

Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Master Thesis

Combinatorics of ordered walks on trees and applications to
categories of partitions

submitted by
Sebastian Volz

under supervision of
(Soon Dr.) Nicolas Faroß
Prof. Dr. Moritz Weber

submitted

Reviewers
Prof. Dr. Moritz Weber
Prof. Dr. TODO

Abstract

This thesis investigates the structure and algorithmic properties of the partition category $\Pi_{k \in \mathbb{N}}$, which plays a central role in the theory of easy quantum groups. Categories of partitions are associated with easy quantum groups. T. Banica and R. Speicher first introduced these categories in 2009, and S. Raum together with M. Weber completed their classification in 2016. We prove that the category $\Pi_{k \in \mathbb{N}}$ is related to ordered walks on rooted plane trees and provide insights about the previous research in this area. Ordered walks, as we use them, were introduced by A. Khorunzhy and V. Vengerovsky in the early 2000s as a combinatorial tool for studying spectra of large random matrices and rooted-tree models. With this work as a foundation, we are able to count the number of partitions in the category $\Pi_{k \in \mathbb{N}}$ up to a partition size of 300 in a few seconds. Furthermore, we propose efficient algorithms to decide whether a specific partition is in $\Pi_{k \in \mathbb{N}}$.

Contents

1	Introduction	2
2	Background	3
2.1	Computer Science Background	3
2.1.1	Asymptotic Analysis and Notation	3
2.1.2	Dynamic Programming	4
2.1.3	Rooted Plane Trees	5
2.1.4	Ordered Walks	7
2.2	Mathematical Background	9
2.2.1	Partitions	9
2.2.2	Categories of Partitions	10
2.2.3	Walks on Rooted Plane Trees via Moments of Adjacency Matrices	11
3	The Category $\Pi_{k \in \mathbb{N}}$	14
3.1	Definitions	15
3.2	Relevant Properties	17
4	The connection between Ordered Walks and the Category $\Pi_{k \in \mathbb{N}}$	18
4.1	Simple Walks, Dyck Paths and the Category NC_2	18
4.2	Ordered Walks and the Category $\Pi_{k \in \mathbb{N}}$	19
4.3	The <i>Holy Quaternity</i> - A fourth Definition of $\Pi_{k \in \mathbb{N}}$ arose	22
4.4	Relationship Between Ordered Walks and the Parameter k in $\Pi_{k \in \mathbb{N}}$	24
5	Algorithmic Methods for Analyzing Properties of $\Pi_{k \in \mathbb{N}}$	28
5.1	Applying Category generating Algorithms	28
5.2	Constructing and Validating Elements in $\Pi_{k \in \mathbb{N}}$	30
6	Discussion	32
A	Overview Categories	34

1 Introduction

In this thesis, we study the partition category $\Pi_{k \in \mathbb{N}}$. More precisely, we propose classification algorithms and draw a connection to ordered walks. Categories of partitions are related to easy quantum groups, as defined in [BCS10], with their classification completed in [SR16]. However, in the context of this thesis, we focus on partition problems and not the relation to easy quantum groups. So in our case we see partitions as combinatorial objects, where a partition $p \in P(k, l)$ is a partition of a set $S = \{1, \dots, k + l\}$ into disjoint subsets. In general such a partition can be represented by a row of k upper and l lower points that are partitioned into disjoint subsets by strings. Nevertheless we often refer to partitions only having upper points, where $l = 0$. Those partitions can be visualized as follows

On those partitions we can define operations like a tensor product, an involution, rotations or a composition. A category of partitions is a set of partitions that is closed under these operations and the base partitions $\cap \in P(2, 0)$ and $\cup \in P(1, 1)$. For example, the following set of partitions is the category of all partitions

The category $\Pi_{k \in \mathbb{N}}$ is defined and investigated in [RW13]. It is generated by $\pi_{k \in \mathbb{N}}$, given by k four blocks on $4k$ points of the form

Besides the generator presentation of the category $\Pi_{k \in \mathbb{N}}$, there are also other definitions using a construction by Raum and Weber, where partitions are obtained by merging blocks of non-crossing pair partitions at the same Dyck level while respecting open pair constraints ([RW13]), as well as a characterization by Mang, who defines $\Pi_{k \in \mathbb{N}}$ via the notion of non-interference between blocks and a so called betweenness relation ([Man22]). We will use and investigate those different representations using our findings, including the connection to ordered walks.

In the twentieth century, walks on trees gained prominence in random matrix theory and spectral graph analysis. Wigner first used walk enumeration to study eigenvalue distributions in random matrices (see [Wig55]), an approach further developed by Füredi and Komlós in the study of random graphs in [FK81]. More recently, Khorunzhy introduced explicit recurrence relations for counting walks on rooted plane trees, highlighting their role in the spectral properties of adjacency matrices of random graphs. Furthermore, he defined those kinds of ordered walk that we will focus on in [KV00].

In this thesis, we were able to prove a connection between $\Pi_{k \in \mathbb{N}}$ and ordered walks on rooted plane trees. A rooted plane tree is a combinatorial structure in which each node has an ordered sequence of children. These trees naturally arise in various combinatorial problems, particularly in our study we focus on counting the different combinations of traversing it.

An ordered walk is a way to traverse a rooted plane tree with some additional conditions that need to be met. An ordered walk must start and end at the root of the tree. Furthermore, each edge of the tree is visited an even positive number of times. If there is a choice of where to move next, the walk either follows an already traversed edge or selects the leftmost edge among those not yet visited. A simple walk is an ordered walk that visits every edge in the tree exactly twice.

Theorem 1.1 (Theorem 4.1). *Let W_s be the set of all simple walks and NC_2 be the category generated by the base partitions, then $NC_2 = W_s$.*

Before proving Theorem 1.2, we have to prove Theorem 1.1 as a foundation.

Theorem 1.2 (Theorem 4.14). *Let W be the set of all ordered walks, then $\Pi_{k \in \mathbb{N}} = W$.*

Additionally, we analyzed the impact of the parameter k in Π_k .

Theorem 1.3 (Theorem 4.24). *Π_r is the partition category of all even-nested partitions with a w -depth of at most r .*

Moreover, we draw a connection to ordered walks for a fixed $k \in \mathbb{N}$ in Π_k .

Theorem 1.4 (Theorem 4.26). *The ordered walk w generated by Π_r for a fixed $r \in \mathbb{N}$ contains arbitrarily many crossings between two (not necessarily different) sub-walks p_1 and p_2 , where the total depth of p_1 plus the depth of p_2 must always be less or equal than r .*

By proving Theorem 1.2 in Section 4.2, we were also able to explore the category $\Pi_{k \in \mathbb{N}}$ further and implemented an algorithm that, given a partition p , decides whether $p \in \Pi_{k \in \mathbb{N}}$ with a time complexity of $O(n)$. Additionally, this discovery enabled us to count the category $\Pi_{k \in \mathbb{N}}$ up to a partition size of approximately almost 200 in a few minutes by leveraging the findings on ordered walks presented in [KV00] and dynamic programming.

Overview of the thesis

The structure of this thesis is generally divided into three parts. In the first part in Section 2, we introduce key preliminaries from both computer science (Section 2.1) and mathematics (Section 2.2). Starting with techniques in area of algorithms and data structures for the implementation part of this thesis, such as dynamic programming, moving on to trees and ordered walks.

The mathematical background in this thesis consists mostly of partitions of sets and their categories (Section 2.2.1 and Section 2.2.2), up to a mathematical interpretation of counting the number of ordered walks, where we introduce moments of adjacency matrices of random graphs in Section 2.2.3.

The second part (Section 4) contains the proofs for both Theorem 1.1 and Theorem 1.2, forming the main achievements of this thesis.

Based on the foundation of the second part, the third part (Section 5) describes and analyzes various implementations to classify the category $\Pi_{k \in \mathbb{N}}$, particularly its elements. From applying generating algorithms to efficiently implementing the problem of deciding partitions in $\Pi_{k \in \mathbb{N}}$. Finally, we present a small discussion at the end of this thesis.

2 Background

In this section, we start with some preliminaries regarding both the computer science background background, as well as the mathematical background. First, we define some aspects in the area of computer science oriented techniques, such as Big-O Notation and data structures such as different kinds of trees, followed by mathematical oriented subjects, such as partitions on sets and their categories.

2.1 Computer Science Background

The objective of this section is to establish a foundation in the core concepts used to implement and analyze classification algorithms for the category for the category $\Pi_{k \in \mathbb{N}}$. In particular, we introduce asymptotic runtime notation (Big-O, Omega), dynamic programming techniques, and data structures such as rooted plane trees and ordered walks, which will later be shown to have a direct connection to this category.

2.1.1 Asymptotic Analysis and Notation

To analyze the algorithms presented in this thesis, especially in Section 5, we employ asymptotic analysis and its notation. In particular, we will focus on the so-called "Big-O" notation and "Omega-notation". Further and more detailed information can be found in [Knu76].

Definition 2.1 (Big-O notation). Let $g, f : \mathbb{N} \rightarrow \mathbb{N}$ be functions. We define $O(g)$ as the following set:

$$O(g(n)) = \{f : \mathbb{N} \rightarrow \mathbb{N} \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$$

More precisely $f \in O(g)$ if and only if we can find a c and a n_0 such that for all n greater than n_0 , we satisfy $0 \leq f(n) \leq c \cdot g(n)$.

Remark 2.2. Definition 2.1 intuitively states that g grows "faster" than f . Note, that this leads to the property

$$\limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty.$$

Example 2.3. The following examples apply.

(a) Let $f : \mathbb{N} \rightarrow \mathbb{N}$, $n \mapsto 100$. Then

$$f \in O(1).$$

If $f : \mathbb{N} \rightarrow \mathbb{N}$, $n \mapsto n$. Then

$$f \notin O(1).$$

(b) Let $g : \mathbb{N} \rightarrow \mathbb{N}$, $n \mapsto 100n$. Then

$$g \in O(n).$$

If $g : \mathbb{N} \rightarrow \mathbb{N}$, $n \mapsto n \cdot \log(n)$. Then

$$g \notin O(n).$$

(c) Let $g : \mathbb{N} \rightarrow \mathbb{N}$, $n \mapsto n^k$, where $k \in \mathbb{N}$ and let $f(n) = \sum_{i=0}^k a_i n^i$ be a polynomial of degree at most k . Then

$$f \in O(g).$$

Proof. To prove the statement $f \in O(g)$, we show that

$$\limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty. \quad (1)$$

If we insert the functions, we get

$$\limsup_{n \rightarrow \infty} \frac{\sum_{i=0}^k a_i n^i}{n^k} < \infty. \quad (2)$$

Since $\limsup_{n \rightarrow \infty} \frac{n^l}{n^k} = 0$, for $l < k$, we can reduce the equation to

$$\limsup_{n \rightarrow \infty} \frac{a_k n^k}{n^k} = a_k < \infty. \quad (3)$$

□

Definition 2.4 (Ω -notation). Let $f, g : \mathbb{N} \rightarrow \mathbb{N}$ be functions. We define $\Omega(g(n))$ as the following set:

$$\Omega(g(n)) = \{f \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}$$

The Ω -notation is used to assign an algorithmic problem a lower bound, regarding its runtime or space complexity. Its application is to make predictions about how efficient an algorithm can be in order to solve the underlying problem.

Example 2.5. The problem of determining the maximum integer in a list of length $n \in \mathbb{N}$ has a lower bound time complexity of $\Omega(n)$, as any algorithm must traverse the entire list at least once to ensure the correct identification of the maximum element.

Remark 2.6. Let $k \in \mathbb{N}_0$. The algorithmic runtimes most commonly used are

$$O(n^k), O(n^k \cdot \log(n)), O(k^n), O(n!).$$

2.1.2 Dynamic Programming

In this section, we explain the foundations of *dynamic programming*, in order to utilize this technique in Section 5.1.

The overall principle of dynamic programming is to reuse already computed sub-results instead of recalculating them multiple times. This is achieved by storing sub-results in a data structure such as a table or array. When the same subproblem arises, the stored result is reused, which avoids redundant computation and improves efficiency.

Example 2.7. Consider the problem of computing the Fibonacci sequence up to a certain number n . The naive standard implementation looks as follows:

```

def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

```

For smaller input numbers, this implementation is absolutely fine. Nevertheless, it has exponential complexity because of the line `return fib(n-1) + fib(n-2)`, where each time it is called, we double the number of recursive calls.

Using dynamic programming we get the implementation:

```

fib_array = [-1 for _ in range(n+1)]
def fib(n):
    if fib_array[n] != -1:
        return fib_array[n]
    if n <= 1:
        return 1
    else:
        fib_array[n] = fib(n-1) + fib(n-2)
        return fib_array[n]

```

By storing intermediate results in `fib_array`, we only compute each sub-result once, resulting in a linear complexity.

In the Fibonacci example from above, we used a so called *top-down* approach of dynamic programming. This often involves recursion, building up the result from greater to smaller input values. While this approach is often easier to implement, it usually has the drawback of an increased running time due to the recursion. Each recursive call requires additional computational overhead to manage the call stack, including the allocation of space for function arguments, return addresses, and local variables. To avoid this additional computation, we also introduce the *bottom-up* approach of dynamic programming. This approach avoids recursion by iteratively building intermediate results, starting from the smallest sub-result.

Example 2.8 (Computing the Fibonacci sequence bottom-up).

```

def fib_bottom_up(n):
    fib = [0, 1]
    for i in range(2, n+1):
        fib.append(fib[i-1] + fib[i-2])
    return fib[n]

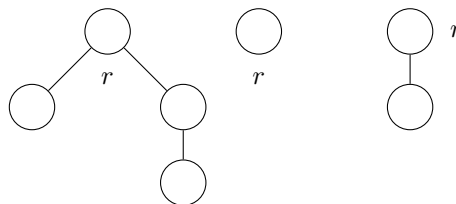
```

2.1.3 Rooted Plane Trees

To establish a connection between *rooted plane trees* and the category $\Pi_{k \in \mathbb{N}}$, we will introduce *ordered walks* on such trees. This connection enables us to investigate further properties of the category $\Pi_{k \in \mathbb{N}}$.

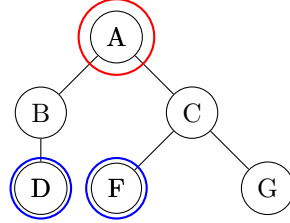
Definition 2.9 (Rooted tree). A *rooted tree* is a connected, acyclic graph $T = (V, E)$ with exactly one distinguished node $r \in V$ called the *root*. For every node $v \in V$, there exists one path from v to r . In other words, a rooted tree is a connected acyclic graph where one vertex is designated as the root, and each node has a unique path to the root.

Example 2.10 (Rooted trees).



Definition 2.11 (Lowest common ancestor). Given a rooted tree with root r , and two vertices u_0 and v_0 , let the sequences u_0, u_1, \dots, u_n, r and v_0, v_1, \dots, v_m, r represent the paths from u and v to the root r , respectively. The *lowest common ancestor (LCA)* of u and v is the vertex w such that w lies on both paths, i.e., $w = u_i = v_j$ for some indices i and j , and w is the farthest vertex from r along the paths that satisfies this condition, meaning i and j are the minimal indices satisfying $u_i = v_j$.

Example 2.12 (LCA). In this example, we can see that the LCA of D and F is the root A .

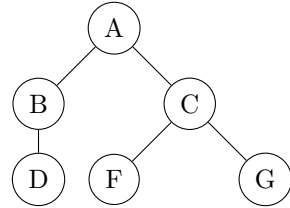


Definition 2.13. Let $T = (V, E)$ be a tree with vertex set V and edge set E . The diameter of T is given by

$$\text{diam}(T) = \max_{u,v \in V} d(u, v)$$

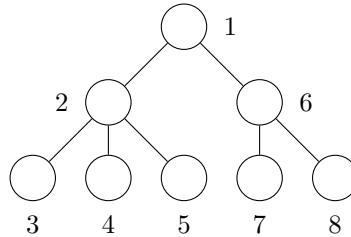
where $d(u, v)$ is the distance between u and v in T .

Example 2.14 (Diameter). The diameter of the tree below is 4, since the longest path in the tree is 4 from F/G to D .



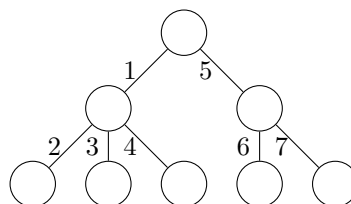
Definition 2.15 (Rooted plane tree). A *rooted plane tree* is a rooted tree in which the children of each node are ordered and unique. The children of each internal node are given by a left-to-right order, making the structure planar.

Example 2.16 (Rooted plane tree).



Remark 2.17. We use a slightly adapted definition of rooted plane trees for our domain. Instead of having ordered children for each node, we name and order the edges of the tree.

Example 2.18 (Our version of a rooted plane tree).



2.1.4 Ordered Walks

We now define *ordered walks* on rooted plane trees introduced in [KV00]. Additionally, we draw a connection to *Dyck paths* defined in [Bar16].

Definition 2.19 (Ordered walks on rooted plane trees). An *ordered walk* on a rooted plane tree (short ordered walk or just walk) can be defined as a sequence of edges (e_1, e_2, \dots, e_n) , where the start as well as the end of the walk is the root. For every $i \in \mathbb{N}$ (where $1 \leq i < n$), the edges e_i and e_{i+1} are adjacent in the tree, ensuring that the sequence of edges form a continuous path. When there is a choice where to move further, the walk chooses either one of the edges that has already been visited or the edge with the smallest value among those that have not yet been visited, more precise $e_1 = 1$ and $e_j \leq \max(e_i)_{1 \leq i < j} + 1$.

Example 2.20 (Ordered walk on a rooted plane tree). An example for a valid ordered walk on the tree of Example 1.6 could be:

$$(1, 2, 2, 3, 3, 4, 4, 3, 3, 1, 5, 6, 6, 7, 7, 5)$$

Remark 2.21. Before working with ordered walks, we first establish some clarifications.

1. Note that the same ordered walk remains identical across different trees. This means that, for example, the ordered walk

$$(1, 2, 2, 1)$$

on Example 2.16 is classified as the same as if the tree consisted only of edges 1 and 2.

2. By definition we have the property $e_j \leq \max(e_i)_{1 \leq i < j} + 1$. This means that the ordered walk

$$w = (1, 2, 2, 4, 4, 1)$$

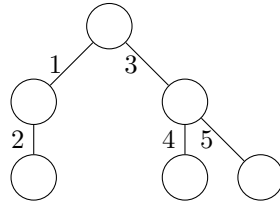
would not be a valid ordered walk since for $e_3 = 2$ the current greatest edge value is 2 such that $e_4 = 4$ violates the definition. In this case we say that w is also equal to its normalization such that

$$w = (1, 2, 2, 4, 4, 1) = (1, 2, 2, 3, 3, 1)$$

similar to the partitions in Section 3.1.

Definition 2.22. Let w be an ordered walk and t be a rooted plane tree. We say that t is the tree of w if each edge in t is visited by w . More precisely, t is the minimal tree that is traversed by w .

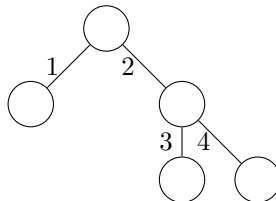
Example 2.23. In this example, we want to clarify a few misunderstandings that can occur when combining rooted plane trees of walks and the property of its normalization (see Remark 2.21). Consider the two ordered walks $w_1 = (1, 1, 2, 3, 3, 4, 4, 2)$ and $w_2 = (1, 2, 2, 1, 3, 4, 4, 5, 5, 3)$ and the rooted plane tree t of the form



Since w_1 is the normalization of the ordered walk $(1, 1, 3, 4, 4, 5, 5, 3)$, both w_1 and w_2 are walks on t . However, t is only the tree of w_2 as w_1 does not visit edge 2 of t .

Definition 2.24. Let $w = (e_1, e_2, \dots, e_n)$ be an ordered walk. Then we say $w' = (e_i, e_j, \dots, e_z)$ with $1 \leq i \leq j \leq z \leq n$ is a *sub-walk* of w , if w' itself (or at least its normalization) is an ordered walk.

Example 2.25. (a) Let $w = (1, 1, 2, 3, 3, 4, 4, 2)$ be an ordered walk on the rooted plane tree



An example of a sub-walk of w could be $w' = (3, 3, 4, 4) = (1, 1, 2, 2)$.

- (b) Let $w = (1, 2, 2, 1, 3, 3, 1, 1)$ be an ordered walk. Then the walk $w' = (1, 2, 2, 1, 1, 1)$ is a sub-walk of w , where, in contrast to w , the edge 3 is not visited.
- (c) Let again $w = (1, 2, 2, 1, 3, 3, 1, 1)$, then $w' = (1, 3, 3, 1) = (1, 2, 2, 1)$ is a sub-walk of w .
- (d) Let $w = (1, 2, 3, 3, 2, 1)$, then $w' = (1, 2, 3)$ is not a sub-walk of w since w' is not a valid ordered walk.

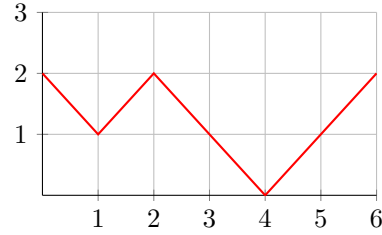
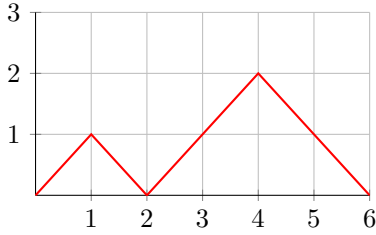
Remark 2.26. Let w be an ordered walk and let w' be a sub-walk of w . Note that, by our definition, $w \setminus w'$ (i.e. w but without w') is not necessarily an ordered walk.

Proof. Let $w = (e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8) = (1, 2, 2, 1, 1, 2, 2, 1)$ be an ordered walk, where $w' = (e_4, e_8) = (1, 1)$ is a sub-walk of w . We can see that $w \setminus w' = (1, 2, 2, 1, 2, 2)$ is not a valid ordered walk. \square

Definition 2.27. Let w be an ordered walk. We say w has a depth of k , if the minimal rooted plane tree (see Definition 2.22), which is visited by w , has a depth of k .

Definition 2.28 (Dyck path). A *Dyck path* on \mathbb{Z}^2 is a path from $(0, 0)$ to $(n, 0)$ (or $(0, a)$ to (n, a) , where a is the maximal level of the Dyck path) with the steps $(1, 1)$ and $(1, -1)$, never stepping below the line $y = 0$ (and above $y = a$).

Remark 2.29. In Definition 2.28, we allow two different representations of a Dyck path. One goes from $(0, 0)$ to $(n, 0)$ and the other from $(0, a)$ to (n, a) , where a is the maximal level of the Dyck path. Note that in principle, both Dyck paths are the same but only mirrored, as we can see in the following example.



Definition 2.30 (Simple walks on rooted plane trees). A *simple walk* on a rooted plane tree is an ordered walk where every edge is visited at most twice within the simple walk, making it a special case of ordered walks.

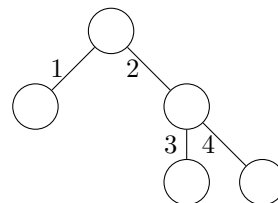
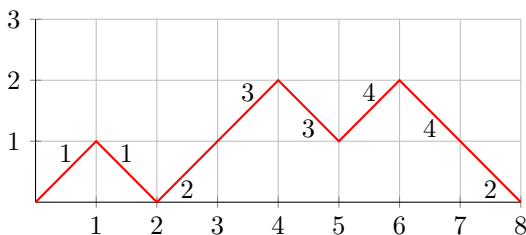
Remark 2.31. Such a Dyck path can illustrate a simple walk on a rooted plane tree. Let us specify the y -value as the *level*. The step $(1, 1)$ indicates, that we visit a new edge of the tree for the first time at *level* n , while a step $(1, -1)$ at *level* n represents returning along this edge.

Definition 2.32. The tree representation t of an ordered walk w is the rooted plane tree traversed by w , where every edge of t is visited during the walk. More precisely t is the tree of w by Definition 2.22.

Example 2.33 (Representations of a simple walk). We can see, that the simple walk

$$(1, 1, 2, 3, 3, 4, 4, 2)$$

can both be represented via a Dyck path with the rules of Remark 2.31 and a rooted plane tree with the rules of 2.32.



2.2 Mathematical Background

While first defining classical partitions of sets in Section 2.2.1, we proceed by introducing categories of those partitions. In Section 3 we go more specifically in the domain of this thesis, where we slightly redefine partitions for our purposes.

2.2.1 Partitions

TODO vielleicht nochmal ein kleiner Hintergrund und mehr einleitende Worte. In the following section, we introduce partitions on sets. As will be explained in the following section and Section 2.2.2, we can operate on them as well as classify their so called categories. All definitions proposed in this section can be found in [BCS10], [BBC07] and [Sta99]. In the scope of this thesis we will slightly modify the representation of those partitions starting at Section 3 in order to get a data structure that can be efficiently processed for our domain (see Definition 3.2).

Definition 2.34 (Partition). Let $k, l \in \mathbb{N}$. A *partition* p is given by k lower and l upper points. It partitions a set $\{1, \dots, k, k+1, \dots, k+l\}$ into disjoint subsets, called blocks. Partitions can also be visualized. Given $p \in P(k, l) \subseteq P(n)$ with $n = k + l$, we first draw a row of k upper points followed by l lower points. After numbering these points from 1 to $k + l$ (all elements from the set $\{1, \dots, k, \dots, k + l\}$) we connect them according to the blocks defined in p .

Example 2.35. Given $p \in P(4, 3)$, where $s = \{1, 2, 3, 4, 5, 6, 7\}$ is the set we partition into $p = \{\{1, 6\}, \{2, 5\}, \{3, 4, 7\}\}$. We can visualize p as follows:

$$\begin{array}{c} \{1, 6\}, \{2, 5\}, \\ \{3, 4, 7\} \end{array} \cong \begin{array}{c} \textcircled{1} \quad \textcircled{2} \quad \textcircled{3} \quad \textcircled{4} \\ \diagdown \quad \diagup \quad | \quad | \\ \textcircled{5} \quad \textcircled{6} \quad \textcircled{7} \quad \textcircled{7} \end{array} \cong \begin{array}{c} \textcircled{1} \quad \textcircled{2} \quad \textcircled{3} \quad \textcircled{4} \\ \diagdown \quad \diagup \quad | \quad | \\ \textcircled{5} \quad \textcircled{6} \quad \textcircled{7} \quad \textcircled{7} \end{array}$$

Remark 2.36. The partitions $\textcircled{1} \textcircled{2} \in P(0, 2)$ and $\textcircled{1} \in P(1, 1)$ are referred to as the *base partitions*.

Definition 2.37 (Tensor Product). Let $p \in P(k_1, l_1)$ and $q \in P(k_2, l_2)$. Then the *tensor product* can be obtained by concatenating p and q in form of $p \otimes q \in P(k_1 + k_2, l_1 + l_2)$.

Example 2.38 (Tensor Product).

$$\begin{array}{c} \textcircled{1} \quad \textcircled{2} \\ \diagdown \quad \diagup \\ \textcircled{3} \quad \textcircled{4} \end{array} \otimes \begin{array}{c} \textcircled{5} \quad \textcircled{6} \\ | \quad | \\ \textcircled{7} \quad \textcircled{8} \end{array} = \begin{array}{c} \textcircled{1} \quad \textcircled{2} \quad \textcircled{5} \quad \textcircled{6} \\ \diagdown \quad \diagup \quad | \quad | \\ \textcircled{3} \quad \textcircled{4} \quad \textcircled{7} \quad \textcircled{8} \end{array}$$

Definition 2.39 (Involution). Let $p \in P(k, l)$. Then the unary operation *involution* $p^* \in P(l, k)$ can be obtained by swapping the upper and lower points.

Example 2.40 (Involution).

$$\left(\begin{array}{c} \textcircled{1} \quad \textcircled{2} \quad \textcircled{5} \quad \textcircled{6} \\ \diagdown \quad \diagup \quad | \quad | \\ \textcircled{3} \quad \textcircled{4} \quad \textcircled{7} \quad \textcircled{8} \end{array} \right)^* = \begin{array}{c} \textcircled{3} \quad \textcircled{4} \quad \textcircled{7} \quad \textcircled{8} \\ \diagup \quad \diagdown \quad | \quad | \\ \textcircled{1} \quad \textcircled{2} \quad \textcircled{5} \quad \textcircled{6} \end{array}$$

Definition 2.41 (Composition). Let $p \in P(k_1, l_1)$ and $q \in P(k_2, l_2)$ and $k_1 = l_2$. Then the *composition* $pq \in P(k_2, l_1)$ can be obtained by connecting p and q by writing q above p and joining each lower point of q with the respective upper point of p so that we connect every point in l_2 to a point in k_1 with respect to the order. After this process, any intermediate points and loops are removed, such that only the upper points of q and the lower points of p are left.

Example 2.42 (Composition).

$$\begin{array}{c} \textcircled{1} \quad \textcircled{2} \quad \textcircled{3} \quad \textcircled{4} \\ | \quad | \quad | \quad | \\ \textcircled{5} \quad \textcircled{6} \quad \textcircled{7} \quad \textcircled{8} \end{array} \cdot \begin{array}{c} \textcircled{9} \quad \textcircled{10} \\ | \quad | \\ \textcircled{11} \quad \textcircled{12} \end{array} = \begin{array}{c} \textcircled{9} \quad \textcircled{10} \\ | \quad | \\ \textcircled{5} \quad \textcircled{6} \quad \textcircled{7} \quad \textcircled{8} \end{array} = \begin{array}{c} \textcircled{9} \quad \textcircled{10} \\ | \quad | \\ \textcircled{5} \quad \textcircled{6} \quad \textcircled{7} \quad \textcircled{8} \end{array}$$

Definition 2.43 (Rotation). Let $p \in P(k, l)$ and $k > 0$. Then the unary operator *rotation* $p^{\text{rot}} \in P(k-1, l+1)$ can be obtained by shifting the very left upper point to the very left of the lower points. Note that all points belong to the same blocks as before. The result of this operation is called *rotated version*. Analogical, we can also rotate from the lower points to the upper points and from the left side or from the right side of the partition p .

Example 2.44 (Rotation). The following figure shows an example of a *left-top rotation*:

$$\left(\begin{array}{c} \circ \quad \circ \\ | \quad | \\ \text{---} \\ | \quad | \\ \circ \quad \circ \end{array} \right)^{\text{rot}} = \begin{array}{c} \circ \\ | \\ \text{---} \\ | \quad | \\ \circ \quad \circ \end{array} \quad \circ$$

Definition 2.45. Let $p \in P(k, l)$. Then the unary operator *vertical reflection* $p^{\leftrightarrow} \in P(k, l)$ can be obtained by reversing the upper points and also reversing the lower points in p .

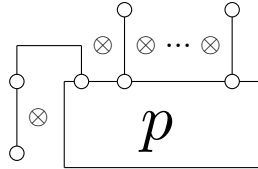
Example 2.46 (Vertical Reflection).

$$\left(\begin{array}{c} \circ \quad \circ \quad \circ \\ | \quad | \quad | \\ \text{---} \\ | \quad | \quad | \\ \circ \quad \circ \quad \circ \end{array} \right)^{\leftrightarrow} = \begin{array}{c} \circ \quad \circ \quad \circ \\ | \quad | \quad | \\ \text{---} \\ | \quad | \quad | \\ \circ \quad \circ \quad \circ \end{array}$$

Definition 2.47 (Category operation). An operation is called *category operation* if and only if it is part of the three main operations *tensor product*, *involution* and *composition*, or it can be constructed with a combination of the main operations and *base partitions* $\varnothing \in P(0, 2)$ and $\mathbb{I} \in P(1, 1)$.

Remark 2.48. Note that the rotation and the vertical reflection are category operations since they can be constructed with a combination of the three main operations tensor product, involution and composition, and the base partitions $\varnothing \in P(0, 2)$ and $\mathbb{I} \in P(1, 1)$.

Rotations as category operation. Let $p \in P(n)$ be a partition with $n \in \mathbb{N}$. Then we can produce the left-top rotation with $(\mathbb{I} \otimes p) \cdot (\varnothing \otimes \mathbb{I} \otimes \dots \otimes \mathbb{I})$, which can be visualized as follows:



As a result we can see that the rotation can be produced with the operations tensor product and composition and the base partitions $\mathbb{I} \in P(1, 1)$ and $\varnothing \in P(0, 2)$. \square

Vertical reflection as category operation. Let $p \in P(k, l)$ be a partition with $k, l \in \mathbb{N}$. Then we can produce the vertical reflection by k top-left rotations, followed by l bottom-right rotations and one involution operation.

$$p^{\leftrightarrow} \Leftrightarrow \left(p^{\text{rot}(\dots)^{\text{rot}}} \right)^*$$

\square

2.2.2 Categories of Partitions

The classification of easy quantum groups began in [BS09] and [BCS10]. As they are closely connected to partitions and their categories, this also marks the beginning of their investigation. In [Web13], Weber discovered that one quantum group was missing in [BS09] and continued the classification process. In 2016, Raum and Weber were able to present a full classification of easy quantum groups (see [SR16]). From these past studies, the concept of the category of partitions arose. In the following chapter, we define them and provide concrete examples. In Section 3, we introduce the category $\Pi_{k \in \mathbb{N}}$, which will be the main focus of this paper. Note that for the sake of this paper, we simply investigate partitions and their categories as combinatorial structures. For more details on the connections to easy quantum groups, see [BS09], the appendix, and the other papers mentioned above.

Definition 2.49 (Categories of Partitions). A *category of partitions* is defined by a subset $\mathcal{C} \subseteq P$, where $P := \bigcup_{k,l} P(k,l)$ is the set of all partitions, such that \mathcal{C} contains the base partitions $\circ \in P(1,1)$ and $\cap \in P(0,2)$ and is closed under the *category operations*. This results in the property $C(k,l) := C \cap P(k,l)$. We can also specify categories of partitions using a set S of so called *generator partitions*, where a category is closed under the set of generator partitions $\langle S \rangle$ with the base partitions.

Definition 2.50 (Non-crossing Partitions). Let p be a partition. Then we say p is a *non-crossing partition*, if for every pair of blocks (b_1, b_2) in p all upper/lower points of b_2 have exclusively smaller or exclusively higher indices than every index of the upper/lower points in b_1 .

Definition 2.51 (Balanced Partitions). Let p be a partition. Then we say p is a *balanced partition*, if every block, that is not a singleton, has the same amount of points with even and odd indices. Additionally, within a balanced partition, the amount of singletons with odd indices matches that of even indices.

Example 2.52 (Categories of Partitions). **TODO strukturierter und vielleicht mehr Bezug zum (Haupt-)Thema.** The following sets are examples of some categories of partitions, some of which we will encounter in later sections of this thesis.

Below, we can see some example categories, followed by the generator partitions which can be used to construct those.

1. The set P of all partitions $= \langle \circ, \cap \rangle$
2. The set NC of all non-crossing partitions $= \langle \cap \rangle$
3. The set P_2 of all pair partitions (every block has a size of two), $= \langle \circ \rangle$
4. The set $P^{1,2}$ of all partitions with block size one or two, $= \langle \circ, \cap \rangle$
5. The set P_{even} of all partitions with an even size, $= \langle \circ, \cap, \circ \rangle$
6. The set $P_{even}^{1,2}$ of all partitions with an even size and block size one or two, $= \langle \circ, \cap \rangle$
7. The set NC_2 of all non-crossing partitions with block size two, $= \langle \cap \rangle$ (i. e. can be produced by only the base partitions)
8. The set NC_{evenBS} of all non-crossing partitions with an even block size, $= \langle \cap \rangle$
9. The set NC_{even} of all non-crossing partitions with even size, $= \langle \cap, \circ \rangle$
10. The set B of all balanced partitions, $= \langle \cap, \circ \rangle$
11. The set B_2 of all balanced partitions of block size two, $= \langle \circ \rangle$

In Section 3 we will introduce the category $\Pi_{k \in \mathbb{N}}$. For more relevant categories of partitions, see Appendix A.

Remark 2.53. The intersection of two categories forms a new category: Let C_1 and C_2 be categories, then $C_1 \cap C_2$ is also a category.

For example,

$$NC_2 := NC \cap P_2.$$

2.2.3 Walks on Rooted Plane Trees via Moments of Adjacency Matrices

In this section, we establish the mathematical groundwork necessary to understand the main results of the paper [KV00] by Khorunzhy and Vengerovsky. We begin by briefly introducing basic concepts from discrete probability theory, then proceed to discuss moments of random variables. Building on this foundation, we define free probability spaces, and ultimately connect these concepts to the findings presented in [KV00].

We first briefly introduce the core concepts relevant to our discussion: probability spaces, random variables, and expected values.

Definition 2.54 (Discrete Probability Space). A *discrete probability space* is a pair (Ω, \mathcal{P}) , where:

- Ω is a finite or countable set called the *probability space*, representing all possible outcomes.
- $\mathcal{P} : \Omega \rightarrow [0, 1]$ is the *probability function*, satisfying $\sum_{\omega \in \Omega} \mathcal{P}(\omega) = 1$.

Example 2.55. Consider the experiment of rolling a six-sided dice once. The set of all possible outcomes is:

$$\Omega = \{1, 2, 3, 4, 5, 6\}.$$

Each outcome represents the number that appears on the upper face of the die. Assuming the die is fair, all outcomes are equally likely. We define the probability function $\mathcal{P} : \Omega \rightarrow [0, 1]$ as:

$$\mathcal{P}(i) = \frac{1}{6} \quad \text{for each } i \in \Omega.$$

This gives us the discrete probability space (Ω, \mathcal{P}) , where:

- Ω is a finite sample space with six outcomes.
- \mathcal{P} assigns a probability of $\frac{1}{6}$ to each outcome, satisfying:

$$\sum_{i=1}^6 \mathcal{P}(i) = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = 1.$$

Thus, this setup defines a valid discrete probability space in which all outcomes are equally weighted.

Definition 2.56 (Random Variable). A *random variable* is a function $X : \Omega \rightarrow \mathbb{R}$ from a sample space in Ω that assigns a real number to each outcome in Ω .

Example 2.57. Consider again the experiment of rolling a six-sided dice, but this time twice. The probability space is (Ω, \mathcal{P}) , where

$$\Omega = \{1, 2, 3, 4, 5, 6\} \times \{1, 2, 3, 4, 5, 6\} \quad \text{and}$$

$$\mathcal{P}((\omega_1, \omega_2)) = \frac{1}{36} \quad \text{for all } (\omega_1, \omega_2) \in \Omega$$

We can now define our random variable $X : \Omega \rightarrow \mathbb{R}$ as

$$X(\omega_1, \omega_2) = \omega_1 + \omega_2 \quad \text{for all } (\omega_1, \omega_2) \in \Omega$$

This random variable represents the sum of the two dice rolls. For example, $X(3, 4) = 7$. Note that while all outcomes in Ω are equally probable, the values of X are not uniformly distributed. Some sums (like 7) occur more frequently than others (like 2 or 12), as multiple outcome pairs can have the same sum.

Definition 2.58 (Expected Value). Let (Ω, \mathcal{P}) be a discrete probability space and let $X : \Omega \rightarrow \mathbb{R}$ be a random variable. The *expected value* of X , denoted by $\mathbb{E}[X]$, is defined as

$$\mathbb{E}[X] = \sum_{\omega \in \Omega} X(\omega) \cdot \mathcal{P}(\omega).$$

Definition 2.59 (Moments of a Random Variable). Let (Ω, \mathcal{P}) be a discrete probability space, $X : \Omega \rightarrow \mathbb{R}$ a random variable and $\mathbb{E}[X]$ the expected value of such a random variable. Then the *k-th moment* of X is $\mathbb{E}[X^k]$, where $k \in \mathbb{N}$.

Remark 2.60. Moments of a random variable can have multiple applications.

- $k = 1$: The first moment of a random variable is the expected value.
- $k = 2$: The second moment is associated with the *variance* of a random variable.
- $k = 3$: The third moment is associated with the *skewness* of a random variable.
- ...

Later we will see that the moments of a particular random variable in the free case is connected to our ordered walks.

Definition 2.61 (Random Graph). A *random graph* on N vertices is an undirected graph (E, V) in which the edges are drawn randomly by the following rule. Let $v_1, v_2 \in V$ and $p \in \mathbb{N}$, then $(v_1, v_2) \in E$ with a probability of $\frac{1}{p}$.

Remark 2.62. We depict random graphs (E, V) by their adjacent matrix representation $A = (f_{ij})_{i,j=1}^{|E|}$ with

$$f_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{else.} \end{cases}$$

Free probability theory is a non-commutative analogue of classical probability theory, developed to study the behavior of large random matrices and operators on Hilbert spaces.

While classical probability is concerned with commutative algebras of random variables, free probability replaces commutativity with a weaker notion called *freeness*. We need to study the implications for matrices in free probability theory used in [KV00].

In classical probability theory, random variables are functions defined on a sample space (Ω, \mathcal{P}) , and all probabilistic behavior is derived from this underlying measure space. In free probability, however, we no longer work with a sample space. Instead, we directly consider an abstract (non-commutative) algebra \mathcal{A} , as the set of all random variables, with a function calculating the respective expected values $\mathbb{E} : \mathcal{A} \rightarrow \mathbb{R}$. Thus, we substitute (Ω, \mathcal{P}) with $(\mathcal{A}, \mathbb{E})$.

Remark 2.63. All information contained in the classical probability space (Ω, \mathcal{P}) are also encoded in the pair $(\mathcal{A}, \mathbb{E})$.

Proof. Let $\omega \in \Omega$. Define a random variable $X_\omega \in \mathcal{A}$ by

$$X_\omega(\omega') = \begin{cases} 1 & \text{if } \omega' = \omega, \\ 0 & \text{otherwise.} \end{cases}$$

This function is just the indicator function of the set $\{\omega\}$.

When applying \mathbb{E} , we get

$$\mathcal{P}(\{\omega\}) = \mathbb{E}[X_\omega],$$

since the expected value is calculated by adding the values of X and multiplying them by \mathcal{P} .

Thus, using only the set of all random variables \mathcal{A} and \mathbb{E} , we can reconstruct the entire probability measure \mathcal{P} . This shows that the information in (Ω, \mathcal{P}) are also encoded in $(\mathcal{A}, \mathbb{E})$. \square

In our context, we specialize the concept of a non-commutative probability space to matrices. Instead of considering a general non-commutative algebra \mathcal{A} , we work with the algebra of $n \times n$ matrices with rational entries, denoted by $M_n(\mathbb{R})$.

Definition 2.64 (Trace of a Matrix). Let $M \in \mathbb{R}^{n \times n}$ be a square matrix. The *trace* of M , denoted by $\text{Tr}(M)$, is defined as the sum of the entries on the main diagonal of M , more precisely,

$$\text{Tr}(M) = \sum_{i=1}^n M_{ii}.$$

Example 2.65. Consider the matrix

$$M = \begin{pmatrix} 2 & 1.5 & 67 \\ -1 & 3.5 & 4 \\ 0 & 8 & -1 \end{pmatrix}.$$

Then we can calculate the trace of M as follows.

$$\text{Tr}(M) = 2 + 3.5 + -1 = 4.5.$$

Definition 2.66 (Free Probability Space of Matrices). Let $\mathcal{A} = M_n(\mathbb{R})$ be the algebra of $n \times n$ matrices over the rationals. We define the expected value $\mathbb{E} : \mathcal{A} \rightarrow \mathbb{R}$ by

$$\mathbb{E}(M) = \frac{1}{n} \text{Tr}(M),$$

where $\text{Tr}(M)$ denotes the usual trace of a matrix M . Then $(\mathcal{A}, \mathbb{E})$ forms a free probability space.

In this setting, the matrices play the role of non-commutative random variables, and the normalized trace acts as the expectation functional. This formulation is particularly suited when it comes to random matrices, such as adjacency matrices or Laplacians of large random graphs.

As a result, for our domain, we substitute the classical probability space (Ω, \mathbb{P}) with the free probability space $(\mathcal{A}, \mathbb{E})$, enabling us to apply non-commutative techniques to study the spectral properties of random graphs as explained in [KV00].

We can now derive the formula for calculating the number of ordered walks in a rooted plane tree, constructed in [KV00].

Let (A, \mathbb{E}) be a free probability space, where $A^{(N,p)} \in \mathbb{R} \times \mathbb{R}$ is our adjacent matrix of a random graph of size $N \times N$ where each edge occurs with a probability of $\frac{1}{p}$, as described above, and \mathbb{E} our expected value calculated by the trace of A . Then $\lim_{N \rightarrow \infty} M_s^{(N,p)}$ counts the number of ordered walks on a rooted plane tree, where

$$M_s^{(N,p)} = \mathbb{E}([A^{(N,p)}]^s) = \frac{1}{N} \sum_{x_i=1}^N A_{ix_1} A_{x_1 x_2} \cdots A_{x_{2k-1} i}.$$

Example 2.67. Consider the example where we want to calculate the number of ordered walks of length $k = 2$. Obviously, there is only one such a walk, namely the walk $(1, 1)$.

Let $A = (f_{ij})_{i,j=1}^n$ be our adjacent matrix of a random graph. Then we need to calculate

$$M_2^{(N,p)} = \mathbb{E}([A^{(N,p)}]^2) = \frac{1}{N} \sum_{x_i=1}^N f_{ik} f_{ki}.$$

Since the adjacent matrix is symmetric, we know that $f_{ik} = f_{ki}$. Additionally, f_{ij} is either 0 or 1 with a probability of $p = N$. As a result we get the following formula.

$$\frac{1}{N} \sum_{x_i=1}^N f_{ik} f_{ki} = \frac{1}{N} \sum_{x_i=1}^N f_{ik} = \frac{1}{N} N = 1$$

As we can see, the result is 1, which confirms our expectations from above.

In [KV00], Khorunzhy and Vengerovsky were able to find a closed formula for those walks denoted by

$$\lim_{N \rightarrow \infty} M_s^{(N,p)} = \begin{cases} m_k, & \text{if } s = 2k, \\ 0, & \text{if } s = 2k + 1. \end{cases}$$

with

$$m_k = \sum_{r=0}^k W_k(r)$$

where the numbers of $W_k(r)$ with $k \geq r \geq 0$ is given by

$$W_u(v) = \sum_{i=1}^v \sum_{j=v-i}^{u-i} \sum_{l=0}^{u-i-j} W_{u-i-j}(l) \binom{l+i-1}{i-1} \binom{v-1}{i-1} W_j(v-i).$$

Later in Section 5, we will implement and analyze the complexity of this formula, where we utilize it to compute the number of partitions in $\Pi_{k \in \mathbb{N}}$, which are connected to ordered walks (proven in Section 4). For more details of the derivation of this formula see [KV00].

3 The Category $\Pi_{k \in \mathbb{N}}$

The main theorem of [RW13] states the following.

Theorem 3.1 (Raum-Weber). *Let G be an orthogonal easy quantum group with associated category of partitions \mathcal{C} . Then \mathcal{C} is either non-hyperoctahedral, or group-theoretical and hyperoctahedral, or else $\mathcal{C} = \langle \pi_\ell \mid \ell \leq k \rangle$ for some $k \in \{1, 2, \dots, \infty\}$.*

That said, the category $\Pi_{k \in \mathbb{N}}$ is relevant in the context of easy quantum groups, making it a subject of further investigation.

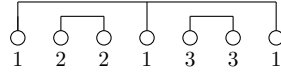
From the following chapter on we define partitions by not distinguishing between upper and lower points so that we can focus on the partition pattern, making it a special case of the classical partitions on sets defined in Section 2.2.1. Furthermore in this chapter we reformulate and use definitions from [RW13]. Especially from Section 4 and 5.

3.1 Definitions

Definition 3.2. Let $n \in \mathbb{N}$. A *partition* p is a tuple $(p_1, \dots, p_n) \in \mathbb{N}^n$, where n is the size of p , such that $p_1 = 1$ and $p_j \leq \max(p_i)_{1 \leq i < j} + 1$. It is similar to the partition in Definition 2.34, where we simply not distinguish between upper and lower points.

Remark 3.3. Also the modified partitions can be visualized by graphs, where each point a_i is a node which is connected to all nodes a_j if $a_i = a_j$.

Example 3.4. The partition $p = (1, 2, 2, 1, 3, 3, 1)$ can be visualized by the graph



Remark 3.5. Similar to ordered walks, we also normalize partitions to ensure they satisfy Definition 3.2. For example, the partition

$$p = (1, 2, 2, 4, 4, 1) = (1, 2, 2, 3, 3, 1)$$

is adjusted accordingly.

Remark 3.6. Recall that any ordered walk w can be represented by a rooted plane tree, where we visit every edge (see Definition 2.32). Furthermore, w can also be depicted by a partition p where the sequence of edges (e_1, e_2, \dots, e_n) of the walk are exactly the points (p_1, p_2, \dots, p_n) of p . Note that at this point there is no guarantee that the points (p_1, p_2, \dots, p_n) form a valid walk sequence according to Definition 2.19, meaning that this has to be proven first.

Definition 3.7 (Open pairs). Let p be a partition (often but not necessarily) in NC_2 of the form (p_1, p_2, \dots, p_n) consisting of $m \in \mathbb{N}$ different blocks b_1, \dots, b_m . For two blocks b_i and b_j with $i < j \leq m$, we define the set of *open pairs* between b_i and b_j as the set of all blocks b_k such that:

- b_k has a point $p_{l_1} \in b_k$ that occurs before any point $p_r \in b_i$ ($l_1 < r$) or after any point $p_r \in b_j$ ($r < l_1$), and
- b_k also has a point p_{l_2} that occurs between any two points of $p_{r_1} \in b_i$ and $p_{r_2} \in b_j$, i.e. $r_1 < l_2 < r_2$.

Example 3.8. Consider the partition $p = (1, 2, 3, 3, 2, 4, 5, 5, 4, 1, 6, 6) \in NC_2$. The blocks in this partition are:

$$b_1 = \{p_1, p_{10}\}, \quad b_2 = \{p_2, p_5\}, \quad b_3 = \{p_3, p_4\}, \quad b_4 = \{p_6, p_9\}, \quad b_5 = \{p_7, p_8\}, \quad b_5 = \{p_{11}, p_{12}\}.$$

We are interested in the open pairs between b_3 and b_5 .

According to the definition, the open pairs between b_3 and b_5 are those blocks b_k that:

- have a point occurring before any point of b_3 or after any point of b_5 , and
- also have a point occurring between the points of b_3 and b_5 in the sequence.

In this case, the only blocks satisfying these conditions are $b_2 = \{p_2, p_5\}$ and $b_4 = \{p_6, p_9\}$, because:

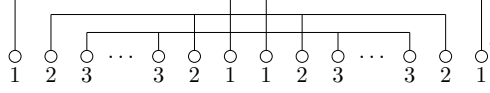
- The point $p_2 = 2$ occurs before any point of b_3 .
- The point $p_9 = 4$ occurs after any point of b_5 .
- The points $p_5 = 2$ and $p_6 = 4$ are between all points of b_3 and b_5 .

Thus, the set of open pairs between b_3 and b_5 is $\{b_2, b_4\}$.

Definition 3.9. The partition $\pi_{k \in \mathbb{N}}$ is given by k four blocks on $4k$ points of the form

$$\pi_k = (p_1, \dots, p_k, p_k, \dots, p_1, p_1, \dots, p_k, p_k, \dots, p_1).$$

More precisely, for $k \geq 3$:



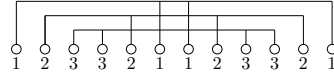
Definition 3.10. Let $p \in \Pi_k = \langle \pi_k, k \in \mathbb{N} \rangle$ and $q \in NC_2$, then p can be constructed from q by merging blocks of q according to the following rules:

1. We may only merge blocks which are on the same level of the Dyck path of q .
2. If we connect two blocks b_1 and b_2 in q , we must also connect all open pairs between b_1 and b_2 .

From now on, when writing Π_k or $\Pi_{k \in \mathbb{N}}$, we refer to the category $\bigcup_{k \in \mathbb{N}} \Pi_k(k, 0)$.

Example 3.11. In this example, we go deeper into the rules of Definition 3.10, where we show an example for both the partition layer as well as the rooted plane tree layer of a partition which will be further discussed in Section 4.2.

- (a) The partition $\pi_3 \in \Pi_k$ of the form

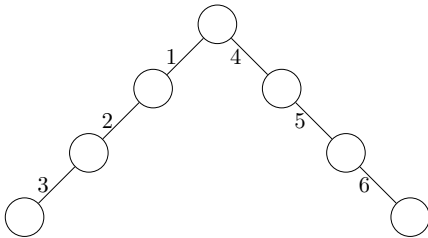


can be retrieved by modifying $p \in NC_2$ of the form



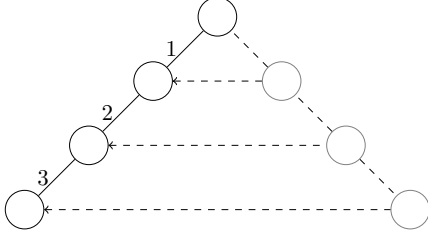
through the merging of blocks 3 and 6. According to the second rule of Definition 3.10, we also have to connect the blocks 1, 4 and 3, 5. Note that every block that we merged was on the same level of the Dyck path of p , thus satisfying the first rule of Definition 3.10.

- (b) Since the rules of Definition 3.10 are crucial for understanding the proof presented in Section 4.2, the following example has the purpose of further visualizing them to gain a deeper understanding. We illustrate and visualize the example of (a) using the rooted plane tree structure. The partition p from (a) (where we can interpret p as an ordered walk) can be represented by the following rooted plane tree:



As we can see, the level of the Dyck path corresponds to the depth of the edges in the tree representation (later shown for the general case in Proposition 4.6). The first rule ensures that only edges with the same depth are merged. According to Definition 2.15 and Remark 2.17, each edge value must be unique. Therefore, simply overwriting edge value 6 with value 3 in p would violate the definition of a rooted plane tree.

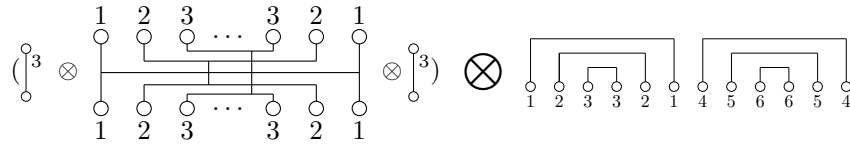
In order to get a valid rooted plane tree after the merging process, we need to apply the second rule. From Example 3.11, we already know that this involves also merging the open pairs, i.e. edge 5, 2 and edge 1, 4.



The resulting rooted plane tree looks like a folded up version of p , where the path from the root to including edge 6 is merged to the path from root to including edge 3. The derived partition π_3 represents the walk (1, 2, 3, 3, 2, 1, 1, 2, 3, 3, 2, 1) on the folded up rooted plane tree.

In general, merging two different edges involves folding the path from the root to these edges. This creates a walk that is no longer simple, as the edges on the resulting path are traversed multiple times. Specifically, the number of traversals for these paths equals the sum of the traversals from the two original paths. Understanding this intuition will be crucial to understand the proof of Theorem 4.14.

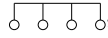
The intuition behind the rules of Definition 3.10 lies in the different partition operations, where we can simulate the rules by specific composition operations with partitions in $\Pi_{k \in \mathbb{N}}$. In Example 3.11 (a), we basically obtain π_3 by the following operation.



3.2 Relevant Properties

In this section, we go deeper into the properties of the category $\Pi_{k \in \mathbb{N}}$, especially those that are relevant for Section 4. First, we want to understand the origin of the category, namely its generator partitions (see Definition definition 3.10).

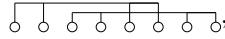
The partition π_1 coincides with



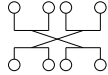
the rotated version of



while π_2 coincides with



the rotated version of



By understanding π_k , we are able to gain more insights about the category.

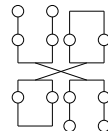
Proposition 3.12. *Let Π_m be generated by $\langle \pi_m \rangle$, then $\pi_l \in \Pi_m$ where $l \leq m$. As a result, we have $\Pi_m = \langle \pi_l, l \leq m \rangle$.*

Proof. From Definition definition 3.9, we know that $\pi_{k \in \mathbb{N}}$ is given by k four blocks on $4k$ points of the form $\pi_k = (p_1, p_2, \dots, p_{k-1}, p_k, p_k, p_{k-1}, \dots, p_2, p_1, p_2, \dots, p_{k-1}, p_k, p_k, p_{k-1}, \dots, p_2, p_1)$. Let $k > 2$, then we can reduce π_k to π_{k-1} by applying category operations with the base partitions \sqcap and \sqcup . We simply compose π_k with the partition

$$(a_1, a_2, \dots, a_{k-1}, a_k, a_k, a_{k-1}, \dots, a_2, a_1, a_1, a_2, \dots, a_{k-1}, a_k, a_k, a_{k-1}, \dots, a_2, a_1; \leftarrow \text{upper points}$$

$$a_1, a_2, \dots, a_{k-1}, a_{k-1}, \dots, a_2, a_1, a_1, a_2, \dots, a_{k-1}, a_{k-1}, \dots, a_2, a_1) \leftarrow \text{lower points}$$

where we basically remove the points of the block a_k . For the case that $k = 1$ we perform the following operations:



The result is the partition π_1 . □

The consequence of Proposition 3.12 is the following series of categories:

$$\langle \pi_1 \rangle \subseteq \langle \pi_2 \rangle \subseteq \langle \pi_l, l = 3 \rangle \subseteq \langle \pi_l, l = 4 \rangle \subseteq \dots \subseteq \Pi_{k \in \mathbb{N}}$$

Definition 3.13. A category \mathcal{C} is *finitely generated* if the set of generators consists of finitely many partitions. A category \mathcal{C} is *singly generated* if $\mathcal{C} = \langle p \rangle$ for some partition p .

Proposition 3.14. $\Pi_{k \in \mathbb{N}}$ is not finitely generated and $\langle \pi_l, l \leq k \rangle \neq \Pi_{k \in \mathbb{N}}$ for $k < \infty$.

Proof. Proposition 5.6. and Section 6 of [RW13]. \square

In preparation for the proof of Theorem 4.14, we again investigate the rules of Definition 3.10 further in Section 4.2.

4 The connection between Ordered Walks and the Category

$\Pi_{k \in \mathbb{N}}$

The main purpose of this section is to prove the equality of ordered walks on rooted plane trees and the category $\Pi_{k \in \mathbb{N}}$. First, we prove the weakened statement that the set of partitions in NC_2 is equal to the set of all simple walks. Building on this foundation, we proceed to prove the main statement. Recall that (unless explicitly stated) we do not distinguish upper and lower points in this thesis, i.e. we use Definition 3.2 in this and primarily also in the upcoming sections. Consequently, by writing NC_2 we refer to those partitions in NC_2 that only have upper points ($NC_2(n \in \mathbb{N}, 0)$).

4.1 Simple Walks, Dyck Paths and the Category NC_2

Theorem 4.1. Let W_s be the set of all simple walks, then $NC_2 = W_s$.

Proof. **TODO ein bisschen mehr Struktur reinbringen.** We need to proof the following two properties:

1. $W_s \subseteq NC_2$
2. $NC_2 \subseteq W_s$

We start by proving 1.

For this proof we will proceed inductively, where n is the length of a simple walk w_s as well as the size of the corresponding partition p .

Base case $n = 0$: We get an empty walk $w_s = ()$ consisting of an empty sequence of edges. The corresponding partition $p = ()$ is the empty partition, which obviously is in NC_2 .

As an induction hypothesis we say that for any simple walk $w_s = (e_1, \dots, e_n)$ of arbitrary even length $n \geq 0$ there is a corresponding partition $p = (e_1, \dots, e_n) \in NC_2$ of size n .

Let w_s be a simple walk of even length n of the form (e_1, e_2, \dots, e_n) , where e_i is the i -th edge of w_s . Let $p = (e_1, e_2, \dots, e_n)$ be the partition $p \in NC_2$ represented by w_s (induction hypothesis).

For the induction step, we show that for any simple walk w'_s of length $n + 2$, constructed from w_s by visiting a new edge, we have a corresponding partition in NC_2 . More precisely, we need to construct w'_s from w_s by performing partition operations. The difference between w'_s and w_s is that w'_s traverses one additional edge compared to w_s . Consequently, we need to demonstrate that a new edge can be introduced at any position within the simple walk through partition operations. These operations are a specific number of rotations and a tensor product.

Let $j \in \mathbb{N}$, where $j \leq n$, be the index of the last edge leading to the node where a new edge and node are to be inserted in w_s . This can be achieved by rotating p exactly $n - j$ times, such that the new edge and node are added to the desired location:

$$\begin{aligned} (e_1, \dots, e_j, \leftarrow \text{upper points} \\ e_n, \dots, e_{j+1}) \leftarrow \text{lower points} \end{aligned}$$

Let this new rotated partition be p' . The only thing left to do, before rotating p' back so that it only has upper or lower points again, is the following tensor product with the base partition $\sqcap \in NC_2$:

$$p' \otimes \sqcap$$

The resulting partition is our w'_s , where we successfully inserted a new edge that is traversed in w'_s at an arbitrary position using partition operations. Since we have proven that it is possible to insert a new edge in any arbitrary position in w_s , we have shown that for any simple walk, there is a partition that can be constructed using the operations above. Furthermore, because rotation and tensor product are category operations, the resulting partition is in NC_2 .

The proof for 2. follows similarly by induction. In the induction step, for constructing (in this case) the partition w'_s of length $n+2$ from w_s of length n , we simply visit a new edge at the corresponding position in the simple walk w_s , where the walk w'_s clearly remains a simple walk. \square

Theorem 4.2. *Let W_s be the set of all simple walks, then W_s is equal to the set of all Dyck paths.*

Proof. Dyck paths represent partitions in NC_2 (see [RW13]) combined with Theorem 4.1. \square

Remark 4.3. Recall Definition 2.32 along with its example, where we defined rooted plane trees for simple walks. Since we have proven that simple walks correspond to partitions in NC_2 , it follows that each partition in NC_2 also has a rooted plane tree representation, namely the tree of its associated simple walk.

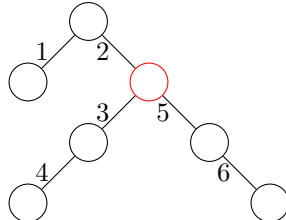
4.2 Ordered Walks and the Category $\Pi_{k \in \mathbb{N}}$

In order to draw a connection between the category $\Pi_{k \in \mathbb{N}}$ and ordered walks, we first need to prove multiple properties about the rules of Definition 3.10. With a combination of Proposition 4.12 and Proposition 4.13, we can finally prove the connection in Theorem 4.14, saying that the set of all ordered walks on rooted plane trees is equal to the set of partitions in the category Π_k . Furthermore, we make use of Remark 4.3, which enables us to operate on the rooted plane tree layer of partitions in NC_2 .

Proposition 4.4. *In a rooted plane tree, representing a partition in NC_2 and thus also a simple walk (according to Theorem 4.1), the open pairs between b_1 and b_2 are precisely those edges that lie on the path from b_1 and b_2 to their lowest common ancestor.*

Proof. Let p be a partition in NC_2 (and a simple walk), and let b_i and b_j represent any two blocks in p , where $i < j$. The open pairs between b_i and b_j are defined as those blocks that contain exactly one point value between b_i and b_j . This property follows from Rule 2 of Definition 3.10, which states that open pairs must satisfy the two conditions that they must include point values that lie between b_i and b_j , and they must exclude point values that lie outside this range. Since p is in NC_2 , all blocks in p must be of size two. Consequently, for open pairs, there is precisely one point value between b_i and b_j . In the rooted plane tree of p , the open pairs are precisely those that lie on the edges of the path from the edge b_i and b_j to their lowest common ancestor. These edges correspond to the blocks that lie between b_i and b_j and were visited exactly once (in the simple walk layer of p) in both positions, since they were traversed to reach b_i and b_j , but were not traversed back. \square

Example 4.5. Let $p = (1, 1, 2, 3, 4, 4, 3, 5, 6, 6, 5, 2)$ with the following rooted plane tree representation t :



Consequently, the open pairs between 4 and 6 are exactly 3 and 5, which are also those edges that lie on the the path to the lowest common ancestor (marked red in the tree t).

Proposition 4.6. *Let p be a partition in NC_2 . In the Dyck path representation of p , the level corresponds to the depth of the rooted plane tree that represents the simple walk of p .*

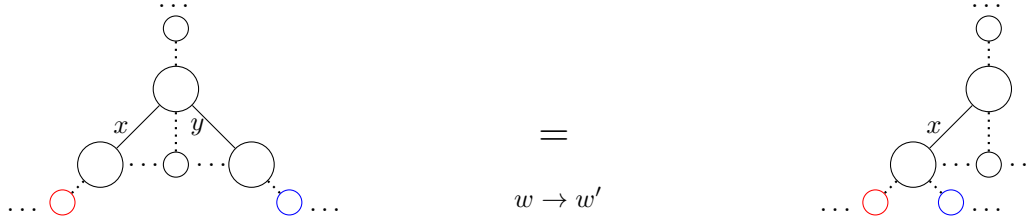
Proof. As stated in Remark 2.31, the step (1, 1) in a Dyck path corresponds to visiting a new edge in the rooted plane tree, which increases both the level in the Dyck path and the depth in the tree. Thus, every upward step in the Dyck path reflects an increase in the depth of the tree. \square

Proposition 4.7. *Let $w_1 = (e_1, \dots, e_n)$ and $w_2 = (v_1, \dots, v_m)$ of size $n, m \in \mathbb{N}$ be two independent ordered walks on distinct rooted plane trees. Then the concatenation $w_1 w_2$ also forms an ordered walk.*

Proof. Since w_1 and w_2 operate on distinct rooted plane trees, we concatenate them by consistently assigning w_2 edge values starting from $\max_i e_i$ such that the result w'_2 is still equal to w_2 (see Remark 2.21) and the edge values in w_1 and w_2 are distinct. The resulting concatenation $w_1 w'_2$ has the form $(e_1, \dots, e_n, v'_1, \dots, v'_m)$, which obviously forms a valid ordered walk. \square

Lemma 4.8. *Let $w = (e_1, \dots, e_n)$ be an ordered walk, then merging two arbitrary valid blocks in the partition $p = w$ according to the rules in Definition 3.10 results again in an ordered walk w' .*

Proof. To show that the result w' still represents an ordered walk, we have to take a look at the rooted plane tree layer of $w (= p)$ and $w' (= p')$. Let us be at an arbitrary valid position in the rooted plane tree of p where we apply the rules of Definition 3.10. Let $w = (e_1, \dots, x, \dots, y, \dots, e_n)$ and let a_i and b_i be valid sub-walks and $x = (x_1, a_1, x_2, a_2, \dots, x_m)$ and $y = (y_1, b_1, y_2, b_2, \dots, y_m, b_m)$ with $m \in \mathbb{N}$ be the paths with the blocks as edge values we want to merge. Note that we are allowed to merge them along x_i to y_i , since they have the same parent and length, regarding the x and y path we want to merge. So for each x_i there is a y_i with the same depth and thus with the same Dyck level. The figure below shows the relation of the (distinct) block sequences (i.e. paths in the tree layer) x and y before and after merging them along the x'_i s and the y'_i s.



Let the red node illustrate the sub tree of the node lead by x and similar the blue ($= b_m$) the sub tree of the node lead by y . After the merging process, we see that y , including the remaining sub tree, simply folded up to x . Consequently we get a partition $p' = w' = (e_1, \dots, x, \dots, x', \dots, e_n) \in \Pi_k$, where the blocks from the x'_i s increased their sizes by 2. After the merging process, x remains identical and y changes to $x' = (x_1, b_1, x_2, b_2, \dots, x_m, b_m)$. From Definition 2.19 we see that w' must fulfill the following properties to be a valid ordered walk:

1. w' has to end and start at the root.

This requirement still holds since x' forms an ordered walk. Therefore, x' starts and ends at the lowest common ancestor of x_m and y_m in w (see Proposition 4.4). As the sub-walk from the root to this lowest common ancestor and the rest of w' remains unchanged, it follows that w' also starts and ends at the root.

2. The edges e_i and e_{i+1} in w' must be adjacent to the tree of w' .

Since every edge value in w' remains identical to w except for the sub-walk x' , we focus on the edges connecting to x' . Both x and x' start with x_1 and end with x_m (b_m is distinct to the rest of w' , i.e. does not need to be taken into account by Proposition 4.7) and because x is correctly connected to the remaining walk, we also know that x' is correctly connected. The edge values within x' are also consistent, because x' itself is a valid ordered walk.

3. When choosing the next step, the w' has to take either a previously visited edge or the unvisited edge with the smallest value.

Again we need to investigate x' in the context of the remaining part of w' . Since the edges x_i for $1 \leq i \leq m$ are already visited, they do not violate this property. As for the remaining edge values in b_i , which are distinct by definition, we can again use Proposition 4.7.

As a result, w is still an ordered walk after the merge process, which resulted in w' , with the difference that the path along the x'_i s is visited twice more. \square

Definition 4.9. Let w be a ordered walk. We define $|w| \in \mathbb{N}$ as the number of breaks required to transform w into a simple walk i.e.

$$|w| = |(a_1, \dots, a_n)| = \frac{n}{2} - \max_i a_i$$

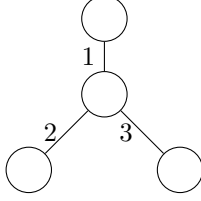
Consequently, the following property applies

$$|w| = 0 \Leftrightarrow w \text{ is simple}$$

Example 4.10. Let w be of the form

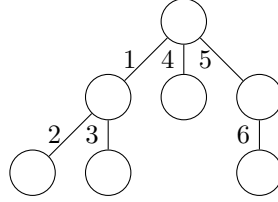
$$(1, 2, 2, 3, 3, 1, 1, 1, 1, 3, 3, 1)$$

where the following tree is traversed



Since the edges 1 and 3 are visited more than twice, we know that the walk w is not a simple walk. To transform w into a simple walk w_s , we need to break the blocks 1 and 3 until every block has size two.

This yields the walk $w_s = (1, 2, 2, 3, 3, 1, 4, 4, 5, 6, 6, 5)$, with the corresponding rooted plane tree



We see that by breaking the blocks, edge 1 is duplicated as 4, and the path 1 – 3 is duplicated as 5 – 6. From the example we get that $|w| = 3$ and $|w_s| = 0$.

Lemma 4.11. Let w be a walk. Breaking a block b_w in w , where b_w represents the value of an edge being the deepest possible edge visited more than twice in the rooted plane tree representation of w , produces a valid walk w' .

Proof. Let w be a walk, and let b_w represent the deepest edge that is visited more than twice. To ensure that breaking b_w into b_{w1} and b_{w2} does not form a cycle in the rooted plane tree layer of w , we recall that all edges below b_w are visited exactly twice. This guarantees that none of these edges can be visited by both b_{w1} and b_{w2} . Consequently, breaking b_w cannot result in the formation of a cycle and thus w' represents an ordered walk on a valid rooted plane tree. \square

Proposition 4.12. Let W be the set of all ordered walks, then $W \subseteq \Pi_{k \in \mathbb{N}}$.

Proof. Let w be a ordered walk of the form (e_1, e_2, \dots, e_m) , where $m \in \mathbb{N}$ is the length of w .

Base case: $n = |w| = 0$:

By definition, w is a simple walk. From Theorem 4.1 we know, that $(e_1, e_2, \dots, e_m) \in NC_2$ and $NC_2 \subset \Pi_k$.

Induction hypothesis: We assume that $w \in \Pi_k$, if $|w| = n$

Induction step: Let $0 < |w| = n + 1$. In order to show $w \in \Pi_k$, we apply Lemma 4.11 and break the block b with greatest depth, having the property $|b| > 2$ into b_1 of size $|b| - 2$ and b_2 of size 2. The resulting walk $w' = (e'_1, \dots, e'_m)$ has the property $|w'| = n$. Using our induction hypothesis we can deduce that $(e'_1, \dots, e'_m) \in \Pi_k$. By applying the first rule of Definition 3.10, we can derive w , where we undo the break from above. Since the rules in Definition 3.10 define Π_k , we know that the resulting points from w are $w = (e_1, e_2, \dots, e_m) \in \Pi_k$. \square

Proposition 4.13. Let W be the set of all ordered walks, then $\Pi_{k \in \mathbb{N}} \subseteq W$.

Proof. Let $p \in \Pi_k$ of the form (p_1, p_2, \dots, p_m) , where $m \in \mathbb{N}$ is the size of p . Furthermore, the function $op : \Pi_k \rightarrow \mathbb{N}$ returns the number of pairs merged according to the rules of Definition 3.10.

Base case: $n = op(p) = 0$:

By Definition, $p \in NC_2$ and consequently according to Theorem 4.1, we know, that $p = (p_1, p_2, \dots, p_n)$ is a simple walk.

Induction hypothesis: We assume that p is an ordered walk, if $op(p) = n$

Induction step: Let first be $op(p) = n$. We now apply rule 1 of Definition 3.10 at arbitrary valid positions, resulting in the merging of up to $m \in \mathbb{N}$ blocks. The value of m depends on the number of open pairs between the chosen valid positions. Specifically, if there are $m - 1$ open pairs, requiring the merging of m blocks, then by selecting different valid positions along the path from the original positions to their lowest common ancestor (see Proposition 4.4), we can achieve any number of merges between 1 and m . From Lemma 4.8 we can deduce that the result p' is still an ordered walk and $op(p') = n + z$ for $0 < z \leq m$. \square

Theorem 4.14. *Let W be the set of all ordered walks, then $\Pi_{k \in \mathbb{N}} = W$.*

Proof. From $W \subseteq \Pi_k$ (Proposition 4.12) and $\Pi_k \subseteq W$ (Proposition 4.13) follows that $\Pi_k = W$. \square

4.3 The *Holy Quaternity* - A fourth Definition of $\Pi_{k \in \mathbb{N}}$ arose

In this section, we examine and revisit the different representations of elements in $\Pi_{k \in \mathbb{N}}$. Furthermore, we introduce and reformulate an alternative definition discovered by Alexander Mang in [Man22], which we independently rediscovered in a different context through our findings in Section 4.2.

The category $\Pi_{k \in \mathbb{N}}$ can be described in several equivalent ways, each offering a different perspective on its structure:

- (a) by its generator set (Definition 3.9 and Definition 3.10)
- (b) by constructing $\Pi_{k \in \mathbb{N}}$ from non-crossing pair partitions (Definition 3.10)
- (c) by ordered walks on rooted plane trees (Theorem 4.14)
- (d) by non-interference ([Man22])/even-nested-ness (in this section)

While $\Pi_{k \in \mathbb{N}}$ can be defined by its generator set $\langle \pi_k, k \in \mathbb{N} \rangle$, Raum and Weber introduced a new perspective by Definition 3.10, which states that we can build any element in $\Pi_{k \in \mathbb{N}}$ by merging blocks of a non-crossing pair partition in a way that respects both the Dyck level structure and the open pair constraints (see Section 3). We were able to prove the connection between ordered walks (recall Theorem 4.14). With this knowledge, we introduced the following different view of $\Pi_{k \in \mathbb{N}}$.

Definition 4.15 (Even-Nested). A partition $p = (p_1, \dots, p_n)$ is called *even-nested* if, for each block, every point value between its leftmost occurrence p_i and rightmost occurrence p_j appears an even number of times.

We can show the relation between this even-nested definition to $\Pi_{k \in \mathbb{N}}$ via ordered walks.

Proposition 4.16. *$p \in \Pi_{k \in \mathbb{N}}$ if and only if p is even-nested.*

Proof. First, recall from Section 4.2 that p is also an ordered walk.

\Rightarrow : By definition, an ordered walk on a rooted plane tree must start and end at the root. Consequently, each edge in the tree is traversed an even number of times. Since every ordered walk maintains this property, it follows that for any sub-walk traversing a sub-tree within p , each edge is also visited an even number of times, ensuring that every traversal eventually returns to its starting point. This guarantees that, within any block of the corresponding partition, all points (edges in a walk) between its leftmost point p_i and rightmost point p_j appear an even number of times. Thus, by Definition 4.15, every ordered walk satisfies the even-nested property.

\Rightarrow : We proceed by induction on the size of p .

Base case: For $|p| = 0$, the partition is empty and trivially satisfies the definition of an ordered walk.

Inductive Hypothesis (I.H.): Assume that for all partitions of size n , if p is even-nested, then it corresponds to an ordered walk.

Inductive Step (I.S.): Consider a partition p of size n . Since p is even-nested, every block maintains the property that the number of points between its leftmost and rightmost occurrence is even, ensuring that any new edge added to the structure does not violate this evenness condition.

Now, inserting a new pair of points into p preserves this property while also ensuring that the walk remains connected. The newly added points must correspond either to a value already present in the walk or introduce a new pair of values.

If the new points correspond to an existing value, the newly added edges traverse the corresponding edge two additional times. Otherwise, if a new pair of values is introduced, the new points must not be placed in a crossing manner with respect to any other block (by definition). Consequently, the new edge is simply appended at some position in the tree, ensuring that it is visited once in each direction.

Thus, by the inductive hypothesis, the structure up to size n was already an ordered walk, and adding a new valid pair while preserving even-nesting ensures that the resulting structure for $n+2$ is also an ordered walk.

By induction, it follows that every even-nested partition corresponds to an ordered walk and therefore also to a partition in $p \in \Pi_{k \in \mathbb{N}}$. \square

However, this finding turned out to be already formulated by Mang in [Man22], where he defined $\Pi_{k \in \mathbb{N}}$ using the notion of non-interference.

Definition 4.17 (Non-Interference). Let p be a partition and $A, B \in p$. We say that A and B are *non-interferent* iff every interval formed by every pair of points in A does include an even number of points in B . More precisely,

$$\forall (\alpha, \alpha') \in A^{\times 2} : \quad |[\alpha, \alpha'] \cap B| \equiv_2 0.$$

With the definition of non-interference, Mang defined $\Pi_{k \in \mathbb{N}}$ as follows.

$$\Pi_{k \in \mathbb{N}} := \{p \mid \forall A \in p : |A| \equiv_2 0 \quad \wedge \quad \forall (A, B) \in p^{\times 2} : A \neq B \Rightarrow A, B \text{ non-interferent}\}$$

Proposition 4.18. *A partition p is even-nested if and only if every block in p has even size, and any two distinct blocks in p are non-interferent.*

Proof. We prove both directions of the equivalence.

\Rightarrow : We start by assuming that p is even-nested. Then by Definition 4.15, for each block A in p , every point value that occurs between the leftmost and rightmost positions of points in A appears an even number of times.

- Since for the endpoints of each block, all point values in between have to occur an even number of times (also the points from the block itself), we can conclude that every block has to have an even number of points.
- Now let A and B be two distinct blocks in p . To show that they are non-interferent, take any pair of points $(\alpha, \alpha') \in A^{\times 2}$. Consider the interval $[\alpha, \alpha']$. Since p is even-nested, every block label that appears between two elements of A , including those from block B , must appear an even number of times, since the points from A itself occur an even number of times. Hence, $|[\alpha, \alpha'] \cap B| \equiv_2 0$, and so A and B are non-interferent.

\Leftarrow : Now assume that every block in p has an even size, and every pair of distinct blocks is non-interferent. Let A be a block in p , and let p_i and p_j be the leftmost and rightmost occurrences of A .

We must show that every value of points that appears in the interval $[i, j]$ appears an even number of times. By the definition of non-interference, every point value of a block except A has to appear an even number of times in any interval constructed of points in A . As a result, this also applies to $[i, j]$. Furthermore, since every block in p has an even size, we know that points belonging to A also have to occur an even number of times in $[i, j]$. Therefore, every point value between p_i and p_j occurs an even number of times. Thus, p is even-nested by Definition 4.15. \square

Definition 4.20 (W-depth). Let p be a partition. Then the w -depth of p , denoted by $w\text{-depth}(p)$, is the largest integer k such that a rotated version of p contains a W -structure of depth k . More precisely, p contains a W -structure of depth k if there exist k distinct elements a_1, \dots, a_k such that p can be written in the form:

$$p = Y_1 \underbrace{a_1 X_1^\alpha a_2 \dots a_k}_{S^\alpha} X_k^\alpha \underbrace{a_k X_k^\beta a_{k-1} \dots a_1}_{S^\beta} Y_2 \underbrace{a_1 X_1^\gamma a_2 \dots a_k}_{S^\gamma} X_k^\gamma \underbrace{a_k X_k^\delta a_{k-1} \dots a_1}_{S^\delta} Y_3$$

where each letter a_i appears an odd number of times in the respective segments $S^\alpha, S^\beta, S^\gamma$, and S^δ formed by these subwords. The number $w\text{-depth}(p)$ thus measures the maximal depth of such a W -structure present in p .

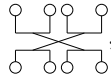
Example 4.21. Since the definition of $w\text{-depth}(p)$ is admittedly confusing, we will discuss some examples and edge cases.

- (a) π_k has a w -depth of k . This results from the definition of π_k

$$\pi_k = (p_1, \dots, p_k, p_k, \dots, p_1, p_1, \dots, p_k, p_k, \dots, p_1)$$

which basically coincides with the definition of $w\text{-depth}(p)$, where $a_i = p_i$ and the remaining segments Y_i, X_i, \dots are empty. In fact, the definition for w -depth is designed in such a way that a partition p with $w\text{-depth}(p) = k$ is like π_k with some additional optional sub-partitions (see Remark 4.8.(f) in [RW13]).

- (b) The partition $p = (1, 2, 2, 1, 3, 3, 1, 2, 2, 1, 3, 3)$ has a w -depth of 3, although it only contains a W -structure of depth 2, namely $a_1 = 1, a_2 = 2$. But since the rotated version $(1, 2, 3, 3, 2, 1, 1, 2, 3, 3, 2, 1)(= \pi_3)$ has the maximal W -structure of all rotated version of p with 3, we say $w\text{-depth}(p) = 3$.
- (c) Recall that by definition each letter a_i has to appear an odd number of times in the respective segments. To get an idea why this rule is important we will look at the following example. Let p be of the form $p = (1^2, 2^2, 3^3, 3^3, 2^2, 1^1, 1^1, 2^2, 3^3, 3^3, 2^2, 1^1)$. More precisely p basically looks similar to π_3 with the difference that we duplicated each point. Intuitively, this partition should have a w -depth of 3 since π_k has a w -depth of 3. Nevertheless, it actually has a w -depth of 2 since it basically is a modified version of π_2 . Recall that by rotating π_2 , we get

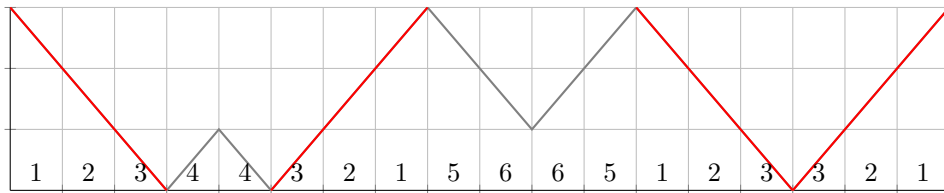


which can be used to swap arbitrary neighboring pair points of a partition by applying composition. So to retrieve p , we can simply start with $(1^2, 1^2, 1^2, 1^2, 2^2, 2^2, 2^2, 2^2, 3^2, 3^2, 3^2, 3^2) \in NC_2$ and swap with the rotated version of π_2 until we get p .

- (d) The term w -depth is named after the Dyck path representation of a partition. Consider the partition

$$p = (1, 2, 3, 4, 4, 3, 2, 1, 5, 6, 6, 5, 1, 2, 3, 3, 2, 1).$$

The Dyck path of p is



We can see that $w\text{-depth}(p) = 3$ since we have the the points 1, 2, 3 in a repeating pattern like in the definition for w -depth. Looking at the Dyck path we can see that the point 1, 2, 3 are drawing a big W (marked in red).

Definition 4.22 (Crossings in Ordered Walks). Let w be an ordered walk represented as

$$w = \dots w_a \dots w_b \dots w_a \dots w_b \dots$$

where w_a and w_b are non-empty sub-walks of w . We say that w_a and w_b *cross each other* in w if they appear interleaved within the sequence as shown above.

Furthermore, if the partition $p = w$ has a W-structure of depth $k > 1$, then w contains a crossing, where the sub-walk associated with the W-structure intersects itself within the walk.

Example 4.23. (a) The walk $w = (1, 2, 2, 1, 3, 3, 4, 4, 1, 2, 2, 1, 4, 4)$ has a crossing between the sub-walks $w_a = (1, 2, 2, 1)$ and $w_b = (4, 4)(= (1, 1))$

(b) The walk $w = (1, 2, 3, 3, 2, 1, 4, 4, 1, 2, 3, 3, 2, 1)$ has a crossing where the sub-walk $w_a = (1, 2, 3, 3, 2, 1, 1, 2, 3, 3, 2, 1)(= \pi_3)$ crosses itself, since it has a W-structure of depth 3.

Theorem 4.24. Π_r is the partition category of all even-nested partitions with a w -depth of at most r .

Proof. Follows from Proposition 4.16 and Section 4 and 5 from [RW13]. \square

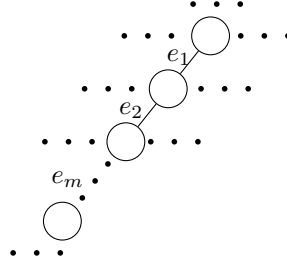
Knowing the parameter k in $p \in \Pi_{k \in \mathbb{N}}$ actually determines the degree of crossings and nestings, as measured by the w -depth, it is necessary to extend this concept of w -depth to our ordered walks. The basic intuition is that a w -depth of k in an ordered walk simply means visiting at most k consecutive edges e_1, e_2, \dots, e_k at least four times in a crossing manner.

Proposition 4.25. Let $p \in \Pi_k$ be a partition with a W-structure of depth m , then the ordered walk $w = p$ traverses m edges e_1, e_2, \dots, e_m that form a path where each e_i is adjacent to e_{i+1} . The walk follows the general visiting pattern

$$w = (e_1, \dots, e_m, e_m, \dots, e_1, e_1, \dots, e_m, e_m, \dots, e_1),$$

where additional sub-walks may occur between, before and after consecutive edges, provided that the overall ordered walk remains valid.

Proof. w visits the following sub-tree as explained in the proposition above.

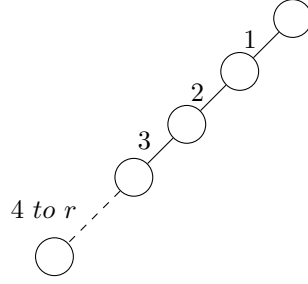


The additional sub-walks that may occur between, before and after consecutive edges are depicted by the "...". This property occurs since all the edges e_1, \dots, e_m are traversed forwards before any edge e_i is traversed backwards. \square

Theorem 4.26. The ordered walk w generated by Π_r for a fixed $r \in \mathbb{N}$ contains arbitrarily many crossings between two (not necessarily different) sub-walks p_1 and p_2 , where the total depth of p_1 plus the depth of p_2 must always be less or equal than r .

Proof. With Proposition 4.25, we understand the impact of W-structure in an ordered walk. However, Definition 4.20 says that the w -depth of a partition p is the largest r such that a rotated version of p contains a W-structure of depth r , which means that we need to investigate the impact for the ordered walk of p by rotating p . We will show that any walk p with a w -depth of r can be rotated (on the partition layer) such that we get a pattern described in Proposition 4.25. Moreover we show that this is the case if the sum of the depth of any two crossing sub-walks p_1 and p_2 of p is at most r .

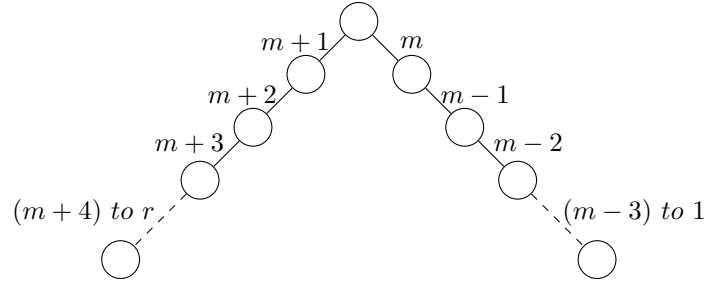
Let $p = \pi_r$ be an ordered walk, traversing the following tree.



By rotating p m times, where each rotation consists of a bottom-left and a top-right rotation (analogously, a bottom-right and top-left rotation would have a similar effect), we obtain the ordered walk

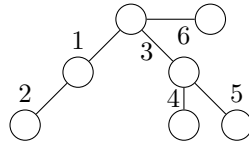
$$p' = (\underbrace{m+1, \dots, r, r, \dots, m+1}_{\text{sub-walk 1}}, \underbrace{m, \dots, 1, 1, \dots, m}_{\text{sub-walk 2}}, \underbrace{m+1, \dots, r, r, \dots, m+1}_{\text{sub-walk 1}}, \underbrace{m, \dots, 1, 1, \dots, m}_{\text{sub-walk 2}})$$

on the tree



Note that for clarity, the point values of p' are not normalized in this case. We can see that different from p where the path $1, \dots, r$ crosses itself, that in p' the paths $m+1, \dots, r, r, \dots, m+1$ and $m, \dots, 1, 1, \dots, m$ are crossing each other, where sub-walk 1 has a depth of $(r-m)$ and sub-walk two has a depth of m , which sums up to r as described in Theorem 4.26. As a result, the path from root to r simply brakes between $m+1$ and m by rotating p . To undo this, one would only have to perform the rotation back. So by rotating we can basically choose a new root in the tree, i.e. a new starting point in the walk. Consequently when we have such a crossing, we can simply choose the deepest node in one of the crossing sub-walks as the root (i.e. starting point) and get an ordered walk according to Definition 4.20 with a w -depth and a w -structure of r (see Proposition 4.25). \square

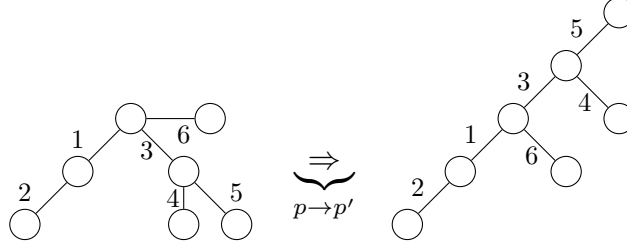
Example 4.27. Note that we will not normalize the walks/partitions in this example just to make it less confusing. Let $p = (1, 2, 2, 1, 3, 4, 4, 5, 5, 3, 6, 6, 1, 2, 2, 1, 3, 4, 4, 5, 5, 3, 6, 6)$. As a result, p is in Π_4 since the rotated version $p' = (5, 3, 6, 6, 1, 2, 2, 1, 3, 4, 4, 5, 5, 3, 6, 6, 1, 2, 2, 1, 3, 4, 4, 5)$ has the maximal possible w -structure of 4, i.e. the sequence 5, 3, 1, 2. Note that we can construct p' by performing a bottom-right and top-left rotation 4 times. The tree which is traversed by p looks as follows.



We can see that the walk p does not violate Theorem 4.26, since the following two sub-walks are crossing each other:

$$\text{walk } 1, 2, 2, 1 \text{ with } 3, 4, 4, 5, 5, 3, 6, 6.$$

From the tree we can see that the walk $1, 2, 2, 1$ has a depth of 2 as well as the walk $3, 4, 4, 5, 5, 3, 6, 6$. As a result, by rotating p to p' we changed the starting points of the walk p to the node with the edge 5 as described at the end of the proof for Theorem 4.26.



TODO now you can also specify the rules of **raum** and **weber** for a fixed k in pi_k , where we may only connect blocks which have k open blocks in between

5 Algorithmic Methods for Analyzing Properties of $\Pi_{k \in \mathbb{N}}$

An other approach for investigating properties of categories is to utilize algorithms and data structures and apply computer algebra research systems. As described and analyzed in **TODO**, published in **TODO** and implemented in **OSCAR** ([OSC25]), we make use of several algorithms, particularly the one for generating categories of partitions. Furthermore, we propose and analyze new algorithms in the domain of the category $\Pi_{k \in \mathbb{N}}$.

5.1 Applying Category generating Algorithms

Open-source computer algebra systems such as **OSCAR** are valuable tools for performing symbolic computations, verifying mathematical properties, and experimenting with algebraic structures. In this section, we demonstrate how we used our algorithms for partitions problems (designed in [Vol23] and implemented in **OSCAR**) to get the idea of drawing a connection between the partition category $\Pi_{k \in \mathbb{N}}$ and ordered walks.

Consider the following problem. Let \mathcal{C} be a category and \mathcal{G} be a set of partitions, where $\langle \mathcal{G} \rangle$ generates \mathcal{C} . Given \mathcal{G} and a $n \in \mathbb{N}$, we are interested in an algorithm to calculate the set

$$\mathcal{C} \cap P(n)$$

and

$$|\mathcal{C} \cap P(n)|,$$

i.e. all partitions in \mathcal{C} of size n .

With such an algorithm, we could generate $|\Pi_{k \in \mathbb{N}} \cap P(n)|$ for different partition sizes n and compare the results with already existing sequences in [OEI]. For this problem, we can use the algorithm described and analyzed in [Vol23] and implemented in **OSCAR**. The following code shows the computation of the twelfth number in the sequence, where we calculate the set $|\Pi_{k \in \mathbb{N}} \cap P(12)|$ using the function `construct_category`.

```
In: using Oscar
In: pi_3 = set_partition([], [1, 2, 3, 3, 2, 1, 1, 2, 3, 3, 2, 1])
In: base_partition = set_partition([1], [1])
In: length(construct_category([pi_3, base_partition], 12))
Out: 22672
```

Due to the high computational power required for partition generation and its significant complexity, we can only compute $\Pi_{k \in \mathbb{N}}$ up to a size of 12, yielding the following values.

partition size n	0	2	4	6	8	10	12
$ \Pi_{k \in \mathbb{N}} \cap P(n) $	1	3	15	84	513	3333	22672
$ \Pi_{k \in \mathbb{N}} \cap P(n, 0) $	1	1	3	12	57	303	1744

Using [OEI] we can see that the sequence **A094149** fits to the sequence of $|\Pi_{k \in \mathbb{N}} \cap P(n, 0)|$, where we filter out only partitions with upper points of size n , which is similar to our adaptation in Definition 3.2. The description of **A094149** corresponds to ordered walks on rooted plane trees.

This was the point in our research where we started to investigate ordered walks in a more detailed way, which finally brought us to the result in Theorem 4.14.

Since we have already proven the connection between $\Pi_{k \in \mathbb{N}}$ and ordered walks, in Section 4.2, we know that the results are valid. Consequently, we can use the formulas for ordered walks from Section 2.2.3 to get further results for $|\Pi_{k \in \mathbb{N}} \cap P(n, 0)|$.

$$|\Pi_{k \in \mathbb{N}} \cap P(n, 0)| = \sum_{r=0}^n W_n(r)$$

where the numbers of $W_n(r)$ with $n \geq r \geq 0$ is given by

$$W_u(v) = \sum_{i=1}^v \sum_{j=v-i}^{u-i} \sum_{l=0}^{u-i-j} W_{u-i-j}(l) \binom{l+i-1}{i-1} \binom{v-1}{i-1} W_j(v-i).$$

In order to implement the formula efficiently, we use dynamic programming (see Section 2.1.2). We implemented both a top-down and a bottom-up approach in Python, which can be found in the GitHub repository [Vol25]. The advantage of using python for such a domain is the support larger numbers than the typical fixed-size integers found in other languages. As discussed in Section 2.1.2, the bottom-up approach often outperforms the top-down approach. For a flexible and efficient implementation of top-down dynamic programming, we used `lru_cache()` from the Python library `functools`. The basic functionality of this decorator is to wrap a function to add caching functionality, implicitly applying top-down dynamic programming (see [Fou25]).

As a comparison and to test the the time needed for computing the sequence up to $|\Pi_{k \in \mathbb{N}} \cap P(n, 0)|$, we executed both approaches for different n 's.

n	20	40	60	80	100	120	140
top-down	0.02s	0.3s	2.5s	13s	40s	140s	250s
bottom-up	< 0.01s	0.2s	1.4s	7s	25s	70s	150s
digit size	16	38	62	89	118	148	179

To get a more general and formal idea of how complex the calculation process of this sequence is, we analyze its time and space complexity using asymptotic growth (see Section 2.1.1).

The calculation $|\Pi_{k \in \mathbb{N}} \cap P(n, 0)|$ consists of three different steps. First we precompute all binomial coefficients

$$\left\{ \binom{i}{j} \mid 0 \leq j \leq i \leq n \right\}$$

using bottom-up dynamic programming and the recursive formula of the binomial coefficient.

Algorithm 1: Precompute Binomial Coefficients

Input: n (the maximum value for binomial coefficients to compute)

Output: `binom` (a 2D array storing binomial coefficients up to n)

```

1 binom  $\leftarrow$  array of size  $(n+1) \times (n+1)$  filled with 0;
2 for  $i \in \{0, 1, \dots, n\}$  do
3   binom[ $i$ ][0]  $\leftarrow$  1  $\triangleright$ Base case:  $\binom{i}{0} = 1$ 
4   binom[ $i$ ][ $i$ ]  $\leftarrow$  1  $\triangleright$ Base case:  $\binom{i}{i} = 1$ 
5   for  $k \in \{1, 2, \dots, i-1\}$  do
6     binom[ $i$ ][ $k$ ]  $\leftarrow$  binom[ $i-1$ ][ $k-1$ ] + binom[ $i-1$ ][ $k$ ]  $\triangleright$ Recursive formula
7   end
8 end
9 return binom
```

This step has a time complexity of $O(n^2)$.

Subsequently, we precompute

$$\{W_i(j) \mid 0 \leq j \leq i \leq n\}$$

using a bottom-up dynamic programming approach. Since computing each $W_i(j)$ involves evaluating three nested sums, the time complexity for calculating a single $W_i(j)$ is $O(n^3)$. As we compute $W_i(j)$ for all $0 \leq j \leq i \leq n$, the overall time complexity for this step is $O(n^3 \cdot n^2) = O(n^5)$. Additionally, note that we can leverage the precomputed binomial coefficients for the calculation of $W_i(j)$.

Finally, we sum up $W_n(r)$ for all $0 \leq r \leq n$ in $O(n)$. Combining all steps, the total time complexity is $O(n^2 + n^5 + n) = O(n^5)$.

5.2 Constructing and Validating Elements in $\Pi_{k \in \mathbb{N}}$

During the process of investigating properties of different categories, it can be helpful to have algorithms that are able to decide whether a certain partition is in a specific category.

The naive approach would be to simply generate the category given the algorithm used in Section 5.1 and implemented in `Oscar`. Nevertheless, this approach has a high complexity and is only sufficient for small categories and small input partitions.

Another approach would be to use algorithms that provide direct insights into specific properties of partitions. Some of these algorithms have already been implemented in `Oscar` by us, including the functions `is_pair`, `is_balanced`, and `is_non_crossing`. These functions all run in linear time, making them significantly more efficient than the naive approach from above.

Example 5.1. Consider the problem of classifying the category of a partition. Given a partition p and the category B_2 (see Section 2.2.2). We now want to verify, whether $p \in B_2$.

Algorithm 2: Naive Approach with Category Generation

Input: p : The partition, B_2 : The category to check against

Output: True if $p \in B_2$, otherwise False

```

1 Generate  $B_2$ ;
2  $B_2 \leftarrow \text{construct\_category}([\text{producer of } B_2], \text{size}(p))$ ;
3 return  $p \in B_2$ ;
```

While this approach is sufficient for a p with $\text{size}(p) \leq 12$, it is generally considered as slow. A more efficient approach is

Algorithm 3: Efficient Approach using Property Functions

Input: p : The partition, B_2 : The category to check against

Output: True if $p \in B_2$, otherwise False

```

1  $\text{is\_pair\_partition} \leftarrow \text{is\_pair}(p)$ ;
2  $\text{is\_balanced} \leftarrow \text{is\_balanced}(p)$ ;
3 return  $\text{is\_pair\_partition} \wedge \text{is\_balanced}$ ;
```

Since this approach has a linear time complexity, it runs for inputs of size up to about 10^7 .

Currently, the only approach available to handle the input category $\Pi_{k \in \mathbb{N}}$ is the naive approach. This is because $\Pi_{k \in \mathbb{N}}$ cannot be described solely using properties such as crossing, balanced, pair, singleton, etc. Fortunately, we were able to prove the connection between $\Pi_{k \in \mathbb{N}}$ and ordered walks in Section 4.2. Consequently, given a partition p and the category $\Pi_{k \in \mathbb{N}}$, we can proceed by checking whether p also describes an ordered walk. Using this property, we can construct the following algorithm:

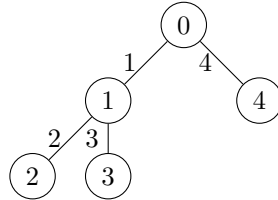
Algorithm 4: Efficiently check membership of $\Pi_{k \in \mathbb{N}}$.

Input: input partition p
Output: True if p corresponds to a valid structure, else False

```
1  $pi\_k \leftarrow p$ ;  $max\_point \leftarrow \max(pi\_k)$ ;
   /* Initialize arrays */
2 fill array  $back\_edge$  and  $visited$  with False up to index  $max\_point$ ;
   /* edge of type array[pair[]] */
3 initialize array  $edge$  up to size  $max\_point$ ;
4  $curr\_node \leftarrow 0$ ;
5 for  $curr\_edge$  in  $pi\_k$  do
6   if not  $back\_edge[curr\_edge]$  and not  $visited[curr\_edge]$  then
7     /* add new edge to tree structure in array  $edge$  */
8      $edge[curr\_edge] \leftarrow (curr\_node, curr\_edge)$ ;
9      $curr\_node \leftarrow curr\_edge$ ;
10    /* expect next occurrence of this edge to be a back edge */
11     $back\_edge[curr\_edge] \leftarrow \text{True}$ ;
12     $visited[curr\_edge] \leftarrow \text{True}$ ;
13    continue;
14   if  $visited[curr\_edge]$  and  $back\_edge[curr\_edge]$  then
15     if  $curr\_node \neq edge[curr\_edge][1]$  then
16       /* violation in tree structure detected (cycle) */
17       return False;
18      $curr\_node \leftarrow edge[curr\_edge][0]$ ;
19      $back\_edge[curr\_edge] \leftarrow \text{False}$ ;
20     continue;
21   if  $visited[curr\_edge]$  and not  $back\_edge[curr\_edge]$  then
22     if  $curr\_node \neq edge[curr\_edge][0]$  then
23       /* violation in tree structure detected (cycle) */
24       return False;
25      $curr\_node \leftarrow edge[curr\_edge][1]$ ;
26      $back\_edge[curr\_edge] \leftarrow \text{True}$ ;
27     continue;
28   /* check whether final node is the root */
29 return  $curr\_node == 0$ ;
```

In Algorithm 4, the array $edge$ is used to store, for each point value (representing an edge in the tree structure), a pair consisting of the current node and the node reached by that edge. Consequently, each node is associated with the value of the edge that first visited it.

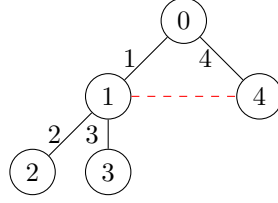
Example 5.2 (Building a rooted plane tree from a partition in $\Pi_{k \in \mathbb{N}}$). Consider the partition $(1, 2, 2, 3, 3, 1, 4, 4) \in \Pi_{k \in \mathbb{N}}$. Algorithm 4 iterates over each point value. For any new point value encountered, the algorithm stores a pair consisting of the current edge and the new point value. Note that the root node is assigned with 0 since it is a special case. The resulting rooted plane tree looks as follows



stored in the array $edges$ of the form $((), (0, 1), (1, 2), (1, 3), (0, 4))$, meaning the edge with value 1 goes from root to node 1 the edge 2 from 1 to 2 and so on.

If we have the case that the input partition is not in $\Pi_{k \in \mathbb{N}}$, the algorithm will find a violation in the tree structure.

Example 5.3 (Input partition not in $\Pi_{k \in \mathbb{N}}$). Consider the partition $(1, 2, 2, 3, 3, 1, 4, 4, 1, 4, 4, 1) \notin \Pi_{k \in \mathbb{N}}$. In this case, the algorithm constructs the tree up to the 9th point value. Since we are in the node 1 now, visiting node 4 would form a cycle. As a result, the algorithm returns **False**.








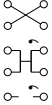









Since the algorithm iterates through the given partition only once and, in each iteration, accesses the array indices with constant time complexity, the overall time complexity is $O(n)$. The implementation of this algorithm can be found in [Vol25].

6 Discussion

References

- [Bar16] Gabriella Baracchini. Dyck paths and up-down walks. https://math.mit.edu/~apost/courses/18.204-2016/18.204_Gabriella_Baracchini_final_paper.pdf, 2016. Accessed: 2024-12-29.
- [BBC07] Teodor Banica, Julien Bichon, and Benoit Collins. The hyperoctahedral quantum group. *J. Ramanujan Math. Soc.*, 22(4):345–384, 2007.
- [BCS10] T. Banica, S. Curran, and R. Speicher. Classification results for easy quantum groups. *Pacific Journal of Mathematics*, 247(1):1–26, 2010.
- [BS09] T. Banica and R. Speicher. Liberation of orthogonal lie groups. *Advances in Mathematics*, 222(4):1461–1501, 2009.
- [FK81] Zoltán Füredi and János Komlós. The eigenvalues of random symmetric matrices. *Combinatorica*, 1(3):233–241, 1981.
- [Fou25] Python Software Foundation. `functools.lru_cache`, 2025. Accessed: 2025-01-14.
- [Knu76] Donald E Knuth. Big omicron and big omega and big theta. *ACM Sigact News*, 8(2):18–24, 1976.
- [KV00] A. Khorunzhy and V. Vengerovsky. On asymptotic solvability of random graph’s laplacians, 2000.
- [Man22] Alexander Mang. *Classification and homological invariants of compact quantum groups of combinatorial type*. PhD thesis, Saarland University, 2022.
- [OEI] OEIS Foundation Inc. The on-line encyclopedia of integer sequences. Published electronically at <http://oeis.org>.
- [OSC25] Oscar – open source computer algebra research system, version 1.0.0, 2025.
- [RW13] Sven Raum and Moritz Weber. The full classification of orthogonal easy quantum groups, 2013.
- [SR16] Moritz Weber Sven Raum. The full classification of orthogonal easy quantum groups. *Communications in Mathematical Physics*, 341:751–779, 2016.
- [Sta99] Richard P. Stanley. *Enumerative Combinatorics*, volume 2. Cambridge University Press, 1999.
- [Vol23] Sebastian Volz. Bachelor’s thesis - design and implementation of efficient algorithms for operations on partitions of sets. <https://www.uni-saarland.de/fileadmin/upload/lehrstuhl/weber-moritz/Abschlussarbeiten/volz-bachelors-thesis.pdf>, 2023.
- [Vol25] Sebastian Volz. Github repository. https://github.com/sebvz777/SetPartitions_python, 2025.
- [Web13] M. Weber. On the classification of easy quantum groups. *Advances in Mathematics*, 245:500–533, 2013.
- [Wig55] Eugene P. Wigner. Characteristic vectors of bordered matrices with infinite dimensions. *Annals of Mathematics*, 62(3):548–564, 1955.

A Overview Categories

Category	Quantum Group	Generators	Description
P	S_n		all partitions
P_2	O_n		partitions with blocksize 2
$P^{1,2}$	B_n		partitions with blocksize 1 and 2
$P_{even}^{1,2}$	B'_n		even part of partitions with blocksize 1 or 2
P_{even}	H_n		partitions with even blocksize
NC_2	O_n^+	-	non-crossing partitions with blocksize 2
P_{even}	S'_n		even part of all partitions
NC	S_n^+		non-crossing partitions
NC_{even}	H_n^+		non-crossing partitions with even blocksize
$NC^{1,2}$	B_n^+		non-crossing partitions with blocksize 1 and 2
NC_{even}	$S_n^{'+}$		even part of non-crossing partitions
$NC_{even}^{1,2}$	$B_n^{'+}$		even part of non-crossing partitions with blocksize 1 and 2
NCB_{even}	$B_n^{\# +}$		non-crossing partitions with balanced pairs and even number of singletons
B_2	O_n^*		balanced partitions with blocksize 2
B	H_n^*		balanced partitions
$B_{even}^{1,2}$	$B_n^{\# *}$		all balanced partitions of block size one and two with an even number of singletons
Π_k	hyperoctahedral and not group-theoretical	π_k	all even-nested partitions of w -depth $\leq k$