | These were used for QEC<br>See Appendix C for unused QEC tokens | Used Quantum Tokens |
| 2Bd | 1001 | | |
| Vc | 118 | | |
| Pc | 101 | | |
| *Graphed Hash* | LTY4NC41cg===;1;1;101x2f11;1;1110101001101101;0;010100110;0.21116655.01050.155.0155.016111;011.1011001 | | Hash Based Output |
| *Compressed Graphed Hash* | LTY4NC41cg0a18a101x2f118a17x10017w8a010100119aovsvc7Bqu0puquB6sv3A118A1011001 | | |
| *Quantum Compressed Hash* | LTY4NC41cg0a18a101x2f118a17x10017w8a0102Bd19aovsvc7Bqu0puquB6sv3AVcAPc2Bd | | |
| *Input Uninterrupted Binary* | 01001000 01100101 01101100 01101100 01101111 00100000 01010111 01101111 01110010 01101100 01100100 □ | | Total Length: 88 (L88) □ |
| *Output* | LTY4NC41cg0a18a101x2f118a17x10017w8a0102Bd19aovsvc7Bqu0puquB6sv3AVcAPc2Bd □ | | Total Length: 73 (L73) □ |
| *Output Uninterrupted* | 11101010 01101101 01010011 0 011 1011001 0.211166 55.01050 .155.015 5.016111 | | Total Length: 67 (L67) |

*Faux-Binary*

**Example 4.** Input data and manipulated outputs, including final "output" showcasing the reduction of length based on the output's binary length after the token swaps and quantum entanglements have been completed. □ The descriptive titles of information below them are embolden while data name are italicized. Answers to internal equations are embolden to be seen easier and the internal equation is given to show some work. Finalized transmuted data is embolden.

***Theorem 4.*** *Superpositioning data with it's index and parity checks does inflate the data but with the various manipulations we can entangle a portion of the data and it's various manipulations to itself to both hide data & reduce the total length of the manipulated data.*

***Proof of Theorem 4.*** *Looking at* ***Example 4****, you can see the input was simply, "Hello World" and the lookup table for Node Outputs resulted in a Node Heat map that was separated into rails based on if the data was beside an "A", "B", "C" or "D" character, transmuted based on the relationship between each digit when compared to the previous digit starting at the left or front of the string. Showcased the alignment checks for the parallelogram data points & sphere data points before finding the magic number and chain weight. Finally we show the SPX tokens used to mix and compress before showcasing the graphed hash before any compression, after compression & after quantum compression before finally giving the original uninterrupted binary of the input & it's length to be compared to both the final output & it's length, but also, the faux-binary plus it's index & it's length to show we indeed compressed the data based on the original input's binary length.*

Looking at the end results shows that these quantum manipulations do lower the length of the binary equivalent of the inputted string, even after index & parity inflation. The end results may not seem magical but does showcase how we are superpositioning text with its index & parity checks while still reducing the size of the end string. Overall, in testing, the reduction of data does seem to grow in difference with the longer the data is from the full reference array. So this works best with inputs 1.5 times longer then the full reference array, in our case that would be inputs with a length of at least 39 or L39+. The quantum entanglement method may seem unorthodox to use un-associated character mixtures or multi-bit tokens, but when we use this alongside the rules put in place, we place tiny amounts of technical data without adding more information. Because the rules are utilizing a natural relationship in the alphabet while the arrays are setup to make each character of the alphabet is equal to it's position in the array, simply seeing where in the array a character is tell us a numerical point of information the same numerical point of information that just seeing

where that character appears in the alphabet would be. Adding in controlled order, we can use quantum entanglement in multiple ways and even stack the replacement tokens on top of each other in situations that would otherwise be irreversible while staying potentially reversible.

Even if we only use the QEC token swaps, we can still reduce the length of almost any hash or repeating-pattern strings without needing to know much more than it is a quantum manipulated hash string, which can be as simple as front running "QEC" or use the SPX hash format.

## *Discussion*

Encoding and Encrypting are vastly different with the exception that they change all data from end-to-end even if they are reversible. SPX takes on the belief that we can keep a small portion of the data and transmute the rest before applying "limited hashing" to only remove bits that are revealing, creating a proof of work method that is based solely on the data and not on any specific data other then the limitations of the programming language & full reference array.

Separating into sections before mixing and compressing, we have the start of a polymorphic encoding. The layers of mixing and compressing reduces the chances of SPX being used as a polymorphic encoder which would allow for sections of the hash to be opened independently from other sections. The real goal for separated sections was to allow for hidden APIs so when the correct receiver obtains the data, the data could tell the receiver what it is used for based on a per-arranged agreement vis-a-vis the API structure. So 3rd party receivers may not need to know what is in the data because it directs that automated receiver of what to do after being opened. This means, any API should be capable of entanglement to it's data with SPX allowing that data to direct itself when used correctly.

The quantum-manipulations and processes used are indeed simple and basic but does allow for a single-blind entanglement of information within the string to the same-looking information in the front of the string. We manipulate the data enough so it's not the data determining the manipulations but instead the end-look of the string that is being entangled to. This should be a way for different data to possibly have the same entanglement tokens but not always, making any QEC not directly tied to the original data being stored.

Using JavaScript as the originating programming language, this process should be repeatable in any new-age programming language as well as C++ or even x86 Assembly

Code, which should eventually allow for single chip computing of the SPX encoding technique. Most of the SPX system is designed around natural relationships between data partners and the human language we pair with the data but also around the concept of single loop finishers. SPX takes one loop to finish the task; even though SPX may need to make many hundreds of micro-loops, it should only need one full loop to do the task.

## *Conclusions*

SPX is a multitude of procedures working together to make a unified object output that should contain less than 1% of the originating data, should be unrecognizable as the original input & should be potentially reversible. SPX shows we don't need to encode everything to have more secure encoding. Using quantum-manipulations to prevent insecure-polymorphic encoding only helps compress our output further.

**Supplementary Materials:** The following supporting information can be downloaded at: https://github.com/DigiMancer3D/SPX/raw/refs/heads/main/DND-Regular.otf, E2EE Font Node [4]; https://github.com/DigiMancer3D/SPX/blob/main/Visuals/Reflection_Wheel.png, ***Figure 1***, Reflection Wheel [9]; https://github.com/DigiMancer3D/SPX/blob/main/Visuals/SuPosXt%20(encoding%20graph).png, SuPosXt (SPX) Encoding Infograph [10]; https://github.com/DigiMancer3D/SPX/blob/main/Visuals/Talk-flow-chart%20(e2ef-meme).png, "Full Hose Talk Model" [11]; https://github.com/DigiMancer3D/SPX/blob/main/Visuals/Token_Map.png, ***Figure 2***, Token Map [12].

**Author Contributions:** 3 Douglas B. Pihl.

**Funding:** "This research received no external funding".

**Data Availability Statement:** All of the data used for this including visualizations, text, previous paper drafts, code, previous test builds & all related information should be available in the SPX repository on the platform GitHub (for as long as GitHub agrees to host the content), https://github.com/DigiMancer3D/SPX.

**Acknowledgments:** I would like to add a special thank you to ***Jake La`Doge*** in assisting in the creation of the self-correcting methods and directing me to the "Futurama-Theorem," which ended up being the glue to getting this concept system to work dynamically with verification. I would like to add another special thank you to ***The Mota Club*** for helping in keeping me motivated to completion as well with spelling corrections for the previous SPX papers and concepts. I would like to add a final special thank you to ***Sydart*** *(@SYDART | SYDART.ETH)* involved in proof-reading and assisting in the grammar for the final draft.

**Conflicts of Interest:** "SPX started off of as a playful comment from the author, whom of which was also the funder. See ***Appendix A*** for more details."

## *Abbreviations*

The following abbreviations are used in this manuscript:

| | |
|---|---|
| MDPI | Multidisciplinary Digital Publishing Institute |
| SPX | Super Positioned Text |
| SuPosXt | Super Positioned Text |
| CHAR | Character |
| CHAR - CHAR | Character [starter] through Character [stopper] |
| IDE | Integrated Development Environment |
| AKA | Also Known As |
| DND | Dot-Node Demonstrated |
| E2EE | End-2-End Encryption |
| CRC | Cyclic Redundancy Check |
| ID | Identifier(s) |
| Arr | Array |
| P# | Prime [#] (the # is the prime position) |
| L# | "Length of [#]" (the # is the length total) |
| API | Application Programming Interface |
| 1/0 | First Array Position is One, Null Array Position is Null |
| ASCII | American Standard Code for Information Interchange |
| MFT | Modified Futurama-Theorem |
| SUM | Sum (or "+" for equations) |
| MOD | Modifier |
| ABS | Absolute Value (always in positive sign) |
| MOD2 | "% 2" (in equations) |
| PI | Pi (3.14[...]) |
| trunc | Truncate |
| sin | Sine |
| POW | "value of a base raised to a power" [JavaScript Maths] |
| LOW | Lower-Cased Character |
| CAP | Capital-Cased Character |
| NUM | Number |
| QEC | Quantum Entangled Compression |
| BIT | Bit |
| ABI | Application Binary Interface |

| | |
|---|---|
| C++ | [C-PLUS-PLUS] Extended C Programming Language |
| ETH | Ethereum (unofficially: Ether Electronic Universal Machine) |

# Appendix A

## SPX History:

To be perfectly honest, SPX started off because I wanted to create a new compression algorithm and was riffing with some friends at the Mota Club (a semi-private online virtual lounge on Telegram) and said something along the lines of, "I bet I could build a model around the way the people spoke to each other on 'Full House'." I designed Node font the next day & SPX began coding the day after that. SPX has already had many revisions and with each revision we were able to get closer to the target goal, to encode & decode with the talk model. Quantum manipulations didn't become apart of the SPX method until I had a semi-working version of SPX decoder & encoder, at that point I started being able to read the data without decoding. So, Quantum Manipulations was considered as I was trying to find the most dynamic pattern swaping I could come up with. Qunatum Manipulation may be a big part of SPX now, it only started as the smallest part of the entire system. Unfortunately not one version of SPX has done all the steps as planned without error, there has been 3 major versions.

- The first version of SPX (v0) only made the graphed hash and nothing more. Almost all parts of this original version are now just look up tables.
- The second version of SPX (v1) was capable of reversing the graphed hash at about 60% of the time successful or partly successful.
- The third version of SPX (v2) is capable of compression (including QEC) with varying results. Which has led to this paper.

Each version of SPX has also gained a new paper to explain the new version of SPX. Although the first version mostly proved the parity checks and graphed hash as capable. The second version helped prove the methods are indeed reversible. The third version has simply proved Quantum Entangled Compression as viable. Below I will have listed links to the code, live demos & their connected paper, as full transparency.

## SPX Versions:

Below are the 3 versions of SPX (v0, v1, v2), which area as follows:

1. SPX v0:

- SPX [v0] Code (via GitHub): https://github.com/DigiMancer3D/SPX/blob/main/OLD_FILES/Binary%20Parity%20Check%20test%20(old).html [13]
- SPX [v0] Live Demo as Proof (via GistHub): https://gistpreview.github.io/?6b7e5258dc97fb7e535ddd558b6d6612 [14]
- SPX [v0] Paper (via GitHub): https://github.com/DigiMancer3D/SPX/blob/main/OLD_FILES/OLD_README [15]

2. SPX v1:

- SPX [v1] Code (via GitHub): https://github.com/DigiMancer3D/SPX/blob/main/OLD_FILES/SuPosXt_encoding_Demo.html [16]
- SPX [v1] Live Demo as Proof (via GistHub): https://gistpreview.github.io/?3e603e4569f9617f4a418fda899160f8 [17]

*Click the "O" in "IO" within the header text, "Encrypt Text, Secure Input, Entangle IO (DEMO)" to attempt to decode any generated graphed hash.*

- SPX [v1] Paper (via GitHub): https://github.com/DigiMancer3D/SPX/blob/main/OLD_FILES/2cd_OLD_README [18]

3. SPX v2:

- SPX [v2] Code (via GitHub): https://github.com/DigiMancer3D/SPX/blob/main/test-state.html [19]
- SPX [v2] Live Demo as Proof (via GistHub): https://gistpreview.github.io/?2397fe2a9076e911a197621114875fe9/Live-SuPosXt-Demo.html [20]
- SPX [v2] Paper: (Currently Reading) Generic PDF style SPX v2 Paper (via GitHub): https://github.com/DigiMancer3D/SPX [21]
- 3rd SPX Paper (currently reading): ***N/A***

# Appendix B

All Reference Images:

***Array Reflection Wheel***

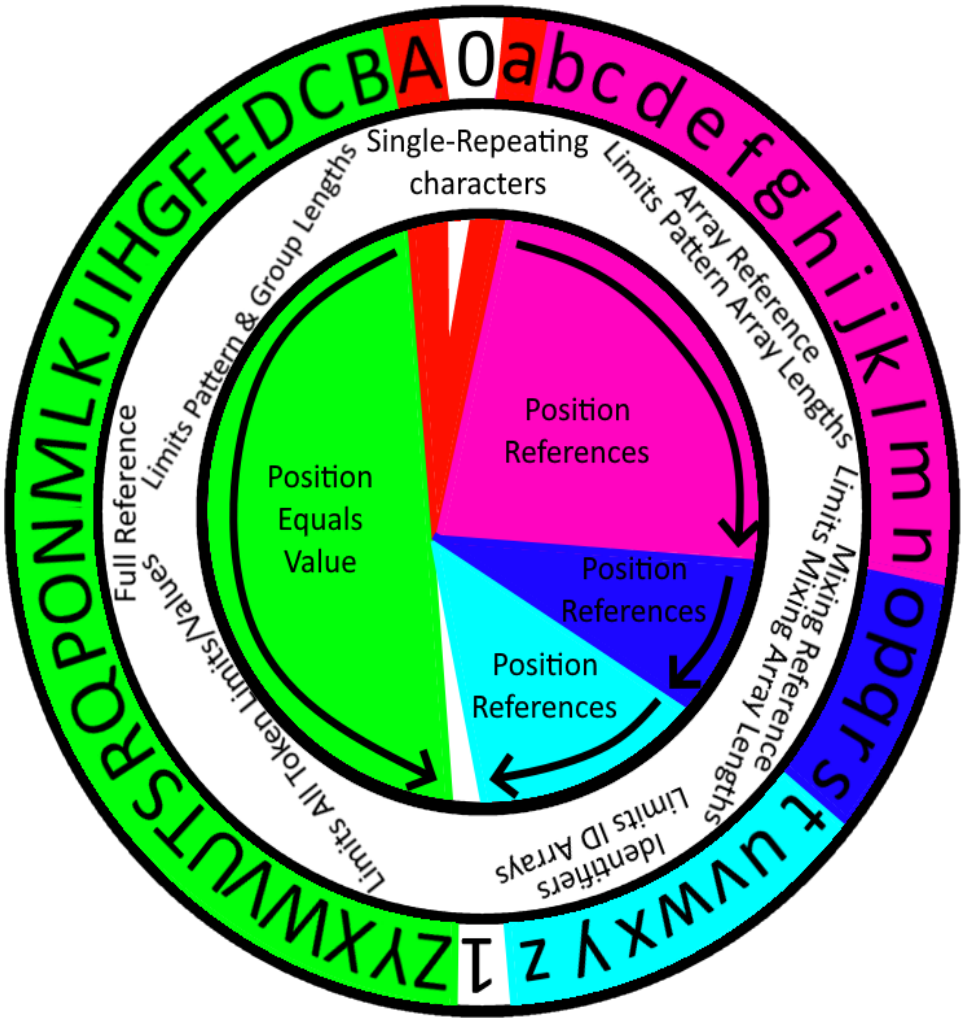



**Figure B1.** The "Array Reflection Wheel" showcases how each array and ID set rely on the full reference array to ensure the quantum stylized tokens work as intended based on the basic patterns and rule sets.

***Token Map***

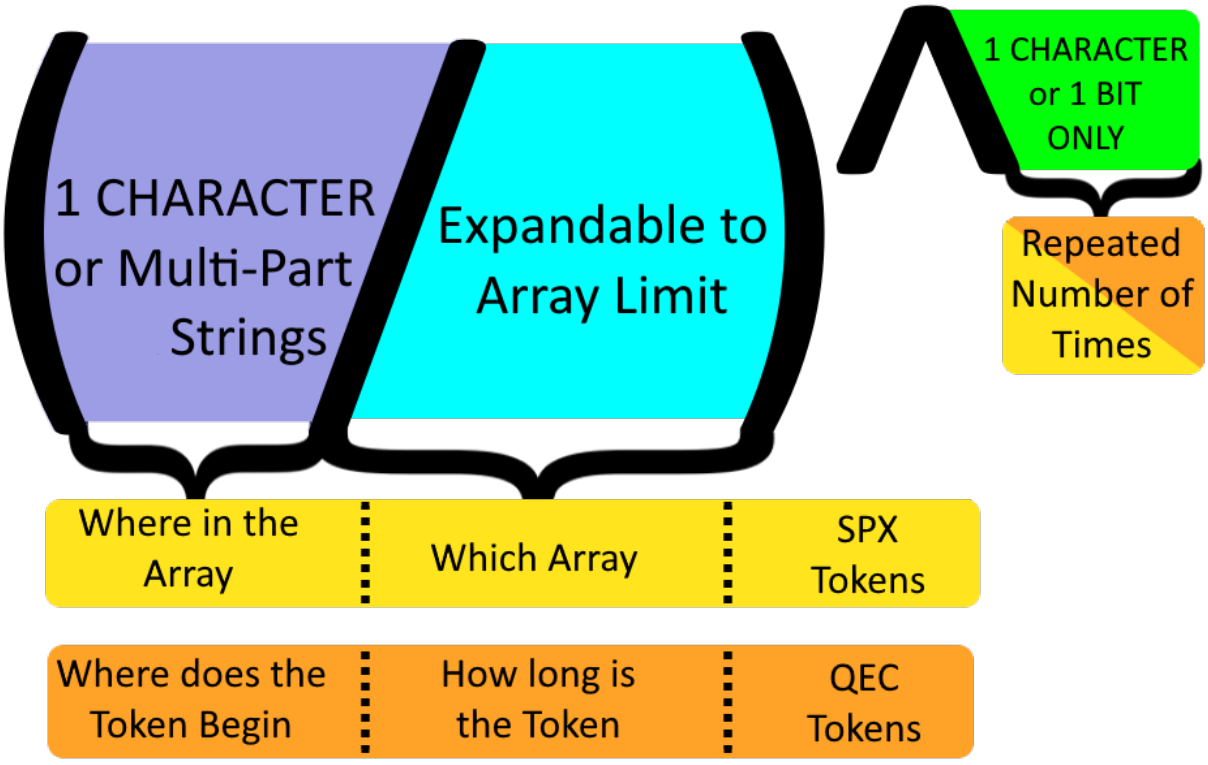



***Figure B2.*** *The "Token Map" showcases how the quantum stylized tokens work for both Quantum Entangled Compression tokens & Non-QEC tokens or Pattern-based tokens.*

*SPX Detailed Information graph*

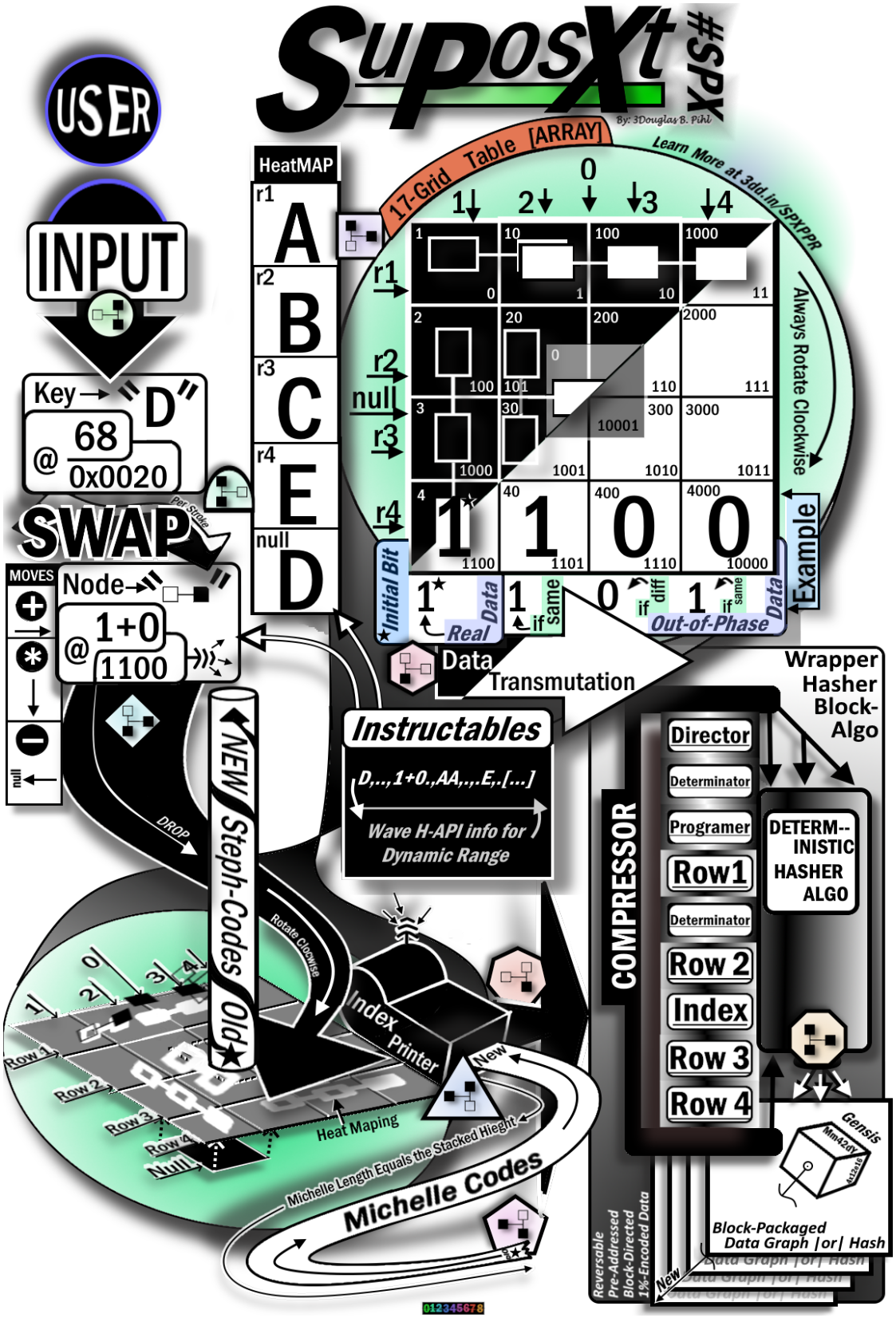


***Figure B3.*** *The "SPX Detailed Information Graph" is the original design behind SPX and the methodologies that became the Quantum Entangled Compression method. This showcases the 3-Dimensional graphing and how the heat mapping is translated from this mapping.*

***Compounding Hamming Code Map***

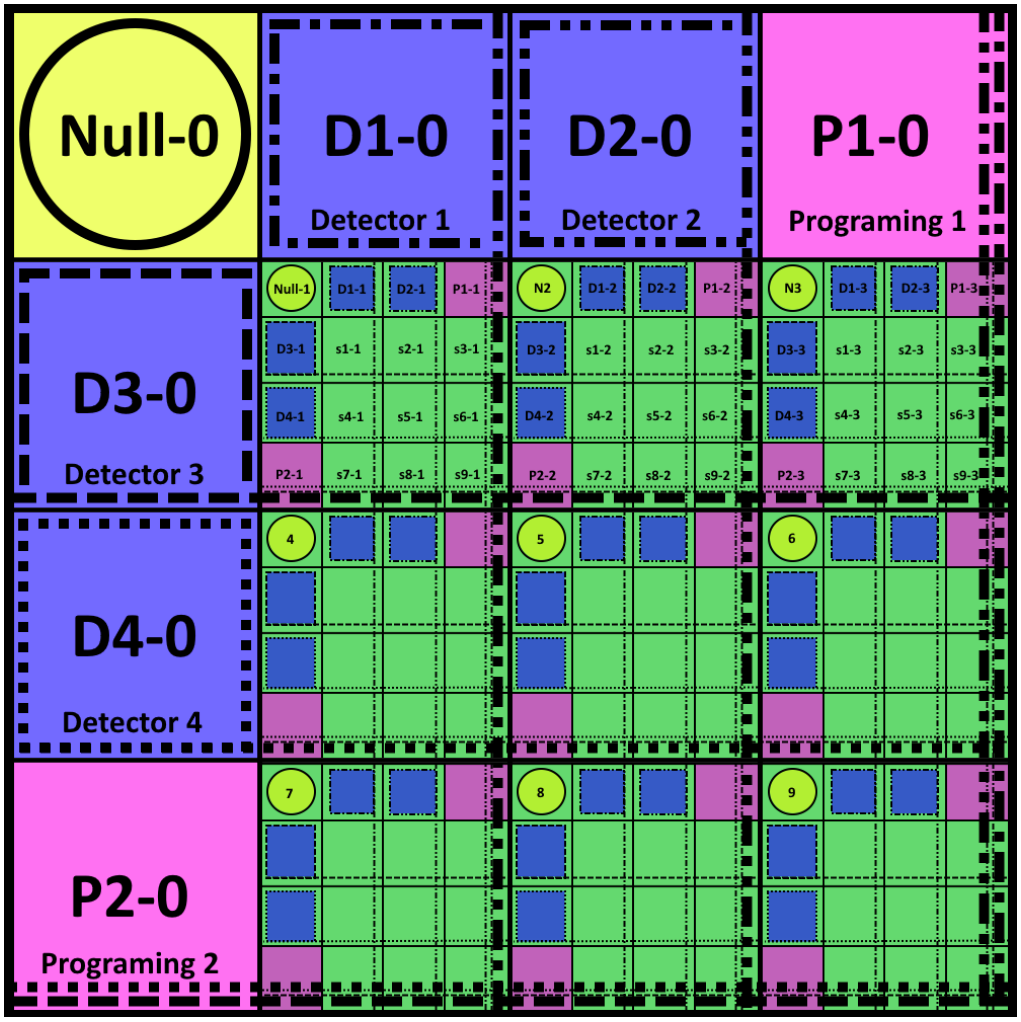



***Figure B4.*** *The "Compounding Hamming Code" image showcases how graphing creates an internal structure to allow for internally stored data that may not be viewable from the external hash. Although no versions of SPX at the time of writing this is setup to use compounding hamming code, SPX is designed initially around this idea of compounding SPX hashes to get smaller outputs overtime from the Quantum Entangled Compression.*

# Appendix C

Following this will be the unused Quantum Tokens (approximately 1271 sets of tokens) that was Generated during the SPX process for ***Example 4.***

Quantum Tokens Generated (*L4 – L26*):

**svc7Bqu0puquB6sv3A118A1011001; vsvc7Bqu0puquB6sv3A118A101100; ovsvc7Bqu0puquB6sv3A118A10110; aovsvc7Bqu0puquB6sv3A118A1011; 9aovsvc7Bqu0puquB6sv3A118A101; 19aovsvc7Bqu0puquB6sv3A118A10; 119aovsvc7Bqu0puquB6sv3A118A1; 0119aovsvc7Bqu0puquB6sv3A118A; 00119aovsvc7Bqu0puquB6sv3A118; 100119aovsvc7Bqu0puquB6sv3A11; 0100119aovsvc7Bqu0puquB6sv3A1; 10100119aovsvc7Bqu0puquB6sv3A; 010100119aovsvc7Bqu0puquB6sv3; a010100119aovsvc7Bqu0puquB6sv; 8a010100119aovsvc7Bqu0puquB6s; w8a010100119aovsvc7Bqu0puquB6; 7w8a010100119aovsvc7Bqu0puquB; 17w8a010100119aovsvc7Bqu0puqu; 017w8a010100119aovsvc7Bqu0puq; 0017w8a010100119aovsvc7Bqu0pu; 10017w8a010100119aovsvc7Bqu0p; x10017w8a010100119aovsvc7Bqu0; 7x10017w8a010100119aovsvc7Bqu; 17x10017w8a010100119aovsvc7Bq; vc7Bqu0puquB6sv3A118A1011001; svc7Bqu0puquB6sv3A118A101100; vsvc7Bqu0puquB6sv3A118A10110; ovsvc7Bqu0puquB6sv3A118A1011; aovsvc7Bqu0puquB6sv3A118A101; 9aovsvc7Bqu0puquB6sv3A118A10; 19aovsvc7Bqu0puquB6sv3A118A1; 119aovsvc7Bqu0puquB6sv3A118A; 0119aovsvc7Bqu0puquB6sv3A118; 00119aovsvc7Bqu0puquB6sv3A11; 100119aovsvc7Bqu0puquB6sv3A1; 0100119aovsvc7Bqu0puquB6sv3A; 10100119aovsvc7Bqu0puquB6sv3; 010100119aovsvc7Bqu0puquB6sv; a010100119aovsvc7Bqu0puquB6s; 8a010100119aovsvc7Bqu0puquB6; w8a010100119aovsvc7Bqu0puquB; 7w8a010100119aovsvc7Bqu0puqu; 17w8a010100119aovsvc7Bqu0puq; 017w8a010100119aovsvc7Bqu0pu; 0017w8a010100119aovsvc7Bqu0p; 10017w8a010100119aovsvc7Bqu0; x10017w8a010100119aovsvc7Bqu; 7x10017w8a010100119aovsvc7Bq; 17x10017w8a010100119aovsvc7B; a17x10017w8a010100119aovsvc7; c7Bqu0puquB6sv3A118A1011001; vc7Bqu0puquB6sv3A118A101100; svc7Bqu0puquB6sv3A118A10110; vsvc7Bqu0puquB6sv3A118A1011; ovsvc7Bqu0puquB6sv3A118A101; aovsvc7Bqu0puquB6sv3A118A10; 9aovsvc7Bqu0puquB6sv3A118A1; 19aovsvc7Bqu0puquB6sv3A118A; 119aovsvc7Bqu0puquB6sv3A118; 0119aovsvc7Bqu0puquB6sv3A11; 00119aovsvc7Bqu0puquB6sv3A1; 100119aovsvc7Bqu0puquB6sv3A; 0100119aovsvc7Bqu0puquB6sv3; 10100119aovsvc7Bqu0puquB6sv; 010100119aovsvc7Bqu0puquB6s; a010100119aovsvc7Bqu0puquB6; 8a010100119aovsvc7Bqu0puquB; w8a010100119aovsvc7Bqu0puqu; 7w8a010100119aovsvc7Bqu0puq; 17w8a010100119aovsvc7Bqu0pu; 017w8a010100119aovsvc7Bqu0p; 0017w8a010100119aovsvc7Bqu0; 10017w8a010100119aovsvc7Bqu; x10017w8a010100119aovsvc7Bq; 7x10017w8a010100119aovsvc7B; 17x10017w8a010100119aovsvc7; a17x10017w8a010100119aovsvc; 8a17x10017w8a010100119aovsv; 7Bqu0puquB6sv3A118A1011001; c7Bqu0puquB6sv3A118A101100; vc7Bqu0puquB6sv3A118A10110; svc7Bqu0puquB6sv3A118A1011; vsvc7Bqu0puquB6sv3A118A101; ovsvc7Bqu0puquB6sv3A118A10; aovsvc7Bqu0puquB6sv3A118A1; 9aovsvc7Bqu0puquB6sv3A118A; 19aovsvc7Bqu0puquB6sv3A118; 119aovsvc7Bqu0puquB6sv3A11; 0119aovsvc7Bqu0puquB6sv3A1; 00119aovsvc7Bqu0puquB6sv3A; 100119aovsvc7Bqu0puquB6sv3; 0100119aovsvc7Bqu0puquB6sv; 10100119aovsvc7Bqu0puquB6s; 010100119aovsvc7Bqu0puquB6; a010100119aovsvc7Bqu0puquB; 8a010100119aovsvc7Bqu0puqu; w8a010100119aovsvc7Bqu0puq; 7w8a010100119aovsvc7Bqu0pu; 17w8a010100119aovsvc7Bqu0p; 017w8a010100119aovsvc7Bqu0; 0017w8a010100119aovsvc7Bqu; 10017w8a010100119aovsvc7Bq; x10017w8a010100119aovsvc7B; 7x10017w8a010100119aovsvc7; 17x10017w8a010100119aovsvc; a17x10017w8a010100119aovsv; 8a17x10017w8a010100119aovs; 18a17x10017w8a010100119aov; Bqu0puquB6sv3A118A1011001; 7Bqu0puquB6sv3A118A101100; c7Bqu0puquB6sv3A118A10110; vc7Bqu0puquB6sv3A118A1011; svc7Bqu0puquB6sv3A118A101; vsvc7Bqu0puquB6sv3A118A10; ovsvc7Bqu0puquB6sv3A118A1; aovsvc7Bqu0puquB6sv3A118A; 9aovsvc7Bqu0puquB6sv3A118; 19aovsvc7Bqu0puquB6sv3A11; 119aovsvc7Bqu0puquB6sv3A1; 0119aovsvc7Bqu0puquB6sv3A; 00119aovsvc7Bqu0puquB6sv3; 100119aovsvc7Bqu0puquB6sv; 0100119aovsvc7Bqu0puquB6s; 10100119aovsvc7Bqu0puquB6; 010100119aovsvc7Bqu0puquB; a010100119aovsvc7Bqu0puqu; 8a010100119aovsvc7Bqu0puq; w8a010100119aovsvc7Bqu0pu; 7w8a010100119aovsvc7Bqu0p; 17w8a010100119aovsvc7Bqu0; 017w8a010100119aovsvc7Bqu; 0017w8a010100119aovsvc7Bq; 10017w8a010100119aovsvc7B; x10017w8a010100119aovsvc7; 7x10017w8a010100119aovsvc;**

17x10017w8a010100119aovsv; a17x10017w8a010100119aovs; 8a17x10017w8a010100119aov; 18a17x10017w8a010100119ao;
118a17x10017w8a010100119a; qu0puquB6sv3A118A1011001; Bqu0puquB6sv3A118A101100; 7Bqu0puquB6sv3A118A10110;
c7Bqu0puquB6sv3A118A1011; vc7Bqu0puquB6sv3A118A101; svc7Bqu0puquB6sv3A118A10; vsvc7Bqu0puquB6sv3A118A1;
ovsvc7Bqu0puquB6sv3A118A; aovsvc7Bqu0puquB6sv3A118; 9aovsvc7Bqu0puquB6sv3A11; 19aovsvc7Bqu0puquB6sv3A1;
119aovsvc7Bqu0puquB6sv3A; 0119aovsvc7Bqu0puquB6sv3; 00119aovsvc7Bqu0puquB6sv; 100119aovsvc7Bqu0puquB6s;
0100119aovsvc7Bqu0puquB6; 10100119aovsvc7Bqu0puquB; 010100119aovsvc7Bqu0puqu; a010100119aovsvc7Bqu0puq;
8a010100119aovsvc7Bqu0pu; w8a010100119aovsvc7Bqu0p; 7w8a010100119aovsvc7Bqu0; 17w8a010100119aovsvc7Bqu;
017w8a010100119aovsvc7Bq; 0017w8a010100119aovsvc7B; 10017w8a010100119aovsvc7; x10017w8a010100119aovsvc;
7x10017w8a010100119aovsv; 17x10017w8a010100119aovs; a17x10017w8a010100119aov; 8a17x10017w8a010100119ao;
18a17x10017w8a010100119a; 118a17x10017w8a010100119; f118a17x10017w8a01010011; u0puquB6sv3A118A1011001;
qu0puquB6sv3A118A101100; Bqu0puquB6sv3A118A10110; 7Bqu0puquB6sv3A118A1011; c7Bqu0puquB6sv3A118A101;
vc7Bqu0puquB6sv3A118A10; svc7Bqu0puquB6sv3A118A1; vsvc7Bqu0puquB6sv3A118A; ovsvc7Bqu0puquB6sv3A118;
aovsvc7Bqu0puquB6sv3A11; 9aovsvc7Bqu0puquB6sv3A1; 19aovsvc7Bqu0puquB6sv3A; 119aovsvc7Bqu0puquB6sv3;
0119aovsvc7Bqu0puquB6sv; 00119aovsvc7Bqu0puquB6s; 100119aovsvc7Bqu0puquB6; 0100119aovsvc7Bqu0puquB;
10100119aovsvc7Bqu0puqu; 010100119aovsvc7Bqu0puq; a010100119aovsvc7Bqu0pu; 8a010100119aovsvc7Bqu0p;
w8a010100119aovsvc7Bqu0; 7w8a010100119aovsvc7Bqu; 17w8a010100119aovsvc7Bq; 017w8a010100119aovsvc7B;
0017w8a010100119aovsvc7; 10017w8a010100119aovsvc; x10017w8a010100119aovsv; 7x10017w8a010100119aovs; 17x10017w8a010100119aov;
a17x10017w8a010100119ao; 8a17x10017w8a010100119a; 18a17x10017w8a010100119; 118a17x10017w8a01010011; f118a17x10017w8a0101001;
2f118a17x10017w8a010100; 0puquB6sv3A118A1011001; u0puquB6sv3A118A101100; qu0puquB6sv3A118A10110;
Bqu0puquB6sv3A118A1011; 7Bqu0puquB6sv3A118A101; c7Bqu0puquB6sv3A118A10; vc7Bqu0puquB6sv3A118A1;
svc7Bqu0puquB6sv3A118A; vsvc7Bqu0puquB6sv3A118; ovsvc7Bqu0puquB6sv3A11; aovsvc7Bqu0puquB6sv3A1;
9aovsvc7Bqu0puquB6sv3A; 19aovsvc7Bqu0puquB6sv3; 119aovsvc7Bqu0puquB6sv; 0119aovsvc7Bqu0puquB6s;
00119aovsvc7Bqu0puquB6; 100119aovsvc7Bqu0puquB; 0100119aovsvc7Bqu0puqu; 10100119aovsvc7Bqu0puq; 010100119aovsvc7Bqu0pu;
a010100119aovsvc7Bqu0p; 8a010100119aovsvc7Bqu0; w8a010100119aovsvc7Bqu; 7w8a010100119aovsvc7Bq; 17w8a010100119aovsvc7B;
017w8a010100119aovsvc7; 0017w8a010100119aovsvc; 10017w8a010100119aovsv; x10017w8a010100119aovs; 7x10017w8a010100119aov;
17x10017w8a010100119ao; a17x10017w8a010100119a; 8a17x10017w8a010100119; 18a17x10017w8a01010011; 118a17x10017w8a0101001;
f118a17x10017w8a010100; 2f118a17x10017w8a01010; x2f118a17x10017w8a0101; puquB6sv3A118A1011001; 0puquB6sv3A118A101100;
u0puquB6sv3A118A10110; qu0puquB6sv3A118A1011; Bqu0puquB6sv3A118A101; 7Bqu0puquB6sv3A118A10; c7Bqu0puquB6sv3A118A1;
vc7Bqu0puquB6sv3A118A; svc7Bqu0puquB6sv3A118; vsvc7Bqu0puquB6sv3A11; ovsvc7Bqu0puquB6sv3A1; aovsvc7Bqu0puquB6sv3A;
9aovsvc7Bqu0puquB6sv3; 19aovsvc7Bqu0puquB6sv; 119aovsvc7Bqu0puquB6s; 0119aovsvc7Bqu0puquB6; 00119aovsvc7Bqu0puquB;
100119aovsvc7Bqu0puqu; 0100119aovsvc7Bqu0puq; 10100119aovsvc7Bqu0pu; 010100119aovsvc7Bqu0p; a010100119aovsvc7Bqu0;
8a010100119aovsvc7Bqu; w8a010100119aovsvc7Bq; 7w8a010100119aovsvc7B; 17w8a010100119aovsvc7; 017w8a010100119aovsvc;
0017w8a010100119aovsv; 10017w8a010100119aovs; x10017w8a010100119aov; 7x10017w8a010100119ao; 17x10017w8a010100119a;
a17x10017w8a010100119; 8a17x10017w8a01010011; 18a17x10017w8a0101001; 118a17x10017w8a010100; f118a17x10017w8a01010;
2f118a17x10017w8a0101; x2f118a17x10017w8a010; 1x2f118a17x10017w8a01; uquB6sv3A118A1011001; puquB6sv3A118A101100;
0puquB6sv3A118A10110; u0puquB6sv3A118A1011; qu0puquB6sv3A118A101; Bqu0puquB6sv3A118A10; 7Bqu0puquB6sv3A118A1;
c7Bqu0puquB6sv3A118A; vc7Bqu0puquB6sv3A118; svc7Bqu0puquB6sv3A11; vsvc7Bqu0puquB6sv3A1; ovsvc7Bqu0puquB6sv3A;
aovsvc7Bqu0puquB6sv3; 9aovsvc7Bqu0puquB6sv; 19aovsvc7Bqu0puquB6s; 119aovsvc7Bqu0puquB6; 0119aovsvc7Bqu0puquB;
00119aovsvc7Bqu0puqu; 100119aovsvc7Bqu0puq; 0100119aovsvc7Bqu0pu; 10100119aovsvc7Bqu0p; 010100119aovsvc7Bqu0;
a010100119aovsvc7Bqu; 8a010100119aovsvc7Bq; w8a010100119aovsvc7B; 7w8a010100119aovsvc7; 17w8a010100119aovsvc;
017w8a010100119aovsv; 0017w8a010100119aovs; 10017w8a010100119aov; x10017w8a010100119ao; 7x10017w8a010100119a;
17x10017w8a010100119; a17x10017w8a01010011; 8a17x10017w8a0101001; 18a17x10017w8a010100; 118a17x10017w8a01010;
f118a17x10017w8a0101; 2f118a17x10017w8a010; x2f118a17x10017w8a01; 1x2f118a17x10017w8a0; 01x2f118a17x10017w8a;
quB6sv3A118A1011001; uquB6sv3A118A101100; puquB6sv3A118A10110; 0puquB6sv3A118A1011; u0puquB6sv3A118A101;
qu0puquB6sv3A118A10; Bqu0puquB6sv3A118A1; 7Bqu0puquB6sv3A118A; c7Bqu0puquB6sv3A118; vc7Bqu0puquB6sv3A11;
svc7Bqu0puquB6sv3A1; vsvc7Bqu0puquB6sv3A; ovsvc7Bqu0puquB6sv3; aovsvc7Bqu0puquB6sv; 9aovsvc7Bqu0puquB6s;
19aovsvc7Bqu0puquB6; 119aovsvc7Bqu0puquB; 0119aovsvc7Bqu0puqu; 00119aovsvc7Bqu0puq; 100119aovsvc7Bqu0pu;
0100119aovsvc7Bqu0p; 10100119aovsvc7Bqu0; 010100119aovsvc7Bqu; a010100119aovsvc7Bq; 8a010100119aovsvc7B; w8a010100119aovsvc7;
7w8a010100119aovsvc; 17w8a010100119aovsv; 017w8a010100119aovs; 0017w8a010100119aov; 10017w8a010100119ao; x10017w8a010100119a;
7x10017w8a010100119; 17x10017w8a01010011; a17x10017w8a0101001; 8a17x10017w8a010100; 18a17x10017w8a01010; 118a17x10017w8a0101;
f118a17x10017w8a010; 2f118a17x10017w8a01; x2f118a17x10017w8a0; 1x2f118a17x10017w8a; 01x2f118a17x10017w8; 101x2f118a17x10017w;
uB6sv3A118A1011001; quB6sv3A118A101100; uquB6sv3A118A10110; puquB6sv3A118A1011; 0puquB6sv3A118A101;
u0puquB6sv3A118A10; qu0puquB6sv3A118A1; Bqu0puquB6sv3A118A; 7Bqu0puquB6sv3A118; c7Bqu0puquB6sv3A11;

vc7Bqu0puquB6sv3A1; svc7Bqu0puquB6sv3A; vsvc7Bqu0puquB6sv3; ovsvc7Bqu0puquB6sv; aovsvc7Bqu0puquB6s; 9aovsvc7Bqu0puquB6; 19aovsvc7Bqu0puquB; 119aovsvc7Bqu0puqu; 0119aovsvc7Bqu0puq; 00119aovsvc7Bqu0pu;

100119aovsvc7Bqu0p; 0100119aovsvc7Bqu0; 10100119aovsvc7Bqu; 010100119aovsvc7Bq; a010100119aovsvc7B; 8a010100119aovsvc7; w8a010100119aovsvc; 7w8a010100119aovsv; 17w8a010100119aovs; 017w8a010100119aov; 0017w8a010100119ao; 10017w8a010100119a; x10017w8a010100119; 7x10017w8a01010011; 17x10017w8a0101001; a17x10017w8a010100; 8a17x10017w8a01010; 18a17x10017w8a0101; 118a17x10017w8a010; f118a17x10017w8a01; 2f118a17x10017w8a0; x2f118a17x10017w8a; 1x2f118a17x10017w8; 01x2f118a17x10017w; 101x2f118a17x10017; a101x2f118a17x1001; B6sv3A118A1011001; uB6sv3A118A101100; quB6sv3A118A10110; uquB6sv3A118A1011; puquB6sv3A118A101; 0puquB6sv3A118A10; u0puquB6sv3A118A1; qu0puquB6sv3A118A; Bqu0puquB6sv3A118; 7Bqu0puquB6sv3A11; c7Bqu0puquB6sv3A1; vc7Bqu0puquB6sv3A; svc7Bqu0puquB6sv3; vsvc7Bqu0puquB6sv; ovsvc7Bqu0puquB6s; aovsvc7Bqu0puquB6; 9aovsvc7Bqu0puquB; 19aovsvc7Bqu0puqu; 119aovsvc7Bqu0puq; 0119aovsvc7Bqu0pu; 00119aovsvc7Bqu0p; 100119aovsvc7Bqu0; 0100119aovsvc7Bqu; 10100119aovsvc7Bq; 010100119aovsvc7B; a010100119aovsvc7; 8a010100119aovsvc; w8a010100119aovsv; 7w8a010100119aovs; 17w8a010100119aov; 017w8a010100119ao; 0017w8a010100119a; 10017w8a010100119; x10017w8a01010011; 7x10017w8a0101001; 17x10017w8a010100; a17x10017w8a01010; 8a17x10017w8a0101; 18a17x10017w8a010; 118a17x10017w8a01; f118a17x10017w8a0; 2f118a17x10017w8a; x2f118a17x10017w8; 1x2f118a17x10017w; 01x2f118a17x10017; 101x2f118a17x1001; a101x2f118a17x100; 8a101x2f118a17x10; 6sv3A118A1011001; B6sv3A118A101100; uB6sv3A118A10110; quB6sv3A118A1011; uquB6sv3A118A101; puquB6sv3A118A10; 0puquB6sv3A118A1; u0puquB6sv3A118A; qu0puquB6sv3A118; Bqu0puquB6sv3A11; 7Bqu0puquB6sv3A1; c7Bqu0puquB6sv3A; vc7Bqu0puquB6sv3; svc7Bqu0puquB6sv; vsvc7Bqu0puquB6s; ovsvc7Bqu0puquB6; aovsvc7Bqu0puquB; 9aovsvc7Bqu0puqu; 19aovsvc7Bqu0puq; 119aovsvc7Bqu0pu; 0119aovsvc7Bqu0p; 00119aovsvc7Bqu0; 100119aovsvc7Bqu; 0100119aovsvc7Bq; 10100119aovsvc7B; 010100119aovsvc7; a010100119aovsvc; 8a010100119aovsv; w8a010100119aovs; 7w8a010100119aov; 17w8a010100119ao; 017w8a010100119a; 0017w8a010100119; 10017w8a01010011; x10017w8a0101001; 7x10017w8a010100; 17x10017w8a01010; a17x10017w8a0101; 8a17x10017w8a010; 18a17x10017w8a01; 118a17x10017w8a0; f118a17x10017w8a; 2f118a17x10017w8; x2f118a17x10017w; 1x2f118a17x10017; 01x2f118a17x1001; 101x2f118a17x100; a101x2f118a17x10; 8a101x2f118a17x1; 18a101x2f118a17x; sv3A118A1011001; 6sv3A118A101100; B6sv3A118A10110; uB6sv3A118A1011; quB6sv3A118A101; uquB6sv3A118A10; puquB6sv3A118A1; 0puquB6sv3A118A; u0puquB6sv3A118; qu0puquB6sv3A11; Bqu0puquB6sv3A1; 7Bqu0puquB6sv3A; c7Bqu0puquB6sv3; vc7Bqu0puquB6sv; svc7Bqu0puquB6s; vsvc7Bqu0puquB6; ovsvc7Bqu0puquB; aovsvc7Bqu0puqu; 9aovsvc7Bqu0puq; 19aovsvc7Bqu0pu; 119aovsvc7Bqu0p; 0119aovsvc7Bqu0; 00119aovsvc7Bqu; 100119aovsvc7Bq; 0100119aovsvc7B; 10100119aovsvc7; 010100119aovsvc; a010100119aovsv; 8a010100119aovs; w8a010100119aov; 7w8a010100119ao; 17w8a010100119a; 017w8a010100119; 0017w8a01010011; 10017w8a0101001; x10017w8a010100; 7x10017w8a01010; 17x10017w8a0101; a17x10017w8a010; 8a17x10017w8a01; 18a17x10017w8a0; 118a17x10017w8a; f118a17x10017w8; 2f118a17x10017w; x2f118a17x10017; 1x2f118a17x1001; 01x2f118a17x100; 101x2f118a17x10; a101x2f118a17x1; 8a101x2f118a17x; 18a101x2f118a17; a18a101x2f118a1; v3A118A1011001; sv3A118A101100; 6sv3A118A10110; B6sv3A118A1011; uB6sv3A118A101; quB6sv3A118A10; uquB6sv3A118A1; puquB6sv3A118A; 0puquB6sv3A118; u0puquB6sv3A11; qu0puquB6sv3A1; Bqu0puquB6sv3A; 7Bqu0puquB6sv3; c7Bqu0puquB6sv; vc7Bqu0puquB6s; svc7Bqu0puquB6; vsvc7Bqu0puquB; ovsvc7Bqu0puqu; aovsvc7Bqu0puq; 9aovsvc7Bqu0pu; 19aovsvc7Bqu0p; 119aovsvc7Bqu0; 0119aovsvc7Bqu; 00119aovsvc7Bq; 100119aovsvc7B; 0100119aovsvc7; 10100119aovsvc; 010100119aovsv; a010100119aovs; 8a010100119aov; w8a010100119ao; 7w8a010100119a; 17w8a010100119; 017w8a01010011; 0017w8a0101001; 10017w8a010100; x10017w8a01010; 7x10017w8a0101; 17x10017w8a010; a17x10017w8a01; 8a17x10017w8a0; 18a17x10017w8a; 118a17x10017w8; f118a17x10017w; 2f118a17x10017; x2f118a17x1001; 1x2f118a17x100; 01x2f118a17x10; 101x2f118a17x1; a101x2f118a17x; 8a101x2f118a17; 18a101x2f118a1; a18a101x2f118a; 0a18a101x2f118; 3A118A1011001; v3A118A101100; sv3A118A10110; 6sv3A118A1011; B6sv3A118A101; uB6sv3A118A10; quB6sv3A118A1; uquB6sv3A118A; puquB6sv3A118; 0puquB6sv3A11; u0puquB6sv3A1; qu0puquB6sv3A; Bqu0puquB6sv3; 7Bqu0puquB6sv; c7Bqu0puquB6s; vc7Bqu0puquB6; svc7Bqu0puquB; vsvc7Bqu0puqu; ovsvc7Bqu0puq; aovsvc7Bqu0pu; 9aovsvc7Bqu0p; 19aovsvc7Bqu0; 119aovsvc7Bqu; 0119aovsvc7Bq; 00119aovsvc7B; 100119aovsvc7; 0100119aovsvc; 10100119aovsv; 010100119aovs; a010100119aov; 8a010100119ao; w8a010100119a; 7w8a010100119; 17w8a01010011; 017w8a0101001; 0017w8a010100; 10017w8a01010; x10017w8a0101; 7x10017w8a010; 17x10017w8a01; a17x10017w8a0; 8a17x10017w8a; 18a17x10017w8; 118a17x10017w; f118a17x10017; 2f118a17x1001; x2f118a17x100; 1x2f118a17x10; 01x2f118a17x1; 101x2f118a17x; a101x2f118a17; 8a101x2f118a1; 18a101x2f118a; a18a101x2f118; 0a18a101x2f11; g0a18a101x2f1; A118A1011001; 3A118A101100; v3A118A10110; sv3A118A1011; 6sv3A118A101; B6sv3A118A10; uB6sv3A118A1; quB6sv3A118A; uquB6sv3A118; puquB6sv3A11; 0puquB6sv3A1; u0puquB6sv3A; qu0puquB6sv3; Bqu0puquB6sv; 7Bqu0puquB6s; c7Bqu0puquB6; vc7Bqu0puquB; svc7Bqu0puqu; vsvc7Bqu0puq; ovsvc7Bqu0pu; aovsvc7Bqu0p; 9aovsvc7Bqu0; 19aovsvc7Bqu; 119aovsvc7Bq; 0119aovsvc7B; 00119aovsvc7; 100119aovsvc; 0100119aovsv; 10100119aovs; 010100119aov; a010100119ao; 8a010100119a; w8a010100119; 7w8a01010011; 17w8a0101001; 017w8a010100; 0017w8a01010; 10017w8a0101; x10017w8a010; 7x10017w8a01; 17x10017w8a0; a17x10017w8a; 8a17x10017w8; 18a17x10017w; 118a17x10017; f118a17x1001; 2f118a17x100; x2f118a17x10; 1x2f118a17x1; 01x2f118a17x; 101x2f118a17; a101x2f118a1; 8a101x2f118a; 18a101x2f118; a18a101x2f11; 0a18a101x2f1; g0a18a101x2f; cg0a18a101x2; 118A1011001; A118A101100; 3A118A10110; v3A118A1011; sv3A118A101; 6sv3A118A10; B6sv3A118A1; uB6sv3A118A; quB6sv3A118; uquB6sv3A11; puquB6sv3A1; 0puquB6sv3A; u0puquB6sv3; qu0puquB6sv; Bqu0puquB6s; 7Bqu0puquB6; c7Bqu0puquB; vc7Bqu0puqu; svc7Bqu0puq; vsvc7Bqu0pu; ovsvc7Bqu0p; aovsvc7Bqu0; 9aovsvc7Bqu; 19aovsvc7Bq; 119aovsvc7B; 0119aovsvc7; 00119aovsvc; 100119aovsv; 0100119aovs; 10100119aov; 010100119ao;

**a010100119a; 8a010100119; w8a01010011; 7w8a0101001; 17w8a010100; 017w8a01010; 0017w8a0101; 10017w8a010; x10017w8a01; 7x10017w8a0; 17x10017w8a; a17x10017w8; 8a17x10017w; 18a17x10017; 118a17x1001; f118a17x100; 2f118a17x10; x2f118a17x1; 1x2f118a17x; 01x2f118a17; 101x2f118a1; a101x2f118a; 8a101x2f118; 18a101x2f11; a18a101x2f1; 0a18a101x2f; g0a18a101x2; cg0a18a101x; 1cg0a18a101; 18A1011001; 118A101100; A118A10110; 3A118A1011; v3A118A101; sv3A118A10; 6sv3A118A1; B6sv3A118A; uB6sv3A118; quB6sv3A11; uquB6sv3A1; puquB6sv3A; 0puquB6sv3; u0puquB6sv; qu0puquB6s; Bqu0puquB6; 7Bqu0puquB; c7Bqu0puqu; vc7Bqu0puq; svc7Bqu0pu; vsvc7Bqu0p; ovsvc7Bqu0; aovsvc7Bqu; 9aovsvc7Bq; 19aovsvc7B; 119aovsvc7; 0119aovsvc; 00119aovsv; 100119aovs; 0100119aov; 10100119ao; 010100119a; a010100119; 8a01010011; w8a0101001; 7w8a010100; 17w8a01010; 017w8a0101; 0017w8a010; 10017w8a01; x10017w8a0; 7x10017w8a; 17x10017w8; a17x10017w; 8a17x10017; 18a17x1001; 118a17x100; f118a17x10; 2f118a17x1; x2f118a17x; 1x2f118a17; 01x2f118a1; 101x2f118a; a101x2f118; 8a101x2f11; 18a101x2f1; a18a101x2f; 0a18a101x2; g0a18a101x; cg0a18a101; 1cg0a18a10; 41cg0a18a1; 8A1011001; 18A101100; 118A10110; A118A1011; 3A118A101; v3A118A10; sv3A118A1; 6sv3A118A; B6sv3A118; uB6sv3A11; quB6sv3A1; uquB6sv3A; puquB6sv3; 0puquB6sv; u0puquB6s; qu0puquB6; Bqu0puquB; 7Bqu0puqu; c7Bqu0puq; vc7Bqu0pu; svc7Bqu0p; vsvc7Bqu0; ovsvc7Bqu; aovsvc7Bq; 9aovsvc7B; 19aovsvc7; 119aovsvc; 0119aovsv; 00119aovs; 100119aov; 0100119ao; 10100119a; 010100119; a01010011; 8a0101001; w8a010100; 7w8a01010; 17w8a0101; 017w8a010; 0017w8a01; 10017w8a0; x10017w8a; 7x10017w8; 17x10017w; a17x10017; 8a17x1001; 18a17x100; 118a17x10; f118a17x1; 2f118a17x; x2f118a17; 1x2f118a1; 01x2f118a; 101x2f118; a101x2f11; 8a101x2f1; 18a101x2f; a18a101x2; 0a18a101x; g0a18a101; cg0a18a10; 1cg0a18a1; 41cg0a18a; C41cg0a18; A1011001; 8A101100; 18A10110; 118A1011; A118A101; 3A118A10; v3A118A1; sv3A118A; 6sv3A118; B6sv3A11; uB6sv3A1; quB6sv3A; uquB6sv3; puquB6sv; 0puquB6s; u0puquB6; qu0puquB; Bqu0puqu; 7Bqu0puq; c7Bqu0pu; vc7Bqu0p; svc7Bqu0; vsvc7Bqu; ovsvc7Bq; aovsvc7B; 9aovsvc7; 19aovsvc; 119aovsv; 0119aovs; 00119aov; 100119ao; 0100119a; 10100119; 01010011; a0101001; 8a010100; w8a01010; 7w8a0101; 17w8a010; 017w8a01; 0017w8a0; 10017w8a; x10017w8; 7x10017w; 17x10017; a17x1001; 8a17x100; 18a17x10; 118a17x1; f118a17x; 2f118a17; x2f118a1; 1x2f118a; 01x2f118; 101x2f11; a101x2f1; 8a101x2f; 18a101x2; a18a101x; 0a18a101; g0a18a10; cg0a18a1; 1cg0a18a; 41cg0a18; C41cg0a1; NC41cg0a; 1011001; A101100; 8A10110; 18A1011; 118A101; A118A10; 3A118A1; v3A118A; sv3A118; 6sv3A11; B6sv3A1; uB6sv3A; quB6sv3; uquB6sv; puquB6s; 0puquB6; u0puquB; qu0puqu; Bqu0puq; 7Bqu0pu; c7Bqu0p; vc7Bqu0; svc7Bqu; vsvc7Bq; ovsvc7B; aovsvc7; 9aovsvc; 19aovsv; 119aovs; 0119aov; 00119ao; 100119a; 0100119; 1010011; 0101001; a010100; 8a01010; w8a0101; 7w8a010; 17w8a01; 017w8a0; 0017w8a; 10017w8; x10017w; 7x10017; 17x1001; a17x100; 8a17x10; 18a17x1; 118a17x; f118a17; 2f118a1; x2f118a; 1x2f118; 01x2f11; 101x2f1; a101x2f; 8a101x2; 18a101x; a18a101; 0a18a10; g0a18a1; cg0a18a; 1cg0a18; 41cg0a1; C41cg0a; NC41cg0; 4NC41cg; 011001; 101100; A10110; 8A1011; 18A101; 118A10; A118A1; 3A118A; v3A118; sv3A11; 6sv3A1; B6sv3A; uB6sv3; quB6sv; uquB6s; puquB6; 0puquB; u0puqu; qu0puq; Bqu0pu; 7Bqu0p; c7Bqu0; vc7Bqu; svc7Bq; vsvc7B; ovsvc7; aovsvc; 9aovsv; 19aovs; 119aov; 0119ao; 00119a; 100119; 010011; 101001; 010100; a01010; 8a0101; w8a010; 7w8a01; 17w8a0; 017w8a; 0017w8; 10017w; x10017; 7x1001; 17x100; a17x10; 8a17x1; 18a17x; 118a17; f118a1; 2f118a; x2f118; 1x2f11; 01x2f1; 101x2f; a101x2; 8a101x; 18a101; a18a10; 0a18a1; g0a18a; cg0a18; 1cg0a1; 41cg0a; C41cg0; NC41cg; 4NC41c; Y4NC41; 11001; 01100; 10110; A1011; 8A101; 18A10; 118A1; A118A; 3A118; v3A11; sv3A1; 6sv3A; B6sv3; uB6sv; quB6s; uquB6;**

**puquB; 0puqu; u0puq; qu0pu; Bqu0p; 7Bqu0; c7Bqu; vc7Bq; svc7B; vsvc7; ovsvc; aovsv; 9aovs; 19aov; 119ao; 0119a; 00119; 10011; 01001; 10100; 01010; a0101; 8a010; w8a01; 7w8a0; 17w8a; 017w8; 0017w; 10017; x1001; 7x100; 17x10; a17x1; 8a17x; 18a17; 118a1; f118a; 2f118; x2f11; 1x2f1; 01x2f; 101x2; a101x; 8a101; 18a10; a18a1; 0a18a; g0a18; cg0a1; 1cg0a; 41cg0; C41cg; NC41c; 4NC41; Y4NC4; TY4NC; 1100; 0110; 1011; A101; 8A10; 18A1; 118A; A118; 3A11; v3A1; sv3A; 6sv3; B6sv; uB6s; quB6; uquB; puqu; 0puq; u0pu; qu0p; Bqu0; 7Bqu; c7Bq; vc7B; svc7; vsvc; ovsv; aovs; 9aov; 19ao; 119a; 0119; 0011; 0100; 1010; 0101; a010; 8a01; w8a0; 7w8a; 17w8; 017w; 0017; 1001; x100; 7x10; 17x1; a17x; 8a17; 118a; f118; 2f11; x2f1; 1x2f; 01x2; 101x; a101; 8a10; 18a1; a18a; 0a18; g0a1; cg0a; 1cg0; 41cg; C41c; NC41; 4NC4; Y4NC; TY4N; LTY4**

## *References*


1. Ho, D. Notepad++ 2024. https://notepad-plus-plus.org
2. Opera 1995. https://www.opera.com
3. Eich, B.; Unnamed Others JavaScript 1995. https://ecma-international.org/publications-and-standards/standards/ecma-262/
4. Pihl, D. DNDregular.Otf. https://github.com/DigiMancer3D/SPX/blob/main/DND-Regular.otf
5. Full House Talk Model (Talk-Flow-Chart) 2023. https://raw.githubusercontent.com/DigiMancer3D/SPX/refs/heads/main/Visuals/Talk-flow-chart%20(e2ef-meme).png
6. Full House. Full House 1987. https://en.wikipedia.org/wiki/Full_House
7. Wave Data Hashed-API 2023. https://github.com/DigiMancer3D/WaveData-HAPI

8. Naor, M. & Dwork, C. Proof of Work. https://en.wikipedia.org/wiki/Proof_of_work
9. Pihl, D. Reflecetion Wheel. https://github.com/DigiMancer3D/SPX/blob/main/Visuals/Reflection_Wheel.png?raw=true
10. Pihl, D. SPX Encoding Infograph. https://github.com/DigiMancer3D/SPX/blob/main/Visuals/SuPosXt%20(encoding%20graph).png?raw=true
11. Pihl, D. Full House Talk Model. https://github.com/DigiMancer3D/SPX/blob/main/Visuals/Talk-flow-chart%20(e2ef-meme).png?raw=true
12. Pihl, D. Token Map. https://github.com/DigiMancer3D/SPX/blob/main/Visuals/Token_Map.png?raw=true
13. Pihl, D. SPX [v0] Code. https://github.com/DigiMancer3D/SPX/blob/main/OLD_FILES/Binary%20Parity%20Check%20test%20(old).html
14. Pihl, D. SPX [v0] Live Demo as Proof. https://gistpreview.github.io/?6b7e5258dc97fb7e535ddd558b6d6612
15. Pihl, D. SPX [v0] Paper. https://github.com/DigiMancer3D/SPX/blob/main/OLD_FILES/OLD_README
16. Pihl, D. SPX v1 Code. https://github.com/DigiMancer3D/SPX/blob/main/OLD_FILES/SuPosXt_encoding_Demo.html
17. Pihl D. SPX [v1] Live Demo as Proof. https://gistpreview.github.io/?3e603e4569f9617f4a418fda899160f8
18. Pihl, D. SPX [v1] Paper. https://github.com/DigiMancer3D/SPX/blob/main/OLD_FILES/2cd_OLD_README
19. Pihl, D. SPX [v2] Code. https://github.com/DigiMancer3D/SPX/blob/main/test-state.html
20. Pihl, D. SPX [v2] Live Demo as Proof. https://gistpreview.github.io/?2397fe2a9076e911a197621114875fe9/Live-SuPosXt-Demo.html
21. Pihl, D. Generic PDF style SPX [v2] Paper. https://github.com/DigiMancer3D/SPX