Explanation

A group is transitive if it has a single orbit.

If `strict` is `False` the group is transitive if it has a single orbit of length different from 1.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([2, 0, 1, 3])
>>> G1 = PermutationGroup([a, b])
>>> G1.is_transitive()
False
>>> G1.is_transitive(strict=False)
True
>>> c = Permutation([2, 3, 0, 1])
>>> G2 = PermutationGroup([a, c])
>>> G2.is_transitive()
True
>>> d = Permutation([1, 0, 2, 3])
>>> e = Permutation([0, 1, 3, 2])
>>> G3 = PermutationGroup([d, e])
>>> G3.is_transitive() or G3.is_transitive(strict=False)
False
```

property is_trivial

Test if the group is the trivial group.

This is true if the group contains only the identity permutation.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> G = PermutationGroup([Permutation([0, 1, 2])])
>>> G.is_trivial
True
```

lower_central_series()

Return the lower central series for the group.

The lower central series for a group G is the series $G = G_0 > G_1 > G_2 > \ldots$ where $G_k = [G, G_{k-1}]$, i.e. every term after the first is equal to the commutator of G and the previous term in $G1$ ([1], p.29).

Returns

A list of permutation groups in the order $G = G_0, G_1, G_2, \ldots$

Examples

```
>>> from sympy.combinatorics.named_groups import (AlternatingGroup,
... DihedralGroup)
>>> A = AlternatingGroup(4)
>>> len(A.lower_central_series())
2
>>> A.lower_central_series()[1].is_subgroup(DihedralGroup(2))
True
```

See also:

commutator (page 300), *derived_series* (page 306)

make_perm(*n*, *seed=None*)

Multiply `n` randomly selected permutations from pgroup together, starting with the identity permutation. If `n` is a list of integers, those integers will be used to select the permutations and they will be applied in L to R order: make_perm((A, B, C)) will give CBA(I) where I is the identity permutation.

`seed` is used to set the seed for the random selection of permutations from pgroup. If this is a list of integers, the corresponding permutations from pgroup will be selected in the order give. This is mainly used for testing purposes.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a, b = [Permutation([1, 0, 3, 2]), Permutation([1, 3, 0, 2])]
>>> G = PermutationGroup([a, b])
>>> G.make_perm(1, [0])
(0 1)(2 3)
>>> G.make_perm(3, [0, 1, 0])
(0 2 3 1)
>>> G.make_perm([0, 1, 0])
(0 2 3 1)
```

See also:

random (page 324)

property max_div

Maximum proper divisor of the degree of a permutation group.

Explanation

Obviously, this is the degree divided by its minimal proper divisor (larger than `1`, if one exists). As it is guaranteed to be prime, the `sieve` from `sympy.ntheory` is used. This function is also used as an optimization tool for the functions `minimal_block` and `_union_find_merge`.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> G = PermutationGroup([Permutation([0, 2, 1, 3])])
>>> G.max_div
2
```

See also:

minimal_block (page 319), *_union_find_merge* (page 292)

minimal_block(*points*)

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

Explanation

If a group `G` acts on a set `S`, a nonempty subset `B` of `S` is called a block under the action of `G` if for all `g` in `G` we have `gB = B` (`g` fixes `B`) or `gB` and `B` have no common points (`g` moves `B` entirely). ([1], p.23; [6]).

The distinct translates `gB` of a block `B` for `g` in `G` partition the set `S` and this set of translates is known as a block system. Moreover, we obviously have that all blocks in the partition have the same size, hence the block size divides `|S|` ([1], p.23). A `G`-congruence is an equivalence relation `~` on the set `S` such that `a ~ b` implies `g(a) ~ g(b)` for all `g` in `G`. For a transitive group, the equivalence classes of a `G`-congruence and the blocks of a block system are the same thing ([1], p.23).

The algorithm below checks the group for transitivity, and then finds the `G`-congruence generated by the pairs `(p_0, p_1), (p_0, p_2), ..., (p_0, p_{k-1})` which is the same as finding the maximal block system (i.e., the one with minimum block size) such that `p_0, ..., p_{k-1}` are in the same block ([1], p.83).

It is an implementation of Atkinson's algorithm, as suggested in [1], and manipulates an equivalence relation on the set `S` using a union-find data structure. The running time is just above $O(|points||S|)$. ([1], pp. 83-87; [7]).

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.minimal_block([0, 5])
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
>>> D.minimal_block([0, 1])
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

See also:

_union_find_rep (page 293), *_union_find_merge* (page 292), *is_transitive* (page 316), *is_primitive* (page 314)

minimal_blocks(*randomized=True*)

For a transitive group, return the list of all minimal block systems. If a group is intransitive, return $False$.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> DihedralGroup(6).minimal_blocks()
[[0, 1, 0, 1, 0, 1], [0, 1, 2, 0, 1, 2]]
>>> G = PermutationGroup(Permutation(1,2,5))
>>> G.minimal_blocks()
False
```

See also:

minimal_block (page 319), *is_transitive* (page 316), *is_primitive* (page 314)

normal_closure(*other*, *k=10*)

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

Parameters

other

a subgroup/list of permutations/single permutation

k

an implementation-specific parameter that determines the number of conjugates that are adjoined to `other` at once

Explanation

If `S` is a subset of a group `G`, the normal closure of `A` in `G` is defined as the intersection of all normal subgroups of `G` that contain `A` ([1], p.14). Alternatively, it is the group generated by the conjugates `x^{-1}yx` for `x` a generator of `G` and `y` a generator of the subgroup `\left\langle S\right\rangle` generated by `S` (for some chosen generating set for `\left\langle S\right\rangle`) ([1], p.73).

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup, AlternatingGroup)
>>> S = SymmetricGroup(5)
>>> C = CyclicGroup(5)
>>> G = S.normal_closure(C)
>>> G.order()
60
>>> G.is_subgroup(AlternatingGroup(5))
True
```

Notes

The algorithm is described in [1], pp. 73-74; it makes use of the generation of random elements for permutation groups by the product replacement algorithm.

See also:

commutator (page 300), *derived_subgroup* (page 306), *random_pr* (page 324)

orbit(*alpha*, *action='tuples'*)

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

Explanation

The time complexity of the algorithm used here is $O(|Orb| * r)$ where $|Orb|$ is the size of the orbit and `r` is the number of generators of the group. For a more detailed analysis, see [1], p.78, [2], pp. 19-21. Here alpha can be a single point, or a list of points.

If alpha is a single point, the ordinary orbit is computed. if alpha is a list of points, there are three available options:

'union' - computes the union of the orbits of the points in the list 'tuples' - computes the orbit of the list interpreted as an ordered tuple under the group action (i.e., g((1,2,3)) = (g(1), g(2), g(3))) 'sets' - computes the orbit of the list interpreted as a sets

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([1, 2, 0, 4, 5, 6, 3])
>>> G = PermutationGroup([a])
>>> G.orbit(0)
{0, 1, 2}
>>> G.orbit([0, 4], 'union')
{0, 1, 2, 3, 4, 5, 6}
```

See also:

orbit_transversal (page 322)

orbit_rep(*alpha*, *beta*, *schreier_vector=None*)

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

Explanation

If `beta` is not in the orbit of `alpha`, the function returns `False`. This implementation makes use of the schreier vector. For a proof of correctness, see [1], p.80

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> G = AlternatingGroup(5)
>>> G.orbit_rep(0, 4)
(0 4 1 2 3)
```

See also:

schreier_vector (page 327)

orbit_transversal(*alpha*, *pairs=False*)

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

Explanation

For a permutation group G, a transversal for the orbit $Orb = \{g(\alpha)|g \in G\}$ is a set $\{g_\beta|g_\beta(\alpha) = \beta\}$ for $\beta \in Orb$. Note that there may be more than one possible transversal. If `pairs` is set to `True`, it returns the list of pairs (β, g_β). For a proof of correctness, see [1], p.79

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(6)
>>> G.orbit_transversal(0)
[(5), (0 1 2 3 4 5), (0 5)(1 4)(2 3), (0 2 4)(1 3 5), (5)(0 4)(1 3),␣
↪(0 3)(1 4)(2 5)]
```

See also:

orbit (page 321)

orbits(*rep=False*)

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

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation(1, 5)(2, 3)(4, 0, 6)
>>> b = Permutation(1, 5)(3, 4)(2, 6, 0)
>>> G = PermutationGroup([a, b])
>>> G.orbits()
[{0, 2, 3, 4, 6}, {1, 5}]
```

order()

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

The number of permutations comprising the group is given by `len(group)`; the length of each permutation in the group is given by `group.size`.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
```

```
>>> a = Permutation([1, 0, 2])
>>> G = PermutationGroup([a])
>>> G.degree
3
>>> len(G)
1
>>> G.order()
2
>>> list(G.generate())
[(2), (2)(0 1)]
```

```
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.order()
6
```

See also:

degree (page 305)

pointwise_stabilizer(*points*, *incremental=True*)

Return the pointwise stabilizer for a set of points.

Explanation

For a permutation group G and a set of points $\{p_1, p_2, \ldots, p_k\}$, the pointwise stabilizer of $p_1, p_2, \ldots, p_k$ is defined as $G_{p_1,\ldots,p_k} = \{g \in G | g(p_i) = p_i \forall i \in \{1, 2, \ldots, k\}\}$ ([1],p20). It is a subgroup of G.

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(7)
>>> Stab = S.pointwise_stabilizer([2, 3, 5])
>>> Stab.is_subgroup(S.stabilizer(2).stabilizer(3).stabilizer(5))
True
```

Notes

When incremental == True, rather than the obvious implementation using successive calls to `.stabilizer()`, this uses the incremental Schreier-Sims algorithm to obtain a base with starting segment - the given points.

See also:

stabilizer (page 328), *schreier_sims_incremental* (page 325)

polycyclic_group()

Return the PolycyclicGroup instance with below parameters:

Explanation

- `pc_sequence` : Polycyclic sequence is formed by collecting all the missing generators between the adjacent groups in the derived series of given permutation group.
- `pc_series` : Polycyclic series is formed by adding all the missing generators of `der[i+1]` in `der[i]`, where `der` represents the derived series.
- `relative_order` : A list, computed by the ratio of adjacent groups in pc_series.

presentation(*eliminate_gens=True*)

Return an $FpGroup$ presentation of the group.

The algorithm is described in [1], Chapter 6.1.

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.

Explanation

For the details of the product replacement algorithm, see `_random_pr_init` In `random_pr` the actual 'product replacement' is performed. Notice that if the attribute `_random_gens` is empty, it needs to be initialized by `_random_pr_init`.

See also:

_random_pr_init (page 291)

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

Random element from the stabilizer of `alpha`.

The schreier vector for `alpha` is an optional argument used for speeding up repeated calls. The algorithm is described in [1], p.81

See also:

random_pr (page 324), *orbit_rep* (page 321)

schreier_sims()

Schreier-Sims algorithm.

Explanation

It computes the generators of the chain of stabilizers $G > G_{b_1} > .. > G_{b1,..,b_r} > 1$ in which $G_{b_1,..,b_i}$ stabilizes $b_1, .., b_i$, and the corresponding `s` cosets. An element of the group can be written as the product $h_1 * .. * h_s$.

We use the incremental Schreier-Sims algorithm.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.schreier_sims()
>>> G.basic_transversals
[{0: (2)(0 1), 1: (2), 2: (1 2)},
 {0: (2), 2: (0 2)}]
```

schreier_sims_incremental(*base=None, gens=None, slp_dict=False*)

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

Parameters

base

The sequence of points to be extended to a base. Optional parameter with default value `[]`.

gens

The generating set to be extended to a strong generating set relative to the base obtained. Optional parameter with default value `self.generators`.

slp_dict

If $True$, return a dictionary $g : gens$ for each strong generator g where $gens$ is a list of strong generators coming before g in $strong_g ens$, such that the product of the elements of $gens$ is equal to g.

Returns

(base, strong_gens)

`base` is the base obtained, and `strong_gens` is the strong generating set relative to it. The original parameters `base`, `gens` remain unchanged.

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> A = AlternatingGroup(7)
>>> base = [2, 3]
>>> seq = [2, 3]
>>> base, strong_gens = A.schreier_sims_incremental(base=seq)
>>> _verify_bsgs(A, base, strong_gens)
True
>>> base[:2]
[2, 3]
```

Notes

This version of the Schreier-Sims algorithm runs in polynomial time. There are certain assumptions in the implementation - if the trivial group is provided, `base` and `gens` are returned immediately, as any sequence of points is a base for the trivial group. If the identity is present in the generators `gens`, it is removed as it is a redundant generator. The implementation is described in [1], pp. 90-93.

See also:

schreier_sims (page 324), *schreier_sims_random* (page 326)

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

Randomized Schreier-Sims algorithm.

Parameters

base

The sequence to be extended to a base.

gens

The generating set to be extended to a strong generating set.

consec_succ

The parameter defining the probability of a wrong answer.

_random_prec

An internal parameter used for testing purposes.

Returns

(base, strong_gens)

`base` is the base and `strong_gens` is the strong generating set relative to it.

Explanation

The randomized Schreier-Sims algorithm takes the sequence `base` and the generating set `gens`, and extends `base` to a base, and `gens` to a strong generating set relative to that base with probability of a wrong answer at most $2^{-consec\_succ}$, provided the random generators are sufficiently random.

Examples

```
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(5)
>>> base, strong_gens = S.schreier_sims_random(consec_succ=5)
>>> _verify_bsgs(S, base, strong_gens)
True
```

Notes

The algorithm is described in detail in [1], pp. 97-98. It extends the orbits `orbs` and the permutation groups `stabs` to basic orbits and basic stabilizers for the base and strong generating set produced in the end. The idea of the extension process is to "sift" random group elements through the stabilizer chain and amend the stabilizers/orbits along the way when a sift is not successful. The helper function `_strip` is used to attempt to decompose a random group element according to the current state of the stabilizer chain and report whether the element was fully decomposed (successful sift) or not (unsuccessful sift). In the latter case, the level at which the sift failed is reported and used to amend `stabs`, `base`, `gens` and `orbs` accordingly. The halting condition is for `consec_succ` consecutive successful sifts to pass. This makes sure that the current `base` and `gens` form a BSGS with probability at least $1 - 1/\text{consec\_succ}$.

See also:

schreier_sims (page 324)

schreier_vector(*alpha*)

Computes the schreier vector for `alpha`.

Explanation

The Schreier vector efficiently stores information about the orbit of `alpha`. It can later be used to quickly obtain elements of the group that send `alpha` to a particular element in the orbit. Notice that the Schreier vector depends on the order in which the group generators are listed. For a definition, see [3]. Since list indices start from zero, we adopt the convention to use "None" instead of 0 to signify that an element does not belong to the orbit. For the algorithm and its correctness, see [2], pp.78-80.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([2, 4, 6, 3, 1, 5, 0])
>>> b = Permutation([0, 1, 3, 5, 4, 6, 2])
>>> G = PermutationGroup([a, b])
>>> G.schreier_vector(0)
[-1, None, 0, 1, None, 1, 0]
```

See also:

orbit (page 321)

stabilizer(*alpha*)

Return the stabilizer subgroup of `alpha`.

Explanation

The stabilizer of α is the group $G_\alpha = \{g \in G | g(\alpha) = \alpha\}$. For a proof of correctness, see [1], p.79.

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(6)
>>> G.stabilizer(5)
PermutationGroup([
    (5)(0 4)(1 3)])
```

See also:

orbit (page 321)

property strong_gens

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

Explanation

A generating set $S = \{g_1, g_2, \ldots, g_t\}$ for a permutation group G is a strong generating set relative to the sequence of points (referred to as a "base") $(b_1, b_2, \ldots, b_k)$ if, for $1 \leq i \leq k$ we have that the intersection of the pointwise stabilizer $G^{(i+1)} := G_{b_1,b_2,\ldots,b_i}$ with S generates the pointwise stabilizer $G^{(i+1)}$. The concepts of a base and strong generating set and their applications are discussed in depth in [1], pp. 87-89 and [2], pp. 55-57.

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(4)
>>> D.strong_gens
[(0 1 2 3), (0 3)(1 2), (1 3)]
>>> D.base
[0, 1]
```

See also:

base (page 295), *basic_transversals* (page 298), *basic_orbits* (page 297), *basic_stabilizers* (page 297)

strong_presentation()

Return a strong finite presentation of group. The generators of the returned group are in the same order as the strong generators of group.

The algorithm is based on Sims' Verify algorithm described in [1], Chapter 6.

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> P = DihedralGroup(4)
>>> G = P.strong_presentation()
>>> P.order() == G.order()
True
```

See also:

presentation (page 324), *_verify* (page 293)

subgroup(*gens*)

Return the subgroup generated by $gens$ which is a list of elements of the group

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

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

Parameters

prop

The property to be used. Has to be callable on group elements and always return `True` or `False`. It is assumed that all group elements satisfying `prop` indeed form a subgroup.

base

A base for the supergroup.

strong_gens

A strong generating set for the supergroup.

tests

A list of callables of length equal to the length of `base`. These are used to rule out group elements by partial base images, so that

`tests[l](g)` returns False if the element `g` is known not to satisfy prop base on where g sends the first `l + 1` base points.

init_subgroup

if a subgroup of the sought group is known in advance, it can be passed to the function as this parameter.

Returns

res

The subgroup of all elements satisfying `prop`. The generating set for this group is guaranteed to be a strong generating set relative to the base `base`.

Explanation

This is done by a depth-first search with respect to base images that uses several tests to prune the search tree.

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> S = SymmetricGroup(7)
>>> prop_even = lambda x: x.is_even
>>> base, strong_gens = S.schreier_sims_incremental()
>>> G = S.subgroup_search(prop_even, base=base, strong_gens=strong_
↪gens)
>>> G.is_subgroup(AlternatingGroup(7))
True
>>> _verify_bsgs(G, base, G.generators)
True
```

Notes

This function is extremely lengthy and complicated and will require some careful attention. The implementation is described in [1], pp. 114-117, and the comments for the code here follow the lines of the pseudocode in the book for clarity.

The complexity is exponential in general, since the search process by itself visits all members of the supergroup. However, there are a lot of tests which are used to prune the search tree, and users can define their own tests via the `tests` parameter, so in practice, and for some computations, it's not terrible.

A crucial part in the procedure is the frequent base change performed (this is line 11 in the pseudocode) in order to obtain a new basic stabilizer. The book mentiones that this can be done by using `.baseswap(...)`, however the current implementation uses a more straightforward way to find the next basic stabilizer - calling the function `.stabilizer(...)` on the previous basic stabilizer.

sylow_subgroup(*p*)

Return a p-Sylow subgroup of the group.

The algorithm is described in [1], Chapter 4, Section 7

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.named_groups import AlternatingGroup
```

```
>>> D = DihedralGroup(6)
>>> S = D.sylow_subgroup(2)
>>> S.order()
4
>>> G = SymmetricGroup(6)
>>> S = G.sylow_subgroup(5)
>>> S.order()
5
```

```
>>> G1 = AlternatingGroup(3)
>>> G2 = AlternatingGroup(5)
>>> G3 = AlternatingGroup(9)
```

```
>>> S1 = G1.sylow_subgroup(3)
>>> S2 = G2.sylow_subgroup(3)
>>> S3 = G3.sylow_subgroup(3)
```

```
>>> len1 = len(S1.lower_central_series())
>>> len2 = len(S2.lower_central_series())
>>> len3 = len(S3.lower_central_series())
```

```
>>> len1 == len2
True
>>> len1 < len3
True
```

property transitivity_degree

Compute the degree of transitivity of the group.

Explanation

A permutation group G acting on $\Omega = \{0, 1, \ldots, n-1\}$ is k-fold transitive, if, for any k points $(a_1, a_2, \ldots, a_k) \in \Omega$ and any k points $(b_1, b_2, \ldots, b_k) \in \Omega$ there exists $g \in G$ such that $g(a_1) = b_1, g(a_2) = b_2, \ldots, g(a_k) = b_k$ The degree of transitivity of G is the maximum k such that G is k-fold transitive. ([8])

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([1, 2, 0])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.transitivity_degree
3
```

See also:

is_transitive (page 316), *orbit* (page 321)

Polyhedron

class sympy.combinatorics.polyhedron.**Polyhedron**(*corners*, *faces=()*, *pgroup=()*)

Represents the polyhedral symmetry group (PSG).

Explanation

The PSG is one of the symmetry groups of the Platonic solids. There are three polyhedral groups: the tetrahedral group of order 12, the octahedral group of order 24, and the icosahedral group of order 60.

All doctests have been given in the docstring of the constructor of the object.

References

[R81]

property array_form

Return the indices of the corners.

The indices are given relative to the original position of corners.

Examples

```
>>> from sympy.combinatorics.polyhedron import tetrahedron
>>> tetrahedron = tetrahedron.copy()
>>> tetrahedron.array_form
[0, 1, 2, 3]
```

```
>>> tetrahedron.rotate(0)
>>> tetrahedron.array_form
[0, 2, 3, 1]
>>> tetrahedron.pgroup[0].array_form
[0, 2, 3, 1]
```

See also:

corners (page 333), *cyclic_form* (page 333)

property corners

Get the corners of the Polyhedron.

The method `vertices` is an alias for `corners`.

Examples

```
>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c, d
>>> p = Polyhedron(list('abcd'))
>>> p.corners == p.vertices == (a, b, c, d)
True
```

See also:

array_form (page 332), *cyclic_form* (page 333)

property cyclic_form

Return the indices of the corners in cyclic notation.

The indices are given relative to the original position of corners.

See also:

corners (page 333), *array_form* (page 332)

property edges

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

Examples

```
>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c
>>> corners = (a, b, c)
>>> faces = [(0, 1, 2)]
>>> Polyhedron(corners, faces).edges
{(0, 1), (0, 2), (1, 2)}
```

property faces

Get the faces of the Polyhedron.

property pgroup

Get the permutations of the Polyhedron.

reset()

Return corners to their original positions.

Examples

```
>>> from sympy.combinatorics.polyhedron import tetrahedron as T
>>> T = T.copy()
>>> T.corners
(0, 1, 2, 3)
>>> T.rotate(0)
>>> T.corners
(0, 2, 3, 1)
>>> T.reset()
>>> T.corners
(0, 1, 2, 3)
```

rotate(*perm*)

Apply a permutation to the polyhedron *in place*. The permutation may be given as a Permutation instance or an integer indicating which permutation from pgroup of the Polyhedron should be applied.

This is an operation that is analogous to rotation about an axis by a fixed increment.

Notes

When a Permutation is applied, no check is done to see if that is a valid permutation for the Polyhedron. For example, a cube could be given a permutation which effectively swaps only 2 vertices. A valid permutation (that rotates the object in a physical way) will be obtained if one only uses permutations from the `pgroup` of the Polyhedron. On the other hand, allowing arbitrary rotations (applications of permutations) gives a way to follow named elements rather than indices since Polyhedron allows vertices to be named while Permutation works only with indices.

Examples

```
>>> from sympy.combinatorics import Polyhedron, Permutation
>>> from sympy.combinatorics.polyhedron import cube
>>> cube = cube.copy()
>>> cube.corners
(0, 1, 2, 3, 4, 5, 6, 7)
>>> cube.rotate(0)
>>> cube.corners
(1, 2, 3, 0, 5, 6, 7, 4)
```

A non-physical “rotation” that is not prohibited by this method:

```
>>> cube.reset()
>>> cube.rotate(Permutation([[1, 2]], size=8))
>>> cube.corners
(0, 2, 1, 3, 4, 5, 6, 7)
```

Polyhedron can be used to follow elements of set that are identified by letters instead of integers:

```
>>> shadow = h5 = Polyhedron(list('abcde'))
>>> p = Permutation([3, 0, 1, 2, 4])
>>> h5.rotate(p)
>>> h5.corners
(d, a, b, c, e)
>>> _ == shadow.corners
True
>>> copy = h5.copy()
>>> h5.rotate(p)
>>> h5.corners == copy.corners
False
```

property size

Get the number of corners of the Polyhedron.

property vertices

Get the corners of the Polyhedron.

The method `vertices` is an alias for `corners`.

Examples

```
>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c, d
>>> p = Polyhedron(list('abcd'))
>>> p.corners == p.vertices == (a, b, c, d)
True
```

See also:

array_form (page 332), *cyclic_form* (page 333)

Prufer Sequences

class sympy.combinatorics.prufer.**Prufer**(**args*, ***kw_args*)

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.

Prufer sequences were first used by Heinz Prufer to give a proof of Cayley's formula.

References

[R82]

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.

All node numbers will be shifted so that the minimum node is 0. It is not a problem if edges are repeated in the runs; only unique edges are returned. There is no

assumption made about what the range of the node labels should be, but all nodes from the smallest through the largest must be present.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer.edges([1, 2, 3], [2, 4, 5]) # a T
([[0, 1], [1, 2], [1, 3], [3, 4]], 5)
```

Duplicate edges are removed:

```
>>> Prufer.edges([0, 1, 2, 3], [1, 4, 5], [1, 4, 6]) # a K
([[0, 1], [1, 2], [1, 4], [2, 3], [4, 5], [4, 6]], 7)
```

next(*delta=1*)

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

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> b = a.next(1) # == a.next()
>>> b.tree_repr
[[0, 2], [0, 1], [1, 3]]
>>> b.rank
1
```

See also:

prufer_rank (page 337), *rank* (page 337), *prev* (page 336), *size* (page 338)

property nodes

Returns the number of nodes in the tree.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).nodes
6
>>> Prufer([1, 0, 0]).nodes
5
```

prev(*delta=1*)

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

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [1, 2], [2, 3], [1, 4]])
>>> a.rank
36
>>> b = a.prev()
>>> b
Prufer([1, 2, 0])
>>> b.rank
35
```

See also:

prufer_rank (page 337), *rank* (page 337), *next* (page 336), *size* (page 338)

prufer_rank()

Computes the rank of a Prufer sequence.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> a.prufer_rank()
0
```

See also:

rank (page 337), *next* (page 336), *prev* (page 336), *size* (page 338)

property prufer_repr

Returns Prufer sequence for the Prufer object.

This sequence is found by removing the highest numbered vertex, recording the node it was attached to, and continuing until only two vertices remain. The Prufer sequence is the list of recorded nodes.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).prufer_repr
[3, 3, 3, 4]
>>> Prufer([1, 0, 0]).prufer_repr
[1, 0, 0]
```

See also:

to_prufer (page 338)

property rank

Returns the rank of the Prufer sequence.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> p = Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]])
>>> p.rank
778
>>> p.next(1).rank
779
>>> p.prev().rank
777
```

See also:

prufer_rank (page 337), *next* (page 336), *prev* (page 336), *size* (page 338)

property size

Return the number of possible trees of this Prufer object.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([0]*4).size == Prufer([6]*4).size == 1296
True
```

See also:

prufer_rank (page 337), *rank* (page 337), *next* (page 336), *prev* (page 336)

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.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> a.prufer_repr
[0, 0]
>>> Prufer.to_prufer([[0, 1], [0, 2], [0, 3]], 4)
[0, 0]
```

See also:

***prufer_repr* (page 337)**
returns Prufer sequence of a Prufer object.

static to_tree(*prufer*)

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

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([0, 2], 4)
>>> a.tree_repr
[[0, 1], [0, 2], [2, 3]]
>>> Prufer.to_tree([0, 2])
[[0, 1], [0, 2], [2, 3]]
```

See also:

tree_repr **(page 339)**
returns tree representation of a Prufer object.

References

[R83]

property tree_repr

Returns the tree representation of the Prufer object.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).tree_repr
[[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]
>>> Prufer([1, 0, 0]).tree_repr
[[1, 2], [0, 1], [0, 3], [0, 4]]
```

See also:

to_tree (page 338)

classmethod unrank(*rank*, *n*)

Finds the unranked Prufer sequence.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer.unrank(0, 4)
Prufer([0, 0])
```

Subsets

class sympy.combinatorics.subsets.**Subset**(*subset*, *superset*)

Represents a basic subset object.

Explanation

We generate subsets using essentially two techniques, binary enumeration and lexicographic enumeration. The Subset class takes two arguments, the first one describes the initial subset to consider and the second describes the superset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
['b']
>>> a.prev_binary().subset
['c']
```

classmethod bitlist_from_subset(*subset*, *superset*)

Gets the bitlist corresponding to a subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> Subset.bitlist_from_subset(['c', 'd'], ['a', 'b', 'c', 'd'])
'0011'
```

See also:

subset_from_bitlist (page 345)

property cardinality

Returns the number of all possible subsets.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.cardinality
16
```

See also:

subset (page 344), *superset* (page 345), *size* (page 344), *superset_size* (page 346)

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.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.iterate_binary(-2).subset
['d']
>>> a = Subset(['a', 'b', 'c'], ['a', 'b', 'c', 'd'])
>>> a.iterate_binary(2).subset
[]
```

See also:

next_binary (page 341), *prev_binary* (page 342)

iterate_graycode(*k*)

Helper function used for prev_gray and next_gray. It performs k step overs to get the respective Gray codes.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset([1, 2, 3], [1, 2, 3, 4])
>>> a.iterate_graycode(3).subset
[1, 4]
>>> a.iterate_graycode(-2).subset
[1, 2, 4]
```

See also:

next_gray (page 341), *prev_gray* (page 342)

next_binary()

Generates the next binary ordered subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
['b']
>>> a = Subset(['a', 'b', 'c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
[]
```

See also:

prev_binary (page 342), *iterate_binary* (page 340)

next_gray()
Generates the next Gray code ordered subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset([1, 2, 3], [1, 2, 3, 4])
>>> a.next_gray().subset
[1, 3]
```

See also:

iterate_graycode (page 341), *prev_gray* (page 342)

next_lexicographic()
Generates the next lexicographically ordered subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_lexicographic().subset
['d']
>>> a = Subset(['d'], ['a', 'b', 'c', 'd'])
>>> a.next_lexicographic().subset
[]
```

See also:

prev_lexicographic (page 343)

prev_binary()
Generates the previous binary ordered subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.prev_binary().subset
['a', 'b', 'c', 'd']
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.prev_binary().subset
['c']
```

See also:

next_binary (page 341), *iterate_binary* (page 340)

prev_gray()
Generates the previous Gray code ordered subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset([2, 3, 4], [1, 2, 3, 4, 5])
>>> a.prev_gray().subset
[2, 3, 4, 5]
```

See also:

iterate_graycode (page 341), *next_gray* (page 341)

prev_lexicographic()

Generates the previous lexicographically ordered subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.prev_lexicographic().subset
['d']
>>> a = Subset(['c','d'], ['a', 'b', 'c', 'd'])
>>> a.prev_lexicographic().subset
['c']
```

See also:

next_lexicographic (page 342)

property rank_binary

Computes the binary ordered rank.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset([], ['a','b','c','d'])
>>> a.rank_binary
0
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_binary
3
```

See also:

iterate_binary (page 340), *unrank_binary* (page 346)

property rank_gray

Computes the Gray code ranking of the subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c','d'], ['a','b','c','d'])
>>> a.rank_gray
2
>>> a = Subset([2, 4, 5], [1, 2, 3, 4, 5, 6])
>>> a.rank_gray
27
```

See also:

iterate_graycode (page 341), *unrank_gray* (page 346)

property rank_lexicographic

Computes the lexicographic ranking of the subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_lexicographic
14
>>> a = Subset([2, 4, 5], [1, 2, 3, 4, 5, 6])
>>> a.rank_lexicographic
43
```

property size

Gets the size of the subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.size
2
```

See also:

subset (page 344), *superset* (page 345), *superset_size* (page 346), *cardinality* (page 340)

property subset

Gets the subset represented by the current instance.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.subset
['c', 'd']
```

See also:

superset (page 345), *size* (page 344), *superset_size* (page 346), *cardinality* (page 340)

classmethod subset_from_bitlist(*super_set*, *bitlist*)

Gets the subset defined by the bitlist.

Examples

```
>>> from sympy.combinatorics import Subset
>>> Subset.subset_from_bitlist(['a', 'b', 'c', 'd'], '0011').subset
['c', 'd']
```

See also:

bitlist_from_subset (page 340)

classmethod 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`.

Examples

```
>>> from sympy.combinatorics import Subset
>>> superset = [1, 3, 2, 5, 4]
>>> Subset.subset_indices([3, 2, 1], superset)
[1, 2, 0]
>>> Subset.subset_indices([1, 6], superset)
[]
>>> Subset.subset_indices([], superset)
[]
```

property superset

Gets the superset of the subset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.superset
['a', 'b', 'c', 'd']
```

See also:

subset (page 344), *size* (page 344), *superset_size* (page 346), *cardinality* (page 340)

property superset_size

Returns the size of the superset.

Examples

```
>>> from sympy.combinatorics import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.superset_size
4
```

See also:

subset (page 344), *superset* (page 345), *size* (page 344), *cardinality* (page 340)

classmethod unrank_binary(*rank*, *superset*)

Gets the binary ordered subset of the specified rank.

Examples

```
>>> from sympy.combinatorics import Subset
>>> Subset.unrank_binary(4, ['a', 'b', 'c', 'd']).subset
['b']
```

See also:

iterate_binary (page 340), *rank_binary* (page 343)

classmethod unrank_gray(*rank*, *superset*)

Gets the Gray code ordered subset of the specified rank.

Examples

```
>>> from sympy.combinatorics import Subset
>>> Subset.unrank_gray(4, ['a', 'b', 'c']).subset
['a', 'b']
>>> Subset.unrank_gray(0, ['a', 'b', 'c']).subset
[]
```

See also:

iterate_graycode (page 341), *rank_gray* (page 343)

subsets.**ksubsets**(*k*)

Finds the subsets of size k in lexicographic order.

This uses the itertools generator.

Examples

```
>>> from sympy.combinatorics.subsets import ksubsets
>>> list(ksubsets([1, 2, 3], 2))
[(1, 2), (1, 3), (2, 3)]
>>> list(ksubsets([1, 2, 3, 4, 5], 2))
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4),     (2, 5), (3, 4), (3,␣
↪5), (4, 5)]
```

See also:

Subset (page 340)

Gray Code

class sympy.combinatorics.graycode.**GrayCode**(*n*, **args*, ***kw_args*)

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. The Gray code for a 3d cube is ['000','100','110','010','011','111','101', '001'].

A Gray code solves the problem of sequentially generating all possible subsets of n objects in such a way that each subset is obtained from the previous one by either deleting or adding a single object. In the above example, 1 indicates that the object is present, and 0 indicates that its absent.

Gray codes have applications in statistics as well when we want to compute various statistics related to subsets in an efficient manner.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> a = GrayCode(4)
>>> list(a.generate_gray())
['0000', '0001', '0011', '0010', '0110', '0111', '0101', '0100',
↪'1100', '1101', '1111', '1110', '1010', '1011', '1001', '1000']
```

References

[R39], [R40]

property current

Returns the currently referenced Gray code as a bit string.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> GrayCode(3, start='100').current
'100'
```

generate_gray(***hints*)

Generates the sequence of bit vectors of a Gray Code.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> list(a.generate_gray(start='011'))
['011', '010', '110', '111', '101', '100']
>>> list(a.generate_gray(rank=4))
['110', '111', '101', '100']
```

See also:

skip (page 349)

References

[R41]

property n

Returns the dimension of the Gray code.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(5)
>>> a.n
5
```

next(*delta=1*)

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

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(3, start='110')
>>> a.next().current
'111'
>>> a.next(-1).current
'010'
```

property rank

Ranks the Gray code.

A ranking algorithm determines the position (or rank) of a combinatorial object among all the objects w.r.t. a given order. For example, the 4 bit binary reflected Gray code (BRGC) '0101' has a rank of 6 as it appears in the 6th position in the canonical ordering of the family of 4 bit Gray codes.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> GrayCode(3, start='100').rank
7
>>> GrayCode(3, rank=7).current
'100'
```

See also:

unrank (page 350)

References

[R42]

property selections

Returns the number of bit vectors in the Gray code.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(3)
>>> a.selections
8
```

skip()

Skips the bit generation.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> a = GrayCode(3)
>>> for i in a.generate_gray():
...     if i == '010':
...         a.skip()
...     print(i)
...
000
001
011
010
111
101
100
```

See also:

generate_gray (page 348)

classmethod 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.

The string here is generated in reverse order to allow for tail-call optimization.

Examples

```
>>> from sympy.combinatorics import GrayCode
>>> GrayCode(5, rank=3).current
'00010'
>>> GrayCode.unrank(5, 3)
'00010'
```

See also:

rank (page 349)

graycode.**random_bitstring**()

Generates a random bitlist of length n.

Examples

```
>>> from sympy.combinatorics.graycode import random_bitstring
>>> random_bitstring(3)
100
```

graycode.**gray_to_bin**()

Convert from Gray coding to binary coding.

We assume big endian encoding.

Examples

```
>>> from sympy.combinatorics.graycode import gray_to_bin
>>> gray_to_bin('100')
'111'
```

See also:

bin_to_gray (page 351)

graycode.**bin_to_gray**()

Convert from binary coding to gray coding.

We assume big endian encoding.

Examples

```
>>> from sympy.combinatorics.graycode import bin_to_gray
>>> bin_to_gray('111')
'100'
```

See also:

gray_to_bin (page 350)

graycode.**get_subset_from_bitstring**(*bitstring*)

Gets the subset defined by the bitstring.

Examples

```
>>> from sympy.combinatorics.graycode import get_subset_from_bitstring
>>> get_subset_from_bitstring(['a', 'b', 'c', 'd'], '0011')
['c', 'd']
>>> get_subset_from_bitstring(['c', 'a', 'c', 'c'], '1100')
['c', 'a']
```

See also:

graycode_subsets (page 351)

graycode.**graycode_subsets**()

Generates the subsets as enumerated by a Gray code.

Examples

```
>>> from sympy.combinatorics.graycode import graycode_subsets
>>> list(graycode_subsets(['a', 'b', 'c']))
[[], ['c'], ['b', 'c'], ['b'], ['a', 'b'], ['a', 'b', 'c'],     ['a', 'c
↪'], ['a']]
>>> list(graycode_subsets(['a', 'b', 'c', 'c']))
[[], ['c'], ['c', 'c'], ['c'], ['b', 'c'], ['b', 'c', 'c'],     ['b', 'c
```

(continues on next page)

(continued from previous page)

```
↪'], ['b'], ['a', 'b'], ['a', 'b', 'c'], ['a', 'b', 'c', 'c'],     ['a',
↪ 'b', 'c'], ['a', 'c'], ['a', 'c', 'c'], ['a', 'c'], ['a']]
```

See also:

get_subset_from_bitstring (page 351)

Named Groups

sympy.combinatorics.named_groups.**SymmetricGroup**(*n*)

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

Explanation

The generators taken are the n-cycle `(0 1 2 ... n-1)` and the transposition `(0 1)` (in cycle notation). (See [1]). After the group is generated, some of its basic properties are set.

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(4)
>>> G.is_group
True
>>> G.order()
24
>>> list(G.generate_schreier_sims(af=True))
[[0, 1, 2, 3], [1, 2, 3, 0], [2, 3, 0, 1], [3, 1, 2, 0], [0, 2, 3, 1],
[1, 3, 0, 2], [2, 0, 1, 3], [3, 2, 0, 1], [0, 3, 1, 2], [1, 0, 2, 3],
[2, 1, 3, 0], [3, 0, 1, 2], [0, 1, 3, 2], [1, 2, 0, 3], [2, 3, 1, 0],
[3, 1, 0, 2], [0, 2, 1, 3], [1, 3, 2, 0], [2, 0, 3, 1], [3, 2, 1, 0],
[0, 3, 2, 1], [1, 0, 3, 2], [2, 1, 0, 3], [3, 0, 2, 1]]
```

See also:

CyclicGroup (page 352), *DihedralGroup* (page 353), *AlternatingGroup* (page 354)

References

[R46]

sympy.combinatorics.named_groups.**CyclicGroup**(*n*)

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

Explanation

The generator taken is the n-cycle `(0 1 2 ... n-1)` (in cycle notation). After the group is generated, some of its basic properties are set.

Examples

```
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> G = CyclicGroup(6)
>>> G.is_group
True
>>> G.order()
6
>>> list(G.generate_schreier_sims(af=True))
[[0, 1, 2, 3, 4, 5], [1, 2, 3, 4, 5, 0], [2, 3, 4, 5, 0, 1],
[3, 4, 5, 0, 1, 2], [4, 5, 0, 1, 2, 3], [5, 0, 1, 2, 3, 4]]
```

See also:

SymmetricGroup (page 352), *DihedralGroup* (page 353), *AlternatingGroup* (page 354)

sympy.combinatorics.named_groups.**DihedralGroup**(*n*)

Generates the dihedral group D_n as a permutation group.

Explanation

The dihedral group D_n is the group of symmetries of the regular n-gon. The generators taken are the n-cycle `a = (0 1 2 ... n-1)` (a rotation of the n-gon) and `b = (0 n-1)(1 n-2)...` (a reflection of the n-gon) in cycle rotation. It is easy to see that these satisfy `a**n = b**2 = 1` and `bab = ~a` so they indeed generate D_n (See [1]). After the group is generated, some of its basic properties are set.

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(5)
>>> G.is_group
True
>>> a = list(G.generate_dimino())
>>> [perm.cyclic_form for perm in a]
[[], [[0, 1, 2, 3, 4]], [[0, 2, 4, 1, 3]],
[[0, 3, 1, 4, 2]], [[0, 4, 3, 2, 1]], [[0, 4], [1, 3]],
[[1, 4], [2, 3]], [[0, 1], [2, 4]], [[0, 2], [3, 4]],
[[0, 3], [1, 2]]]
```

See also:

SymmetricGroup (page 352), *CyclicGroup* (page 352), *AlternatingGroup* (page 354)

References

[R47]

sympy.combinatorics.named_groups.**AlternatingGroup**(*n*)

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

Explanation

For `n > 2`, the generators taken are `(0 1 2), (0 1 2 ... n-1)` for n odd and `(0 1 2), (1 2 ... n-1)` for n even (See [1], p.31, ex.6.9.). After the group is generated, some of its basic properties are set. The cases `n = 1, 2` are handled separately.

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> G = AlternatingGroup(4)
>>> G.is_group
True
>>> a = list(G.generate_dimino())
>>> len(a)
12
>>> all(perm.is_even for perm in a)
True
```

See also:

SymmetricGroup (page 352), *CyclicGroup* (page 352), *DihedralGroup* (page 353)

References

[R48]

sympy.combinatorics.named_groups.**AbelianGroup**(**cyclic_orders*)

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

Explanation

According to the structure theorem for finite abelian groups ([1]), every finite abelian group can be written as the direct product of finitely many cyclic groups.

Examples

```
>>> from sympy.combinatorics.named_groups import AbelianGroup
>>> AbelianGroup(3, 4)
PermutationGroup([
        (6)(0 1 2),
        (3 4 5 6)])
>>> _.is_group
True
```

See also:

DirectProduct (page 363)

References

[R49]

Nilpotent, Abelian and Cyclic Numbers

sympy.combinatorics.group_numbers.**is_nilpotent_number**(*n*)

Check whether n is a nilpotent number. A number n is said to be nilpotent if and only if every finite group of order n is nilpotent. For more information see [R43].

Examples

```
>>> from sympy.combinatorics.group_numbers import is_nilpotent_number
>>> from sympy import randprime
>>> is_nilpotent_number(21)
False
>>> is_nilpotent_number(randprime(1, 30)**12)
True
```

References

[R43]

sympy.combinatorics.group_numbers.**is_abelian_number**(*n*)

Check whether n is an abelian number. A number n is said to be abelian if and only if every finite group of order n is abelian. For more information see [R44].

Examples

```
>>> from sympy.combinatorics.group_numbers import is_abelian_number
>>> from sympy import randprime
>>> is_abelian_number(4)
True
>>> is_abelian_number(randprime(1, 2000)**2)
True
>>> is_abelian_number(60)
False
```

References

[R44]

sympy.combinatorics.group_numbers.**is_cyclic_number**(*n*)

Check whether n is a cyclic number. A number n is said to be cyclic if and only if every finite group of order n is cyclic. For more information see [R45].

Examples

```
>>> from sympy.combinatorics.group_numbers import is_cyclic_number
>>> from sympy import randprime
>>> is_cyclic_number(15)
True
>>> is_cyclic_number(randprime(1, 2000)**2)
False
>>> is_cyclic_number(4)
False
```

References

[R45]

Utilities

sympy.combinatorics.util.**_base_ordering**(*base, degree*)

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

Parameters

``**base**`` : the base

``**degree**`` : the degree of the associated permutation group

Returns

A list `base_ordering` such that `base_ordering[point]` is the number of `point` in the ordering.