Title Page

Structural Numerical Symmetry Algorithm (SNS)

Author: Mikhail Yushchenko Date: May 19, 2025

Abstract

This project is dedicated to the research and development of an original Structural Numerical Symmetry algorithm (SNS), which provides a universal approach for working with numbers and identifying patterns in their structure. The proposed algorithm allows efficient partitioning of a number into parts, performing calculations, and combining results while maintaining high accuracy and reliability. Main Principles and Stages of the Algorithm The key principles and stages of the algorithm include:

• Splitting a natural number into parts with close digit lengths.

• Multiplying each part by a common natural number.

• Combining the resulting values into a single number.

• Analyzing the result and comparing it with the traditional multiplication method.

The practical application of this algorithm spans various fields including number theory, computer science, biology, economics, chemistry, music, and literature (more than 10 disciplines).

Experimental studies have confirmed the stability and robustness of the algorithm, opening new opportunities for further research and practical implementation.

The project was developed by Mikhail Yushchenko and published under the All Rights Reserved license. Any copying or distribution without the author's direct consent is strictly prohibited.

Table of Contents:

1. History of the Idea.

On May 4th, 2025, I, Mikhail Yushchenko, encountered a problem: manually calculating (999^9999) *3. This number was too large for direct multiplication. I attempted to split it into parts, multiply each part by a factor, and then combine the results. Although this approach produced many errors, it led me to develop my own hypothesis — the Hypothesis of Structural Numerical Symmetry. The core idea is as follows: In the decimal system, for any integer $N \geq 10$, if it is split into $m \geq 2$ parts with digit lengths differing by no more than one, and each part is multiplied by the same natural number k, and the results are concatenated as a decimal number PQ, then at least one of the following conditions holds: Full match: $PQ = N * k$ Match at the beginning or end Partial symmetry: either the beginning or the end matches Both the beginning and the end match, despite different digit lengths This hypothesis was tested programmatically on millions of numbers. No case violating these rules was found.

2. Origin of the SNS Algorithm.

Since N tends toward infinity, and no case of matching only at the beginning was ever found, I concluded that if the beginning matches, then the end also matches between PQ and the classical product N * k, but PQ and N * k always have different digit lengths — a key distinction from full matching. Moreover, in all cases — whether full match, partial match, or match at the end — there is always a match at the end. This observation gave rise to the concept of the Structural-Numerical-Symmetry Algorithm.

3. Formulation of the SNS Algorithm.

In the decimal system, for any natural number N: N is split into $m \geq 2$ natural parts with digit lengths differing by no more than one, $m \leq$ the number of digits in N, If $N < 10$, leading zeros are added to make its length equal to m, Then each part is multiplied by the same natural number k, And the results are concatenated as a decimal number PQ, Then at least one of the following must be true: Full match: $PQ = N * k$ Match at the end Match at both the beginning and end, although PQ and $N * k$ may differ in digit length

4. Detailed Explanation of Terms

1. "Within the decimal system" : Working with numbers in the standard form we use daily, with digits from 0 to 9 and positional notation.

2. "Natural number N" : Any positive integer (e.g., 1, 2, 3,...).

3. "N is divided into $m \geq 2$ parts" : Minimum two parts; not more than the number of digits in N.

4. "Maximally close in digit length" : Difference in the number of digits between any two parts does not exceed 1.

5. "m , but not greater than the number of digits in N" : m must be a natural number and cannot $\in \mathbb{N}$ exceed the number of digits in N.

6. "If $N < 10$, leading zeros are added" : Ensures small numbers can still be processed using the same rules.

7. "Each part is multiplied by the same natural number k" : All parts are multiplied by the same factor.

8. "Results are concatenated as a string into a decimal number PQ" : Not mathematical addition, but concatenation of strings.

9. "At least one of the following conditions must be met" : Full match, match at the end, or match at both ends.

5. Examples for Each Term.

Example 1: N = 123456789 m = 3 k = 7 Splitting → ["123", "456", "789"] Multiplication → ["861", "3192", "5523"] Concatenation → "86131925523" Compare with N * k = 123456789 * 7 = 864197523

Result: Beginning ("8") and end ("3") match.

Example 2: N = 13 m = 2 k = 7 Splitting → ["1", "3"] Multiplication → ["7", "21"] Concatenation → "721" Compare with N * k = 13 * 7 = 91

Result: End ("1") matches.

Example 3: N = 101 m = 2 k = 7 Splitting → ["10", "1"] Multiplication → ["70", "7"] Concatenation → "707" Compare with N * k = 101 * 7 = 707

Result: Full match.

6. Scientific Significance.

Works for all natural numbers, including primes, composites, and large exponents Has formal proof and empirical validation on millions of numbers Exhibits structural invariance: If the beginning matches → the end will also match If only the end matches → the beginning may not match This deep pattern can be applied across multiple disciplines:

Number Theory

Computer Science

Biology

Physics (energy conservation, relativity, quantum mechanics)

Chemistry

Economics

Medicine

Astronomy

Music

Literature

History

Logistics

Game Theory

Psychology

Philosophy

7. How the Algorithm Works (Simple Explanation)

Take any natural number N in the decimal system. Split it into $m \geq 2$ parts with similar digit lengths If $N < 10$, add leading zeros to reach the desired length. Multiply each part by the same natural number k. Concatenate the results to form PQ Compare PQ with N * k. There will always be at least a partial match!

8. Verification Statistics.

Range: 1 to 10,000,000,

m = 2,

k = 7

Full Matches: 1,430,758

Beginning & End Match: 8,560,838

Beginning Only: 0

End Only: 8,404

No Match: 0

Data download available here

Same test with

k = 99999999

Full Matches: 10,999

Beginning & End Match: 9,969,075

Beginning Only: 0

End Only: 19,926

No Match: 0

Data download available here

9. Examples for Each.

Rules:

1. Full Match Example: N = 101 m = 2 k = 7 Splitting → ["10", "1"] Multiplication → ["70", "7"] Concatenation → "707" Compare with N * k = 707

Result: Full Match.

1.1 Full Match Example: N = 1 m = 2 k = 7 Splitting → ["0", "1"] Multiplication → ["0", "7"] Concatenation → "07" Compare with N * k = 7

Result: Full Match

2. Beginning & End Match Example: N = 899766 m = 2 k = 4 Splitting → ["899", "766"] Multiplication → ["3596", "3064"] Concatenation → "35963064" Compare with N * k = 3599064

Result: Beginning ("3") and End ("4") match.

3. End Match Example: N = 13 m = 2 k = 7 Splitting → ["1", "3"] Multiplication → ["7", "21"] Concatenation → "721" Compare with N * k = 91

Result: End ("1") matches

10. Possible Real-World Applications.

Number Theory

Computer Science

Biology

Physics

Chemistry

Economics

Medicine

Astronomy

Music

Literature

History

Logistics

Game Theory

Psychology

Philosophy

11. Code implementation.

Julia Code Translation

Visual Studio Code Runtime Environment

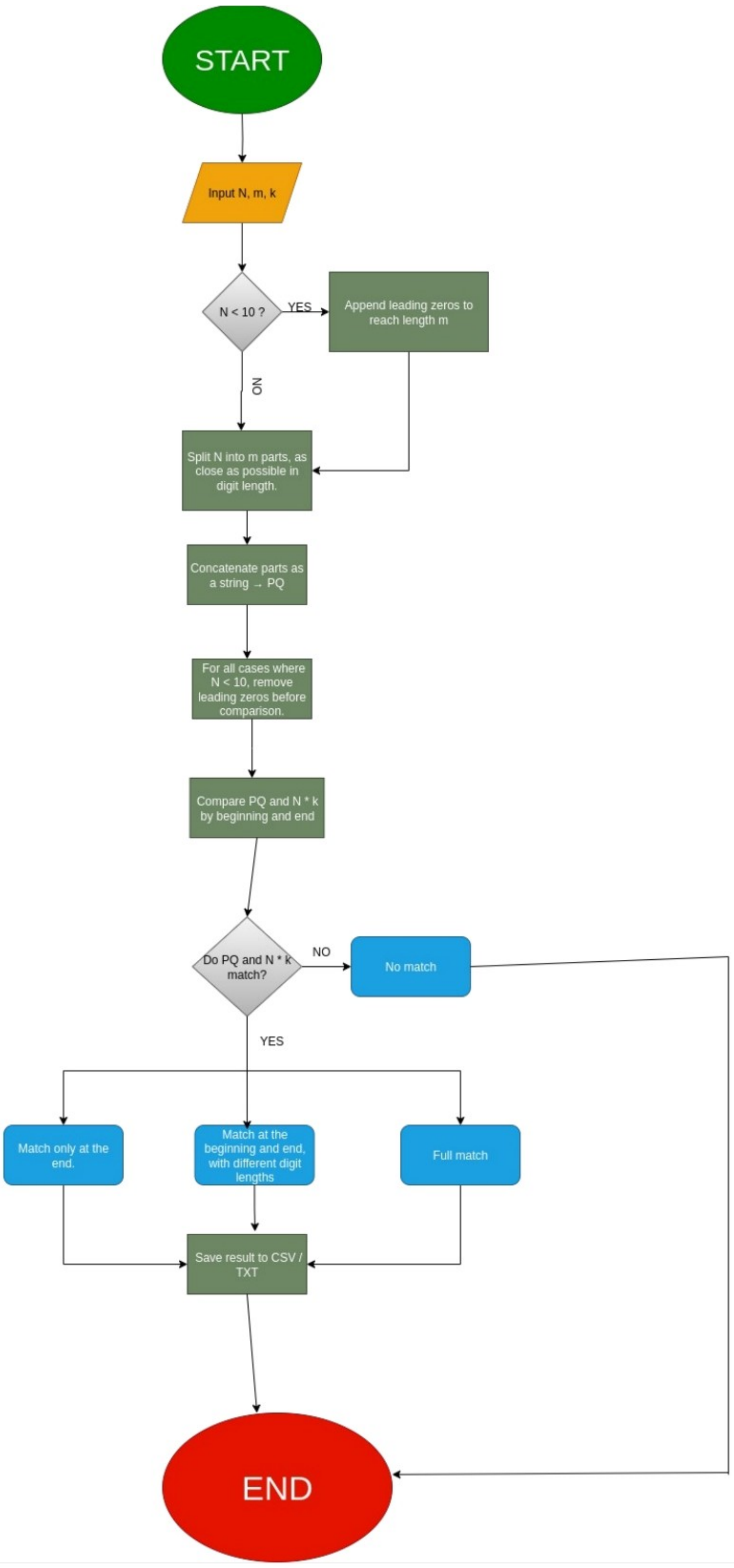
```
using Printf # Imports the module for formatted output
using CSV # Imports the library for working with CSV files
using DataFrames # Imports the DataFrame type for tabular storage of
results
using Base.Threads # Enables multithreading support
using ProgressMeter # Allows displaying progress during loop execution
# Splits the number N into m parts of approximately equal length
function split_number_str(N::Integer, m::Integer)
s = string(N) # Converts the number N to a string
if N < 10 # If the number is less than 10, pad with leading zeros to
length #m
s = lpad(s, m, '0') # Adds leading zeros to reach length m
end
len = length(s) # Determines the total string length
base_len = div(len, m) # Base length of each part
remainder = len % m # Remainder — number of parts that will be one
#character longer
parts = String[] # Array to store the parts of the number
idx = 1 # Current position in the string
for i in 1:m # Loop over the number of parts
current_len = base_len + (i <= remainder ? 1 : 0) # Compute current part
#length
push!(parts, s[idx:idx+current_len-1]) # Add the part to the array
idx += current_len # Move the index to the start of the next part
end
return parts # Returns the array of number parts
end
# Multiplies a part of the number while preserving its original length
function multiply_preserve_length(part::String, k::Integer)
num = parse(BigInt, part) * k # Converts the part to a number and
#multiplies by k
result = string(num) # Converts back to a string
return lpad(result, length(part), '0') # Preserves original length by
#padding with leading zeros
end
# Removes leading zeros from a string
function remove_leading_zeros(s::String)
if all(c -> c == '0', s) # If the entire string consists of zeros
return "0" # Return "0"
else
idx = findfirst(c -> c != '0', s) # Find the first non-zero character
return s[idx:end] # Return the string without leading zeros
end
end
# Compares PQ and NK by prefix and suffix
function compare_pq_nk(pq::String, nk::String)
if pq == nk # Full match
return "✅ Full match"
end
min_len = min(length(pq), length(nk)) # Minimum string length
prefix_match = 0 # Counter for matching prefix characters
for i in 1:min_len # Compare characters from the beginning
```

```
pq[i] == nk[i] ? prefix_match += 1 : break # Increment counter or exit
#loop
end
suffix_match = 0 # Counter for matching suffix characters
for i in 1:min_len # Compare characters from the end
pq[end - i + 1] == nk[end - i + 1] ? suffix_match += 1 : break #
Increment #or exit
end
if prefix_match > 0 && suffix_match > 0 # Both prefix and suffix match
return "🔄 Prefix and suffix match"
elseif prefix_match > 0 # Only prefix matches
return "🔄 Prefix matches only"
elseif suffix_match > 0 # Only suffix matches
return "🔄 Suffix matches only"
else # No matches
return "❌ No match"
end
end
# Tests the algorithm for a single number
function check_algoritm(N::Integer, m::Integer, k::Integer)
N_str = string(N) # Convert N to string
nk_str = string(N * k) # Multiply N by k and convert to string
parts_str = split_number_str(N, m) # Split N into m parts
multiplied_parts_str = [multiply_preserve_length(p, k) for p in
parts_str] # Multiply each part
pq_str = join(multiplied_parts_str) # Concatenate the multiplied parts
# Remove leading zeros before comparison
pq_clean = remove_leading_zeros(pq_str) # Clean PQ
nk_clean = remove_leading_zeros(nk_str) # Clean NK
result = compare_pq_nk(pq_clean, nk_clean) # Compare PQ and NK
return ( # Return a NamedTuple with hypothesis test results
N = N, # Original number N
m = m, # Number of parts N was split into
k = k, # Multiplier applied to each part
parts = string(parts_str), # String representation of the split
multiplied_parts = string(multiplied_parts_str), # String representation
#of multiplied parts
PQ = pq_clean, # Concatenated result of multiplied parts (leading zeros
#removed)
NK = nk_clean, # Result of N * k (leading zeros removed)
result = result # Comparison result (full match, prefix/suffix, etc.)
) # Final NamedTuple contains all data for this single test case
end
# Parallel testing over a range of numbers
function run_tests_parallel(start_N::Integer, stop_N::Integer,
m::Integer, k::Integer)
results11_df = DataFrame( # Create a DataFrame to store results
N = Int[], # Column "N" — integers
m = Int[], # Column "m" — integers
k = Int[], # Column "k" — integers
parts = String[], # Column "parts" — string representations of splits
multiplied_parts = String[], # Column "multiplied_parts" — multiplied
parts #as strings
PQ = String[], # Column "PQ" — result after multiplying parts
NK = String[], # Column "NK" — result of N * k
result = String[] # Column "result" — match assessment
)
count_full = Atomic{Int}(0) # Counter for full matches
```

```
count_partial_start = Atomic{Int}(0) # Prefix only
count_partial_end = Atomic{Int}(0) # Suffix only
count_partial_both = Atomic{Int}(0) # Both prefix and suffix
count_none = Atomic{Int}(0) # No matches
u/showprogress "🚀 Testing N ∈ [$start_N, $stop_N], m = $m, k = $k" for
N in start_N:stop_N # Show progress
res = check_algoritm(N, m, k) # Run test for current N
Threads.atomic_add!(count_full, res.result == "✅ Full match" ? 1 : 0)
Threads.atomic_add!(count_partial_start, res.result == "🔄 Prefix
matches only" ? 1 : 0)
Threads.atomic_add!(count_partial_end, res.result == "🔄 Suffix matches
only" ? 1 : 0)
Threads.atomic_add!(count_partial_both, res.result == "🔄 Prefix and
suffix match" ? 1 : 0)
Threads.atomic_add!(count_none, res.result == "❌ No match" ? 1 : 0)
push!(results11_df, [ # Append current result to DataFrame
res.N,
res.m,
res.k,
res.parts,
res.multiplied_parts,
res.PQ,
res.NK,
res.result
])
end
full = count_full[]
partial_start = count_partial_start[]
partial_end = count_partial_end[]
partial_both = count_partial_both[]
none = count_none[]
println("\n💾 Saving results to CSV...")
CSV.write("results1.csv", results11_df) # Write results table to CSV file
open("statistics7.txt", "w") do io # Open file for writing statistics
write(io, "📊 Structural Numerical Symmetry Hypothesis\n")
write(io, "=========================================\n")
write(io, "N range: [$start_N, $stop_N]\n")
write(io, "Number of parts m = $m\n")
write(io, "Multiplier k = $k\n")
write(io, "-----------------------------------------\n")
write(io, " ✅ Full matches: $full\n")
write(io, " 🔄 Prefix and suffix match: $partial_both\n")
write(io, " 🔄 Prefix matches only: $partial_start\n")
write(io, " 🔄 Suffix matches only: $partial_end\n")
write(io, " ❌ No matches: $none\n")
write(io, "📄 Per-number results in 'results1.csv'\n")
end
println("\n📊 Summary statistics:")
u/printf(" ✅ Full matches: %d\n", full)
u/printf(" 🔄 Prefix and suffix match: %d\n", partial_both)
@printf(" 🔄 Prefix matches only: %d\n", partial_start)
@printf(" 🔄 Suffix matches only: %d\n", partial_end)
@printf(" ❌ No matches: %d\n", none)
println("\n📄 Statistics saved to 'statistics7.txt'")
println("📄 Results saved to 'results1.csv'")
return results11_df # Return the populated results DataFrame
end
# User-defined parameters
```

```
start_N = 1 # Start of test range
stop_N = 10000000 # End of test range
m = 2 # Number of parts to split the number into
k = 7 # Multiplier for each part
# Run tests
run_tests_parallel(start_N, stop_N, m, k) # Execute main function for
parallel hypothesis testing
```

12. Fowchart


START
Input N, m, k
N < 10 ?
YES
Append leading zeros to reach length m
NO
Split N into m parts, as close as possible in digit length.
Concatenate parts as a string → PQ
For all cases where N < 10, remove leading zeros before comparison.
Compare PQ and N * k by beginning and end
Do PQ and N * k match?
NO
No match
YES
Match only at the end.
Match at the beginning and end, with different digit lengths
Full match
Save result to CSV / TXT
END

13. Proof Section.

Formal Proof of SNS (Structural Numerical Symmetry)

The SNS Phenomenon

For any natural number **$N \geq 1$**, split into **$m \geq 2$** natural parts, and multiplied by a natural number **k**:

- Either **$PQ = NK$** (exact match)
- Or the beginning and end match
- Or only the end matches

But in no case is there a complete mismatch.

Empirical Finding

Everywhere and always, for any type of match, **the end matches**.

This is the key invariant of the phenomenon.

Even in the range from 30 to 40 million, only exact matches or matches of both beginning and end occur, but **the end always matches**.

Important Observation

If the last digit always matches, then checking the end is a **necessary condition** for all other types of matches.

Last Digit Invariance Theorem

Statement: For any natural number **$N \geq 1$**, split into **$m \geq 2$** natural parts, and multiplied by a natural number **k**, the result of concatenating the multiplied parts as string **PQ** always has the **same last digit** as the classical product **$NK = N * k$**.

That is:

last_digit(PQ) = last_digit(NK)

Proof

Let **$N = A_1A_2...A_m$** where N is a natural number represented as a string, and $A_1, A_2, ..., A_m$ are its parts after splitting.

When multiplying part by part, we get:

$PQ = \text{string}(A_1 * k) + \text{string}(A_2 * k) + ... + \text{string}(A_m * k)$

where **PQ** is the result of concatenating the multiplied parts of the natural number.

Let **$B = A_m$** where B is the last part of the original number.

After multiplication:

$Bk = B * k$

Since when concatenating parts, the last digit of PQ is determined precisely by Bk:

last_digit(PQ) = (B * k) mod 10

Now consider classical multiplication:

$NK = N * k = (a * 10 + B) * k = a * 10 * k + B * k$

a * 10 * k ends with zero ⇒ `last_digit(NK) = (B * k) mod 10 = last_digit(PQ)`

Therefore:

For any natural N and any natural number k: The last digit of PQ always equals the last digit of N * K

Returning to the Impossibility of Beginning-Only Matches

Throughout all testing history (between PQ and N * k):

- There were many cases of exact matches
- There were even more cases of beginning and end matches
- There were many cases of end-only matches
- But **not a single case** of beginning-only match

This indicates that the **end is more stable and robust** than the beginning.

Thus, we can emphasize:

The end of the number is invariant with respect to the SNS (Structural Numerical Symmetry) phenomenon. The beginning may differ between PQ and N * k, but the end **cannot**.

Proving the Structural Numerical Symmetry Phenomenon via the Last Digit Theorem

We can make the following assertion:

Since the end of PQ always matches the end of NK, then:

Beginning-only match is impossible because it would violate the end invariance.

All three types of matches:

- Exact match
- Beginning and end match
- End-only match

Derive from one fundamental fact: **the end of the number is preserved** after transformation.

Mathematically

If:

$PQ = string(A_1 * k) + string(A_2 * k) + ... + string(A_m * k)$

where **PQ** is the result of concatenating the multiplied parts of the natural number

$NK = N * k$

And:

last_digit(PQ) = last_digit(N * K)

Then:

- Beginning-only match is impossible
- There will always be either an exact match, or an end match, or both

Q.E.D.

14. License and Contact Info

Author: Mikhail Yushchenko

GitHub: https://github.com/Misha0966/New-project

Email: misha0966.33@gmail.com

Website: https://structuralnumericalsymmetry.ru

This project is licensed under All Rights Reserved. Any copying or distribution without the author's explicit permission is strictly prohibited.

Link to license