Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation([2, 0, 3, 1]); p.rank_nonlex()
5
>>> p = p.next_nonlex(); p
Permutation([3, 0, 1, 2])
>>> p.rank_nonlex()
6
```

See also:

rank_nonlex (page 279), *unrank_nonlex* (page 284)

next_trotterjohnson()

Returns the next permutation in Trotter-Johnson order. If self is the last permutation it returns None. See [4] section 2.4. If it is desired to generate all such permutations, they can be generated in order more quickly with the `generate_bell` function.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation([3, 0, 2, 1])
>>> p.rank_trotterjohnson()
4
>>> p = p.next_trotterjohnson(); p
Permutation([0, 3, 2, 1])
>>> p.rank_trotterjohnson()
5
```

See also:

rank_trotterjohnson (page 279), *unrank_trotterjohnson* (page 284), *sympy.utilities.iterables.generate_bell* (page 2074)

order()

Computes the order of a permutation.

When the permutation is raised to the power of its order it equals the identity permutation.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation([3, 1, 5, 2, 4, 0])
>>> p.order()
4
>>> (p**(p.order()))
Permutation([], size=6)
```

See also:

identity (page 309), *cardinality* (page 265), *length* (page 275), *rank* (page 279), *size* (page 282)

parity()

Computes the parity of a permutation.

Explanation

The parity of a permutation reflects the parity of the number of inversions in the permutation, i.e., the number of pairs of x and y such that `x > y` but `p[x] < p[y]`.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.parity()
0
>>> p = Permutation([3, 2, 0, 1])
>>> p.parity()
1
```

See also:

_af_parity (page 286)

classmethod random(*n*)

Generates a random permutation of length `n`.

Uses the underlying Python pseudo-random number generator.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.random(2) in (Permutation([1, 0]), Permutation([0,␣
↪1]))
True
```

rank()

Returns the lexicographic rank of the permutation.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank()
0
>>> p = Permutation([3, 2, 1, 0])
>>> p.rank()
23
```

See also:

next_lex (page 276), *unrank_lex* (page 283), *cardinality* (page 265), *length* (page 275), *order* (page 277), *size* (page 282)

rank_nonlex(*inv_perm=None*)

This is a linear time ranking algorithm that does not enforce lexicographic order [3].

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank_nonlex()
23
```

See also:

next_nonlex (page 276), *unrank_nonlex* (page 284)

rank_trotterjohnson()

Returns the Trotter Johnson rank, which we get from the minimal change algorithm. See [4] section 2.4.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank_trotterjohnson()
0
>>> p = Permutation([0, 2, 1, 3])
>>> p.rank_trotterjohnson()
7
```

See also:

unrank_trotterjohnson (page 284), *next_trotterjohnson* (page 277)

resize(*n*)

Resize the permutation to the new size n.

Parameters

n : int

The new size of the permutation.

Raises

ValueError

If the permutation cannot be resized to the given size. This may only happen when resized to a smaller size than the original.

Examples

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

Increasing the size of a permutation:

```
>>> p = Permutation(0, 1, 2)
>>> p = p.resize(5)
>>> p
(4)(0 1 2)
```

Decreasing the size of the permutation:

```
>>> p = p.resize(4)
>>> p
(3)(0 1 2)
```

If resizing to the specific size breaks the cycles:

```
>>> p.resize(2)
Traceback (most recent call last):
...
ValueError: The permutation cannot be resized to 2 because the
cycle (0, 1, 2) may break.
```

static rmul(**args*)

Return product of Permutations [a, b, c, ...] as the Permutation whose ith value is a(b(c(i))).

a, b, c, ... can be Permutation objects or tuples.

Examples

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

```
>>> a, b = [1, 0, 2], [0, 2, 1]
>>> a = Permutation(a); b = Permutation(b)
>>> list(Permutation.rmul(a, b))
[1, 2, 0]
>>> [a(b(i)) for i in range(3)]
[1, 2, 0]
```

This handles the operands in reverse order compared to the * operator:

```
>>> a = Permutation(a); b = Permutation(b)
>>> list(a*b)
[2, 0, 1]
>>> [b(a(i)) for i in range(3)]
[2, 0, 1]
```

Notes

All items in the sequence will be parsed by Permutation as necessary as long as the first item is a Permutation:

```
>>> Permutation.rmul(a, [0, 2, 1]) == Permutation.rmul(a, b)
True
```

The reverse order of arguments will raise a TypeError.

classmethod rmul_with_af(**args*)

same as rmul, but the elements of args are Permutation objects which have _array_form

runs()

Returns the runs of a permutation.

An ascending sequence in a permutation is called a run [5].

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([2, 5, 7, 3, 6, 0, 1, 4, 8])
>>> p.runs()
[[2, 5, 7], [3, 6], [0, 1, 4, 8]]
>>> q = Permutation([1,3,2,0])
>>> q.runs()
[[1, 3], [2], [0]]
```

signature()

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

The signature is calculated as (-1)^<number of inversions>

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2])
>>> p.inversions()
0
>>> p.signature()
1
>>> q = Permutation([0,2,1])
>>> q.inversions()
1
>>> q.signature()
-1
```


See also:

inversions (page 272)

property size

Returns the number of elements in the permutation.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([[3, 2], [0, 1]]).size
4
```

See also:

cardinality (page 265), *length* (page 275), *order* (page 277), *rank* (page 279)

support()

Return the elements in permutation, P, for which P[i] != i.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([[3, 2], [0, 1], [4]])
>>> p.array_form
[1, 0, 3, 2, 4]
>>> p.support()
[0, 1, 2, 3]
```

transpositions()

Return the permutation decomposed into a list of transpositions.

Explanation

It is always possible to express a permutation as the product of transpositions, see [1]

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([[1, 2, 3], [0, 4, 5, 6, 7]])
>>> t = p.transpositions()
>>> t
[(0, 7), (0, 6), (0, 5), (0, 4), (1, 3), (1, 2)]
>>> print(''.join(str(c) for c in t))
(0, 7)(0, 6)(0, 5)(0, 4)(1, 3)(1, 2)
>>> Permutation.rmul(*[Permutation([ti], size=p.size) for ti in t])␣
↪== p
True
```

References

[R80]

classmethod unrank_lex(*size*, *rank*)

Lexicographic permutation unranking.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> a = Permutation.unrank_lex(5, 10)
>>> a.rank()
10
>>> a
Permutation([0, 2, 4, 1, 3])
```

See also:

rank (page 279), *next_lex* (page 276)

classmethod unrank_nonlex(*n*, *r*)

This is a linear time unranking algorithm that does not respect lexicographic order [3].

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> Permutation.unrank_nonlex(4, 5)
Permutation([2, 0, 3, 1])
>>> Permutation.unrank_nonlex(4, -1)
Permutation([0, 1, 2, 3])
```

See also:

next_nonlex (page 276), *rank_nonlex* (page 279)

classmethod unrank_trotterjohnson(*size*, *rank*)

Trotter Johnson permutation unranking. See [4] section 2.4.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> Permutation.unrank_trotterjohnson(5, 10)
Permutation([0, 3, 1, 2, 4])
```

See also:

rank_trotterjohnson (page 279), *next_trotterjohnson* (page 277)

class sympy.combinatorics.permutations.**Cycle**(**args*)

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

Explanation

A cycle shows the rule to use to move subsets of elements to obtain a permutation. The Cycle class is more flexible than Permutation in that 1) all elements need not be present in order to investigate how multiple cycles act in sequence and 2) it can contain singletons:

```
>>> from sympy.combinatorics.permutations import Perm, Cycle
```

A Cycle will automatically parse a cycle given as a tuple on the rhs:

```
>>> Cycle(1, 2)(2, 3)
(1 3 2)
```

The identity cycle, Cycle(), can be used to start a product:

```
>>> Cycle()(1, 2)(2, 3)
(1 3 2)
```

The array form of a Cycle can be obtained by calling the list method (or passing it to the list function) and all elements from 0 will be shown:

```
>>> a = Cycle(1, 2)
>>> a.list()
[0, 2, 1]
>>> list(a)
[0, 2, 1]
```

If a larger (or smaller) range is desired use the list method and provide the desired size – but the Cycle cannot be truncated to a size smaller than the largest element that is out of place:

```
>>> b = Cycle(2, 4)(1, 2)(3, 1, 4)(1, 3)
>>> b.list()
[0, 2, 1, 3, 4]
>>> b.list(b.size + 1)
[0, 2, 1, 3, 4, 5]
>>> b.list(-1)
[0, 2, 1]
```

Singletons are not shown when printing with one exception: the largest element is always shown – as a singleton if necessary:

```
>>> Cycle(1, 4, 10)(4, 5)
(1 5 4 10)
>>> Cycle(1, 2)(4)(5)(10)
(1 2)(10)
```

The array form can be used to instantiate a Permutation so other properties of the permutation can be investigated:

```
>>> Perm(Cycle(1, 2)(3, 4).list()).transpositions()
[(1, 2), (3, 4)]
```

Notes

The underlying structure of the Cycle is a dictionary and although the __iter__ method has been redefined to give the array form of the cycle, the underlying dictionary items are still available with the such methods as items():

```
>>> list(Cycle(1, 2).items())
[(1, 2), (2, 1)]
```

See also:

Permutation (page 257)

list(*size=None*)

Return the cycles as an explicit list starting from 0 up to the greater of the largest value in the cycles and size.

Truncation of trailing unmoved items will occur when size is less than the maximum element in the cycle; if this is desired, setting `size=-1` will guarantee such trimming.

Examples

```
>>> from sympy.combinatorics import Cycle
>>> p = Cycle(2, 3)(4, 5)
>>> p.list()
[0, 1, 3, 2, 5, 4]
>>> p.list(10)
[0, 1, 3, 2, 5, 4, 6, 7, 8, 9]
```

Passing a length too small will trim trailing, unchanged elements in the permutation:

```
>>> Cycle(2, 4)(1, 2, 4).list(-1)
[0, 2, 1]
```

sympy.combinatorics.permutations.**_af_parity**(*pi*)

Computes the parity of a permutation in array form.

Explanation

The parity of a permutation reflects the parity of the number of inversions in the permutation, i.e., the number of pairs of x and y such that x > y but p[x] < p[y].

Examples

```
>>> from sympy.combinatorics.permutations import _af_parity
>>> _af_parity([0, 1, 2, 3])
0
>>> _af_parity([3, 2, 0, 1])
1
```

See also:

Permutation (page 257)

Generators

generators.**symmetric**()

Generates the symmetric group of order n, Sn.

Examples

```
>>> from sympy.combinatorics.generators import symmetric
>>> list(symmetric(3))
[(2), (1 2), (2)(0 1), (0 1 2), (0 2 1), (0 2)]
```

generators.**cyclic**()

Generates the cyclic group of order n, Cn.

Examples

```
>>> from sympy.combinatorics.generators import cyclic
>>> list(cyclic(5))
[(4), (0 1 2 3 4), (0 2 4 1 3),
 (0 3 1 4 2), (0 4 3 2 1)]
```

See also:

dihedral (page 287)

generators.**alternating**()

Generates the alternating group of order n, An.

Examples

```
>>> from sympy.combinatorics.generators import alternating
>>> list(alternating(3))
[(2), (0 1 2), (0 2 1)]
```

generators.**dihedral**()

Generates the dihedral group of order 2n, Dn.

The result is given as a subgroup of Sn, except for the special cases n=1 (the group S2) and n=2 (the Klein 4-group) where that's not possible and embeddings in S2 and S4 respectively are given.

Examples

```
>>> from sympy.combinatorics.generators import dihedral
>>> list(dihedral(3))
[(2), (0 2), (0 1 2), (1 2), (0 2 1), (2)(0 1)]
```

See also:

cyclic (page 287)

Permutation Groups

class sympy.combinatorics.perm_groups.**PermutationGroup**(**args*, *dups=True*, ***kwargs*)

The class defining a Permutation group.

Explanation

`PermutationGroup([p1, p2, ..., pn])` returns the permutation group generated by the list of permutations. This group can be supplied to Polyhedron if one desires to decorate the elements to which the indices of the permutation refer.

Examples

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

The permutations corresponding to motion of the front, right and bottom face of a 2×2 Rubik's cube are defined:

```
>>> F = Permutation(2, 19, 21, 8)(3, 17, 20, 10)(4, 6, 7, 5)
>>> R = Permutation(1, 5, 21, 14)(3, 7, 23, 12)(8, 10, 11, 9)
>>> D = Permutation(6, 18, 14, 10)(7, 19, 15, 11)(20, 22, 23, 21)
```

These are passed as permutations to PermutationGroup:

```
>>> G = PermutationGroup(F, R, D)
>>> G.order()
3674160
```

The group can be supplied to a Polyhedron in order to track the objects being moved. An example involving the 2×2 Rubik's cube is given there, but here is a simple demonstration:

```
>>> a = Permutation(2, 1)
>>> b = Permutation(1, 0)
>>> G = PermutationGroup(a, b)
>>> P = Polyhedron(list('ABC'), pgroup=G)
>>> P.corners
(A, B, C)
```

(continues on next page)

(continued from previous page)

```
>>> P.rotate(0) # apply permutation 0
>>> P.corners
(A, C, B)
>>> P.reset()
>>> P.corners
(A, B, C)
```

Or one can make a permutation as a product of selected permutations and apply them to an iterable directly:

```
>>> P10 = G.make_perm([0, 1])
>>> P10('ABC')
['C', 'A', 'B']
```

See also:

sympy.combinatorics.polyhedron.Polyhedron (page 332), *sympy.combinatorics.permutations.Permutation* (page 257)

References

[R51], [R52], [R53], [R54], [R55], [R56], [R57], [R58], [R59],[10],[11],[12],[13],[14]

__contains__(*i*)

Return True if *i* is contained in PermutationGroup.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> p = Permutation(1, 2, 3)
>>> Permutation(3) in PermutationGroup(p)
True
```

__mul__(*other*)

Return the direct product of two permutation groups as a permutation group.

[10] https://en.wikipedia.org/wiki/Centralizer_and_normalizer
[11] http://groupprops.subwiki.org/wiki/Derived_subgroup
[12] https://en.wikipedia.org/wiki/Nilpotent_group
[13] http://www.math.colostate.edu/~hulpke/CGT/cgtnotes.pdf
[14] https://www.gap-system.org/Manuals/doc/ref/manual.pdf

Explanation

This implementation realizes the direct product by shifting the index set for the generators of the second group: so if we have `G` acting on `n1` points and `H` acting on `n2` points, `G*H` acts on `n1 + n2` points.

Examples

```
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> G = CyclicGroup(5)
>>> H = G*G
>>> H
PermutationGroup([
    (9)(0 1 2 3 4),
    (5 6 7 8 9)])
>>> H.order()
25
```

static __new__(*cls*, **args*, *dups=True*, ***kwargs*)

The default constructor. Accepts Cycle and Permutation forms. Removes duplicates unless `dups` keyword is `False`.

__weakref__

list of weak references to the object (if defined)

_coset_representative(*g*, *H*)

Return the representative of Hg from the transversal that would be computed by `self.coset_transversal(H)`.

classmethod _distinct_primes_lemma(*primes*)

Subroutine to test if there is only one cyclic group for the order.

property _elements

Returns all the elements of the permutation group as a list

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> p = PermutationGroup(Permutation(1, 3), Permutation(1, 2))
>>> p._elements
[(3), (3)(1 2), (1 3), (2 3), (1 2 3), (1 3 2)]
```

_eval_is_alt_sym_monte_carlo(*eps=0.05*, *perms=None*)

A test using monte-carlo algorithm.

Parameters

eps : float, optional

The criterion for the incorrect `False` return.

perms : list[Permutation], optional

If explicitly given, it tests over the given candidats for testing.

If None, it randomly computes N_eps and chooses N_eps sample of the permutation from the group.

See also:

_check_cycles_alt_sym (page 357)

_eval_is_alt_sym_naive(*only_sym=False*, *only_alt=False*)

A naive test using the group order.

_p_elements_group(*p*)

For an abelian p-group, return the subgroup consisting of all elements of order p (and the identity)

_random_pr_init(*r*, *n*, *_random_prec_n=None*)

Initialize random generators for the product replacement algorithm.

Explanation

The implementation uses a modification of the original product replacement algorithm due to Leedham-Green, as described in [1], pp. 69-71; also, see [2], pp. 27-29 for a detailed theoretical analysis of the original product replacement algorithm, and [4].

The product replacement algorithm is used for producing random, uniformly distributed elements of a group G with a set of generators S. For the initialization `_random_pr_init`, a list `R` of $\max\{r, |S|\}$ group generators is created as the attribute `G._random_gens`, repeating elements of S if necessary, and the identity element of G is appended to `R` - we shall refer to this last element as the accumulator. Then the function `random_pr()` is called `n` times, randomizing the list `R` while preserving the generation of G by `R`. The function `random_pr()` itself takes two random elements `g, h` among all elements of `R` but the accumulator and replaces `g` with a randomly chosen element from $\{gh, g(\ h), hg, (\ h)g\}$. Then the accumulator is multiplied by whatever `g` was replaced by. The new value of the accumulator is then returned by `random_pr()`.

The elements returned will eventually (for `n` large enough) become uniformly distributed across G ([5]). For practical purposes however, the values `n = 50, r = 11` are suggested in [1].

Notes

THIS FUNCTION HAS SIDE EFFECTS: it changes the attribute self._random_gens

See also:

random_pr (page 324)

_sylow_alt_sym(*p*)

Return a p-Sylow subgroup of a symmetric or an alternating group.

Explanation

The algorithm for this is hinted at in [1], Chapter 4, Exercise 4.

For Sym(n) with n = p^i, the idea is as follows. Partition the interval [0..n-1] into p equal parts, each of length p^(i-1): [0..p^(i-1)-1], [p^(i-1)..2*p^(i-1)-1]...[(p-1)*p^(i-1)..p^i-1]. Find a p-Sylow subgroup of Sym(p^(i-1)) (treated as a subgroup of `self`) acting on each of the parts. Call the subgroups P_1, P_2...P_p. The generators for the subgroups P_2...P_p can be obtained from those of P_1 by applying a "shifting" permutation to them, that is, a permutation mapping [0..p^(i-1)-1] to the second part (the other parts are obtained by using the shift multiple times). The union of this permutation and the generators of P_1 is a p-Sylow subgroup of `self`.

For n not equal to a power of p, partition [0..n-1] in accordance with how n would be written in base p. E.g. for p=2 and n=11, 11 = 2^3 + 2^2 + 1 so the partition is [[0..7], [8..9], {10}]. To generate a p-Sylow subgroup, take the union of the generators for each of the parts. For the above example, {(0 1), (0 2)(1 3), (0 4), (1 5)(2 7)} from the first part, {(8 9)} from the second part and nothing from the third. This gives 4 generators in total, and the subgroup they generate is p-Sylow.

Alternating groups are treated the same except when p=2. In this case, (0 1)(s s+1) should be added for an appropriate s (the start of a part) for each part in the partitions.

See also:

sylow_subgroup (page 330), *is_alt_sym* (page 310)

_union_find_merge(*first*, *second*, *ranks*, *parents*, *not_rep*)

Merges two classes in a union-find data structure.

Explanation

Used in the implementation of Atkinson's algorithm as suggested in [1], pp. 83-87. The class merging process uses union by rank as an optimization. ([7])

Notes

THIS FUNCTION HAS SIDE EFFECTS: the list of class representatives, `parents`, the list of class sizes, `ranks`, and the list of elements that are not representatives, `not_rep`, are changed due to class merging.

See also:

minimal_block (page 319), *_union_find_rep* (page 293)

References

[R60], [R66]

_union_find_rep(*num*, *parents*)

Find representative of a class in a union-find data structure.

Explanation

Used in the implementation of Atkinson's algorithm as suggested in [1], pp. 83-87. After the representative of the class to which `num` belongs is found, path compression is performed as an optimization ([7]).

Notes

THIS FUNCTION HAS SIDE EFFECTS: the list of class representatives, `parents`, is altered due to path compression.

See also:

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

References

[R62], [R68]

_verify(*K*, *phi*, *z*, *alpha*)

Return a list of relators `rels` in generators `gens`_h` that are mapped to ``H.generators` by `phi` so that given a finite presentation <gens_k | rels_k> of `K` on a subset of `gens_h` <gens_h | rels_k + rels> is a finite presentation of `H`.

Explanation

`H` should be generated by the union of `K.generators` and `z` (a single generator), and `H.stabilizer(alpha) == K`; `phi` is a canonical injection from a free group into a permutation group containing `H`.

The algorithm is described in [1], Chapter 6.

Examples

```
>>> from sympy.combinatorics import free_group, Permutation,␣
↪PermutationGroup
>>> from sympy.combinatorics.homomorphisms import homomorphism
>>> from sympy.combinatorics.fp_groups import FpGroup
```

```
>>> H = PermutationGroup(Permutation(0, 2), Permutation (1, 5))
>>> K = PermutationGroup(Permutation(5)(0, 2))
>>> F = free_group("x_0 x_1")[0]
>>> gens = F.generators
>>> phi = homomorphism(F, H, F.generators, H.generators)
>>> rels_k = [gens[0]**2] # relators for presentation of K
>>> z= Permutation(1, 5)
>>> check, rels_h = H._verify(K, phi, z, 1)
>>> check
True
>>> rels = rels_k + rels_h
>>> G = FpGroup(F, rels) # presentation of H
>>> G.order() == H.order()
True
```

See also:

strong_presentation (page 329), *presentation* (page 324), *stabilizer* (page 328)

abelian_invariants()

Returns the abelian invariants for the given group. Let G be a nontrivial finite abelian group. Then G is isomorphic to the direct product of finitely many nontrivial cyclic groups of prime-power order.

Explanation

The prime-powers that occur as the orders of the factors are uniquely determined by G. More precisely, the primes that occur in the orders of the factors in any such decomposition of G are exactly the primes that divide |G| and for any such prime p, if the orders of the factors that are p-groups in one such decomposition of G are p^{t_1} >= p^{t_2} >= ... p^{t_r}, then the orders of the factors that are p-groups in any such decomposition of G are p^{t_1} >= p^{t_2} >= ... p^{t_r}.

The uniquely determined integers p^{t_1} >= p^{t_2} >= ... p^{t_r}, taken for all primes that divide |G| are called the invariants of the nontrivial group G as suggested in ([14], p. 542).

Notes

We adopt the convention that the invariants of a trivial group are [].

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.abelian_invariants()
[2]
>>> from sympy.combinatorics import CyclicGroup
>>> G = CyclicGroup(7)
>>> G.abelian_invariants()
[7]
```

property base

Return a base from the Schreier-Sims algorithm.

Explanation

For a permutation group G, a base is a sequence of points $B = (b_1, b_2, \ldots, b_k)$ such that no element of G apart from the identity fixes all the points in B. 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.

An alternative way to think of B is that it gives the indices of the stabilizer cosets that contain more than the identity permutation.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> G = PermutationGroup([Permutation(0, 1, 3)(2, 4)])
>>> G.base
[0, 2]
```

See also:

strong_gens (page 328), *basic_transversals* (page 298), *basic_orbits* (page 297), *basic_stabilizers* (page 297)

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.

Parameters

base, strong_gens

The base and strong generating set.

pos

The position at which swapping is performed.

randomized

A switch between randomized and deterministic version.

transversals

The transversals for the basic orbits, if known.

basic_orbits

The basic orbits, if known.

strong_gens_distr

The strong generators distributed by basic stabilizers, if known.

Returns

(base, strong_gens)

`base` is the new base, and `strong_gens` is a generating set relative to it.

Explanation

If a base for a group G is given by $(b_1, b_2, \ldots, b_k)$, this function returns a base $(b_1, b_2, \ldots, b_{i+1}, b_i, \ldots, b_k)$, where i is given by `pos`, and a strong generating set relative to that base. The original base and strong generating set are not modified.

The randomized version (default) is of Las Vegas type.

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> S = SymmetricGroup(4)
>>> S.schreier_sims()
>>> S.base
[0, 1, 2]
>>> base, gens = S.baseswap(S.base, S.strong_gens, 1,␣
↪randomized=False)
>>> base, gens
([0, 2, 1],
[(0 1 2 3), (3)(0 1), (1 3 2),
 (2 3), (1 3)])
```

check that base, gens is a BSGS

```
>>> S1 = PermutationGroup(gens)
>>> _verify_bsgs(S1, base, gens)
True
```

Notes

The deterministic version of the algorithm is discussed in [1], pp. 102-103; the randomized version is discussed in [1], p.103, and [2], p.98. It is of Las Vegas type. Notice that [1] contains a mistake in the pseudocode and discussion of BASESWAP: on line 3 of the pseudocode, $|\beta_{i+1}^{\langle T\rangle}|$ should be replaced by $|\beta_i^{\langle T\rangle}|$, and the same for the discussion of the algorithm.

See also:

schreier_sims (page 324)

property basic_orbits

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

Explanation

If $(b_1, b_2, \ldots, b_k)$ is a base for a group G, and $G^{(i)} = G_{b_1,b_2,\ldots,b_{i-1}}$ is the i-th basic stabilizer (so that $G^{(1)} = G$), the i-th basic orbit relative to this base is the orbit of b_i under $G^{(i)}$. See [1], pp. 87-89 for more information.

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(4)
>>> S.basic_orbits
[[0, 1, 2, 3], [1, 2, 3], [2, 3]]
```

See also:

base (page 295), *strong_gens* (page 328), *basic_transversals* (page 298), *basic_stabilizers* (page 297)

property basic_stabilizers

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

Explanation

The i-th basic stabilizer $G^{(i)}$ relative to a base $(b_1, b_2, \ldots, b_k)$ is $G_{b_1,b_2,\ldots,b_{i-1}}$. For more information, see [1], pp. 87-89.

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> A = AlternatingGroup(4)
>>> A.schreier_sims()
>>> A.base
[0, 1]
>>> for g in A.basic_stabilizers:
...     print(g)
```

(continues on next page)

(continued from previous page)

```
...
PermutationGroup([
    (3)(0 1 2),
    (1 2 3)])
PermutationGroup([
    (1 2 3)])
```

See also:

base (page 295), *strong_gens* (page 328), *basic_orbits* (page 297), *basic_transversals* (page 298)

property basic_transversals

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

Explanation

The basic transversals are transversals of the basic orbits. They are provided as a list of dictionaries, each dictionary having keys - the elements of one of the basic orbits, and values - the corresponding transversal elements. See [1], pp. 87-89 for more information.

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> A = AlternatingGroup(4)
>>> A.basic_transversals
[{0: (3), 1: (3)(0 1 2), 2: (3)(0 2 1), 3: (0 3 1)}, {1: (3), 2: (1 2␣
↪3), 3: (1 3 2)}]
```

See also:

strong_gens (page 328), *base* (page 295), *basic_orbits* (page 297), *basic_stabilizers* (page 297)

center()

Return the center of a permutation group.

Explanation

The center for a group G is defined as $Z(G) = \{z \in G | \forall g \in G, zg = gz\}$, the set of elements of G that commute with all elements of G. It is equal to the centralizer of G inside G, and is naturally a subgroup of G ([9]).

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(4)
>>> G = D.center()
>>> G.order()
2
```

Notes

This is a naive implementation that is a straightforward application of `.centralizer()`

See also:

centralizer (page 299)

centralizer(*other*)

Return the centralizer of a group/set/element.

Parameters

other

a permutation group/list of permutations/single permutation

Explanation

The centralizer of a set of permutations `S` inside a group `G` is the set of elements of `G` that commute with all elements of `S`:

```
`C_G(S) = \{ g \in G | gs = sg \forall s \in S\}` ([10])
```

Usually, `S` is a subset of `G`, but if `G` is a proper subgroup of the full symmetric group, we allow for `S` to have elements outside `G`.

It is naturally a subgroup of `G`; the centralizer of a permutation group is equal to the centralizer of any set of generators for that group, since any element commuting with the generators commutes with any product of the generators.

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup)
>>> S = SymmetricGroup(6)
>>> C = CyclicGroup(6)
>>> H = S.centralizer(C)
>>> H.is_subgroup(C)
True
```

Notes

The implementation is an application of `.subgroup_search()` with tests using a specific base for the group `G`.

See also:

subgroup_search (page 329)

commutator(*G*, *H*)

Return the commutator of two subgroups.

Explanation

For a permutation group `K` and subgroups `G`, `H`, the commutator of `G` and `H` is defined as the group generated by all the commutators $[g, h] = hgh^{-1}g^{-1}$ for `g` in `G` and `h` in `H`. It is naturally a subgroup of `K` ([1], p.27).

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> S = SymmetricGroup(5)
>>> A = AlternatingGroup(5)
>>> G = S.commutator(S, A)
>>> G.is_subgroup(A)
True
```

Notes

The commutator of two subgroups H, G is equal to the normal closure of the commutators of all the generators, i.e. $hgh^{-1}g^{-1}$ for h a generator of H and g a generator of G ([1], p.28)

See also:

derived_subgroup (page 306)

composition_series()

Return the composition series for a group as a list of permutation groups.

Explanation

The composition series for a group G is defined as a subnormal series $G = H_0 > H_1 > H_2 \ldots$ A composition series is a subnormal series such that each factor group $H(i+1)/H(i)$ is simple. A subnormal series is a composition series only if it is of maximum length.

The algorithm works as follows: Starting with the derived series the idea is to fill the gap between $G = der[i]$ and $H = der[i+1]$ for each i independently. Since, all subgroups of the abelian group G/H are normal so, first step is to take the generators g of G and add them to generators of H one by one.

The factor groups formed are not simple in general. Each group is obtained from the previous one by adding one generator g, if the previous group is denoted by H then the next group K is generated by g and H. The factor group K/H is cyclic and it's order is $K.order()//G.order()$. The series is then extended between K and H by groups generated by powers of g and H. The series formed is then prepended to the already existing series.

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> S = SymmetricGroup(12)
>>> G = S.sylow_subgroup(2)
>>> C = G.composition_series()
>>> [H.order() for H in C]
[1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1]
>>> G = S.sylow_subgroup(3)
>>> C = G.composition_series()
>>> [H.order() for H in C]
[243, 81, 27, 9, 3, 1]
>>> G = CyclicGroup(12)
>>> C = G.composition_series()
>>> [H.order() for H in C]
[12, 6, 3, 1]
```

conjugacy_class(x)

Return the conjugacy class of an element in the group.

Explanation

The conjugacy class of an element `g` in a group `G` is the set of elements `x` in `G` that are conjugate with `g`, i.e. for which

```
g = xax^{-1}
```

for some `a` in `G`.

Note that conjugacy is an equivalence relation, and therefore that conjugacy classes are partitions of `G`. For a list of all the conjugacy classes of the group, use the conjugacy_classes() method.

In a permutation group, each conjugacy class corresponds to a particular $cyclestructure'$: $forexample, in``S_3`$, the conjugacy classes are:

- the identity class, `{()}`
- all transpositions, `{(1 2), (1 3), (2 3)}`
- all 3-cycles, `{(1 2 3), (1 3 2)}`

Examples

```
>>> from sympy.combinatorics import Permutation, SymmetricGroup
>>> S3 = SymmetricGroup(3)
>>> S3.conjugacy_class(Permutation(0, 1, 2))
{(0 1 2), (0 2 1)}
```

Notes

This procedure computes the conjugacy class directly by finding the orbit of the element under conjugation in G. This algorithm is only feasible for permutation groups of relatively small order, but is like the orbit() function itself in that respect.

conjugacy_classes()

Return the conjugacy classes of the group.

Explanation

As described in the documentation for the .conjugacy_class() function, conjugacy is an equivalence relation on a group G which partitions the set of elements. This method returns a list of all these conjugacy classes of G.

Examples

```
>>> from sympy.combinatorics import SymmetricGroup
>>> SymmetricGroup(3).conjugacy_classes()
[{(2)}, {(0 1 2), (0 2 1)}, {(0 2), (1 2), (2)(0 1)}]
```

contains(*g*, *strict=True*)

Test if permutation g belong to self, G.

Explanation

If g is an element of G it can be written as a product of factors drawn from the cosets of G's stabilizers. To see if g is one of the actual generators defining the group use G.has(g).

If strict is not True, g will be resized, if necessary, to match the size of permutations in self.

Examples

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

```
>>> a = Permutation(1, 2)
>>> b = Permutation(2, 3, 1)
>>> G = PermutationGroup(a, b, degree=5)
>>> G.contains(G[0]) # trivial check
True
>>> elem = Permutation([[2, 3]], size=5)
>>> G.contains(elem)
True
>>> G.contains(Permutation(4)(0, 1, 2, 3))
False
```

If strict is False, a permutation will be resized, if necessary:

```
>>> H = PermutationGroup(Permutation(5))
>>> H.contains(Permutation(3))
False
>>> H.contains(Permutation(3), strict=False)
True
```

To test if a given permutation is present in the group:

```
>>> elem in G.generators
False
>>> G.has(elem)
False
```

See also:

coset_factor (page 303), *sympy.core.basic.Basic.has* (page 933), *__contains__* (page 289)

coset_factor(*g*, *factor_index=False*)

Return G's (self's) coset factorization of g

Explanation

If g is an element of G then it can be written as the product of permutations drawn from the Schreier-Sims coset decomposition,

The permutations returned in f are those for which the product gives g: `g = f[n]*...f[1]*f[0]` where `n = len(B)` and `B = G.base`. f[i] is one of the permutations in `self._basic_orbits[i]`.

If factor_index==True, returns a tuple `[b[0],..,b[n]]`, where `b[i]` belongs to `self._basic_orbits[i]`

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation(0, 1, 3, 7, 6, 4)(2, 5)
>>> b = Permutation(0, 1, 3, 2)(4, 5, 7, 6)
>>> G = PermutationGroup([a, b])
```

Define g:

```
>>> g = Permutation(7)(1, 2, 4)(3, 6, 5)
```

Confirm that it is an element of G:

```
>>> G.contains(g)
True
```

Thus, it can be written as a product of factors (up to 3) drawn from u. See below that a factor from u1 and u2 and the Identity permutation have been used:

```
>>> f = G.coset_factor(g)
>>> f[2]*f[1]*f[0] == g
True
>>> f1 = G.coset_factor(g, True); f1
[0, 4, 4]
>>> tr = G.basic_transversals
>>> f[0] == tr[0][f1[0]]
True
```

If g is not an element of G then [] is returned:

```
>>> c = Permutation(5, 6, 7)
>>> G.coset_factor(c)
[]
```

See also:

sympy.combinatorics.util._strip (page 361)

coset_rank(*g*)

rank using Schreier-Sims representation.

Explanation

The coset rank of g is the ordering number in which it appears in the lexicographic listing according to the coset decomposition

The ordering is the same as in G.generate(method='coset'). If g does not belong to the group it returns None.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation(0, 1, 3, 7, 6, 4)(2, 5)
>>> b = Permutation(0, 1, 3, 2)(4, 5, 7, 6)
>>> G = PermutationGroup([a, b])
>>> c = Permutation(7)(2, 4)(3, 5)
>>> G.coset_rank(c)
16
>>> G.coset_unrank(16)
(7)(2 4)(3 5)
```

See also:

coset_factor (page 303)

coset_table(*H*)

Return the standardised (right) coset table of self in H as a list of lists.

coset_transversal(*H*)

Return a transversal of the right cosets of self by its subgroup H using the second method described in [1], Subsection 4.6.7

coset_unrank(*rank*, *af=False*)

unrank using Schreier-Sims representation

coset_unrank is the inverse operation of coset_rank if 0 <= rank < order; otherwise it returns None.

property degree

Returns the size of the permutations in the group.

Explanation

The number of permutations comprising the group is given by `len(group)`; the number of permutations that can be generated by the group is given by `group.order()`.

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)]
```

See also:

order (page 322)

derived_series()

Return the derived series for the group.

Returns

A list of permutation groups containing the members of the derived series in the order $G = G_0, G_1, G_2, \ldots$.

Explanation

The derived series for a group G is defined as $G = G_0 > G_1 > G_2 > \ldots$ where $G_i = [G_{i-1}, G_{i-1}]$, i.e. G_i is the derived subgroup of G_{i-1}, for $i \in \mathbb{N}$. When we have $G_k = G_{k-1}$ for some $k \in \mathbb{N}$, the series terminates.

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup, DihedralGroup)
>>> A = AlternatingGroup(5)
>>> len(A.derived_series())
1
>>> S = SymmetricGroup(4)
>>> len(S.derived_series())
4
>>> S.derived_series()[1].is_subgroup(AlternatingGroup(4))
True
>>> S.derived_series()[2].is_subgroup(DihedralGroup(2))
True
```

See also:

derived_subgroup (page 306)

derived_subgroup()

Compute the derived subgroup.

Explanation

The derived subgroup, or commutator subgroup is the subgroup generated by all commutators $[g, h] = hgh^{-1}g^{-1}$ for $g, h \in G$; it is equal to the normal closure of the set of commutators of the generators ([1], p.28, [11]).

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([1, 0, 2, 4, 3])
>>> b = Permutation([0, 1, 3, 2, 4])
>>> G = PermutationGroup([a, b])
>>> C = G.derived_subgroup()
>>> list(C.generate(af=True))
[[0, 1, 2, 3, 4], [0, 1, 3, 4, 2], [0, 1, 4, 2, 3]]
```

See also:

derived_series (page 306)

property elements

Returns all the elements of the permutation group as a set

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> p = PermutationGroup(Permutation(1, 3), Permutation(1, 2))
>>> p.elements
{(1 2 3), (1 3 2), (1 3), (2 3), (3), (3)(1 2)}
```

equals(*other*)

Return True if PermutationGroup generated by elements in the group are same i.e they represent the same PermutationGroup.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> p = Permutation(0, 1, 2, 3, 4, 5)
>>> G = PermutationGroup([p, p**2])
>>> H = PermutationGroup([p**2, p])
>>> G.generators == H.generators
False
>>> G.equals(H)
True
```

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

Return iterator to generate the elements of the group.

Explanation

Iteration is done with one of these methods:

```
method='coset'  using the Schreier-Sims coset representation
method='dimino' using the Dimino method
```

If `af = True` it yields the array form of the permutations

Examples

```
>>> from sympy.combinatorics import PermutationGroup
>>> from sympy.combinatorics.polyhedron import tetrahedron
```

The permutation group given in the tetrahedron object is also true groups:

```
>>> G = tetrahedron.pgroup
>>> G.is_group
True
```

Also the group generated by the permutations in the tetrahedron pgroup – even the first two – is a proper group:

```
>>> H = PermutationGroup(G[0], G[1])
>>> J = PermutationGroup(list(H.generate())); J
PermutationGroup([
    (0 1)(2 3),
    (1 2 3),
    (1 3 2),
    (0 3 1),
    (0 2 3),
    (0 3)(1 2),
    (0 1 3),
    (3)(0 2 1),
    (0 3 2),
    (3)(0 1 2),
    (0 2)(1 3)])
>>> _.is_group
True
```

generate_dimino(*af=False*)

Yield group elements using Dimino's algorithm.

If `af == True` it yields the array form of the permutations.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([0, 2, 3, 1])
>>> g = PermutationGroup([a, b])
>>> list(g.generate_dimino(af=True))
[[0, 1, 2, 3], [0, 2, 1, 3], [0, 2, 3, 1],
 [0, 1, 3, 2], [0, 3, 2, 1], [0, 3, 1, 2]]
```

References

[R64]

generate_schreier_sims(*af=False*)

Yield group elements using the Schreier-Sims representation in coset_rank order

If `af = True` it yields the array form of the permutations

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([0, 2, 3, 1])
>>> g = PermutationGroup([a, b])
>>> list(g.generate_schreier_sims(af=True))
[[0, 1, 2, 3], [0, 2, 1, 3], [0, 3, 2, 1],
 [0, 1, 3, 2], [0, 2, 3, 1], [0, 3, 1, 2]]
```

generator_product(*g*, *original=False*)

Return a list of strong generators $[s1, \ldots, sn]$ s.t $g = sn \times \cdots \times s1$. If `original=True`, make the list contain only the original group generators

property generators

Returns the generators of the group.

Examples

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

property identity

Return the identity element of the permutation group.

index(*H*)

Returns the index of a permutation group.

Examples

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

property is_abelian

Test if the group is Abelian.

Examples

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

is_alt_sym(*eps=0.05, _random_prec=None*)

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

Explanation

More specifically, it is one-sided Monte Carlo with the answer True (i.e., G is symmetric/alternating) guaranteed to be correct, and the answer False being incorrect with probability eps.

For degree < 8, the order of the group is checked so the test is deterministic.

Notes

The algorithm itself uses some nontrivial results from group theory and number theory: 1) If a transitive group `G` of degree `n` contains an element with a cycle of length `n/2 < p < n-2` for `p` a prime, `G` is the symmetric or alternating group ([1], pp. 81-82) 2) The proportion of elements in the symmetric/alternating group having the property described in 1) is approximately $\log(2)/\log(n)$ ([1], p.82; [2], pp. 226-227). The helper function `_check_cycles_alt_sym` is used to go over the cycles in a permutation and look for ones satisfying 1).

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.is_alt_sym()
False
```

See also:

_check_cycles_alt_sym (page 357)

property is_alternating

Return True if the group is alternating.

Examples

```
>>> from sympy.combinatorics import AlternatingGroup
>>> g = AlternatingGroup(5)
>>> g.is_alternating
True
```

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> g = PermutationGroup(
...     Permutation(0, 1, 2, 3, 4),
...     Permutation(2, 3, 4))
>>> g.is_alternating
True
```


Notes

This uses a naive test involving the computation of the full group order. If you need more quicker taxonomy for large groups, you can use *PermutationGroup.is_alt_sym()* (page 310). However, *PermutationGroup.is_alt_sym()* (page 310) may not be accurate and is not able to distinguish between an alternating group and a symmetric group.

See also:

is_alt_sym (page 310)

property is_cyclic

Return True if the group is Cyclic.

Examples

```
>>> from sympy.combinatorics.named_groups import AbelianGroup
>>> G = AbelianGroup(3, 4)
>>> G.is_cyclic
True
>>> G = AbelianGroup(4, 4)
>>> G.is_cyclic
False
```

Notes

If the order of a group n can be factored into the distinct primes $p_1, p_2, \ldots, p_s$ and if

$$\forall i, j \in \{1, 2, \ldots, s\} : p_i \not\equiv 1 \pmod{p_j}$$

holds true, there is only one group of the order n which is a cyclic group [R65]. This is a generalization of the lemma that the group of order $15, 35, \ldots$ are cyclic.

And also, these additional lemmas can be used to test if a group is cyclic if the order of the group is already found.

- If the group is abelian and the order of the group is square-free, the group is cyclic.
- If the order of the group is less than 6 and is not 4, the group is cyclic.
- If the order of the group is prime, the group is cyclic.

References

[R65]

is_elementary(p)

Return True if the group is elementary abelian. An elementary abelian group is a finite abelian group, where every nontrivial element has order p, where p is a prime.

Examples

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

property is_nilpotent

Test if the group is nilpotent.

Explanation

A group G is nilpotent if it has a central series of finite length. Alternatively, G is nilpotent if its lower central series terminates with the trivial group. Every nilpotent group is also solvable ([1], p.29, [12]).

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup)
>>> C = CyclicGroup(6)
>>> C.is_nilpotent
True
>>> S = SymmetricGroup(5)
>>> S.is_nilpotent
False
```

See also:

lower_central_series (page 317), *is_solvable* (page 315)

is_normal(*gr*, *strict=True*)

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

Explanation

G is normal in gr if for each g2 in G, g1 in gr, `g = g1*g2*g1**-1` belongs to G It is sufficient to check this for each g1 in gr.generators and g2 in G.generators.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([1, 2, 0])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G1 = PermutationGroup([a, Permutation([2, 0, 1])])
>>> G1.is_normal(G)
True
```

property is_perfect

Return `True` if the group is perfect. A group is perfect if it equals to its derived subgroup.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation(1,2,3)(4,5)
>>> b = Permutation(1,2,3,4,5)
>>> G = PermutationGroup([a, b])
>>> G.is_perfect
False
```

property is_polycyclic

Return `True` if a group is polycyclic. A group is polycyclic if it has a subnormal series with cyclic factors. For finite groups, this is the same as if the group is solvable.

Examples

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

is_primitive(*randomized=True*)

Test if a group is primitive.

Explanation

A permutation group `G` acting on a set `S` is called primitive if `S` contains no nontrivial block under the action of `G` (a block is nontrivial if its cardinality is more than `1`).

Notes

The algorithm is described in [1], p.83, and uses the function minimal_block to search for blocks of the form $\{0, k\}$ for `k` ranging over representatives for the orbits of G_0, the stabilizer of `0`. This algorithm has complexity $O(n^2)$ where `n` is the degree of the group, and will perform badly if G_0 is small.

There are two implementations offered: one finds G_0 deterministically using the function `stabilizer`, and the other (default) produces random elements of G_0 using `random_stab`, hoping that they generate a subgroup of G_0 with not too many more orbits than G_0 (this is suggested in [1], p.83). Behavior is changed by the `randomized` flag.

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.is_primitive()
False
```

See also:

minimal_block (page 319), *random_stab* (page 324)

property is_solvable

Test if the group is solvable.

`G` is solvable if its derived series terminates with the trivial group ([1], p.29).

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(3)
>>> S.is_solvable
True
```

See also:

is_nilpotent (page 312), *derived_series* (page 306)

is_subgroup(*G*, *strict=True*)

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

If `strict` is `False` then if `self`'s degree is smaller than `G`'s, the elements will be resized to have the same degree.

Examples

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

Testing is strict by default: the degree of each group must be the same:

```
>>> p = Permutation(0, 1, 2, 3, 4, 5)
>>> G1 = PermutationGroup([Permutation(0, 1, 2), Permutation(0, 1)])
>>> G2 = PermutationGroup([Permutation(0, 2), Permutation(0, 1, 2)])
>>> G3 = PermutationGroup([p, p**2])
>>> assert G1.order() == G2.order() == G3.order() == 6
>>> G1.is_subgroup(G2)
True
>>> G1.is_subgroup(G3)
False
>>> G3.is_subgroup(PermutationGroup(G3[1]))
False
>>> G3.is_subgroup(PermutationGroup(G3[0]))
True
```

To ignore the size, set `strict` to `False`:

```
>>> S3 = SymmetricGroup(3)
>>> S5 = SymmetricGroup(5)
>>> S3.is_subgroup(S5, strict=False)
True
>>> C7 = CyclicGroup(7)
>>> G = S5*C7
>>> S5.is_subgroup(G, False)
True
>>> C7.is_subgroup(G, 0)
False
```

property is_symmetric

Return `True` if the group is symmetric.

Examples

```
>>> from sympy.combinatorics import SymmetricGroup
>>> g = SymmetricGroup(5)
>>> g.is_symmetric
True
```

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> g = PermutationGroup(
...     Permutation(0, 1, 2, 3, 4),
...     Permutation(2, 3))
>>> g.is_symmetric
True
```

Notes

This uses a naive test involving the computation of the full group order. If you need more quicker taxonomy for large groups, you can use *PermutationGroup.is_alt_sym()* (page 310). However, *PermutationGroup.is_alt_sym()* (page 310) may not be accurate and is not able to distinguish between an alternating group and a symmetric group.

See also:

is_alt_sym (page 310)

is_transitive(*strict=True*)

Test if the group is transitive.