

Combinatoric CheatSheet

SymPy.org

Partition

Path: from sympy.combinatorics.partitions

Methods

random_integer_partition(n, seed=None)
Generates a random integer partition summing to n as a list of reverse-sorted integers

RGS_generalized(m)
Computes the $m + 1$ generalized unrestricted growth strings and returns them as rows in matrix

RGS_enum(m)
computes the total number of restricted growth strings possible for a superset of size m

RGS_unrank(rank, m)
Gives the unranked restricted growth string for a given superset size

RGS_rank(rgs)
Computes the rank of a restricted growth string.

Subclass Partition

A partition is a set of disjoint sets whose union equals a given set. This Class represent abstract partition.

| | |
|--------------------------------|--|
| rgs | Restricted Growth String |
| from_rgs (rgs,elements) | Creates a set partition from a RSG |
| rank | Gets the rank of a partition |
| partition | Return partition as a sorted list of lists |
| sort_key (order=None) | Return a canonical key that can be used for sorting. |

Subclass IntegerPartition

This class represents an integer partition.

| | |
|------------------------------|--|
| as_dict () | Return the partition as a dictionary whose keys are the partition integers and the values are the multiplicity of that integer |
| as_ferrers (char='#') | Prints the ferrer diagram of a partition |
| conjugate | Computes the conjugate partition of itself |
| next_lex () | Return the next partition of the integer, n, in lexical order |
| prev_lex () | Return the previous partition of the integer, n, in lexical order |

Permutation

Path sympy.combinatorics.permutations.Permutation

Methods

array_form
This is used to convert from cyclic notation to the canonical notation

ascents()

Returns the positions of ascents in a permutation, i.e., the location where $p[i] < p[i + 1]$

descents() Returns the positions of descents in a permutation, i.e., the location where $p[i] > p[i + 1]$

atoms()
Returns all the elements of a permutation

cardinality
Returns the number of all possible permutations.

commutator(x)
Return the commutator of self and x: $x * self * x * self$

commutes_with(other)
Checks if the elements are commuting.

cycle_structure
Return the cycle structure of the permutation as a dictionary indicating the multiplicity of each cycle length.

cycles
Returns the number of cycles contained in the permutation (including singletons).

cyclic_form
This is used to convert to the cyclic notation from the canonical notation. Singletons are omitted.

from_inversion_vector(inversion)
Calculates the permutation from the inversion vector.

from_sequence(i, key=None)
Return the permutation needed to obtain i from the sorted elements of i. If custom sorting is desired, a key can be given.

full_cyclic_form
Return permutation in cyclic form including singletons.

get_adjacency_distance(other)
Computes the adjacency distance between two permutations.

get_adjacency_matrix()
Computes the adjacency matrix of a permutation.

get_positional_distance(other)
Computes the positional distance between two permutations.

get_precedence_distance(other)
Computes the precedence distance between two permutations.

get_precedence_matrix()
Gets the precedence matrix. This is used for computing the distance between two permutations.

index()
Returns the index of a permutation.

inversion_vector()
Return the inversion vector of the permutation.

inversions()
Computes the number of inversions of a permutation.

is_Empty
Checks to see if the permutation is a set with zero elements

is_Identity
Returns True if the Permutation is an identity permutation.

is_Singleton
Checks to see if the permutation contains only one number and is thus the only possible permutation of this set of numbers.

is_even
Checks if a permutation is even.

is_odd
Checks if a permutation is odd.

josephus(m, n, s=1)

Return as a permutation the shuffling of range(n) using the Josephus scheme in which every m-th item is selected until all have been chosen.

length()
Returns the number of integers moved by a permutation.

list(size=None)
Return the permutation as an explicit list

max()
The maximum element moved by the permutation.

min()
The minimum element moved by the permutation

next_lex()
Returns the next permutation in lexicographical order.

next_nonlex()
Returns the next permutation in nonlex order.

next_trotterjohnson()
Returns the next permutation in Trotter-Johnson order.

order()
Computes the order of a permutation.

parity()
Computes the parity of a permutation.

random(n)
Generates a random permutation of length n.

rank(i=None)
Returns the lexicographic rank of the permutation (default) or the ith ranked permutation of self.

rank_nonlex(inv_perm=None)
This is a linear time ranking algorithm that does not enforce lexicographic order.

rank_trotterjohnson()
Returns the Trotter Johnson rank, which we get from the minimal change algorithm.

static rmul(*args)
Return product of Permutations $[a, b, c, \dots]$ as the Permutation whose i th value is $a(b(c(i)))$.

runs()
Returns the runs of a permutation.

signature()
Gives the signature of the permutation needed to place the elements of the permutation in canonical order.

size
Returns the number of elements in the permutation.

support()
Return the elements in permutation, P , for which $P[i] \neq i$.

transpositions()
Return the permutation decomposed into a list of transpositions.

unrank_lex(size, rank)
Lexicographic permutation unranking.

unrank_nonlex(n, r)
This is a linear time unranking algorithm that does not respect lexicographic order.

unrank_trotterjohnson(size, rank)
Trotter Johnson permutation unranking.

Subclass Cycle(*args)

Wrapper around dict which provides the functionality of a disjoint cycle.

Subclass Generators

symmetric(n)

Generates the symmetric group of order **n**, **Sn**.

cyclic(n)

Generates the cyclic group of order **n**, **Cn**.

alternating(n)

Generates the alternating group of order **n**, **An**.

dihedral(n)

Generates the dihedral group of order **2n**, **Dn**.

PermutationGroup

Path : `sympy.combinatorics.perm_groups.PermutationGroup`

Methods

base

Return a base from the Schreier-Sims algorithm.

baseswap(base, strong_gens, pos, randomized=False, transversals=None, basic_orbits=None, strong_gens_distr=None)

Swap two consecutive base points in base and strong generating set.

basic_orbits

Return the basic orbits relative to a base and strong generating set.

basic_stabilizers

Return a chain of stabilizers relative to a base and strong generating set.

basic_transversals

Return basic transversals relative to a base and strong generating set.

center()

Return the center of a permutation group.

centralizer(other)

Return the centralizer of a group/set/element.

commutator(G, H)

Return the commutator of two subgroups.

contains(g, strict=True)

Test if permutation **g** belong to self.

coset_factor(g, af=False)

Return **G**'s (self's) coset factorization, **f**, of **g**.

coset_rank(g)

rank using Schreier-Sims representation

coset_unrank(rank, af=False)

unrank using Schreier-Sims representation

degree

Returns the size of the permutations in the group.

derived_series()

Return the derived series for the group.

derived_subgroup()

Compute the derived subgroup.

generate(method='coset', af=False)

Return iterator to generate the elements of the group

generate_dimino(af=False)

Yield group elements using Dimino's algorithm

generate_schreier_sims(af=False)

Yield group elements using the Schreier-Sims representation.

generators

Returns the generators of the group.

is_abelian

Test if the group is Abelian.

is_alt_sym(eps=0.05, _random_prec=None)

Monte Carlo test for the symmetric/alternating group for degrees ≥ 8 .

is_group()

Return True if the group if identity is present, the inverse of every element is also an element, and the product of any two elements is also an element.

is_nilpotent

Test if the group is nilpotent.

is_normal(gr)

Test if **G**=self is a normal subgroup of **gr**.

is_primitive(randomized=True)

Test if a group is primitive.

is_solvable

Test if the group is solvable.

is_subgroup(G, strict=True)

Return True if all elements of self belong to **G**.

is_transitive(strict=True)

Test if the group is transitive.

is_trivial

Test if the group is the trivial group.

lower_central_series()

Return the lower central series for the group.

make_perm(n, seed=None)

Multiply **n** randomly selected permutations from **pgroup** together, starting with the identity permutation.

max_div

Maximum proper divisor of the degree of a permutation group.

minimal_block(points)

For a transitive group, finds the block system generated by **points**.

normal_closure(other, k=10)

Return the normal closure of a subgroup/set of permutations.

orbit(alpha, action='tuples')

Compute the orbit of $\alpha \setminus \{g(\alpha) \mid g \in G\}$ as a set.

orbit_rep(alpha, beta, schreier_vector=None)

Return a group element which sends **alpha** to **beta**.

orbit_transversal(alpha, pairs=False)

Computes a transversal for the orbit of **alpha** as a set.

orbits(rep=False)

Return the orbits of self, ordered according to lowest element in each orbit.

order()

Return the number of permutations that can be generated from elements of the group.

pointwise_stabilizer(points, incremental=False)

Return the pointwise stabilizer for a set of points.

random(af=False)

Return a random group element.

random_pr(gen_count=11, iterations=50, _random_prec=None)

Return a random group element using product replacement.

random_stab(alpha, schreier_vector=None, _random_prec=None)

Random element from the stabilizer of **alpha**.

schreier_sims()

Schreier-Sims algorithm.

schreier_sims_incremental(base=None, gens=None)

Extend a sequence of points and generating set to a base and strong generating set.

schreier_sims_random(base=None, gens=None, consec_succ=10, _random_prec=None)

Randomized Schreier-Sims algorithm.

schreier_vector(alpha)

Computes the schreier vector for **alpha**.

stabilizer(alpha)

Return the stabilizer subgroup of **alpha**.

stabilizer_cosets(af=False)

Return a list of cosets of the stabilizer chain of the group as computed by the Schreier-Sims algorithm.

stabilizer_gens(af=False)

Return the generators of the chain of stabilizers of the Schreier-Sims representation.

strong_gens

Return a strong generating set from the Schreier-Sims algorithm.

subgroup_search(prop, base=None, strong_gens=None, tests=None, init_subgroup=None)

Find the subgroup of all elements satisfying the property **prop**.

transitivity_degree

Compute the degree of transitivity of the group.

Polyhedron

Path : `sympy.combinatorics.polyhedron.Polyhedron`

Represents the polyhedral symmetry group (PSG).

Methods

array_form

Return the indices of the corners.

corners

Get the corners of the Polyhedron.

cyclic_form

Return the indices of the corners in cyclic notation.

edges

Given the faces of the polyhedra we can get the edges.

faces

Get the faces of the Polyhedron.

pgroup

Get the permutations of the Polyhedron.

reset()

Return corners to their original positions.

rotate(perm)

Apply a permutation to the polyhedron in place.

size

Get the number of corners of the Polyhedron.

vertices

Get the corners of the Polyhedron.

Prufer

Path: `sympy.combinatorics.prufer.Prufer`

The Prufer correspondence is an algorithm that describes the bijection between labeled trees and the Prufer code. A Prufer code of a labeled tree is unique up to isomorphism and has a length of $n - 2$.

Methods

static edges(*runs)

Return a list of edges and the number of nodes from the given runs that connect nodes in an integer-labelled tree.

next(delta=1)

Generates the Prufer sequence that is delta beyond the current one.

nodes

Returns the number of nodes in the tree.

prev(delta=1)

Generates the Prufer sequence that is -delta before the current one.

prufer_rank()

Computes the rank of a Prufer sequence.

prufer_repr

Returns Prufer sequence for the Prufer object.

rank

Returns the rank of the Prufer sequence.

size

Return the number of possible trees of this Prufer object.

static to_prufer(tree, n)

Return the Prufer sequence for a tree given as a list of edges where n is the number of nodes in the tree.

static to_tree(prufer)

Return the tree (as a list of edges) of the given Prufer sequence.

tree_repr

Returns the tree representation of the Prufer object.

unrank(rank, n)

Finds the unranked Prufer sequence.

Subset

Path: `sympy.combinatorics.subsets.Subset`

Represents a basic subset object.

Methods

bitlist_from_subset(subset, superset)

Gets the bitlist corresponding to a subset.

cardinality

Returns the number of all possible subsets.

iterate_binary(k)

This is a helper function. It iterates over the binary subsets by k steps. This variable can be both positive or negative.

iterate_graycode(k)

It performs k step overs to get the respective Gray codes.

next_binary()

Generates the next binary ordered subset.

next_gray()

Generates the next Gray code ordered subset.

next_lexicographic()

Generates the next lexicographically ordered subset. NOT IMPLEMENTED

prev_binary()

Generates the previous binary ordered subset.

prev_gray()

Generates the previous Gray code ordered subset.

prev_lexicographic()

Generates the previous lexicographically ordered subset. NOT IMPLEMENTED

rank_binary

Computes the binary ordered rank.

rank_gray

Computes the Gray code ranking of the subset.

rank_lexicographic

Computes the lexicographic ranking of the subset.

size

Gets the size of the subset.

subset

Gets the subset represented by the current instance.

subset_from_bitlist(super_set, bitlist)

Gets the subset defined by the bitlist.

subset_indices(subset, superset)

Return indices of subset in superset in a list; the list is empty if all elements of **subset** are not in **superset**.

superset

Gets the superset of the subset.

superset_size

Returns the size of the superset.

unrank_binary(rank, superset)

Gets the binary ordered subset of the specified rank.

unrank_gray(rank, superset)

Gets the Gray code ordered subset of the specified rank.

subsets.ksubsets(superset, k)

Finds the subsets of size k in lexicographic order.

Gray Code

Path: `sympy.combinatorics.graycode.GrayCode` A Gray code is essentially a Hamiltonian walk on an n-dimensional cube with edge length of one. The vertices of the cube are represented by vectors whose values are binary. The Hamilton walk visits each vertex exactly once.

Methods

current

Returns the currently referenced Gray code as a bit string.

generate_gray(hints)**

Generates the sequence of bit vectors of a Gray Code.

n

Returns the dimension of the Gray code.

next(delta=1)

Returns the Gray code a distance delta (default = 1) from the current value in canonical order.

rank

Ranks the Gray code.

selections

Returns the number of bit vectors in the Gray code.

skip()

Skips the bit generation.

unrank(n, rank)

Unranks an n-bit sized Gray code of rank k. This method exists so that a derivative GrayCode class can define its own code of a given rank.

graycode.random_bitstring(n)

Generates a random bitlist of length n.

graycode.gray_to_bin(bin_list)

Convert from Gray coding to binary coding.

graycode.bin_to_gray(bin_list)

Convert from binary coding to gray coding.

graycode.get_subset_from_bitstring(super_set, bitstring)

Gets the subset defined by the bitstring.

graycode.graycode_subsets(gray_code_set)

Generates the subsets as enumerated by a Gray code.

Named Groups

Path: `sympy.combinatorics.named_groups`

Methods

SymmetricGroup(n)

Generates the symmetric group on n elements as a permutation group.

CyclicGroup(n)

Generates the cyclic group of order n as a permutation group.

DihedralGroup(n)

Generates the dihedral group Dn as a permutation group.

AlternatingGroup(n)

Generates the alternating group on n elements as a permutation group.

AbelianGroup(*cyclic_orders)

Returns the direct product of cyclic groups with the given orders.

Utilities

Path: `sympy.combinatorics.util`

Methods

_base_ordering(base, degree)

Order $\{0, 1, \dots, n\}$ so that base points come first and in order

_check_cycles_alt_sym(perm)

Checks for cycles of prime length p with $n/2 < p < n - 2$.

_distribute_gens_by_base(base, gens)

Distribute the group elements **gens** by membership in basic stabilizers.

_handle_precomputed_bsgs(base, strong_gens,

transversals=None, basic_orbits=None,

strong_gens_distr=None)

Calculate BSGS-related structures from those present.

_orbits_transversals_from_bsgs(base, strong_gens_distr,

transversals_only=False)

Compute basic orbits and transversals from a base and strong generating set.

_remove_gens(base, strong_gens,

basic_orbits=None, strong_gens_distr=None)

Remove redundant generators from a strong generating set.

_strip(g, base, orbits, transversals)

Attempt to decompose a permutation using a (possibly partial) BSGS structure.

`_strong_gens_from_distr(strong_gens_distr)`

Retrieve strong generating set from generators of basic stabilizers.

Group Constructors

Path: `sympy.combinatorics.group_constructs`

Method

`DirectProduct(*groups)`

Returns the direct product of several groups as a permutation group.

Test Utilities

Path: `sympy.combinatorics.testutil`

Methods

`_cmp_perm_lists(first, second)`

Compare two lists of permutations as sets.

`_naive_list_centralizer(self, other)`

`_verify_bsgs(group, base, gens)`

Verify the correctness of a base and strong generating set.

`_verify_centralizer(group, arg, centr=None)`

Verify the centralizer of a group/set/element inside another group.

`_verify_normal_closure(group, arg, closure=None)`

<https://www.sympy.org/cheatsheets>