Thermal-Aware Code Optimization

Monotonicity Properties of Thermal-Aware Channel Capacities

by

L. Tollenaar

Supervisor: J. H. Weber
EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial fulfillment of the Requirements
For the Bachelor of Applied Mathematics
June 20, 2025

Student number:	5876907	
Project Course Code:	AM3001	
Thesis committee:	Dr. ir. J. H. Weber,	TU Delft
	Dr. J. A. M. de Groot,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Layman's Summary

In modern communication systems, data is often being sent using binary codes (sequences consisting of zeros and ones) through some transmission unit. This process causes changes in temperature, which can sometimes disrupt signals or even lead to system failures. One way to prevent this is by controlling which input sequences are allowed, based on how much heat they generate. By carefully selecting these sequences, we can make sure the temperature always stays within safe limits.

The aim of this research project was to investigate the number of admissible sequences for communication systems, dependent on the amount of heating or cooling generated during transmission. We used a computer program to look for certain patterns between different situations. Subsequently, a mathematical proof was derived to support one of the observations. Overall, our results suggest that sequences starting at a temperature near the middle of the allowable temperature range often lead to the highest number of admissible sequences.

Although some of the claims remain unproven, they are supported by numerical evidence. If future work could establish this analytically, the results would provide insight into how communication systems can be designed to transmit data safely and efficiently.

Summary

This research investigates communication systems, where temperature constraints limit the transmission of data. One method to prevent overheating, thus avoiding data corruption or system failure, is to design the system such that the maximum allowable temperature is never exceeded. Accordingly, this study focuses on thermal-aware (TA) channels in the finite domain.

A TA-channel transmits data using binary input sequences of fixed length, where each bit contributes to the system's thermal state. The channel is characterized by four parameters: N, q, p, and n, representing the system's maximum allowable temperature, cooling gradient, heating gradient, and input length, respectively.

By identifying all binary sequences of length n that keep the system's temperature within the allowed range, we determine the number of admissible sequences for various parameter configurations. To make the TA-channel applicable for real-world use, sequences must also remain within the temperature limits when put into cascade. Sets of admissible sequences that satisfy this requirement are represented with C_a, where a denotes the initial temperature level.

Using transition matrices derived from the (N, q, p) TA-channel model, we computed the cardinality of each C_a given an input length n. The aim was to find the value of a that maximizes the number of valid sequences, i.e., the size of C_a. Existing results showed monotonicity in $|C_a|$ when the ratio of the heating and cooling gradient is integer [6]; we extended this to a different case and proved our claim analytically using an injective mapping technique. Furthermore, a series of conjectures for more general parameter configurations are proposed, based on patterns observed through numerical analysis.

Although an analytical proof of these conjectures has yet to be established, numerical results consistently support their validity. The results indicate that the largest sets of admissible TA-sequences occur when a is close to $N/2$ (where N is the maximum allowable temperature), particularly when either the heating or cooling gradient is small. This offers both theoretical insight and a foundation for future research on computing thermal-aware channel capacities.

Contents

1
Introduction

Temperature control is an important aspect of the design of modern communication systems, ranging from portable mobile devices to high-performance computing platforms. In these systems, data is transmitted through wiring in a chip. As chips become more advanced, the heat generated during transmission can significantly affect system stability and efficiency. This has led to the emergence of thermally-aware designs, which aim to effectively manage these concerns while minimizing power consumption [3, 4, 12].

A method to regulate temperature at the coding level is the use of thermal-aware communication channels. By constraining input sequences, the channel temperature can be kept within its allowed range, avoiding signal disruptions or hardware degradation. This is especially important in systems such as laser-based storage and tightly packed signal wires [7, 9].

To apply thermal-aware channels in practice, understanding their capacity, i.e., the maximum amount of information that can be transmitted under temperature constraints is essential. Previous research introduced methods to quantify this capacity in the infinite domain [5], showing connections between thermal-aware coding and earlier work on constrained channels, like those restricted by running digital sums [9] or charge balance [2]. Follow-up work further investigated code constructions that maintain temperature constraints over finite sequences, making them more practically applicable [6].

This report aims to generalize and build upon these capacity results for thermal-aware channels with fixed-length code sequences, as presented in [6]. In Chapter 2, we outline theoretical bounds in the asymptotic regime. Next, we review established finite-length results that form the foundation of our work in Chapter 3. Finally, in Chapter 4, we present the findings of this research project, including the numerical methods implemented and an analytical proof supporting the claims.

Before proceeding, we provide a mathematical formulation of the thermal-aware channel model, which forms the basis for the subsequent analysis. We also acknowledge that this report has been written with the assistance of AI (in particular, ChatGPT), which was used for formula formatting and writing style improvements. All theorems, conjectures, proofs and other arguments have been independently developed by the author, together with the support of the supervisor (dr. ir. J.H. Weber).

1.1. Introduction to Thermal-Aware Communication Systems

Communication systems use binary codes, input sequences consisting of zeros and ones, to transmit information. One method to implement these code is on-off keying, where a 1 represents an electrical pulse ("on") and a 0 results in no electrical activity ("off"). Each 1 increases the temperature of the transmission medium, while each 0 causes cooling down [6].

In such systems, the temperature must remain within safe limits. If it rises above a certain threshold, it may disrupt data transmission or damage the system. Thermal-aware codes prevent this by ensuring that sequences never exceed the system's maximum temperature. This property should hold not only for individual sequences but also for concatenated sequences, a requirement known as *cascadability* [6].

1.1.1. The Thermal-Aware Channel Model

The thermal-aware (TA) channel is a binary, noiseless channel subject to temperature constraints. It models how each bit (0 or 1) affects the system's thermal state during transmission. The model is defined by the following parameters:

- $T_{\min}$: minimum temperature of the channel,
- $T_{\max}$: maximum allowable temperature,
- t_1: heating gradient (temperature increase after a 1),
- t_0: cooling gradient (temperature decrease after a 0).

The channel temperature never drops below $T_{\min}$, which serves as a lower bound inherent to the system. Conversely, the channel does not have a strict upper temperature limit. Instead, $T_{\max}$ represents the threshold beyond which signal degradation may occur. By constraining the admissible input, we ensure that the channel temperature remains within the range $[T_{\min}, T_{\max}]$. Applying linear transformation, we can set $T_{\min} = 0$ and work within the range $[0, T]$. If t_1 and t_0 are rational, their ratio is written as $k = \frac{t_1}{t_0} = \frac{p}{q}$, where p and q are positive co-prime integers. This simplification makes it easier to analyze the channel using calculations with only integers [6].

1.1.2. Channel Dynamics and Admissibility

Each finite binary input sequence of length n is represented as:

$$\mathbf{x} = (x_1, x_2, \ldots, x_n). \tag{1.1}$$

Transmitting $\mathbf{x}$ results in a temperature sequence $\mathbf{s}_{t_0,t_1}(\mathbf{x})$, which tracks the system's temperature after sending each bit:

$$\mathbf{s}_{t_0,t_1}(\mathbf{x}) = (s_1, s_2, \ldots, s_n).$$

Starting at temperature 0, each s_i is recursively defined as:

$$s_i = \begin{cases} s_{i-1} + t_1, & \text{if } x_i = 1, \\ \max\{0, s_{i-1} - t_0\}, & \text{if } x_i = 0. \end{cases} \tag{1.2}$$

A sequence $\mathbf{x}$ is *admissible* if $s_i \le T$ for all i. The set of all admissible sequences for a given sequence length n is denoted $\mathscr{A}(T, t_0, t_1, n)$. A (T, t_0, t_1) TA-channel accepts only such sequences.

Example 1.1.1. Let $T = 4$, $t_1 = 2$, and $t_0 = 1$. For the sequence $\mathbf{x} = (1, 0, 1)$, the temperature changes as:

$$0 \to 2 \to 1 \to 3,$$

yielding $\mathbf{s}_{1,2}(\mathbf{x}) = (2, 1, 3)$. Since the temperature stays within $[0, 4]$, the sequence is admissible.

1.1.3. Integer Scaling and Graph Representation

To simplify the analysis, note that for any (T, t_0, t_1) TA-channel, it holds that

$$\mathscr{A}(T, t_0, t_1, n) = \mathscr{A}(\alpha T, \alpha t_0, \alpha t_1, n), \quad \forall \alpha > 0.$$

Hence, the channel is scale-invariant, and we can rescale it by setting $\alpha = \frac{q}{t_0}$,. This transforms a (T, t_0, t_1) TA-channel into an equivalent (N, q, p) TA-channel, where $N = \lfloor \frac{qT}{t_0} \rfloor$ [6].

The channel's behavior can be modeled by a directed graph with $N + 1$ states (from 0 to N). Each state represents a temperature level, and transitions are:

- from i to $i + p$ when transmitting a 1,
- from i to $i - q$, or to 0 if $i < q$, when transmitting a 0.

This graphical representation defines a transition matrix $D_{N,q,p}$, which encodes all valid temperature transitions. When analyzing the capacity of sets of admissible thermal-aware sequences, which will be discussed in the next chapters, this matrix turns out to be a useful tool.

Example 1.1.2. Consider $N = 6$, $q = 2$, and $p = 3$. The state graph has 7 nodes (0–6), illustrated in Figure 1.1.

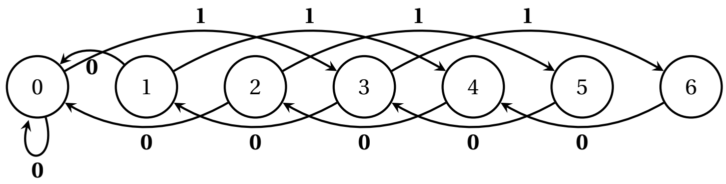


Figure 1.1: State graph of a TA-channel with parameters $N = 6$, $q = 2$, and $p = 3$.

The corresponding transition matrix is:

$$D_{6,2,3} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{1.3}$$

For $\mathbf{x} = (0,1,0,0,1)$, starting at 0 gives:

$$0 \to 0 \to 3 \to 1 \to 0 \to 3,$$

yielding $\mathbf{s}_{2,3}(\mathbf{x}) = (0,3,1,0,3)$, which stays within $[0,6]$. Thus, the sequence is admissible.

In contrast, $\mathbf{x} = (1,1,0,1,0)$ yields:

$$0 \to 3 \to 6 \to 4 \to 7 \to 5,$$

resulting in $\mathbf{s}_{2,3}(\mathbf{x}) = (3,6,4,7,5)$. Since the temperature exceeds $T = 6$, this sequence is inadmissible.

In Chapter 2, we briefly examine theoretical upper bounds for capacities of (N, q, p) TA-channels in the infinite case. This research, however, focuses on finite-length sequences, so we now introduce the additional concepts needed for the finite setting.

1.1.4. Thermal-Aware Channel Capacities for Finite Sequences

To support claims in the finite domain, discussed in Chapter 3 and 4, some extra concepts are required. Building on the definition of temperature sequences given in (1.2), we now introduce $\mathbf{v}_{q,p}(\mathbf{x}, u) = (v_1, v_2, \ldots, v_n)$, where $v_0 = u$ and:

$$v_i = \begin{cases} v_{i-1} + p, & \text{if } x_i = 1, \\ \max\{0, v_{i-1} - q\}, & \text{if } x_i = 0. \end{cases} \tag{1.4}$$

Each 1 increases the temperature by p, while each 0 decreases it by q, without dropping below zero. Note that $\mathbf{v}_{q,p}(\mathbf{x}, 0)$ corresponds exactly to $\mathbf{s}_{q,p}(\mathbf{x})$.

We define C_a as the set of all sequences $\mathbf{x}$ that:

- satisfy $v_i \leq N$ for all i,
- start at $v_0 = a$,
- end at $v_n \leq a$.

As a result of these criteria, sequences in C_a can be cascaded without exceeding the maximum temperature of the transmission system [6].

We also define the *weighted running digital sum* $\mathbf{t}_{q,p}(\mathbf{x}) = (t_1, t_2, \ldots, t_n)$, where:

$$t_i = \sum_{j=1}^{i} \big(p x_j + q(x_j - 1)\big) = -qi + (p+q) \sum_{j=1}^{i} x_j. \tag{1.5}$$

Unlike $\mathbf{v}_{q,p}(\mathbf{x}, u)$, this sequence allows negative values and will play a useful role in analyzing C_a in Chapters 3 and 4.

2

Theoretical Limits of Thermal-Aware Channel Capacities

This chapter investigates the theoretical limits of thermal-aware (TA) channel capacities. These limits are derived for infinite input sequences and serve as upper bounds for the finite-length TA-channels explored in Chapters 3 and 4.

The capacity of a (N, q, p) TA-channel is defined as the maximum achievable asymptotic rate:

$$\mathrm{cap}_{\mathrm{TA}}(N, q, p) = \limsup_{n\to\infty} \frac{\log_2 |\mathscr{A}(N, q, p, n)|}{n}. \tag{2.1}$$

Without thermal constraints, all binary sequences of length n would be admissible, resulting in $|\mathscr{A}(N, q, p, n)| = 2^n$, and thus yielding a channel capacity of 1. For TA-channels, however, admissibility depends on the parameters N, q, and p, therefore restricting the number of valid sequences. The capacity in (2.1) represents the fraction of binary sequences that remain valid under these constraints and is useful for efficient thermal-aware coding.

2.1. Computing TA-Channel Capacities via Transition Matrices

As established in research on constrained systems [8, 10, 11], the capacity of a finite-state constrained channel can be determined using its corresponding graph representation. For TA-channels, this graph consists of $N+1$ nodes, representing discrete temperature states from 0 to N, as introduced in Chapter 1.1.3. The edges describe valid transitions based on heating and cooling gradients, subject to the temperature limits.

The transitions are encoded in the $(N+1)\times(N+1)$ matrix $D_{N,q,p}$, where non-zero entries indicate valid transitions between states after transmitting either a 0 or a 1. The asymptotic capacity is the base-2 logarithm of the largest real eigenvalue of $D_{N,q,p}$ [11]:

$$\mathrm{cap}_{\mathrm{TA}}(N, q, p) = \log_2 \lambda, \tag{2.2}$$

where λ is the dominant real root of the characteristic polynomial:

$$\Gamma_{N,q,p}(z) = \det[zI - D_{N,q,p}]. \tag{2.3}$$

This gives an exact expression for the asymptotic capacity for any (N, q, p) TA-channel. However, for large N, computing the determinant and identifying the dominant root becomes computationally intensive and sometimes impossible.

2.2. Upper Bounds on TA-Channel Capacities

Before further investigation, we rescale the (T, t_0, t_1) TA-channel to an $(M, 1, k)$ TA-channel by setting $\alpha = \frac{1}{t_0}$. This rescaling reduces the number of free parameters, thereby simplifying the analysis in the asymptotic domain. Note that we also saw this scaling in Chapter 1.1.3, where we used $\alpha = \frac{q}{t_0}$ to obtain the (N, q, p) TA-channel.

Previous results show that $\text{cap}_{\text{TA}}(M,1,k)$ increases with M, since larger temperature limits allow more admissible sequences. Conversely, the capacity decreases as k increases, since stronger heating relative to cooling tightens constraints [6]. This leads to the following result:

Theorem 2.2.1 (Adapted from [6])**.** *For any $k \le 1$ and $M \ge k$,*

$$\text{cap}_{TA}(M,1,k) > \log_2\left(2\cos\left(\frac{\pi}{\lfloor M/k \rfloor + 2}\right)\right).$$

This result implies that for any $k \le 1$, the capacity $\text{cap}_{\text{TA}}(M,1,k)$ approaches 1 as M becomes large. For an in-depth argument of this proof, we refer to [6].

Furthermore, since $\text{cap}_{\text{TA}}(M,1,k)$ decreases in k, it leads to the following approximations:

Proposition 2.2.2 (From [6])**.** *Let $\text{cap}_{TA}(M,1,k)$ denote the capacity of the TA-channel with parameters $(M,1,k)$, where $k = p/q$.*

Let $p', q', p'', q'' \in \mathbb{Z}_{>0}$ satisfy

$$\frac{p'}{q'} \le \frac{p}{q} \le \frac{p''}{q''}, \quad \textit{with} \quad \frac{p'}{q'} \approx \frac{p}{q} \approx \frac{p''}{q''},$$

and both q' and q'' much smaller than q. Then:

$$\text{cap}_{TA}(M,1,p''/q'') \le \text{cap}_{TA}(M,1,k) \le \text{cap}_{TA}(M,1,p'/q').$$

In the special case where $q = p = 1$, i.e., equal heating and cooling gradients, an explicit formula exists:

Theorem 2.2.3 (From [6])**.** *The capacity of the $(N,1,1)$ TA-channel is*

$$\text{cap}_{TA}(N,1,1) = \log_2\left(2\cos\left(\frac{\pi}{2N+3}\right)\right).$$

In all other cases where $q \ne p$, no explicit formula for computing the capacity of the TA-channel is currently known. In the following chapter, we turn our attention to the finite-length domain by analyzing the subsets $C_a \subseteq \mathscr{A}(N,q,p,n)$. These consist of TA-sequences of fixed length n that begin and end in specified temperature states. Again using the properties of the transition matrix, we will present several results regarding the maximal size of these subsets.

3

Thermal-Aware Channel Capacities of Finite-Length Sequences

This chapter focuses on the analysis of thermal-aware (TA) channel capacities for input sequences of fixed length. In practice, communication systems transmit sequences of finite length, making it important to understand how temperature constraints impact the overall capacity of communication systems in the finite domain.

To enable this analysis, we make use of the transition matrices associated with TA channels, which capture all permissible state transitions. Proofs of all theorems stated here can be found in Appendix A.

3.1. Properties of Transition Matrices

As established in Chapter 1.1.3, each (N, q, p) TA-channel induces a transition matrix $D_{N,q,p}$ of size $(N+1) \times (N+1)$; see Example 1.3. From linear algebra and graph theory, it is known that the number of walks of length n between states in a finite-state graph equals the corresponding entry in the matrix $D^{(n)}_{N,q,p}$:

Theorem 3.1.1 (From [1])**.** *Let $G = (V, E)$ be a finite graph with adjacency matrix A. Then, for all integers $n \geq 0$, the entry $(A^n)_{ij}$ equals the number of walks of length n from vertex i to vertex j.*

Using Theorem 3.1.1, we can compute the sizes of the sets C_a. For each $i \in \{0, 1, \ldots, N\}$, $|C_i|$ equals the number of walks of length n starting at state i and ending at any state $j \leq i$. This follows directly from the fact that, to ensure sequences can be cascaded under temperature constraints, valid sequences must end at or below their starting state [6].

Consequently, the size of C_i can be expressed as:

$$|C_i| = \sum_{j=0}^{i} (D^{(n)}_{N,q,p})_{ij}. \tag{3.1}$$

Example 3.1.2. Let $N = 4$, $q = 2$, $p = 3$, and $n = 6$. Then:

$$D_{4,2,3} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \to D^{(6)}_{4,2,3} = \begin{bmatrix} 8 & 3 & 1 & 5 & 2 \\ 7 & 3 & 1 & 4 & 2 \\ 5 & 2 & 1 & 3 & 1 \\ 4 & 2 & 1 & 3 & 1 \\ 3 & 1 & 1 & 2 & 1 \end{bmatrix}$$

Using (3.1):

$$\begin{aligned} |C_0| &= 8, \\ |C_1| &= 7 + 3 = 10, \\ |C_2| &= 5 + 2 + 1 = 8, \\ |C_3| &= 4 + 2 + 1 + 3 = 10, \\ |C_4| &= 3 + 1 + 1 + 2 + 1 = 8. \end{aligned}$$

We can now directly compute the value of a that maximizes $|C_a|$ for a given channel configuration using only the transition matrix. The next section presents additional theoretical properties that further narrow the search space for this optimal a.

3.2. Properties of TA-Channel Capacities in the Finite Domain

Recall that $C_a \subseteq \mathscr{A}(N,q,p,n)$ denotes the subset of admissible sequences starting at temperature level a and ending at or below a. These sets form the foundation for constructing thermally-aware code sequences of fixed length.

The following result from [6] highlights the practical importance of C_a:

Theorem 3.2.1 (From [6]). *For any $0 \le a \le N$, the code $C_a \subseteq \mathscr{A}(N,q,p,n)$ can be used to encode (and decode) messages from a message set of size at most $|C_a|$ into (from) binary codewords of length n. Cascading such codewords results in a valid (N,q,p) thermal-aware sequence.*

As our goal is to find the largest sets of binary codes of fixed length n that are thermally admissible both individually and when concatenated, identifying the value of a for which $|C_a|$ is maximized is very informative. Since this value yields the largest possible set of thermal-aware sequences of length n for a given channel configuration, it directly gives us the optimal set of admissible sequences of fixed length for a specific (N,q,p) TA channel.

3.2.1. Symmetry and Monotonicity of C_a

Several structural properties of C_a have been established in [6] and serve as a basis for our further analysis.

First, we introduce sequence reversal. For any sequence $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, define:

$$\mathbf{x}^R = (x_n, x_{n-1}, \ldots, x_1).$$

Theorem 3.2.2 (From [6]). *For all $a \in \{0,1,\ldots,N\}$,*

$$\mathbf{x} \in C_a \iff \mathbf{x}^R \in C_{N-a}.$$

Corollary 3.2.3 (From [6]). *For all $a \in \{0,1,\ldots,N\}$,*

$$|C_a| = |C_{N-a}|.$$

The symmetry of C_a allows us to limit our search for the maximizing a to values up to $N/2$.

Example 3.2.4. Let $N = 5$, $q = 1$, $p = 3$, and $n = 6$. Then:

$$D^{(6)}_{5,1,3} = \begin{bmatrix} 4 & 1 & 4 & 5 & 2 & 1 \\ 4 & 1 & 1 & 7 & 2 & 1 \\ 4 & 1 & 1 & 1 & 6 & 1 \\ 1 & 3 & 1 & 1 & 0 & 3 \\ 1 & 0 & 3 & 1 & 0 & 0 \\ 1 & 0 & 0 & 3 & 0 & 0 \end{bmatrix}.$$

Computing $|C_a|$ yields:

$$|C_0| = 4, \quad |C_1| = 5, \quad |C_2| = 6, \quad |C_3| = 6, \quad |C_4| = 5, \quad |C_5| = 4.$$

The maximum occurs at $a = 2$, where $|C_2| = 6$. The corresponding code rate is

$$\frac{\log_2(6)}{6} \approx 0.43,$$

The maximum asymptotic capacity of the $(5,1,3)$ TA-channel, as defined in Chapter 2, is approximately 0.58. The resulting code rate therefore achieves about 74% of this theoretical limit.

Example 3.2.4 illustrates not only symmetry, but also monotonicity of $|C_a|$. The latter turns out to hold under the condition that $q = 1$, i.e., when the ratio of the heating to cooling gradient is an integer:

Theorem 3.2.5 (From [6]). *If $q = 1$ and $0 \leq a \leq \frac{N}{2} - 1$, then*

$$|C_a| \leq |C_{a+1}|.$$

However, prior work also shows that this monotonicity does not hold in general. For arbitrary values of q and p, the size of C_a may not increase with a. This is illustrated in Example 3.2.6 below and can also be deduced from Example 3.1.2.

Example 3.2.6. Let $N = 6$, $q = 4$, $p = 5$, and $n = 8$. Then:

$$D^{(8)}_{6,4,5} = \begin{bmatrix} 19 & 8 & 3 & 0 & 0 & 12 & 5 \\ 17 & 7 & 3 & 0 & 0 & 11 & 4 \\ 12 & 5 & 2 & 0 & 0 & 8 & 3 \\ 12 & 5 & 2 & 0 & 0 & 8 & 3 \\ 12 & 5 & 2 & 0 & 0 & 8 & 3 \\ 11 & 4 & 2 & 0 & 0 & 7 & 3 \\ 8 & 3 & 1 & 0 & 0 & 5 & 2 \end{bmatrix}$$

The corresponding $|C_a|$ values are:

$$|C_0| = 19, \quad |C_1| = 24, \quad |C_2| = 19, \quad |C_3| = 19, \quad |C_4| = 19, \quad |C_5| = 24, \quad |C_6| = 19.$$

It follows that $|C_1| > |C_2|$, showing that this monotonicity does not hold in general.

Since the general case remains open for investigation, it has led to this research project. In the next chapter, we present our numerical methods to explore different parameter configurations, along with an analytical proof that supports one of the findings.

4

Main Results: Thermal-Aware Channel Capacity Properties in the Finite Domain

Building on the theoretical bounds for thermal-aware (TA) channels established in Chapter 2, and the finite-length channel capacity properties studied in Chapter 3, this chapter presents the main results of this research project. The goal was to investigate the capacity of (N, q, p) TA-channels for input sequences of fixed length n. As in Chapter 3.2.1, this was done by determining the values of a that maximize $|C_a|$ (see Chapter 1.1.4 for definitions), and identifying potential relations between the optimal a and the parameters N, q, p, and n.

The investigation started with extensive numerical analysis. Various (N, q, p) TA-channel configurations were generated, and their corresponding transition matrices were computed to determine the capacities C_a. The implementation was performed in Python using the online platform *Kaggle*. We note that the development of the code was done with the assistance of AI. Details of the numerical implementation can be found in Appendix C. For each configuration, the optimal capacity $|C_a|$ and its corresponding index a were determined, allowing for manual inspection of potential patterns between different parameter configurations.

4.1. Numerical Analysis of TA-Channel Capacities

We started the numerical analysis with a broad approach, examining general parameter relations such as $p > q$, $q > p$, $p - q = 1$, and $p + q = N$. Subsequently, we studied the channel capacities (by determining $|C_a|$ for all values of a) associated with transition matrices that satisfied these conditions. In these cases, each of the parameters (q, p, N, and n) influenced the determination of the optimal a. That is, no single parameter could be considered independent. If any parameter had been irrelevant, it could have been excluded from the analysis, simplifying the derivation of an explicit expression for the maximal capacity of C_a.

As a result, this investigation suggested that the monotonicity property discussed in Chapter 3.2 does not hold for general values of the parameters q, p, N, and n. Instead, the wide range of parameter combinations made it difficult to identify clear patterns through manual inspection.

To address this, we decided to focus on more specific cases. For instance, we fixed $q = 2$ and then studied how $|C_a|$ behaves as p, N, and n vary. For some specific configurations, numerical analysis did suggest monotonicity in the cardinality of C_a. These cases will be discussed in the following section.

4.2. Monotonicity Properties of TA-Channel Capacities in the Finite Domain

Theorem 3.2.5 proves monotonicity for $q = 1$. Numerical analysis indicates similar behavior for $p = 1$. This result is formalized as follows:

4.2.1. Monotonicity for the Case $p = 1$

Theorem 4.2.1. *If* $p = 1$ *and* $0 \leq a \leq N/2 - 1$, *then*

$$|C_a| \leq |C_{a+1}|.$$

Proof. **Overview.** This proof is established by constructing an injective mapping g from C_a to C_{a+1}. Let $\mathbf{x}$ be any sequence in C_a, noting that $v_0 = a$. Define:

$$\mathbf{v}_{q,p}(\mathbf{x}, a) = (v_1, v_2, \ldots, v_n), \quad \mathbf{t}_{q,p}(\mathbf{x}) = (t_1, t_2, \ldots, t_n), \quad \mathbf{t}_{q,p}(\mathbf{x}^R) = (t'_1, t'_2, \ldots, t'_n).$$

1. Definition $h_{\mathbf{x}}$.
Define $h_{\mathbf{x}}$ as follows:

$$h_{\mathbf{x}} = \min\{i : \mathbf{v}_{q,p}(\mathbf{x}, a)_i = a+1\}.$$

If no such index exists, i.e., if $v_i \leq a$ for all $1 \leq i \leq n$, then set $h = n$.

2: Definition of the mapping g.
We decompose $\mathbf{x}$ as

$$\mathbf{x} = (\mathbf{u}, \mathbf{w}),$$

with $\mathbf{u}$ of length h and $\mathbf{w}$ of length $n-h$, where $h = h_{\mathbf{x}}$. Define the sequence $\mathbf{y}$:

$$\mathbf{y} = g(\mathbf{x}) = (\mathbf{w}, \mathbf{u}^R).$$

Let

$$\mathbf{v}_{q,p}(\mathbf{y}, a+1) = (v''_1, v''_2, \ldots, v''_n), \quad \mathbf{t}_{q,p}(\mathbf{y}) = (t''_1, t''_2, \ldots, t''_n).$$

3. Proof that $\mathbf{y} \in C_{a+1}$.
We will now show that $\mathbf{y} \in C_{a+1}$ by verifying:

a) $v''_i \leq N$ for all i,

b) $v''_n \leq a+1$.

Furthermore, we prove that $\mathbf{y}$ is uniquely determined by $\mathbf{x}$, i.e., g is injective. This implies $|C_a| \leq |C_{a+1}|$ for all $0 \leq a \leq N/2 - 1$, which completes the proof.

(a)
First, consider the case $h < n$, so there exists an index h such that $v_h = a+1$. Since the function g maps $\mathbf{x} \in C_a$ to $\mathbf{y} \in C_{a+1}$, $v''_0 = a+1$. By the definition of h and the construction of $\mathbf{y}$, it follows that:

$$v''_i = v_{i+h} \tag{4.1}$$

for all $1 \leq i \leq n-h$. Moreover, since $\mathbf{x} \in C_a$, we know $v_j \leq N$ for all j, so:

$$v''_i \leq N$$

for all $1 \leq i \leq n-h$.
Equation (4.1) also implies:

$$v''_{n-h} = v_n \leq a. \tag{4.2}$$

To establish that $v''_i \leq N$ for all $n-h < i \leq n$, we proceed by contradiction. Suppose there exists a $\beta \in \{n-h+1, \ldots, n\}$ such that $v''_\beta = N+1$. Define $\alpha \in \{n-h+1, \ldots, \beta-1\}$ by:

$$\alpha = \left\{i : v''_i = a+1 \text{ and } v''_j > a+1 \text{ for all } i < j < \beta\right\}. \tag{4.3}$$

Note that from (4.2), we know such an α exists. Then:

$$t''_\beta - t''_\alpha = v''_\beta - v''_\alpha = N+1-(a+1) \geq a+2,$$

since $a \leq N/2 - 1$ implies $N \geq 2(a+1)$, and hence $N+1-(a+1) \geq a+2$.

Observe that for $n-h < i \leq n$, $t''_i = t'_i$ by definition of $\mathbf{y}$. Moreover, it holds that $t_j - t_i = t'_{n-i} - t'_{n-j}$ for all $1 \leq i \leq j \leq n$. Therefore we obtain:

$$v_{n-\alpha} \geq v_{n-\beta} + t_{n-\alpha} - t_{n-\beta} = v_{n-\beta} + t'_\beta - t'_\alpha = v_{n-\alpha} + t''_\beta - t''_\alpha \geq 0 + a + 2.$$

However, by the definition of h and since $p = 1$, it must be that $v_{n-\alpha} < v_h = a+1$, which yields a contradiction. Thus:

$$v''_i \leq N$$

for all $n - h < i \leq n$.

If $h = n$, a similar argument applies. Again, assume there exists an index $\beta \in \{1,\dots,n\}$ such that $v''_\beta = N+1$. Define $\alpha \in \{0,\dots,\beta-1\}$ as in (4.3). Note that by definition of $\mathbf{y}$, at least one valid index α exists. Again, we obtain the inequality that

$$v_{n-\alpha} \geq v_{n-\beta} + t_{n-\alpha} - t_{n-\beta} = v_{n-\alpha} + t'_\beta - t'_\alpha = v_{n-\alpha} + t''_\beta - t''_\alpha \geq a+2$$

However, since $h = n$, in this case no j exists with $v_j \geq a+1$. Therefore, $v_{n-\alpha} \geq a+2$ leads to a contradiction. Hence:

$$v''_i \leq N$$

for all $1 \leq i \leq n$.

(b)
Now consider the upper bound on v''_n. We begin with the case $h < n$. Suppose, for contradiction, that $v''_n \geq a+2$. Define $\alpha \in \{n-h+1,\dots,\beta-1\}$ as in (4.3) with $\beta = n$. Then:

$$t''_n - t''_\alpha = v''_n - v''_\alpha \geq a+2-(a+1) = 1.$$

Since $t_{n-\alpha} - t_0 = t'_n - t'_\alpha$ (where $t_0 = 0$) and $t''_n - t''_\alpha = t'_n - t'_\alpha$, it follows that:

$$v_{n-\alpha} \geq a + t_{n-\alpha} - t_0 = a + t'_n - t'_\alpha = a + t''_n - t''_\alpha \geq a+1.$$

However, $v_{n-\alpha} < v_h = a+1$, yielding a contradiction. Hence:

$$v''_n \leq a+1.$$

Now consider the case $h = n$. Again, assume for contradiction that $v''_n \geq a+2$. Let $\beta = n$ and define $\alpha \in \{0,\dots,\beta-1\}$ as in (4.3). Then it follows, again, that:

$$t''_n - t''_\alpha = v''_n - v''_\alpha \geq a+2-(a+1) = 1.$$

Once more, we find:

$$v_{n-\alpha} \geq a + t_{n-\alpha} - t_0 = a + t'_n - t'_\alpha = a + t''_n - t''_\alpha \geq a+1.$$

However, in this case $h = n$, so $v_j \leq a$ for all j. Therefore, $v_{n-\alpha} \geq a+1$ leads to a contradiction. We conclude that:

$$v''_n \leq a+1.$$

4. Injectivity of mapping g.
Lastly, we need to show that the mapping g is injective. This is done by noting that $h = h_{\mathbf{x}} = h_{\mathbf{y}^R}$. Thus, we can obtain h from $\mathbf{y}$ and invert the mapping:

$$g^{-1}(\mathbf{y}) = g^{-1}(\mathbf{w},\mathbf{u}^R) = ((\mathbf{u}^R)^R,\mathbf{w}) = (\mathbf{u},\mathbf{w}) = \mathbf{x}.$$

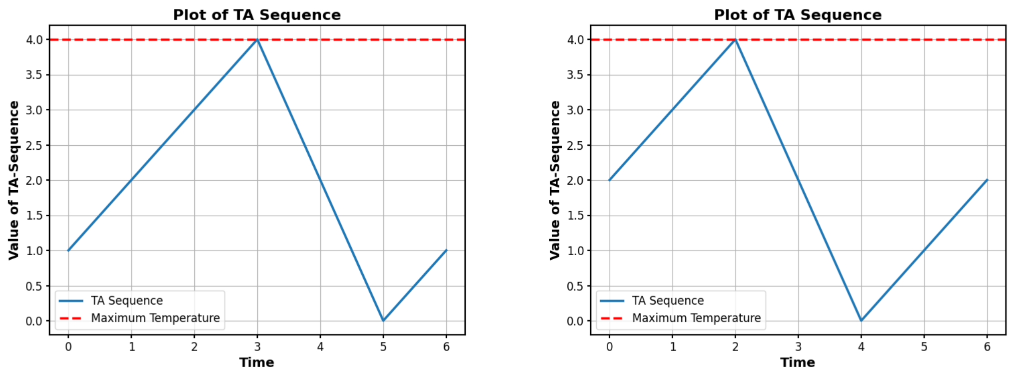
Hence, g is injective, completing the proof.

□

Example 4.2.2. To illustrate the mapping g defined in the proof of Theorem 4.2.1, consider the parameters $N = 4$, $q = 2$, $p = 1$, and $n = 6$. Let $\mathbf{x}_1 = (1,1,1,0,0,1) \in C_1$. Then: $\mathbf{v}_{2,1}(\mathbf{x}_1,1) = (2,3,4,2,0,1)$. Here, for $h_{\mathbf{x}_1} = 1$, we have $v_1 = a+1 = 2$, and the mapping gives:

$$\mathbf{y}_1 = g(\mathbf{x}_1) = (1,1,0,0,1,1) \quad and \quad \mathbf{v}_{2,1}(\mathbf{y}_1,2) = (3,4,2,0,1,2),$$

showing that $\mathbf{y}_1 \in C_2$. A visual representation is provided in Figure 4.1.



(a) TA-sequence $\mathbf{x}_1 \in C_1$ (b) Mapped sequence $\mathbf{y}_1 = g(\mathbf{x}_1) \in C_2$

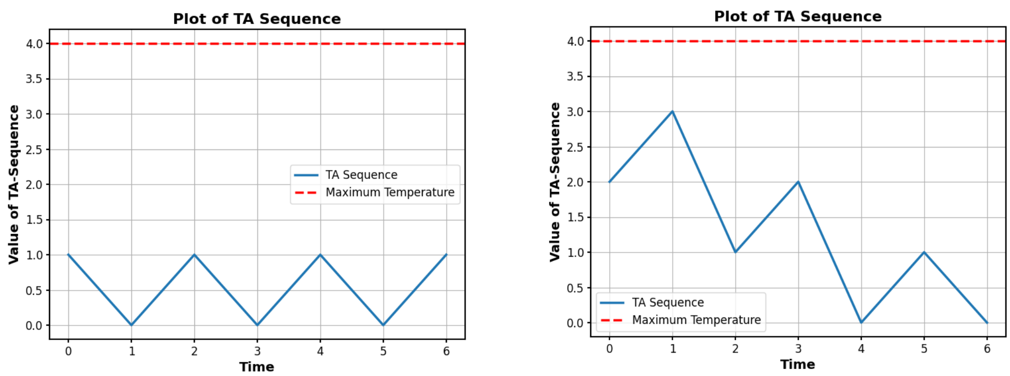
Figure 4.1: Example of a mapping with valid $h_{\mathbf{x}} < n$.

Next, we examine a case where $h_{\mathbf{x}} = n$.

This occurs when $v_i \leq a$ for all $0 \leq i \leq n$. Consider again $N = 4$, $q = 2$, $p = 1$, and $n = 6$. Let $\mathbf{x}_2 = (0,1,0,1,0,1) \in C_1$. Then: $\mathbf{v}_{2,1}(\mathbf{x}_2, 1) = (0,1,0,1,0,1)$ Since no valid index $< n$ exists, we set $h_{\mathbf{x}_2} = n$ and define:

$$\mathbf{y}_2 = g(\mathbf{x}_2) = \mathbf{x}_2^R = (1,0,1,0,1,0) \quad and \quad \mathbf{v}_{2,1}(\mathbf{y}_2, 2) = (3,1,2,0,1,0),$$

showing that $\mathbf{y}_2 \in C_2$. See Figure 4.2 for a graphical illustration.



(a) TA-sequence $\mathbf{x}_2 \in C_1$ (b) Mapped sequence $\mathbf{y}_2 = \mathbf{x}_2^R \in C_2$

Figure 4.2: Example of a mapping when $h_{\mathbf{x}} = n$.

From Theorem 4.2.1 and the symmetry of $|C_a|$ stated in Corollary 3.2.3, we obtain the following result about the cardinality of C_a.

Corollary 4.2.3. *If $p = 1$, then the cardinality of C_a is maximized when $a = \lfloor N/2 \rfloor$, i.e.,*

$$|C_a| \leq \left|C_{\lfloor N/2 \rfloor}\right| \quad \textit{for all } a = 0, 1, \ldots, N.$$

This result suggests that certain symmetry properties may exist between q and p. The observations in the following sections further support this pattern and future research could entail the investigation of this symmetry.

4.2.2. Monotonicity Conjectures for Specific Parameters

Numerical analysis indicated not only the monotonic behavior of $|C_a|$ with respect to a for the case $p = 1$, but also for some other parameter configurations:

Conjecture 1. *If $q = 2$ or $p = 2$ and N is odd, then the cardinality of C_a is maximized when $a = \lfloor N/2 \rfloor$, that is:*

$$|C_a| \le \left|C_{\lfloor N/2 \rfloor}\right| \quad \text{for all } a \in \{0, 1, \ldots, N\}.$$

Conjecture 2. *If $p = 3$ or $q = 3$, then the cardinality of C_a is maximized when $a = \lfloor N/2 \rfloor$ or $a = \lfloor N/2 \rfloor - 1$, that is:*

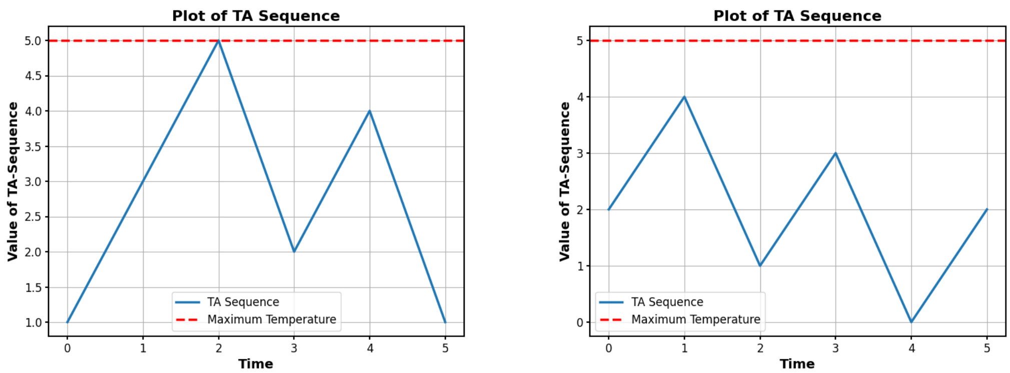
$$|C_a| \le \max\{\left|C_{\lfloor N/2 \rfloor - 1}\right|, \left|C_{\lfloor N/2 \rfloor}\right|\} \quad \text{for all } a \in \{0, 1, \ldots, N\}.$$

Proving these conjectures turned out to be considerably more challenging than for the cases $p = 1$ or $q = 1$. We have tried to construct mappings similar to those in Theorems 3.2.5 and 4.2.1. However, these mappings failed, because they violated the properties of C_a (i.e., they either exceeded the maximum temperature N or ended at a temperature higher than a) or did not meet the injectivity criterion. Below, we present an explicit example where the mapping technique that worked for $p = 1$ was extended, but failed for $p = 2$.

Example 4.2.4. To illustrate that extending the mapping defined in Theorem 4.2.1 to the case where $p = 2$ does not work for all TA-sequences, we will provide two TA-sequences mapped using the function g. This time, define $h_{\mathbf{x}} = \min\{i : \mathbf{v}_{q,p}(\mathbf{x}, a)_i = a + 2\}$. Consider the parameters $N = 5$, $q = 3$, $p = 2$, and $n = 5$. Let $\mathbf{x}_3 = (1, 1, 0, 1, 0) \in C_1$ *and* $\mathbf{v}_{3,2}(\mathbf{x}_3, 1) = (3, 5, 2, 4, 1)$. Here, for $h_{\mathbf{x}_3} = 1$, we have $v_1 = a + 2$. Therefore, the mapping results in:

$$\mathbf{y}_3 = g(\mathbf{x}_1) = (1, 0, 1, 0, 1) \quad \textit{and} \quad \mathbf{v}_{3,2}(\mathbf{y}_3, 2) = (4, 1, 3, 0, 2)$$

showing that $\mathbf{y}_3 \in C_2$. Hence, for this particular TA-sequence, the defined mapping has worked. A visual representation is provided in Figure 4.3.



(a) TA-sequence $\mathbf{x}_3 \in C_1$

(b) Mapped sequence $\mathbf{y}_1 = g(\mathbf{x}_3) \in C_2$

Figure 4.3: Example of a mapping g where $v_{h_{\mathbf{x}_3}} = a + 2$ is defined to be the split index.

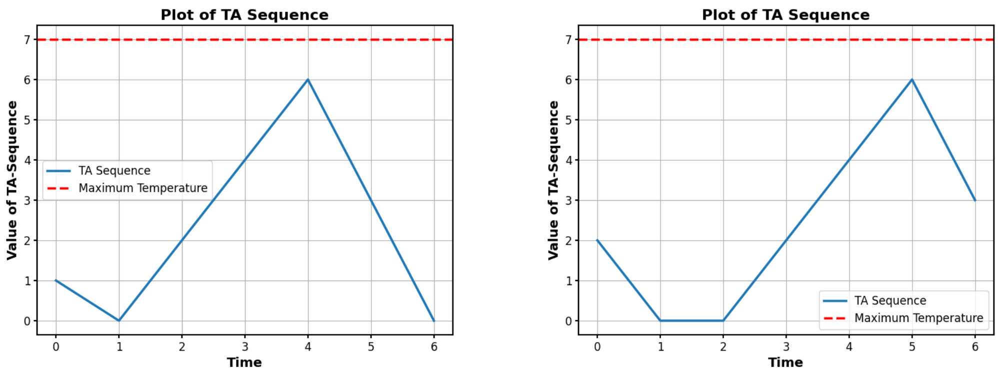
Next, we examine a case where $\mathbf{v}_{q,p}(\mathbf{x}, a)_{h_{\mathbf{x}}} = a + 2$ does exist, but the requirement that $v_j \le N$ for all j fails. Consider $N = 7$, $q = 3$, $p = 2$ and $n = 6$. Let:

$$\mathbf{x}_4 = (0, 1, 1, 1, 0, 0) \in C_1 \quad \textit{and} \quad \mathbf{v}_{3,2}(\mathbf{x}_4, 1) = (0, 2, 4, 6, 3, 0).$$

For index $i = 5$, we have $v_i = a + 2 = 3$. Therefore, we set $h_{\mathbf{x}_4} = 5$ and find:

$$\mathbf{y}_4 = g(\mathbf{x}_4) = (0, 0, 1, 1, 1, 0) \quad \textit{and} \quad \mathbf{v}_{3,2}(\mathbf{y}_4, 2) = (0, 0, 2, 4, 6, 3),$$

showing that $\mathbf{y}_4 \notin C_2$. See Figure 4.4 for a graphical illustration.



(a) TA-sequence $\mathbf{x}_4 \in C_1$ (b) Mapped sequence $\mathbf{y}_2 = g(\mathbf{x}_4) \in C_3$

Figure 4.4: Example of a mapping g where $v_{h_{\mathbf{x}_4}} = a+2$ as split index is invalid.

Example 4.2.4 illustrates one instance of a mapping where certain requirements are not generally satisfied. This issue arises for all explored mappings from C_a to some C_{a+i} with $a < i \leq p$, both for $p = 2$ with N an odd integer, and for $q = 2$ with N odd. An overview of all attempted approaches to prove Conjecture 1 is provided in Appendix B. Nevertheless, numerical analysis strongly supports the general expressions described in Conjecture 1, and no counterexample has been found to date. To showcase this, consider the following example.

Example 4.2.5. Let $N = 5$, $q = 2$, $p = 3$, and $n = 6$. The corresponding matrix $D^{(6)}_{5,2,3}$ is given by:

$$D^{(6)}_{5,2,3} = \begin{bmatrix} 8 & 3 & 1 & 6 & 2 & 1 \\ 8 & 3 & 1 & 4 & 3 & 1 \\ 7 & 3 & 1 & 4 & 1 & 2 \\ 4 & 3 & 1 & 3 & 1 & 0 \\ 4 & 1 & 2 & 3 & 1 & 0 \\ 3 & 1 & 0 & 3 & 1 & 0 \end{bmatrix}$$

Using (3.1), we can determine the corresponding sizes of the sets C_a:

$$|C_0| = 8, \quad |C_1| = 8+3 = 11, \quad |C_2| = 7+3+1 = 11, \quad |C_3| = 4+3+1+3 = 11,$$

$$|C_4| = 4+1+2+3+1 = 11, \quad |C_5| = 3+1+0+3+1+0 = 8.$$

It follows that $|C_a|$ is indeed maximized at $a = \frac{N}{2} = 2$, as stated in Conjecture 1.

Now consider the case where N is an even integer. We take the same (N, q, p) TA-channel as in Example 3.1.2 and let $N = 4$, $q = 2$, $p = 3$, and $n = 6$. The corresponding matrix $D^{(6)}_{4,2,3}$ again is given by:

$$D^{(6)}_{4,2,3} = \begin{bmatrix} 8 & 3 & 1 & 5 & 2 \\ 7 & 3 & 1 & 4 & 2 \\ 5 & 2 & 1 & 3 & 1 \\ 4 & 2 & 1 & 3 & 1 \\ 3 & 1 & 1 & 2 & 1 \end{bmatrix}$$

The corresponding sizes of the sets C_a are:

$$|C_0| = 8, \quad |C_1| = 7+3 = 10, \quad |C_2| = 5+2+1 = 8, \quad |C_3| = 4+2+1+3 = 10, \quad |C_4| = 3+1+1+2+1 = 8.$$

This time, $|C_a|$ is not maximized at $a = \frac{N}{2} = 2$, but at $a = 1$ and by symmetry also at $N - a = 3$. This indicates that the assumption of N being odd is essential for the validity of Conjecture 1.

As for Conjecture 2, no analytical proof has been attempted yet. However, note that the result of Conjecture 2 also holds for both examples from 4.2.5. Together with the established expressions of Theorem 3.2.5 and Corollary 4.2.3, these observations have led to the following hypothesis for general values of p and q.

4.2.3. A General Monotonicity Conjecture

Numerical analysis of the dependence of $|C_a|$ on the parameters of an (N,q,p) TA-channel, combined with the analytical results for the special cases $q=1$ (Chapter 3.2) and $p=1$ (Chapter 4.2.1), suggests that a more general monotonicity property may hold.

Conjecture 3. *Let $\mu=\min\{q,p\}$. Then the cardinality of C_a is maximized for some $a\in\{0,1,\dots,N\}$ satisfying:*

$$\left\lceil \frac{N-\mu}{2} \right\rceil \le a \le \left\lfloor \frac{N}{2} \right\rfloor.$$

This conjecture would directly follow if it could be proven that $|C_a|$ is maximized for some a within the intervals

$$\left\lceil \frac{N-p}{2} \right\rceil \le a \le \left\lfloor \frac{N}{2} \right\rfloor \quad \text{and} \quad \left\lceil \frac{N-q}{2} \right\rceil \le a \le \left\lfloor \frac{N}{2} \right\rfloor.$$

However, no general proof for these intervals has been found thus far. Attempts to apply similar techniques as those used in Theorems 3.2.5 and 4.2.1, namely, by constructing injective mappings to establish monotonicity, have not been successful.

The only results we that we believe can be proven using an injective mapping are the following:

Proposition 4.2.6. *For $0\le a\le \frac{N-p}{2}$,*

$$|C_a| \le \sum_{k=1}^{p} |C_{a+k}|.$$

Proposition 4.2.7. *For $0\le a\le \frac{N-q}{2}$,*

$$|C_a| \le \sum_{k=1}^{q} |C_{a+k}|.$$

Unfortunately, these findings appear to be of limited significance with respect to Conjecture 3. Moreover, the established proofs of these propositions have not been fully verified by the supervisor and are therefore not included in this report.

Even though an analytical proof remains open, we note that Conjecture 3 is consistent with all previous analytical results and numerical observations, as well as the demonstrated examples. In the following, we support this claim with an additional example.

Example 4.2.8. Let $N=6$, $q=4$, $p=5$, and $n=8$. The corresponding matrix $D^{(8)}_{6,4,5}$ is given by:

$$D^{(8)}_{6,4,5} = \begin{bmatrix} 19 & 8 & 3 & 0 & 0 & 12 & 5 \\ 17 & 7 & 3 & 0 & 0 & 11 & 4 \\ 12 & 5 & 2 & 0 & 0 & 8 & 3 \\ 12 & 5 & 2 & 0 & 0 & 8 & 3 \\ 12 & 5 & 2 & 0 & 0 & 8 & 3 \\ 11 & 4 & 2 & 0 & 0 & 7 & 3 \\ 8 & 3 & 1 & 0 & 0 & 5 & 2 \end{bmatrix}$$

Using (3.1), we can determine the corresponding sizes of the sets C_a:

$$\begin{aligned} |C_0| &= 19, \\ |C_1| &= 17+7 = 24, \\ |C_2| &= 12+5+2 = 19, \\ |C_3| &= 12+5+2+0 = 19, \\ |C_4| &= 12+5+2+0+0 = 19, \\ |C_5| &= 11+4+2+0+0+7 = 24, \\ |C_6| &= 8+3+1+0+0+5+2 = 19. \end{aligned}$$

It follows that $|C_a|$ is maximized at $a=1$, and by symmetry also at $N-a=5$. This result does not coincide with the expression found for $q=1$ and $p=1$, where the maximum occurs at $\frac{N}{2}=3$. However, the interval proposed in Conjecture 3 states that the optimal value of a lies within

$$\left[\left\lceil \frac{N-\mu}{2} \right\rceil, \left\lfloor \frac{N}{2} \right\rfloor \right] = [1,3],$$

which remains consistent with the proposed claim.

In conclusion, the belief that Conjecture 3, and consequently Conjectures 1 and 2, are valid remains. Future research may focus on establishing analytical proofs for these statements. Since injective mappings techniques from one set C_a to another C_{a+i} for some $a < i \leq p$ have been extensively explored without success, alternative approaches could be considered. For example, one could look at matrix properties following from the transitioning matrix induced by a (N, q, p) TA-channel. Additionally, the derivation of more explicit expressions and the investigation of possible symmetry between q and p remain open for investigation. Due to time constraints, these aspects were not addressed in this research project.

5 Conclusion and Future Recommendations

In this research project, we investigated the capacity of thermal-aware (TA) channels for sequences of fixed length, focusing on the sets $C_a \subseteq \mathscr{A}(N, q, p, n)$. The parameters N, q, p and n represent the maximum allowable temperature, cooling and heating gradient and sequence length, respectively. The sets C_a consist of admissible sequences of length n that start at temperature a and end at or below a, making them useful for transmitting cascaded sequences. The main goal was to see how the size of C_a depends on the channel parameters and to potentially find a general expression for the value of a that maximizes $|C_a|$.

We started by reviewing known results in the asymptotic domain, which provide theoretical upper bounds. In the finite-length setting, we extended an established monotonicity result for $q = 1$, by proving that the same holds when $p = 1$. This showed that for certain parameters, $|C_a|$ increases with a up to a symmetry point. Numerical analysis indicated similar patterns for other configurations, leading to several conjectures. Finally, this led to a conjecture for general values of N, q, p, n, stating that the maximum of $|C_a|$ occurs within a specific interval centered around $N/2$. The size of this interval is proportional to $\min(q, p)$, making the result particularly informative when these parameters are relatively small.

Although many mapping strategies were tried, a rigorous proof of the proposed conjectures could not be found. Still, all numerical evidence points toward their validity, and no counterexamples have been found. These observations support the idea that the optimal starting value a lies somewhere around the midpoint of the temperature range, dependent on the heating and cooling gradient of the TA-channel.

The results contribute to the understanding of how thermal constraints affect the overall code rate in the finite domain. Future research could explore different techniques to prove the stated conjectures, using properties of the transition matrices rather than sequence mappings, for example. Another option would be to derive better approximations or even closed-form expressions for the optimal index a. In addition, follow-up work might look into potential symmetries between p and q, or study the relation between $|C_a|$ and the parameters N and n.

Overall, this project has extended known results and introduced new claims, which remain open for future work on computing TA-channel capacities in the finite domain. Finding these capacities allows us to compare the actual code rate with the theoretical rate and, ultimately, to determine the efficiency and real-world applicability of thermal-aware communication systems.

Bibliography

[1] Norman Biggs. *Algebraic Graph Theory*. Cambridge University Press, 2nd edition edition, 1993.

[2] Yeow Meng Chee, Charles J. Colbourn, and Alan C. H. Ling. Optimal memoryless encoding for low power off-chip data buses. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 369–374, 2006. doi: 10.1109/ICCAD.2006.320040.

[3] Yeow Meng Chee, Tuvi Etzion, Han Mao Kiah, and Alexander Vardy. Cooling codes: Thermal-management coding for high-performance interconnects. *IEEE Transactions on Information Theory*, 64(4):3062–3085, 2018.

[4] Yeow Meng Chee, Tuvi Etzion, Hieu M. Kiah, Alexander Vardy, and Heung Kwan Wei. Low-power cooling codes with efficient encoding and decoding. *IEEE Transactions on Information Theory*, 66(8):4804–4818, 2020. doi: 10.1109/TIT.2020.2977228.

[5] Yeow Meng Chee, Tuvi Etzion, Kees A. Schouhamer Immink, Tuan Thanh Nguyen, Van Khu Vu, Jos H. Weber, and Eitan Yaakobi. Thermal-aware channel capacity. In *2023 IEEE International Symposium on Information Theory (ISIT)*, pages 2661–2666, Taipei, Taiwan, 2023. doi: 10.1109/ISIT54713.2023.10206738.

[6] Yeow Meng Chee, Tuvi Etzion, Kees A. Schouhamer Immink, Tuan Thanh Nguyen, Van Khu Vu, Jos H. Weber, and Eitan Yaakobi. Thermal-aware communication. *IEEE Transactions on Information Theory*, 71(6):4145–4154, June 2025. doi: 10.1109/TIT.2025.3555665.

[7] Yong Liang Guan, Guojie Han, Lin Kong, Ka Sing Chan, Kui Cai, and Jie Zheng. Coding and signal processing for ultra-high density magnetic recording channels. In *2014 International Conference on Computing, Networking and Communications (ICNC)*, pages 194–199, 2014. doi: 10.1109/ICCNC.2014.6785287.

[8] Kees A. S. Immink. *Codes for Mass Data Storage Systems*. Shannon Foundation Publishers, 2004.

[9] Kees A. Schouhamer Immink. A survey of codes for optical disk recording. *IEEE Journal on Selected Areas in Communications*, 19(4):756–764, 2001. doi: 10.1109/49.918572.

[10] Douglas Lind and Brian Marcus. *An Introduction to Symbolic Dynamics and Coding*. Cambridge University Press, 1995.

[11] Brian Marcus and Douglas Lind. *Symbolic Dynamics and Coding*. Cambridge University Press, reprint with corrections edition, 2001.

[12] S. Ramprasad, N. R. Shanbhag, and I. N. Hajj. Information-theoretic bounds on average signal transition activity in vlsi systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(3):359–368, 1999. doi: 10.1109/92.777450.

A

Proofs of Chapter 3

A.1. Proof of Theorem 3.1.1

Theorem A.1.1 (From [1]). *[Restatement of Theorem 3.1.1] Let $G = (V, E)$ be a finite graph with adjacency matrix A. Then, for all integers $n \geq 0$, the entry $(A^n)_{ij}$ equals the number of walks of length n from vertex i to vertex j.*

Proof. We prove the theorem by induction on n.

Base case: When $n = 0$, $A^0 = I$, the identity matrix. Since a walk of length 0 starts and ends at the same vertex, we have $(A^0)_{ij} = \delta_{ij}$, where δ_{ij} is the Kronecker delta. This matches the number of walks of length 0 from i to j.

Inductive step: Assume the statement holds for $n = k$, i.e., $(A^k)_{ij}$ gives the number of walks of length k from i to j. We must show that it holds for $n = k+1$.

By the properties of matrix multiplication:

$$(A^{k+1})_{ij} = \sum_l (A^k)_{il} \cdot A_{lj}.$$

By the inductive hypothesis, $(A^k)_{il}$ is the number of walks of length k from i to l, and A_{lj} is 1 if there is an edge from l to j (i.e., a walk of length 1), and 0 otherwise. Hence, $(A^{k+1})_{ij}$ counts the number of walks of length $k+1$ from i to j.

This completes the induction, and the theorem follows.

□

A.2. Proof of Theorem 3.2.2

Theorem A.2.1 (From [6]). *[Restatement of Theorem 3.2.2] For all $a \in \{0, 1, \ldots, N\}$ it holds that*

$$x \in C_a \iff x^R \in C_{N-a}.$$

Proof. Let x be any sequence in C_a, and let $v_{q,p}(x, a) = (v_1, v_2, \ldots, v_n)$, $t_{q,p}(x) = (t_1, t_2, \ldots, t_n)$, $v_{q,p}(x^R, N-a) = (v'_1, v'_2, \ldots, v'_n)$, and $t_{q,p}(x^R) = (t'_1, t'_2, \ldots, t'_n)$. We first show that $x \in C_a$ implies $x^R \in C_{N-a}$, i.e.,

(a) $v'_i \leq N$ for all i, and

(b) $v'_n \leq N - a$.

(a) To prove $v'_i \leq N$ for all i, suppose there exists $j \in \{1, 2, \ldots, n\}$ such that $v'_j > N$. We consider two cases:
Case (i): There is no $g \leq j$ such that $v'_g = 0$. Then $v'_j = N - a + t'_j$ and thus

$$t'_j = v'_j - N + a > N - N + a = a.$$

Hence,

$$v_n \geq v_{n-j} + t_n - t_{n-j} = v_{n-j} + t'_j > 0 + a = a,$$

which contradicts $x \in C_a$.

Case (ii): There exists $g \leq j$ such that $v'_g = 0$, and let g^* be the largest such index. Then,

$$t'_j - t'_{g^*} = v'_j - v'_{g^*} > N - 0 = N,$$

which implies

$$v_{n-g^*} \geq v_{n-j} + t_{n-g^*} - t_{n-j} = t'_j - t'_{g^*} > N,$$

again contradicting $x \in C_a$. Thus, $v'_i \leq N$ for all i.

(b) Now we show $v'_n \leq N - a$. Suppose instead that $v'_n > N - a$. We again consider two cases:

Case (i): There is no $g \leq n$ such that $v'_g = 0$. Then $v'_n = N - a + t'_n$ and

$$t'_n = v'_n - N + a > N - a - N + a = 0.$$

So,

$$v_n \geq a + t_n = a + t'_n > a,$$

contradicting $x \in C_a$.

Case (ii): There exists $g \leq n$ such that $v'_g = 0$, and let g^* be the largest such index. Then,

$$t'_n - t'_{g^*} = v'_n - v'_{g^*} = v'_n > N - a,$$

and it follows that

$$v_{n-g^*} \geq a + t_{n-g^*} = a + t'_n - t'_{g^*} > a + N - a = N,$$

again contradicting $x \in C_a$. Hence, $v'_n \leq N - a$.

It follows from (a) and (b) that the "⇒" statement holds. The "⇐" direction follows by symmetry, since $(x^R)^R = x$. □

A.3. Proof of Theorem 3.2.5

Theorem A.3.1 (From [6])**.** *[Restatement of Theorem 3.2.5] If* $q = 1$ *and* $0 \leq a \leq N/2 - 1$*, then*

$$|C_a| \leq |C_{a+1}|.$$

Proof. The proof is established by providing an injective mapping f from C_a to C_{a+1}. Let x be any sequence in C_a, and let

$$v_{q,p}(x, a) = (v_1, v_2, \ldots, v_n), \quad t_{q,p}(x) = (t_1, t_2, \ldots, t_n), \quad \text{and} \quad t_{q,p}(x^R) = (t'_1, t'_2, \ldots, t'_n).$$

Define z_x as the smallest index for which the running digital sum of x becomes negative, i.e.,

$$t_i \geq 0 \quad \text{for all } 1 \leq i \leq z_x - 1, \quad \text{and} \quad t_{z_x} = -1.$$

If such an index does not exist, i.e., if $t_i \geq 0$ for all $1 \leq i \leq n$, then set $z_x = n$.

We decompose x as

$$x = (u, w),$$

with u of length $n - z$ and w of length z, where $z = z_x^R$, and map x to

$$y = f(x) = (w^R, u).$$

Let $v_{q,p}(y, a+1) = (v''_1, v''_2, \ldots, v''_n)$ and $t_{q,p}(y) = (t''_1, t''_2, \ldots, t''_n)$. We will show that $y \in C_{a+1}$, that is:

(a) $v''_i \leq N$ for all i,

(b) $v''_n \leq a + 1$,

and furthermore, that y is unique for every x.

Observe that if there exists an index i such that $t'_i = -1$, then from the definitions of z and y, we have

$$v''_z = a + 1 + t'_z = a + 1 - 1 = a$$

and

$$v''_i = v_{i-z} \quad \text{for all } z+1 \le i \le n. \tag{A.1}$$

(a) Note that $t''_i = t'_i \le a$ for all $1 \le i \le z$, since $t'_j \ge a+1$ for any $j \in \{1, 2, \ldots, z\}$ would imply

$$v_n \ge v_{n-j} + t'_j \ge a + 1,$$

which contradicts $x \in C_a$. Hence,

$$v''_i = a + 1 + t''_i \le a + 1 + a = 2a + 1 \le N$$

for all $1 \le i \le z$. If $z < n$, then from (A.1) we have

$$v''_i = v_{i-z} \le N \quad \text{for all } z+1 \le i \le n.$$

(b) If there exists an index i such that $t'_i = -1$, then from (A.1),

$$v''_n = v_{n-z} \le v_n - t'_z \le a + 1,$$

where the last inequality follows from $v_n \le a$ and $t'_z = -1$.

If no such index exists, then $z = n$, $y = x^R$, and thus

$$v''_n = a + 1 + t'_n = a + 1 + t_n \le a + 1,$$

since $t_n \le 0$. (If $t_n > 0$, then $v_n \ge a + t_n > a$, which contradicts $x \in C_a$.)

Therefore, we conclude that (a) $v''_i \le N$ for all i, and (b) $v''_n \le a+1$, so $y \in C_{a+1}$.

Finally, note that $z = z^R_x = z_y$, so we can retrieve z from y and thus establish the inverse mapping

$$f^{-1}(y) = f^{-1}((w^R, u)) = (u, (w^R)^R) = (u, w) = x.$$

Hence, f is indeed an injective mapping from C_a to C_{a+1}, which proves the theorem. □

B

Explored Mappings in Chapter 4

Here, the explored mappings that were considered in the search for an injective function to prove the conjectures stated in Chapter 4.2.2 Chapter 4.2.3 are presented. It should be noted that none of these mappings were successful. For detailed definitions of the involved variables, we refer to the proofs of Theorems 3.2.5 and 4.2.1.

B.1. Explored Mappings to Prove Conjecture 1

B.1.1. Explored Mappings to prove Conjecture 1 for $p = 2$

Table B.1: Decompositions of the vector $\mathbf{X} = (u, w)$ and definition of the split index h

Case	**Decomposition of x**	**Split index** h	**Shift** k**:** $C_a \mapsto C_{a+k}$
1	$(\mathbf{w}, \mathbf{u}^R)$	$\min\{i : t_i = 1\}$	1
2	$(\mathbf{w}, \mathbf{u}^R)$	$\min\{i : t_i = 2\}$	1
3	$(\mathbf{w}, \mathbf{u}^R)$	$\min\{i : v_i = a+1\}$	1
4	$(\mathbf{w}, \mathbf{u}^R)$	$\min\{i : v_i = a+2\}$	2
5	$(\mathbf{u}, \mathbf{w}^R)$	$\min\{i : v_i = N-a+1\}$	1
6	$(\mathbf{u}, \mathbf{w}^R)$	$\min\{i : v_i = N-a+1\}$	1
7	$(\mathbf{w}, \mathbf{u})$	$\min\{i : v_i = a+1\}$	1
8	$(\mathbf{w}, \mathbf{u})$	$\min\{i : v_i = a+2\}$	2
9	$(\mathbf{w}, \mathbf{u})$	$\min\{i : t_i = 1\}$	1
10	$(\mathbf{w}, \mathbf{u}^R)$	$\min\{i : t_i = 2\}$	1

Furthermore, several configurations in which **x** was decomposed into three parts were explored. These are presented in Table B.2. However, these attempts were also unsuccessful and generally failed to satisfy the injectivity criterion.

Table B.2: Decompositions of the vector $\mathbf{x} = (\mathbf{u}, \mathbf{v}, \mathbf{w})$ and definition of the split indices

Case	**Decomposition of x**	**Split index** h_1	**Split index** h_2	**Shift** k**:** $C_a \mapsto C_{a+k}$
1	$(\mathbf{w}, \mathbf{v}, \mathbf{u}^R)$	$\min\{i : t_i = -1\}$	$\min\{i : t'_i = 2\}$	1
2	$(\mathbf{w}^R, \mathbf{v}, \mathbf{u})$	$\min\{i : t_i = 1\}$	$\min\{i : t'_i = -2\}$	1
3	$(\mathbf{w}^R, \mathbf{v}, \mathbf{u}^R)$	$\min\{i : t_i = 1\}$	$\min\{i : t'_i = -2\}$	1
4	$(\mathbf{w}, \mathbf{v}, \mathbf{u}^R)$	$\min\{i : t_i = -1\}$	$\min\{i : t'_i = 2\}$	1
5	$(\mathbf{w}^R, \mathbf{v}, \mathbf{u})$	$\min\{i : t_i = 1\}$	$\min\{i : t'_i = -2\}$	1
6	$(\mathbf{w}^R, \mathbf{v}, \mathbf{u}^R)$	$\min\{i : t_i = 1\}$	$\min\{i : t'_i = -2\}$	1
7	$(\mathbf{w}, \mathbf{v}, \mathbf{u})$	$\min\{i : t_i = -1\}$	$\min\{i : t'_i = 2\}$	1

B.1.2. Explored Mappings to Prove Conjecture 1 for $q = 2$

Here, we present the decompositions that were considered in an attempt to prove Conjecture 1 for the case $q = 2$. These are based, though not exclusively, on the mapping defined in Theorem 3.2.5.

Table B.3: Decompositions of the vector $\mathbf{X} = (u, w)$ and definition of the split index z

Case	Decomposition of x	Split index z	Shift k
1	$(\mathbf{w}^R, \mathbf{u})$	$\min\{i : t'_i = -1\}$	1
2	$(\mathbf{w}^R, \mathbf{u})$	$\min\{i : t'_i = -2\}$	1
3	$(\mathbf{w}^R, \mathbf{u})$	$\min\{i : t'_i = -3\}$	1
4	$(\mathbf{w}^R, \mathbf{u})$	$\min\{i : t'_i = -1\}$	2
5	$(\mathbf{w}^R, \mathbf{u})$	$\min\{i : t'_i = -2\}$	2
6	$(\mathbf{w}, \mathbf{u})$	$\min\{i : t'_i = -1\}$	1
7	$(\mathbf{w}, \mathbf{u})$	$\min\{i : t'_i = -2\}$	1

Furthermore, as in the case where $p = 2$, several configurations in which $\mathbf{x}$ was decomposed into three parts were explored. Since these decompositions are similar to those presented in Table B.2, they are omitted here to avoid repetition. However, these attempts were also unsuccessful for $q = 2$, which led to the conclusion that, in general, a decomposition into three parts may not be the most logical approach for constructing a valid mapping to prove these conjectures.

B.2. Explored Mappings to Prove Conjecture 3

Similar to the previous section, we now present the mappings that were considered in an attempt to prove Conjecture 3. Although all mappings were injective, none satisfied both criteria of C_k for $k = p$ or $k = q$. Specifically, in all cases either the condition $v_j \leq N$ failed for some j, or the condition $v_n \leq a + k$ was violated.

Table B.4: Decompositions of the vector $\mathbf{x} = (u, w)$ and definition of the split index

Case	Decomposition of x	Definition Split Index	Shift k: $C_a \mapsto C_{a+k}$
1	$(\mathbf{w}, \mathbf{u}^R)$	$\min\{i : v_i = a + p\}$	p
2	$(\mathbf{w}, \mathbf{u}^R)$	$\min\{i : a < v_i \leq a + p\}$	p
3	$(\mathbf{w}, \mathbf{u}^R)$	$\min\{i : a + p \leq v_i < a + 2p\}$	p
4	$(\mathbf{w}^R, \mathbf{u})$	$\min\{i : t_i = -q\}$	q
5	$(\mathbf{w}^R, \mathbf{u})$	$\min\{i : -q \leq t_i < 0\}$	q
6	$(\mathbf{w}^R, \mathbf{u})$	$\min\{i : -2q < t_i \leq q\}$	q

C
Python Code

C.1. Code for General Investigations

```python
import numpy as np
import pandas as pd
import math
```

FUNCTIONS DEFINING

```python
def generate_matrix(N, q, p, n):
    # Step 1: Build D matrix
    D = np.zeros((N + 1, N + 1), dtype=int)

    for i in range(N + 1):
        if i <= N - p:
            D[i][i + p] = 1
        if i >= q:
            D[i][i - q] = 1
        if i < q:
            D[i][0] = 1

    # Step 2: Compute D^n
    Dn = np.linalg.matrix_power(D, n)

    return Dn


# def find_optimal_capacity(Dn):
#     N = Dn.shape[0] - 1
#     half_index = math.ceil(N / 2)
#     capacities = [np.sum(Dn[i, :i + 1]) for i in range(half_index +
1)]

#     # Check if all capacities are the same
#     all_same = len(set(capacities)) == 1
#     if all_same:
#         same_value = capacities[0]
#     else:
#         same_value = None

#     # Find the optimal index and capacity
#     max_cap = max(capacities)
#     opt_index = capacities.index(max_cap)

#     return opt_index, max_cap, all_same, same_value

def find_optimal_capacity(Dn):
    N = Dn.shape[0] - 1
    half_index = N // 2  # Go only up to N//2

    # Calculate capacities for indices from 0 to N//2
```

```
    capacities = [np.sum(Dn[i, :i + 1]) for i in range(half_index +
1)]

    # Check if all capacities are the same
    all_same = len(set(capacities)) == 1
    if all_same:
        same_value = capacities[0]
    else:
        same_value = None

    # Find the maximum capacity
    max_cap = max(capacities)

    # Find all indices with the maximum capacity, only within the
range 0 to N//2
    indices_with_max_cap = [i for i, cap in enumerate(capacities) if
cap == max_cap]

    # Return the highest index with the maximum capacity
    opt_index = max(indices_with_max_cap)

    return opt_index, max_cap, all_same, same_value

def get_index_relation(index, N):
    # Check if the optimal index equals 0
    if index == 0:
        return "i is 0"
    elif index == N // 2:
        return "i == N//2"
    elif index == N // 2 - 1:
        return "i == N//2 - 1"
    elif index == N // 2 + 1:
        return "i == N//2 + 1"
    else:
        return "i = " + str(index) + " N = " + str(N)

# #function to check whether p/q actually smallest prime numbers
# def is_prime(num):
#     if num <= 1:
#         return False
#     for i in range(2, int(num ** 0.5) + 1):
#         if num % i == 0:
#             return False
#     return True

#check whether p/q is rational number (smallest possible fraction)
def is_smallest_fraction(p, q):
    # Check if the GCD of p and q is 1
    return math.gcd(p, q) == 1
```

```python
def run_configurations(N_values, q_values, p_values, n_values,
constraint, output_path):
    results = []

    for N in N_values:
        for q in q_values:
            for p in p_values:
                for n in n_values:
                    # skip invalid configs
                    if p > N or q > N:
                        continue

                    # Check if both p and q are smallest fraction
                    if not is_smallest_fraction(p,q):
                        continue

                    # Check if constraint
                    if constraint(N,q,p,n):
                        continue

                    # Generate the matrix and find optimal capacity
                    Dn = generate_matrix(N, q, p, n)
                    opt_index, cap, all_same, same_value =
find_optimal_capacity(Dn)
                    rel = get_index_relation(opt_index, N)

                    # Add the results to the list
                    results.append({
                        "N": N,
                        "q": q,
                        "p": p,
                        "n": n,
                        "Optimal Index": opt_index,
                        "Capacity": cap,
                        "Idx Relation": rel,
                        "All Capacities Same": all_same,
                    })

    # Convert the results into a DataFrame
    df = pd.DataFrame(results)
    df.to_csv(output_path, index=False)

    return df
```

Use examples to validate code

```python
def constraint_test(N,p,q,n):
    return False
```

```
# Example Article 3
matrix_example_3 = generate_matrix(3,1,2,12)
print(matrix_example_3)

opt_idx_example_3, cap_example_3, _, _ =
find_optimal_capacity(matrix_example_3)
print(opt_idx_example_3, cap_example_3)

df_example_3 = run_configurations([3],[1],[2], [12], constraint_test,
"/kaggle/working/runs_example_3.csv")
print(df_example_3)
```

```
[[98 56 83 36]
 [83 51 76 27]
 [56 27 51 20]
 [36 20 27 15]]
1 134
   N  q  p   n  Optimal Index  Capacity Idx Relation  All Capacities
Same
0  3  1  2  12              1       134    i == N//2
False
```

```
# Example Article 4
matrix_example_4 = generate_matrix(7,3,4,5)
print(matrix_example_4)

opt_idx_example_4, cap_example_4, _, _=
find_optimal_capacity(matrix_example_4)
print(opt_idx_example_4, cap_example_4)

df_example_4 = run_configurations(range(7,8),[3],[4], [5],
constraint_test, "/kaggle/working/runs_example_4.csv")
print(df_example_4)
```

```
[[5 2 1 0 3 1 1 0]
 [5 2 1 0 3 1 0 1]
 [4 3 1 0 3 1 0 0]
 [4 1 2 0 3 1 0 0]
 [3 1 0 1 2 1 0 0]
 [3 1 0 0 3 1 0 0]
 [3 1 0 0 1 2 0 0]
 [2 1 0 0 1 0 1 0]]
2 8
   N  q  p  n  Optimal Index  Capacity   Idx Relation  All Capacities
Same
0  7  3  4  5              2         8  i == N//2 - 1
False
```

```
# check for q = 1 whether code works (should always be N//2)
df_test_q_1 = run_configurations(range(2,30),[1],range(1,500),
range(1,20), constraint_test,
```

```
"/kaggle/working/runs_test_q_equal_1.csv")
print(df_test_q_1)
```

```
        N  q   p   n  Optimal Index  Capacity Idx Relation  All
Capacities Same
0       2  1   1   1              1         1     i == N//2
True
1       2  1   1   2              1         3     i == N//2
False
2       2  1   1   3              1         4     i == N//2
False
3       2  1   1   4              1         9     i == N//2
False
4       2  1   1   5              1        14     i == N//2
False
...    .. ..  ..  ..            ...       ...           ...
...
8241   29  1  29  15             14         1     i == N//2
True
8242   29  1  29  16             14         1     i == N//2
True
8243   29  1  29  17             14         1     i == N//2
True
8244   29  1  29  18             14         1     i == N//2
True
8245   29  1  29  19             14         1     i == N//2
True

[8246 rows x 8 columns]
```

```
#check if no optimal indices are unequal to N//2
df_check_test = df_test_q_1[df_test_q_1["Idx Relation"] != "i ==
N//2"]
print(df_check_test)

# CONCLUSION: CODE SEEMS TO WORK!
```

```
Empty DataFrame
Columns: [N, q, p, n, Optimal Index, Capacity, Idx Relation, All
Capacities Same]
Index: []
```

A: Investigating Ratio P/Q < 1

```
# define constraint
def constraint_a(N, q, p, n):
    return p / q >= 1 # Define your custom constraint logic
```

```
# example to check

example_matrix = generate_matrix(9,3,2,5)
print(example_matrix)

o,m,a,s = find_optimal_capacity(example_matrix)
#check if all capacities are the same
print(a)

opt_idx = get_index_relation(o,9)
#print optimal index (first optimal)
print(opt_idx)

df_test = run_configurations([9],[3],[2],[5], constraint_a,
"/kaggle/working/runs_test.csv")
print(df_test)
```

```
[[12  2  6  2  3  3  2  0  1  0]
 [10  4  6  2  3  0  5  0  1  0]
 [10  1  9  2  3  0  1  3  1  0]
 [10  1  4  7  3  0  1  0  4  0]
 [10  1  4  0  9  0  1  0  0  3]
 [10  1  4  0  1  8  1  0  0  0]
 [ 5  5  4  0  1  0  7  0  0  0]
 [ 5  0  9  0  1  0  0  5  0  0]
 [ 4  0  1  5  1  0  0  0  3  0]
 [ 4  0  1  0  5  0  0  0  0  2]]
False
i == N//2
   N  q  p  n  Optimal Index  Capacity Idx Relation  All Capacities
Same
0  9  3  2  5              4        24    i == N//2
False
```

```
#run for many configurations
df_runs_a =
run_configurations(range(10,50),range(1,100),range(1,100),range(5,7),
constraint_a, "/kaggle/working/runs_pq_smaller_1.csv")

print(df_runs_a)
```

```
        N   q   p  n  Optimal Index  Capacity  Idx Relation  \
0      10   1   1  5              5        16     i == N//2
1      10   1   1  6              5        42     i == N//2
2      10   2   1  5              5        26     i == N//2
3      10   2   1  6              5        57     i == N//2
4      10   3   1  5              5        26     i == N//2
...    ..  ..  .. ..            ...       ...           ...
25147  49  49  46  6              3        13  i = 3 N = 49
25148  49  49  47  5              2         8  i = 2 N = 49
```

```
25149  49  49  47  6              2         13  i = 2 N = 49
25150  49  49  48  5              1          8  i = 1 N = 49
25151  49  49  48  6              1         13  i = 1 N = 49

       All Capacities Same
0                    False
1                    False
2                    False
3                    False
4                    False
...                    ...
25147                False
25148                False
25149                False
25150                False
25151                False

[25152 rows x 8 columns]

# check for p/q < 0.5
df_check_pq_smaller_half = df_runs_a[df_runs_a["p"] / df_runs_a["q"] <
0.5]

df_check_a = df_check_pq_smaller_half[df_check_pq_smaller_half["Idx
Relation"] == "i == N//2"]
print(df_check_a)

df_check_a = df_check_pq_smaller_half[df_check_pq_smaller_half["Idx
Relation"] != "i == N//2"]

print(df_check_a)

# CONCLUSION: p/q < 0.5 niet genoeg om optimale idx altijd N//2 te
laten zijn

        N   q   p  n  Optimal Index  Capacity Idx Relation  \
4      10   3   1  5              5        26    i == N//2
5      10   3   1  6              5        57    i == N//2
8      10   4   1  5              5        31    i == N//2
9      10   4   1  6              5        57    i == N//2
12     10   5   1  5              5        31    i == N//2
...    ..  ..  .. ..            ...       ...          ...
25105  49  49  22  6             24        31    i == N//2
25106  49  49  23  5             24        17    i == N//2
25107  49  49  23  6             24        31    i == N//2
25108  49  49  24  5             24        17    i == N//2
25109  49  49  24  6             24        31    i == N//2

       All Capacities Same
4                    False
```

```
5                     False
8                     False
9                     False
12                    False
...                     ...
25105                 False
25106                 False
25107                 False
25108                 False
25109                 False

[9561 rows x 8 columns]
        N   q   p  n  Optimal Index  Capacity   Idx Relation  \
14     10   5   2  5              4        25  i == N//2 - 1
15     10   5   2  6              4        52  i == N//2 - 1
26     10   7   2  5              4        25  i == N//2 - 1
27     10   7   2  6              4        52  i == N//2 - 1
28     10   7   3  5              4        21  i == N//2 - 1
...    ..  ..  .. ..            ...       ...            ...
25091  49  49  13  6             23        41  i == N//2 - 1
25092  49  49  15  5             19        21  i = 19 N = 49
25093  49  49  15  6             19        41  i = 19 N = 49
25094  49  49  16  5             17        21  i = 17 N = 49
25095  49  49  16  6             17        41  i = 17 N = 49

       All Capacities Same
14                    False
15                    False
26                    False
27                    False
28                    False
...                     ...
25091                 False
25092                 False
25093                 False
25094                 False
25095                 False

[2935 rows x 8 columns]

#perform test to check relations
df_check_a = df_runs_a[df_runs_a["Idx Relation"] == "i == N//2"]
print(df_check_a)

df_check_a = df_runs_a[df_runs_a["Idx Relation"] == "i == N//2 - 1"]
print(df_check_a)

df_check_a.to_csv("/kaggle/working/runs_pq_smaller_1.csv", index =
False)
```

```
# CONCLUSION: NOTHING REALLY TO BE FOUND?
        N   q   p  n  Optimal Index  Capacity Idx Relation  \
0      10   1   1  5              5        16     i == N//2
1      10   1   1  6              5        42     i == N//2
2      10   2   1  5              5        26     i == N//2
3      10   2   1  6              5        57     i == N//2
4      10   3   1  5              5        26     i == N//2
...    ..  ..  .. ..            ...       ...           ...
25107  49  49  23  6             24        31     i == N//2
25108  49  49  24  5             24        17     i == N//2
25109  49  49  24  6             24        31     i == N//2
25110  49  49  25  5             24         8     i == N//2
25111  49  49  25  6             24        13     i == N//2

       All Capacities Same
0                    False
1                    False
2                    False
3                    False
4                    False
...                    ...
25107                False
25108                False
25109                False
25110                 True
25111                 True

[14542 rows x 8 columns]
        N   q   p  n  Optimal Index  Capacity   Idx Relation  \
6      10   3   2  5              4        25  i == N//2 - 1
7      10   3   2  6              4        41  i == N//2 - 1
10     10   4   3  5              4        15  i == N//2 - 1
11     10   4   3  6              4        34  i == N//2 - 1
14     10   5   2  5              4        25  i == N//2 - 1
...    ..  ..  .. ..            ...       ...            ...
25045  49  48  13  6             23        41  i == N//2 - 1
25090  49  49  13  5             23        21  i == N//2 - 1
25091  49  49  13  6             23        41  i == N//2 - 1
25112  49  49  26  5             23         8  i == N//2 - 1
25113  49  49  26  6             23        13  i == N//2 - 1

       All Capacities Same
6                    False
7                    False
10                   False
11                   False
14                   False
...                    ...
```

```
25045                    False
25090                    False
25091                    False
25112                    False
25113                    False

[2019 rows x 8 columns]
```

B: Investigate P/Q > 1

```
# define constraint
def constraint_b(N, q, p, n):
    return (p / q <= 1 or p%q == 0) # Define your custom constraint logic

#run for many configurations
df_runs_b = 
run_configurations(range(10,50),range(1,50),range(1,100),range(5,7), constraint_b, "/kaggle/working/runs_pq_bigger_N.csv")

print(df_runs_b)
```

```
        N   q   p  n  Optimal Index  Capacity   Idx Relation  \
0      10   2   3  5              4        15  i == N//2 - 1
1      10   2   3  6              4        21  i == N//2 - 1
2      10   2   5  5              5         6      i == N//2
3      10   2   5  6              5         7      i == N//2
4      10   2   7  5              5         4      i == N//2
...    ..  ..  .. ..            ...       ...            ...
22787  49  47  48  6              1        10    i = 1 N = 49
22788  49  47  49  5             24         4      i == N//2
22789  49  47  49  6             24         6      i == N//2
22790  49  48  49  5             24         4      i == N//2
22791  49  48  49  6             24         6      i == N//2

       All Capacities Same
0                    False
1                    False
2                    False
3                    False
4                    False
...                    ...
22787                False
22788                False
22789                False
22790                 True
22791                 True
```

```
[22792 rows x 8 columns]

#perform test to check relations
df_check_b = df_runs_b[df_runs_b["Idx Relation"] == "i == N//2"]
df_check_b_idx = df_runs_b[df_runs_b["Idx Relation"] == "i == N//2 -
1"]

print(df_check_b)
print(df_check_b_idx)

# CONCLUSION: if all capacities same then optimal idx == 0 (duhh want
alle hetzelfde dus je neemt eerste)
# CONCLUSION: for small amount, all capacities are the same, (about 1%
of cases) -> not really conclusion

df_check_b.to_csv("/kaggle/working/runs_pq_bigger_N_filtered.csv",
index=False)

        N   q   p  n  Optimal Index  Capacity Idx Relation  \
2      10   2   5  5              5         6    i == N//2
3      10   2   5  6              5         7    i == N//2
4      10   2   7  5              5         4    i == N//2
5      10   2   7  6              5         5    i == N//2
6      10   2   9  5              5         1    i == N//2
...    ..  ..  .. ..            ...       ...          ...
22785  49  46  49  6             24         6    i == N//2
22788  49  47  49  5             24         4    i == N//2
22789  49  47  49  6             24         6    i == N//2
22790  49  48  49  5             24         4    i == N//2
22791  49  48  49  6             24         6    i == N//2

       All Capacities Same
2                    False
3                    False
4                    False
5                    False
6                     True
...                    ...
22785                False
22788                False
22789                False
22790                 True
22791                 True

[14734 rows x 8 columns]
        N   q   p  n  Optimal Index  Capacity   Idx Relation  \
0      10   2   3  5              4        15  i == N//2 - 1
1      10   2   3  6              4        21  i == N//2 - 1
58     11   3   7  5              4         5  i == N//2 - 1
```

```
59      11   3    7  6                 4          6  i == N//2 - 1
62      11   3   10  5                 4          3  i == N//2 - 1
...     ..  ..   .. ..               ...        ...            ...
22303   49  21   47  6                23          4  i == N//2 - 1
22378   49  23   49  5                23          3  i == N//2 - 1
22379   49  23   49  6                23          4  i == N//2 - 1
22380   49  24   25  5                23          8  i == N//2 - 1
22381   49  24   25  6                23         12  i == N//2 - 1

       All Capacities Same
0                    False
1                    False
58                   False
59                   False
62                   False
...                    ...
22303                False
22378                False
22379                False
22380                False
22381                False

[1114 rows x 8 columns]
```

C: INVESTIGATE Q == 2

```
def constraint_c(N, q, p, n):
    return False

#run for many configurations
df_runs_c = run_configurations(range(10,50),
[2],range(1,200),range(11,13), constraint_c,
"/kaggle/working/runs_q_equal_2.csv")

print(df_runs_c)
```

```
       N  q   p   n  Optimal Index  Capacity   Idx Relation  \
0     10  2   1  11              5      1804      i == N//2
1     10  2   1  12              5      3741      i == N//2
2     10  2   3  11              4       416  i == N//2 - 1
3     10  2   3  12              4       626  i == N//2 - 1
4     10  2   5  11              5       100      i == N//2
...   .. ..  ..  ..            ...       ...            ...
1195  49  2  45  12             24         1      i == N//2
1196  49  2  47  11             24         1      i == N//2
1197  49  2  47  12             24         1      i == N//2
1198  49  2  49  11             24         1      i == N//2
1199  49  2  49  12             24         1      i == N//2
```

```
      All Capacities Same
0                   False
1                   False
2                   False
3                   False
4                   False
...                   ...
1195                 True
1196                 True
1197                 True
1198                 True
1199                 True

[1200 rows x 8 columns]

#perform test to check relations
df_check_c = df_runs_c[df_runs_c["Idx Relation"] == "i == N//2"]
print(df_check_c)

df_check_c = df_runs_c[df_runs_c["Idx Relation"] == "i == N//2 - 1"]
print(df_check_c)

       N  q   p   n  Optimal Index  Capacity Idx Relation  All
Capacities Same
0     10  2   1  11              5      1804    i == N//2
False
1     10  2   1  12              5      3741    i == N//2
False
4     10  2   5  11              5       100    i == N//2
False
5     10  2   5  12              5       149    i == N//2
False
6     10  2   7  11              5        25    i == N//2
False
...   .. ..  ..  ..            ...       ...          ...
...
1195  49  2  45  12             24         1    i == N//2
True
1196  49  2  47  11             24         1    i == N//2
True
1197  49  2  47  12             24         1    i == N//2
True
1198  49  2  49  11             24         1    i == N//2
True
1199  49  2  49  12             24         1    i == N//2
True

[1127 rows x 8 columns]
      N  q   p   n  Optimal Index  Capacity   Idx Relation  \
```

```
2     10  2   3  11               4      416  i == N//2 - 1
3     10  2   3  12               4      626  i == N//2 - 1
28    12  2   7  11               5       35  i == N//2 - 1
29    12  2   7  12               5       45  i == N//2 - 1
30    12  2   9  11               5       13  i == N//2 - 1
..    .. ..  ..  ..             ...      ...            ...
537   34  2   9  12              16       78  i == N//2 - 1
616   36  2  19  11              17       11  i == N//2 - 1
617   36  2  19  12              17       12  i == N//2 - 1
619   36  2  21  12              17       10  i == N//2 - 1
771   40  2  21  12              19       12  i == N//2 - 1

     All Capacities Same
2                  False
3                  False
28                 False
29                 False
30                 False
..                   ...
537                False
616                False
617                False
619                False
771                False

[73 rows x 8 columns]
```

D: Check for |p-q| =1

```
def constraint_d(N,q,p,n):
    return np.abs(p-q) != 1

#run for many configurations
df_runs_d = 
run_configurations(range(10,50),range(1,50),range(1,50),range(5,6),
constraint_d, "/kaggle/working/runs_pq_difference_N.csv")

print(df_runs_d)
```

```
       N   q   p  n  Optimal Index  Capacity   Idx Relation  \
0     10   1   2  5              5         6      i == N//2
1     10   2   1  5              5        26      i == N//2
2     10   2   3  5              4        15  i == N//2 - 1
3     10   3   2  5              4        25  i == N//2 - 1
4     10   3   4  5              5        14      i == N//2
...   ..  ..  .. ..            ...       ...            ...
2275  49  47  46  5              3         8   i = 3 N = 49
2276  49  47  48  5              1         6   i = 1 N = 49
```

```
2277   49   48   47   5              2          8   i = 2 N = 49
2278   49   48   49   5             24          4      i == N//2
2279   49   49   48   5              1          8   i = 1 N = 49

      All Capacities Same
0                   False
1                   False
2                   False
3                   False
4                   False
...                   ...
2275                False
2276                False
2277                False
2278                 True
2279                False

[2280 rows x 8 columns]
```

```
# check
df_check_d = df_runs_d[df_runs_d["Idx Relation"] == "i == N//2"]
df_check_d_idx = df_runs_d[df_runs_d["Idx Relation"] == "i == N//2 -
1"]

print(df_check_d)
print(df_check_d_idx)
```

```
        N    q    p   n   Optimal Index   Capacity Idx Relation   All
Capacities Same
0      10    1    2   5               5          6     i == N//2
False
1      10    2    1   5               5         26     i == N//2
False
4      10    3    4   5               5         14     i == N//2
False
6      10    4    5   5               5         10     i == N//2
False
7      10    5    4   5               5         14     i == N//2
False
...    ..   ..   .. ..             ...        ...           ...
...
2228   49   23   24   5              24         12     i == N//2
False
2229   49   24   23   5              24         14     i == N//2
False
2231   49   25   24   5              24         14     i == N//2
False
2233   49   26   25   5              24          8     i == N//2
True
2278   49   48   49   5              24          4     i == N//2
```

```
True

[1012 rows x 8 columns]
       N   q   p  n  Optimal Index  Capacity   Idx Relation  \
2     10   2   3  5              4        15  i == N//2 - 1
3     10   3   2  5              4        25  i == N//2 - 1
5     10   4   3  5              4        15  i == N//2 - 1
11    10   7   6  5              4         8  i == N//2 - 1
26    11   5   6  5              4         8  i == N//2 - 1
...   ..  ..  .. ..            ...       ...            ...
2139  48  26  25  5             23         8  i == N//2 - 1
2206  49  12  13  5             23        15  i == N//2 - 1
2209  49  14  13  5             23        15  i == N//2 - 1
2230  49  24  25  5             23         8  i == N//2 - 1
2235  49  27  26  5             23         8  i == N//2 - 1

      All Capacities Same
2                   False
3                   False
5                   False
11                  False
26                  False
...                   ...
2139                False
2206                False
2209                False
2230                False
2235                False

[101 rows x 8 columns]
```

```
# # run for many configurations to check if n same
# #define count to keep track when n does matter
# count = 0

# for N in range(5,30):
#     for q in range(1,30):
#         for p in range(1,30):
#             df_runs_d_single = run_configurations([N],[q],
[p],range(10,30), constraint_d, "/kaggle/working/runs_d_single.csv")
#             if df_runs_d_single.shape[0] > 0:
#                 if df_filtered['Optimal Index'].nunique() != 1:
#                     count += 1
# print(count)
```

F: P == 1 --> WORKS!!!

```
def constraint_f(N,q,p,n):
    return False
```

PAS OP DEZE DUURT LANG!!!!

```
#run for many configurations
df_runs_f = run_configurations(range(2,100),range(1,100),
[1],range(1,20), constraint_f, "/kaggle/working/runs_p_equal_1.csv")

print(df_runs_f)
```

```
        N   q  p   n  Optimal Index  Capacity Idx Relation  \
0       2   1  1   1              1         1    i == N//2
1       2   1  1   2              1         3    i == N//2
2       2   1  1   3              1         4    i == N//2
3       2   1  1   4              1         9    i == N//2
4       2   1  1   5              1        14    i == N//2
...    ..  .. ..  ..            ...       ...          ...
94026  99  99  1  15             49     32767    i == N//2
94027  99  99  1  16             49     65535    i == N//2
94028  99  99  1  17             49    131071    i == N//2
94029  99  99  1  18             49    262143    i == N//2
94030  99  99  1  19             49    524287    i == N//2

       All Capacities Same
0                     True
1                    False
2                    False
3                    False
4                    False
...                    ...
94026                False
94027                False
94028                False
94029                False
94030                False

[94031 rows x 8 columns]
```

```
# check if for all values of q and p = 1 it holds that the optimal idx
is N//2

df_check_f = df_runs_d[df_runs_d["Idx Relation"] != "i == N//2"]
print(df_check_f)
```

```
Empty DataFrame
Columns: [N, q, p, n, Optimal Index, Capacity, Idx Relation, All
```

```
Capacities Same]
Index: []
```

E: Investigate role of n

```
def run_configurations_n(N_values, q_values, p_values, n_values,
constraint, output_path):
    results = []

    for N in N_values:
        for q in q_values:
            for p in p_values:
                for n in n_values:
                    # skip invalid configs
                    if p > N or q > N:
                        continue

                    # Check if both p and q are prime
                    if not (is_prime(p) and is_prime(q)):  # Ensure
both p and q are prime
                        continue

                    # Check if constraint
                    if constraint(N, q, p, n):
                        continue

                    # Generate the matrix and find optimal capacity
                    Dn = generate_matrix(N, q, p, n)
                    opt_index, cap, all_same, same_value =
find_optimal_capacity(Dn)
                    rel = get_index_relation(opt_index, N)

                    # Add the results to the list
                    results.append({
                        "N": N,
                        "q": q,
                        "p": p,
                        "n": n,
                        "Optimal Index": opt_index,
                        "Capacity": cap,
                        "Idx Relation": rel,
                        "All Capacities Same": all_same,
                    })

    # Convert the results into a DataFrame
    df = pd.DataFrame(results)

    # Group by 'N', 'q', 'p' and check if all Optimal Indices are the
```

```
same within each group (ignoring 'n')
    def check_optimal_indices_same(group):
        return group['Optimal Index'].nunique() == 1  # Return True if
all values are the same

    # Group by the combination of 'N', 'q', 'p'
    grouped = df.groupby(['N', 'q', 'p'])

    # Use transform to align the results with the original DataFrame
    df['All Optimal Indices Same'] = grouped['Optimal
Index'].transform(lambda x: x.nunique() == 1)

    # Filter out rows where the optimal indices are the same
    df_filtered = df[df['All Optimal Indices Same'] == False]

    # Save the filtered DataFrame to a CSV file
    df_filtered.to_csv(output_path, index=False)

    return df_filtered

def constraint_e(N,q,p,n):
    # if (np.abs(p-q) <= N//2) and (n >= N):
    #     return False
    if n >= N:
        return False
    else:
        return True

#run for many configurations
df_runs_e =
run_configurations_n(range(3,20),range(1,20),range(1,20),range(5,6),
constraint_e, "/kaggle/working/runs_influence_n.csv")

print(df_runs_e)
```

```
Empty DataFrame
Columns: [N, q, p, n, Optimal Index, Capacity, Idx Relation, All
Capacities Same, All Optimal Indices Same]
Index: []
```

C.2. Code to Confirm Mapping of Theorem 4.2.1

```
# This Python 3 environment comes with many helpful analytics
libraries installed
# It is defined by the kaggle/python Docker image:
https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/"
directory
# For example, running this (by clicking run or pressing Shift+Enter)
will list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/)
that gets preserved as output when you create a version using "Save &
Run All"
# You can also write temporary files to /kaggle/temp/, but they won't
be saved outside of the current session

import random
import numpy as np
import math
```

Create Set C_a

```
def generate_binary_vectors(k, n):
    return [[random.choice([0, 1]) for _ in range(n)] for _ in
range(k)]

def calculate_v(x, a, p, q):
    v = [0] * (len(x)+1)
    v[0] = a
    for i in range(1, len(x)+1):
        if x[i-1] == 1:
            v[i] = v[i-1] + p
        else:
            v[i] = max(0, v[i-1] - q)
    return v

def create_vectors(k, n, a, p, q, N):
    binary_vectors = generate_binary_vectors(k, n)
    C_a = set()
```

```
    for x in binary_vectors:
        # if tuple(x) in C_a:
        #     continue
        v = calculate_v(x, a, p, q)

        # Check condition 1: all values of v must be <= N
        if all(vi <= N for vi in v):
            # Check condition 2: the last value of v must be <= u
            if v[-1] <= a:
                C_a.add(tuple(x))  # Add x as a tuple (to be hashable)
to set C_u
                #print(x,v, "ca")

    return C_a
```

Create C_a+1

```
def weighted_running_digital_sum(x, p, q):
    t = [0] * len(x)
    for i in range(1, len(x) + 1):
        t[i-1] = -q * i + (p + q) * sum(x[:i])
    return t

def find_h1(t, N,n,alpha):
    hx = -1
    for i in range(0,min(n,alpha)):
        if t[i] == 1:
            hx = i+1 # The index should be 1-based
            break
    return hx if hx != -1 else len(t)

def find_h2(t, N,n,alpha):
    hx = -1
    for i in range(0,min(n,alpha)):
        if t[i] == 2:
            hx = i+1 # The index should be 1-based
            break
    return hx if hx != -1 else -1


# def apply_mapping_to_vectors(C_a, p, q, N,n,a):
#     C_a_plus_1 = list()

#     for x in C_a:
#             # Compute the weighted running digital sum for the
reversed vector
#         t_seq = weighted_running_digital_sum(x, p, q)
```

```
#         t_rev = weighted_running_digital_sum(x[::-1],p,q)
#         vx = calculate_v(x, a, p, q)
#             # Determine hx using the reversed vector
#         alpha = N - vx[-1]
#         hx = find_h2(t_seq, N,n, alpha)

#             # Decompose x into u and w
#         x_a = x[:hx]
#         x_b = x[hx:]


#         t_xb = weighted_running_digital_sum(x_b,p,q)
#         vxb = calculate_v(x_b, a+1, p, q)
#             # Determine hx using the reversed vector
#         alpha = N - vxb[-1]
#         hx_b = find_h1(t_xb, N,len(x_b), alpha)
#         u_b = x_b[:hx_b]
#         w_b = x_b[hx_b:]

#             # Apply the transformation: y = (w^R, a)
#         y = x_a + w_b + u_b[::-1]  # Reverse w and concatenate with
u
#         C_a_plus_1.append(list(y))  # Add to C_a+1 as a tuple

#     return C_a_plus_1

# def apply_mapping_to_vectors(C_a, p, q, N,n,a):
#     C_a_plus_1 = list()

#     for x in C_a:
#             # Compute the weighted running digital sum for the
reversed vector
#         t_seq = weighted_running_digital_sum(x, p, q)
#         t_rev = weighted_running_digital_sum(x[::-1],p,q)
#         vx = calculate_v(x, a, p, q)
#             # Determine hx using the reversed vector
#         alpha2 =n
#         alpha1 = math.ceil((N - vx[-1])/2)
#         hx2 = find_h2(t_seq, N,n, alpha2)
#         if hx2 != -1:
#             continue
#         #print(x)
#         # hx = find_h1(t_seq, N,n, alpha1)
#         #     # Decompose x into u and w
#         # x_a = x[:hx]
#         # x_b = x[hx:]


#         # t_xb = weighted_running_digital_sum(x_b,p,q)
#         # vxb = calculate_v(x_b, a+1, p, q)
```

```
#         #     # Determine hx using the reversed vector
#         # alpha = N - vxb[-1]
#         # hx_b = find_h1(t_xb, N,len(x_b), alpha)
#         # u_b = x_b[:hx_b]
#         # w_b = x_b[hx_b:]

#             # Apply the transformation: y = (w^R, a)
#         #y = x_b + x_a[::-1]  # Reverse w and concatenate with u
#         y = x[::-1]
#         C_a_plus_1.append(list(y))  # Add to C_a+1 as a tuple

    # return C_a_plus_1
```

a = oneven, idx 2

```
def apply_mapping_to_vectors(C_a, p, q, N,n,a):
    C_a_plus_1 = list()

    for x in C_a:
            # Compute the weighted running digital sum for the
reversed vector
        t_seq = weighted_running_digital_sum(x, p, q)
        t_rev = weighted_running_digital_sum(x[::-1],p,q)
        vx = calculate_v(x, a, p, q)
            # Determine hx using the reversed vector
        alpha2 = N - vx[-1]
        alpha1 = math.ceil((N - vx[-1])/2)
        hx2 = find_h2(t_seq, N,n, alpha2)
        # if hx2 != -1:
        #     continue
        # #print(x)
        # hx = find_h1(t_seq, N,n, alpha1)
        #     # Decompose x into u and w
        x_a = x[:hx2]
        x_b = x[hx2:]


        # t_xb = weighted_running_digital_sum(x_b,p,q)
        # vxb = calculate_v(x_b, a+1, p, q)
        #     # Determine hx using the reversed vector
        # alpha = N - vxb[-1]
        # hx_b = find_h1(t_xb, N,len(x_b), alpha)
        # u_b = x_b[:hx_b]
        # w_b = x_b[hx_b:]

            # Apply the transformation: y = (w^R, a)
        y = x_b + x_a[::-1]  # Reverse w and concatenate with u
        # y = x[::-1]
```

```
        C_a_plus_1.append(list(y))  # Add to C_a+1 as a tuple

    return C_a_plus_1
```

Check conditions

```
def check_conditions(C_a_plus_1, a, N, p, q):
    valid_vectors = []

    for y in C_a_plus_1:
        #print(y)
        v = calculate_v(list(y), (a+1), p, q)
        #print(v)
        # Condition 1: Check if each value of v <= N
        if all(vi <= N for vi in v):
            # Condition 2: Check if the last value of v <= u + 1
            if v[-1] <= a + 1:
                valid_vectors.append(list(y))  # Add to valid vectors
if both conditions hold
            else:
                print(y, v, "no a")
        else:
            print(y,v, "no v")

    return len(valid_vectors) == len(C_a_plus_1)

def check_unique(C_a_plus_1):
    # Check if all vectors in C_u+1 are unique by comparing length
with a set
    unique_vectors = list(set(tuple(v) for v in C_a_plus_1))
    return len(C_a_plus_1) == len(unique_vectors)
```

P = 2

```
C_a = [
    [1,1, 0, 1, 0],  # First custom vector
    [0, 1, 1,0, 0],
    [0,1,1,0,1,1,0,0]# Second custom vector
]

C_a_1 = apply_mapping_to_vectors(C_a, 2, 3, 5,5,1)

print(C_a_1)
print(check_conditions(C_a_1, 1, 5, 2, 3))
print(check_unique(C_a_1))
```

```python
C_a = [
    [1,1, 1, 0, 1,0],  # First custom vector
    [0, 1, 1,1,1, 0],
    [1,1,1,0,0,1]# Second custom vector
]

C_a_1 = apply_mapping_to_vectors(C_a, 2, 5, 9,6,3)

print(C_a_1)
print(check_conditions(C_a_1, 3, 9, 2, 5))
print(check_unique(C_a_1))
```

P = 1

Manual Example

```python
C_a = [
    [1, 0, 0, 0, 1, 0],  # First custom vector
    [1, 0, 1, 0, 0, 0],  # Second custom vector
]

C_a_1 = apply_mapping_to_vectors(C_a, 1, 3, 5,6,1)

print(C_a_1)
print(check_conditions(C_a_1, 1, 5, 1, 3))
print(check_unique(C_a_1))
```

Run for many p = 2

```python
k = 100

for n in range(6,20):
    for q in range(3,20):
        if q % 2 == 0:
            continue
        for N in range(5,20):
            if N%2 == 0:
                continue
            a = N//2-1
            if a % 2 == 0:
                continue
            p = 2
            C_a = create_vectors(k, n, a, p, q, N)
            C_a_1 = apply_mapping_to_vectors(C_a, p, q, N,n,a)
```

```
            check_cond = check_conditions(C_a_1, a, N, p, q)
            if check_cond == False:
                #print(C_a)
                print(check_conditions, q,N,a)
            check_uniq = check_unique(C_a_1)
            if check_uniq == False:
                for i in range(len(C_a_1)):
                    for j in range(i+1,len(C_a_1)):
                        if C_a_1[i] == C_a_1[j]:
                            print(C_a_1[i])
                print(check_uniq,q,N,a)
```

Example Run

```
# k = 1000  # number of binary vectors
# n = 6  # length of each binary vector
# a = 2  # starting value for v
# p = 1  # increment when xi = 1
# q = 3  # decrement when xi = 0
# N = 6  # maximum value for any v_i

# C_a = create_vectors(k, n, a, p, q, N)

# C_a_1 = apply_mapping_to_vectors(C_a, p, q, N)

# print(check_conditions(C_a_1, a, N, p, q))
# print(check_unique(C_a_1))
```

Run for many samples

--> CONCLUSION: WORKS! MAPPING IS BOTH IN C_A+1 AND INJECTIVE

```
k = 100




for n in range(6,20):
    for q in range(3,50):
        for N in range(5,50):
            a = N//2-1
```

```
            p = 1
            C_a = create_vectors(k, n, a, p, q, N)

            C_a_1 = apply_mapping_to_vectors(C_a, p, q, N,n,a)

            check_cond = check_conditions(C_a_1, a, N, p, q)
            if check_cond == False:
                print(check_conditions, n,q,N,a)
            check_uniq = check_unique(C_a_1)
            if check_uniq == False:
                print(check_uniq,n,q,N,a)

for n in range(6,20):
    for q in range(5,50):
        for N in range(5,50):
            a = N//2-2
            p = 1
            C_a = create_vectors(k, n, a, p, q, N)

            C_a_1 = apply_mapping_to_vectors(C_a, p, q, N,n,a)

            check_cond = check_conditions(C_a_1, a, N, p, q)
            if check_cond == False:
                print(check_conditions, n,q,N,a)
            check_uniq = check_unique(C_a_1)
            if check_uniq == False:
                print(check_uniq,n,q,N,a)

for n in range(5,20):
    for q in range(3,50):
        for N in range(5,50):
            a = N//2-3
            p = 1
            C_a = create_vectors(k, n, a, p, q, N)

            C_a_1 = apply_mapping_to_vectors(C_a, p, q, N,n,a)

            check_cond = check_conditions(C_a_1, a, N, p, q)
            if check_cond == False:
                print(check_conditions, n,q,N,a)
            check_uniq = check_unique(C_a_1)
            if check_uniq == False:
                print(check_uniq,n,q,N,a)
```

C.3. Code for Matrix Generation and Plotting

```
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
```

FUNCTIONS

```
def generate_matrix(N, q, p, n):
    # Step 1: Build D matrix
    D = np.zeros((N + 1, N + 1), dtype=int)

    for i in range(N + 1):
        if i <= N - p:
            D[i][i + p] = 1
        if i >= q:
            D[i][i - q] = 1
        if i < q:
            D[i][0] = 1

    # Step 2: Compute D^n
    Dn = np.linalg.matrix_power(D, n)

    return D, Dn

def find_optimal_capacity(Dn):
    N = Dn.shape[0] - 1
    half_index = N // 2  # Go only up to N//2

    # Calculate capacities for indices from 0 to N//2
    capacities = [np.sum(Dn[i, :i + 1]) for i in range(half_index +
1)]

    # Check if capacities are non-decreasing
    is_non_decreasing = all(capacities[i] <= capacities[i + 1] for i
in range(len(capacities) - 1))

    # Find the maximum capacity
    max_cap = max(capacities)

    # Find all indices with the maximum capacity, only within the
range 0 to N//2
    indices_with_max_cap = [i for i, cap in enumerate(capacities) if
cap == max_cap]

    # Return the highest index with the maximum capacity
    opt_index = max(indices_with_max_cap)

    return opt_index, max_cap, is_non_decreasing
```

```
def generate_example(N, q, p, n):
    D, Dn = generate_matrix(N,q,p,n)
    idx_dn, cap_dn, increasing_dn = find_optimal_capacity(Dn)
    print(D)
    print(Dn)
    print("Idx, Cap, Increasing:", idx_dn, cap_dn, increasing_dn)
```

Q = 1 EXAMPLE

```
generate_example(4,1,3,6)

generate_example(4,2,3,6)
```

P = 1 EXAMPLE

```
generate_example(4,2,1,6)

# --> N = 4, Q = 2, P = 1, n = 4 WORKS

generate_example(4,2,3,6)

# we see that it would not work for q = 2 (since N is even!) so the
determining factor really is that p = 1

# --> N = 4, Q = 2, P = 3, n = 4 WORKS NOT (so from this we can find a
sequence where it does not work anymore)
```

P = 1 EXTRA EXAMPLE

```
generate_example(6,5,1,7)

# --> WORKS

generate_example(6,5,2,7)

# --> WORKS NOT ANYMORE BEC P NOT 1
```

P = 1 EXTRA EXTRA EXAMPLE

```
generate_example(6,3,1,10)

generate_example(6,3,2,10)
```

Q = 2, N = ODD

```
generate_example(7,2,3,4)

# --> N = 5, Q = 2, P = 5, n = 4 WORKS!

[[1 0 0 1 0 0 0 0]
 [1 0 0 0 1 0 0 0]
 [1 0 0 0 0 1 0 0]
 [0 1 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 1]
 [0 0 0 1 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 0 0 1 0 0]]
[[3 1 2 2 2 0 1 2]
 [3 1 0 4 2 0 1 0]
 [3 1 0 1 4 0 1 0]
 [3 1 0 1 0 4 1 0]
 [1 3 0 1 0 0 3 0]
 [1 0 2 1 0 0 0 2]
 [1 0 0 3 0 0 0 0]
 [1 0 0 0 2 0 0 0]]
Idx, Cap, Increasing: 3 5 True

generate_example(4,2,3,6)


# --> Reason: N now even

[[1 0 0 1 0]
 [1 0 0 0 1]
 [1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]]
[[8 3 1 5 2]
 [7 3 1 4 2]
 [5 2 1 3 1]
 [4 2 1 3 1]
 [3 1 1 2 1]]
Idx, Cap, Increasing: 1 10 False

generate_example(5,4,3,6)

# --> Reason: Q not 2

[[1 0 0 1 0 0]
 [1 0 0 0 1 0]
 [1 0 0 0 0 1]
 [1 0 0 0 0 0]
 [1 0 0 0 0 0]
 [0 1 0 0 0 0]]
```

```
[[13  0  0  8  0  0]
 [13  0  0  8  0  0]
 [13  0  0  8  0  0]
 [ 8  0  0  5  0  0]
 [ 8  0  0  5  0  0]
 [ 8  0  0  5  0  0]]
Idx, Cap, Increasing: 2 13 True
```

```
generate_example(6,3,5,6)
```

```
[[1 0 0 0 0 1 0]
 [1 0 0 0 0 0 1]
 [1 0 0 0 0 0 0]
 [1 0 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 0 0 1 0 0 0]]
[[6 0 3 0 0 4 0]
 [6 0 3 0 0 4 0]
 [4 0 2 0 0 3 0]
 [4 0 2 0 0 3 0]
 [4 0 2 0 0 3 0]
 [3 0 1 0 0 2 0]
 [3 0 1 0 0 2 0]]
Idx, Cap, Increasing: 3 6 True
```

```
# !!!!!! NOG EVEN CHECKEN: 6,3,5,4 (BEIDE CONSTRAINTS NIET SATISFIED!)
!!!!!!

generate_example(5,3,5,4)
```

PLOT GRAPHS

```
def plot_example(x, y, N, filename):
    # Create the plot
    plt.figure(figsize=(8, 6))
    plt.plot(x, y)
    plt.axhline(y=N, color='r', linestyle='--', label=f'y = {N}')

    # Add labels and title
    plt.xlabel('Time')
    plt.ylabel('Value of TA-Sequence')
    plt.title('Plot of TA Sequence')
    plt.grid(True)


    # Save the figure
    plt.savefig(filename)
    plt.close()
```

```
# # Create plot for p = 1 example
#N=4,q=2,p=1,n=6
N = 4
t1 = np.arange(0,7)
x1 = [1,2,3,4,2,0,1]
y1 = [1,3,4,2,0,1,2]

plot_example(t1, x1, 4, "plot_x1.png")
plot_example(t1, y1, 4, "plot_y1.png")

# # Create plot for p = 1 example
#N=4,q=2,p=1,n=10
N = 4
t2 = np.arange(0,10)
x2 = [0,1,0,1,2,3,4,2,0,1]
y2 = [3,1,0,1,2,3,4,2,3,1]

plot_example(t2, x2, 4, "plot_x2.png")
plot_example(t2, y2, 4, "plot_y2.png")
```

EXAMPLE ARTICLE TO SHOW P and Q NOT SYMMETRIC

```
generate_example(7,4,3,5)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_31/2001573701.py in <cell line: 0>()
----> 1 generate_example(7,4,3,5)

NameError: name 'generate_example' is not defined

generate_example(7,3,4,5)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_31/2033262758.py in <cell line: 0>()
----> 1 generate_example(7,3,4,5)

NameError: name 'generate_example' is not defined
```

C.4. Code to Create C_a

```python
import numpy as np
import math
import random
from collections import defaultdict
```

Generate Matrix

```python
def generate_matrix(N, q, p, n):
    # Step 1: Build D matrix
    D = np.zeros((N + 1, N + 1), dtype=int)

    for i in range(N + 1):
        if i <= N - p:
            D[i][i + p] = 1
        if i >= q:
            D[i][i - q] = 1
        if i < q:
            D[i][0] = 1

    # Step 2: Compute D^n
    Dn = np.linalg.matrix_power(D, n)

    return D, Dn


def find_optimal_capacity(Dn):
    N = Dn.shape[0] - 1
    half_index = N // 2  # Go only up to N//2

    # Calculate capacities for indices from 0 to N//2
    capacities = [np.sum(Dn[i, :i + 1]) for i in range(half_index +
1)]

    # Check if capacities are non-decreasing
    is_non_decreasing = all(capacities[i] <= capacities[i + 1] for i
in range(len(capacities) - 1))

    # Find the maximum capacity
    max_cap = max(capacities)

    # Find all indices with the maximum capacity, only within the
range 0 to N//2
    indices_with_max_cap = [i for i, cap in enumerate(capacities) if
cap == max_cap]

    # Return the highest index with the maximum capacity
    opt_index = max(indices_with_max_cap)

    return opt_index, max_cap, is_non_decreasing
```

```
def generate_example(N, q, p, n):
    D, Dn = generate_matrix(N,q,p,n)
    idx_dn, cap_dn, increasing_dn = find_optimal_capacity(Dn)
    print(D)
    print(Dn)
    print("Idx, Cap, Increasing:", idx_dn, cap_dn, increasing_dn)
```

Create C_a

```
def generate_binary_vector(n):
    return [random.choice([0, 1]) for _ in range(n)]

def calculate_v(x, a, p, q):
    v = [0] * (len(x) + 1)
    v[0] = a
    for i in range(1, len(x) + 1):
        if x[i - 1] == 1:
            v[i] = v[i - 1] + p
        else:
            v[i] = max(0, v[i - 1] - q)
    return v

def create_CA(M, n, a, p, q, N):
    grouped_vectors = defaultdict(list)
    seen = set()

    while sum(len(vectors) for vectors in grouped_vectors.values()) <
M:
        x = generate_binary_vector(n)
        x_tuple = tuple(x)
        if x_tuple in seen:
            continue

        v = calculate_v(x, a, p, q)
        if all(vi <= N for vi in v) and v[-1] <= a:
            grouped_vectors[v[-1]].append(x_tuple)
            seen.add(x_tuple)

    # Convert to sorted dict
    sorted_grouped_vectors = {
        key: sorted(grouped_vectors[key])
        for key in sorted(grouped_vectors.keys())
    }

    return sorted_grouped_vectors
```

EXAMPLES Q = 2 N = ODD

N = 7, P = 3

```
# N = 7, q = 2, p = 3, n = 5
generate_example(7,2,3,5)

# N = 7, q = 2, p = 3, n = 5
# create_CA(M, n, a, p, q, N)

Ca_1 = create_CA(8,5,2,3,2,7)
Ca_2 = create_CA(9,5,3,3,2,7)

print(Ca_1)
print(Ca_2)
```

N = 5, P = 3

```
# N = 5, q = 2, p = 3, n = 5
generate_example(5,2,3,6)

# N = 5, q = 2, p = 3, n = 5
# create_CA(M, n, a, p, q, N)

Ca_1 = create_CA(11,6,1,3,2,5)
Ca_2 = create_CA(11,6,2,3,2,5)

print(Ca_1)
print(Ca_2)
```

N = 9, P = 3

```
# 9 = 5, q = 2, p = 3, n = 5
generate_example(9,2,3,8)
```

```
[[1 0 0 1 0 0 0 0 0 0]
 [1 0 0 0 1 0 0 0 0 0]
 [1 0 0 0 0 1 0 0 0 0]
 [0 1 0 0 0 0 1 0 0 0]
 [0 0 1 0 0 0 0 1 0 0]
 [0 0 0 1 0 0 0 0 1 0]
 [0 0 0 0 1 0 0 0 0 1]
 [0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 1 0 0]]
```

```
[[30 15  6 20 27 10 20  9  5 16]
 [29 15  6 20  8 26 20  9  5  4]
 [21 23  6 20  8  5 36  9  5  4]
 [21  5 21 20  8  5  5 34  5  4]
 [20  5  2 35  8  5  5  3 21  4]
 [20  5  2  6 32  5  5  3  0 20]
 [18  5  2  6  2 29  5  3  0  1]
 [ 6 12  2  5  2  0 20  3  0  1]
 [ 6  1 11  5  2  0  1 18  0  1]
 [ 5  1  0 12  2  0  1  0  8  1]]
Idx, Cap, Increasing: 4 70 True

# N = 9, q = 2, p = 3, n = 5
# create_CA(M, n, a, p, q, N)
Ca_1 = create_CA(30,8,0,3,2,9)
Ca_2 = create_CA(34,8,1,3,2,9)

print(Ca_1)
print(Ca_2)

{0: [(0, 0, 0, 0, 0, 0, 0, 0), (0, 0, 0, 0, 0, 1, 0, 0), (0, 0, 0, 0,
1, 0, 0, 0), (0, 0, 0, 1, 0, 0, 0, 0), (0, 0, 0, 1, 0, 1, 0, 0), (0,
0, 0, 1, 1, 0, 0, 0), (0, 0, 1, 0, 0, 0, 0, 0), (0, 0, 1, 0, 0, 1, 0,
0), (0, 0, 1, 0, 1, 0, 0, 0), (0, 0, 1, 1, 0, 0, 0, 0), (0, 1, 0, 0,
0, 0, 0, 0), (0, 1, 0, 0, 0, 1, 0, 0), (0, 1, 0, 0, 1, 0, 0, 0), (0,
1, 0, 1, 0, 0, 0, 0), (0, 1, 1, 0, 0, 0, 0, 0), (1, 0, 0, 0, 0, 0, 0,
0), (1, 0, 0, 0, 0, 1, 0, 0), (1, 0, 0, 0, 1, 0, 0, 0), (1, 0, 0, 1,
0, 0, 0, 0), (1, 0, 0, 1, 0, 1, 0, 0), (1, 0, 0, 1, 1, 0, 0, 0), (1,
0, 1, 0, 0, 0, 0, 0), (1, 0, 1, 0, 0, 1, 0, 0), (1, 0, 1, 0, 1, 0, 0,
0), (1, 0, 1, 1, 0, 0, 0, 0), (1, 1, 0, 0, 0, 0, 0, 0), (1, 1, 0, 0,
0, 1, 0, 0), (1, 1, 0, 0, 1, 0, 0, 0), (1, 1, 0, 1, 0, 0, 0, 0), (1,
1, 1, 0, 0, 0, 0, 0)]}
{0: [(0, 0, 0, 0, 0, 1, 0, 0), (0, 0, 0, 1, 0, 0, 0, 0), (0, 0, 0, 1,
1, 0, 0, 0), (0, 0, 1, 0, 0, 0, 0, 0), (0, 0, 1, 0, 0, 1, 0, 0), (0,
0, 1, 0, 1, 0, 0, 0), (0, 0, 1, 1, 0, 0, 0, 0), (0, 1, 0, 0, 0, 0, 0,
0), (0, 1, 0, 0, 0, 1, 0, 0), (0, 1, 0, 0, 1, 0, 0, 0), (0, 1, 1, 0,
0, 0, 0, 0), (1, 0, 0, 0, 0, 0, 0, 0), (1, 0, 0, 0, 0, 1, 0, 0), (1,
0, 0, 1, 0, 0, 0, 0), (1, 0, 0, 1, 1, 0, 0, 0), (1, 0, 1, 0, 0, 0, 0,
0), (1, 0, 1, 0, 1, 0, 0, 0), (1, 0, 1, 1, 0, 0, 0, 0), (1, 1, 0, 0,
0, 0, 0, 0), (1, 1, 0, 0, 0, 1, 0, 0), (1, 1, 0, 0, 1, 0, 0, 0), (1,
1, 0, 1, 0, 0, 0, 0)], 1: [(0, 0, 0, 0, 0, 0, 1, 0), (0, 0, 0, 1, 0,
0, 1, 0), (0, 1, 0, 0, 0, 0, 1, 0), (0, 1, 0, 1, 0, 0, 1, 0), (0, 1,
0, 1, 0, 1, 0, 0), (0, 1, 0, 1, 1, 0, 0, 0), (0, 1, 1, 0, 0, 0, 1, 0),
(0, 1, 1, 0, 0, 1, 0, 0), (0, 1, 1, 0, 1, 0, 0, 0), (1, 0, 0, 0, 0, 0,
1, 0), (1, 0, 1, 0, 0, 0, 1, 0), (1, 1, 0, 0, 0, 0, 1, 0)]}
```