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

Dr. Nicolas Faroß

Prof. Dr. Moritz Weber

submitted

Reviewers

Prof. Dr. Moritz Weber

Prof. Dr. Markus Bläser

Erklärung
Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Zur sprachlichen Ausgestaltung (insbesondere Formulierungen und Zeichensetzung) habe ich in einzelnen Fällen KI verwendet.

Statement
I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis. For the linguistic refinement (in particular phrasing and punctuation), I have, in some cases, used AI.

Einverständniserklärung
Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik/Mathematik aufgenommen und damit veröffentlicht wird.

Declaration of Consent
I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science/Mathmatics Department.

Saarbrücken

Abstract

This thesis investigates the structure and algorithmic properties of the category of partitions Π_∞, which plays a central role in the theory of easy quantum groups. These Categories of partitions were first introduced by T. Banica and R. Speicher in 2009 as a combinatorial framework for describing easy quantum groups, and were later fully classified by S. Raum and M. Weber in 2016. We prove that the category Π_∞ 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 used 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 Π_∞ up to a partition size of 300 in a few seconds. Furthermore, we propose efficient algorithms to decide whether a specific partition is contained in Π_∞.

Acknowledgments

I want to thank both Moritz Weber and Nicolas Faroß for all the opportunities they gave me as well as for the excellent supervision. I am grateful for all the things Moritz made possible for me, including those beyond this thesis. In the last years as a member of his chair, I had multiple opportunities to gain wonderful experiences, starting with my bachelor thesis, continuing with well-organized exhibitions like *Women and Mathematics*, then great job possibilities as a student research assistant, and even the *ISSAC 2025* conference in Guanajuato, Mexico. Every single experience was great and really shaped my studies in a very positive way.

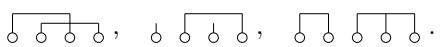
In particular, I would like to thank Nicolas Faroß for his outstanding supervision and guidance, which was of the highest quality in every area possible. He was such a competent and dedicated supervisor that I can only express my deepest gratitude.

Furthermore, I would like to thank my friends and family for their support both outside and during my studies. Especially Lisa Heidmann, who was and is always there for me no matter what, as well as Yvonne Neuy and Robert Rabbe, with whom I had great and funny discussions about our studies and beyond.

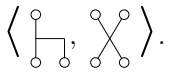
Contents

1 Introduction

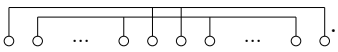
In this thesis, we study the category of partitions Π_∞. More precisely, we propose algorithms for classifying the elements of Π_∞ and draw a connection to ordered walks. Categories of partitions are related to easy quantum groups, as defined in [BCS10], with their classification completed in [SR16]. 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 $S = \{1, ..., k+l\}$ 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 lower points, where $k = 0$. 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 contains the base partitions $\sqcap \in P(2,0)$ and $| \in P(1,1)$. For example, the following set of partitions is the category P of all partitions



In this case, the partitions $\sqcap$ and $\times$ are the generator partitions of the category of all partitions. The category Π_∞ is defined and investigated in [SR16]. It is generated by π_k for $k \in \mathbb{N}$, given by k four blocks on $4k$ points of the form

$\pi_k =$ 

Besides the generator representation of the category Π_∞, 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 Π_∞ via the notion of non-interference between blocks and a so called betweenness relation in [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 started to occur 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 Π_∞ and ordered walks on rooted plane trees. In a rooted plane tree, each node has an ordered sequence of children. These trees naturally occur in various combinatorial problems, particularly in our study we focus on counting the different combinations of traversing them.

An ordered walk is a way to traverse a rooted plane tree with some additional conditions that need to be met. It must start and end at the root of the tree. Furthermore, each edge of the tree is visited an even 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 each edge in the tree exactly twice.

We first show that a simple walk is related to the category NC_2, all non-crossing partitions of block size two.

Proposition 1.1 (Proposition 4.3)**.** *Let W_s be the set of all simple walks and NC_2 be the category generated by the base partitions, then $NC_2 = W_s$.*

With this finding, we are additionally able to prove a bijection between ordered walks and the category Π_∞. While S. Raum and M. Weber already discussed a correspondence between trees and the category NC_2 in [RW13], we deepened their construction and also went one step further by generalizing this concept to the category Π_∞ with ordered walks, which forms the main theorem of this thesis.

Theorem 1.2 (Theorem 4.17)**.** *Let W be the set of all ordered walks, then $\Pi_\infty = W$.*

Additionally, we analyzed the impact of the parameter k in Π_k for $k \in \{1, \dots, \infty\}$. As a result, we derived an additional characterization of the elements in Π_∞, namely even-nested property, which we were able to prove using ordered walks.

Theorem 1.3 (Theorem 4.29)**.** *Π_k is the category of partitions of all even-nested partitions with a w-depth of at most k.*

Furthermore, we showed that this definition is similar to non-interference defined by M. Mang in [Man22]. Although this meant our definition is technically not a new finding, it allowed us to further validate Mang's work.

Moreover, we establish a connection to ordered walks for a fixed $k \in \{1, \dots, \infty\}$ in Π_k and investigate the impact on ordered walks.

Theorem 1.4 (Theorem 4.31)**.** *Let $w \in \Pi_k$ for a fixed $k \in \mathbb{N}$. Then for every pair of sub-walks w_1 and w_2 in w that cross each other, the sum of their depths satisfies*

$$\operatorname{depth}(w_1) + \operatorname{depth}(w_2) \leq k.$$

By proving Theorem 1.2 in Section 4.2, we were also able to explore the category Π_∞ further and implemented an algorithm that, given a partition p of size n, decides whether $p \in \Pi_\infty$ with a time complexity of $O(n)$.

Theorem 1.5 (Theorem 5.6)**.** *Given a partition p of size n, the problem of deciding whether $p \in \Pi_\infty$ can be solved in time $O(n)$, which coincides with the lower bound $\Omega(n)$ of this problem. Thus, the algorithm is asymptotically optimal.*

Additionally, we present an algorithm, which computes a specialized version of the problem in Theorem 1.2, by computing the w-depth of a partition p in Π_∞. With this information, we can find the smallest k such that $p \in \Pi_k$.

Theorem 1.6 (Theorem 5.8)**.** *Let $p \in \Pi_\infty$ be a partition of size $n \in \mathbb{N}$. Then the problem of computing the smallest k such that $p \in \Pi_k$ can be solved in $O(n^2)$.*

Moreover, the equivalence to ordered walks enabled us to count the category Π_∞ 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, Section 2, we introduce key preliminaries from both computer science (Section 2.1) and mathematics (Section 2.2). Starting with techniques in the areas 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). In addition, in Section 2.2.3, we introduce the mathematical foundation for counting ordered walks, for which we need to define moments of adjacency matrices of random graphs.

The second part (Section 3 and Section 4) finally introduces the partition category Π_∞ including some relevant properties. Furthermore, it contains the proofs for both Proposition 1.1 and Theorem 1.2, forming the main results of this thesis. Additionally, in Section 4.3, we explore the category Π_∞ in more detail, summarizing existing definitions and presenting a new formulation using ordered walks. We conclude the second part by analyzing the meaning of the parameter k in $\Pi_{k\in\{1,\dots,\infty\}}$ for ordered walks, which results in Theorem 4.31.

Based on the foundation of the second part, the third part (Section 5) describes and analyzes various implementations to classify the category Π_∞, particularly its elements. From applying generating algorithms to efficiently implementing the problem of deciding partitions in Π_∞. 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, as well as the mathematical background. First, we define some aspects in the area of computer science, such as Big-O notation, dynamic programming, and data structures as different kinds of trees, followed by mathematical subjects, such as partitions of sets, their categories and some basics in free probability theory, which is relevant for later parts of this thesis.

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 Π_∞. In particular, we introduce asymptotic runtime notation (Big-O, Ω), 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 Ω notation. Further and more detailed information can be found in [Knu76].

Definition 2.1 (Big-O notation)**.** Let $g : \mathbb{N} \to \mathbb{N}$ be a function. We define $O(g)$ as the following set:

$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{N} \mid \exists_{c>0}, \exists_{n_0>0}, \forall_{n\geq n_0} : 0 \leq f(n) \leq c \cdot g(n)\}.$$

In other words, $f \in O(g)$ 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

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

Example 2.3. The following examples apply.

(a) Let $f : \mathbb{N} \to \mathbb{N},\ n \mapsto 100$. Then
$$f \in O(1).$$
If $f : \mathbb{N} \to \mathbb{N},\ n \mapsto n$. Then
$$f \notin O(1).$$

(b) Let $g : \mathbb{N} \to \mathbb{N},\ n \mapsto 100n$. Then
$$g \in O(n).$$
If $g : \mathbb{N} \to \mathbb{N},\ n \mapsto n \cdot log(n)$. Then
$$g \notin O(n).$$

(c) Let $g : \mathbb{N} \to \mathbb{N},\ n \mapsto n^k$, 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
$$f \in O(g).$$

Definition 2.4 (Ω-notation)**.** Let $g : \mathbb{N} \to \mathbb{N}$ be a function. We define $\Omega(g(n))$ as the 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 $k \in \mathbb{N}$. 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 subresults instead of recalculating them multiple times. This is achieved by storing subresults 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. We will explain this technique with an example on the *Fibonacci sequence*.

Definition 2.7 (Fibonacci Sequence)**.** The *Fibonacci sequence* $(F_n)_{n\in\mathbb{N}}$ is defined recursively by the case distinction

$$F_n = \begin{cases} 0, & \text{if } n = 0, \\ 1, & \text{if } n = 1, \\ F_{n-1} + F_{n-2}, & \text{if } n \geq 2. \end{cases}$$

Example 2.8. (Computing the Fibonacci sequence) Consider the problem of computing the Fibonacci sequence up to a certain number n. The naive standard implementation in `Python` 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 subresult.

Example 2.9 (Computing the Fibonacci sequence bottom-up)**.**

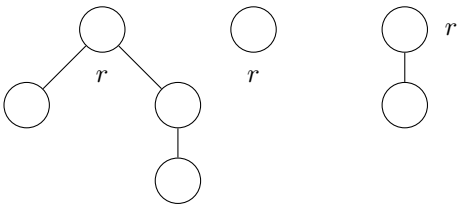
```
def fib_bottom_up(n):
    fib = [0, 1]
    for i in range(2, n+1):
        fib.append(fib[i-1] + fib[i-2])
    return fib[n]
```

2.1.3 Rooted Plane Trees

To establish a connection between *rooted plane trees* and the category Π_∞ in Section 4.2, we will introduce *ordered walks* on such trees. This connection enables us to investigate further properties of the category Π_∞.

Definition 2.10 (Rooted tree)**.** A *rooted tree* is a connected, acyclic graph $T = (V, E, r)$ 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 2.11 (Rooted trees)**.**

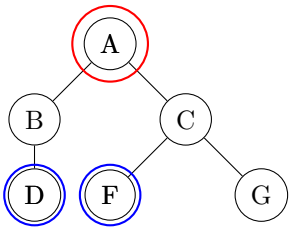


In further illustrations, we also mark the root by drawing it as the top node. Since rooted trees will play a central role in later arguments, we now introduce key structural properties that will be used throughout this thesis.

Definition 2.12. Let (V, E, r) be a rooted tree. Then the depth of a node $v \in V$ is the length of the minimal path from v to r. The depth of an edge $e \in E$ is the minimal number of edges from e to r including e itself. For a rooted tree (V, E, r), the depth of the tree is defined as the maximum distance from the root r to any other node in the tree.

Definition 2.13 (Lowest common ancestor)**.** Given a rooted tree with root r, and two vertices u_0 and v_0, let the sequences $u_0, u_1, \ldots, u_n, r$ and $v_0, v_1, \ldots, v_m, r$ 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 2.14 (LCA)**.** In this example, we can see that the LCA of D and F is the root A.

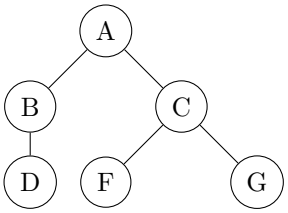


Definition 2.15. Let $T = (V, E)$ be a tree with vertex set V and edge set E. The diameter of T is given by

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

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

Example 2.16 (Diameter)**.** The diameter of the tree below is 4, since the longest path in the tree is 4 from G to D.



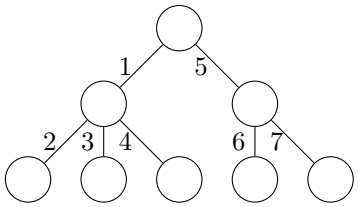
With rooted trees as a foundation, we introduce *rooted plane trees*. This data structure is an important part of the main theorem of this thesis, as *ordered walks* (defined in the next section) act on rooted plane trees.

Definition 2.17 (Rooted plane tree)**.** A *rooted plane tree* (V, E, r, α) is a rooted tree with root $r \in V$. The function $\alpha : E \to \mathbb{N}$ is injective and assigns a unique integer to each edge. The function α maintains a left-to-right order on the children of every node, where for each node u with children $v_1, \ldots, v_k$, it holds that

$$\alpha((u, v_1)) < \alpha((u, v_2)) < \cdots < \alpha((u, v_k)).$$

Consequently, the tree has a unique drawing, making it planar.

Example 2.18 (Rooted plane tree)**.**



2.1.4 Ordered Walks

We now define *ordered walks* on rooted plane trees introduced in [KV00]. Additionally, we recall a connection to *Dyck paths* defined in [Bar16] from [RW13].

Definition 2.19 (Ordered walks on rooted plane trees)**.** An *ordered walk* on a rooted plane tree can be defined as a sequence of edges $(e_1, e_2, ..., e_n)$, 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} (with $e_i \neq e_{i+1}$) 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 already have been visited or the edge with the smallest value among those that have not yet been visited

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.

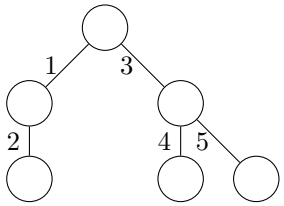
1. Note that the same ordered walk remains identical across different trees. This means that, for example, the ordered walk
$$(1,\ 2,\ 2,\ 1)$$
on Example 2.18 is classified as the same as if the tree consisted only of edges 1 and 2.

2. By definition, the sequence
$$w = (3, 2, 2, 4, 4, 3)$$
would be a valid ordered walk. However, it is often better to work with the normalization of such a walk similar to the partitions in Section 3.1, where we also normalize the point values. We say that an ordered walk $w = (e_1, \ldots, e_n)$ of size n is normalized if $e_1 = 1$ and $e_j \leq \max(e_i)_{1 \leq i < j \leq n} + 1$. As a result, the normalization of the walk above is
$$w' = (1, 2, 2, 3, 3, 1).$$
In general, we only consider normalized ordered walks, unless it is explicitly mentioned different.

3. We say that two ordered walks are equal, if their normalization is equal.

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.

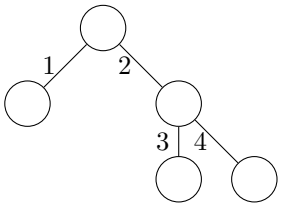
Example 2.23. In this example, we want to clarify a few misunderstandings that can occur when combining rooted plane trees of walks and the property of its normalization (see Remark 2.21). Consider the two ordered walks $w_1 = (1, 1, 2, 3, 3, 4, 4, 2)$ and $w_2 = (1, 2, 2, 1, 3, 4, 4, 5, 5, 3)$ and the 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 $w' = (e_{i_1}, e_{i_2}, \dots, e_{i_k})$ with $1 \leq i_1 \leq i_2 \leq \cdots \leq i_k \leq n$ is a *sub-walk* of w, if w' itself (or at least its normalization) is an ordered walk as well as $w \backslash w'$ (i.e. w but without the points of w').

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.

(e) Consider the walk $w = (e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8) = (1, 2, 2, 1, 1, 2, 2, 1)$. Then $w' = (e_4, e_8) = (1, 1)$ is not a sub-walk of w since $w \setminus w' = (1, 2, 2, 1, 2, 2)$ is not a valid ordered walk.

Definition 2.26. 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.

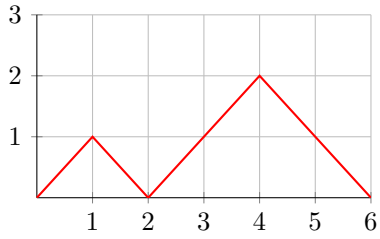
Since ordered walks are a relatively general concept, we introduce a simpler variant that will later serve as the basis for inductive proofs, namely *simple walks*.

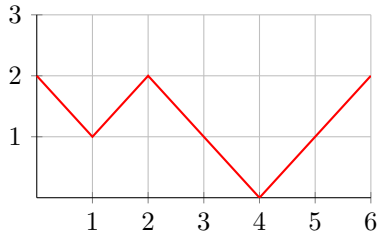
Definition 2.27 (Simple walks on rooted plane trees)**.** A *simple walk* on a rooted plane tree is an ordered walk where every edge is visited at most twice within the simple walk, making it a special case of ordered walks.

In further sections, we will see that ordered walks, rooted plane trees, and *Dyck paths* are closely connected. Therefore, we also define Dyck paths and introduce relevant properties.

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.



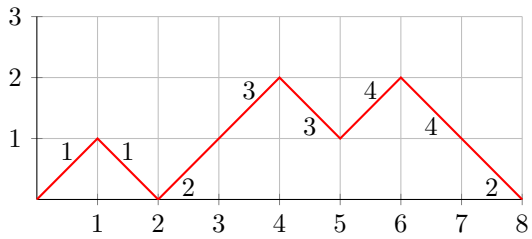


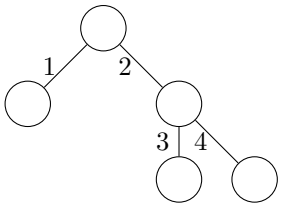
Remark 2.30. Such a Dyck path can illustrate a simple walk on a rooted plane tree. Let us specify the y-value as the *level*. The step $(1,1)$ indicates, that we visit a new edge of the tree for the first time at *level* n, while a step $(1,-1)$ at *level* n represents returning along this edge.

Example 2.31 (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 2.30 and a rooted plane tree with the rules of 2.22.





2.2 Mathematical Background

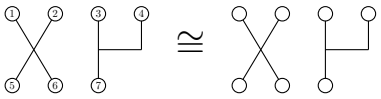
In this section, we first define classical partitions of sets in Section 2.2.1, where we proceed by introducing categories of partitions. In Section 2.2.3, we introduce some basic concepts of free probability to later be able to analyze and count ordered walks. In Section 3, we focus more specifically on the domain of this thesis, where we slightly adapt the definition of partitions to suit our purposes.

2.2.1 Partitions

In this section, we introduce partitions of sets, which serve as one of the central combinatorial structures throughout this thesis, besides ordered walks. As will be explained in the following section and in Section 2.2.2, we can define operations on them as well as classify their so-called categories of partitions. Understanding how these partitions are defined and how operations can be applied to them is essential for connecting them later to ordered walks. T. Banica and R. Speicher first introduced them in 2009 to classify *easy quantum groups*, where later S. Raum together with M. Weber completed their classification in 2016. 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 in Section 3 in order to obtain a data structure that can be processed efficiently for the domain of this thesis (see Definition 3.2).

Definition 2.32 (Partition)**.** Let $k,l \in \mathbb{N}$. A *partition* p is given by k lower and l upper points. It partitions the set $\{1,...,k,k+1,...,k+l\}$ into disjoint subsets called blocks. Partitions can also be visualized. Given $p \in P(k,l) \subseteq P(n)$ with $n = k+l$, where $P(k,l)$ is the set of all partitions with k upper and l lower points and $P(n)$ the set of all partitions of size n, 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 $\{1,...,k,...,k+l\}$) we connect them according to the blocks defined in p.

Example 2.33. 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:

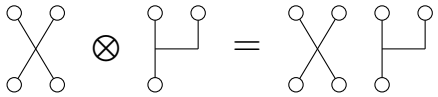
$\{\{1,6\},\{2,5\},$
$\{3,4,7\}\}$ $\cong$  $\cong$

Remark 2.34. The partitions $\sqcap \in P(0,2)$ and $\vert \in P(1,1)$ are referred to as the *base partitions*.

After defining partitions of sets, we introduce operations on them to provide a structured way to manipulate, process and combine partitions. This allows us later in Section 2.2.2 to define categories of partitions, which are sets of partitions closed under these operations.

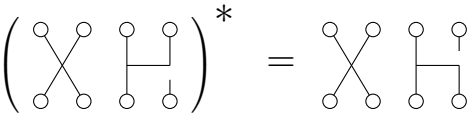
Definition 2.35 (Tensor Product)**.** Let $p \in P(k_1, l_1)$ and $q \in P(k_2, l_2)$. Then the *tensor product* can be obtained by concatenating p and q in form of $p \otimes q \in P(k_1 + k_2, l_1 + l_2)$.

Example 2.36 (Tensor Product)**.**



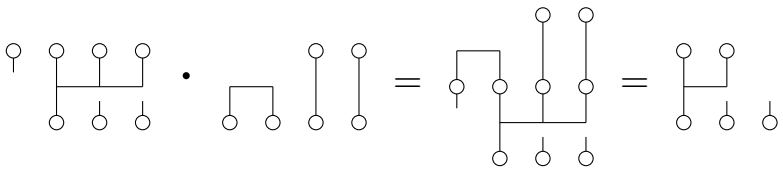
Definition 2.37 (Involution)**.** Let $p \in P(k,l)$. Then the unary operation *involution* $p^* \in P(l,k)$ can be obtained by swapping the upper and lower points.

Example 2.38 (Involution)**.**



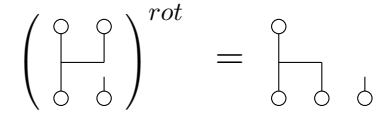
Definition 2.39 (Composition)**.** Let $p \in P(k_1, l_1)$, $q \in P(k_2, l_2)$ and $k_1 = l_2$. Then the *composition* $pq \in P(k_2, l_1)$ can be obtained by connecting p and q by writing q above p and joining each lower point of q with the respective upper point of p. As a result, we connect every 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 2.40 (Composition)**.**



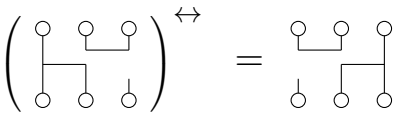
Definition 2.41 (Rotation)**.** Let $p \in P(k,l)$ and $k > 0$. Then the unary operator *rotation* $p^{rot} \in P(k-1, l+1)$ can be obtained by shifting the very left upper point to the very left of the lower points. Note that all points belong to the same blocks as before. The result of this operation 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 2.42 (Rotation)**.** The following figure shows an example of a *left-top rotation*:



Definition 2.43. 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.

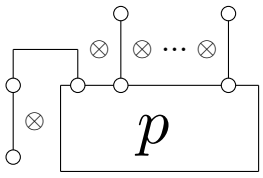
Example 2.44 (Vertical Reflection)**.**



Definition 2.45 (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* $\sqcap \in P(0,2)$ and $| \in P(1,1)$.

Remark 2.46. 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 $\sqcap \in P(0,2)$ and $| \in P(1,1)$.

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 $(| \otimes p) \cdot (\sqcap \otimes | \otimes ... \otimes |)$, 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 $| \in P(1,1)$ and $\sqcap \in P(0,2)$. □

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.

$$p^{\leftrightarrow} = \left(p^{\mathrm{rot}\,(\dots)^{\mathrm{rot}}}\right)^*$$

□

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. Later 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. The following chapter defines them and provides concrete examples, while Section 3 introduces the category Π_∞, 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 2.47 (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 $| \in P(1,1)$ and $\sqcap \in P(0,2)$, and is closed under the *category operations*. This results in 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 $\langle S \rangle$ a category that is closed under the set of generator partitions S with the base partitions.

We can characterize categories of partitions by defining some core properties. Besides self-explanatory properties like *even-blocksize* and *pair partitions* (partitions with blocksize two), there are a few more interesting characteristics like *non-crossing* partitions and *balances* partitions. Especially the non-crossing property will be relevant in later parts of this thesis, when analyzing the category Π_∞ more detailed. Additionally, in Section 4.3, we introduce new category characteristics that will help us analyze the category Π_∞.

Definition 2.48 (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 2.49 (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 2.50 (Categories of Partitions)**.** The following table lists several example categories of partitions along with the generator partitions that define them. Some of these categories will be explored in later sections of this thesis.

Category	**Generators**
P (all partitions)	$\langle$  $\rangle$
NC (non-crossing partitions)	$\langle$  $\rangle$
P_2 (pair partitions)	$\langle$ $\rangle$
NC_2 (non-crossing pair partitions)	$\langle\ \rangle$ (base partitions only)
NC_{even} (non-crossing, even size)	$\langle$  $\rangle$
B (balanced partitions)	$\langle$ $\rangle$
B_2 (balanced, block size two)	$\langle$ $\rangle$

In Section 3, we will introduce the category Π_∞. For more relevant categories of partitions, see Appendix A.

Remark 2.51. 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. The definitions for those basic concepts can be found in [Ros14].

Definition 2.52 (Discrete Probability Space)**.** A *discrete probability space* is a pair $(\Omega, \mathcal{P})$, where:

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

Example 2.53. 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 $\mathcal{P} : \Omega \to [0, 1]$ as:

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

This gives us the discrete probability space $(\Omega, \mathcal{P})$, 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 2.54 (Random Variable)**.** A *random variable* is a function $X : \Omega \to \mathbb{R}$ from a sample space in Ω that assigns a real number to each outcome in Ω.

Example 2.55. Consider again the experiment of rolling a six-sided dice, but this time twice. The probability space is $(\Omega, \mathcal{P})$, where

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

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

We can now define our random variable $X : \Omega \to \mathbb{R}$ 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 2.56 (Expected Value)**.** Let $(\Omega, \mathcal{P})$ be a discrete probability space and let $X : \Omega \to \mathbb{R}$ 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 2.57 (Moments of a Random Variable)**.** Let $(\Omega, \mathcal{P})$ be a discrete probability space, $X : \Omega \to \mathbb{R}$ 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 2.58. 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 2.59 (Random Graph)**.** A *random graph* on N vertices is an undirected graph (V, E) in which the edges are drawn independently 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 2.60. We depict random graphs (E, V) by their adjacent matrix representation $A = (f_{ij})_{i,j=1}^{|V|}$ with

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

where the matrix A is symmtric with $f_{ij} = f_{ji}$.

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.

In classical probability, random variables are real-valued functions defined on a sample space $(\Omega, \mathcal{P})$, where expected values are derived from an underlying probability measure. In contrast, free probability theory replaces this structure with a non-commutative algebra $\mathcal{A}$, in which elements play the role of random variables, and a linear functional $\mathbb{E} : \mathcal{A} \to \mathbb{R}$ that serves as a non-commutative expectation. For example, in Remark 2.60 the entries f_{ij} from the matrix can be seen as random variables and the *normalized trace* of the matrix (defined later in Definition 2.62) as the expectation function.

Thus, the classical pair $(\Omega, \mathcal{P})$ is replaced by $(\mathcal{A}, \mathbb{E})$, where $\mathcal{A}$ encodes the algebra of random matrices and the expectation function $\mathbb{E}$ often corresponds to the normalized trace. This definition enables the analysis of complex random systems, especially those involving large random matrices, without relying on point wise probability measures. This framework is particularly useful for asymptotic spectral analysis, as shown in [KV00].

Remark 2.61. All information contained in the classical probability space $(\Omega, \mathcal{P})$ 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 $(\Omega, \mathcal{P})$ are also encoded in $(\mathcal{A}, \mathbb{E})$. □

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.

Definition 2.62 (Trace of a Matrix)**.** Let $M \in \mathbb{R}^{n \times n}$ be a square matrix. The *trace* of M, denoted by $\mathrm{Tr}(M)$, is defined as the sum of the entries on the main diagonal of M, more precisely,

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

Example 2.63. 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.

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

Definition 2.64 (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 $\mathbb{E} : \mathcal{A} \to \mathbb{R}$ by

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

where $\mathrm{Tr}(M)$ 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 $(\Omega, \mathcal{P})$ with the free probability space $(\mathcal{A}, \mathbb{E})$. With this new setting, A. Khorunzhy and V. Vengerovsky were able 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 $\frac{1}{p}$, and $\mathbb{E}$ our expected value calculated by the trace of A. Then $\lim_{N\to\infty} M_s^N$ counts the number of ordered walks on a rooted plane tree of length s, where

$$M_s^N = \mathbb{E}([A^{(N,p)}]^s) = \frac{1}{N} \sum_{x_i=1}^{N} A_{x_1 x_2} A_{x_2 x_3} \cdots A_{x_{s-1} x_1}.$$

Example 2.65. 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

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

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], A. Khorunzhy and V. Vengerovsky were able to find a closed formula for those walks denoted by

$$\lim_{N\to\infty} M_s^N = \begin{cases} m_k, & \text{if } s = 2k, \\ 0, & \text{if } s = 2k+1, \end{cases} \tag{1}$$

with

$$m_k = \sum_{r=0}^{k} W_k(r) \tag{2}$$

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

$$W_u(v) = \sum_{i=1}^{v} \sum_{j=v-i}^{u-i} \sum_{l=0}^{u-i-j} W_{u-i-j}(l) \binom{l+i-1}{i-1} \binom{v-1}{i-1} W_j(v-i) \tag{3}$$

Admittedly, it is not obvious from its appearance that formula (3) accurately counts ordered walks. In the following, we aim to motivate the structure of the formula by unpacking its components and illustrating how it encodes the recursive construction of such ordered walks on rooted plane trees. In the closed formula of (3), $W_u(v)$ counts all ordered walks of length $2u$ that return v times to the root. Consider an arbitrary set of ordered walks with the walks $w_k = (w_1, \ldots, w_{2u})$ and its rooted plane trees of the form $t = (V, E, r, \alpha)$. Let 1 be the value of the edge $e_1 = (r, v_1) \in E$ that is taken by the walk as a first step in the rooted plane tree t. Now we say that e_1 is passed $2i \geq 2$ times, i times forth, and i times back.

Subsequently, we can see, that the sub-walks of w_k, starting and ending at node v_1 (where v_1 is the root), can also be seen as valid walks, traversing the rooted plane tree of t where v_1 is the root and r is ignored. With the above in mind, we can again look at the formula (3) and further investigate its components.

$$W_u(v) = \sum_{i=1}^{v} \sum_{j=v-i}^{u-i} \sum_{l=0}^{u-i-j} \underbrace{W_{u-i-j}(l)}_{\text{(3.1) first sub-walk}} \cdot \underbrace{\binom{l+i-1}{i-1}}_{\text{(3.2) insert } v_1 \to r} \cdot \underbrace{\binom{v-1}{i-1}}_{\text{(3.3) insert } r \to v_1} \cdot \underbrace{W_j(v-i)}_{\text{(3.4) second sub-walk}} \tag{3}$$

Let the first sub-walks w_{sub_1} be those that start and end at v_1. The second sub-walks are defined as those that start and end at r and never visit v_1. More precisely, we define the second sub-walks as

$$w_{\text{sub}_2} = (w_k \setminus w_{\text{sub}_1}) \setminus (r, v_1),$$

where $\setminus(r, v_1)$ indicates that the walk avoids the edge (r, v_1) entirely. The number of first sub-walks is given recursively by $W_{u-i-j}(l)$, whereas the second sub-walks are counted recursively by $W_j(v-i)$. Recall that the general case of $W_u(v)$, counts all ordered walks of length $2u$, which return v times to the root. Consequently, we can deduce that $2j$ is the length of the second sub-walks, which returns $v-i$ times to its root r. The first sub-walks are of length $2(u-i-j)$ and return l times to its root v_1. Also, recall that the edge (r, v_1) is traversed $2i$ times. Since the ordered walks in the overall set of walks we want to count with $W_u(v)$ is staring and ending at r, we have $2i - 1(\text{starting}) - 1(\text{ending}) = 2(i-1)$ other traversals of (r, v_1) that we need to take into account for sub-walks one and two. Therefore, component (3.2) counts the first $i-1$ occurrences of edge (v_1, r) in the first sub-walks, where the first sub-walks return l times to v_1 and thus have l possibilities to additionally return back to r. Furthermore, component (3.3) counts how the remaining $i-1$ traversals of (r, v_1) are distributed in the second sub-walks. Since the second sub-walks return $v-i$ times to the root r, we get $v-i$ slots where we could visit the edge (r, v_1). As a result, we count the combinations of distributing $i-1$ visits for $v-i+i-1 = v-1$ slots in the second sub-walks.

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 Π_∞, which are connected to ordered walks (proven in Section 4). For more details of the derivation of this formula see [KV00].

3 The Category Π_∞

In 2009, T. Banica and R. Speicher introduced categories of partitions as a combinatorial framework for describing easy quantum groups. In [BS09] and later in [Web13] all seven non-crossing categories of partitions and all six crossing categories of partitions were classified. The crossing categories contain the partition  , which reflects that the underlying system is commutative. Those thirteen categories are also called *non-hyperoctahedral*. Later in [RW14] and [RW15] hyperoctahedral categories, containing  , were classified. However, there is a third kind of category which is actually related to Π_∞. The main theorem of [SR16] 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 $k \in \{1, 2, \ldots, \infty\}$.*

That said, the category Π_∞ is relevant in the context of easy quantum groups, making it a subject of further investigation.
From the following chapter on, we define partitions where we not distinguish between upper and lower points so that we can focus on the partition structure, 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

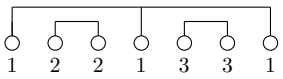
We start by slightly adapting the definition of partitions so that we remove the distinction between upper and lower points, because we are only interested in the partition structure as a sequence. This can be done because we can still recover the upper and lower points from our new definition of partitions by the rotation operation.

Definition 3.2. Let $n \in \mathbb{N}$. A *partition* p is a tuple $(p_1, \ldots, p_n) \in \mathbb{N}^n$, where n is the size of p, such that each point p_i is connected (i.e. in the same partition set/block) to any point p_j if $p_i = p_j$.

This is similar to the partition in Definition 2.32, 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. More precisely, for a partition $p = (p_1, \ldots, p_n)$, we say that $p_1 = 1$ and $p_j \leq \max(p_i)_{1 \leq i < j} + 1$.

Now that we reformulated the definition of partitions, we start to introduce the category Π_∞ by first introducing every relevant aspect needed for the definition.

Definition 3.6 (Open blocks)**.** Let p be a partition of the form $(p_1, p_2, \ldots, p_n)$ consisting of $m \in \mathbb{N}$ different blocks $b_1, \ldots, b_m$. For two blocks b_i and b_j with $i < j \leq m$, we define the set of *open blocks* between b_i and b_j as the set of all blocks b_k with $k \neq i \neq j$ such that:

- b_k has a point $p_l \in b_k$ that occurs before any point $p_r \in b_i$ $(l < r)$ or after any point $p_r \in b_j$ $(r < l)$, and
- b_k also has an odd number of points that occur between any two points of b_i and b_j.

Remark 3.7. For a partition p in NC_2, we can refer to the open blocks as *open pairs*, since in NC_2 every block consists of exactly two points.

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 blocks (or in this case 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:

- has a point occurring before any point of b_3 or after any point of b_5, and
- also has an odd number of points 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:

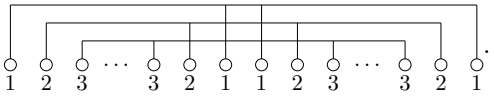
- The point $p_2 = 2$ occurs before any point of b_3.
- The point $p_9 = 4$ occurs after any point of b_5.
- The points $p_5 = 2$ and $p_6 = 4$ are between all points of b_3 and b_5.

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

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

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

More precisely, for $k \geq 3$:

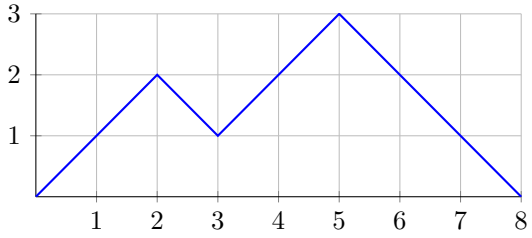


Definition 3.10. The category Π_∞ can be defined by the generator set $\langle \pi_k, k \in \mathbb{N} \rangle$. Π_k for $k \in \mathbb{N}$ is generated by $\langle \pi_k \rangle$.

Additionally to the definition of the generator sets, S. Raum and M. Weber defined Π_∞ via open pairs and Dyck paths.

Remark 3.11. Let p be a partition in NC_2, then p can be depicted by a Dyck path, where for each first appearance of a new point value corresponds to a $(1, 1)$ step in the Dyck path and each second (last) appearance to a $(1, -1)$ step.

Example 3.12. Consider the partition $p = (1, 2, 2, 3, 4, 4, 3, 1)$. Then the corresponding Dyck path looks as follows.



Definition 3.13. Let $p \in \Pi_\infty = \langle \pi_k, k \in \mathbb{N} \rangle$ and $q \in NC_2$, then p can be constructed from q by merging blocks of q according to the following rules:

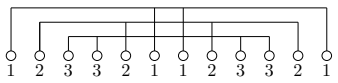
1. We may only merge blocks which are on the same level of the Dyck path of q.
2. If we connect two blocks b_1 and b_2 in q, we must also connect all open pairs between b_1 and b_2.

From now on, when writing Π_∞ or $\Pi_{k \in \{1,\dots,\infty\}}$, we refer to the category $\bigcup_{k \in \mathbb{N}} \Pi_k(0, n)$ with $n \in \mathbb{N}$.

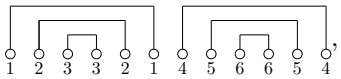
Before diving deeper towards our main result Theorem 4.17 in Section 4.2 which formally establishes a connection between the category Π_∞ and ordered walks on rooted plane trees, we already offer some intuition through the following example. Later, we will formally show that elements in Π_∞ can be interpreted as such walks, where each block corresponds to a traversed edge and the structure of the walk is encoded by partitions. In this example, we illustrate how the merging rules from Definition 3.13 affect the corresponding rooted plane tree representation, which is shown in Lemma 4.15. This provides an early conceptual bridge between the elements in Π_∞ and ordered walks, helping to build a clear foundation for understanding the categorical and combinatorial mechanisms behind the main result in Theorem 4.17.

Example 3.14. In this example, we go deeper into the rules of Definition 3.13, 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 $\pi_3 \in \Pi_\infty$ of the form

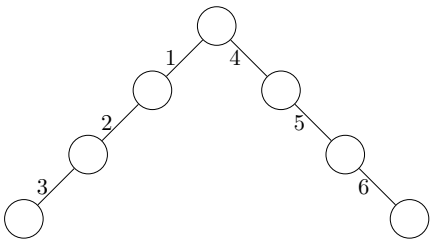


can be retrieved by modifying $p \in NC_2$ of the form



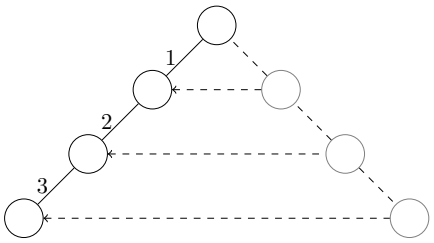
through the merging of blocks 3 and 6. According to the second rule of Definition 3.13, we also have to connect the blocks $1, 4$ and $3, 5$. Note that every block that we merged was on the same Dyck level in p, thus satisfying the first rule of Definition 3.13.

(b) Since the rules of Definition 3.13 are crucial for understanding the proof of our main theorem (Theorem 4.17) 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 2.17, 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.

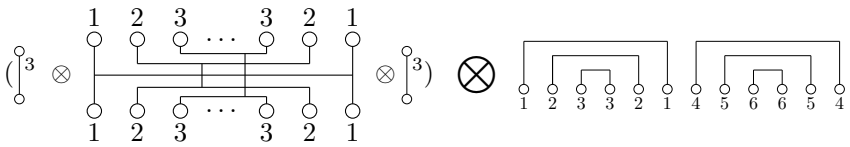
In order to get a valid rooted plane tree after the merging process, we need to apply the second rule. From Example 3.14, we already know that this involves also merging the open pairs, i.e. edge $5, 2$ and edge $1, 4$.



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 4.17.

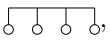
Remark 3.15. The intuition behind the rules of Definition 3.13 lies in the different partition operations, where we can simulate the rules by specific composition operations with partitions in Π_∞. In Example 3.14 (a), we basically obtain π_3 by the following operation.



3.2 Relevant Properties

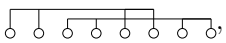
In this section, we go deeper into the properties of the category Π_∞, especially those that are relevant for Section 4. Most of the properties can also be found in [RW13]. First, we want to understand the origin of the category, namely its generator partitions (see Definition definition 3.13).

The partition π_1 coincides with

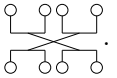


the rotated version of

while π_2 coincides with



the rotated version of



By understanding π_k, we can gain more information about the category.

Proposition 3.16. *Let $\Pi_m = \langle \pi_m \rangle$, 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 3.9, we know that π_k is given by k blocks on $4k$ points of the form $\pi_k = (p_1, p_2, ..., p_{k-1}, p_k, p_k, p_{k-1}, ..., p_2, p_1, p_1, p_2, ..., p_{k-1}, p_k, p_k, p_{k-1}, ..., p_2, p_1)$. Let $k > 2$, then we can reduce π_k to π_{k-1} by applying category operations with the base partitions ⊓ and ⋮. We simply compose π_k with the partition

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

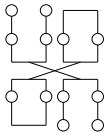
$$a_1, a_2, ..., a_{k-1}, a_{k-1}, ..., a_2, a_1, a_1, a_2, ..., a_{k-1}, a_{k-1}, ..., a_2, a_1) \leftarrow \text{lower points.}$$

Alternatively to get a better overview of this operation we can also say that we compose π_k with

$$(\text{points of } \pi_k; \leftarrow \text{upper points}$$

$$\text{points of } \pi_{k-1}) \leftarrow \text{lower points.}$$

With this operation 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 proposition 3.16 is the following series of categories:

$$\langle \pi_1 \rangle \subseteq \langle \pi_2 \rangle \subseteq \langle \pi_3 \rangle \subseteq \langle \pi_4 \rangle \subseteq \cdots \subseteq \Pi_\infty$$

Remark 3.17. Note that $\Pi_\infty = \Pi_{k \in \{1,...,\infty\}}$, where we use the notation $\Pi_{k \in \{1,...,\infty\}}$ in later parts of this thesis, when it comes to analyzing the parameter k in Π_k.

Definition 3.18. 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 3.19. $\Pi_{k \in \{1,...,\infty\}} (= \Pi_\infty)$ *is not finitely generated and* $\langle \pi_l, l < k \rangle \neq \Pi_k$ *for* $k \in \mathbb{N}$.

Proof. Theorem 3.9. of [SR16]. □

In preparation for the proof of Theorem 4.17, we again investigate the rules of Definition 3.13 further in Section 4.2.

4 The connection between Ordered Walks and the Category Π_∞

The main purpose of this section is to prove the equality of ordered walks on rooted plane trees and the category Π_∞. 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 lower points ($NC_2(0, n)$ for $n \in \mathbb{N}$).

4.1 Simple Walks, Dyck Paths and the Category NC_2

Proposition 4.1. *Let* W_s *be the set of all simple walks, then* $W_s \subseteq NC_2$.

Proof. 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.
Base case $n = 1$: There is no valid simple walk of length one, since each walk has to start and end at the root which is not possible in one step. Furthermore there is also no such partition in NC_2, as the category only comprises partitions of block size two.

As an **induction hypothesis** we say that for any simple walk $w_s = (e_1, \ldots, e_k)$ of arbitrary even length $k \leq n$, $w_s \in NC_2$.
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. Let w_s' be a simple walk of length $n+2$. Then we remove an arbitrary traversal of an edge directly leading to a leaf in the rooted plane tree of w_s' (a more detailed proof that such an edge always exists can be found in Lemma 4.19). As a result we get a simple walk w_s of length n. By applying our induction hypothesis, we get that $w_s \in NC_2$. Now in order to correctly continue the induction step, we need to construct w_s' again 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 as shown below.

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 the rooted plane tree of w_s. This can be achieved by rotating w_s exactly $n - j$ times, such that the new edge and node are added to the desired location:

$$(e_1, ..., e_j, \leftarrow \mathit{upper\_points}$$

$$e_n, ..., e_{j+1}) \leftarrow \mathit{lower\_points}$$

Let this new rotated partition be w_s^{rot}. The only thing left to do, before rotating w_s^{rot} back so that it only has lower points again, is the following tensor product with the base partition $\sqcap\!\!\circ\circ \in NC_2$:

$$w_s^{rot} \otimes \sqcap\!\!\circ\circ$$

The resulting partition is our w_s' of size $n+2$, 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. □

Proposition 4.2. *Let W_s be the set of all simple walks, then $NC_2 \subseteq W_s$.*

Proof. Follows similarly by induction as in the proof for Proposition 4.1. 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. □

Proposition 4.3. *Let W_s be the set of all simple walks, then $NC_2 = W_s$.*

Proof. From both Proposition 4.1 and Proposition 4.2, we know that

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

Consequently, we see that $NC_2 = W_s$. □

Theorem 4.4. *Let W_s be the set of all simple walks, then W_s is in bijection to the set of all Dyck paths.*

Proof. Dyck paths represent partitions in NC_2 (see [RW13]) combined with Proposition 4.3. □

Remark 4.5. Recall Definition 2.22 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 Π_∞

In order to draw a connection between the category Π_∞ and ordered walks, we first need to prove multiple properties about the rules of Definition 3.13. With a combination of Proposition 4.11 and Proposition 4.16, we can finally prove the bijection in Theorem 4.17, saying that the set of all ordered walks on rooted plane trees is equal to the set of partitions in the category Π_∞. Furthermore, we make use of Remark 4.5, which enables us to operate on the rooted plane tree layer of partitions in NC_2.

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 2.30, 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. Similarly, decreasing the Dyck level by taking a $(1, -1)$ step matches the backward traversal of an already visited edge in the tree resulting in a decrease of depth. □

Definition 4.7. Let p be a partition. Then we define a break in p by partitioning a block $b \in p$ with $|b| > 2$ into two different blocks b' and b'', where $|b'| \equiv_2 0$.

Definition 4.8. Let w be an ordered walk. We define $|w|_b \in \mathbb{N}$ as the number of breaks required to transform w into a simple walk i.e.

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

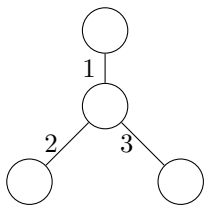
Consequently, the following property applies

$$|w|_b = 0 \Leftrightarrow w \textit{ is simple}$$

Example 4.9. Let w be of the form

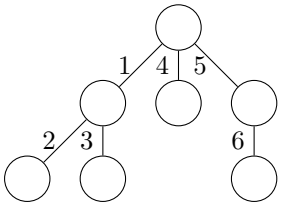
$$(1,\ 2,\ 2,\ 3,\ 3,\ 1,\ 1,\ 1,\ 1,\ 3,\ 3,\ 1)$$

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 $|w|_b = 3$ and $|w_s|_b = 0$.

Lemma 4.10. *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 the blocks b_{w_1} and b_{w_2} 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_{w_1} and b_{w_2}. 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. □

Proposition 4.11. *Let W be the set of all ordered walks, then $W \subseteq \Pi_\infty$.*

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

Base case: $n = |w|_b = 0$:
By definition, w is a simple walk. From Proposition 4.3 we know, that $(e_1, e_2, ..., e_m) \in NC_2$ and $NC_2 \subset \Pi_\infty$.

Induction hypothesis: We assume that $w \in \Pi_\infty$, if $|w|_b = n$.

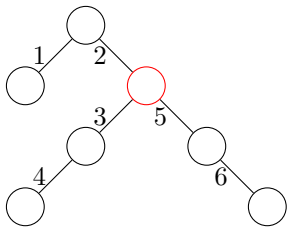
Induction step: Let $|w|_b = n+1$. In order to show $w \in \Pi_\infty$, we apply Lemma 4.10 and break the block b with greatest possible depth, having the property $|b| > 2$, into b_1 of size $|b|-2$ and b_2 of size 2. The resulting walk $w' = (e'_1, ..., e'_m)$ has the property $|w'|_b = n$. Using our induction hypothesis we can deduce that $(e'_1, ..., e'_m) \in \Pi_\infty$. By applying the first rule of Definition 3.13, we can reconstruct the walk w by reversing the break operation described above. The blocks can be merged according to this rule because they lie on the same Dyck level. Since the rules in Definition 3.13 define Π_∞, we know that the resulting points from w are $w = (e_1, e_2, ..., e_m) \in \Pi_\infty$. □

Proposition 4.12. *In an ordered walk, also being a partition in Π_∞ (according to Proposition 4.11), the open blocks 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 an ordered walk and a partition in Π_∞, and let b_i and b_j represent any two blocks in p, where $i < j$. According to Rule 2 of Definition 3.13, open blocks must satisfy the two conditions that they must include (an odd number of) point values that lie between b_i and b_j, and they must exclude point values that lie outside this range.

In the rooted plane tree of p, the open blocks 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 an odd number of times (in the ordered walk layer of p) in both positions, since they were traversed to reach b_i and b_j, but were not traversed back. □

Example 4.13. 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).

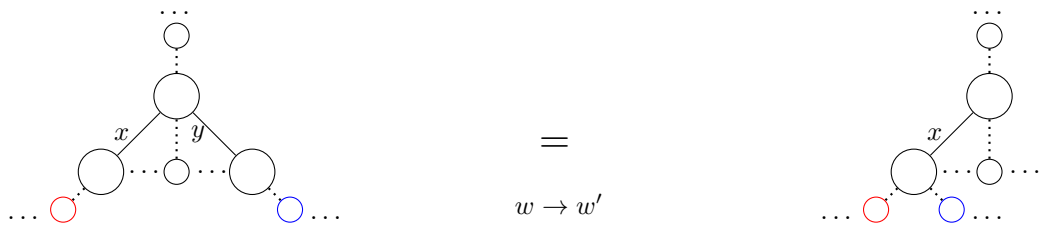
Lemma 4.14. *Let $w_1 = (e_1, \ldots, e_n)$ and $w_2 = (v_1, \ldots, v_m)$ of size $n, m \in \mathbb{N}$ be two independent ordered walks on disjoint labels. Then the concatenation $w_1 w_2 = (e_1, \ldots, e_n, v_1, \ldots, v_m)$ also forms an ordered walk.*

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 $(e_1, \ldots, e_n, v'_1, \ldots, v'_m)$, which obviously forms a valid ordered walk.. □

Lemma 4.15. *Let $w = (e_1, \ldots, e_n)$ be an ordered walk, then merging two arbitrary valid blocks in the partition $p = w$ according to the rules in Definition 3.13 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 3.13. Let $w = (e_1, ..., x, ..., y, ..., e_n)$ and let a_i and b_i be valid sub-walks and $x = (x_1, a_1, x_2, a_2, \ldots, x_m)$ and $y = (y_1, b_1, y_2, b_2, \ldots, y_m, b_m)$ 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 x_1 and y_1 have the same parent and x_i, y_i the same 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 $p' = w' = (e_1, ..., x, ..., x', ..., e_n) \in \Pi_\infty$, where the blocks from the $x_i's$ increased their sizes by 2. After the merging process, x remains identical and y changes to $x' = (x_1, b_1, x_2, b_2, \ldots, x_m, b_m)$. 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 4.12). 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 Lemma 4.14) 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 4.14.

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. □

Proposition 4.16. *Let W be the set of all ordered walks, then $\Pi_\infty \subseteq W$.*

Proof. Let $p \in \Pi_\infty$ of the form $(p_1, p_2, ..., p_m)$, where $m \in \mathbb{N}$ is the size of p. Furthermore, the function $|\cdot|_m : \Pi_\infty \to \mathbb{N}$ returns the number of pairs merged according to the rules of Definition 3.13.

Base case: $n = |p|_m = 0$:
By Definition, $p \in NC_2$ and consequently according to Proposition 4.3, we know, that $p = (p_1, p_2, ..., p_n)$ is a simple walk.

Induction hypothesis: We assume that p is an ordered walk, if $|p|_m = n$

Induction step: Let first be $|p|_m = n + 1$. According to Definition 3.13 there is a partition $q \in NC_2$ and a sequence of merges of length m, which turns q into p. By reversing the last merge of the sequence (i.e. performing a break), we get an intermediate partition p' with $|p'|_m = n$. Consequently, p' is also an ordered walk according to the induction hypothesis.

Finally, we perform the merge, which we first reversed above, and turn p' back into p. By Lemma 4.15, we can deduce that p also has to be a valid ordered walk. □

Theorem 4.17. *Let W be the set of all ordered walks, then $\Pi_\infty = W$.*

Proof. From $W \subseteq \Pi_\infty$ (Proposition 4.11) and $\Pi_\infty \subseteq W$ (Proposition 4.16) follows that $\Pi_\infty = W$. □

4.3 The *"Holy"* Quaternity - A fourth Definition of Π_∞ arose

In this section, we examine and revisit the different representations of elements in Π_∞. 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 Π_∞ 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 3.13)

(b) by constructing Π_∞ from non-crossing pair partitions (Definition 3.13)

(c) by ordered walks on rooted plane trees (Theorem 4.17)

(d) by non-interference ([Man22])/even-nested-ness (in this section)

While Π_∞ can be defined by its generator set $\langle \pi_k, k \in \mathbb{N} \rangle$, Raum and Weber introduced a new perspective by Definition 3.13, which states that we can build any element in Π_∞ 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 4.17). With this knowledge, we introduced the following different view of Π_∞.

Definition 4.18 (Even-Nested)**.** A partition $p = (p_1, \dots, p_n)$ is called *even-nested* if, for every block B with point values b, the positions of all points in B can be grouped into consecutive distinct pairs $(p_{i_1}, p_{j_1}), (p_{i_2}, p_{j_2}), \dots$, such that for each pair $p_{i_k} = p_{j_k} = b$, where the sequence $p_{sub} = (p_{i_k+1}, \dots, p_{j_k-1})$ does not contain b and each point value in p_{sub} occurs an even number of times.

Lemma 4.19. *Let $p = (p_1, \dots, p_n)$ be an even-nested partition of size $2 \leq n \in \mathbb{N}$. Then there exists at least one pair of neighboring points (p_i, p_{i+1}) in p, where (p_i, p_{i+1}) belongs to the grouping described in Definition 4.18.*

Proof. We prove this Lemma by induction over the size $|p|$ of the even-nested partition p.
Base case: For $|p| = 2$ we get the ordered walk $p = (1, 1)$ which consists of two directly neighboring points from the same block.
Induction Hypothesis (I.H.): We assume that (p_i, p_{i+1}) in p with $p_i = p_{i+1}$ exists if $|p| < n$.
Induction Step: Let $|p| = n$ and let A be an arbitrary block in p. Then, by definition, the points in A can be grouped into consecutive distinct pairs $(p_{i_1}, p_{j_1}), (p_{i_2}, p_{j_2}), \dots$. Since all points between (p_{i_1}, p_{j_1}) occur an even number of times and also have to maintain the even-nested property, we can conclude that the partition $p' = (p_{(i_1)+1}, \dots, p_{(j_1)-1})$ itself is even-nested. Consequently, we can apply the I.H. since $|p'| < n$, which shows that there is at least one direct neighboring pair (p_i, p_{i+1}) in p' with $p_i = p_{i+1}$. Because p' is a part of p, we can conclude that also p has at least one neighboring pair (p_i, p_{i+1}) in p' with $p_i = p_{i+1}$. □

We can show the relation between this even-nested definition to Π_∞ via ordered walks.

Proposition 4.20. *$p \in \Pi_\infty$ 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 consecutive distinct pairs (p_{i_k}, p_{j_k}) (as in the definition of even-nested partitions) appear an even number of times. Thus, by Definition 4.18, every ordered walk satisfies the even-nested property.

$\Leftarrow$: 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: Assume from I.H. that any even-nested partition of length n corresponds to an ordered walk. Let p be an even-nested partition of length $n+2$.

We now remove an arbitrary pair of neighboring points (p_i, p_{i+1}) from the grouping described in Definition 4.18, where $p_i = p_{i+1}$. Note that such a pair always exists in a non-empty even-nested partition according to Lemma 4.19. The resulting partition p' is still even-nested, as the removal of a consecutive pair (p_i, p_{i+1}) does not affect the even-nestedness of the remaining blocks. More precisely:

- Every block can still be grouped into distinct consecutive pairs as defined in the definition of even-nested partitions.
- All other blocks in p maintain their previous pairings/groupings, and the parity of point values between their consecutive pairs remains unchanged because we removed a directly neighboring pair so that nonetheless every pair of the block grouping wraps every different point value an even number of times.

Because $|p'| = n$, we know from I.H. that p' is an ordered walk. Reinserting the removed pair in p is equivalent to traversing one more edge forth and back in the ordered walk, where we can conclude that p 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 $p \in \Pi_\infty$. □

However, this finding turned out to be already formulated by Mang in [Man22], where he defined Π_∞ using the notion of non-interference.

Definition 4.21 (Non-Interference)**.** Let p be a partition and $A, B \in p$ be blocks in p. We say that A and B are *non-interferent* if and only if every interval formed by every pair of points in A does include an even number of points in B. More precisely,

$$\forall \alpha, \alpha' \in A: \quad |[\alpha, \alpha'] \cap B| \equiv_2 0,$$

where $|[\alpha, \alpha'] \cap B|$ is the number of points of B that are between the points α and α' in the partition sequence p.

With the definition of non-interference, Mang defined Π_∞ as follows.

$$\Pi_\infty := \{p \,|\, \forall A \in p : |A| \equiv_2 0 \quad \wedge \quad \forall A, B \in p : A \neq B \Rightarrow A, B \text{ non-interferent}\}$$

Example 4.22 (Even-nested and non-interferent partitions)**.** Consider the partition

$$p = (p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}) = (1, 2, 2, 1, 3, 4, 4, 3, 1, 2, 2, 1) \in \Pi_\infty.$$

This partition is both even-nested and every distinct pair of blocks in p is non-interferent. We could show this by checking for every pair of points (α, α') in every block whether there is an even number of other point values between $[\alpha, \alpha']$. We only check the conditions for $A = (p_1, p_4, p_9, p_{12})$ with value 1, since the other calculations are similar.

- $[p_1, p_4]$ encloses the point value 2 which occurs an even number of times (exactly 2) ✓ ✓
- $[p_1, p_9]$ encloses the point values $1, 2, 3, 4$ which all occur two times (except the 1, which is not taken into account) ✓
- $[p_1, p_{12}]$ encloses the point values $1, 2, 3, 4$ which all occurs an even number of times ✓
- $[p_4, p_9]$ encloses the point values $3, 4$ which both occur two times ✓
- $[p_4, p_{12}]$ encloses the relevant point values $2, 3, 4$ which all occur two times ✓
- $[p_9, p_{12}]$ encloses the point value 2 which occurs two times ✓ ✓

As we can see, we need to check less pairs for the even-nested condition than for non-interference. This can be an advantage when implementing an algorithm for (for example) checking whether a given partition p is in Π_∞.

Proposition 4.23. *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 4.18, every block A in p can be grouped into consecutive neighboring pairs and the sequences within those pairs contain only point values that occur an even number of times in those sequences and do not include the block label itself.

- Since we can group every block A in distinct pairs, we can conclude that every block A itself has to be of even size.
- Instead of allowing for every pair of points $\alpha, \alpha' \in A$ only an even number of points for every other block between $[\alpha, \alpha']$, it is sufficient to check this property for every distinct consecutive neighboring pair (β, β') (as in Definition 4.18), since the global evenness condition over any interval $[\alpha, \alpha']$ can be obtained by summing the local parity conditions between each such consecutive pair within that interval. The additivity of the evenness parity ensures that if the condition holds locally between all neighboring pairs, then it must also hold for any larger enclosing pair. As a result, this shows that every distinct pair of blocks in an even-nested partition is non-interferent.

$\Leftarrow$: This direction is trivial, since if any two distinct blocks A, B in a partition p are non-interferent, then the following applies by definition.

$$\forall \alpha, \alpha' \in A: \quad |[\alpha, \alpha'] \cap B| \equiv_2 0.$$

As a result every occurrences between **all** α and α' have to be of even size, while even-nestedness is already satisfied if the points between **any distinct neighboring** α and α' occur an even number of times. □

In this case, we see that having alternative definitions, such as our ordered walks, can help uncover additional properties and perspectives. Although the even-nested property itself is not a novel result, we were able to establish its connection to Π_∞ via ordered walks and thereby further reinforcing the definition of Mang. In Section 4.4, we revisit the definitions from this section.

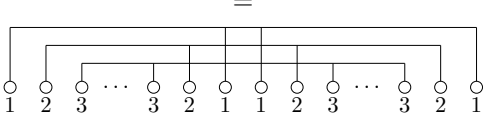
4.4 Relationship Between Ordered Walks and the Parameter k in $\Pi_{k\in\{1,\dots,\infty\}}$

We defined Π_∞ in many different ways. Now that we have established a connection to ordered walks, we are interested in the significance of the parameter k in $\Pi_{k\in\{1,\dots,\infty\}}$. Since k occurs in the context of the generator partition definition, we will also focus on this perspective in this section. Furthermore, as we perform operations on partitions in this section (see Section 2.2.1), we once again distinguish between upper and lower points when necessary.

From Definition 3.9, we know that the partition $\pi_{k\in\mathbb{N}}$ is given by k four blocks on $4k$ points of the form

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

$$=$$



Recall that the partitions in NC_2 are basically simple walks and those in $\Pi_{k\in\{1,\dots,\infty\}}$ are ordered walks. In order to classify k in $\Pi_{k\in\{1,\dots,\infty\}}$, we need to understand what changes by assigning k with different values. For $k = 1$ we have $\Pi_1 = \langle$  $\rangle$ which coincides with the category $NC_{even\ BS}$ (see appendix). This category differs from NC_2 in that the block sizes can grow arbitrarily large while remaining even. As the name of this category suggests, there are still no crossings between the blocks. Thus, it can be interpreted as "non-crossing ordered walks", where paths can be traversed arbitrarily many times without introducing crossings. Combining those finding with Proposition 3.16, we see that

$$NC_{even\ BS} \subseteq \Pi_{k\in\{1,\dots,\infty\}}.$$

Using the category-generating algorithm from `OSCAR`, as described in [Vol23], we observe that

$$\pi_2 = \overline{\circ\,\overline{\circ\,\overline{\circ\,\overline{\circ\,\overline{\circ\,\overline{\circ\,\overline{\circ\,\overline{\circ\,\circ}}}}}}}} \notin NC_{even\ BS},$$

and therefore

$$NC_{even\ BS} \subset \Pi_{k\in\{1,\dots,\infty\}}$$

which is entirely reasonable since π_2 contains crossings and the ordering from Proposition 3.16 is actually strict (see [SR16]). Intuitively, this suggests that as k increases, the number of possible crossings and nestings also grow.

Now, we aim to analyze the significance of k in the context of the rules in Definition 3.13 and to understand how the rules change when considering a fixed, finite $k \in \mathbb{N}$.
Given the assumption that the parameter k in $\Pi_{k\in\{1,\dots,\infty\}}$ is related to the crossings and nestings of its elements, we want to use a measure that quantifies the degree of crossing complexity. Both Raum and Weber as well as Mang investigated the implications of the parameter k in $\Pi_{k\in\{1,\dots,\infty\}}$. Raum and Weber used a so called *w-depth*, where Mang defined a *betweeness* relation.

We first start to introduce the betweeness relation by Mang.

Definition 4.24 (Betweenness and the Category $\Pi_{k\in\{1,\dots,\infty\}}$)**.** Let p be a partition. We define the *betweenness relation* χ_p as

$$\chi_p = \left\{ (A,B,C) \in p^{\times 3} \;\middle|\; \neg(A=B=C) \ \wedge\ \exists\,(\alpha,\gamma) \in A \times C : |\,[\alpha,\gamma]_p \cap B| \equiv_2 1 \right\}.$$

Using this relation, we define the category Π_k as the set of all partitions $p \in \Pi_\infty$ satisfying the following condition:

$$\forall A, C \in p \text{ with } A \text{ crossing } C, \quad |\{B \in p \mid (A,B,C) \in \chi_p\}| \leq k.$$

As we can see, Mang was able to measure nestings in a partition $p \in \Pi_k$ for a fixed $k \in \mathbb{N}$. However, for the level of crossings in p, we need another point of view. For this purpose, we could use the so-called *w-depth*, defined in [RW13].

Definition 4.25 (W-depth)**.** Let p be a partition. Then the *w-depth* of p, denoted by w-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-depth(p) thus measures the maximal depth of such a W-structure present in p.

Example 4.26. Since the definition of w-depth(p) is admittedly confusing, we will discuss some examples and edge cases.

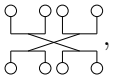
(a) π_k has a w-depth of k. This results from the definition of π_k

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

which basically coincides with the definition of w-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-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-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 $p = (1^2, 2^2, 3^2, 3^2, 2^2, 1^2, 1^2, 2^2, 3^2, 3^2, 2^2, 1^2)$. More precisely p basically looks similar to π_3 with the difference that we duplicated each point. Intuitively, this partition should has 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

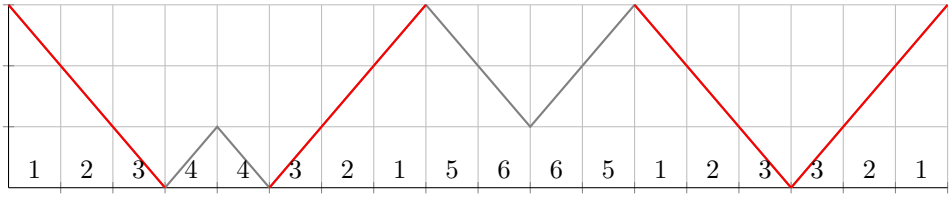


which can be used to swap arbitrary neighboring pair points of a partition by applying composition. So to retrieve p, we can simply start with $(1^2, 1^2, 1^2, 1^2, 2^2, 2^2, 2^2, 2^2, 3^2, 3^2, 3^2, 3^2) \in NC_2$ and swap with the rotated version of π_2 until we get p.

(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-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 4.27 (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 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 and both w_a, w_b are non-empty.

If one of the sub-walks is empty, we say that the other crosses itself in w.

Example 4.28. (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 $w_a = (1, 2, 3, 3, 2, 1)$ crosses itself, since $w = w_a(4, 4)w_a$.

Theorem 4.29. *Π_k is the category of partitions of all even-nested partitions with a w-depth of at most k.*

Proof. Follows from Proposition 4.20 and Section 4 and 5 from [RW13]. □

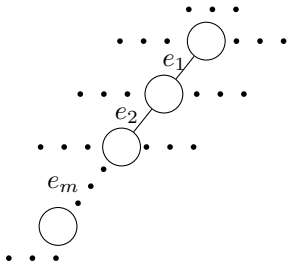
Knowing the parameter k in $p \in \Pi_{k\in\{1,\dots,\infty\}}$ actually determines the degree of crossings and nestings, as measured by the w-depth, it is necessary to extend the concept of w-depth to our ordered walks. The basic intuition is that a w-depth of k in an ordered walk simply means visiting at most k consecutive edges $e_1, e_2, \dots, e_k$ at least four times in a crossing manner.

Proposition 4.30. *Let $p \in \Pi_{k\in\{1,\dots,\infty\}}$ be a partition with a W-structure of (at least) 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. The walk 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 $e_1, \ldots, e_m$ are traversed forwards before any edge e_i is traversed backwards. □

Finally, at this point, we are able to present an interpretation for the parameter k in $\Pi_{k\in\{1,\ldots,\infty\}}$ in terms of the ordered walk representation on rooted plane trees.

Theorem 4.31. *Let $w \in \Pi_k$ for a fixed $k \in \mathbb{N}$. Then for every pair of sub-walks w_1 and w_2 in w that cross each other, the sum of their depths satisfies*

$$\operatorname{depth}(w_1) + \operatorname{depth}(w_2) \leq k.$$

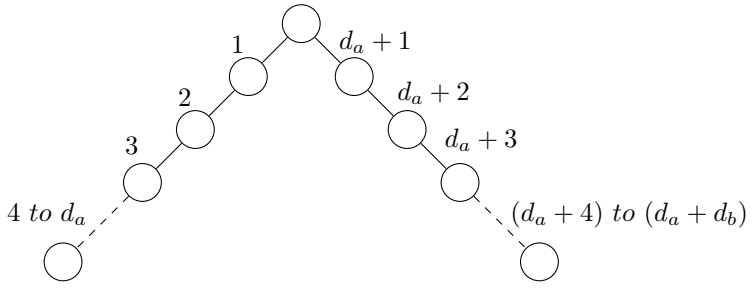
Proof. We prove this statement by contradiction. Let $w \in \Pi_k$ with $k \in \mathbb{N}$ be an ordered walk in which the sub-walks w_a and w_b cross each other. We assume additionally that $d_a + d_b > k$, where $d_a = \operatorname{depth}(w_a)$ and $d_b = \operatorname{depth}(w_b)$. As a result, the walk w has the following form

$$w = w_1 w_a w_2 w_b w_3 w_a w_4 w_b w_5.$$

Now we remove the walks w_i with $i \in \{1, 2, 3, 4, 5\}$ where the resulting walk w' is still an ordered walk according to Definition 2.24 (alternatively we could also remove them by partition operations). The walk

$$w' = (\underbrace{1, \ldots, d_a, d_a, \ldots, 1}_{\text{sub-walk } w_a}, \underbrace{d_a+1, \ldots, d_a+d_b, d_a+d_b, \ldots, d_a+1}_{\text{sub-walk } w_b},$$
$$\underbrace{1, \ldots, d_a, d_a, \ldots, 1}_{\text{sub-walk } w_a}, \underbrace{d_a+1, \ldots, d_a+d_b, d_a+d_b, \ldots, d_a+1}_{\text{sub-walk } w_b})$$

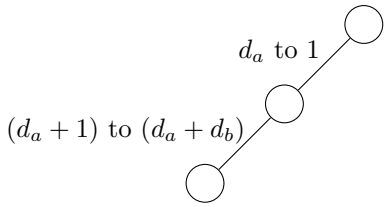
traverses the tree



We can now rotate w' to w'', where w'' is depicted below, without normalizations due to readability.

$$w'' = (d_a, \ldots, 1, d_a+1, \ldots d_a+d_b, d_a+d_b, \ldots, d_a+1, 1, \ldots, d_a,$$
$$d_a, \ldots, 1, d_a+1, \ldots d_a+d_b, d_a+d_b, \ldots, d_a+1, 1, \ldots, d_a)$$

Finally, the tree of w'' has the following form.

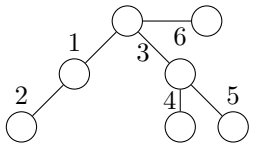


Consequently, w'' has a W-structure of depth $d_a + d_b$ and the partition depicted by w'' is generated by $\langle\ \Pi_l \mid l \geq d_a + d_b > k\rangle$. Since we are able to transform w to w'' by simply removing sub-walks and applying the category operation of rotating in the partition layer, we can deduce that $w \notin \Pi_k$. However, we assumed that $w \in \Pi_k$, which leads us to the contradiction and proves our statement.

This can be explained intuitively: 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 with (according to Definition 4.25) a w-depth and a W-structure of $d_a + d_b$ (see Proposition 4.30).

□

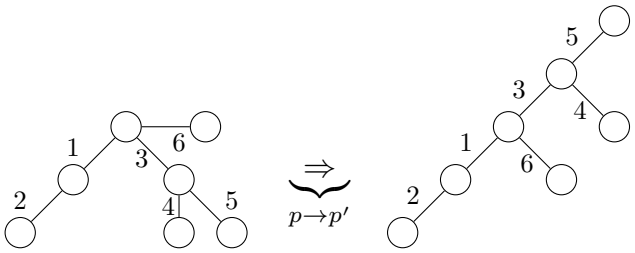
Example 4.32. Note that we will not normalize the walks/partitions in this example just to make it less confusing. Let $p = (1, 2, 2, 1, 3, 4, 4, 5, 5, 3, 6, 6, 1, 2, 2, 1, 3, 4, 4, 5, 5, 3, 6, 6)$. As a result, p is in Π_4 since the rotated version $p' = (5, 3, 6, 6, 1, 2, 2, 1, 3, 4, 4, 5, 5, 3, 6, 6, 1, 2, 2, 1, 3, 4, 4, 5)$ 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 4.31, 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 4.31.



5 Algorithmic Methods for Analyzing Properties of Π_∞

An other approach for investigating the properties of categories is to utilize algorithms and data structures and apply computer algebra research systems. As described and analyzed in [Vol23], presented at `ISSAC` conference 2025 ([FV25]) 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 Π_∞. More precisely, using Theorem 4.17, we solve the problem of deciding whether $p \in \Pi_{k \in \{1,\ldots,\infty\}}$ for a partition p.

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 Π_∞ 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 $|\Pi_\infty \cap P(n)|$ 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 $|\Pi_\infty \cap P(12)|$ 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 Π_∞ up to a size of 12, yielding the following values.

partition size n	0	2	4	6	8	10	12
$\lvert\Pi_\infty \cap P(n)\rvert$	1	3	15	84	513	3333	22672
$\lvert\Pi_\infty \cap P(n,0)\rvert$	1	1	3	12	57	303	1744

Note that for odd partition sizes we get zero. Using [OEI] we can see that the sequence `A094149` fits to the sequence of $|\Pi_\infty \cap P(0,n)|$, where we only filter out partitions with lower 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 4.17.

Since we have already proven the connection between Π_∞ 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 $|\Pi_\infty \cap P(n,0)|$.

$$|\Pi_\infty \cap P(0,n)| = \sum_{r=0}^{n} W_n(r)$$

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 $|\Pi_\infty \cap P(n,0)|$, 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 $|\Pi_\infty \cap P(n,0)|$ 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 ← array of size (n + 1) × (n + 1) filled with 0;
2 for i ∈ {0, 1, ..., n} do
3 |   binom[i][0] ← 1 ▷Base case:  (i 0) = 1
4 |   binom[i][i] ← 1 ▷Base case:  (i i) = 1
5 |   for k ∈ {1, 2, ..., i − 1} do
6 |   |   binom[i][k] ← binom[i − 1][k − 1] + binom[i − 1][k] ▷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.

Proposition 5.1. *Calculating the number of ordered walks (or partitions in Π_∞) of length n has an upper bound time complexity of $O(n^5)$.*

Proof. 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 $O(n^2 + n^5 + n) = O(n^5)$. □

Note that we assume basic operations like addition and multiplication to run in $O(1)$ time. In practice, however, this does not always hold when operands become large, as their bit-length grows and arithmetic operations require logarithmic time.

5.2 Constructing and Validating Elements in Π_∞

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 p is in a specific category $\mathcal{C}$. However, in general, the decision problem of determining membership $p \in \mathcal{C}$ is undecidable. Theorem 4.9 in [FV25] states the following.

Theorem 5.2 (Theorem 4.9 in [FV25])**.** *There exists a recursively enumerable category of partitions $\mathcal{C}$ such that the problem of determining if $p \in \mathcal{C}$ for a partition p is undecidable.*

However, in this section, for the category Π_∞, we present an efficient algorithm that decides whether a given partition belongs to Π_∞.

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`. All of these functions run in linear time, making them significantly more efficient than the naive approach from above.

Example 5.3. 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 ← construct_category([producer of B_2], size(p));
3 return p ∈ B_2;
```

While this approach is sufficient for a p of $size(p) \leq 12$, it is generally considered as slow. A more efficient approach is the following.

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 is_pair_partition ← is_pair(p);
2 is_balanced ← is_balanced(p);
3 return is_pair_partition ∧ is_balanced;
```

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

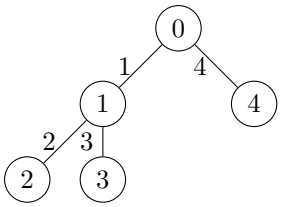
Currently, the only approach available to handle the input category Π_∞ is the naive approach. This is because Π_∞ cannot be described solely using properties such as crossing, balanced, pair, singleton, etc. Fortunately, we were able to prove the connection between Π_∞ and ordered walks in Section 4.2. Consequently, given a partition p and the category Π_∞, 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 Π_∞.

```
   Input: input partition p
   Output: True if p corresponds to a valid structure, else False
 1 pi_k ← p; max_point ← 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 ← 0;
 5 for curr_edge in pi_k do
 6     if not back_edge[curr_edge] and not visited[curr_edge] then
           /* add new edge to tree structure in array edge               */
 7         edge[curr_edge] ← (curr_node, curr_edge);
 8         curr_node ← curr_edge;
           /* expect next occurrence of this edge to be a back edge      */
 9         back_edge[curr_edge] ← True;
10         visited[curr_edge] ← True;
11         continue;
12     if visited[curr_edge] and back_edge[curr_edge] then
13         if curr_node ≠ edge[curr_edge][1] then
               /* violation in tree structure detected (cycle)           */
14             return False;
15         curr_node ← edge[curr_edge][0];
16         back_edge[curr_edge] ← False;
17         continue;
18     if visited[curr_edge] and not back_edge[curr_edge] then
19         if curr_node ≠ edge[curr_edge][0] then
               /* violation in tree structure detected (cycle)           */
20             return False;
21         curr_node ← edge[curr_edge][1];
22         back_edge[curr_edge] ← True;
23         continue;
   /* check whether final node is the root                               */
24 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. The algorithm iterates through the points of the input partition p and builds the tree structure in the array `edge`, which is traversed by the ordered walk p. If the algorithm iterates over an already visited point value, it checks whether the current node in the traversal matches the expected node stored in the array `edge`, if not, this indicates a violation of the tree structure, and the algorithm returns `False`. Finally, the algorithm returns `True`, if the walk p terminates at the starting point, i.e. the root.
In the following example, we demonstrate a run of Algorithm 4, where the input partition is actually in Π_∞. In this case, the algorithm successfully builds the rooted plane tree while traversing it in the form of an ordered walk.

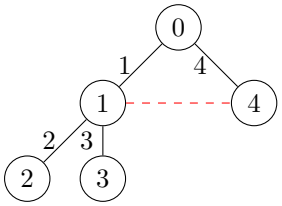
Example 5.4. Consider the partition $(1,\ 2,\ 2,\ 3,\ 3,\ 1,\ 4,\ 4) \in \Pi_\infty$. Algorithm 4 iterates over each point value. For any new point value encountered, the algorithm stores a pair consisting of the current edge and the new point value. Note that the root node is assigned with 0 since it 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 Π_∞, the algorithm will find a violation in the tree structure.

Example 5.5 (Input partition not in Π_∞)**.** Consider the partition (1, 2, 2, 3, 3, 1, 4, 4, 1, 4, 4, 1) $\notin \Pi_\infty$. 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`.



Theorem 5.6. *Given a partition p of size n, the problem of deciding whether $p \in \Pi_\infty$ can be solved in time $O(n)$, which coincides with the lower bound $\Omega(n)$ of this problem. Thus, the algorithm is asymptotically optimal.*

Proof. Algorithm 4 solves this problem with linear time complexity. 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 lower bound time complexity for this problem is $\Omega(n)$, because we need to check each point in p in order to decide whether p is an ordered walk. □

After presenting an algorithm that decides whether a partition belongs to Π_∞, we now aim to design an algorithm for a more specific problem. Let p be a partition and $k \in \mathbb{N}$. We want an algorithm that checks whether $p \in \Pi_k$. In Section 4.4, we defined multiple ways to characterize the elements in Π_k for a fixed k. One of them is the w-depth, where we look at the Dyck path layer of the partition. The other two characteristics are the betweenness relation of A. Mang and the depth of the crossings in the ordered walk layer. However, we consider the w-depth for the specialized membership problem.

Definition 4.25 stated that we can compute the w-depth by rotating the partition and searching for the deepest W-structure.

Lemma 5.7. *Let $p = (p_1, \ldots, p_n) \in \Pi_\infty$ be a partition with w-depth k. Then there exists a rotated version p' of p such that the W-structure of depth k in p' begins at the first point, i.e. $p'_1 = a_1$, where a_i denotes the W-structure as defined in Definition 4.25.*

Proof. As p is in Π_∞, it is also an ordered walk. In the proof of Theorem 4.31, we showed that rotating p allows us to change the root of its rooted plane tree. Consequently, we may rotate p so that the root lies at the first edge of the deepest W-structure. □

Using Lemma 5.7, we can construct an algorithm with a time complexity of $O(n^2)$.

Algorithm 5: Compute w-depth of a partition in Π_∞.

```
   Input: Partition p ∈ Π_k
   Output: Th w-depth p
 1 out ← 0;
   /* Iterate over all rotated versions of the partition                    */
 2 for i ← 0 to size(p) do
 3     rotate points p_1, ..., p_i to right side of partition;
 4     max_point ← max(p);
 5     initialize array back_edge of size max_point + 1 with False;
 6     deepest_in_wdepth ← −1;
 7     level ← 0;   max_level ← 0;   second_round ← False;
 8     for point in p do
 9         if not second_round then
10             if point == 1 and back_edge[1] then
11                 back_edge[1] ← False;
12                 second_round ← True;
13                 continue;
14             if not back_edge[point] then
15                 level ← level + 1;
16                 if max_level ≤ level then
17                     max_level ← level;
18                     deepest_in_wdepth ← point;
19             else
20                 level ← level − 1;
21         else
22             if point == deepest_in_wdepth then
23                 out ← max(out, max_level);
24         back_edge[point] ← ¬back_edge[point];
25 return out;
```

Algorithm 5 outputs the w-depth of a partition $p \in \Pi_\infty$. As written in Definition 4.25, the algorithm iterates over every rotated version p' of the input partition p. Note that according to Theorem 4.17, p as well as p' are also walks on rooted plane trees. For every p' it calculates the maximal Dyck level (or depth) of the first sub-walk w_{sub} that starts at the root and ends at the root, i.e. the first and the last move of the sub-walk has the value of p'_1. Additionally, the algorithm stores the deepest point p_d encountered during the walk w_{sub}. After completing w_{sub}, the algorithm continues iterating through p', and if the value p_d occurs again in $p' \setminus w_{\text{sub}}$, it updates the output variable with the depth of p_d, if this value is greater than the current output, which may have been set by a previous rotation of p'. According to Lemma 5.7, there is a w_{sub} which is part of the W-structure that sets the w-depth of p.

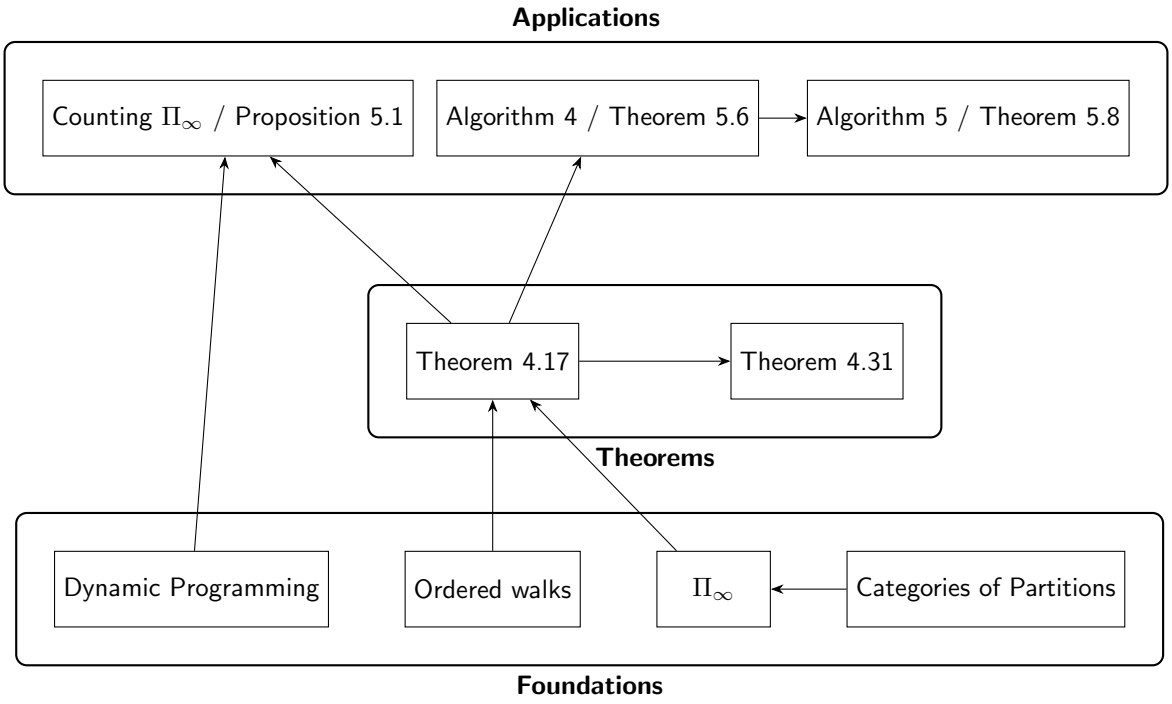
Theorem 5.8. *Let $p \in \Pi_\infty$ be a partition of size $n \in \mathbb{N}$. Then the problem of computing the smallest k such that $p \in \Pi_k$ can be solved in $O(n^2)$.*

Proof. Algorithm 5 solves the problem from above. In worst case, the algorithm iterates for every rotated version of p through each point. Since there are n different rotated versions of p and for each version we iterate over all points, we get an overall time complexity of $O(n^2)$. □

The implementation of this algorithm can be found in [Vol25].

6 Discussion

In this section, we conclude the thesis by summarizing its key contributions and providing a brief outlook on potential directions for future work based on the presented results. The following graph shows a general overview of the main topics of this thesis.



Our main achievement is the discovery and proof of the connection between the category Π_∞ and ordered walks on rooted plane trees, which is formulated by Theorem 4.17. Based on this result, we derived a polynomial-time algorithm for counting the number of partitions in Π_∞ of size n. Before that, such calculations were only possible for values of n up to ~ 12, due to the exponential growth of the category. Additionally, given a partition p, we can now solve the membership problem of whether $p \in \Pi_\infty$ in linear time. If $p \in \Pi_\infty$, we can also compute the smallest k such that $p \in \Pi_k$ in quadratic time. Moreover, we were able to further validate Mang's Definition 4.21, for which we also provided our own interpretation, called even-nestedness, and proved it via ordered walks.

Future Work

Definition 3.13 by Raum and Weber defines the category Π_∞ via the notation of open blocks. The limitation of this definition is that it does not take the parameter k in $\Pi_{k\in\{1,\dots,\infty\}}$ into account.

Question 6.1. In what way must the rules in Definition 3.13 be modified to characterize the category Π_k for a fixed $k \in \mathbb{N}$?

This problem could be solved by looking at the interpretation of ordered walks generated by Π_k for a fixed $k \in \mathbb{N}$ discussed in Theorem 4.31. In this interpretation, open pairs correspond to paths that are traversed forward but not backward, and Dyck levels represent the depth within the rooted plane tree.
Continuing with the parameter k in $\Pi_{k\in\{1,\dots,\infty\}}$, the formula from Khorunzhy and Vengerovsky counts the number of ordered walks (see Section 2.2.3), i.e. the category Π_∞. Consequently, it would be interesting to get a closed formula for counting the category Π_k for a fixed $k \in \mathbb{N}$.

Question 6.2. Is it possible to find a closed formula for counting all ordered walks generated by Π_k for a fixed k?

Such a formula could be retrieved by further investigating the formula of Khorunzhy and Vengerovsky and adapting it with the help of Theorem 4.31.

Question 6.3. Currently, we still need to construct Π_∞ via the generator partitions. Could this be done more efficiently by enumerating Π_∞ through the even-nested property?

It may be possible to construct Π_∞ via a grammar that relies on the even-nested property. This approach could be more efficient, as it naturally avoids partitions outside Π_∞ and does not rely on partition operations, unlike the generator enumeration method.

References

[Bar16] G. 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: 2025-03-14.

[BBC07] T. Banica, J. Bichon, and B. 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] Z. Füredi and J. Komlós. The eigenvalues of random symmetric matrices. *Combinatorica*, 1(3):233–241, 1981.

[Fou25] Python Software Foundation. functools.lru_cache, 2025. Accessed: 2025-02-14.

[FV25] N. Faroß and S. Volz. Algorithmic problems in categories of partitions, 2025. arXiv:2502.05373 [cs.DS].

[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. arXiv:math-ph/0009028.

[Man22] A. 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.

[Ros14] S. Ross. *Introduction to Probability Models*. Academic Press, Boston, 11th edition, 2014.

[RW13] S. Raum and M. Weber. The full classification of orthogonal easy quantum groups, 2013. arXiv:1312.3857 [math.QA].

[RW14] S. Raum and M. Weber. The combinatorics of an algebraic class of easy quantum groups. *Infinite Dimensional Analysis, Quantum Probability and Related Topics*, 17(03):1450016, 2014.

[RW15] Sven Raum and Moritz Weber. Easy quantum groups and quantum subgroups of a semi-direct product quantum group. *Journal of Noncommutative Geometry*, 9(4):1261–1293, 2015.

[SR16] M. Weber S. Raum. The full classification of orthogonal easy quantum groups. *Communications in Mathematical Physics*, 341:751–779, 2016.

[Sta99] R. Stanley. *Enumerative Combinatorics*, volume 2. Cambridge University Press, 1999.

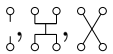
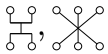
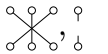
[Vol23] S. Volz. Bachelor's thesis - design and implementation of efficient algorithms for operations on partitions of sets. `https://www.uni-saarland.de/fileadmin/upload/lehrstuhl/weber-moritz/Abschlussarbeiten/volz-bachelors-thesis.pdf`, 2023.

[Vol25] S. 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] E. 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
$P^{1,2}_{even}$	B'_n		even part of partitions with blocksizes 1 or 2
$P_{even\ BS}$	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
$NC_{even\ BS}$	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
$NC^{1,2}_{even}$	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
$B^{1,2}_{even}$	$B^{\#}_n*$		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 $\leq k$