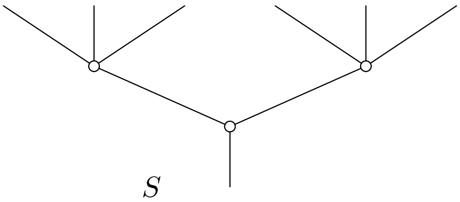
Annexos

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

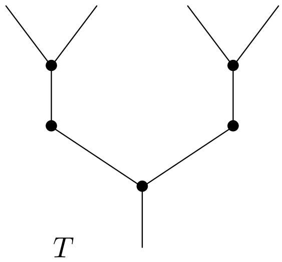
Contents

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

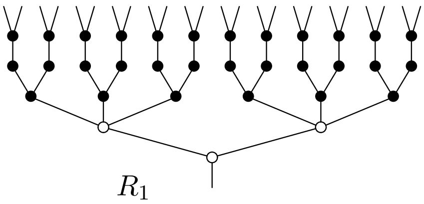


S

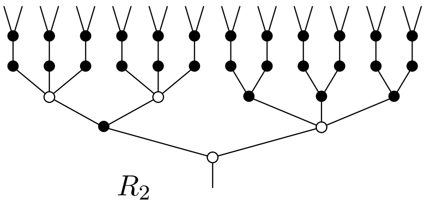


T

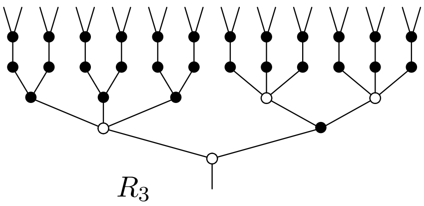
Esto es un ejemplo tedioso de calcular el conjunto de shuffles ya que según la función sh existen 296 shuffles diferentes. Mostramos todos ellos:



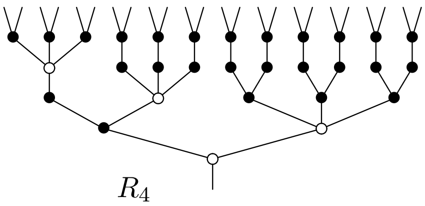
R_1



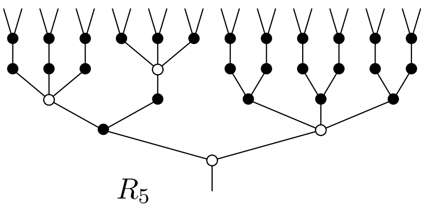
R_2



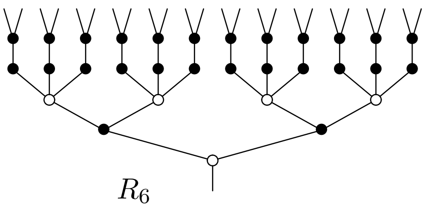
R_3



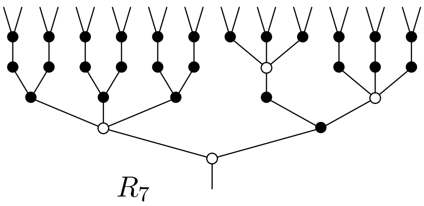
R_4



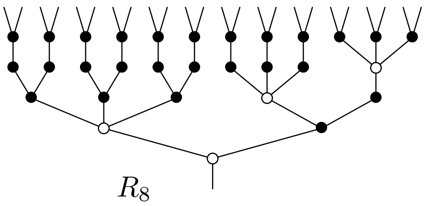
R_5



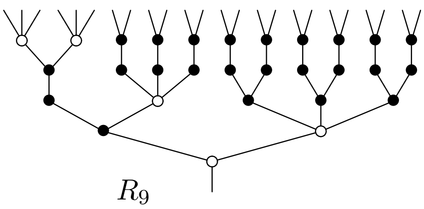
R_6



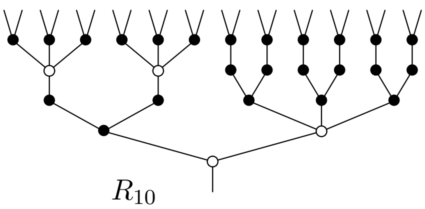
R_7



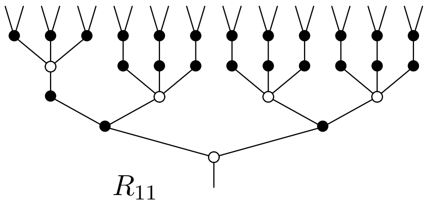
R_8



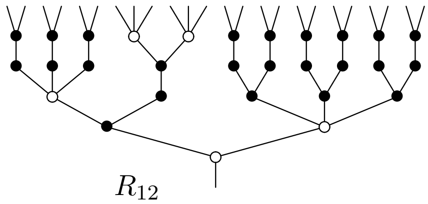
R_9



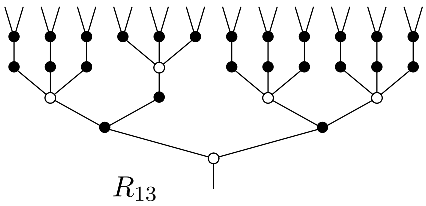
R_{10}



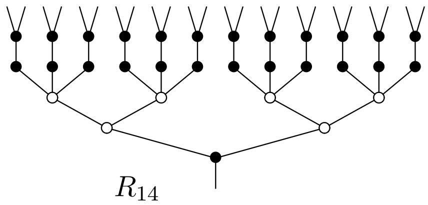
R_{11}



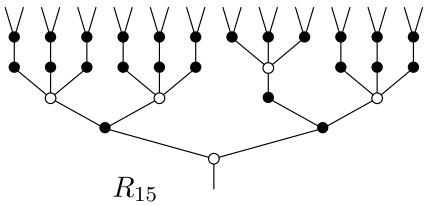
R_{12}



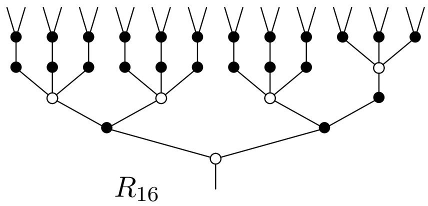
R_{13}



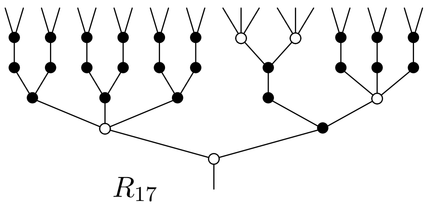
R_{14}



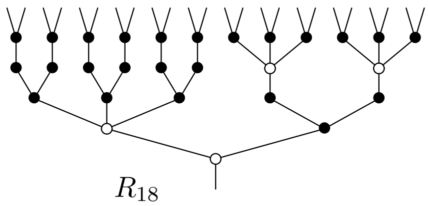
R_{15}



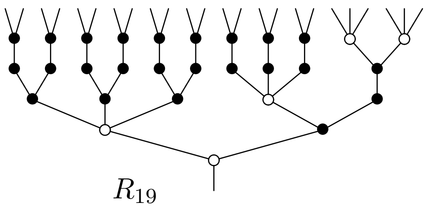
R_{16}



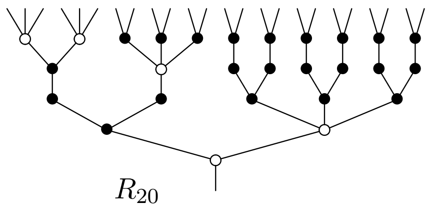
R_{17}



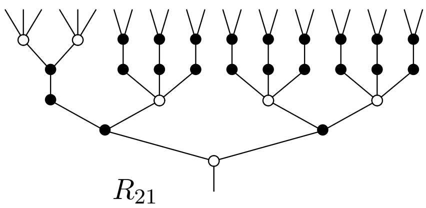
R_{18}



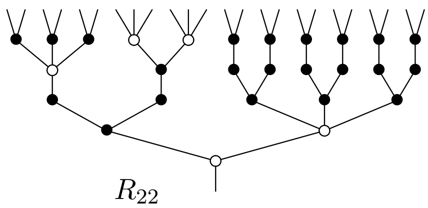
R_{19}



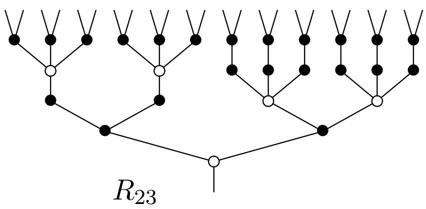
R_{20}



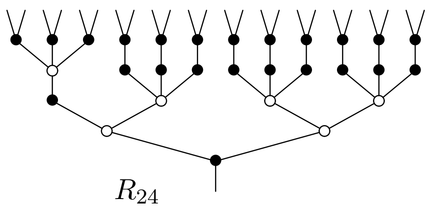
R_{21}



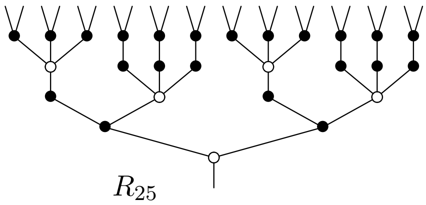
R_{22}



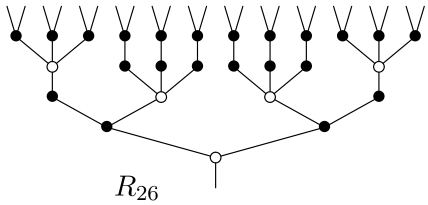
R_{23}



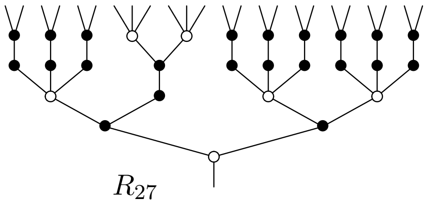
R_{24}



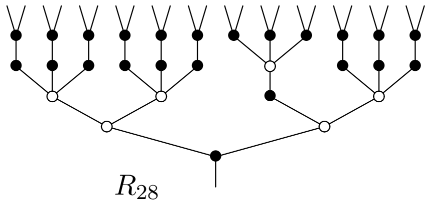
R_{25}



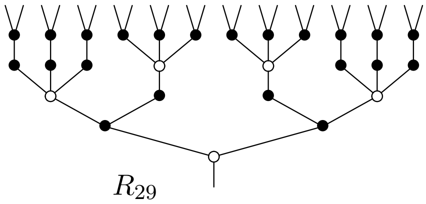
R_{26}



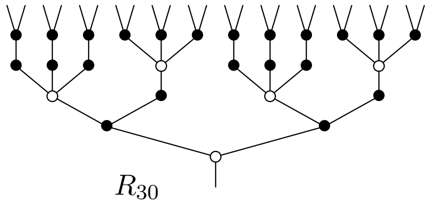
R_{27}



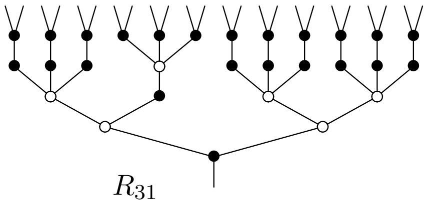
R_{28}



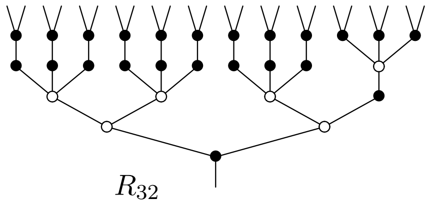
R_{29}



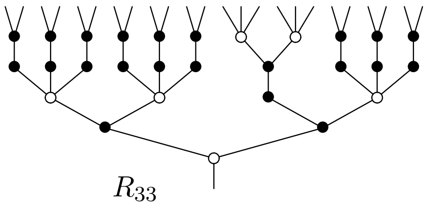
R_{30}



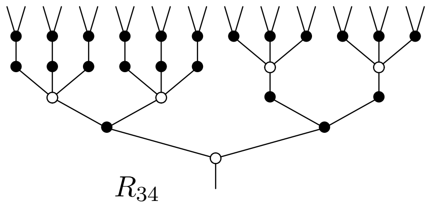
R_{31}



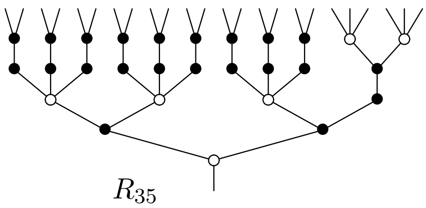
R_{32}



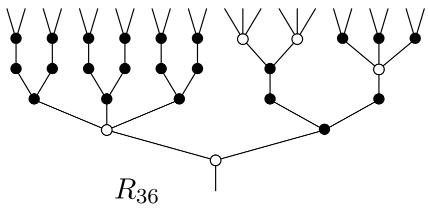
R_{33}



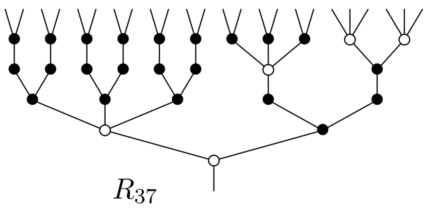
R_{34}



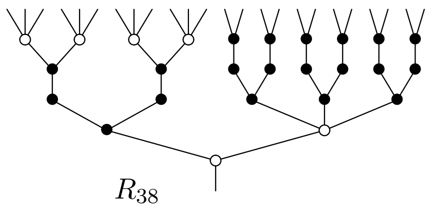
R_{35}



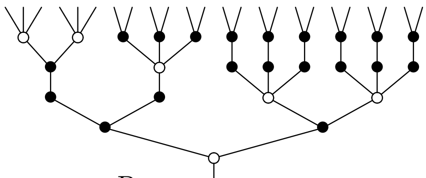
R_{36}



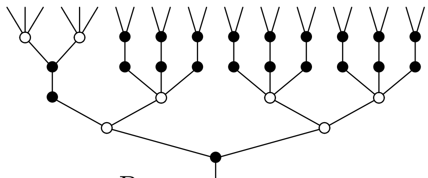
R_{37}



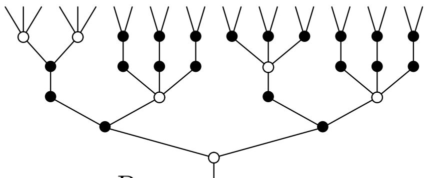
R_{38}



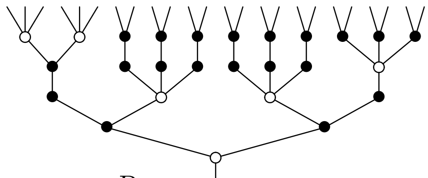
R_{39}



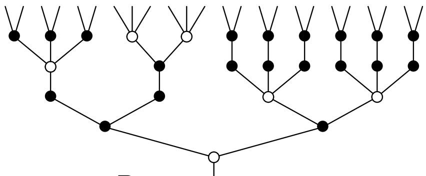
R_{40}



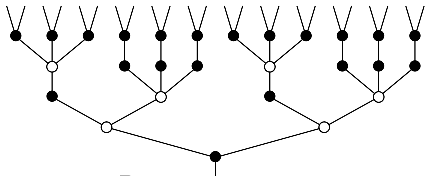
R_{41}



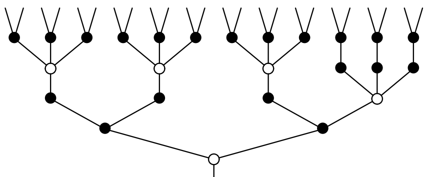
R_{42}



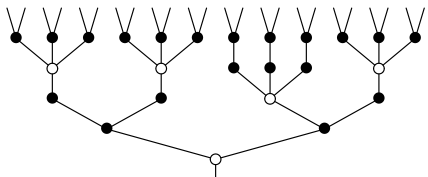
R_{43}



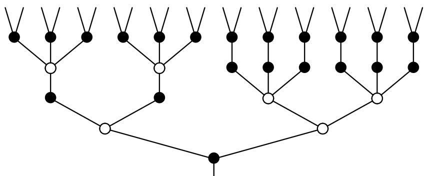
R_{44}



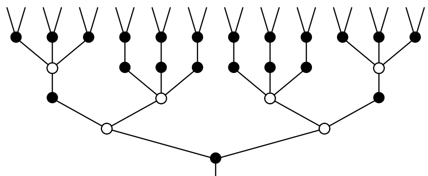
R_{45}



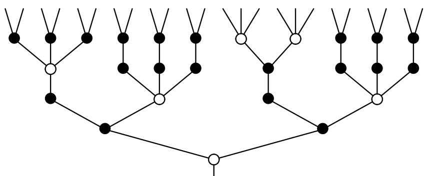
R_{46}



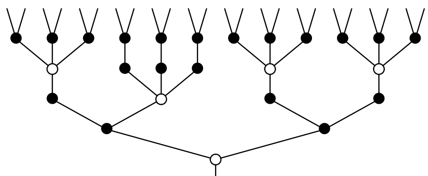
R_{47}



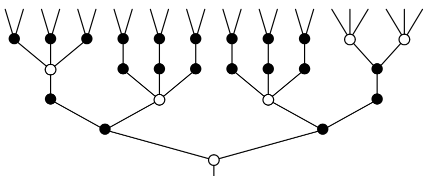
R_{48}



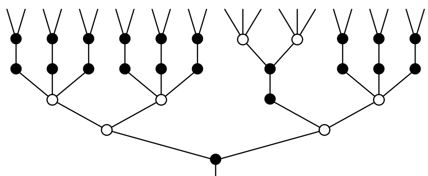
R_{49}



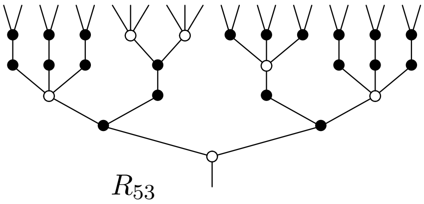
R_{50}



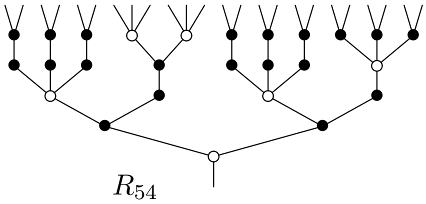
R_{51}



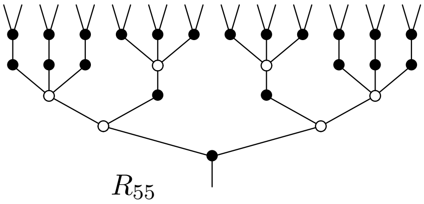
R_{52}



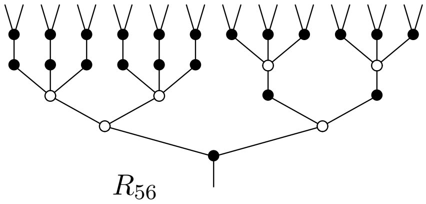
R_{53}



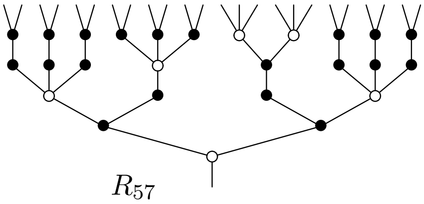
R_{54}



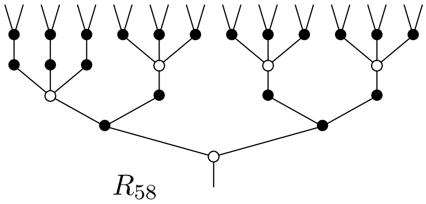
R_{55}



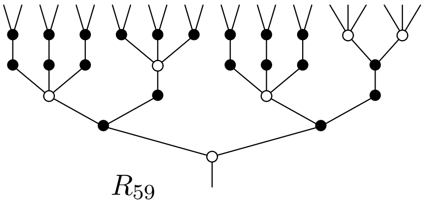
R_{56}



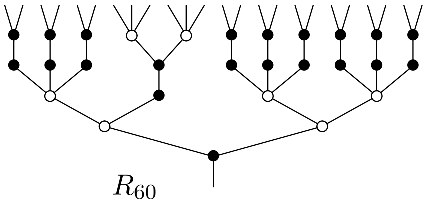
R_{57}



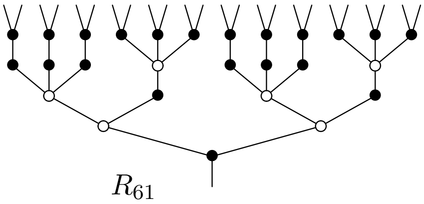
R_{58}



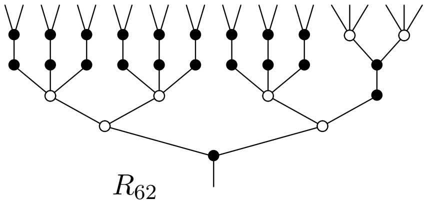
R_{59}



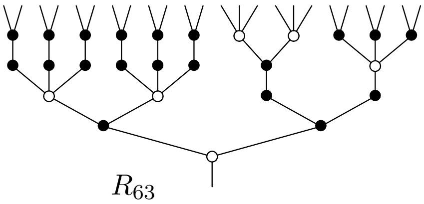
R_{60}



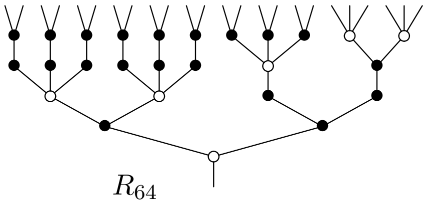
R_{61}



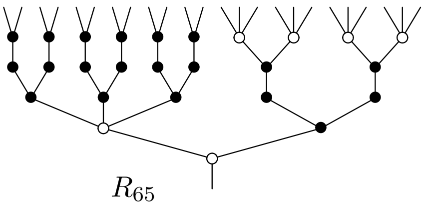
R_{62}



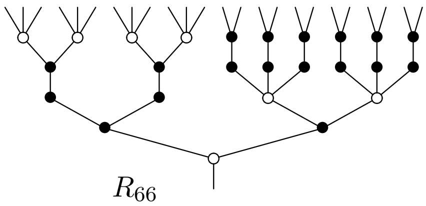
R_{63}



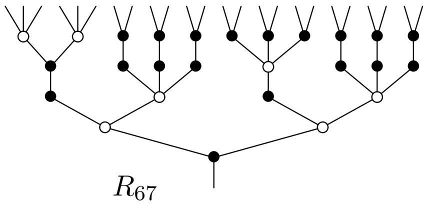
R_{64}



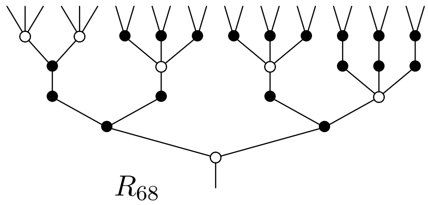
R_{65}



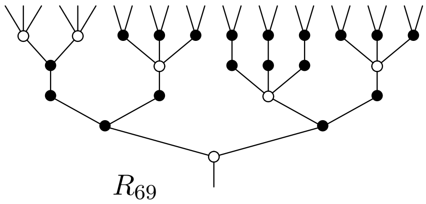
R_{66}



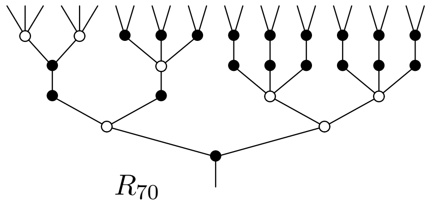
R_{67}



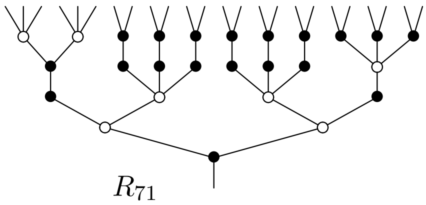
R_{68}



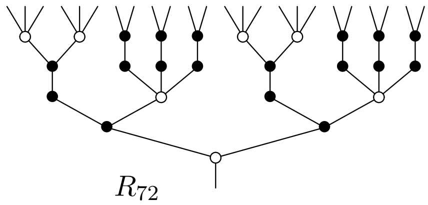
R_{69}



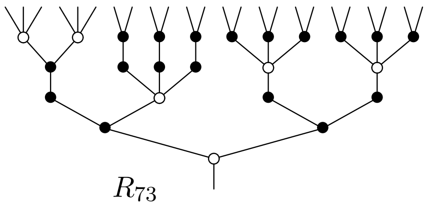
R_{70}



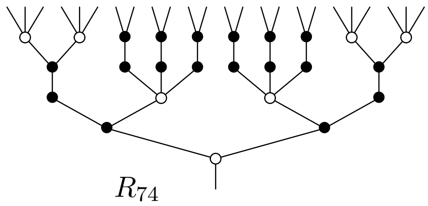
R_{71}



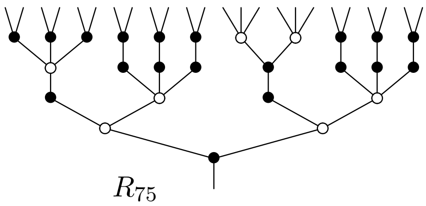
R_{72}



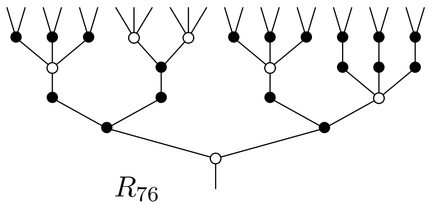
R_{73}



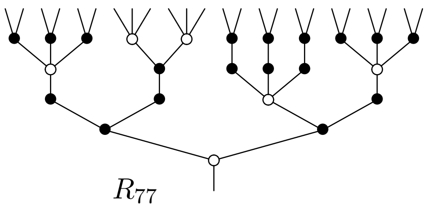
R_{74}



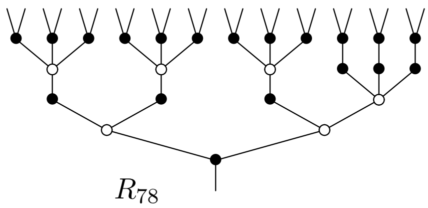
R_{75}



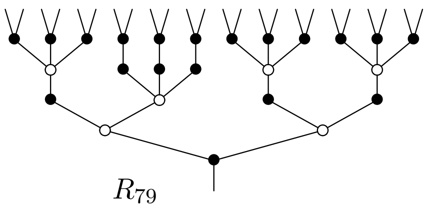
R_{76}



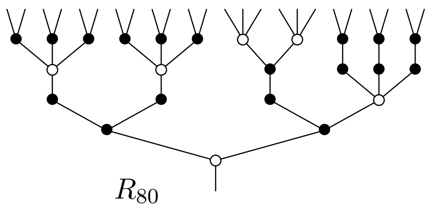
R_{77}



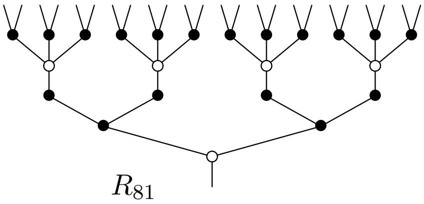
R_{78}



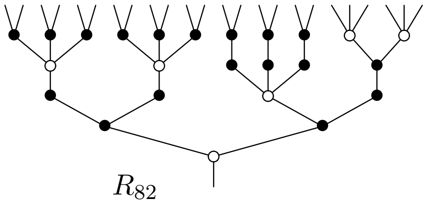
R_{79}



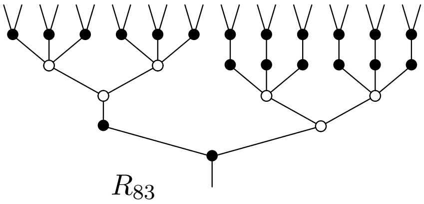
R_{80}



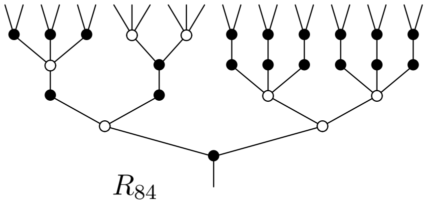
R_{81}



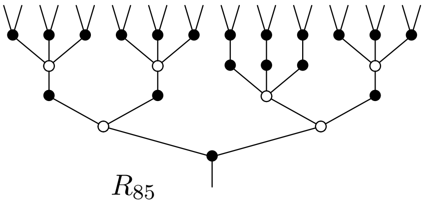
R_{82}



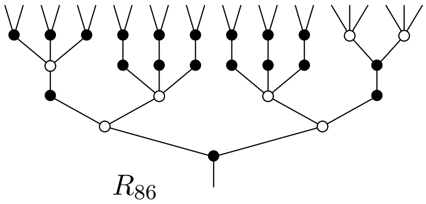
R_{83}



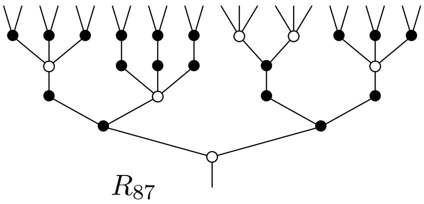
R_{84}



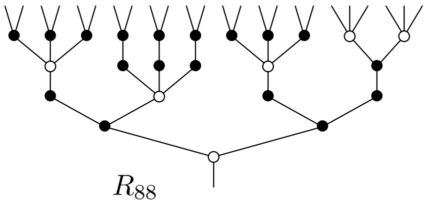
R_{85}



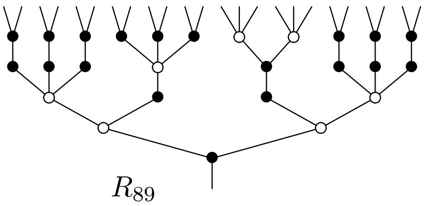
R_{86}



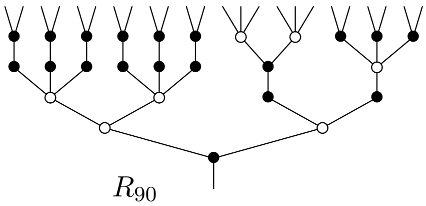
R_{87}



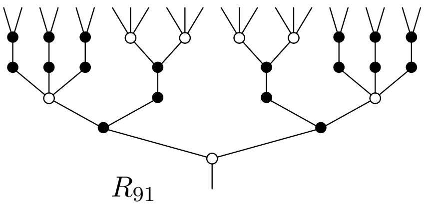
R_{88}



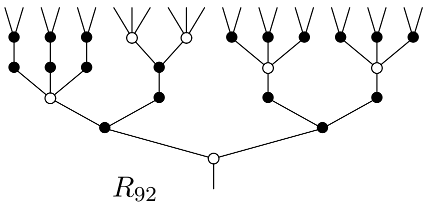
R_{89}



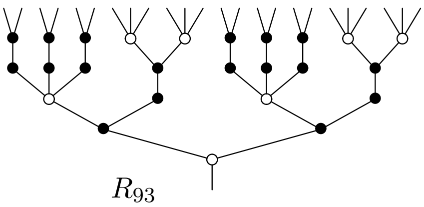
R_{90}



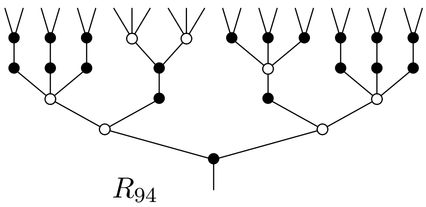
R_{91}



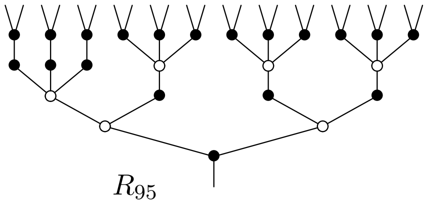
R_{92}



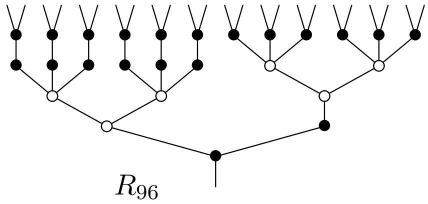
R_{93}



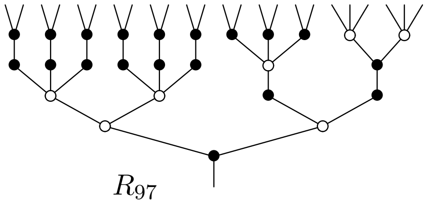
R_{94}



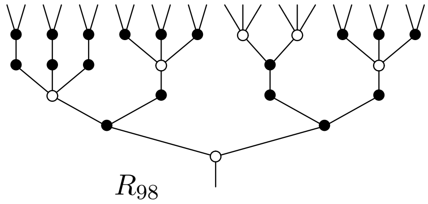
R_{95}



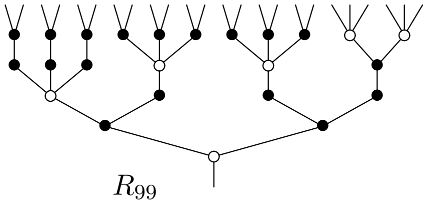
R_{96}



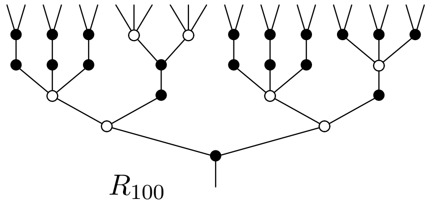
R_{97}



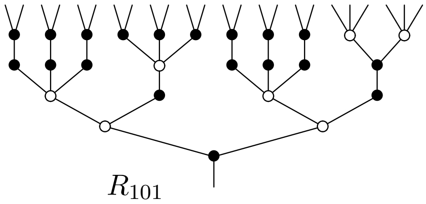
R_{98}



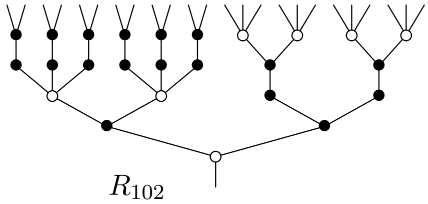
R_{99}



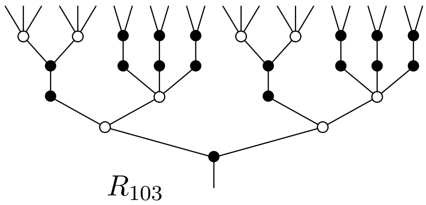
R_{100}



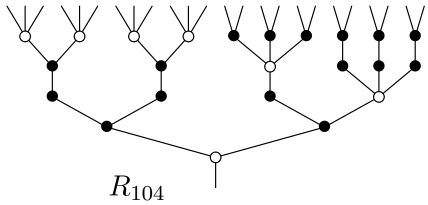
R_{101}



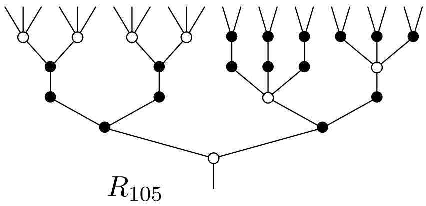
R_{102}



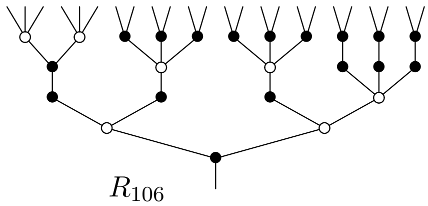
R_{103}



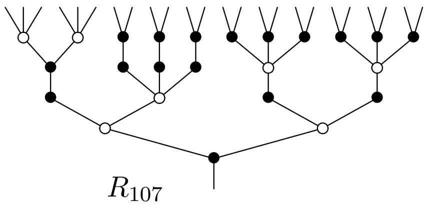
R_{104}



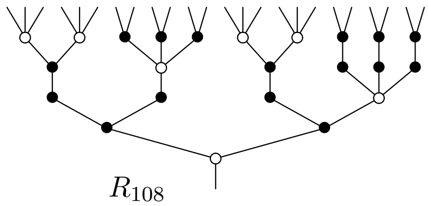
R_{105}



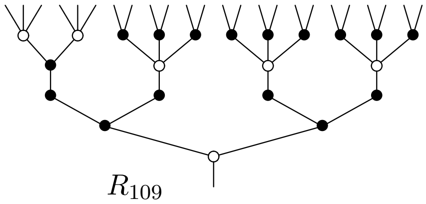
R_{106}



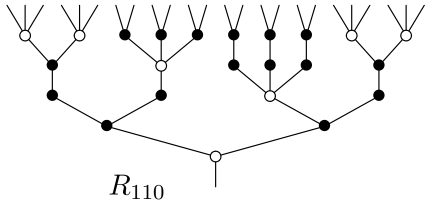
R_{107}



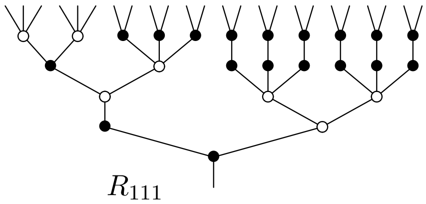
R_{108}



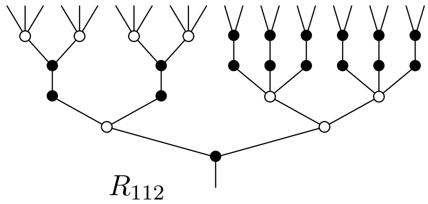
R_{109}



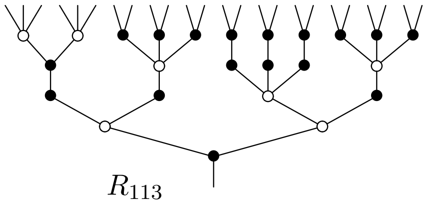
R_{110}



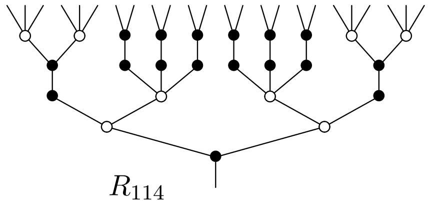
R_{111}



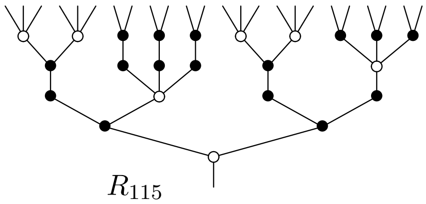
R_{112}



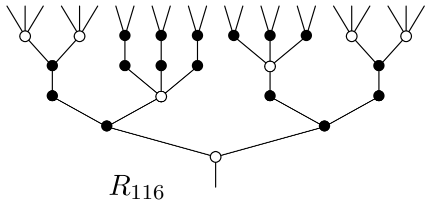
R_{113}



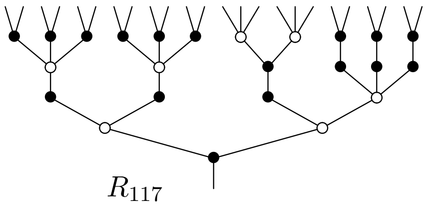
R_{114}



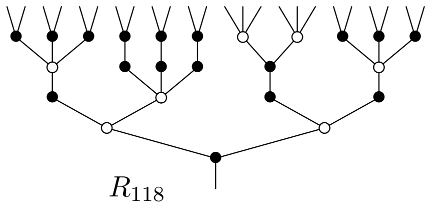
R_{115}



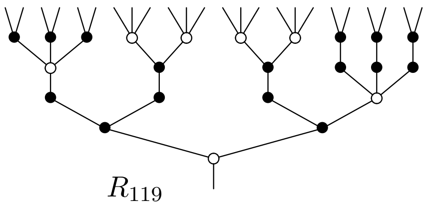
R_{116}



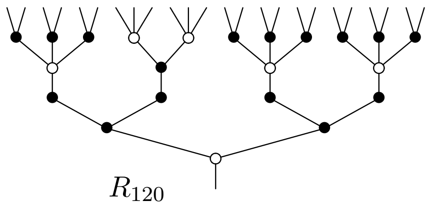
R_{117}



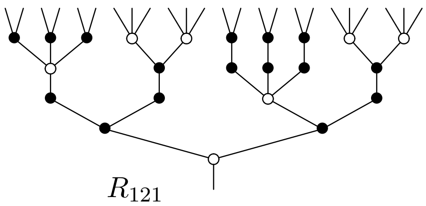
R_{118}



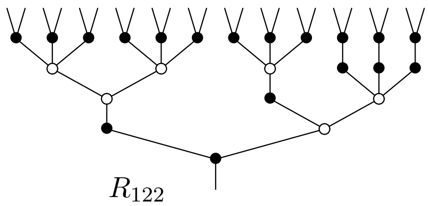
R_{119}



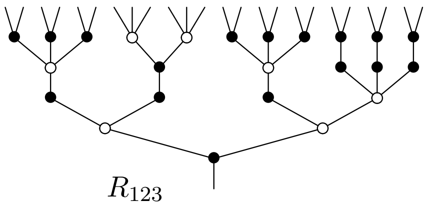
R_{120}



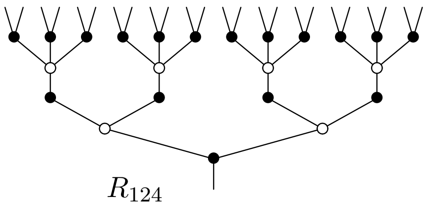
R_{121}



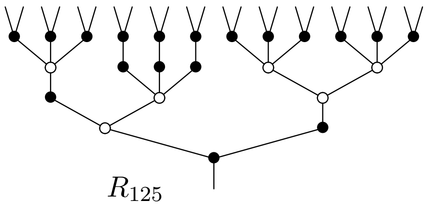
R_{122}



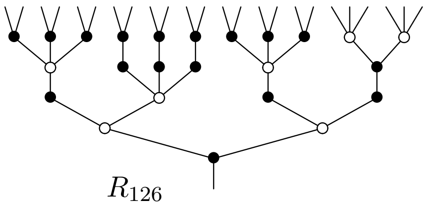
R_{123}



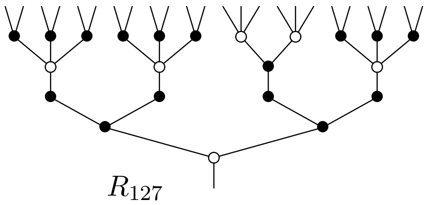
R_{124}



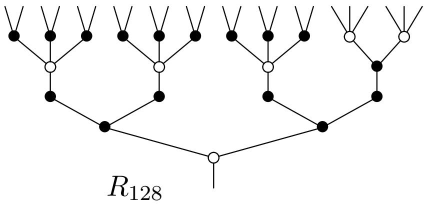
R_{125}



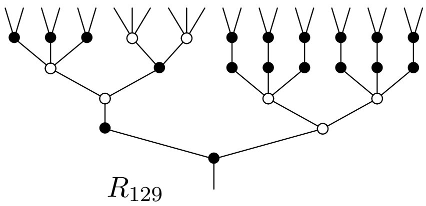
R_{126}



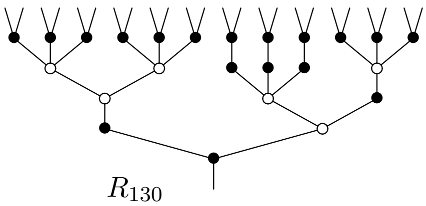
R_{127}



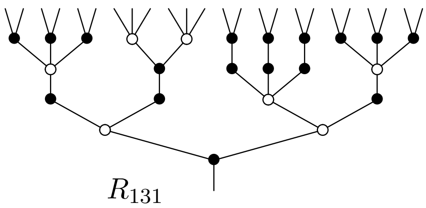
R_{128}



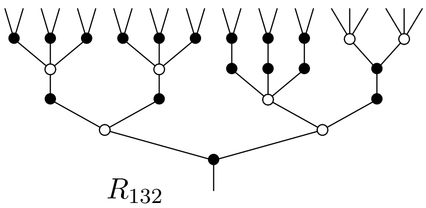
R_{129}



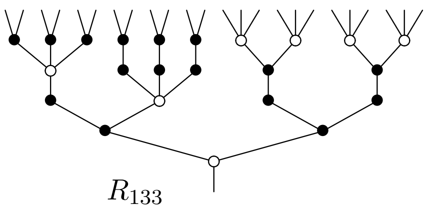
R_{130}



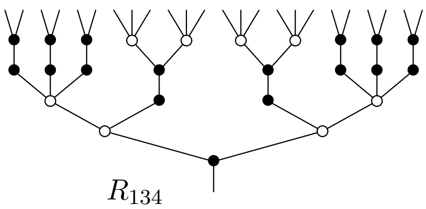
R_{131}



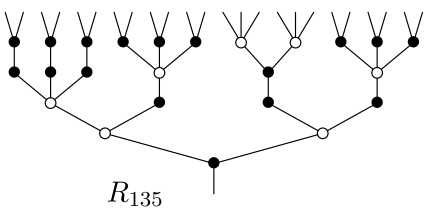
R_{132}



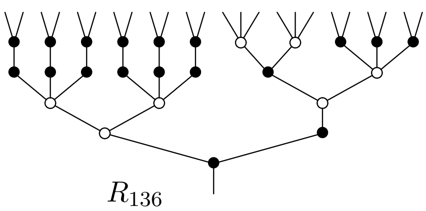
R_{133}



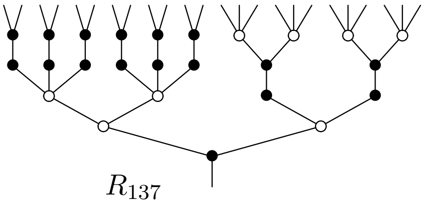
R_{134}



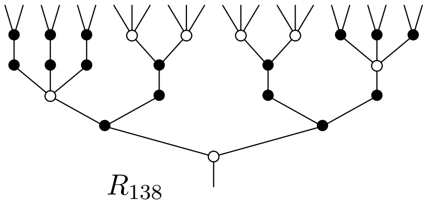
R_{135}



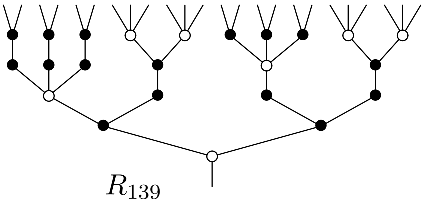
R_{136}



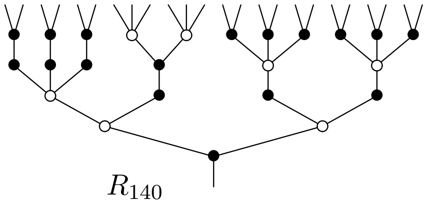
R_{137}



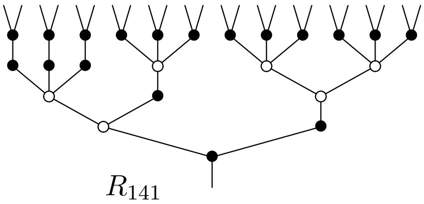
R_{138}



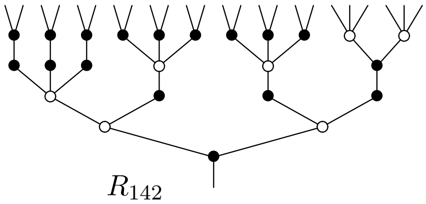
R_{139}



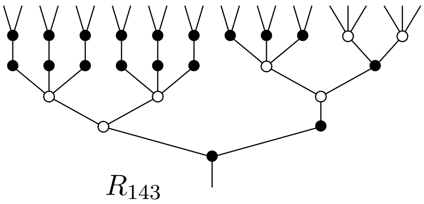
R_{140}



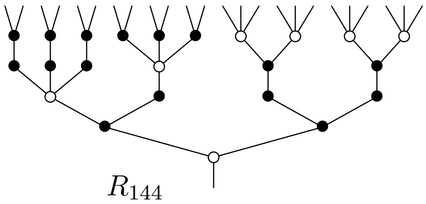
R_{141}



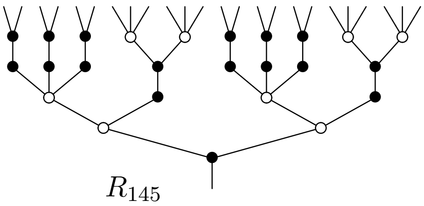
R_{142}



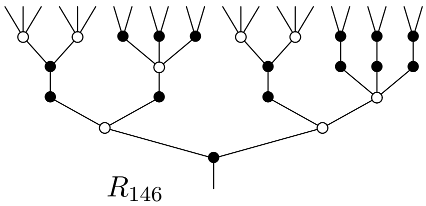
R_{143}



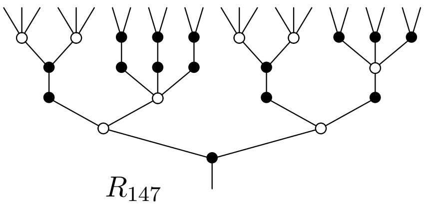
R_{144}



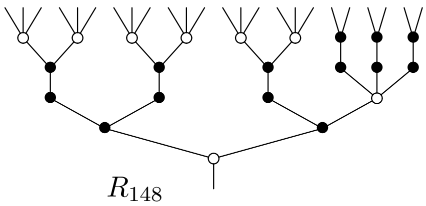
R_{145}



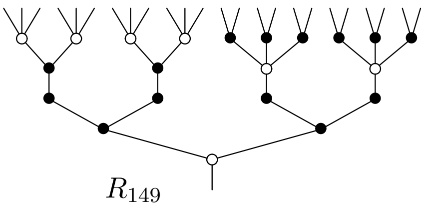
R_{146}



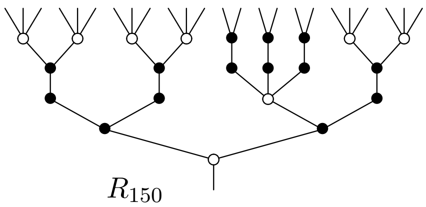
R_{147}



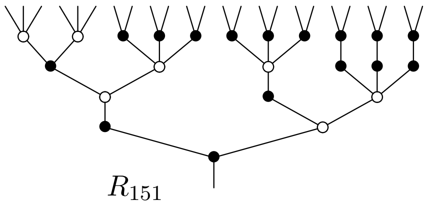
R_{148}



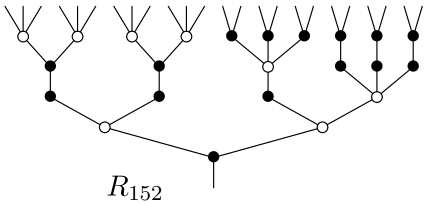
R_{149}



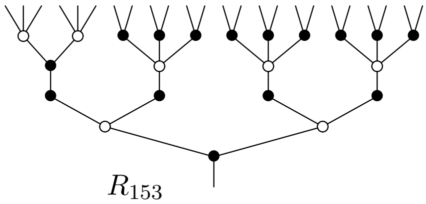
R_{150}



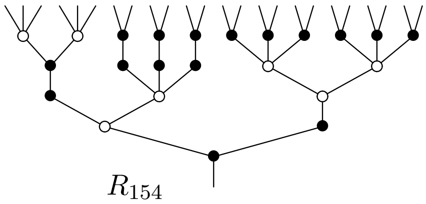
R_{151}



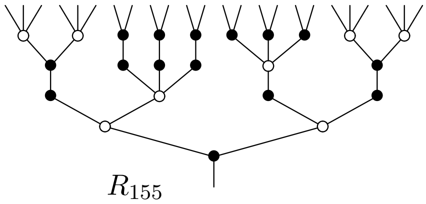
R_{152}



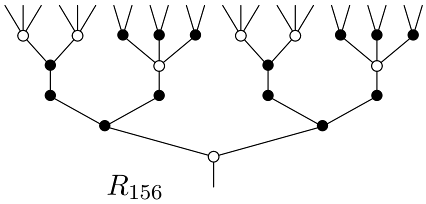
R_{153}



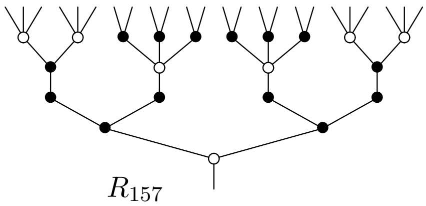
R_{154}



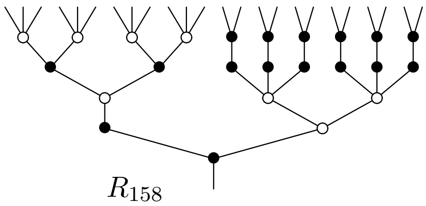
R_{155}



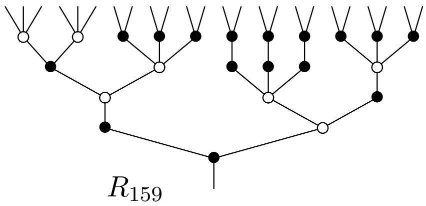
R_{156}



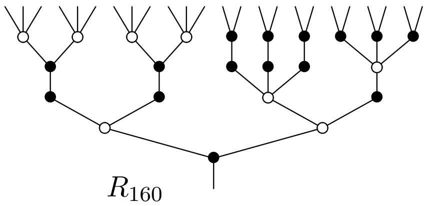
R_{157}



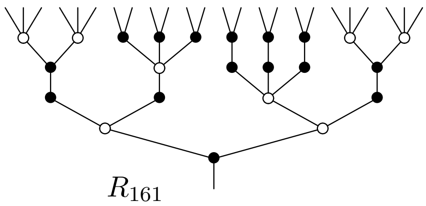
R_{158}



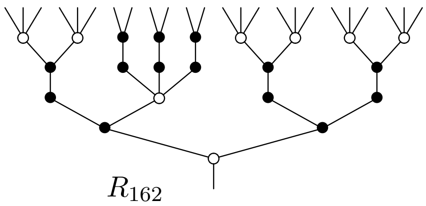
R_{159}



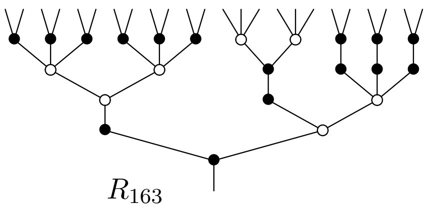
R_{160}



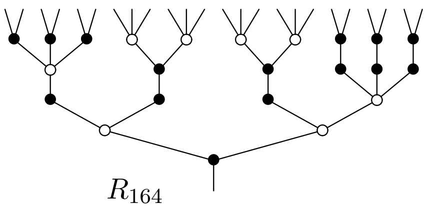
R_{161}



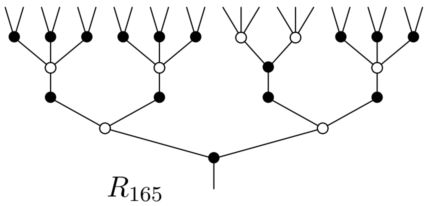
R_{162}



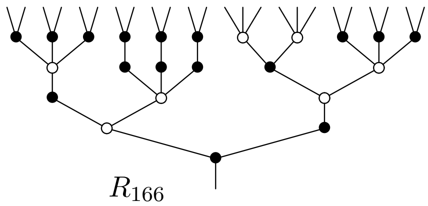
R_{163}



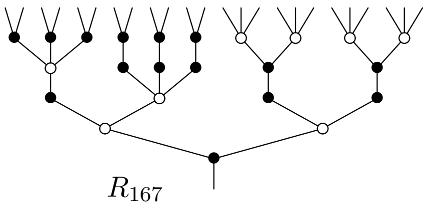
R_{164}



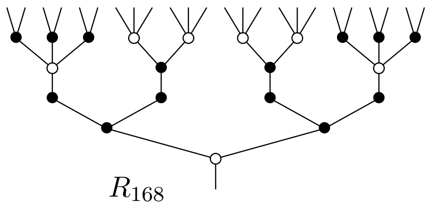
R_{165}



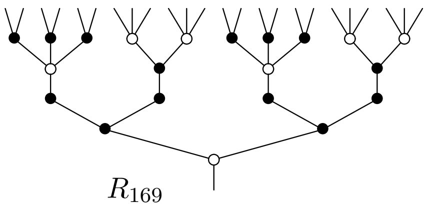
R_{166}



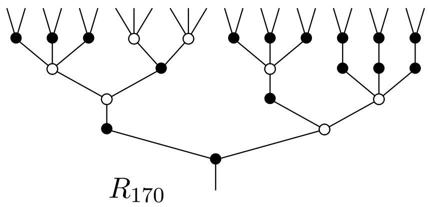
R_{167}



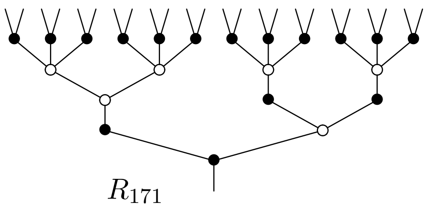
R_{168}



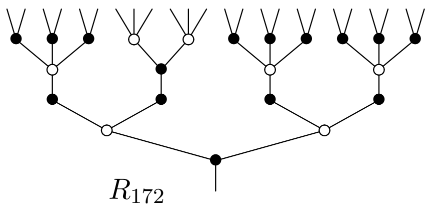
R_{169}



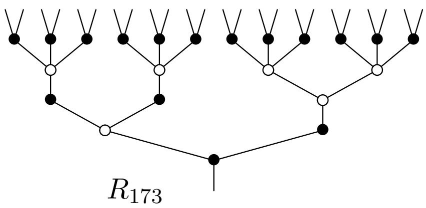
R_{170}



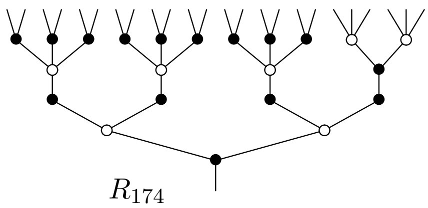
R_{171}



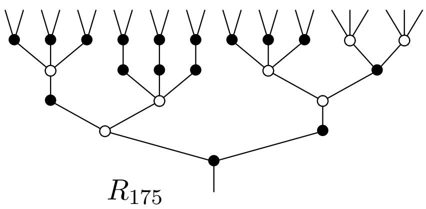
R_{172}



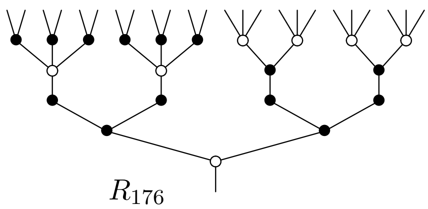
R_{173}



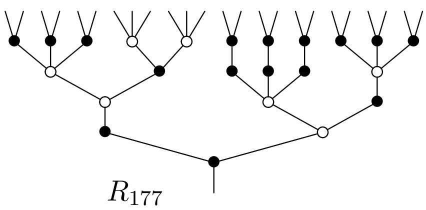
R_{174}



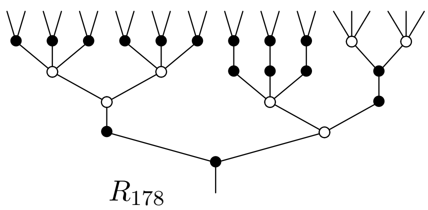
R_{175}



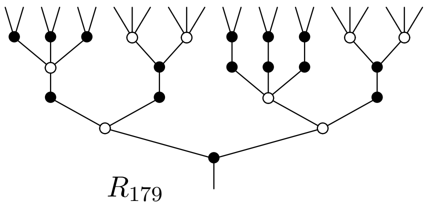
R_{176}



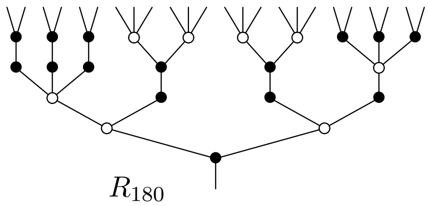
R_{177}



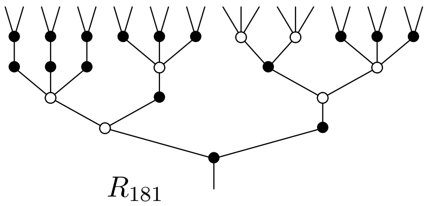
R_{178}



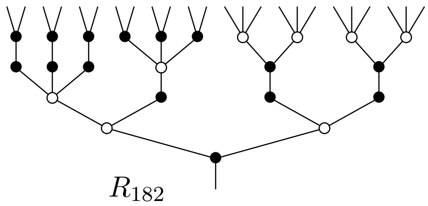
R_{179}



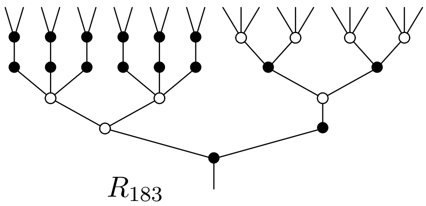
R_{180}



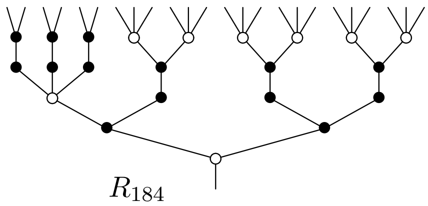
R_{181}



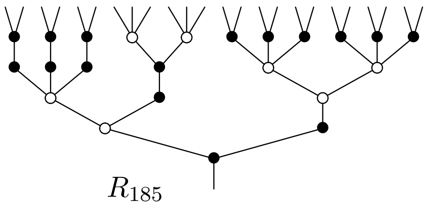
R_{182}



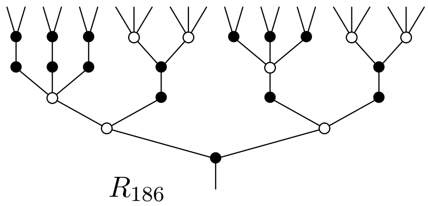
R_{183}



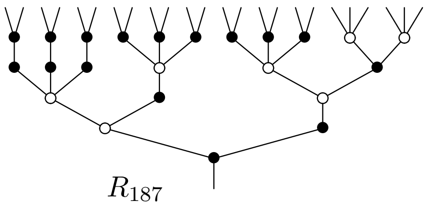
R_{184}



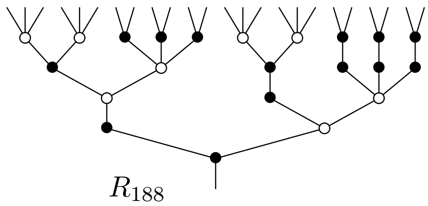
R_{185}



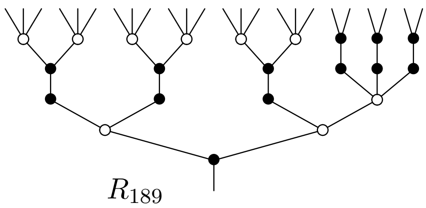
R_{186}



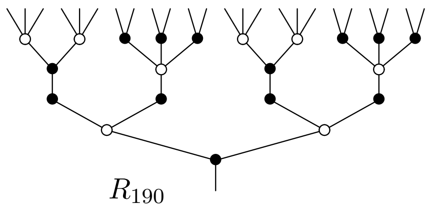
R_{187}



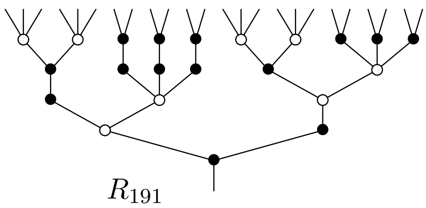
R_{188}



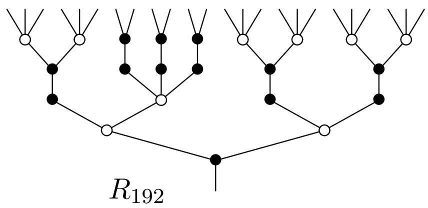
R_{189}



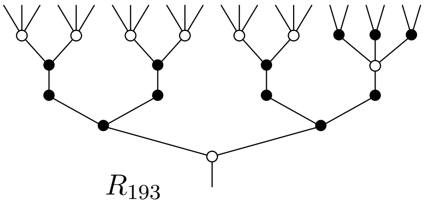
R_{190}



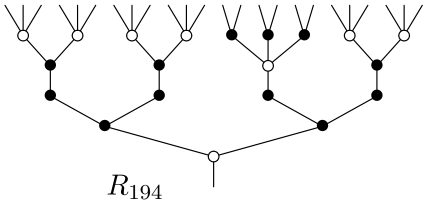
R_{191}



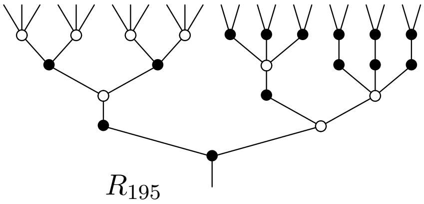
R_{192}



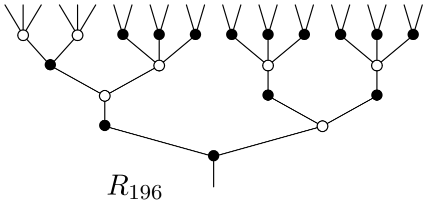
R_{193}



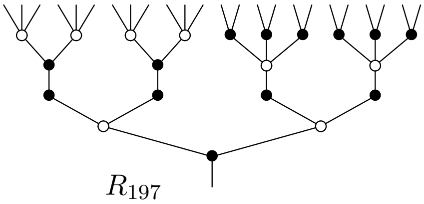
R_{194}



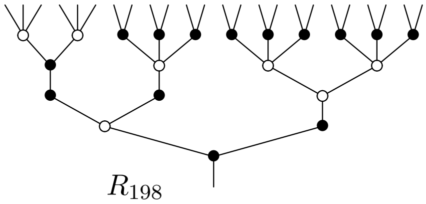
R_{195}



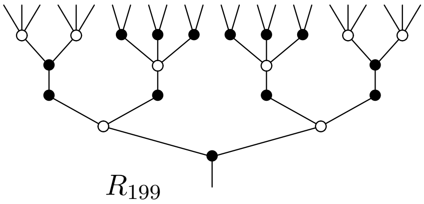
R_{196}



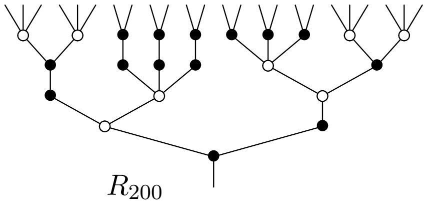
R_{197}



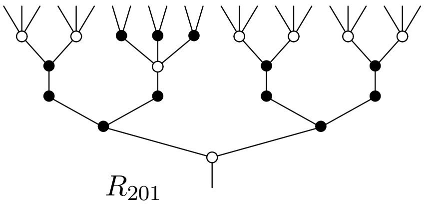
R_{198}



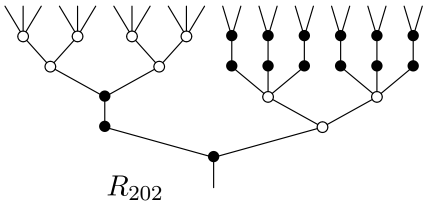
R_{199}



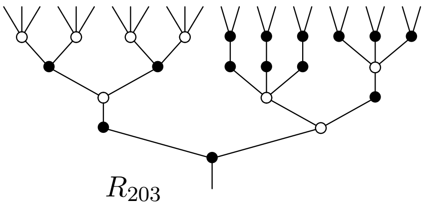
R_{200}



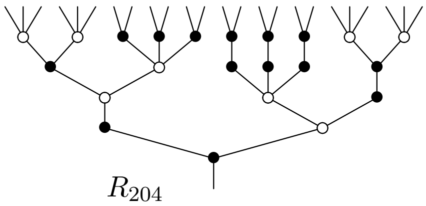
R_{201}



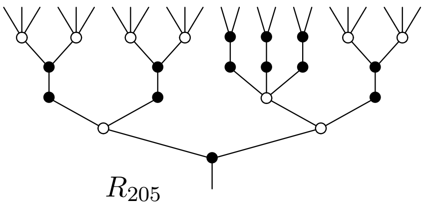
R_{202}



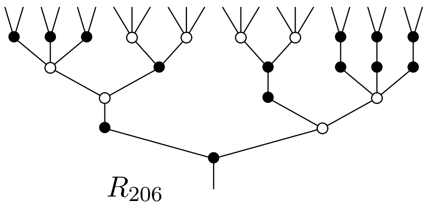
R_{203}



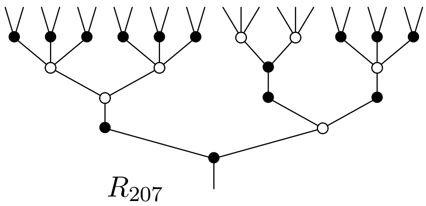
R_{204}



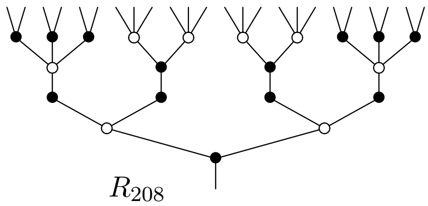
R_{205}



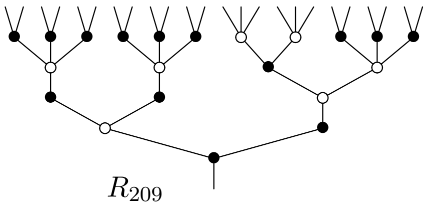
R_{206}



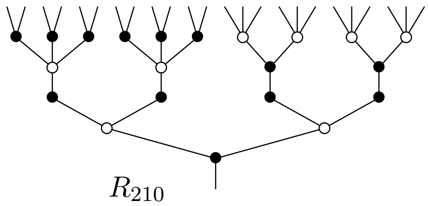
R_{207}



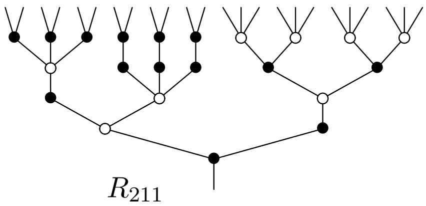
R_{208}



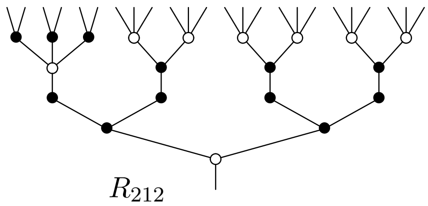
R_{209}



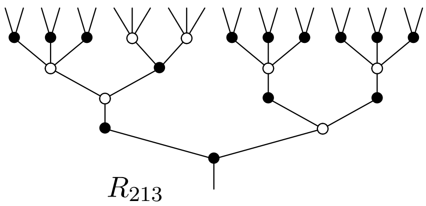
R_{210}



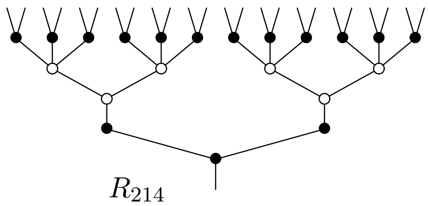
R_{211}



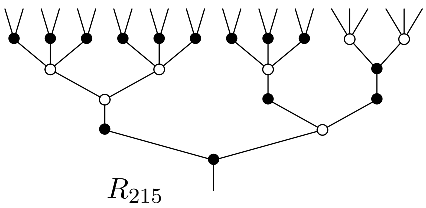
R_{212}



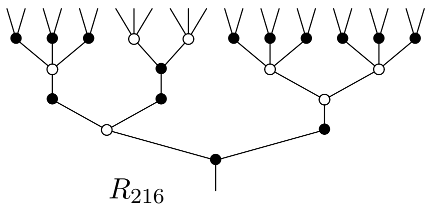
R_{213}



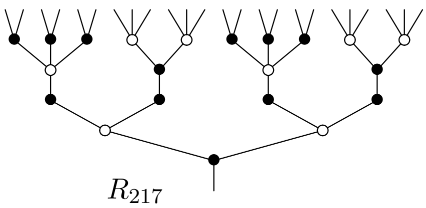
R_{214}



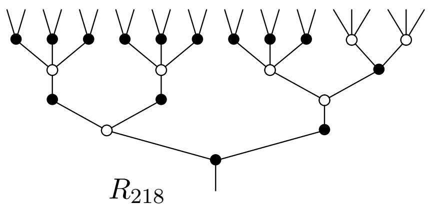
R_{215}



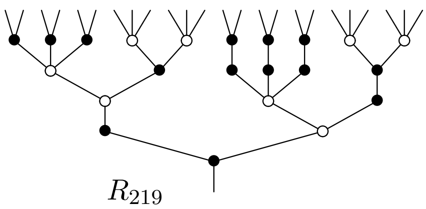
R_{216}



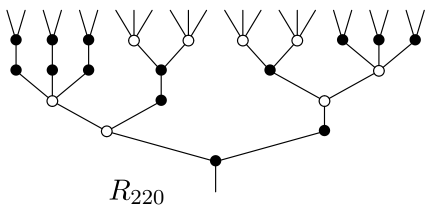
R_{217}



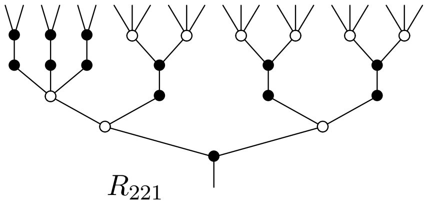
R_{218}



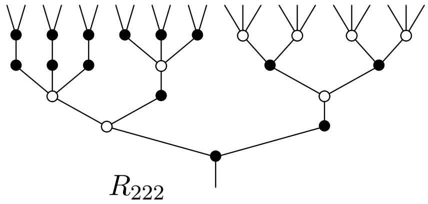
R_{219}



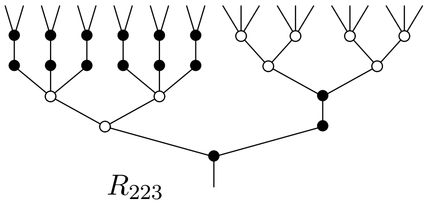
R_{220}



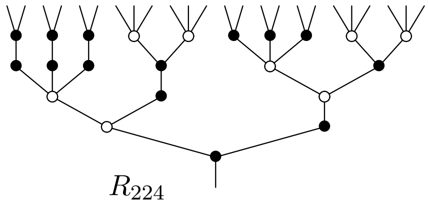
R_{221}



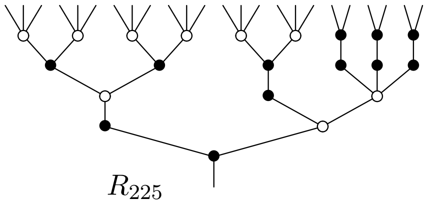
R_{222}



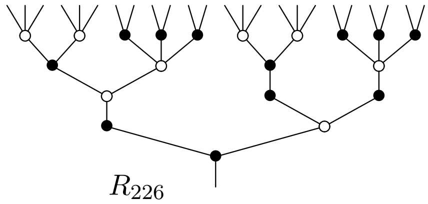
R_{223}



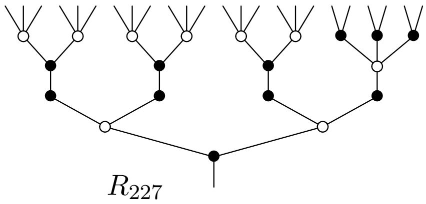
R_{224}



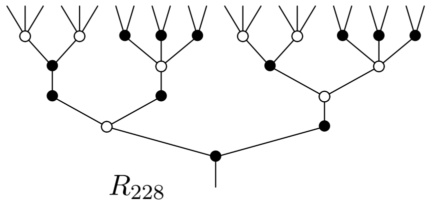
R_{225}



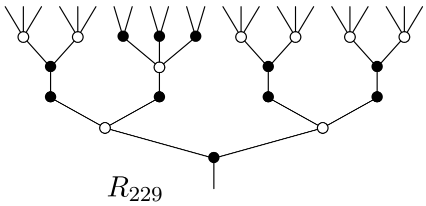
R_{226}



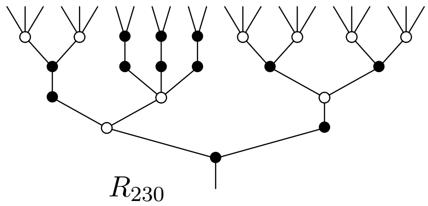
R_{227}



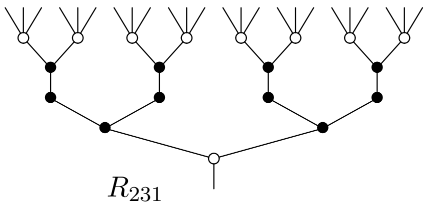
R_{228}



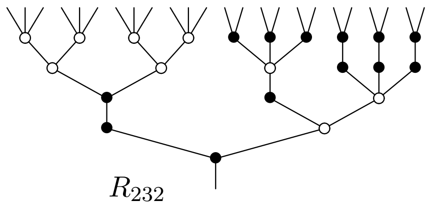
R_{229}



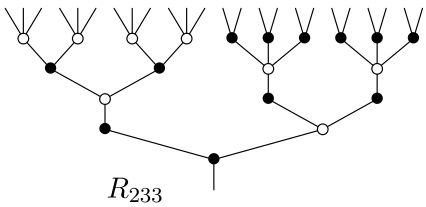
R_{230}



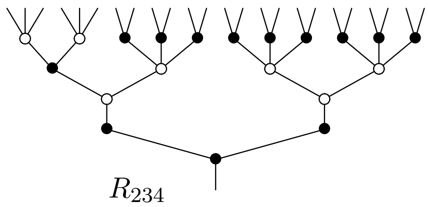
R_{231}



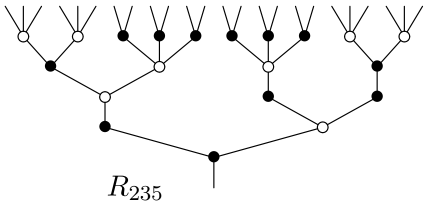
R_{232}



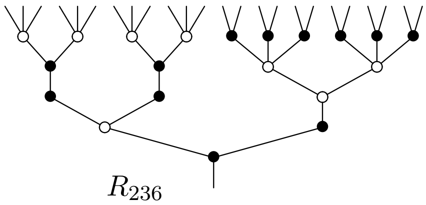
R_{233}



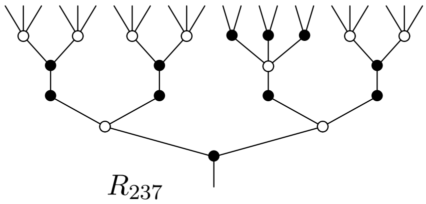
R_{234}



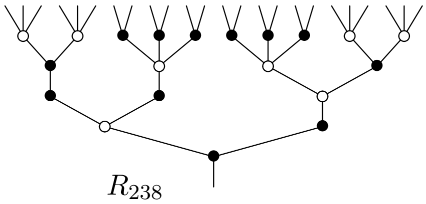
R_{235}



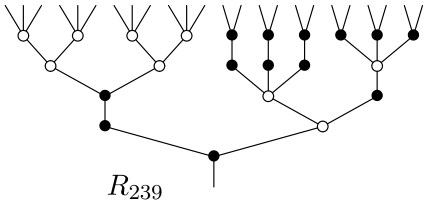
R_{236}



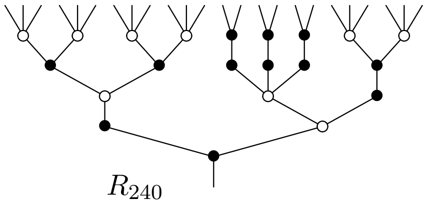
R_{237}



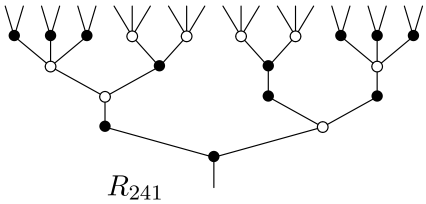
R_{238}



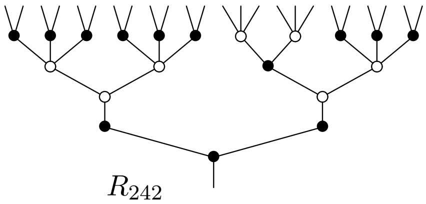
R_{239}



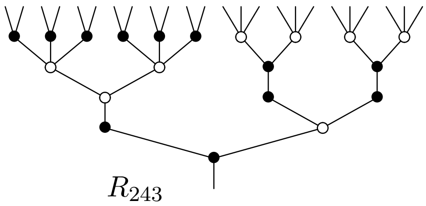
R_{240}



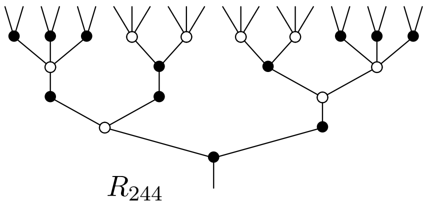
R_{241}



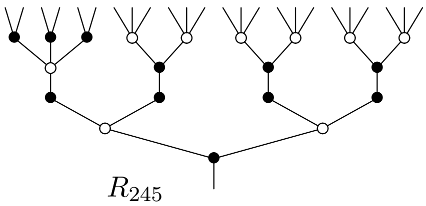
R_{242}



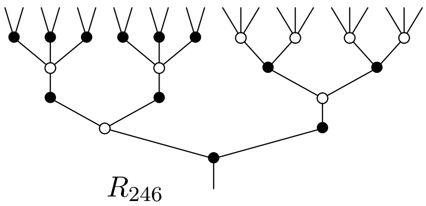
R_{243}



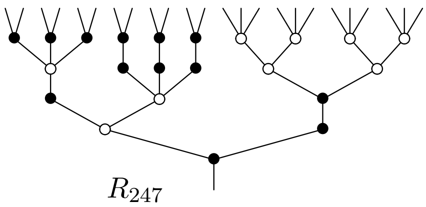
R_{244}



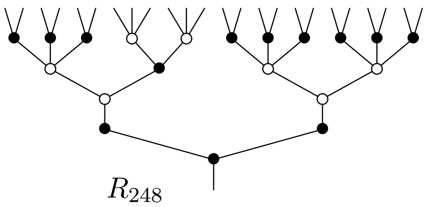
R_{245}



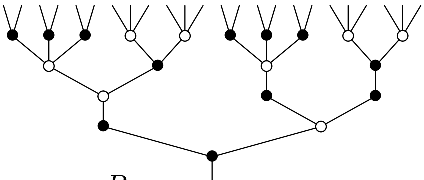
R_{246}



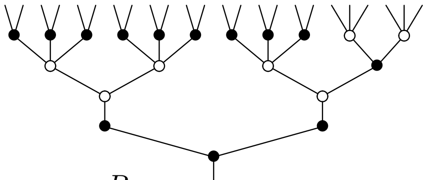
R_{247}



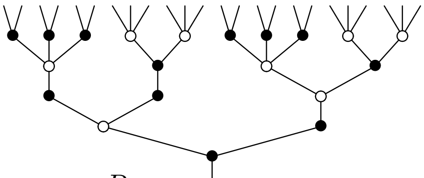
R_{248}



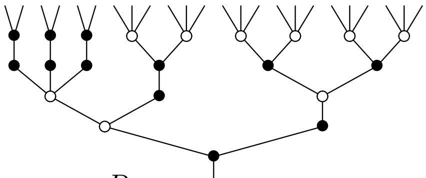
R_{249}



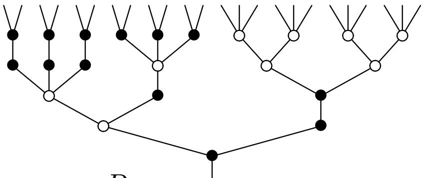
R_{250}



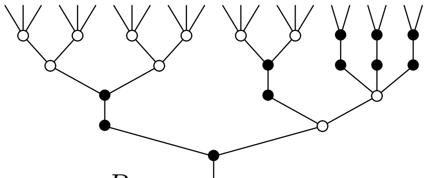
R_{251}



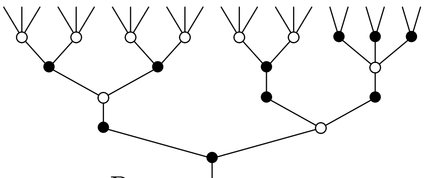
R_{252}



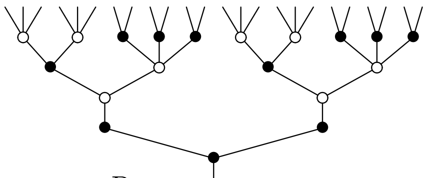
R_{253}



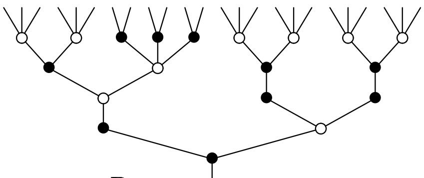
R_{254}



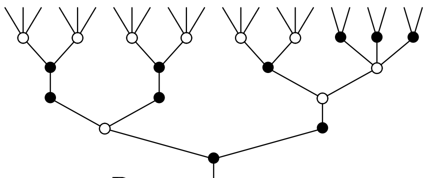
R_{255}



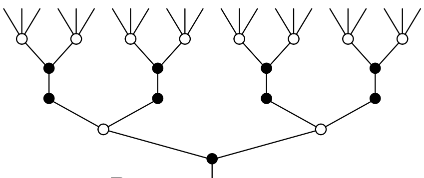
R_{256}



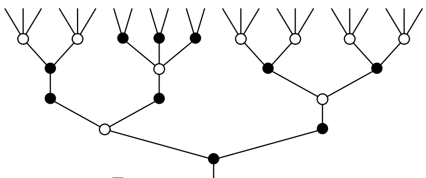
R_{257}



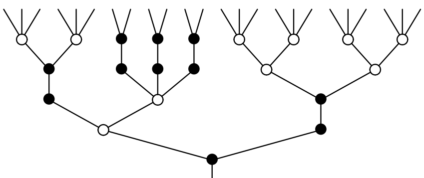
R_{258}



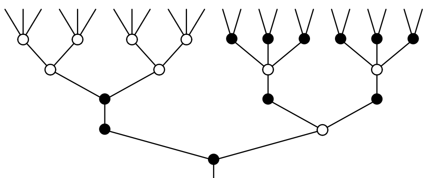
R_{259}



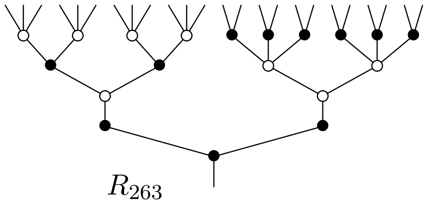
R_{260}



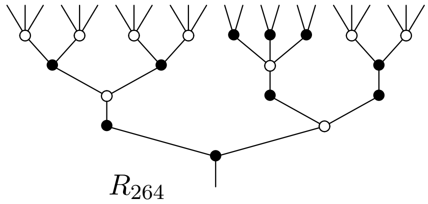
R_{261}



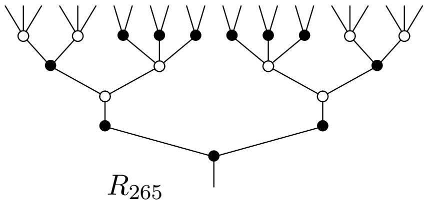
R_{262}



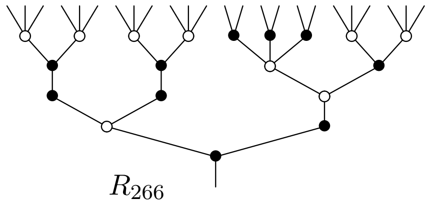
R_{263}



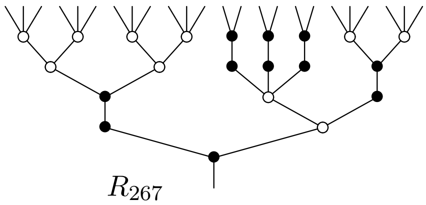
R_{264}



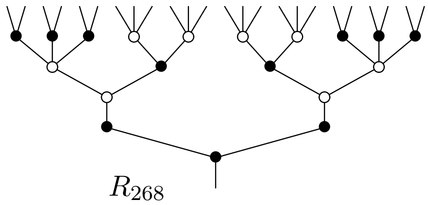
R_{265}



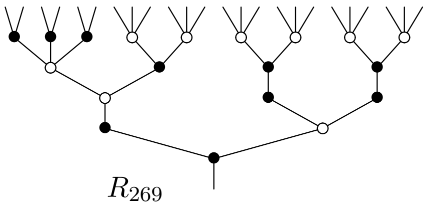
R_{266}



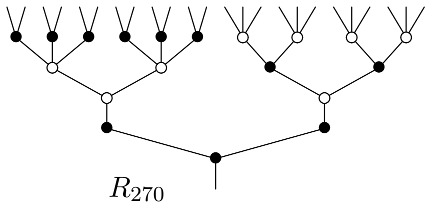
R_{267}



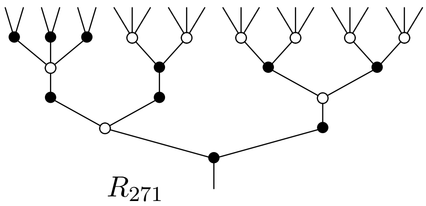
R_{268}



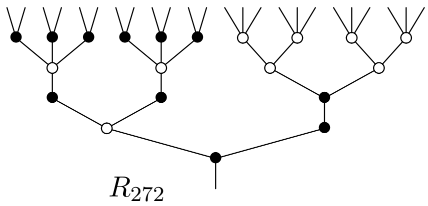
R_{269}



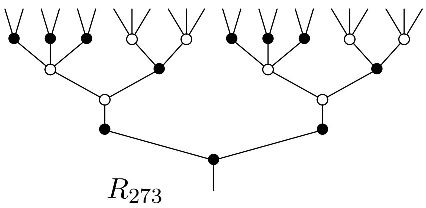
R_{270}



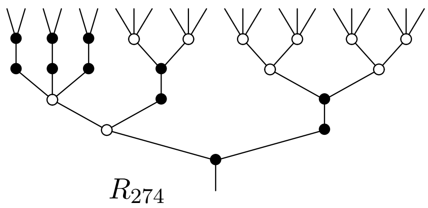
R_{271}



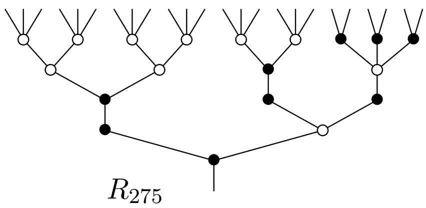
R_{272}



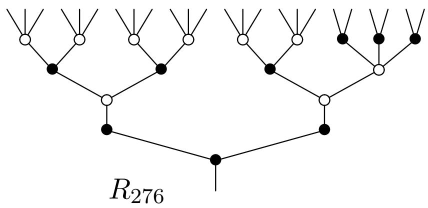
R_{273}



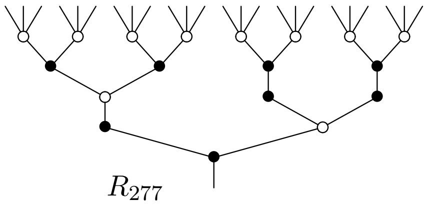
R_{274}



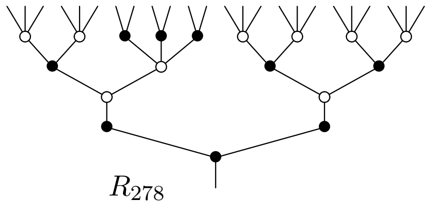
R_{275}



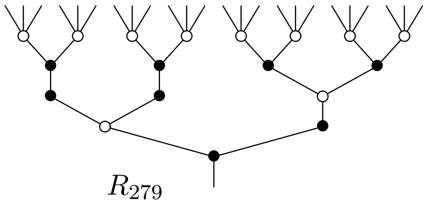
R_{276}



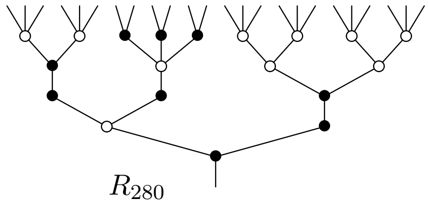
R_{277}



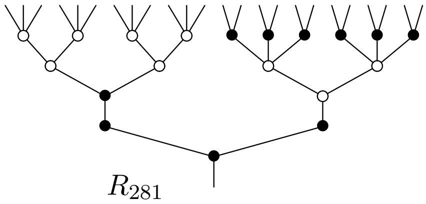
R_{278}



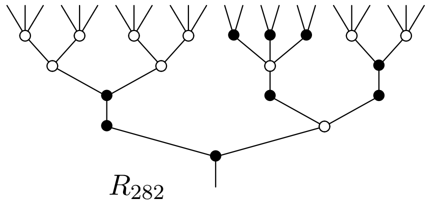
R_{279}



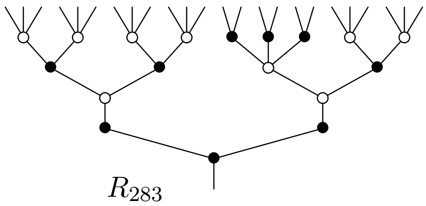
R_{280}



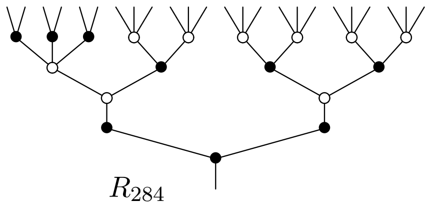
R_{281}



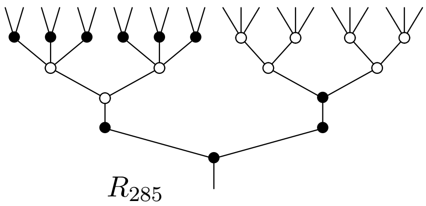
R_{282}



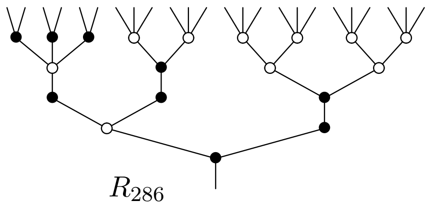
R_{283}



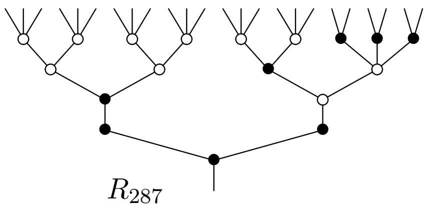
R_{284}



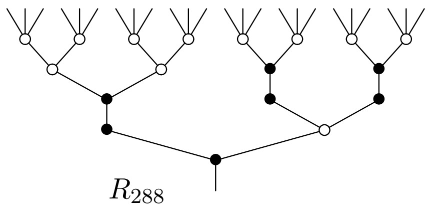
R_{285}



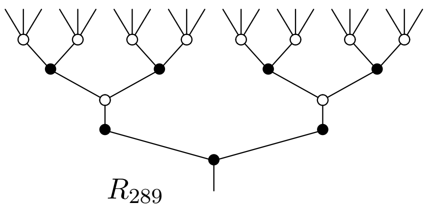
R_{286}



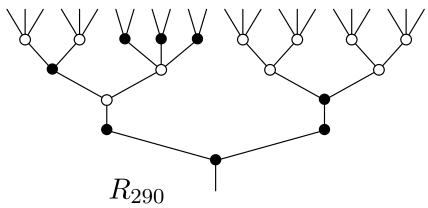
R_{287}



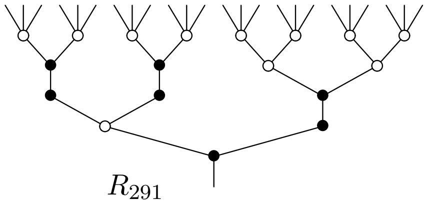
R_{288}



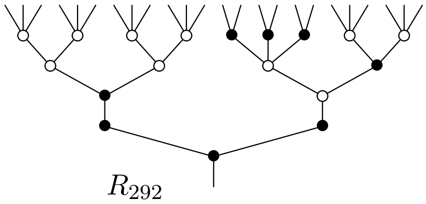
R_{289}



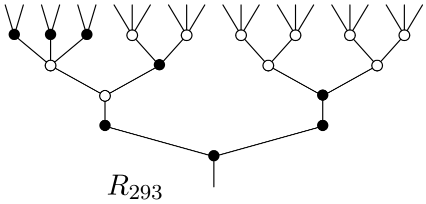
R_{290}



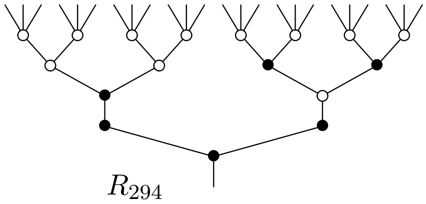
R_{291}



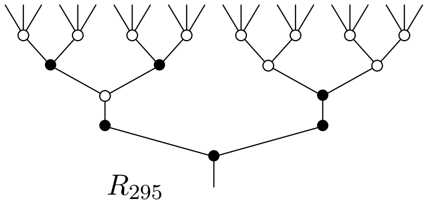
R_{292}



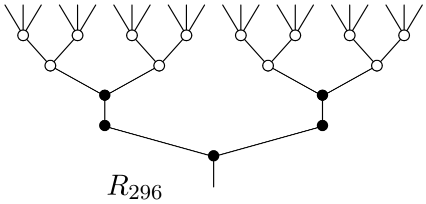
R_{293}



R_{294}



R_{295}



R_{296}

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

Fichero 1: operad.py

```python
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
```

Fichero 2: tree.py

```
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]
```

```python
new_oper = self.T_og_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()
```

```python
        )

        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,
            )
```

Fichero 6: latex_gen.py

```python
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")
```