

# Anexos

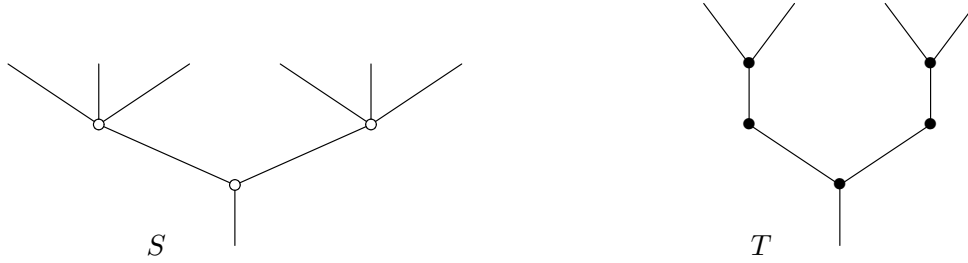
En este anexo se encuentra el ejemplo completo de un conjunto de shuffles y el código Python del paquete.

## Contents

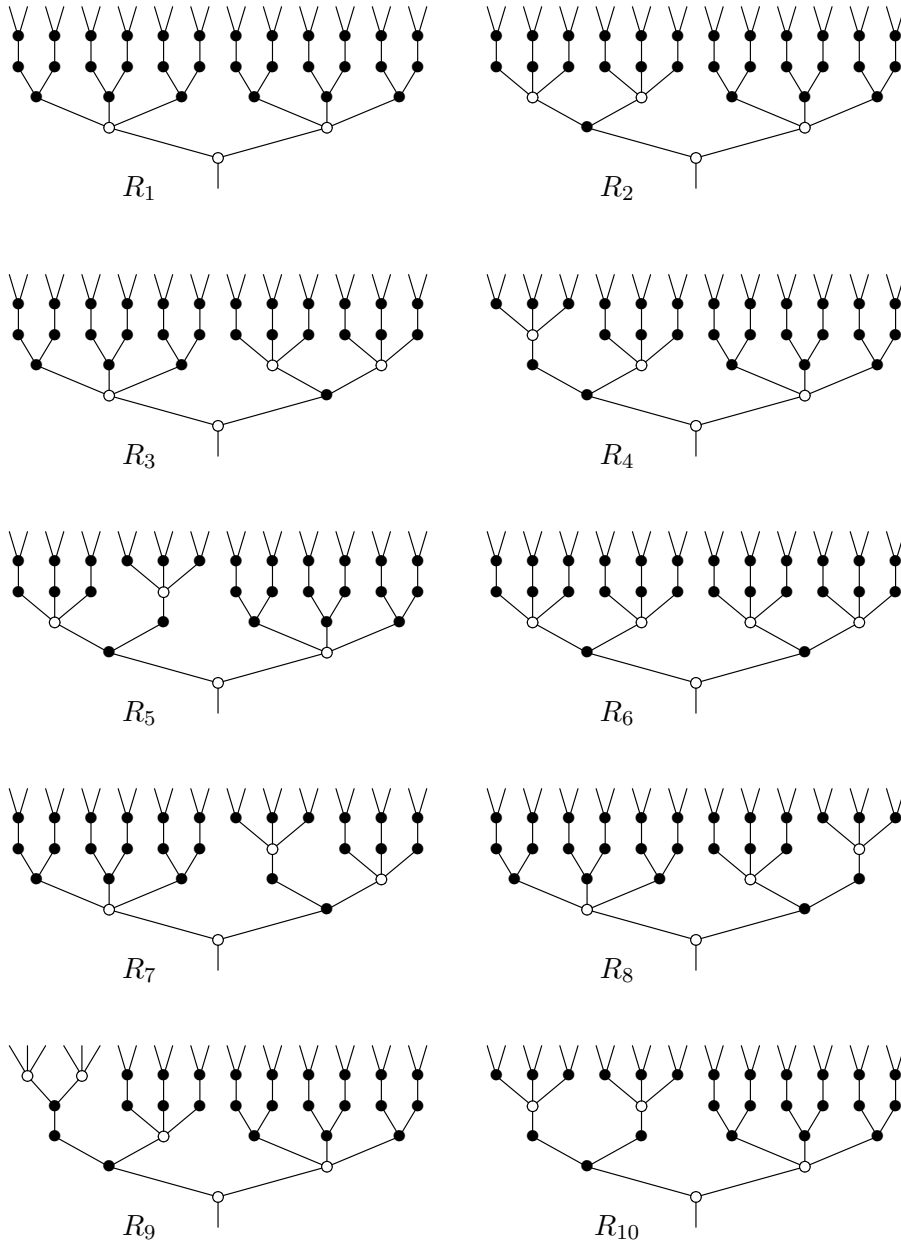
<b>Anexos</b>	<b>1</b>
<b>A Ejemplo de conjunto de shuffles</b>	<b>2</b>
<b>B Código Python</b>	<b>24</b>

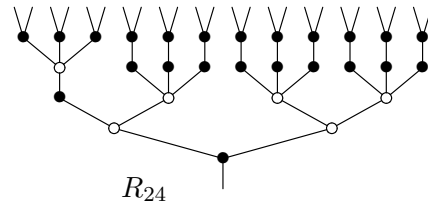
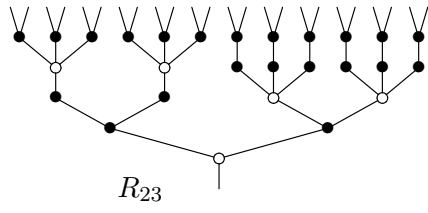
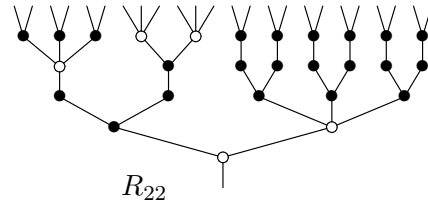
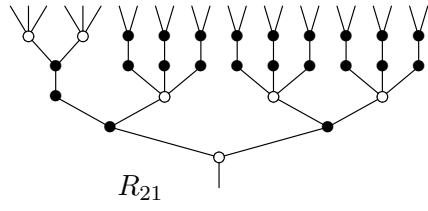
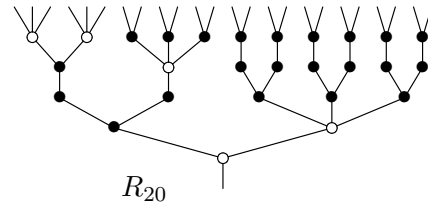
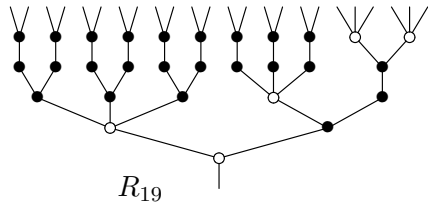
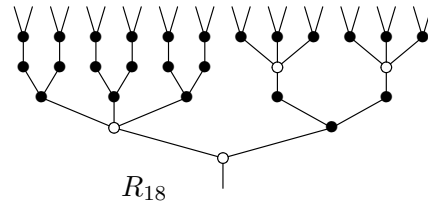
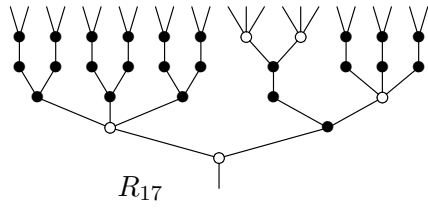
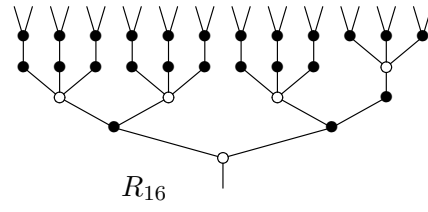
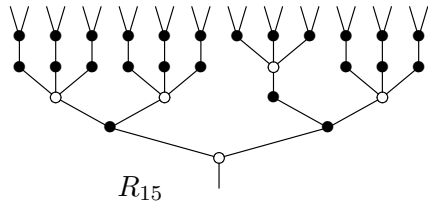
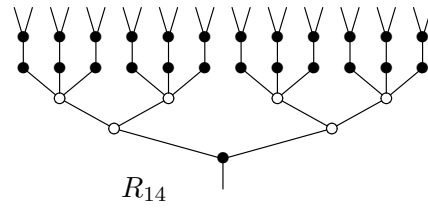
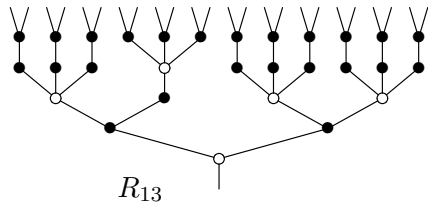
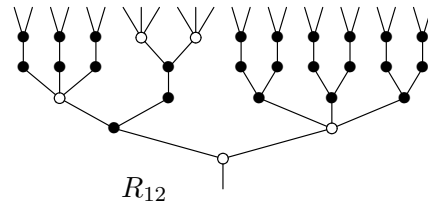
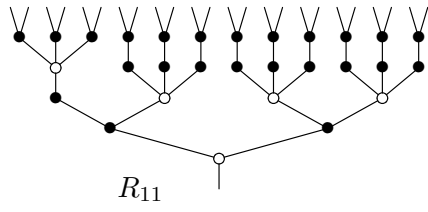
## A Ejemplo de conjunto de shuffles

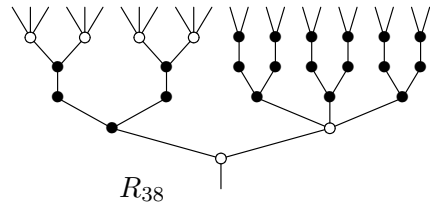
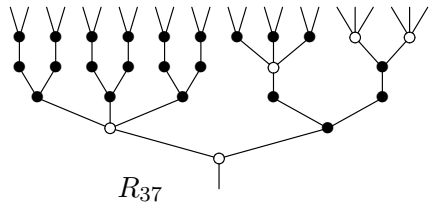
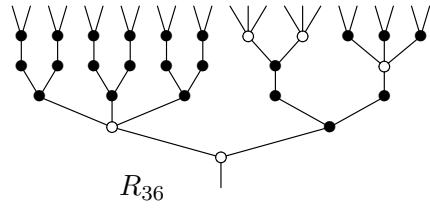
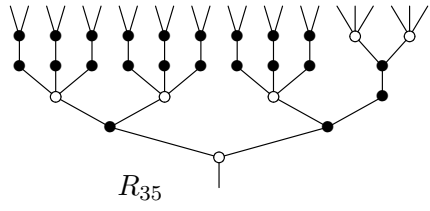
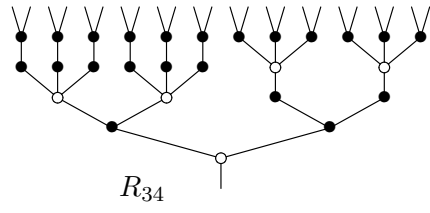
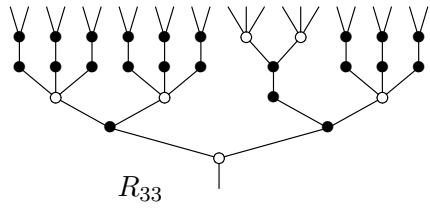
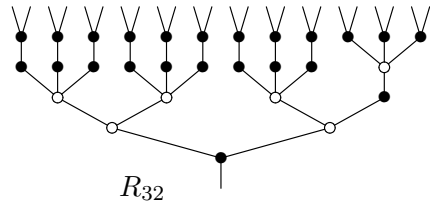
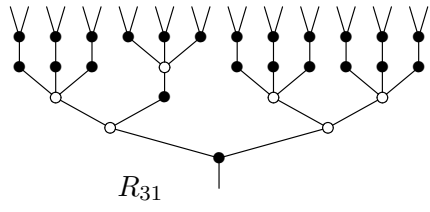
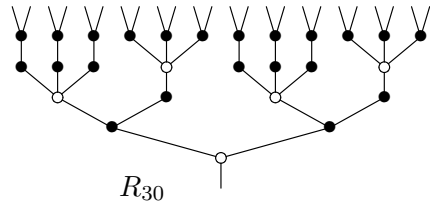
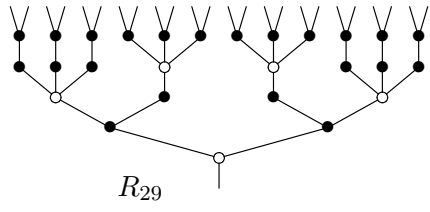
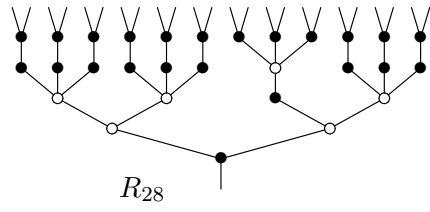
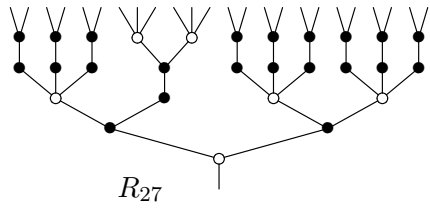
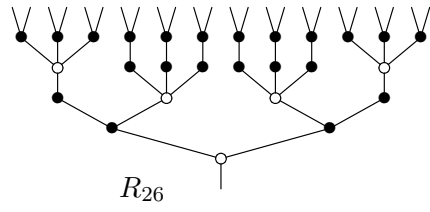
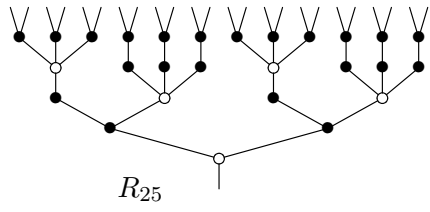
Pondremos un ejemplo completo del conjunto de shuffles que es generado por la clase *ShuffleLattice*. Sean  $S$  y  $T$  los árboles

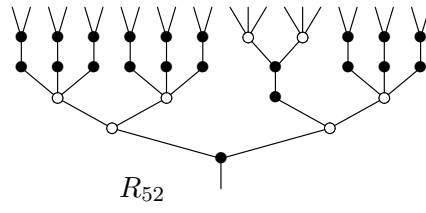
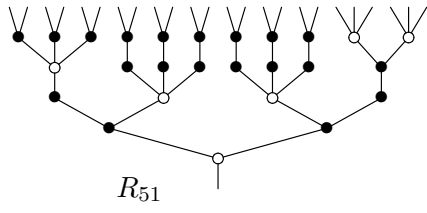
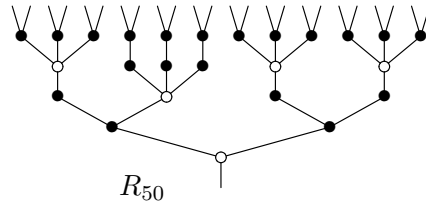
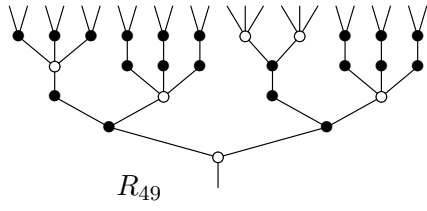
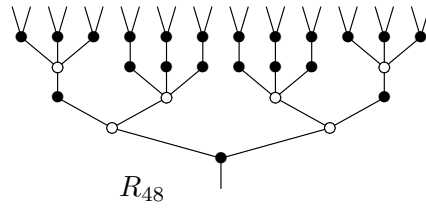
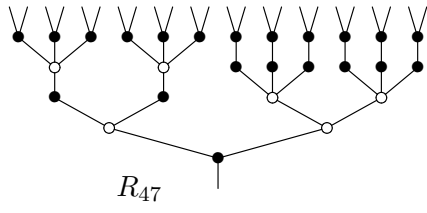
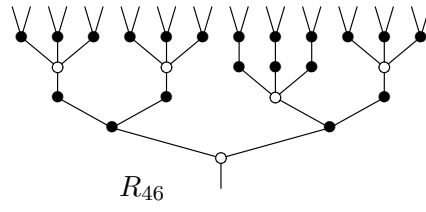
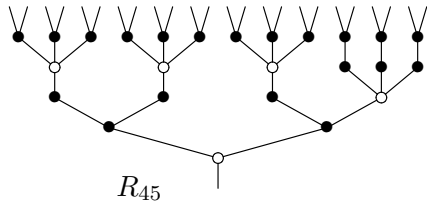
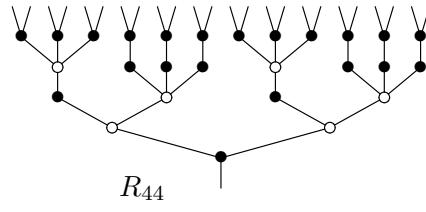
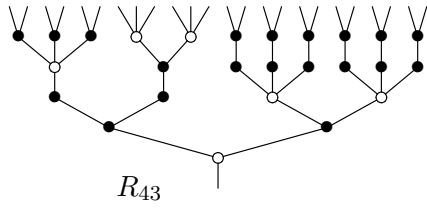
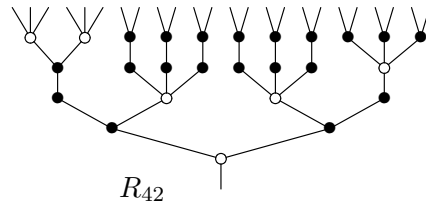
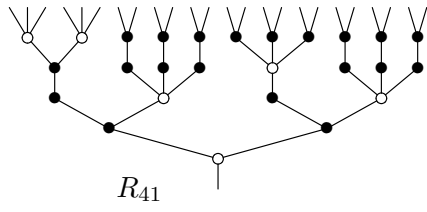
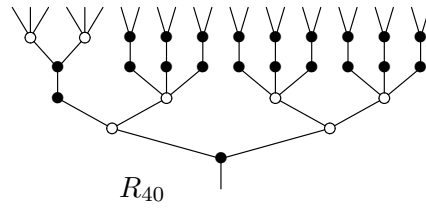
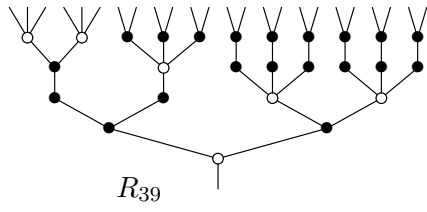


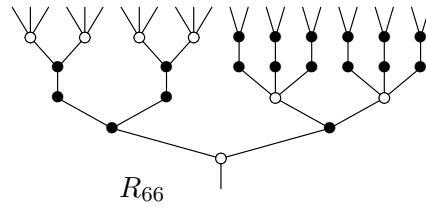
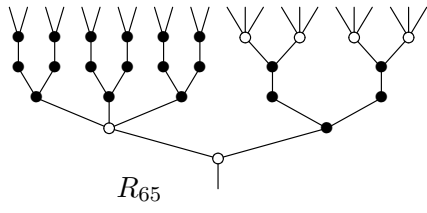
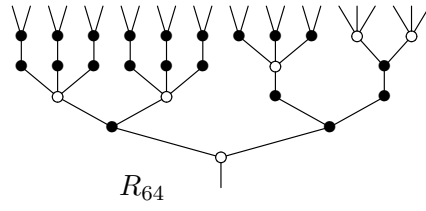
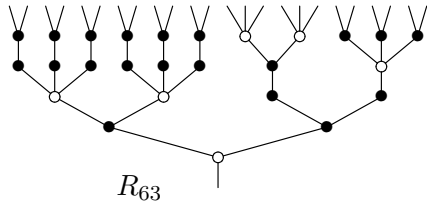
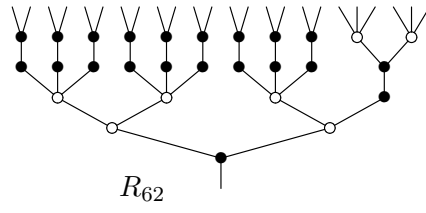
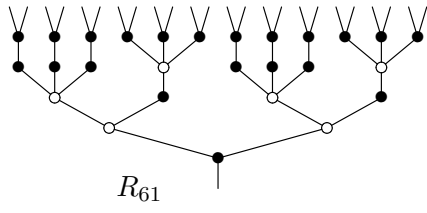
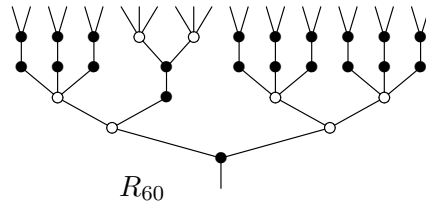
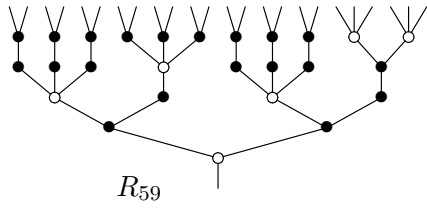
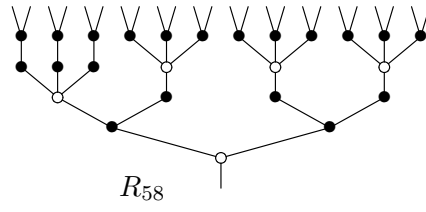
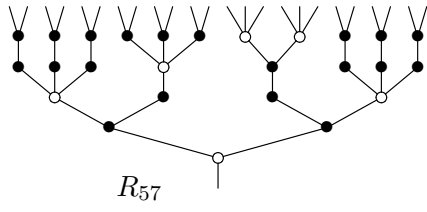
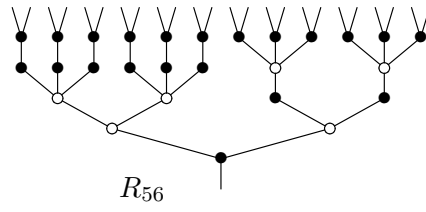
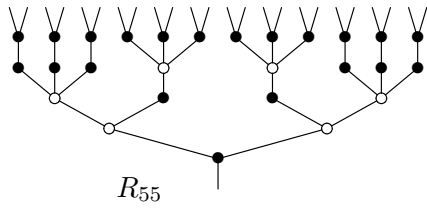
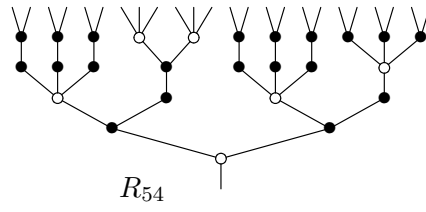
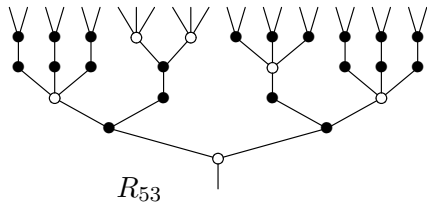
Esto es un ejemplo tedioso de calcular el conjunto de shuffles ya que según la función  $sh$  existen 296 shuffles diferentes. Mostramos todos ellos:

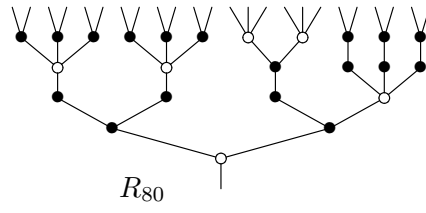
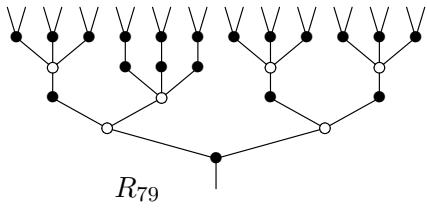
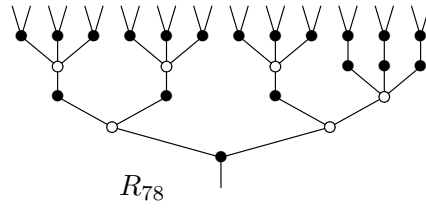
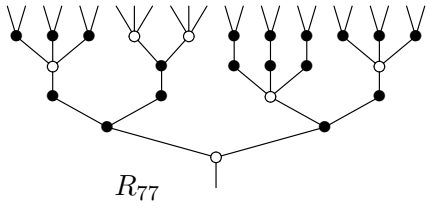
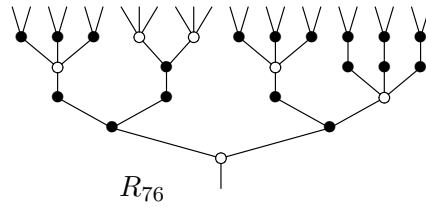
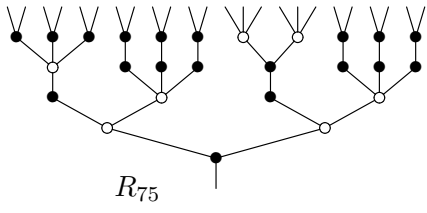
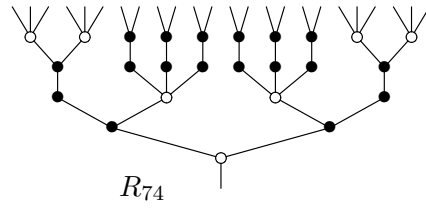
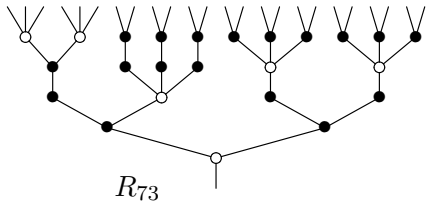
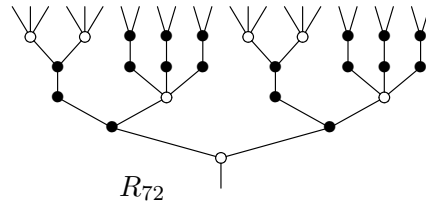
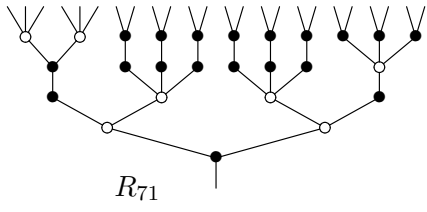
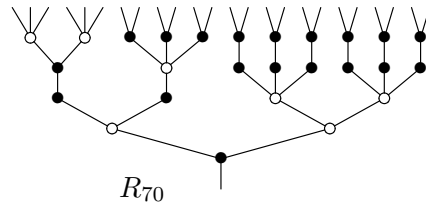
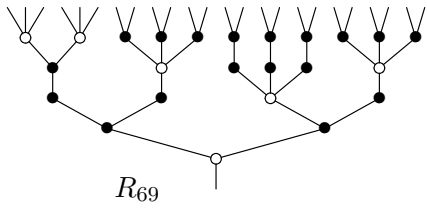
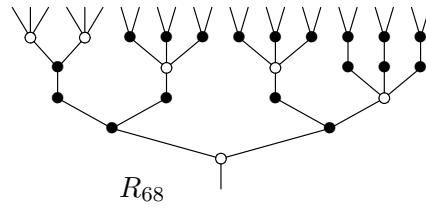
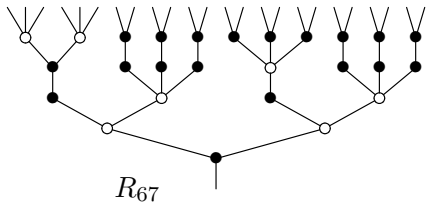


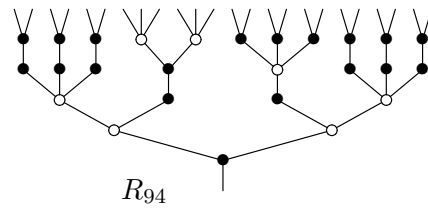
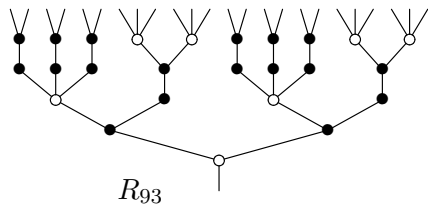
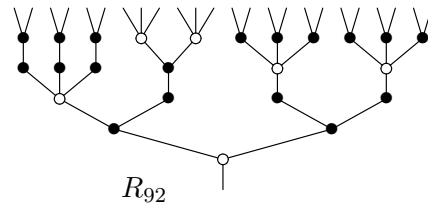
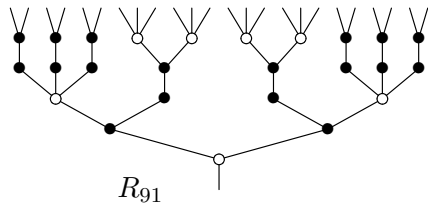
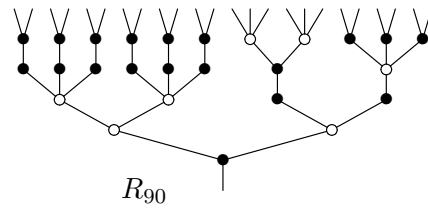
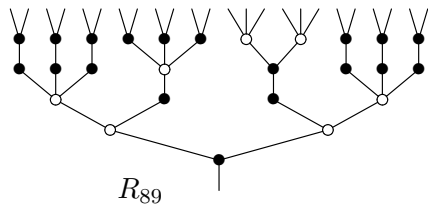
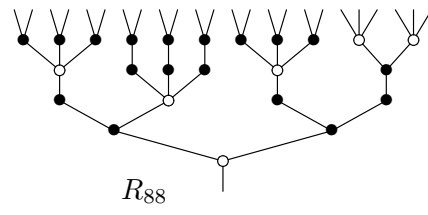
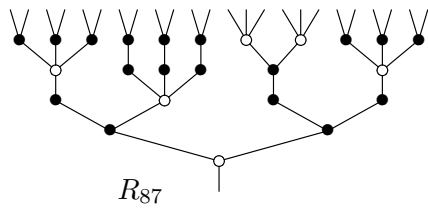
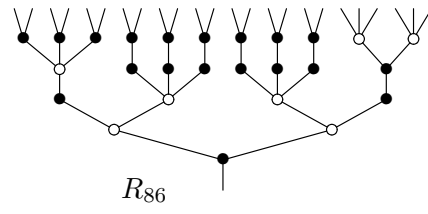
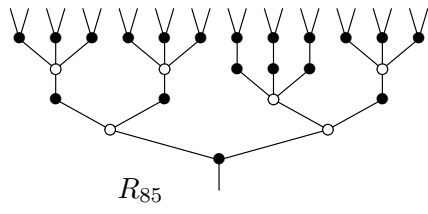
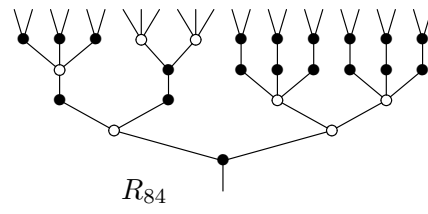
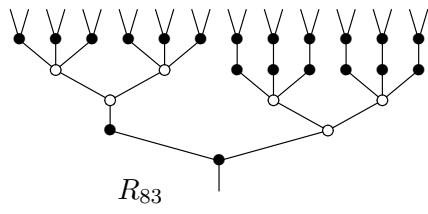
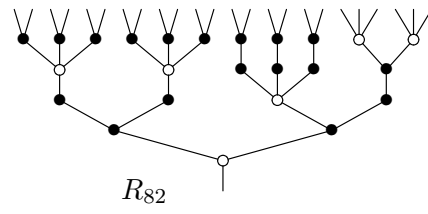
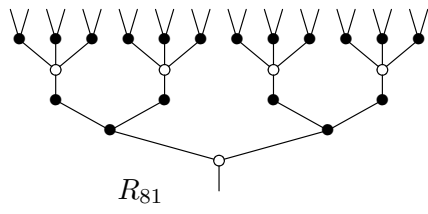




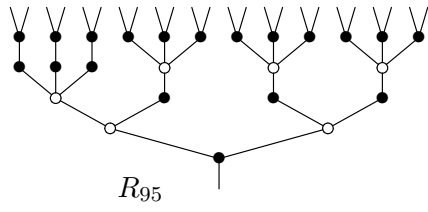




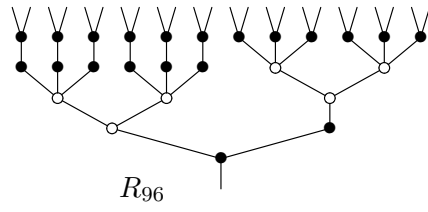




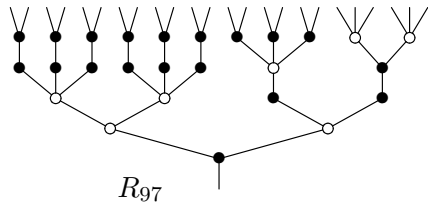




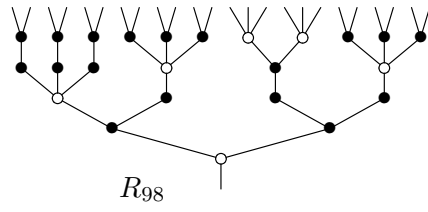
$R_{95}$



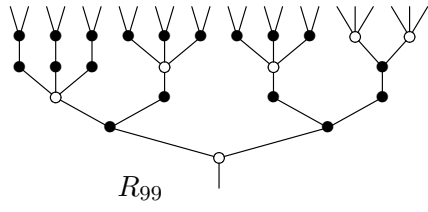
$R_{96}$



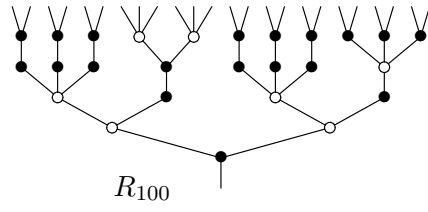
$R_{97}$



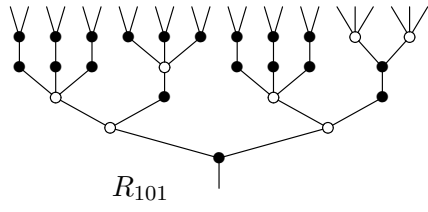
$R_{98}$



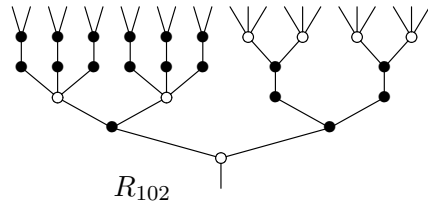
$R_{99}$



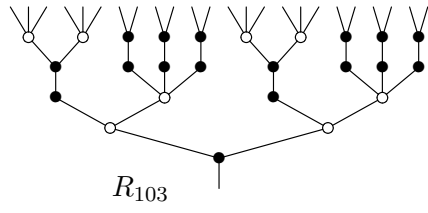
$R_{100}$



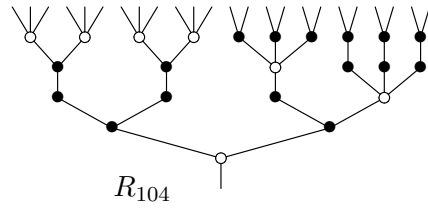
$R_{101}$



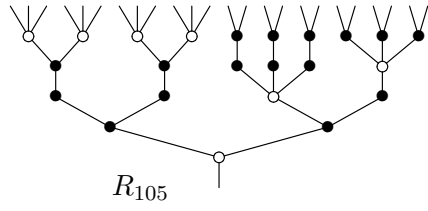
$R_{102}$



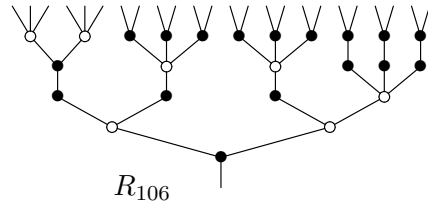
$R_{103}$



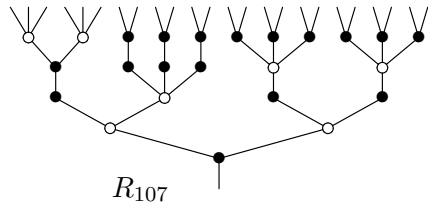
$R_{104}$



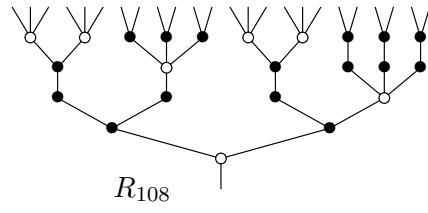
$R_{105}$



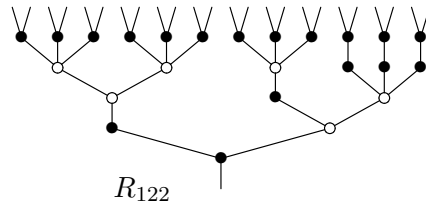
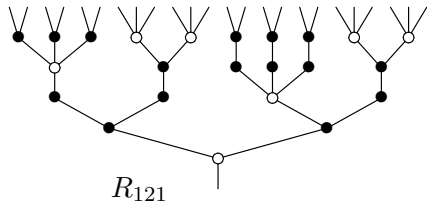
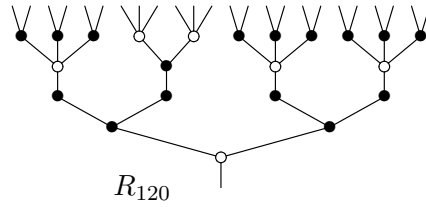
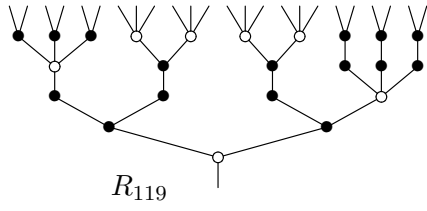
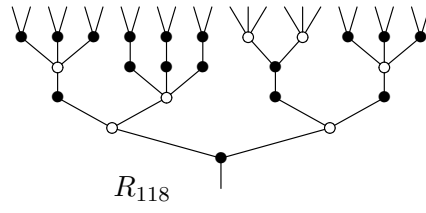
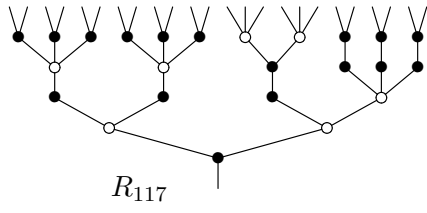
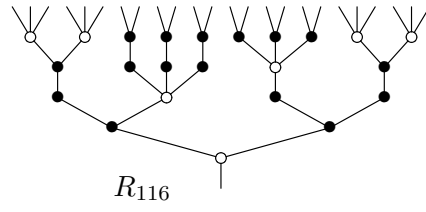
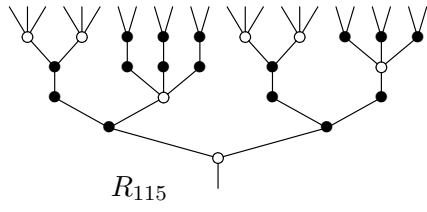
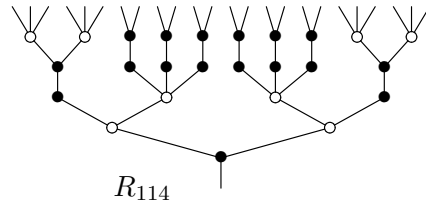
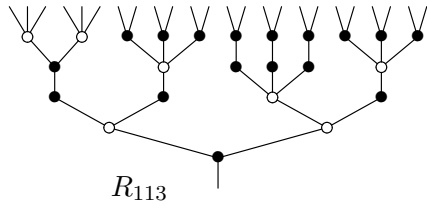
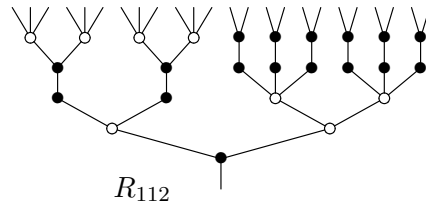
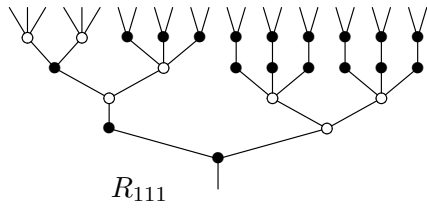
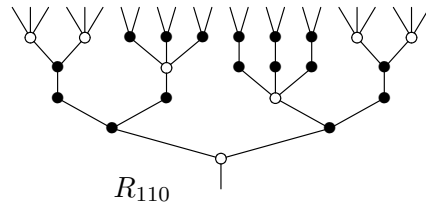
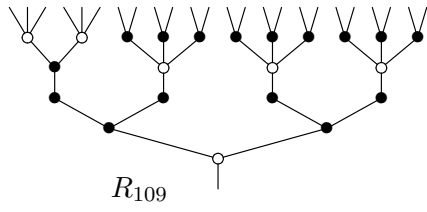
$R_{106}$

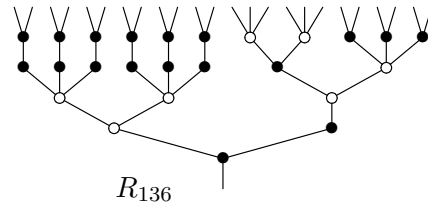
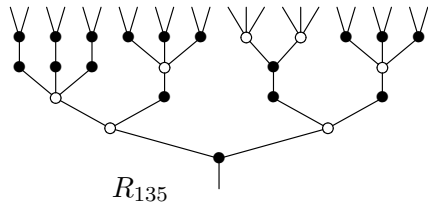
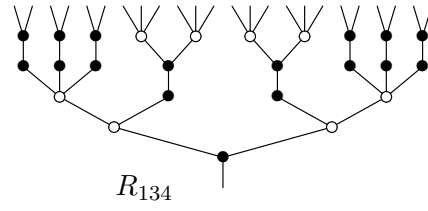
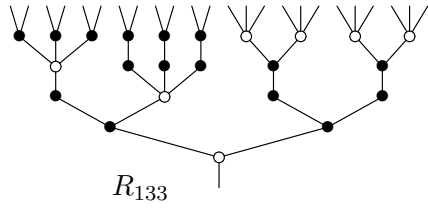
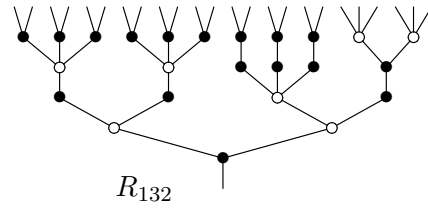
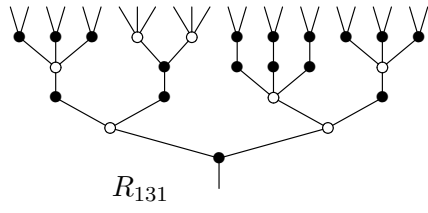
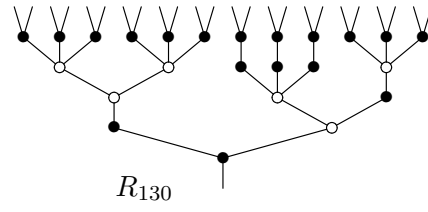
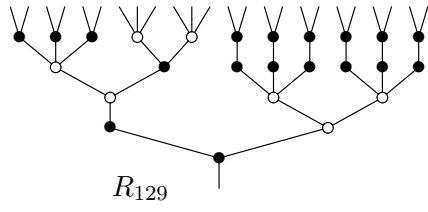
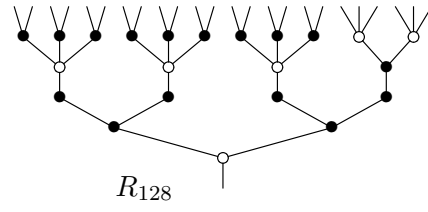
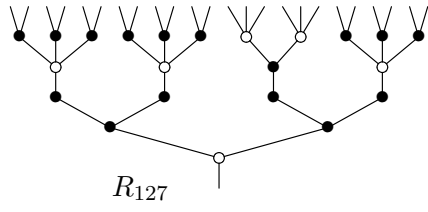
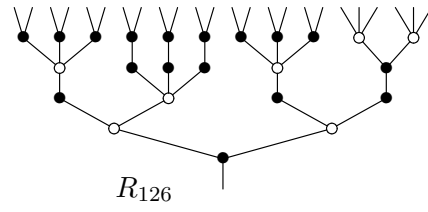
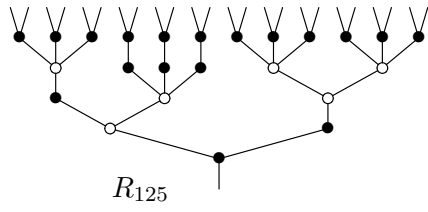
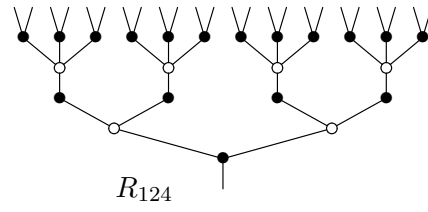
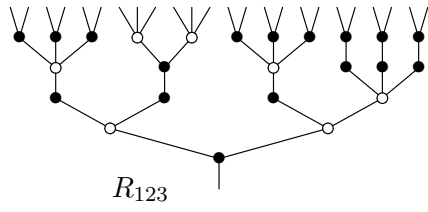


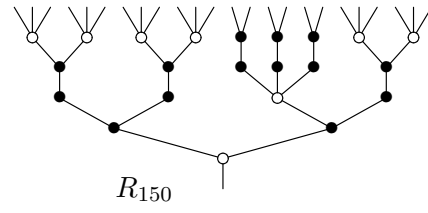
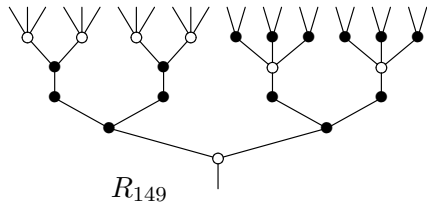
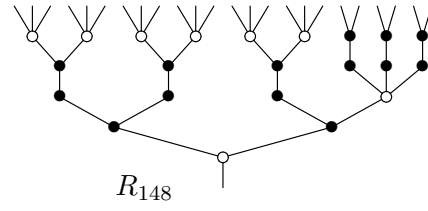
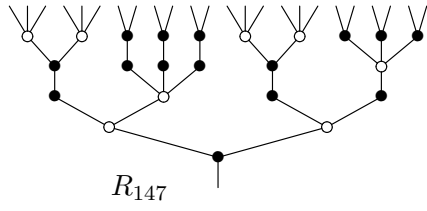
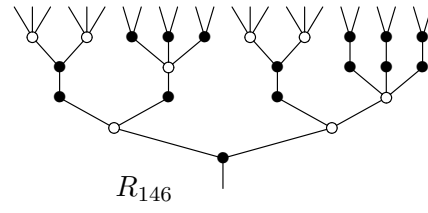
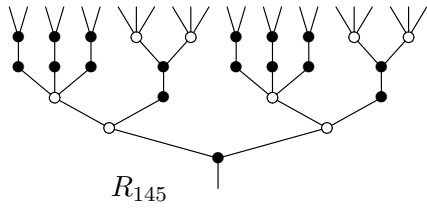
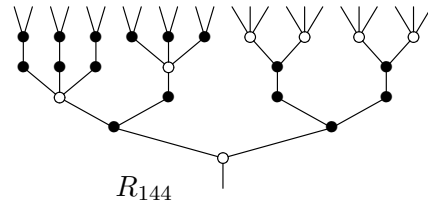
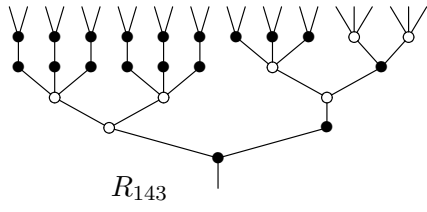
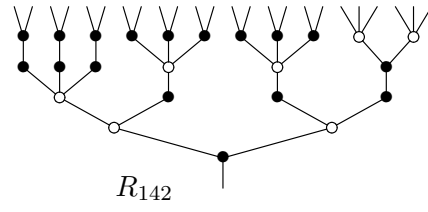
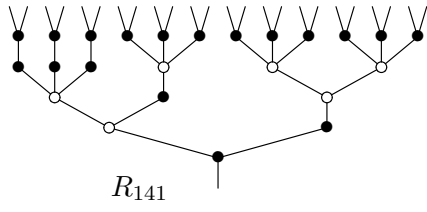
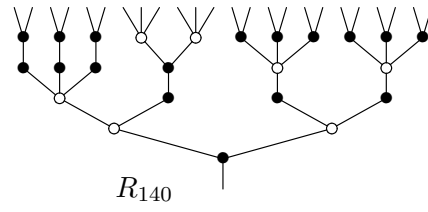
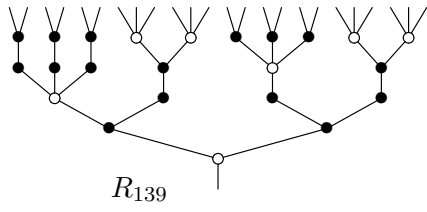
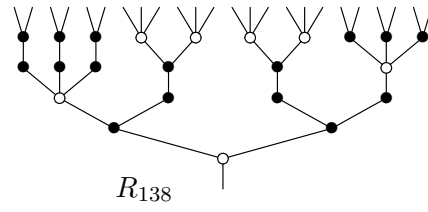
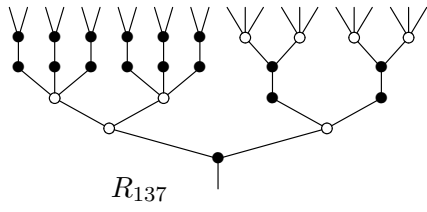
$R_{107}$

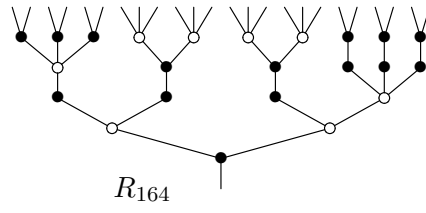
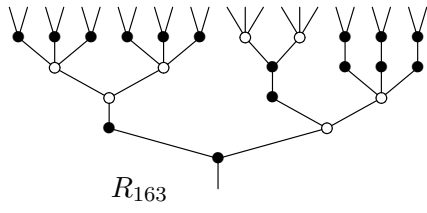
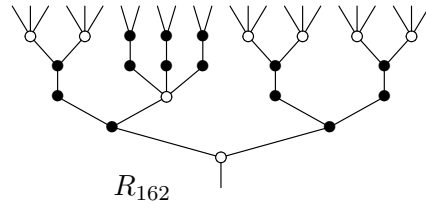
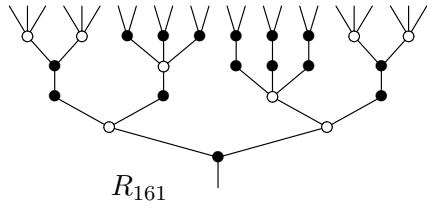
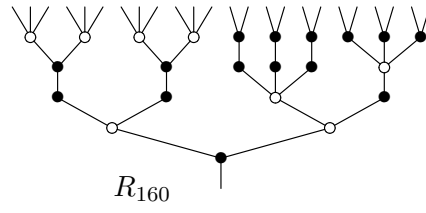
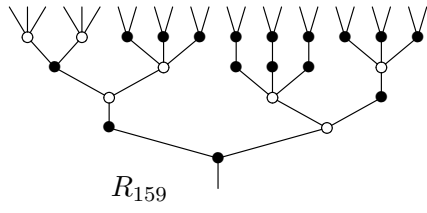
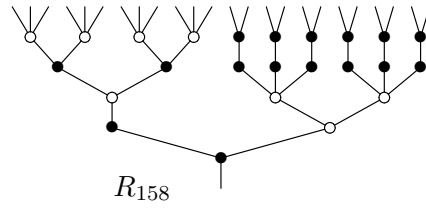
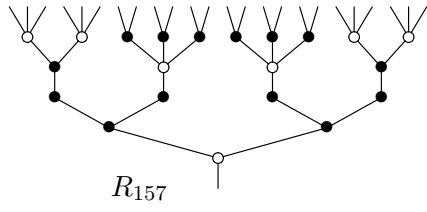
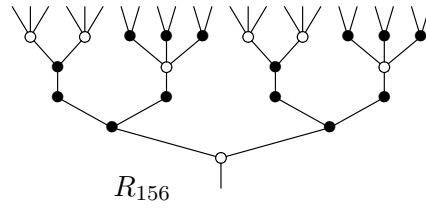
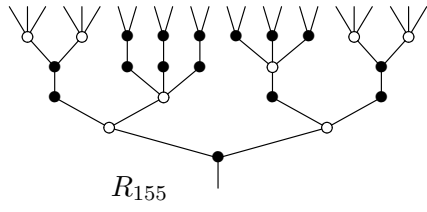
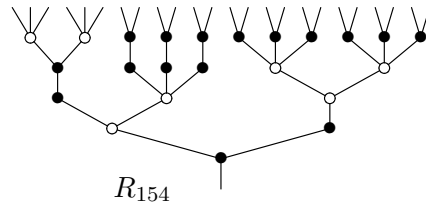
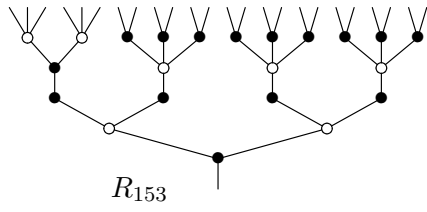
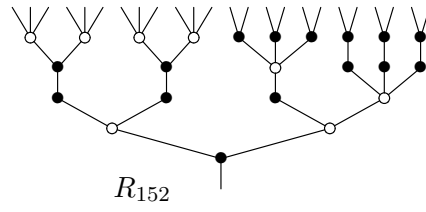
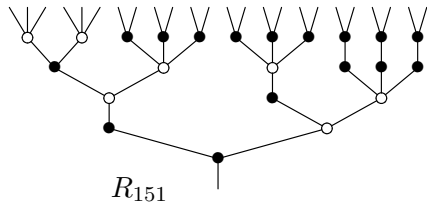


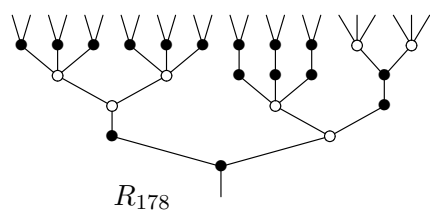
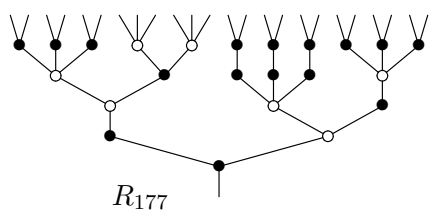
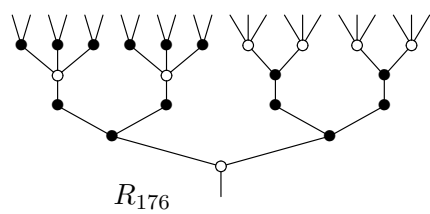
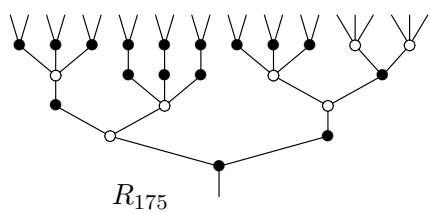
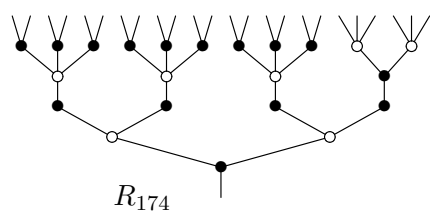
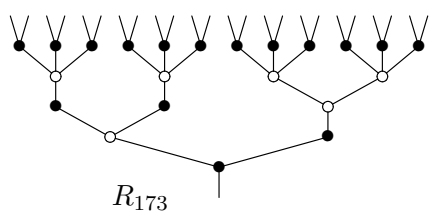
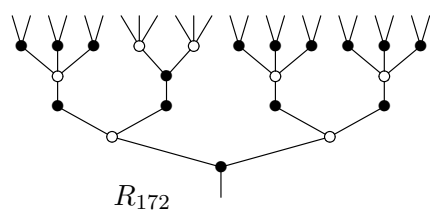
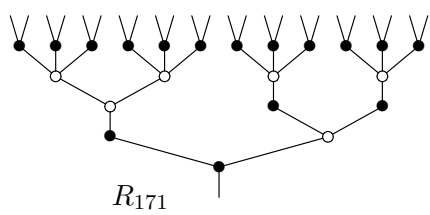
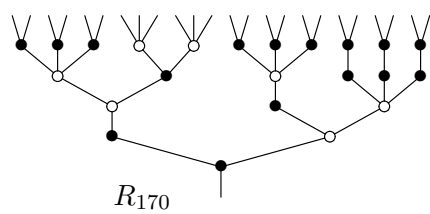
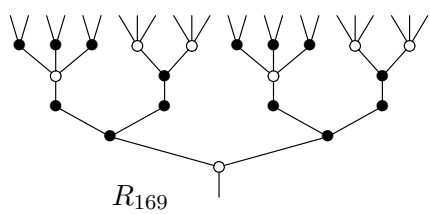
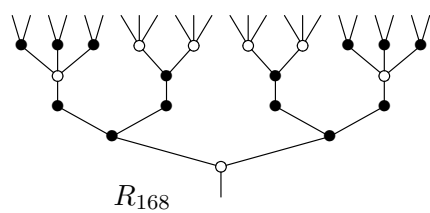
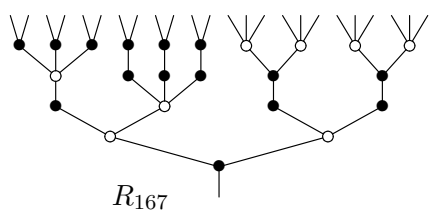
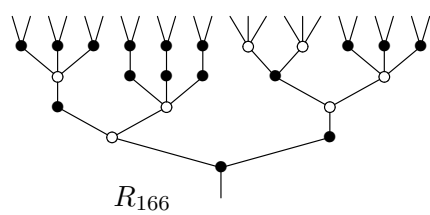
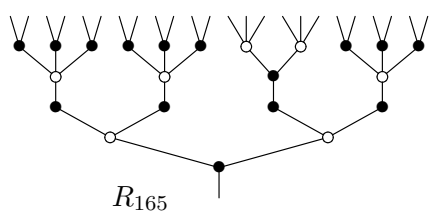
$R_{108}$

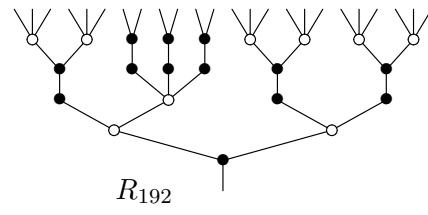
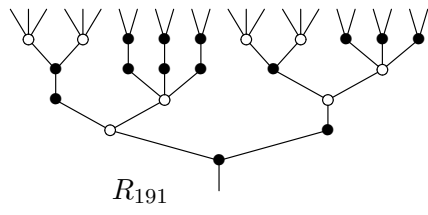
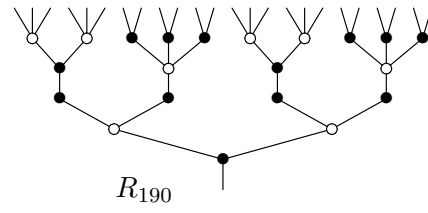
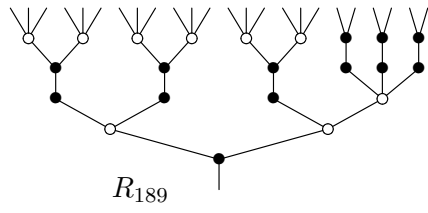
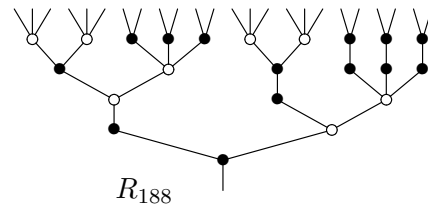
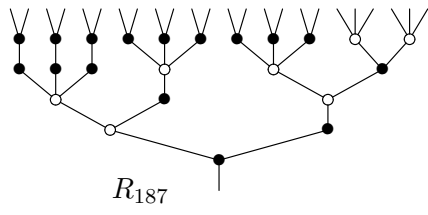
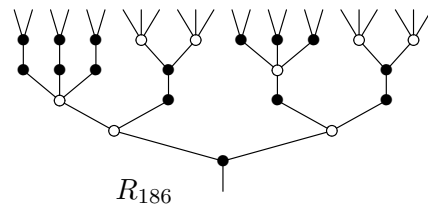
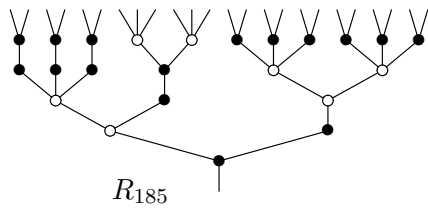
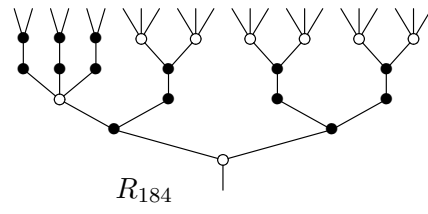
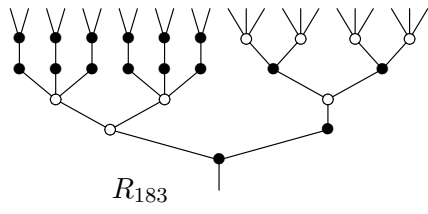
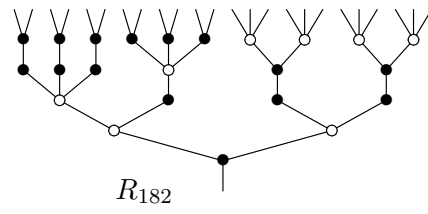
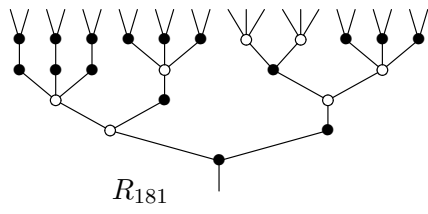
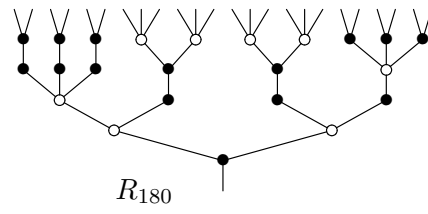
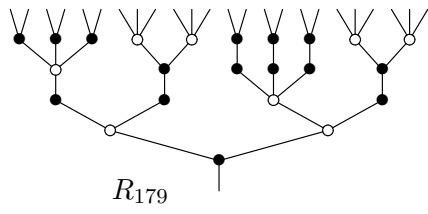


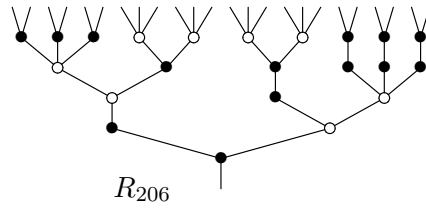
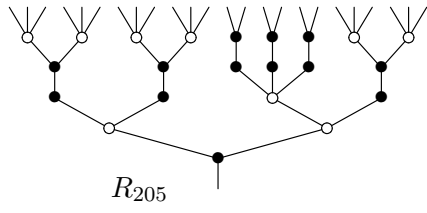
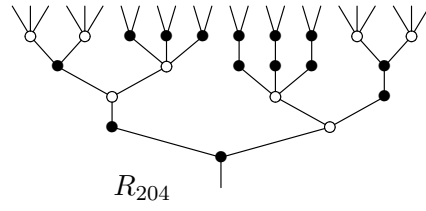
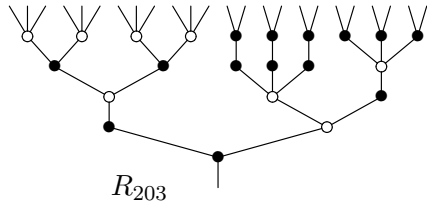
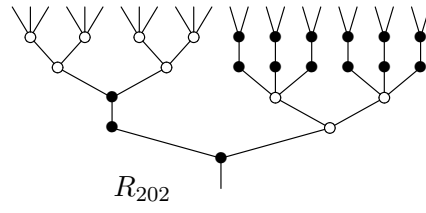
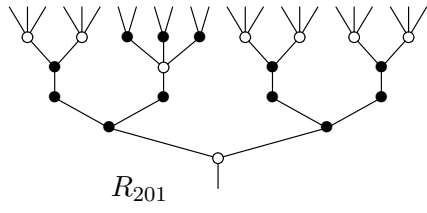
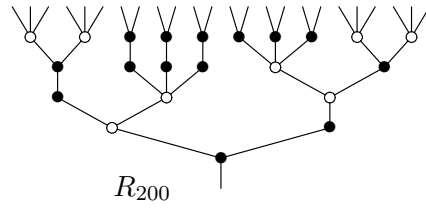
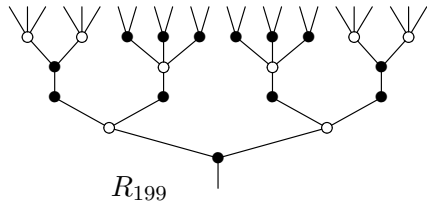
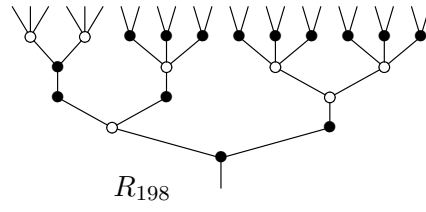
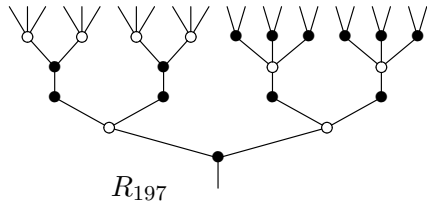
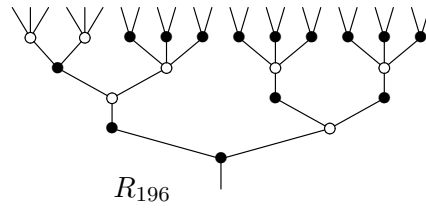
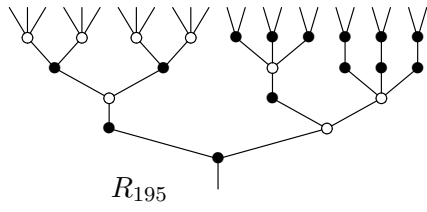
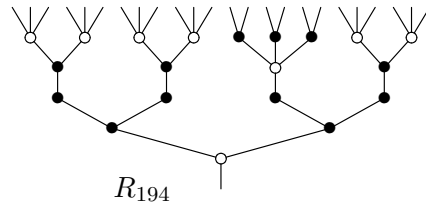
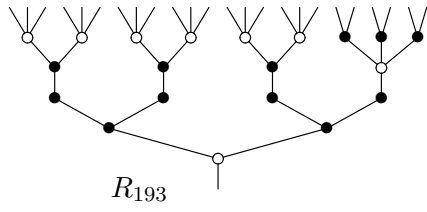




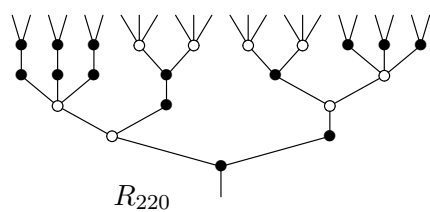
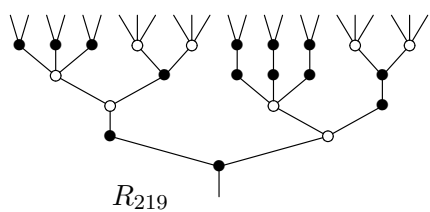
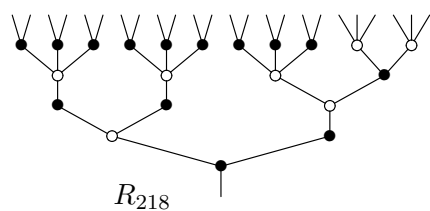
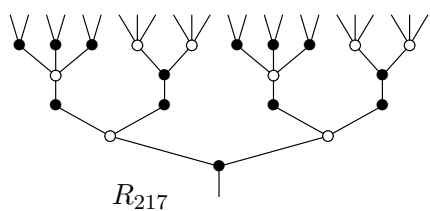
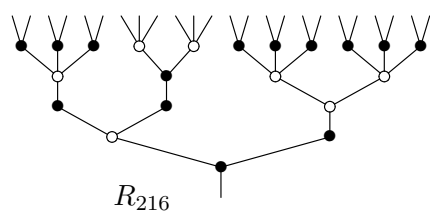
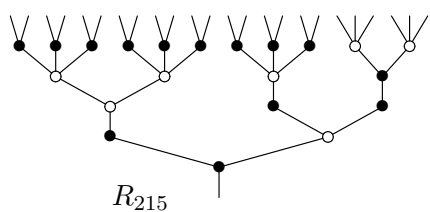
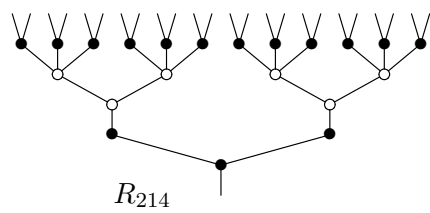
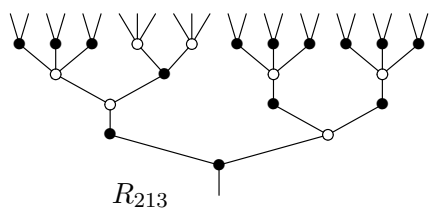
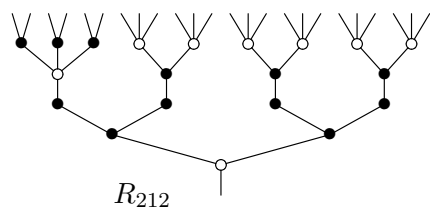
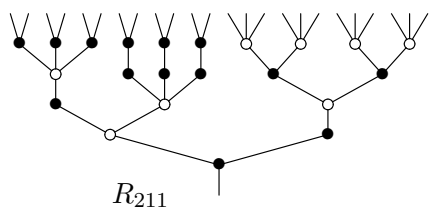
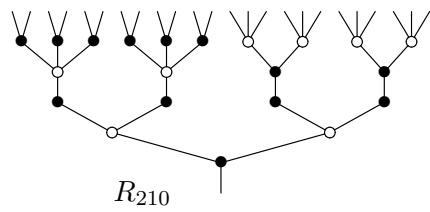
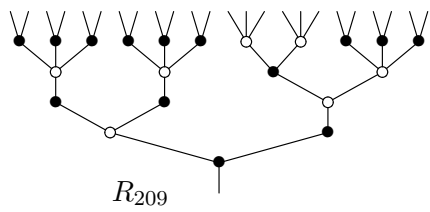
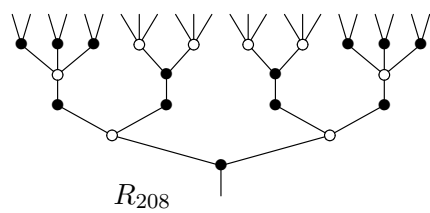
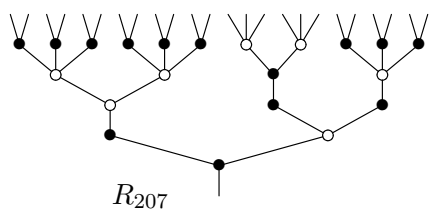


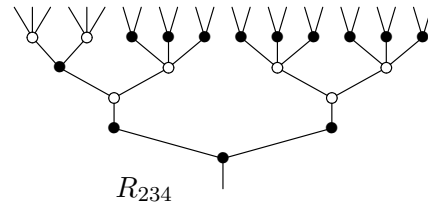
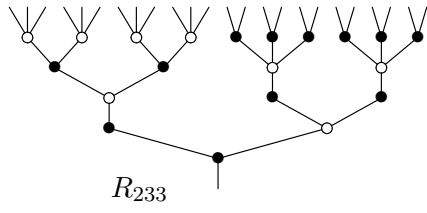
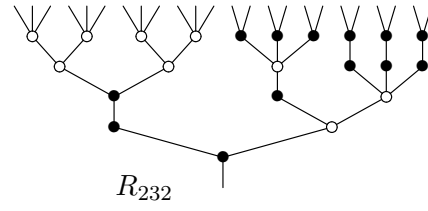
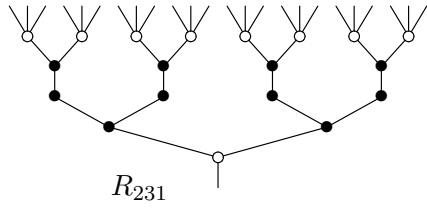
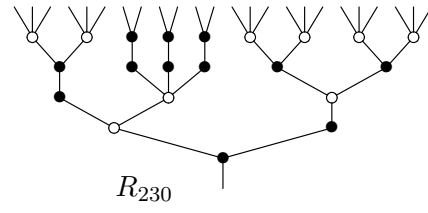
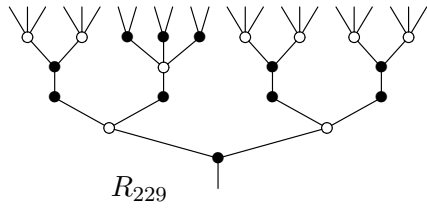
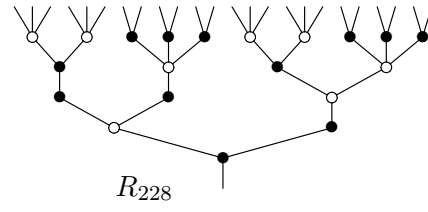
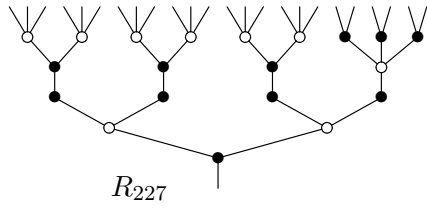
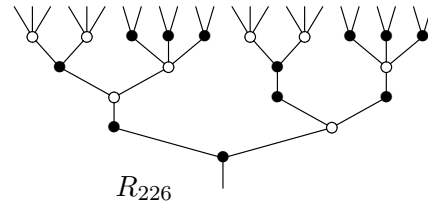
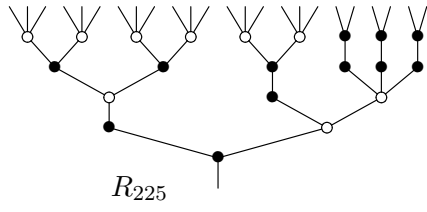
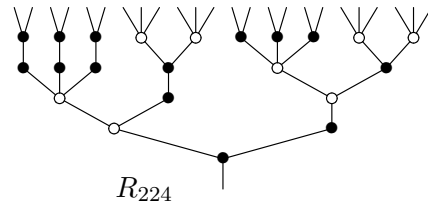
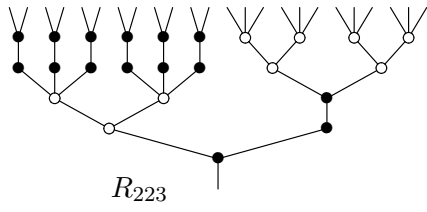
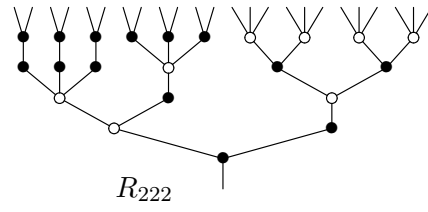
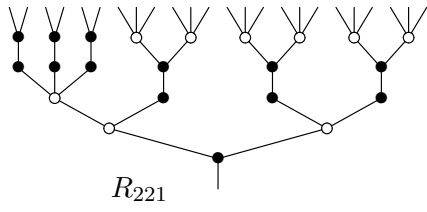


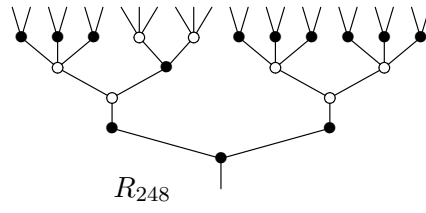
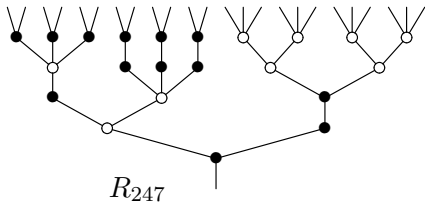
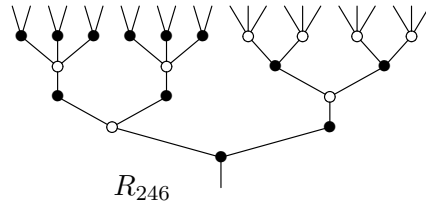
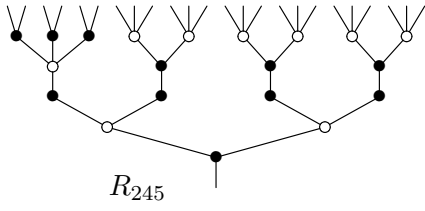
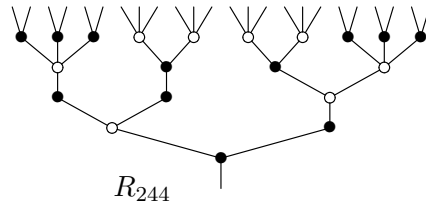
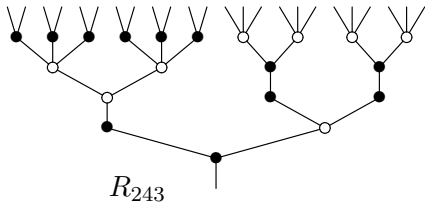
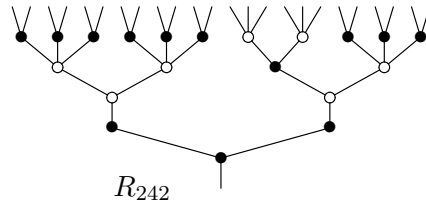
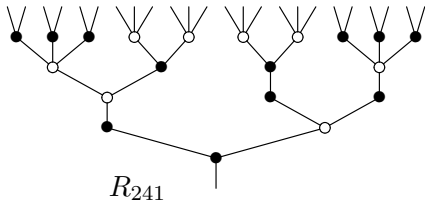
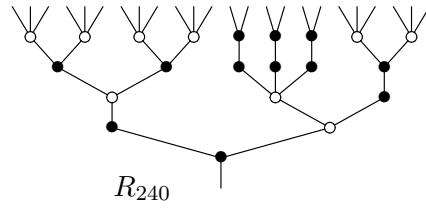
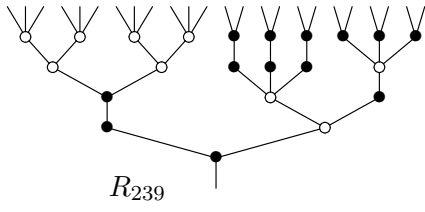
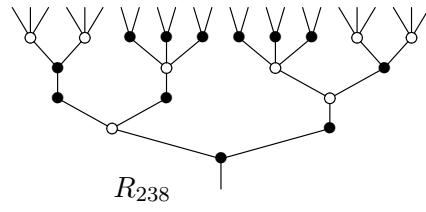
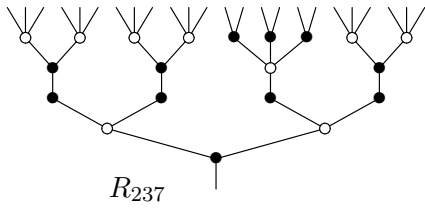
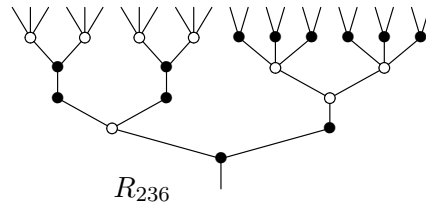
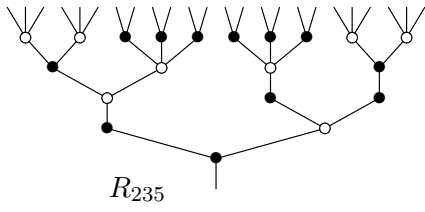


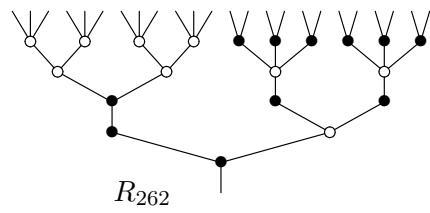
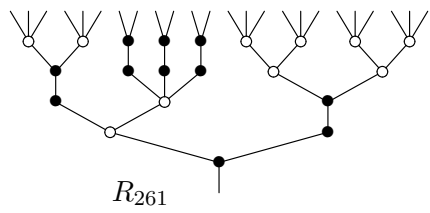
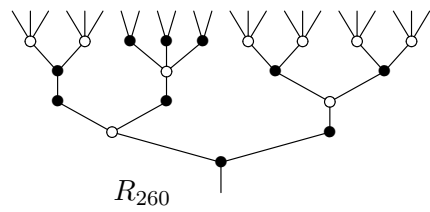
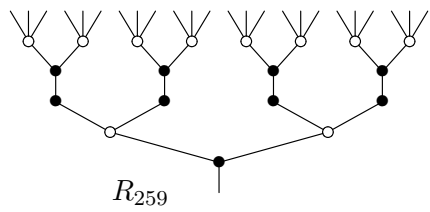
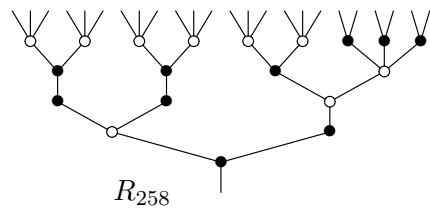
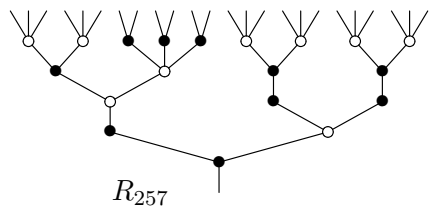
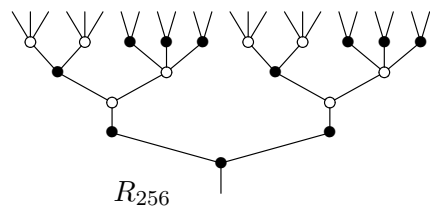
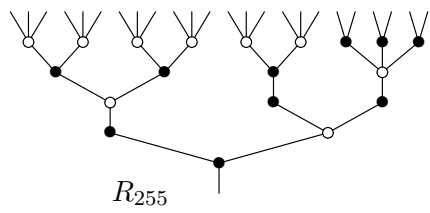
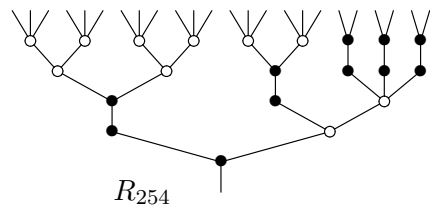
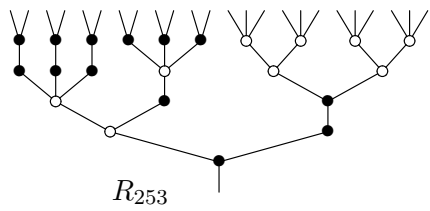
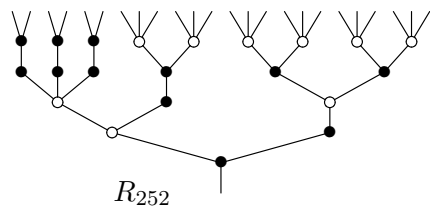
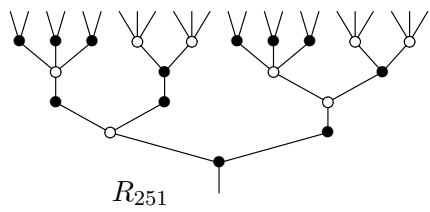
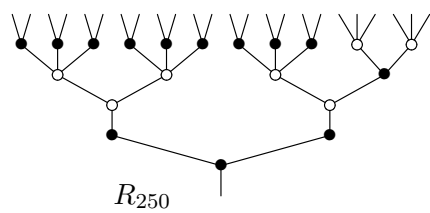
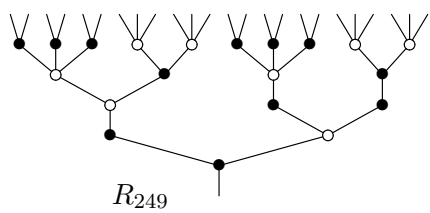


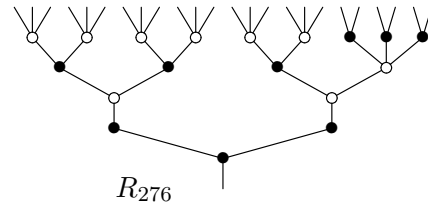
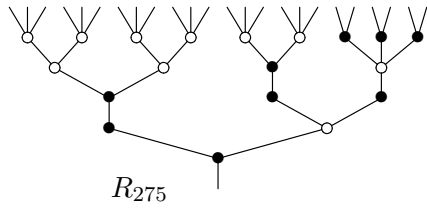
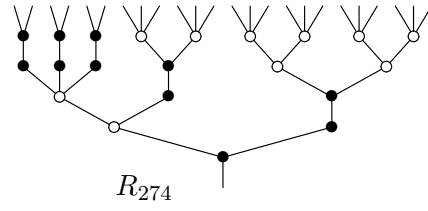
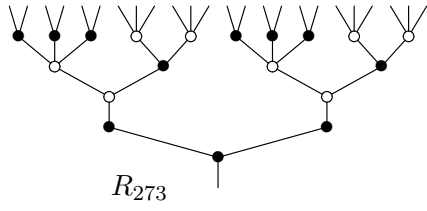
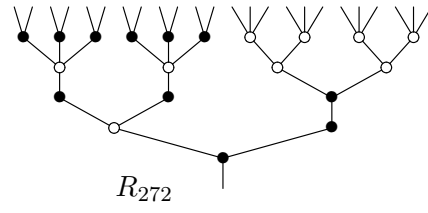
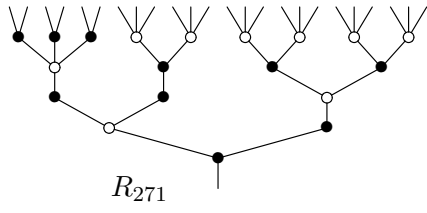
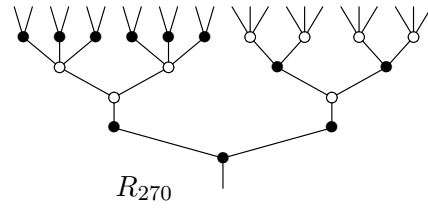
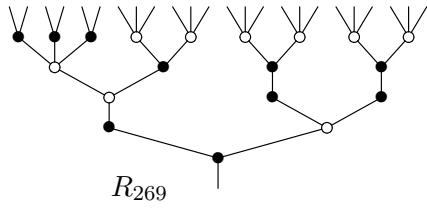
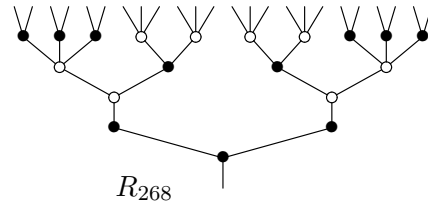
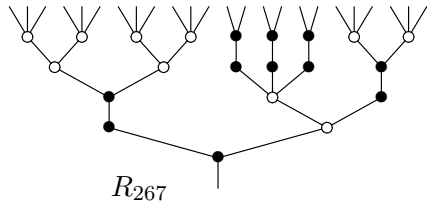
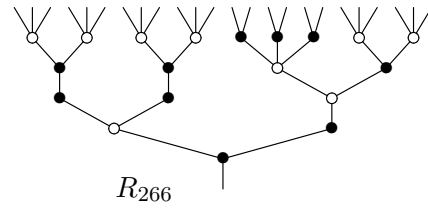
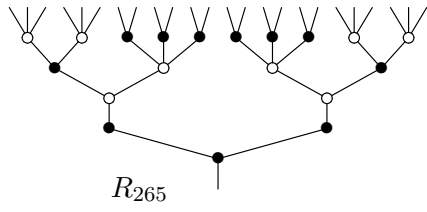
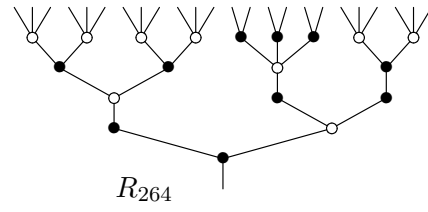
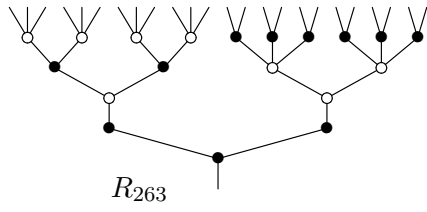


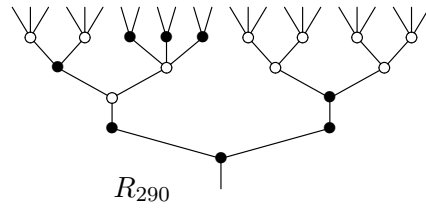
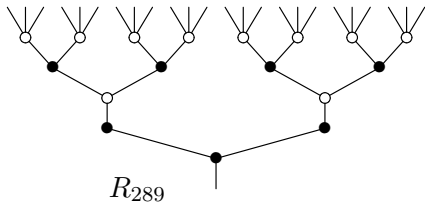
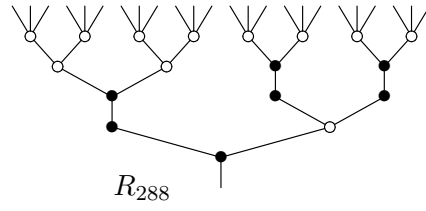
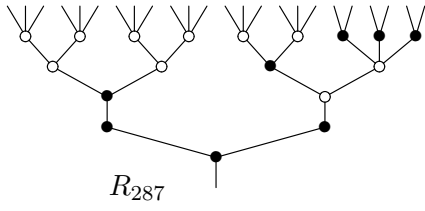
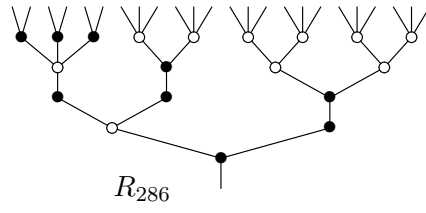
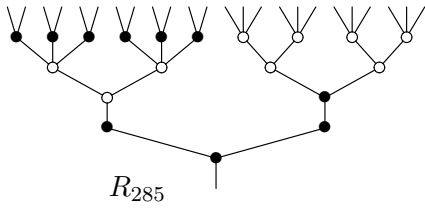
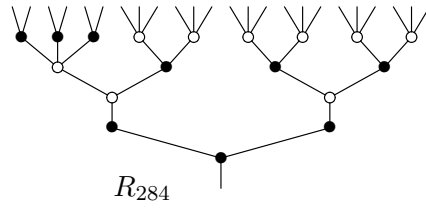
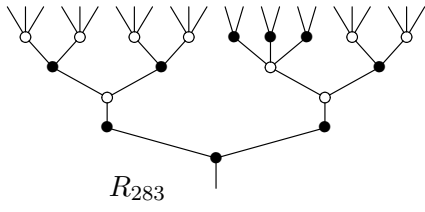
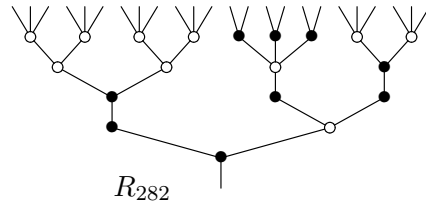
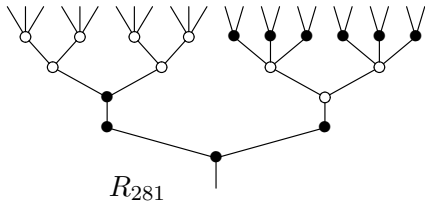
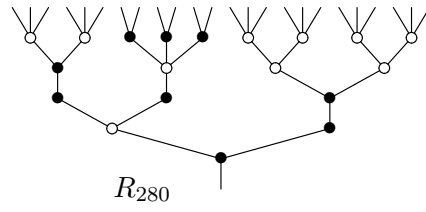
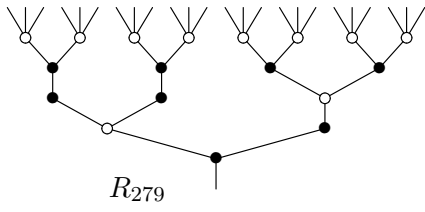
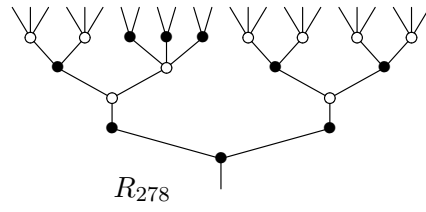
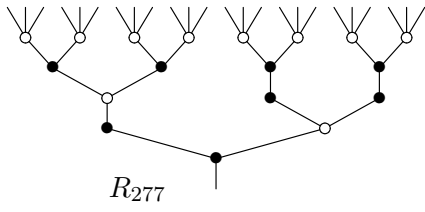


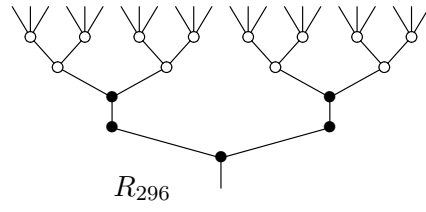
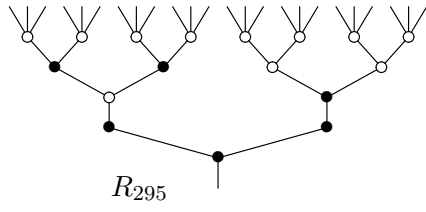
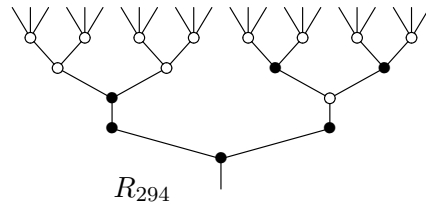
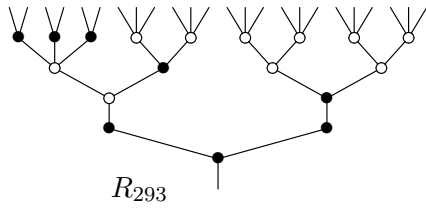
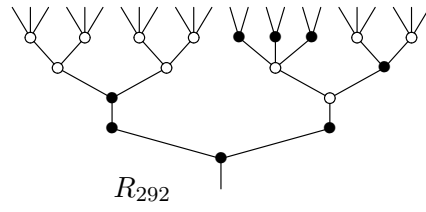
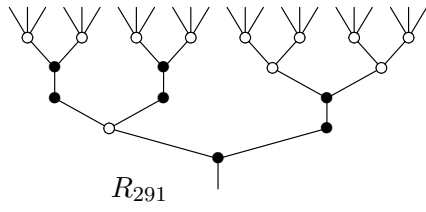












## B Código Python

Aquí se encuentra el código Python del paquete desarrollado pero también se puede encontrar en el repositorio público de código de Github: [Trees Shuffling](#).

### Lista de ficheros del paquete

1	operad.py . . . . .	25
2	tree.py . . . . .	26
3	utils.py . . . . .	28
4	operations.py . . . . .	30
5	lattice.py . . . . .	33
6	latex_gen.py . . . . .	37



Fichero 1: operad.py

```
class Operad:
    def __init__(self):
        self.colors = {}
        self.operations = {}

    def add_color(self, name, prop={}):
        if t := self.colors.get(name):
            self.colors[name] = (t, prop)
        else:
            self.colors[name] = prop
        return name, t

    def add_operation(self, name, prop={}):
        self.operations[name] = prop
        return name
```

```

class Tree:
    def __init__(
        self, operation, trunk, branches, operad, root=False
    ):
        self.root = root

        trunk, parent = operad.add_color(trunk, self)
        self.trunk = trunk
        if parent:
            parent.branches[trunk] = self
            self.depth = parent.depth + 1
        else:
            self.depth = 0
        self.branches = {
            operad.add_color(branch, self)[0]: uTree(
                branch, depth=self.depth
            )
            for branch in branches
        }
        self.node = operad.add_operation(operation, self)

    def print_edges(self):
        return "".join(
            [f"{str(self.trunk)}\n"]
            + [
                "\t" * (self.depth + 1)
                + f"{branch.print_edges()}\n"
                for branch in self.branches.values()
            ]
        )

    def print_nodes(self):
        return "".join(
            [f"{str(self.node)}\n"]
            + [
                "\t" * (self.depth + 1)
                + f"{branch.print_nodes()}\n"
                for branch in self.branches.values()
            ]
        )

    def print_self(self):
        return "".join(
            [f"{str(self)}\n"]
            + [
                "\t" * (self.depth + 1)
                + f"{branch.print_self()}\n"
                for branch in self.branches.values()
            ]
        )

```

```

    ]
)

def __str__(self):
    return (
        f"{self.node}({', '.join(self.branches.keys())}"
        + f";{self.trunk})"
    )

def __repr__(self):
    return (
        f"{self.node}({', '.join(self.branches.keys())}"
        + f";{self.trunk})"
    )

class uTree(Tree):
    def __init__(self, name, depth, root=False):
        self.trunk = name
        self.branches = {}
        self.node = None
        self.depth = depth
        self.root = root

```

Fichero 3: utils.py

```

from tree import Tree, uTree
from operad import Operad
import re

def string_to_tree_space(string, operad, sort=True):
    operads = []
    operations = string.split("|")
    if sort:
        operations = sorted(operations)
    for i, operation in enumerate(operations):
        operation, trunk, branches = extract_info(operation)
        operads.append(
            Tree(operation, trunk, branches, operad, not i)
        )
    return operads[0], operad

def tree_space_to_string(tree_space):
    _, operad = tree_space
    return "|" .join(
        str(tree) for tree in operad.operations.values()
    )

def _recursive_str(tree, s):
    s.append(str(tree))
    for branch in tree.branches.values():
        if not isinstance(branch, uTree):
            _recursive_str(branch, s)
    return s

def tree_to_string(tree):
    operations = sorted(_recursive_str(tree, []))
    return "|" .join(operations)

def extract_info(item):
    regex = r"(.)\(((.*)\)"
    match = re.match(regex, item)
    if not match:
        raise RuntimeError(
            f"Operation not defined correctly {item}"
        )
    operation, parameters = match.groups()
    branches, trunk = parameters.split(";")
    return (

```

```

        operation ,
        trunk ,
        branches.split(",") if branches else [] ,
    )

def sorted(operations):
    n = len(operations)
    items = list(
        map(lambda x: (extract_info(x), x), operations)
    )
    i = 0
    while i < n:
        item1 = items[i]
        for j in range(i + 1, n):
            item2 = items[j]
            if item1[0][1] in item2[0][2]:
                items.pop(i)
                items.insert(j, item1)
                break
        else:
            i += 1
    return list(map(lambda x: x[1], items))

def print_tree(tree, mode, sort=True):
    if mode in ["self", "nodes", "edges"]:
        T = string_to_tree_space(tree, Operad(), sort)
        print(eval(f"T[0].print_{mode}()"))
    else:
        raise RuntimeError(
            f"Print_tree_{mode}_{mode}_does_"
            + "not_exist ,_try:_self ,_nodes_or_edges"
        )

```

Fichero 4: operations.py

```
from copy import deepcopy
from utils import (
    tree_to_string,
    sorted,
    string_to_tree_space,
)
from tree import uTree

class TreeMerger:
    def __init__(self, S, T):
        self.S_og_space = S
        self.T_og_space = T
        self.tree_str = self.merge(
            deepcopy(S[0]), deepcopy(T[0])
        )

    def get_result(self):
        return (
            self.tree_str,
            self.S_og_space[1].operations,
            self.T_og_space[1].operations,
        )

    def merge(self, S, T):
        self.add_color_base(S, T.trunk)
        return tree_to_string(S)

    def add_color_edge(self, T, c, depth):
        T.trunk = f"{c}-{T.trunk}"
        T.branches = self.rename_keys(T, c, True)
        T.depth += depth
        for Ti in T.branches.values():
            self.add_color_edge(Ti, c, depth)
        return T

    def rename_keys(self, S, c, inv=False):
        aux = {}
        for i, Si in S.branches.items():
            if not inv:
                aux[f"{i}-{c}"] = Si
            else:
                aux[f"{c}-{i}"] = Si
        return aux

    def add_color_base(self, S, c):
        S.trunk = f"{S.trunk}-{c}"
        S.branches = self.rename_keys(S, c)
```

```

    for i, Si in S.branches.items():
        if isinstance(Si, uTree):
            S.branches[i] = self.add_color_edge(
                deepcopy(self.T_og_space[0]),
                i.split("-")[0],
                S.depth + 1,
            )
        else:
            self.add_color_base(Si, c)

class TreeManipulator:
    def __init__(
        self,
        tree_str,
        S_og_operations,
        T_og_operations,
        operad,
    ):
        self.tree_str = tree_str
        self.S_og_operations = S_og_operations
        self.T_og_operations = T_og_operations
        self.tree, self.operad = string_to_tree_space(
            tree_str, operad
        )

    def find_percolant_branches(self, found, X=None):
        if not X:
            X = self.tree
        if X.node in self.S_og_operations.keys():
            if all(
                branch.node in self.T_og_operations.keys()
                for branch in X.branches.values()
            ):
                found.append(X)

        for Xi in X.branches.values():
            self.find_percolant_branches(found, X=Xi)
        return found

    def make_percolation(self, location):
        operations = self.tree_str.split("|")
        branches = list(location.branches.values())
        new_node = branches[0].node
        old_node = location.node
        to_change = [str(location)] + [
            str(branch) for branch in branches
        ]
        old_oper = self.S_og_operations[old_node]

```

```

new_oper = self.T_log_operations[new_node]

morfed = []
for _, c in new_oper.branches.items():
    labels = map(
        lambda x: x + "-" + c.trunk,
        old_oper.branches.keys(),
    )
    morfed.append(
        f"{old_oper.node}({', '.join(labels)})"
        + f";{old_oper.trunk}-{c.trunk}"
    )
    labels = map(
        lambda x: old_oper.trunk + "-" + x,
        new_oper.branches.keys(),
    )
    morfed.append(
        f"{new_oper.node}({', '.join(labels)})"
        + f";{old_oper.trunk}-{new_oper.trunk}"
    )

for branch in to_change:
    operations.remove(branch)
operations += morfed
operations = sorted(operations)
new_tree = "|".join(operations)
return new_tree

```



Fichero 5: lattice.py

```

from operations import TreeMerger , TreeManipulator
from operad import Operad
from tree import uTree
from utils import sorted , print_tree
from latex_gen import tree_to_latex , separator
from copy import deepcopy
from collections import defaultdict
from math import prod

class ShuffleLattice:
    def __init__(self , S , T):
        self.tm_i = TreeMerger(deepcopy(S) , deepcopy(T))
        (
            self.i_tree ,
            *self.operations ,
        ) = self.tm_i.get_result()
        self.skeleton = defaultdict(set)
        self.dictionary = defaultdict(str)
        self.nb_percolations = self.sh(S[0] , T[0])
        self.initialize()
        self.generate_shuffles()

    def initialize(self):
        self.dictionary["R_{1}"] = set(
            self.i_tree.split("|")
        )
        self.skeleton["R_{1}"] = set()

    def sh(self , S , T):
        if isinstance(S , uTree) or (S.root and not S.node):
            return 1
        if isinstance(T , uTree) or (T.root and not T.node):
            return 1
        return prod(
            [self.sh(Si , T) for Si in S.branches.values()]
        ) + prod(
            [self.sh(S , Ti) for Ti in T.branches.values()]
        )

    def generate_shuffles(self):
        queue = []
        queue.append(self.i_tree)

        while len(queue):
            tree = queue.pop(0)
            manipulator = TreeManipulator(
                tree , *self.operations , Operad()
            )

```

```

    )

    tree_key = self.find_key(set(tree.split("|")))
    if not tree_key:
        raise RuntimeError(
            "Parent_key_should_be"
            + f"present_on_dict_{tree}"
        )

    for (
        location
    ) in manipulator.find_percolant_branches(
        found=[]
    ):
        fingerprint = manipulator.make_percolation(
            location
        )
        if not self.fingerprint_in_skeleton(
            fingerprint, tree_key
        ):
            queue.append(fingerprint)

def fingerprint_in_skeleton(
    self, fingerprint, parent_key
):
    operations = set(fingerprint.split("|"))
    if key := self.find_key(operations):
        self.skeleton[key].add(parent_key)
        return True
    key = "R_" + str(len(self.dictionary) + 1) + "_"
    self.dictionary[key] = operations
    self.skeleton[key].add(parent_key)
    return False

def find_key(self, operations):
    re = filter(
        lambda key: operations == self.dictionary[key],
        self.dictionary.keys(),
    )
    return next(re, None)

def get_dictionary(self):
    return {
        key: "|" . join(sorted(list(values)))
        for key, values in self.dictionary.items()
    }

def print_result_skeleton(self):
    print("Number_of_trees:", self.nb_percolations)

```

```

    print(
        "Number of trees generated: ",
        len(self.dictionary),
    )
    print(
        "Dictionary of trees: ",
        dict(self.get_dictionary()),
    )
    print("Skeleton of trees: ", dict(self.skeleton))

def print_latex(
    self,
    key=None,
    sort=True,
    size_f=(15, 10),
    labels=True,
    label_b=(3, (-2, 0)),
    between=10,
    every=5,
    slice=slice(None),
):
    sorted_dict = self.get_dictionary()
    if key:
        print(f"Tree_{key}: {sorted_dict[key]}")
        tree_to_latex(
            sorted_dict[key],
            sort=sort,
            size_f=size_f,
            labels=labels,
            label_b=label_b,
            tree_name=key,
        )
    else:
        print("$$")
        li = list(sorted_dict.items())
        for i, (name, value) in enumerate(li[slice]):
            tree_to_latex(
                value,
                sort=sort,
                size_f=size_f,
                labels=labels,
                label_b=label_b,
                tree_name=name,
            )
            separator(between)
        if not (i + 1) % every:
            print("$$")
            print("")
            print("$$")

```

```

        print(" $$")

def print_trees(
    self,
    key=None,
    sort=True,
    mode="self",
    slice=slice(None),
):
    sorted_dict = self.get_dictionary()
    if key:
        print(f"Tree_{key}:_{sorted_dict[key]}")
        print_tree(
            sorted_dict[key],
            mode=mode,
            sort=sort,
        )
    else:
        li = list(sorted_dict.items())
        for name, value in li[slice]:
            print(f"Tree_{name}:_{value}")
            print_tree(
                value,
                mode=mode,
                sort=sort,
            )

```

```

import igraph as ig
from utils import string_to_tree_space
from operad import Operad

def tree_to_latex(
    tree,
    sort=False,
    size_f=(15, 10),
    labels=True,
    label_b=(3, (-2, 0)),
    tree_name=None,
):
    T = string_to_tree_space(tree, Operad(), sort=sort)
    G = ig.Graph()
    _recursive_add_edges(T[0], G)
    layout = G.layout_reingold_tilford(mode="in", root=[0])
    vertices = list(zip(G.vs()["label"], layout.coords))
    edges = list((e["label"], e.tuple) for e in G.es)

    layout = layout.scale(1)

    scaled_vertices = list(
        map(
            lambda x: (
                x[0],
                (
                    round(x[1][0] * size_f[0], 2),
                    round(x[1][1] * size_f[1], 2),
                ),
            ),
            vertices,
        )
    )

    print("\\xy")
    print("<0.08cm, _0cm>:")

    print("%Vertices%")
    for i, vertex in enumerate(scaled_vertices):
        print(
            str(vertex[1])
            + node_name(vertex[0])
            + f'="{(i+1)}";_%{vertex[0]} '
        )

    print("%Edges%")
    for _, edge in edges:

```

```

    print(
        f""{edge[0]+1}";" {edge[1]+1}" ' + "⊥**\ dir{-};"
    )

def get_label(name):
    if "root" in name or "leaf" in name:
        return None
    name = name.strip("B").strip("W")
    return (
        f"({name.split('-')[0]}, "
        + "\\text{⊥}"
        + f"{name.split('-')[1]})"
        if "-" in name
        else name
    )

if labels:
    print("%Labels%")
    for i, vertex in enumerate(scaled_vertices):
        pos = (vertex[1][0] + label_b[0], vertex[1][1])
        if name := get_label(vertex[0]):
            print(
                str(pos)
                + "*="0{\\scriptstyle ⊥}"
                + name
                + "};")
    )

for name, (s, t) in edges:
    label = get_label(name)
    s_pos = scaled_vertices[s][1]
    t_pos = scaled_vertices[t][1]
    pos = (
        round(
            (s_pos[0] + t_pos[0]) / 2
            + label_b[1][0],
            2,
        ),
        round(
            (s_pos[1] + t_pos[1]) / 2
            + label_b[1][1],
            2,
        ),
    )
    print(
        str(pos)
        + "*="0{\\scriptstyle ⊥}"
        + label
        + "};")

```

```

    )
    if tree_name:
        print("(-13,0)*{" + tree_name + "}");")
    print("\\endxy")

def has_node(graph, name):
    try:
        graph.vs.find(name=name)
    except:
        return False
    return True

def _recursive_add_edges(T, G):
    if T.root:
        s_n = f"root_{T.trunk}"
        t_n = (
            f"{T.node}-{T.trunk}"
            if T.node
            else f"leaf_{T.trunk}"
        )
        t_label = T.node if T.node else f"leaf_{T.trunk}"
        if not has_node(G, s_n):
            G.add_vertex(s_n, label=s_n)
        if not has_node(G, t_n):
            G.add_vertex(t_n, label=t_label)
        G.add_edge(s_n, t_n, label=T.trunk)

    if T.node:
        s_n = f"{T.node}-{T.trunk}"
        s_label = f"{T.node}"
        if not has_node(G, s_n):
            G.add_vertex(s_n, label=s_label)
        for branch in T.branches.values():
            t_n = (
                f"{branch.node}-{branch.trunk}"
                if branch.node
                else f"leaf_{branch.trunk}_from_{T.node}"
            )
            t_label = (
                branch.node
                if branch.node
                else f"leaf_{branch.trunk}"
            )
            if not has_node(G, t_n):
                G.add_vertex(t_n, label=t_label)
        G.add_edge(s_n, t_n, label=branch.trunk)
        _recursive_add_edges(branch, G)

```

```

def node_name(name):
    if "W" in name:
        return "*\ cir <2pt>{"
    if "B" in name:
        return "*=0{\ \bullet}"
    return "*{"

```

```

def separator(size):
    print("\ \xy")
    print("<0.08cm, _0cm>:")
    print(f"(-{size}" + ', _0.0)*{"="1"; ')
    print(f"({size}" + ', _10.0)*{"="2"; ')
    print("\ endxy")

```