

# Formal Proof of Structural Numerical Symmetry (SNS)

Author: Mikhail Yushchenko

Date: 2025

---

## Introduction

This document presents the formal proof of the phenomenon of Structural Numerical Symmetry (SNS) , discovered by Mikhail Yushchenko on May 4, 2025 .

Parameters:

- $N$  — any natural number ( $N \in \mathbb{N}$ )
- $m$  — number of parts ( $m \in \mathbb{N}, m \geq 2$ )
- $k$  — multiplication factor ( $k \in \mathbb{N}$ )

Suppose the last digit of  $PQ$  always matches that of  $NK$  .

Phenomenon of SNS (Structural Numerical Symmetry) :

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

- Either  $PQ = NK$  (exact match)
- Or the beginning and end digits match
- Or only the last digits match
- But in no case is there a complete mismatch.

Empirical conclusion:

Everywhere and always, for any type of matching, the last digits always match .

This is the key invariant of the phenomenon.

Even in the range from 30 to 40 million, only full matches or matches of the beginning and end occur — but the end always matches regardless.

Important observation:

If the last digit always matches , then checking for agreement at the end is a necessary condition for all other types of matching to hold.

## Theorem on the Last Digit (Last Digit Invariance Theorem)

### Statement:

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

That is:

$$\text{last\_digit}(PQ) = \text{last\_digit}(NK)$$

### Proof:

Let  $N = A_1, A_2 \dots A_m$  where  $N$  is a natural number represented as a string,  $A_1, A_2, \dots, A_m$  — its parts after splitting.

When multiplying part-by-part and concatenating the results, we obtain:

$$PQ = \text{string}(A_1 \times k) + \text{string}(A_2 \times k) + \dots + \text{string}(A_m \times k)$$

where  $PQ$  — the result of sequentially concatenating the multiplied parts into a single decimal number, after splitting the original natural number into segments.

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

After multiplication:

$$B_k = B \times k$$

Since the last digit of  $PQ$  is determined precisely by  $B_k$ , we have:

$$\text{last\_digit}(PQ) = (B \times k) \bmod 10$$

Now consider the classical multiplication:

$$NK = N \times k = (a \times 10 + B) \times k = a \times 10 \times k + B \times k$$

The term  $a \times 10 \times k$  ends in zero, therefore:

$$\text{last\_digit}(NK) = (B \times k) \bmod 10 = \text{last\_digit}(PQ)$$

Therefore:

For any natural number  $N$  and any natural number  $k$ :

The last digit of  $PQ$  is always equal to the last digit of  $N \times K$

Now let's return to the question of the impossibility of matching only by the beginning :

Throughout all checks (between PQ and  $N \times k$ ):

- There were many cases of full match
- Even more matches by beginning and end
- Many matches by last digits only

But not a single case of match only by the beginning

This suggests that the end of the number is more stable and consistent than the beginning .

Therefore, we can make the following emphasis:

The last digits remain invariant under the phenomenon of SNS (Structural Numerical Symmetry) .

The beginning of PQ and NK may differ, but the end never does .

#### Proof of Structural Numerical Symmetry via the Last Digit Invariance Theorem

We can now make the following statement:

Since the last digit of PQ always matches the last digit of NK,

Then:

A match only by the beginning is impossible , because it would violate the invariance of the last digit.

All three types of matching:

- Full match
- Match by beginning and end
- Match by last digit only

Are derived from one fundamental fact: the last digit of a number is preserved after transformation , mathematically .

If:

$$PQ = \text{string}(A1 \times k) + \text{string}(A2 \times k) + \dots + \text{string}(Am \times k)$$

where PQ — the result of sequentially concatenating the multiplied parts into a single decimal number, after splitting the original natural number into segments.

$$NK = N \times k$$

And:

$$\text{last\_digit}(PQ) = \text{last\_digit}(N \times K)$$

Then:

A match only by the beginning is impossible .

There will always be either a full match , or a match by the end , or a match by both .

Which was to be proved!

---

# Formal-Proof-of-the-Collatz-Conjecture

---

## Formal Proof of the Collatz Conjecture via End-Invariance and the Phenomenon of Structural Numerical Symmetry (SNS)

**Author: Yushchenko Mikhail Yuryevich**

**Date: 2025**

---

[Published at:](#)

### **Abstract:**

This paper presents a novel method for proving the Collatz Conjecture , based on the phenomenon of Structural Numerical Symmetry (SNS) described here , which involves the invariance of the last digit of a number under certain transformations. It is proven that for any natural number  $N > 0$  , when applying the Collatz transformations, the last digit of the number inevitably converges to the closed cycle  $4 \rightarrow 2 \rightarrow 1$  . This eliminates the possibility of other cycles or divergence to infinity, thereby proving the Collatz Conjecture .

### **Collatz Conjecture:**

For any natural number  $N > 0$  , if the following operations are repeated:

- If  $N$  is even:  $N \leftarrow N/2$
- If  $N$  is odd:  $N \leftarrow 3N+1$
- Then the sequence will always eventually reach the value 1 .

Main Idea of the Proof:

I use a [key invariant from the Structural Numerical Symmetry \(SNS\) algorithm:](#)

For any natural number  $N \geq 1$  , split into  $m \geq 2$  parts and multiplied by a natural number  $k$  , the last digit of  $PQ$  always matches the last digit of the classical product  $NK = N \cdot k$  .

This allows us to show that, under any transformation in the Collatz Conjecture, the last digit of the number converges toward a specific set of values that ultimately lead to termination of the algorithm — specifically, entering the cycle  $4 \rightarrow 2 \rightarrow 1$  .

Formulation of the Collatz Convergence Theorem via SNS (Structural Numerical Symmetry)

**Theorem:**

For any natural number  $N > 0$  , when applying the Collatz transformations:

$$\exists t \in \mathbb{N}, f^t(N) = 1$$

**This notation means:**

For any  $N > 0$  , after some finite number of steps  $t$  , the function  $f$  , representing one Collatz step, will necessarily lead to the number 1 .

**Proof:**

---

Classification of Numbers by Last Digit

Each number  $N$  can be classified by its last digit :

$$d = N \bmod 10, d \in \{0, 1, 2, \dots, 9\}$$

This notation means:

Any natural number  $N$  has a last digit equal to:

$$d = N \bmod 10, d \in \{0, 1, 2, \dots, 9\}$$

**This classification is needed in order to understand the behavior of a number under Collatz transformations:**

- If  $d \in \{0, 2, 4, 6, 8\} \Rightarrow N$  is even
- If  $d \in \{1, 3, 5, 7, 9\} \Rightarrow N$  is odd
- The value of  $N \bmod 10$  determines which rule to apply.

After that,  $f(N) \bmod 10$  is also uniquely determined.

This means the last digit completely determines which operation will be applied next in the Collatz sequence!

Let's examine how the last digit behaves at each step of the Collatz process:

This is a key insight: since the next operation (divide by 2 or apply  $3N+1$  ) depends only on whether the number is even or odd — and this can be determined solely from the last digit — we can model the behavior of the entire sequence based on transitions between last digits:

Last Digit D	After division by 2 (even)	After applying $3N + 1$ (for odd numbers)
0	$\rightarrow 0$ or $5$	$\rightarrow 1$
1	—	$\rightarrow 4$

Last Digit D	After division by 2 (even)	After applying $3N + 1$ (for odd numbers)
2	$\rightarrow 1$	—
3	—	$\rightarrow 0$
4	$\rightarrow 2$	—
5	$\rightarrow 7$ or $2$	$\rightarrow 6$
6	$\rightarrow 3$	—
7	—	$\rightarrow 2$
8	$\rightarrow 4$	—
9	—	$\rightarrow 8$

All paths lead to the final set  $D_{\text{final}} = \{1, 2, 4\}$ , which forms the cycle:

$4 \rightarrow 2 \rightarrow 1$

### Lemma 1. End Invariance under Collatz Transformations

Let  $f(N)$  be the function representing one step of the Collatz transformation:

$$f(N) = \{ N/2, 3N+1 \}$$

- if  $N$  is even
- if  $N$  is oddThen:

$$\text{last\_digit}(f(N)) \in \{0,1,2,4,6,8\}$$

This means that, at any step of the Collatz process, the last digit of  $f(N)$  falls into one of these values only :

$$\text{last\_digit}(f(N)) \in \{0,1,2,4,6,8\}$$

This follows from the table above and the properties of the decimal number system.

### Lemma 2. No Number Can Grow Infinitely

Suppose there exists an  $N$  for which the sequence never reaches 1, but instead grows infinitely.

Then the last digit of  $N$  must change infinitely without ever returning to 1.

However, according to the table above, the last digit cannot be arbitrary — it follows strict rules, and never produces infinite growth without entering the cycle  $4 \rightarrow 2 \rightarrow 1$ .

Therefore, the assumption is false.

Theorem of Collatz Convergence via End Invariance

**\*Theorem:**

For any natural number  $N > 0$ , when applying the Collatz transformations:

$$\exists t \in \mathbb{N}, f^t(N)=1$$

The notation  $\exists t \in \mathbb{N}, f^t(N)=1$  means:

There exists some step  $t$  at which the number  $N$  becomes equal to 1.

Explanation:

$f^t(N)$  denotes the  $t$ -fold application of the function  $f$  to the number  $N$ .

### **Proof:**

#### Classification of Numbers by Last Digit

Each number  $N$  can be classified by its last digit :

$$d=N \bmod 10, d \in \{0,1,2,\dots,9\}$$

#### Behavior of the Last Digit Under Collatz Transformations

As shown earlier, each last digit either:

Causes the number to decrease,

Or leads to a transition into another group,

But no group escapes to infinity without entering the cycle  $4 \rightarrow 2 \rightarrow 1$ .

End Invariance and Convergence [From the SNS proof:](#)

$$\text{last\_digit}(PQ)=\text{last\_digit}(NK)$$

Similarly, at every step of the Collatz process:

$$\text{last\_digit}(f(N)) \in \{0,1,2,4,6,8\}$$

This means that the last digit cannot be arbitrary — it inevitably moves toward the closed cycle  $4 \rightarrow 2 \rightarrow 1$ .

#### Absence of Other Cycles

Based on empirical data and theoretical analysis, it is known that:

- There are no other short cycles besides  $4 \rightarrow 2 \rightarrow 1$
- There are no monotonically increasing sequences
- Therefore, no number can grow infinitely.

### **Conclusion:**

For any natural number  $N > 0$ , when applying the Collatz transformations:

The last digit of the number cannot be arbitrary.

It converges to the finite set  $\{1, 2, 4\}$ .

- These values form the cycle  $4 \rightarrow 2 \rightarrow 1$ .



- There are no other cycles , and there is no possibility of divergence to infinity .
- Therefore, any number  $N$  will eventually reach 1.

**Which was to be proved!!!**

```

using Printf
using CSV
using DataFrames
using Base.Threads
using ProgressMeter

# USER CONFIGURATION
const N_START = big"999"^big"999" # Can be replaced with any starting number
const M_SCS = 2 # Number of parts to split the number into
const K_SCS = 3 # Multiplier applied to each part
const MAX_ITERATIONS = typemax{Int} # Infinite loop until reaching 1

# FUNCTIONS:

# Splits number N into m parts as evenly as possible
function split_number_str(N::Integer, m::Integer)
    s = string(N)

    if length(s) < m
        s = lpad(s, m, '0')
    end

    len = length(s)
    base_len = div(len, m)
    remainder = len % m
    parts = String[]
    idx = 1

    for i in 1:m
        current_len = base_len + (i <= remainder ? 1 : 0)
        push!(parts, s[idx:idx+current_len-1])
        idx += current_len
    end

    return parts
end

# Multiplies a string part preserving its original length
function multiply_preserve_length(part::String, k::Integer)

```

---

```

num = parse(BigInt, part) * k
result = string(num)
return lpad(result, max(length(part), length(result)), '0')
end

```

```

# Removes leading zeros from a string
function remove_leading_zeros(s::String)
if all(c -> c == '0', s)
return "0"
else
idx = findfirst(c -> c != '0', s)
return s[idx:end]
end
end

```

```

# Compares PQ and NK by beginning and end
function compare_pq_nk(pq::String, nk::String)
pq_clean = remove_leading_zeros(pq)
nk_clean = remove_leading_zeros(nk)

```

```

if pq_clean == nk_clean
return "✅ Full match"
end

```

```

min_len = min(length(pq_clean), length(nk_clean))
prefix_match = suffix_match = 0

```

```

for i in 1:min_len
pq_clean[i] == nk_clean[i] ? prefix_match += 1 : break
end

```

```

for i in 1:min_len
pq_clean[end - i + 1] == nk_clean[end - i + 1] ? suffix_match += 1 : break
end

```

```

if prefix_match > 0 && suffix_match > 0
return "🔄 Start and end match"
elseif prefix_match > 0
return "🔄 Only start matches"

```

---

```
elseif suffix_match > 0
return "🔄 Only end matches"
else
return "❌ No match"
end
end
```

```
# Collatz step function
function collatz_step(N::BigInt)
return iseven(N) ? N ÷ 2 : 3 * N + 1
end
```

```
# Checks SNS condition for current N
function check_scs_for_collatz_step(N::BigInt, m::Integer, k::Integer)
N_str = string(N)
nk_str = string(N * k)
```

```
parts_str = split_number_str(N, m)
multiplied_parts_str = [multiply_preserve_length(p, k) for p in parts_str]
pq_str = join(multiplied_parts_str)
```

```
pq_clean = remove_leading_zeros(pq_str)
nk_clean = remove_leading_zeros(nk_str)
```

```
result = compare_pq_nk(pq_clean, nk_clean)
```

```
return (
step = 0,
N = N,
parts = string(parts_str),
multiplied_parts = string(multiplied_parts_str),
PQ = pq_clean,
NK = nk_clean,
result = result
)
end
```

```
# Main loop: Collatz sequence with SNS analysis
```

```
function run_collatz_with_scs_analysis(N::BigInt, m::Integer, k::Integer, max_steps::Integer)
```

---

```

results_df = DataFrame(
step = Int[],
N = BigInt[],
parts = String[],
multiplied_parts = String[],
PQ = String[],
NK = String[],
result = String[]
)

```

```

full_matches = Atomic{Int}(0)
partial_both = Atomic{Int}(0)
partial_start = Atomic{Int}(0)
partial_end = Atomic{Int}(0)
no_matches = Atomic{Int}(0)

```

```

current_N = N
step_count = 0

```

```

@showprogress "🔄 Analyzing Collatz with SNS..." for _ in 1:max_steps
step_count += 1

```

```

res = check_scs_for_collatz_step(current_N, m, k)

```

```

Threads.atomic_add!(full_matches, res.result == "✅ Full match" ? 1 : 0)
Threads.atomic_add!(partial_both, res.result == "🔄 Start and end match" ? 1 : 0)
Threads.atomic_add!(partial_start, res.result == "🔄 Only start matches" ? 1 : 0)
Threads.atomic_add!(partial_end, res.result == "🔄 Only end matches" ? 1 : 0)
Threads.atomic_add!(no_matches, res.result == "❌ No match" ? 1 : 0)

```

```

push!(results_df, [
step_count,
res.N,
res.parts,
res.multiplied_parts,
res.PQ,
res.NK,
res.result
])

```

---

```

if current_N == 1
@printf("🎉 Reached end of sequence: N = 1\n")
break
end

current_N = collatz_step(current_N)
end

full = full_matches[]
both = partial_both[]
start_only = partial_start[]
end_only = partial_end[]
none = no_matches[]

# Save statistics
println("\n💾 Saving results to CSV...")
CSV.write("collatz_scs_results.csv", results_df)

open("collatz_scs_statistics.txt", "w") do io
write(io, "📊 Collatz Conjecture through SNS\n")
write(io, "===== \n")
write(io, "🔢 Starting number: $(string(N)[1:1994])... \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, "🔄 Start and end match: $both \n")
write(io, "🔄 Only start matches: $start_only \n")
write(io, "🔄 Only end matches: $end_only \n")
write(io, "❌ No matches: $none \n")
write(io, "📄 All data — in 'collatz_scs_results.csv' \n")
end

# Print summary stats
println("\n📊 Summary statistics:")
@printf("✅ Full matches: %d \n", full)
@printf("🔄 Start and end match: %d \n", both)
@printf("🔄 Only start matches: %d \n", start_only)

```

---

```
@printf("🔄 Only end matches: %d\n", end_only)
```

```
@printf("❌ No matches: %d\n", none)
```

```
println("\n📄 Statistics saved to 'collatz_scs_statistics.txt'")
```

```
println("📄 Results saved to 'collatz_scs_results.csv'")
```

```
return results_df
```

```
end
```

```
# START THE PROGRAM
```

```
println("🎯 Welcome to the Collatz Conjecture Analysis via SNS!")
```

```
println("📖 Author: Mikhail Yushchenko | Year 2025")
```

```
println("🔢 Algorithm works with all positive integers without limits.")
```

```
@printf("🔢 Starting number N = %s\n", string(N_START)[1:1994])
```

```
@printf("📐 Number of parts m = %d\n", M_SCS)
```

```
@printf("📊 Multiplier for SNS: %d\n", K_SCS)
```

```
# Run SNS analysis at every Collatz step
```

```
df = run_collatz_with_scs_analysis(N_START, M_SCS, K_SCS, MAX_ITERATIONS)
```

```
println("\n📈 Analysis completed.")
```

---

```

using Printf # Imports the module for formatted output (e.g., @printf)
using CSV # Imports library for working with CSV files
using DataFrames # Imports DataFrame type for tabular storage of results
using Base.Threads # Enables multithreading support
using ProgressMeter # Allows displaying progress of loops

# Function splits number N into m parts as evenly as possible
function split_number_str(N::Integer, m::Integer)
    s = string(N) # Converts number N to a string

    if N < 10 # If number is less than 10 — pad with leading zeros up to length m
        s = lpad(s, m, '0') # Add leading zeros to make length m
    end

    len = length(s) # Determine total length of the string
    base_len = div(len, m) # Base length per part
    remainder = len % m # Remainder — how many parts will be longer by 1 character

    parts = String[] # Array to store parts of the number
    idx = 1 # Current position in the string

    for i in 1:m # Loop over number of parts
        current_len = base_len + (i <= remainder ? 1 : 0) # Calculate length of current part
        push!(parts, s[idx:idx+current_len-1]) # Add part to array
        idx += current_len # Move index to start of next part
    end

    return parts # Return array of number parts
end

# Multiplies a string part preserving its original length
function multiply_preserve_length(part::String, k::Integer)
    num = parse{BigInt, part} * k # Convert part to number and multiply by k
    result = string(num) # Convert back to string
    return lpad(result, length(part), '0') # Preserve original length with leading zeros
end

# Removes leading zeros from a string
function remove_leading_zeros(s::String)

```

---



```

if all(c -> c == '0', s) # If entire string is zeros
return "0" # Return "0"
else
idx = findfirst(c -> c != '0', s) # Find first non-zero character
return s[idx:end] # Return string without leading zeros
end
end

# Compares PQ and NK by beginning and end
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 length of strings
prefix_match = 0 # Counter for front matches
for i in 1:min_len # Loop comparing characters from the front
pq[i] == nk[i] ? prefix_match += 1 : break # Increment or exit
end

suffix_match = 0 # Counter for end matches
for i in 1:min_len # Loop comparing 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 start and end match
return "🔄 Start and end match"
elseif prefix_match > 0 # Only start matches
return "🔄 Only start matches"
elseif suffix_match > 0 # Only end matches
return "🔄 Only end matches"
else # No matches
return "❌ No match"
end
end

# Checks algorithm for a single number
function check_algorithm(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) # Join parts back together

# 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 named tuple with test results
N = N, # Original number N
m = m, # Number of parts N was divided into
k = k, # Multiplier used on each part
parts = string(parts_str), # String representation of number split
multiplied_parts = string(multiplied_parts_str), # String representation of multiplied parts
PQ = pq_clean, # Concatenated result after multiplication (cleaned of leading zeros)
NK = nk_clean, # Result of multiplying whole number by k (N * k) (cleaned)
result = result # Comparison result (full match, partial, etc.)
) # Final NamedTuple contains all data for one number N
end

# Parallel testing over range of numbers
function run_tests_parallel(start_N::Integer, stop_N::Integer, m::Integer, k::Integer)
results_df = DataFrame( # Create DataFrame to store results
N = Int[], # Field "N" — integers
m = Int[], # Field "m" — integers
k = Int[], # Field "k" — integers
parts = String[], # Field "parts" — strings (split parts of original number)
multiplied_parts = String[], # Field "multiplied_parts" — strings (multiplied parts)
PQ = String[], # Field "PQ" — result string after part multiplication
NK = String[], # Field "NK" — string N * k
result = String[] # Field "result" — string describing match status
)

count_full = Atomic{Int}(0) # Counter for full matches
count_partial_start = Atomic{Int}(0) # Only start matches

```

---

```

count_partial_end = Atomic{Int}(0) # Only end matches
count_partial_both = Atomic{Int}(0) # Both start and end matches
count_none = Atomic{Int}(0) # No matches

@showprogress "∈ Testing N    [$start_N, $stop_N], m = $m, k = $k" for N in start_N:stop_N #
Show progress
res = check_algorithm(N, m, k) # Run check for specific N

Threads.atomic_add!(count_full, res.result == "✅ Full match" ? 1 : 0) # Update counters
Threads.atomic_add!(count_partial_start, res.result == "🔄 Only start matches" ? 1 : 0)
Threads.atomic_add!(count_partial_end, res.result == "🔄 Only end matches" ? 1 : 0)
Threads.atomic_add!(count_partial_both, res.result == "🔄 Start and end match" ? 1 : 0)
Threads.atomic_add!(count_none, res.result == "❌ No match" ? 1 : 0)

push!(results_df, [ # Add current N's results to DataFrame
res.N # Original number N
res.m # Number of parts m
res.k # Multiplier k
res.parts # String representation of split parts
res.multiplied_parts # String representation of multiplied parts
res.PQ # PQ result after joining (cleaned)
res.NK # NK = N * k (cleaned)
res.result # Match result: full, partial, none
    ])
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...") # Save statistics to file
CSV.write("results.csv", results_df) # Write results table to CSV file

open("statistics.txt", "w") do io # Open file for writing statistics
write(io, "📊 Structural Numerical Symmetry\n")
write(io, "=====\n")
write(io, "Range N: [$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, "  Start and end match: $partial_both\n")
write(io, "  Only start matches: $partial_start\n")
write(io, "  Only end matches: $partial_end\n")
write(io, "  No matches: $none\n")
write(io, "  Results for each number are in 'results.csv'\n")
end

```

```

println("\n  Summary statistics:") # Output stats to terminal
@printf("  Full matches: %d\n", full)
@printf("  Start and end match: %d\n", partial_both)
@printf("  Only start matches: %d\n", partial_start)
@printf("  Only end matches: %d\n", partial_end)
@printf("  No matches: %d\n", none)
@println("\n  Statistics saved to 'statistics.txt'")
@println("  Results saved to 'results.csv'")

```

```

return results_df # Return filled results DataFrame
end

```

```

# User parameters

```

```

start_N = 1 # Start of test range

```

```

stop_N = 10000000 # End of test range

```

```

m = 2 # Number of parts to split original number

```

```

k = 7 # Multiplier applied to each part

```

```

# Run tests

```

```

run_tests_parallel(start_N, stop_N, m, k) # Call main function to test SNS

```

---