

# 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.

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

#### Subclass IntegerPartition

This class represents an integer partition.

<b>as_dict()</b>	Return the partition as a dictionary whose keys are the partition integers and the values are the multiplicity of that integer
<b>as_ferrers(char='#')</b>	Prints the ferrer diagram of a partition
<b>conjugate</b>	Computes the conjugate partition of itself
<b>next_lex()</b>	Return the next partition of the integer, n, in lexical order
<b>prev_lex()</b>	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:  $\sim x \sim self * x * self$

**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 (selfs) 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 Diminos 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  $\geq 8$ .

**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 a 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)`

---

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