

Posets

v1.0.2.25.9.25.7.40.24

William Gustafson

September 25, 2025

Contents

Introduction	5
Installation	5
Usage	5
Poset	8
Operations	9
adjoin_onehat	9
adjoin_zerohat	9
bddProduct	9
bddUnion	9
cartesianProduct	10
diamondProduct	10
dual	10
element_union	10
identify	10
prism	10
pyr	11
starProduct	11
union	11
Subposet Selection	11
complSubposet	11
filter	11
ideal	11
interval	12
max	12
min	12
properPart	12
rankSelection	12
subposet	12
Internal Computations	12
isAntichain	12
join	12
less	12
lesseq	13
mobius	13

rank	13
Queries	13
covers	13
isEulerian	13
isGorenstein	13
isLattice	14
isRanked	14
Invariants	14
abIndex	14
bettiNumbers	14
cdIndex	14
fVector	15
flagVectors	15
flagVectorsLatex	15
hVector	15
orderComplex	15
sparseKVector	15
Maps	16
buildIsomorphism	16
is_isomorphic	16
Miscellaneous	16
__contains__	16
__eq__	16
__getitem__	16
__hash__	16
__init__	16
__iter__	16
__len__	17
__repr__	17
__str__	17
chains	17
copy	17
fromSage	17
img	17
isoClass	18
latex	18
make_ranks	19
relabel	19
relations	19
reorder	19
show	19
shuffle	20
sort	20
toSage	20
transClose	20
zeta_from_relations	21
linearize	21
PosetIsoClass	21

Genlatt	21
Del	22
__init__	22
_minors	22
contract	22
delete	22
minor	22
minorPoset	22
minors	23
HasseDiagram	23
__init__	27
latex	27
line_options	27
loc_x	27
loc_y	28
nodeDraw	28
nodeLabel	28
nodeName	28
nodeTikz	28
node_options	28
tkinter	28
validate	28
SubposetsHasseDiagram	29
Q_nodeName	29
__init__	29
make_line_options	30
make_node_options	30
minNodeLabel	30
Built in posets	30
Antichain	30
Bnq	30
Boolean	31
Bruhat	31
Butterfly	32
Chain	32
Cube	33
DistributiveLattice	33
Empty	34
GluedCube	34
Grid	34
Intervals	35
KleinBottle	36
LatticeOfFlats	36
MinorPoset	37
NoncrossingPartitionLattice	39
PartitionLattice	40
Polygon	41

ProjectiveSpace	41
Root	42
Simplex	42
Torus	42
Uncrossing	43
UniformMatroid	44
Polynomial	45
__add__	46
__bool__	46
__eq__	46
__floordiv__	46
__ge__	46
__getitem__	46
__gt__	46
__init__	46
__iter__	47
__le__	47
__len__	47
__lt__	47
__mod__	47
__mul__	47
__neg__	47
__pow__	47
__add__	47
__repr__	48
__mul__	48
__setitem__	48
__str__	48
__sub__	48
__truediv__	48
_coeff_str	48
_monom_str	48
_poly_add_prepoly	48
_prepoly_mul_poly	49
abToCd	49
cdToAb	49
strip	49
sub	49
TriangularArray	49
__eq__	50
__getitem__	50
__init__	50
__repr__	50
__setitem__	50
__str__	50
col	50
inverse	50
revtranspose	50

row	51
subarray	51
ZetaHasseDiagram	51
__init__	51

Introduction

This module provides a class **Poset** that encodes a finite partially ordered set (poset). Most notably, this module can efficiently compute flag vectors, the **ab**-index and the **cd**-index. Quasigraded posets, in the sense of [4], can be encoded and the **ab**-index and **cd**-index of quasigraded posets can be computed. Latex code for Hasse diagrams can be produced with a very flexible interface. There are methods for common operations and constructions such as Cartesian products, disjoint unions, interval lattices, lattice of ideals, etc. Various examples of posets are provided such as Boolean algebras, the face lattice of the n -dimensional cube, (noncrossing) partition lattices, the type A_n Bruhat and weak orders, uncrossing orders etc. General subposets can be selected as well as particular ones of interest such as intervals and rank selections. Posets from this module can also be converted to and from posets from sagemath and Macaulay2.

Terminology and notation on posets generally follows [8] and [3].

The full documentation for the current version can be found [here](#).

Installation

Install with pip via `python -m pip install posets`. Alternatively, download the whl file [here](#) and install it with pip via `python -m pip posets-*-py3-none-any.whl`.

Usage

Here we give an introduction to using the posets module.

In the code snippets below we assume the module is imported via

```
from posets import *
```

Constructing a poset:

```
P = Poset(relations={'': ['a', 'b'], 'a': ['ab'], 'b': ['ab']})
Q = Poset(relations=[['', 'a', 'b'], ['a', 'ab'], ['b', 'ab']])
R = Poset(elements=['ab', 'a', 'b', ''], less=lambda x, y: return x in y)
S = Poset(zeta = [[0,1,1,1], [0,0,0,1], [0,0,0,1], [0,0,0,0]], elements=['', 'a', 'b', 'ab'])
```

Built in examples (see page 30):

```
Boolean(3) #Boolean algebra of rank 3
Cube(3) #Face lattice of the 3-dimensional cube
Bruhat(3) #Bruhat order on symmetric group of order 3!
Bnq(n=3,q=2) #Lattice of subspaces of  $F_2^3$ 
DistributiveLattice(P) #lattice of ideals of P
Intervals(P) #meet semilattice of intervals of P
```

These examples come with default drawing methods, for example, when making latex code by calling `DistributiveLattice(P).latex()` the resulting figure depicts elements of the lattice as Hasse diagrams of P with elements of the ideal highlighted (again, see page 30). Note, you will have to set the `height`, `width` and possibly `nodescale` parameters in order to get sensible output.

Two posets compare equal when they have the same set of elements and the same zeta values (i.e. the same order relation with the same weights):

```
P == Q and Q == R and R == S #True
P == Poset(relations={'': ['a', 'b']}) #False
P == Poset(relations={'': ['ab'], 'a': ['ab'], 'b': ['ab']}) #False
P == Poset(zeta=[[0,1,1,2], [0,0,0,3], [0,0,0,4], [0,0,0,0]],
           elements=['', 'a', 'b', 'ab']) #False
```

Use `is_isomorphic` or `PosetIsoClass` to check whether posets are isomorphic:

```
P.is_isomorphic(Boolean(2)) #True
P.isoClass()==Boolean(2).isoClass() #True
P.is_isomorphic(Poset(relations={'': ['a', 'b']})) #False
```

Viewing and creating Hasse diagrams:

```
P.show() #displays a Hasse diagram in a new window
P.latex() #returns latex code: \begin{tikzpicture}...
P.latex(standalone=True) #latex code for a
#standalone document: \documentclass{preview}...
display(P.img()) #Display a poset when in a Jupyter notebook
#this uses the output of latex()
```

Computing invariants:

```
Cube(2).fVector() #{(): 1, (1,): 4, (2,): 4, (1, 2): 8}
Cube(2).hVector() #{(): 1, (1,): 3, (2,): 3, (1, 2): 1}
Boolean(5).sparseKVector() #{(3,): 8, (2,): 8, (1, 3): 4, (1,): 3, (): 1}
Boolean(5).cdIndex() #Polynomial({'ccd': 3, 'cdc': 5, 'dd': 4, 'dcc': 3, 'cccc': 1})
print(Boolean(5).cdIndex()) #c4+3c2d+5cdc+3dc2+4d2
```

Polynomial operations:

```
#Create noncommutative polynomials from dictionaries,
#keys are monomials, values are coefficients
p=Polynomial({'ab':1})
q=Polynomial({'a':1, 'b':1})

#get and set coefficients like a dictionary
q['a'] #1
q['x'] #0
p['ba'] = 1

#print latex
str(p) #ab+ba
```

```

#basic arithmetic, polynomials form an algebra
p+q #ab+ba+a+b
p*q #aba+ab2+ba2+bab
q*p #a2b+aba+bab+b2a
2*p #2ab+2ba
p**2 #abab+ab2a+ba2b+baba
p**(-1) #raises TypeError
p**q #raises TypeError

#substitutions and conversions
p.sub(q,'a') #ab+ba+2b2 substitute q for a in p
p.abToCd() #d rewrite a's and b's
#in terms of c=a+b and d=ab+ba when possible
Polynomial({'c':1,'d':1}).cdToAb() #a+b+ab+ba rewrite c's and d's
#in terms of a's and b's

```

Converting posets to and from SageMath:

```

P.toSage() #Returns a SageMath class, must be run under sage
Poset.fromSage(Q) #Take a poset Q made with SageMath and return an instance of Poset

```

Converting to and from Macaulay2:

```

-- In M2
load "convertPosets.m2" --Also loads Python and Posets packages
import "posets" --This module must be installed to system version of python
P = posets@@Boolean(3) --Calling python functions
pythonPosetToMac(P) --Returns an instance of the M2 class Posets
macPosetToPython(Q) --Take a poset made with M2 and return an
--instance of the python class Poset

```

Quasigraded posets:

```

#Provide the zeta and rank functions explicitly
#To construct a 2-chain with top two elements rank 2 and 3
#and with zeta value -1 between minimum and the element covering it:
T = Poset([[1,-1,1],[1,1],[1]], ranks=[[0],[1],[2]])

```

The poset T above is from [4, Example 6.14] with M taken to be the 3-dimensional solid torus.

You can calculate the flag vectors and the **cd**-index just as you would for a classical poset, for example, `T.cdIndex()` returns the polynomial $c^2 - 2d$.

When plotting a quasigraded poset by default only the underlying poset is shown with element heights based on rank, the zeta values are not shown. If you wish to display the zeta values you can use the class `ZetaHasseDiagram` to draw a Hasse diagram of your poset with an element p depicted as the associated filter, namely the subposet $\{q : q \geq p\}$, and with elements of the filters labeled by the corresponding zeta value. To do so, either construct the poset with `hasse_class=ZetaHasseDiagram` such as in `Poset([[1,-1,1],[1,1],[1]], ranks=[[0],[1],[2]],hasse_class=ZetaHasseDiagram)` or set the Hasse diagram attribute on the poset as below:

```
T = Poset([[1,-1,1],[1,1],[1]], ranks=[[0],[1],[2]])
T.hasseDiagram = ZetaHasseDiagram(T)
```

You can also represent elements with ideals instead of filters by passing `filters=False`. See `ZetaHasseDiagram` and `SubposetsHasseDiagram` for a thorough explanation of the options.

Poset

class Poset

A class representing a finite partially ordered set (possibly quasigraded).

Posets are encoded by a list `elements`, a zeta function `zeta` describing the relations and a list `ranks` that specifies the length of each element. Instances of `Poset` also have an attribute `hasseDiagram` which is an instance of the `HasseDiagram` class used for plotting the poset.

To construct a poset you must pass one of the zeta matrix `zeta`, a function `less` or a list/dictionary `relations` to describe the relations. Additionally you may wish to specify the elements as a list called `elements` or by using the `relations` argument. The full list of constructor arguments are listed below.

Usually, the values of the zeta function are all 0 or 1, with 1 indicating a relation. The given zeta function may take other values, with 0 always indicating no relation and any other value indicating a relation with the specified weight. Additionally, if you provide a value for `ranks` you can specify any non-negative integer for the ranks of elements.

zeta – A triangular array indexed by i, j such that $i < j$ whose entries are 0 if i and j are incomparable and otherwise an arbitrary weight. This argument may be an instance of `TriangularArray`, a nested iterable or a flat iterable; in the latter case `flat_zeta` should be `True`. Elements are read row-wise when `zeta` is an iterable.

Note, unless `trans_close` is `False` it is only necessary to specify cover relations in the zeta matrix as the transitive closure will be computed.

elements – A list specifying the elements of the poset.

The default value is `[0, ..., len(zeta)-1]` Note, the list will be reordered into a linear extension as this is necessary to store the zeta function as a triangular array.

ranks – A list of lists. The i th list is a list of indices of element of length i . This argument is inessential, if not provided it will be computed by the constructor. If constructing a large poset with an easily computed rank function you may wish to compute and pass the rank function to the constructor. This option is also useful if you are constructing a quasigraded poset, the provided rank list does not necessarily have to match up with the usual length function.

relations – Either a list of pairs (x, y) such that $x < y$ or a dictionary whose values are lists of elements greater than the associated key. This is used to construct the zeta function if it is not provided. Note, it is only necessary to specify cover relations.

less – A function that given two elements p, q returns `True` when $p < q$. This is used to construct the zeta function if neither `zeta` nor `relations` are provided. It is only necessary to specify cover relations.

indices – A boolean indicating indices instead of elements are used in **relations** and **less**. The default is **False**.

name – An optional identifier. If not provided no name attribute is set.

hasse_class – An instance of **hasse_class** is constructed with arguments being this poset plus all keyword arguments passed to **Poset**, i.e.:

```

this.hasseDiagram = hasse_class(this, **kwargs)

```

If you subclass **HasseDiagram** to change default drawing behavior pass your subclass when constructing a poset.

The default value is **HasseDiagram**.

trans_close – If **True** the transitive closure of **zeta** is computed, this should be **False** only if the provided matrix satisfies $\text{zeta}[i,j]=0$ for i and j incomparable and $\text{zeta}[i,j] \neq 0$ otherwise. Similarly, if providing **relations** or **less** the argument **trans_close** should be **False** only if all relations were specified by the given data.

Any extra keyword arguments are passed to **HasseDiagram** (or **hasse_class** if specified).

Function calls to several of the more costly computations are cached. Generally, functions in this class do not change the poset but instead return a new poset. **Poset** objects may be considered immutable (this is not enforced in any way). If you alter a poset you should clear the cache via: **this.cache = {}**.

Operations

```

def adjoin_onehat(this, label=None)

```

Returns a new poset with a new maximum adjoined.

The label default is the same as **Poset.adjoin_zerohat**

```

def adjoin_zerohat(this, label=None)

```

Returns a new poset with a new minimum adjoined.

By default the label is the first non-negative integer that is not an element of the poset.

```

def bddProduct(this, that)

```

Computes the Cartesian product of two posets with maximum and minimum adjoined, that is, the poset

$$((P \setminus \{\max P, \min P\}) \times (Q \setminus \{\max Q, \min Q\})) \cup \{\hat{0}, \hat{1}\}.$$

```

def bddUnion(this, that)

```

Computes the disjoint union of two posets with maximum and minimums identified, that is, the poset

$$((P \setminus \{\max P, \min P\}) \sqcup (Q \setminus \{\max Q, \min Q\})) \cup \{\widehat{0}, \widehat{1}\}.$$

The labels in the returned poset are the same as in `element_union`.

```
def cartesianProduct(this, that)
```

Computes the Cartesian product.

```
def diamondProduct(this, that)
```

Computes the diamond product which is the Cartesian product of the two posets with their minimums removed and then adjoined with a new minimum.

```
def dual(this)
```

Returns the dual poset, this has the same elements and has relation $p \leq q$ when $q \leq p$ in the original poset.

```
def element_union(E, F)
```

Computes the disjoint union of lists E and F.

If E and F have empty intersection return value is $E+F$ otherwise return value is $(E \times \{0\}) \cup (F \times \{1\})$. This is used by operation methods such as `Poset.union` and `Poset.starProduct`.

```
def identify(this, X, indices=False)
```

Returns a new poset after making identifications indicated by the argument X.

The new relation is defined by $p \leq q$ when there exists any representatives p' and q' such that $p' \leq q'$ in the original poset.

The argument X should either be a dictionary where keys are the representatives and the value is a list of elements to identify with the key, or a list of lists where the first element of each list is the representative. Trivial equivalence classes need not be specified.

Raises `ValueError` if the identifications do not yield a poset (due to violation of the anti-symmetry axiom).

```
def prism(this)
```

Computes the prism of a poset, that is, the diamond product with `Cube(1)`.

```
def pyr(this)
```

Computes the pyramid of a poset, that is, the Cartesian product with a length 1 chain.

```
def starProduct(this, that)
```

Computes the star product of two posets.

This is the union of `this` with the maximum removed and `that` with the minimum removed and all relations $p < q$ for p in `this` and q in `that`.

```
def union(this, that)
```

Computes the disjoint union of two posets.

The labels in the returned poset are determined by `element_union`.

Subposet Selection

```
def complSubposet(this, S, indices=False)
```

Returns the subposet of elements not contained in `S`.

```
def filter(this, x, indices=False, strict=False)
```

Returns the subposet of elements greater than or equal to any element of `x`.

If `strict` is `True` then the element `x` is not included in the returned poset and if it is `False` the element `x` is included.

If `indices` is `True` then the element `x` is interpreted as indices into the poset, either way the return value is a subposet of the original poset.

Note, `x` should not be an element of the poset but an iterable of elements; to construct a principal filter for an element `p` of a poset `P` use `P.filter((p,))`.

```
def ideal(this, x, indices=False, strict=False)
```

Returns the subposet of elements less than or equal to any element of `x`.

See `Poset.filter` for an explanation of the arguments.

```
def interval(this, i, j, indices=False)
```

Returns the closed interval $[i, j]$.

```
def max(this, indices=False)
```

Returns a list of the maximal elements of the poset.

```
def min(this, indices=False)
```

Returns a list of the minimal elements of the poset.

```
def properPart(this)
```

Returns the subposet of all elements that are neither maximal nor minimal.

```
def rankSelection(this, S)
```

Returns the subposet of elements whose rank is contained in S .

Note, this does not automatically include the minimum or maximum.

```
def subposet(this, S, indices=False, keep_hasseDiagram=True)
```

Returns the subposet of elements in S .

Internal Computations

```
def isAntichain(this, A, indices=False)
```

Returns whether the given set is an antichain ($i \not\leq j$ for all i and j).

```
def join(this, i, j, indices=False)
```

Computes the join of i and j , if it does not exist returns `None`.

```
def less(this, i, j, indices=False)
```

Returns whether i is strictly less than j .

```
def lesseq(this, i, j, indices=False)
```

Returns whether i is less than or equal to j .

```
def mobius(this, i=None, j=None, indices=False)
```

Computes the value of the Möbius function from i to j .

If neither i nor j are provided returns all values of the Möbius function as an instance of `TriangularArray`.

If i (respectively j) is not provided then it is assumed to be the minimum (maximum) and raises an exception of type `ValueError` if the poset does not have a unique minimum (maximum).

```
def rank(this, i, indices=False)
```

Returns the length of i (the length of the longest chain ending at i).

Returns `None` if i is not an element (or i is not a valid index if `indices` is `True`).

Queries

```
def covers(this, indices=False)
```

Returns the list of covers of the poset.

An element q covers p when $p < q$ and $p < r < q$ implies $r = p$ or $r = q$.

```
def isEulerian(this)
```

Checks whether the given poset is Eulerian (every interval with at least 2 elements has an equal number of odd and even rank elements).

```
def isGorenstein(this)
```

Checks if the poset is Gorenstein*.

A poset is Gorenstein* if the proper part of all intervals with more than two elements have sphere homology. In other words, this function checks that

```
    this.interval(p,q).properPart().bettiNumbers()
```

is either $[2]$ or $[1,0,\dots,0,1]$ for all $p \leq q$ such that $|\text{rk}(q) - \text{rk}(p)| \geq 2$.

```
def isLattice(this)
```

Checks if the poset is a lattice.

Note, returns `True` if `this` is an empty poset.

```
def isRanked(this)
```

Checks whether the poset is ranked.

Invariants

```
def abIndex(this)
```

Returns the ab-index of the poset as an instance of `Polynomial`.

If the poset has a unique minimum and maximum but isn't ranked this computes the ab-index considering the poset to be quasigraded (in the sense of [4]) with $\bar{\zeta} = \zeta$ and ρ the rank function $x \mapsto \mathbf{P}.\text{rank}(x)$.

For more information on the ab-index see [1]

```
def bettiNumbers(this)
```

Computes the Betti numbers of the poset, that is, the ranks of the homology of the order complex (the simplicial complex of all chains).

```
def cdIndex(this)
```

Returns the `cd`-index as an instance of `Polynomial`.

If the given poset does not have a `cd`-index then a `cd`-polynomial is still returned, but this is not meaningful. If you wish to check whether a poset has a `cd`-index then check the Boolean below:

```
    this.abIndex().cdToAb() != this.abIndex()
```

If the given poset is semi-Eulerian then the **cd**-index as defined in [6] is computed.

This function correctly computes the **cd**-index of a quasigraded Eulerian poset (see [4]).

For computation we use the sparse k -vector formula see [2, Proposition 7.1]. For more info on the **cd**-index see [1].

```
def fVector(this)
```

Returns flag \bar{f} -vector as a dictionary with keys $S \subseteq [n]$.

This method is intended for use with a poset, possibly quasi-graded, that has a unique minimum and maximum. On a classical poset this counts chains that contain the first minimal element, `this[this.ranks[0][0]]` and ignores the final rank.

Chains are weighted by the product of the zeta values along the chain.

```
def flagVectors(this)
```

Returns the table of flag \bar{f} - and \bar{h} -vectors as a dictionary with keys $S \subseteq [n]$ encoded as tuples and with elements `(f_S, h_S)`.

```
def flagVectorsLatex(this,standalone=False)
```

Returns a string of latex code representing the table of flag vectors of the poset.

Note, the package longtable is required to compile the output.

```
def hVector(this)
```

Returns the flag \bar{h} -vector of the poset.

```
def orderComplex(this, indices=False)
```

Returns the poset of all chains ordered by inclusion.

```
def sparseKVector(this)
```

Returns the sparse k -vector $k_S = \sum_{T \subseteq S} (-1)^{|S \setminus T|} \bar{h}_T$.

The sparse k -vector only has entries k_S for sparse sets S , that is, sets $S \subseteq [\text{rk}(P) - 1]$ such that if $i \in S$ then $i + 1 \notin S$. The sparse k -vector is returned as a dictionary whose keys are tuples.

Maps

```
def buildIsomorphism(this, that, indices=False)
```

Returns an isomorphism from `this` to `that` as a dictionary or `None` if the posets are not isomorphic.
If `indices` is `True` then the dictionary keys and values are indices, otherwise they are elements.

```
def is_isomorphic(this, that)
```

Returns `True` if the posets are isomorphic and `False` otherwise.

Miscellaneous

```
def __contains__(this, p)
```

Wrapper for `this.elements.__contains__`.

```
def __eq__(this, that)
```

Returns `True` when `that` is a `Poset` representing the same poset as `this` and `False` otherwise.

```
def __getitem__(this, i)
```

Wrapper for `this.elements.__getitem__`.

```
def __hash__(this)
```

Hashes the poset, dependent on `this.elements` and relations between ranks.

```
def __init__(this, zeta=None, elements=None, ranks=None, less=None, name='',  
hasse_class=None, trans_close=True, relations=None, indices=False, flat_zeta=False,  
that=None, **kwargs)
```

See `Poset`.

```
def __iter__(this)
```


Wrapper for `this.elements__iter__`.

```
def __len__(this)
```

Wrapper for `this.elements.__len__`.

```
def __repr__(this)
```

Gives a string that can be evaluated to recreate the poset.

To use the returned string with `eval` the class `Poset` must be in the namespace and `repr(this.elements)` must return a suitable string for evaluation.

```
def __str__(this)
```

Returns a nicely formatted string listing the zeta matrix, the ranks list and the elements of the poset.

```
def chains(this, indices=False)
```

Returns a list of all nonempty chains of the poset (subsets $p_1 < \dots < p_r$).

```
def copy(this)
```

Returns a shallow copy of the poset.

Making a shallow copy via the copy module i.e. `Q = copy.copy(P)` doesn't update the self reference in `Q.hasseDiagram` (in this example `Q.hasseDiagram.P` is `P`). This doesn't matter if you treat posets as immutable, but otherwise could cause issues when displaying or generating hasse diagrams. The returned poset has the self reference updated.

```
def fromSage(P)
```

Convert an instance of `sage.combinat.posets.poset.FinitePoset` to an instance of `Poset`.

```
def img(this, tmpfile='a.tex', tmpdir=None, **kwargs)
```

Produces latex code (via calling `latex`) compiles it with `pdflatex` and returns a `wand.image.Image` object constructed from the pdf.

In a Jupyter notebook calling `display` on the return value will show the Hasse diagram in the output cell. By default `tmpdir` is `tempfile.gettempdir()`.

This function converts the compiled pdf to an image using `imagemagick`, this may fail due to `imagemagick`'s default security policies. For more info and how to fix the issue see <https://askubuntu.com/questions/1127260/imagemagick-convert-not-allowed>

Keyword arguments are passed to `latex()` but `standalone` is always set to `True` (otherwise the pdf would not compile). Note this function may hang if `pdflatex` fails to compile.

```
def isoClass(this)
```

Returns an instance of `PosetIsoClass` representing the isomorphism class of the poset.

```
def latex(this, **kwargs)
```

Returns a string of tikz code to draw the Hasse diagram of the poset for use in a \LaTeX document. This is a wrapper for `HasseDiagram.latex`.

For a full list of keyword arguments see `HasseDiagram`. The most common arguments are:

height – The height in tikz units of the diagram.

The default value is 10.

width – The width in tikz units of the diagram.

The default value is 8.

labels – If `False` elements are represented by filled circles. If `True` by default elements are labeled by the result of casting the poset element to a string.

The default value is `True`.

ptsize – When `labels` is `False` this is the size of the circles used to represent elements. This has no effect if `labels` is `True`.

The default value is `'2pt'`.

nodescale – Each node is wrapped in `'\scalebox{'+nodescale+'}'`.

The default value is `'1'`.

standalone – When `True` a preamble is added to the beginning and `'\end{document}'` is added to the end so that the returned string is a full \LaTeX document that can be compiled. Compiling requires the \LaTeX packages `tikz` (`pgf`) and `preview`. The resulting figure can be incorporated into another \LaTeX document with `\includegraphics`.

When `False` only the code for the figure is returned; the return value begins with `\begin{tikzpicture}` and ends with `\end{tikzpicture}`.

The default is `False`.

nodeLabel – A function that takes the `HasseDiagram` object and an index and returns the label for the indicated element as a string. For example, the default implementation `HasseDiagram.nodeLabel` returns the element cast to a string and is defined as below:

```
def nodeLabel(H, i):
    return str(H.P[i])
```

Note `H.P` is `this`.

```
def make_ranks(zeta)
```

Used by the constructor to compute the ranks list for a poset when it isn't provided.

```
def relabel(this, elements=None)
```

Returns a new `Poset` object with the `elements` attribute as given.

If `elements` is `None` then the returned poset has `elements` attribute set to `list(range(len(this)))`.

```
def relations(this, indices=False)
```

Returns a list of all pairs (e, f) where $e \leq f$.

```
def reorder(this, perm, indices=False)
```

Returns a new `Poset` object (representing the same poset) with the elements reordered.

`perm` should be a list of elements if `indices` is `False` or a list of indices if `True`. The returned poset has elements in order of a linear extension created from the given permutation. If the list `perm` defines a linear extension then `perm[i]` is the i th element.

If `perm` is not a linear extension it is coerced into a linear extension by repeatedly removing the first element whose ideal has already been removed.

```
def show(this, **kwargs)
```

Opens a window displaying the Hasse diagram of the poset.

This is a wrapper for `HasseDiagram.tkinter`.

For a full list of keyword arguments see `HasseDiagram`. The most common arguments are:

height – The height of the diagram.

The default value is 10.

width – The width of the diagram.

The default width is 8.

labels – If **False** elements are represented as filled circles. If **True** by default elements are labeled by the result of casting the poset element to a string.

The default value is **True**.

ptsize – When **labels** is **False** controls the size of the circles representing elements. This can be an integer or a string, if the value is a string the last two characters are ignored.

The default value is **'2pt'**.

scale – Scale of the diagram.

The default value is 1.

padding – A border of this width is added around all sides of the diagram.

The default value is 1.

nodeLabel – A function that takes the **HasseDiagram** object and an index and returns the label for the indicated element as a string. For example, the default implementation **HasseDiagram.nodeLabel** returns the element cast to a string and is defined as below:

```
def nodeLabel(H, i):  
    return str(H.P[i])
```

Note **H.P** is **this**.

```
def shuffle(this)
```

Returns a new **Poset** object (representing the same poset) with the elements in a random order.

```
def sort(this, key = None, indices=False)
```

Returns a new **Poset** object (representing the same poset) with the elements sorted.

If the given key does not define a linear extension then the ordering is coerced to one in the same way as **reorder**.

```
def toSage(this)
```

Converts the poset to an instance of **sage.combinat.posets.posets.FinitePoset**.

```
def transClose(T)
```

Given an instance of **TriangularArray** encoding a (possibly weighted) relation, via $x \sim y$ when the x, y entry is nonzero computes the transitive closure.

Note, an induced relation $i < j$ is weighted by $T[i,k]*T[k,j]$ where j and k are the minimal indices greater than i satisfying the conditions $T[i,k] \neq 0$ and $T[k,j] \neq 0$.

```
def zeta_from_relations(relations,elements)
```

Given a dictionary of relations and a list of elements returns the zeta matrix and the elements reordered in a linear extension.

The dictionary `relations` should have keys that are indices into the list `elements` and values that are lists of indices into `elements`. If an index `i` is contained in the list `relations[j]` then the j th elements is less than the i th element in the poset.

```
def linearize(this, X, indices=False)
```

Given a list `X` of elements, returns a new list of the elements in a linear extension of the subposet.

PosetIsoClass

```
class PosetIsoClass(Poset)
```

This class encodes the isomorphism type of a poset.

Internally, this class inherits from `Poset` and thus instances of `PosetIsoClass` are also instances of `Poset`. The major differences between this class and `Poset` are that `PosetIsoClass.__eq__` returns `True` when the two posets are isomorphic and all methods in `Poset` that return a `Poset` object in `PosetIsoClass` instead return an instance of `PosetIsoClass`.

Construct an instance of `PosetIsoClass` the same way as you would an instance of `Poset` or given a poset `P` use `P.isoClass()`.

Genlatt

```
class Genlatt(Poset)
```

A class to encode a “generator-enriched lattice” which is a lattice L along with a set $G \subseteq L \setminus \{\hat{0}\}$ that generates L under the join operation.

This class is mainly provided for the `minorPoset` method.

Constructor arguments are the same as for `Poset` except that this constructor accepts two additional keyword only arguments:

- `G` - An iterable specifying the generating set, can either contain elements of the lattice or indices into the lattice. The join irreducibles are automatically added and may be omitted. If `G` is not provided the generating set will consist of the join irreducibles.
- `G_indices` - If `True` then the provided argument `G` should consist of indices otherwise `G` should consist of elements.

Note, a lattice L enriched with a generating set G is denoted as the pair (L, G) . See [5] for more on generator-enriched lattices¹.

```
def Del(this, K)
```

Return the deletion set of the minor K .

The argument K should be an instance of `Genlatt`.

The deletion set of a minor (K, H) of (L, G) is the set

$$\text{Del}(K, H) = \{g \in G : g \vee \widehat{0}_K \notin H \cup \{\widehat{0}_K\}\}$$

This is the minimal set of generators that must be deleted to form (K, H) from (L, G) .

```
def __init__(this, *args, G=None, G_indices=False, **kwargs)
```

See `Genlatt`.

```
def _minors(this, minors, rels, i, weak, L)
```

Recursion backend to `minors`.

```
def contract(this, g, weak=False, L=None)
```

Return the `Genlatt` obtained by contracting the generator g , if `weak` is `True` performs the weak contraction with respect to L (default value for L is `this`).

```
def delete(this, g)
```

Return the `Genlatt` obtained by deleting the generator g .

```
def minor(this, H, z)
```

Given an iterable H of generators and an element z returns the `Genlatt` with minimum z and generating set H and with the same order as `this`.

```
def minorPoset(this, weak=False, **kwargs)
```

¹perhaps too much more

Returns the minor poset of the given `Genlatt` instance.

When generating a Hasse diagram with `latex()` use the prefix `L_` to control options for the node diagrams.

```
def minors(this, weak=False)
```

Returns a list of minors of the given `Genlatt` instance and an incomplete dictionary of relations.

The relations when transitively closed yield the relations for the minor poset.

HasseDiagram

class HasseDiagram

A class that can produce latex/tikz code for the Hasse diagram of a poset or display the diagram in a window using tkinter.

Overview

An instance of this class is attached to each instance of `Poset`. This class is used to produce latex code for a poset when `Poset.latex()` is called or to display a poset in a new window when `Poset.show()` is called. These functions are wrappers for `HasseDiagram.latex()` and `HasseDiagram.tkinter()`.

The constructor for this class takes keyword arguments that control how the Hasse diagram is drawn. These keyword arguments set the default options for that given instance of `HasseDiagram`. When calling `latex()` to produce latex code or `tkinter()` to draw the diagram in a tkinter window the same keyword arguments can be passed to control how the diagram is drawn during that particular operation.

There are two types of options: constant values such as `height`, `width` or `scale` and function values such as `loc_x`, `loc_y` or `nodeLabel`.

Keyword arguments

Options that affect both `latex()` and `tkinter()`:

width – Width of the diagram. When calling `latex()` this is the width in tikz units (by default centimeters), for `tkinter()` the units are $\frac{1}{30}$ th of tkinter's units.

The default value is 8.

height – Height of the diagram, uses the same units as width.

The default value is 10.

labels – If this is `True` display labels, obtained from `nodeLabels`, for elements; if this is `False` display filled circles for elements. The default value is `True`.

ptsize – No effect when `labels` is `True`, when `labels` is `False` this is the size of the circles shown for elements. When calling `tkinter()` this can be either a number or a string. For compatibility if `ptsize` is a string the last two characters are ignored. When calling `latex()` this should be a string and include units.

The default value is '2pt'.

`indices_for_nodes` – If `True` then `this.nodeLabel` is not called and the node text is the index of the element in the poset. If `labels` is `False` this argument has no effect.

The default value is `False`.

`nodeLabel` – A function that given `this` and an index returns a string to label the corresponding element by.

The default value is `HasseDiagram.nodeLabel`.

`loc_x` – A function that given `this` and an index returns the x -coordinate of the element in the diagram as a string. Positive values extend rightward and negative leftward.

The default value is `HasseDiagram.loc_x`.

`loc_y` – A function that given `this` and an index returns the y -coordinate of the element in the diagram as a string. Positive values extend upward and negative values downward.

The default value is `HasseDiagram.loc_y`.

`jiggle` `jiggle_x` `jiggle_y` – Coordinates of all elements are perturbed by a random vector in the rectangle

$$\begin{aligned} -\text{jiggle} - \text{jiggle_x} &\leq x \leq \text{jiggle} + \text{jiggle_x} \\ -\text{jiggle} - \text{jiggle_y} &\leq y \leq \text{jiggle} + \text{jiggle_y} \end{aligned}$$

This can be useful if you want to prevent cover lines from successive ranks aligning to form the illusion of a line crossing between two ranks; or when drawing unranked posets if a line happens to cross over an element. The perturbation occurs in `loc_x` and `loc_y` so if these are overwritten and you want to preserve this behaviour add a line to the end of your implementation of `loc_x` such as

```
x = x+random.uniform(-this.jiggle-this.jiggle_x,this.jiggle+this.jiggle_x)
```

The default values are 0.

`scale` – In `latex()` this is the scale parameter for the `tikz` environment, i.e. the `tikz` environment containing the figure begins

```
'\\begin{tikzpicture}[scale='+tikzscale+']'
```

In `tkinter()` all coordinates are scaled by `scale`.

The default value is `'1'`, this parameter may be a string or a numeric type.

Options that affect only `latex()`:

`preamble` – A string that when calling `latex()` is placed in the preamble. It should be used to include any extra packages or define commands needed to produce node labels. This has no effect when standalone is `False`.

The default value is `''`.

`nodescale` – Each node is wrapped in `'\\scalebox{'+nodescale+'}'`.

The default value is `'1'`.

`line_options` – Tikz options to be included on lines drawn, i.e. lines will be written as

```
'\\draw['+line_options+'](...'
```

The value for `line_options` can be either a string or a function; when it is a string the same options are placed on every line and when the value is a function it is passed `this`, the `HasseDiagram` object, `i`, the index to the element at the bottom of the cover and `j`, the index to the element at the top of the cover.

The default value is `''`.

node_options – Tikz options to be included on nodes drawn, i.e. nodes will be written as

```
'\\node['+node_options_'](...'
```

Just as with **line_options** the value for **node_options** can be either a string or a function; if it is a function it is passed **this**, the **HasseDiagram** object, and **i**, the index to the element being drawn.

northsouth – If **True** lines are not drawn between nodes directly but from **node.north** to **node.south** which makes lines come together just beneath and above nodes. When **False** lines are drawn directly to nodes which makes lines directed towards the center of nodes.

The default is **True**.

lowsuffix – When this is nonempty lines will be drawn to **node.lowsuffix** instead of directly to nodes for the higher node in each cover. If **northsouth** is **True** this has no effect and **'.south'** is used for the low suffix.

The default is **''**.

highsuffix – This is the suffix for the bottom node in each cover. If **northsouth** is **True** this has no effect and **'.north'** is used for the high suffix.

The default is **''**.

nodeName – A function that takes **this** and an index **i** representing an element whose node is to be drawn and returns the name of the node in tikz. This does not affect the image but is useful if you intend to edit the latex code and want the node names to be human readable.

The default value **HasseDiagram.nodeName** returns **str(i)**.

standalone – When **True** a preamble is added to the beginning and **'\\end{document}'** is added to the end so that the returned string is a full latex document that can be compiled. Compiling requires the latex packages tikz (pgf) and preview. The resulting figure can be incorporated into another latex document with **\includegraphics**.

When **False** only the code for the figure is returned, which case the return value begins with **\begin{tikzpicture}** and ends with **\end{tikzpicture}**.

The default is **False**.

Options that affect only **tkinter()**:

padding – A border of this width is added around all sides of the diagram. This is affected by **scale**.

The default value is 3.

offset – Cover lines start above the bottom element and end below the top element, this controls the separation.

The default value is 0.5.

nodeDraw – When **labels** is **False** this function is called instead of placing anything for the node. The function is passed **this** and an index to the element to be drawn. **nodeDraw** should use the **tkinter.Canvas** object **this.canvas** to draw. The center of your diagram should be at the point with coordinates given below.

```
x = float(this.loc_x(this,i)) * float(this.scale) + float(this.scale) * \
float(this.width)/2 + float(this.padding)
```

```
y = 2 * float(this.padding) + float(this.height) * \
float(this.scale) - (float(this.loc_y(this,i)) * float(this.scale) + \
float(this.padding))
```

For larger diagrams make sure to increase `height` and `width` as well as `offset`.

The default value is `HasseDiagram.nodeDraw`.

Overriding function parameters

Function parameters can be overridden in two ways. The first option is to make a function with the same signature as the default function and to pass that function as a keyword argument to the constructor or `latex()`/`tkinter()` when called.

For example:

```
def nodeLabel(this, i):
    return str(this.P.mobius(0, this.P[i]))

#P is a Poset already constructed that has a minimum 0
P.hasseDiagram.tkinter(nodeLabel = nodeLabel)
```

The code above will show a Hasse Diagram of `P` with the elements labeled by the Möbius values $\mu(0, p)$. When overriding function parameters the first argument is always the `HasseDiagram` instance. The class `HasseDiagram` has an attribute for each of the options described above as well as the following attributes:

- `P` – The poset to be drawn.
- `in_tkinter` – Boolean indicating whether `tkinter()` is being executed.
- `in_latex` – Boolean indicating whether `latex()` is being executed.
- `canvas` – While `tkinter()` is being executed this is the `tkinter.Canvas` object being drawn to.

Note that any function parameters, such as `nodeLabel`, are set via

```
this.nodeLabel = #provided function
```

so if you intend to call these functions you must pass `this` as an argument via

```
this.nodeLabel(this, i)
```

The class methods remain unchanged of course, for example `HasseDiagram.nodeLabel` always refers to the default implementation.

Subclassing

The second way to override a function parameter is via subclassing. This is more convenient if overriding several function parameters at once or if the computations are more involved. It is also useful for adding extra parameters. Any variables initialized in the constructor are saved at the beginning of `latex()` or `tkinter()`, overridden during execution of the function by any provided keyword arguments, and restored at the end of execution. The Möbius example above can be accomplished by subclassing as follows:

```
class MobiusHasseDiagram(HasseDiagram):

    def nodeLabel(this, i):
        zerohat = this.P.min()[0]
        return str(this.P.mobius(zerohat, this.P[i]))

P.hasseDiagram = MobiusHasseDiagram(P)
P.hasseDiagram.tkinter()
```

To provide an option that changes what element the Möbius value is computed from just set the value in the constructor.

```
class MobiusHasseDiagram(HasseDiagram):

    def __init__(this, P, z = None, **kwargs):
        super().__init__(P, **kwargs)

        if z == None:
            this.z = this.P.min()[0] #z defaults to first minimal element
        else:
            this.z = z

    def nodeLabel(this, i):
        return str(this.P.mobius(this.z, this.P[i]))

    #P is a Poset with minimum 0
    P.hasseDiagram = MobiusHasseDiagram(P)
    P.hasseDiagram.tkinter() #labels are  $\mu(0, x)$ 
    P.hasseDiagram.tkinter(z = P[0]) #labels are  $\mu(P_0, x)$ 
    P.hasseDiagram.tkinter() #labels are  $\mu(0, x)$ 
```

Note you can pass a class to the Poset constructor to construct a poset with a `hasseDiagram` of that class.

```
def __init__(this, P=None, that=None,**kwargs)
```

See HasseDiagram.

```
def latex(this, **kwargs)
```

Returns a string to depict the Hasse diagram in \LaTeX .

The keyword arguments are described in HasseDiagram.

```
def line_options(this,i,j)
```

This is the default implementation of `line_options`, it returns an empty string.

```
def loc_x(this, i)
```

This is the default implementation of `loc_x`.

This spaces elements along each rank evenly. The length of a rank is the ratio of the logarithms of the size of the rank to the size of the largest rank.

The return value is a string.

```
def loc_y(this,i)
```

This is the default value of `loc_y`.

This evenly spaces ranks. The return value is a string.

```
def nodeDraw(this, i)
```

This is the default implementation of `nodeDraw`.

This draws a filled black circle of radius `ptsize/2`.

```
def nodeLabel(this,i)
```

This is the default implementation of `nodeLabel`.

The i th element is returned cast to a string.

```
def nodeName(this,i)
```

This is the default implementation of `nodeName`.

i is returned cast to a string.

```
def nodeTikz(this,i)
```

This is the default implementation of `nodeTikz` used to draw nodes when `labels` is `False`.

```
def node_options(this,i)
```

This is the default implementation of `node_options`, it returns an empty string.

```
def tkinter(this, **kwargs)
```

Opens a window using `tkinter` and draws the Hasse diagram.

The keyword arguments are described in `HasseDiagram`.

```
def validate(this)
```

Validates and corrects any variables on `this` that may need preprocessing before drawing.

SubposetsHasseDiagram

```
class SubposetsHasseDiagram(HasseDiagram)
```

This is a class used to draw posets whose elements are themselves subposets of some global poset, such as interval posets or lattices of ideals.

The elements of the poset P to be drawn are subposets of a poset Q . The nodes in the Hasse diagram of P are represented as posets. The entire poset Q is drawn for each element of P , the elements and edges contained in the given subposet are drawn in black and elements and edges not contained in the subposet are drawn in gray.

Options can be passed to this class in order to control the drawing of the diagram in the same way as for the class `HasseDiagram`. For example, calling `latex(width=5)` on an instance of `SubposetsHasseDiagram` sets the width of the entire diagram (that of P) to 5. To control options for the subposets a prefix, by default 'Q', is used. For example, `latex(Q_width=5,width=40)` would set the width of each subposet to 5 and the width of the entire diagram to 40.

```
def Q_nodeName(this, i)
```

Returns a node name for an element of P .

To ensure the node names of the larger figure and of the subdiagrams do not clash all node names are prefixed with `this.prefix`.

```
def __init__(this, P, Q, is_in=lambda x,X:x in X, prefix='Q', draw_min=True, func_args=None,
**kwargs)
```

Constructor arguments:

prefix – String to prefix options to be passed to the instances of `HasseDiagram` that draw the subdiagrams.

The argument **prefix** should be a valid tikz node name. It is recommended that **prefix** is also a valid python variable name.

is_in – A function used by the constructor to test whether elements of Q are elements of a subposet. The function **is_in** takes two arguments, an element x of the poset Q and the subposet object X to test containment with. The default value returns `x in X`.

draw_min – If `True` all elements of P are represented by a Hasse diagram. If `False` minimal elements are not drawn but instead labeled by the return value of `this.minNodeLabel`.

func_args – A dictionary whose keys are names of keyword arguments to `HasseDiagram.latex` and whose values are functions that take this instance of `SubposetsHasseDiagram` and an index into the poset P . When the subposet for an element p at index i in P is drawn both **this** and **i** are passed to each function and the corresponding option is set to the return value when drawing the subposet.

All keyword arguments not beginning with the string `this.prefix+'_'` are handled the same as in the class `HasseDiagram`. Keyword arguments that begin with the string `this.prefix+'_'`

are saved as attributes and passed to the instances of `HasseDiagram` drawing the subposets when `latex()` is called.

```
def make_line_options(q)
```

Returns a function to be supplied as `line_options` in the `latex()` call to draw a diagram for `q`.

```
def make_node_options(q)
```

Returns a function to be supplied as `node_options` in the `latex()` call to draw a diagram for `q`.

```
def minNodeLabel(this)
```

Returns `r'\emptyset'`.

This function is called by `nodeLabel` to get a node label for minimal elements if `draw_min` is `False`. To change the label for minimal elements provide your own version of `minNodeLabel`.

Built in posets

```
def Antichain(n)
```

Returns the poset on $1, \dots, n$ with no relations.

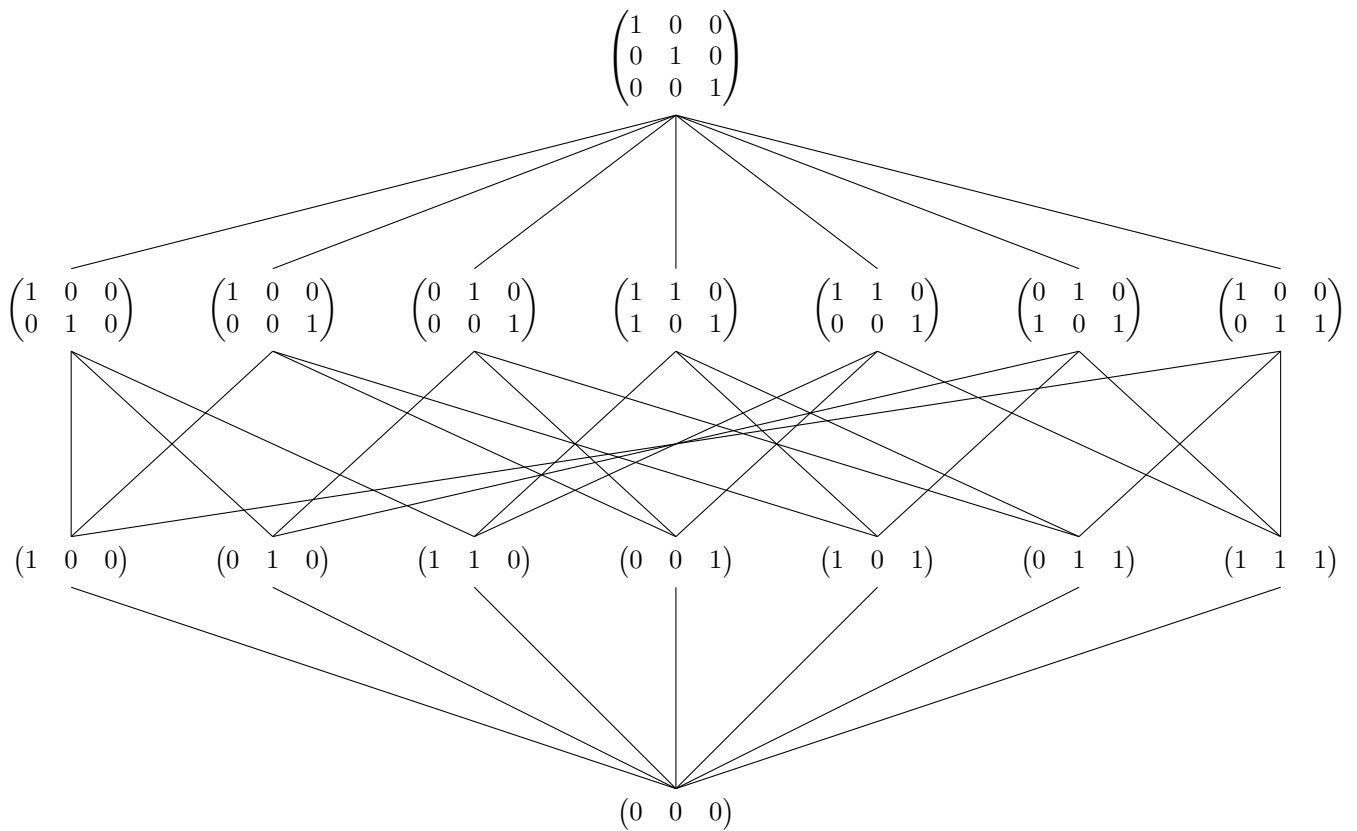
1 2 3

The poset `Antichain(3)`.

```
def Bnq(n=2, q=2)
```

Returns the poset of subspaces of the vector space \mathbb{F}_q^n where \mathbb{F}_q is the field with `q` elements.

Currently only implemented for `q` a prime. Raises an instance of `NotImplementedError` if `q` is not prime.

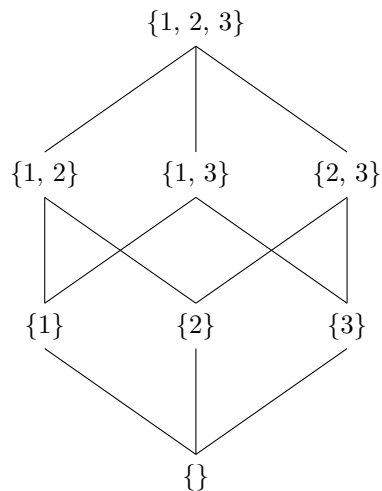


The poset $\text{Bnq}(3,2)$.

```
def Boolean(n)
```

Returns the poset of subsets of a set, ordered by inclusion.

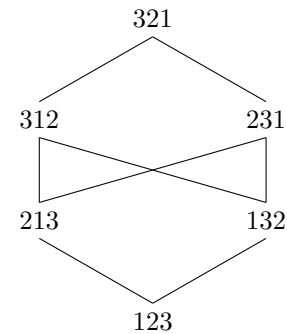
The parameter n may be an integer, in which case the poset of subsets of $\{1, \dots, n\}$ is returned, or an iterable in which case the poset of subsets of n is returned.



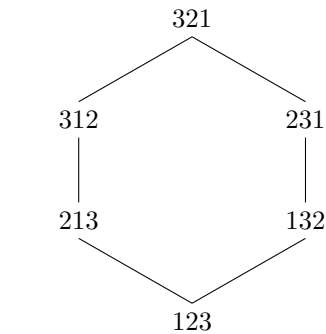
The poset $\text{Boolean}(3)$.

```
def Bruhat(n, weak=False)
```

Returns the type A_{n-1} Bruhat order (the symmetric group S_n) or the type A_{n-1} left weak order.



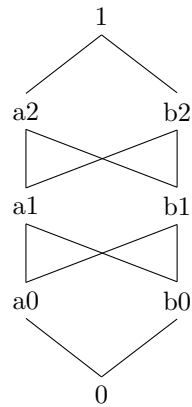
The poset `Bruhat(3)`



The poset `Bruhat(3, True)`

```
def Butterfly(n)
```

Returns the rank $n + 1$ bounded poset where ranks $1, \dots, n$ have two elements and all comparisons between ranks.



The poset `Butterfly(3)`.

```
def Chain(n)
```

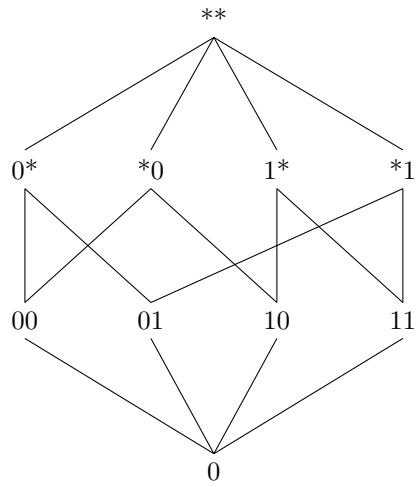
Returns the poset on $0, \dots, n$ ordered linearly (i.e. by usual ordering of integers).



The poset `Chain(3)`.


```
def Cube(n)
```

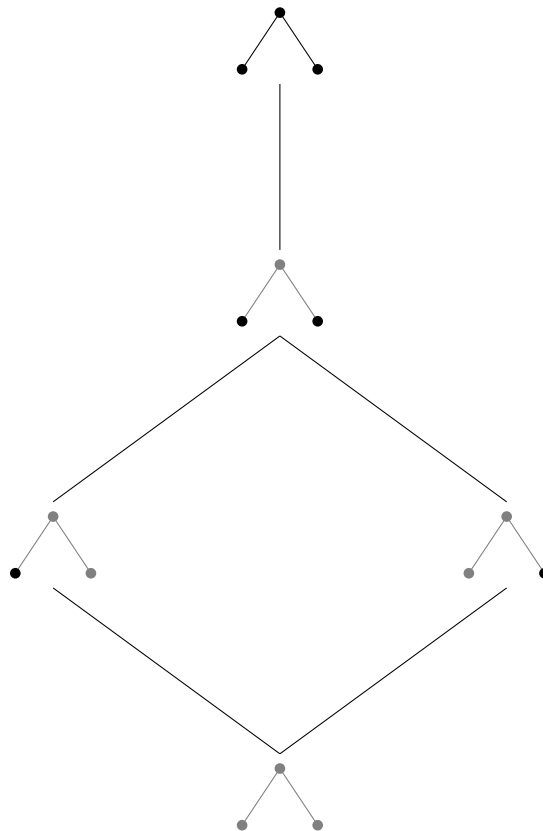
Returns the face lattice of the n -dimensional cube.



The poset `Cube(2)`.

```
def DistributiveLattice(P, indices=False)
```

Returns the lattice of ideals of a given poset.



The poset `DistributiveLattice(Root(3))`.

When generating a Hasse diagram with `latex()` use the prefix `irr_` to control options for the node diagrams.

```
def Empty()
```

Returns an empty poset.

```
def GluedCube(orientations = None)
```

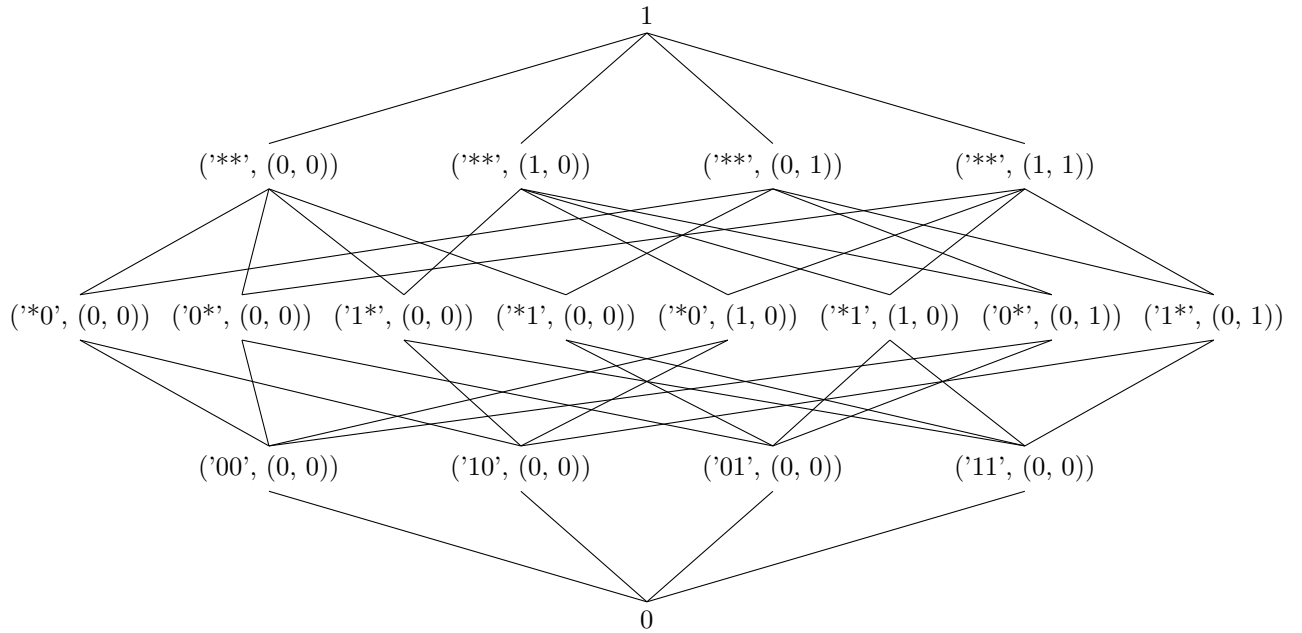
Returns the face poset of the cubical complex obtained from a $2 \times \cdots \times 2$ grid of cubes of dimension `len(orientations)` via a series of gluings as indicated by the parameter `orientations`.

If `orientations` is $[1, \dots, 1]$ a torus is constructed and if `orientations` is $[-1, \dots, -1]$ the projective space of dimension n is constructed.

If `orientations[i] == 1` the two ends of the large cube are glued so that points with the same image under projecting out the i th coordinate are identified.

If `orientations[i] == -1` points on the two ends of the large cube are identified with their antipodes.

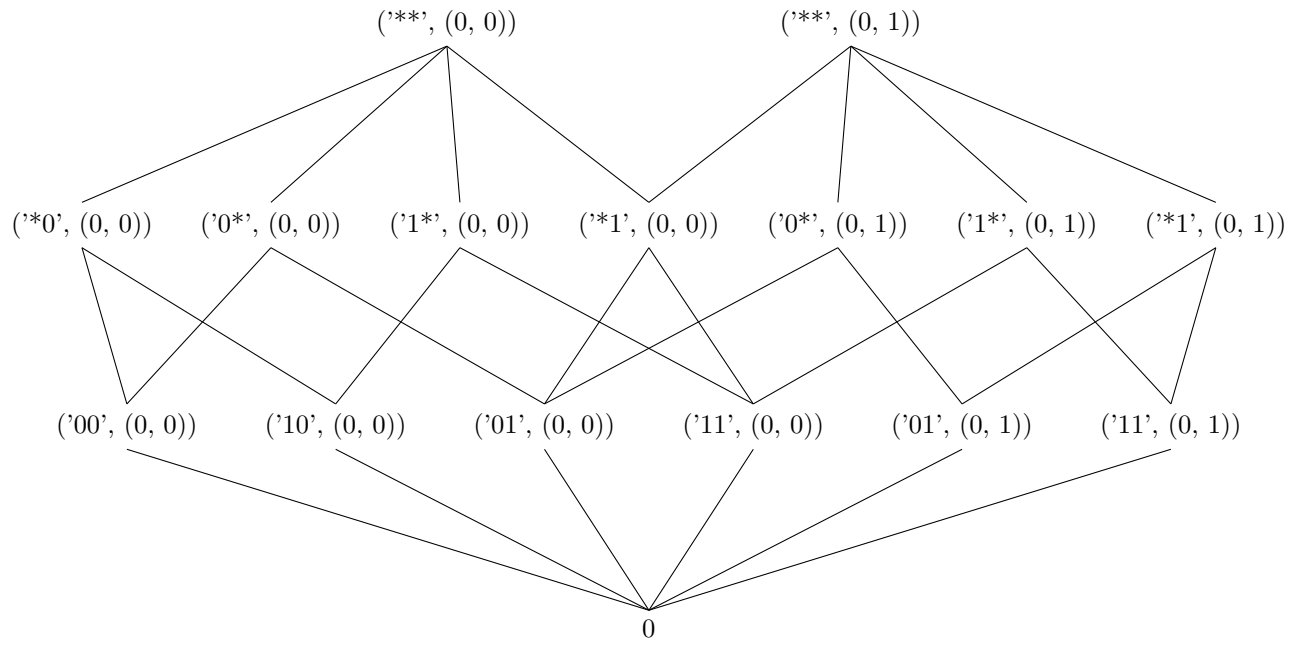
If `orientations[i]` is any other value no gluing is performed for that component.



The poset `GluedCube([-1, 1])`.

```
def Grid(n=2,d=None)
```

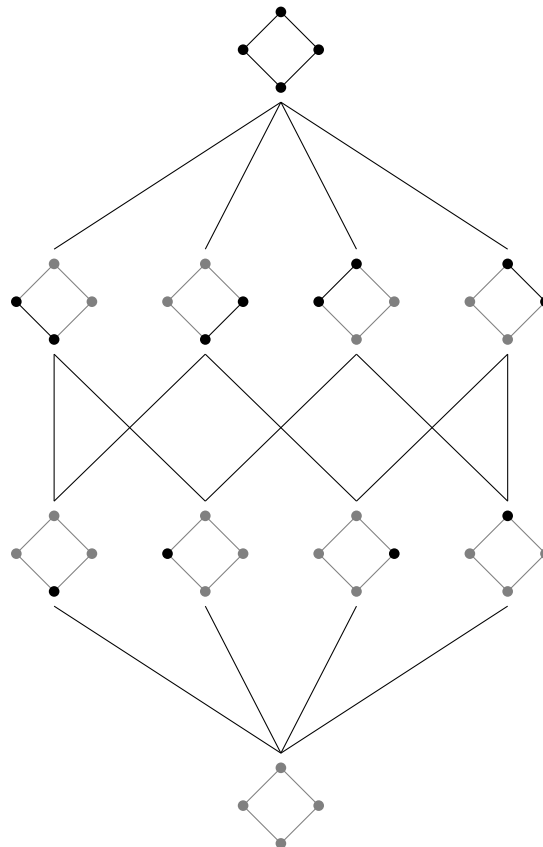
Returns the face poset of the cubical complex consisting of a $d[0] \times \cdots \times d[-1]$ grid of n -cubes.



The poset $\text{Grid}(2, [1, 1])$.

```
def Intervals(P)
```

Returns the lattice of intervals of a given poset (including the empty interval).



The poset $\text{Intervals}(\text{Boolean}(2))$.

When generating a Hasse diagram with `latex()` use the prefix `int_` to control options for the node diagrams.

```
def KleinBottle()
```

Returns the face poset of a cubical complex homeomorphic to the Klein Bottle.

Pseudonym for `GluedCube([-1,1])`.

```
def LatticeOfFlats(data,as_genlatt=False)
```

Returns the lattice of flats given either a list of edges of a graph or the rank function of a (poly)matroid.

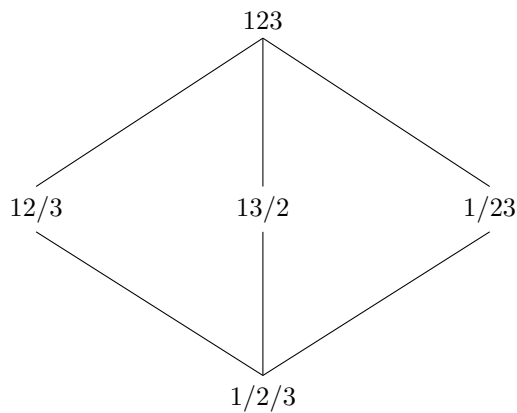
When the input represents a graph it should be in the format `[[i_1,j_1],...,[i_n,j_n]]` where the pair `[i_k,j_k]` represents an edge between `i_k` and `j_k` in the graph.

When the input represents a (poly)matroid the input should be a list of the ranks of sets ordered reverse lexicographically (i.e. binary order). For example, if `f` is the rank function of a (poly)matroid with ground set size 3 the input should be

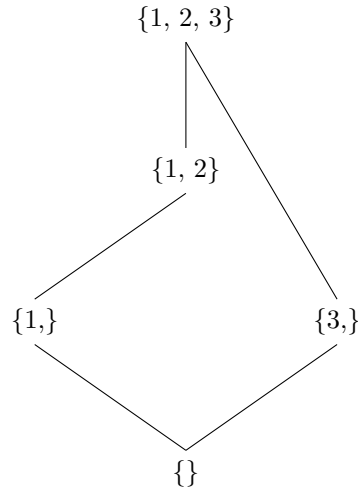
$$[f(\{\}), f(\{1\}), f(\{2\}), f(\{1,2\}), f(\{3\}), f(\{1,3\}), f(\{2,3\}), f(\{1,2,3\})].$$

When `as_genlatt` is `True` the return value is an instance of `Genlatt` with generating set the closures of singletons.

This function may return a poset that isn't a lattice if the input function isn't submodular or a preorder that isn't a poset if the input is not order-preserving.



The poset `LatticeOfFlats([[1,2],[2,3],[3,1]])`.



The poset `LatticeOfFlats([0,1,2,2,1,3,3,3])`.

```
def MinorPoset(L,genL=None, weak=False)
```

Returns the minor poset given a lattice `L` and a list of generators `genL`, or a list of edges specifying a graph.

The join irreducibles are automatically added to `genL`. If `genL` is not provided the generating set will be only the join irreducibles.

If `L` is an instance of the `Poset` class then it is assumed to be a lattice, an instance of `Genlatt` is created from `L` and `genL` and the minor poset of the encoded generator-enriched lattice is returned. In this case the returned poset when plotted with `Poset.latex` has elements represented as generator-enriched lattices.

If `L` is not an instance of the `Poset` class it should be an iterable of length 2 iterables that specify edges of a graph. For example, `L=[[1,2],[2,3],[3,1]]` specifies the 3-cycle graph. The minor poset of the graph is returned. In this case when plotting the returned poset with `Poset.latex` the elements are represented as graphs. Furthermore, there are a few additional options you can use to control the presentation of the graphs in the Hasse diagram:

`G_scale` – Scale of the graph, default is 1.

`G_pt_size` – size in points to use for the vertices, default is 2.

`G_node_options` – Options to place on nodes in the graph, default is ''.

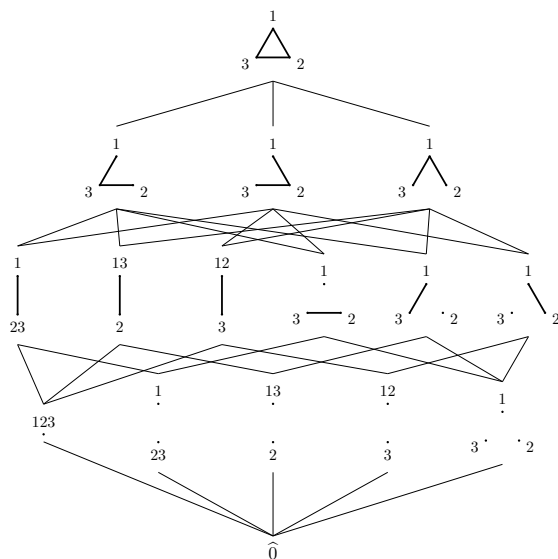
`G_node_sep` – String used to separate names of vertices in the vertex names for minors, default is '/'.

`G_label_dist` – Distance of vertex to its label, default is 1/4.

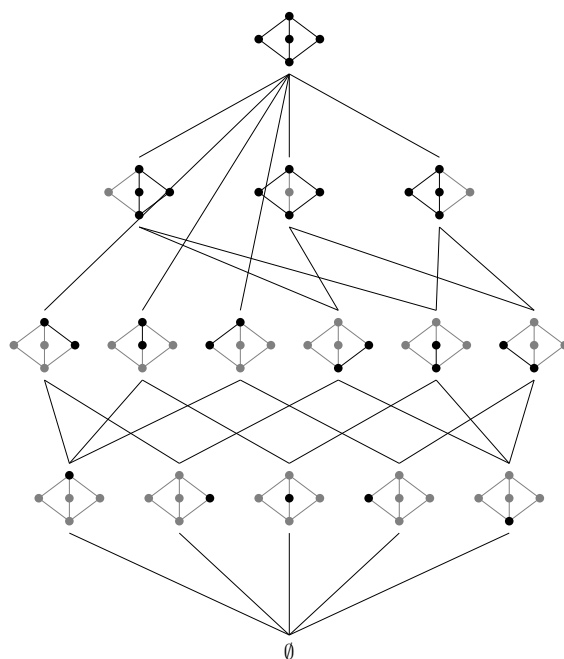
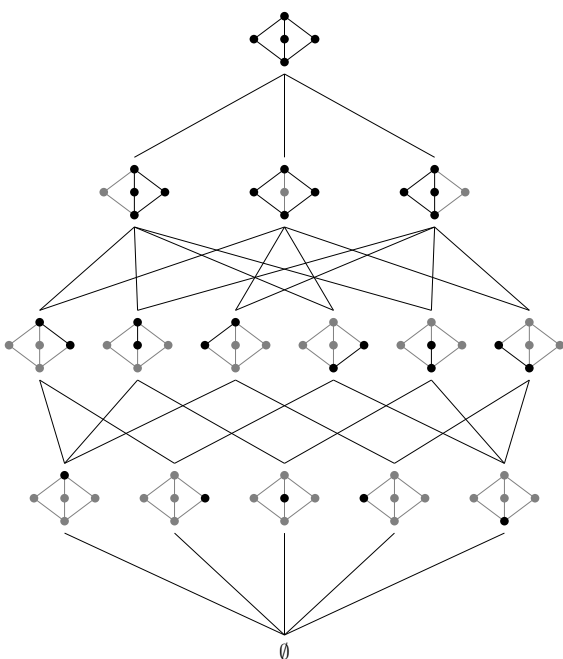
`G_label_scale` – Scale factor for the vertex labels, default is 1.

If `weak` is `True` then the weak minor poset is returned. Briefly, this poset does not have relations $(K, H) \leq (M, I)$ when some generator g was deleted to form (M, I) and $g \leq \widehat{0}_K$.

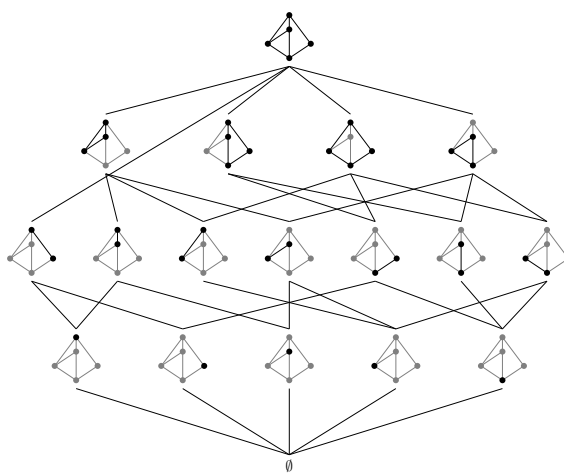
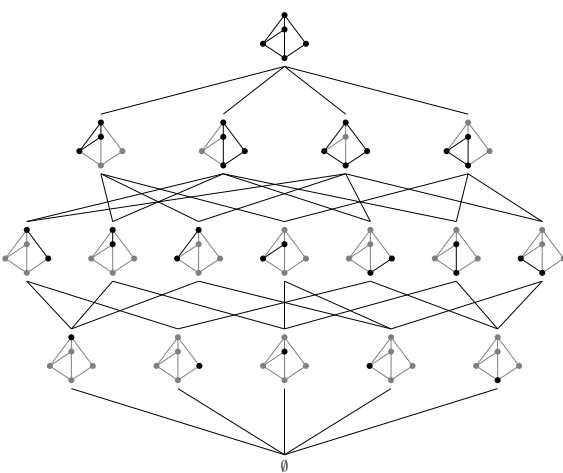
For more info on minor posets see [5].



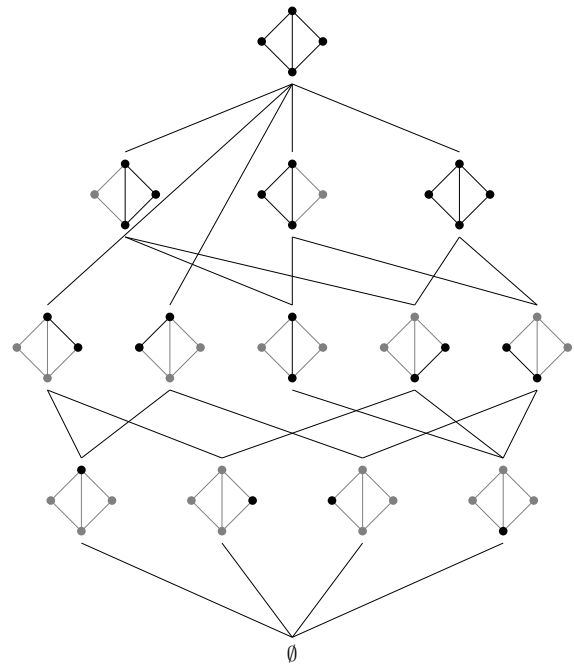
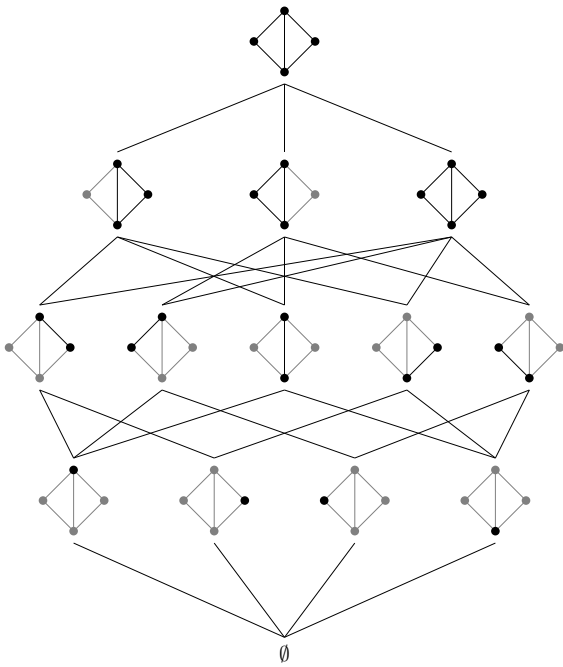
The poset $\text{MinorPoset}([1,2], [2,3], [3,1])$.



On the left the poset $\text{MinorPoset}(\text{LatticeOfFlats}([1,2], [2,3], [3,1]))$ and on the right the poset $\text{MinorPoset}(\text{LatticeOfFlats}([1,2], [2,3], [3,1]), \text{weak}=\text{True})$.



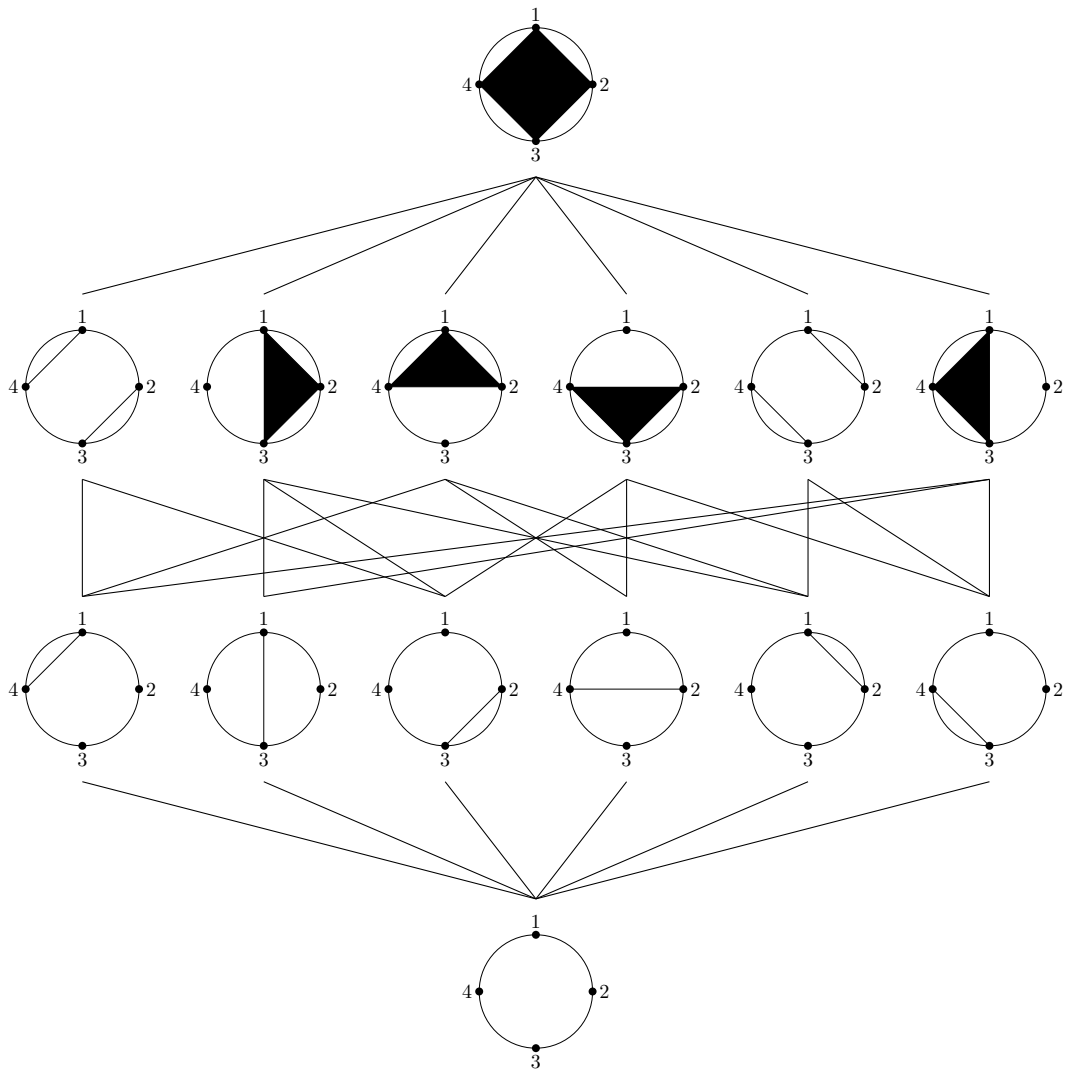
On the left the poset `MinorPoset(LatticeOfFlats([0,1,2,2,1,3,3,3]))` and on the right the poset `MinorPoset(LatticeOfFlats([0,1,2,2,1,3,3,3]), weak=True)`.



On the right the poset `MinorPoset(LatticeOfFlats(Boolean(2), Boolean(2)[1:4]))` and on the left the poset `MinorPoset(LatticeOfFlats(Boolean(2), Boolean(2)[1:4]), weak=True)`.

```
def NoncrossingPartitionLattice(n=3)
```

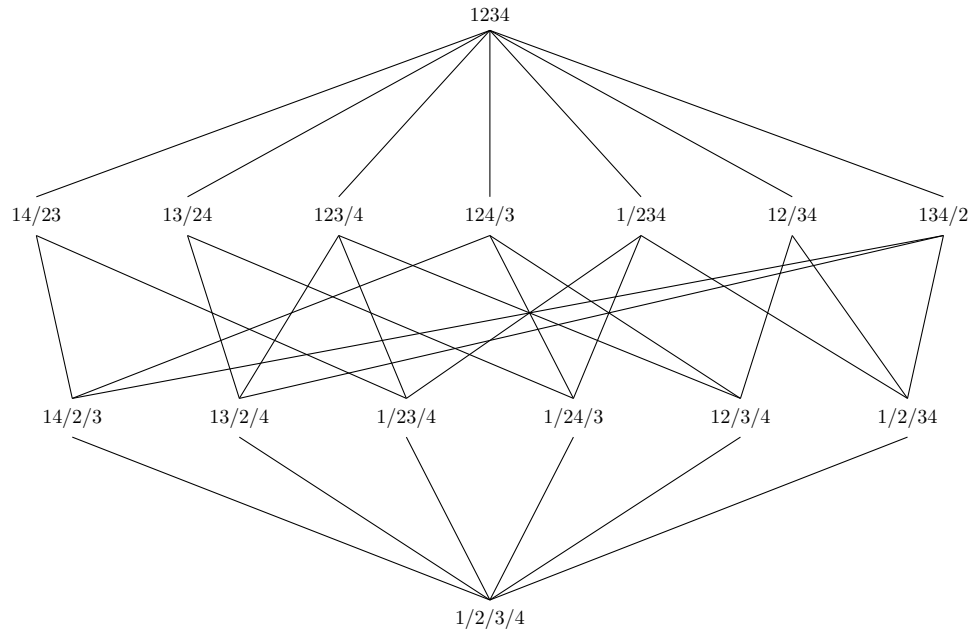
Returns the lattice of noncrossing partitions of $1, \dots, n$ ordered by refinement.



The noncrossing partition lattice NC_4 .

```
def PartitionLattice(n=3)
```

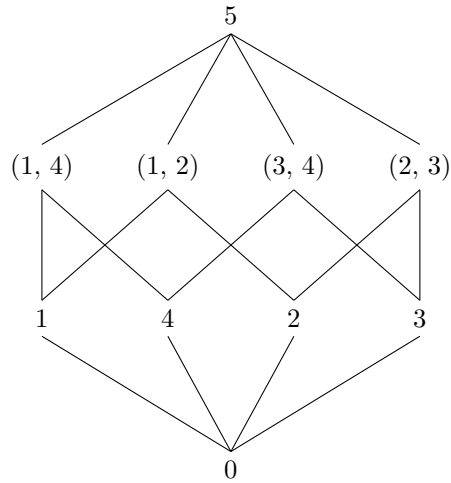
Returns the lattice of partitions of $1, \dots, n$ ordered by refinement.



The partition lattice Π_4 .

```
def Polygon(n)
```

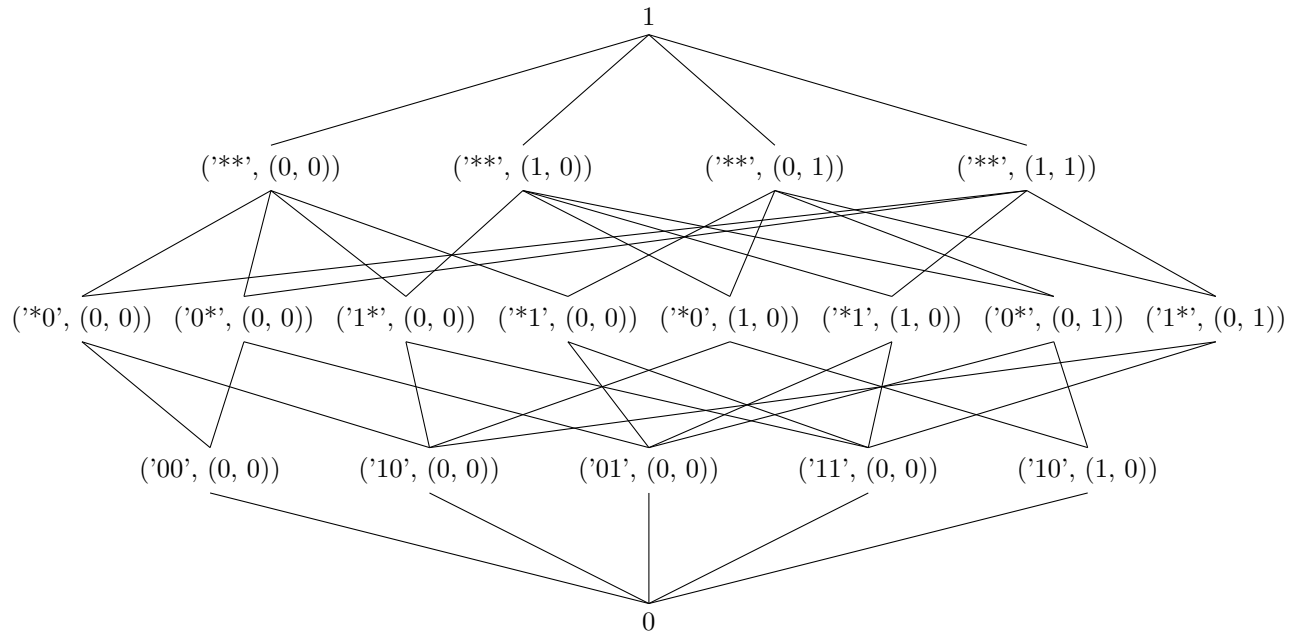
Returns the face lattice of the n -gon.



The poset $\text{Polygon}(4)$.

```
def ProjectiveSpace(n=2)
```

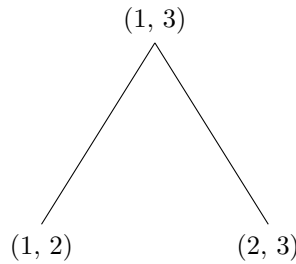
Returns the face poset of a Cubical complex homeomorphic to projective space of dimension n .
Pseudonym for `GluedCube([-1, ..., -1])`.



The poset `ProjectiveSpace(2)`.

```
def Root(n=3)
```

Returns the type A_{n+1} root poset.



The poset `Root(3)`.

```
def Simplex(n)
```

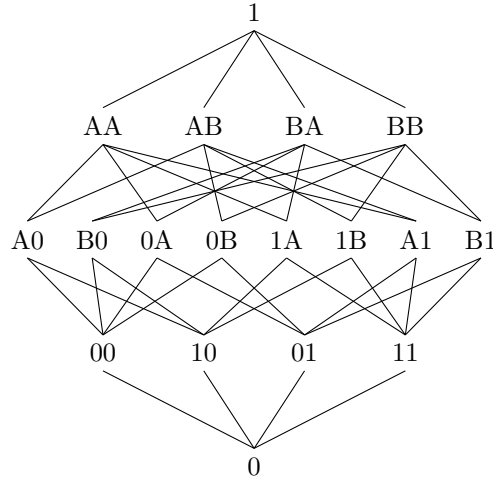
Returns `Boolean(n+1)` the face lattice of the n -dimensional simplex.

```
def Torus(n=2, m=2)
```

Returns the face poset of a cubical complex homeomorphic to the n -dimensional Torus.

This poset is isomorphic to the Cartesian product of n copies of P_m with minimum and maximum adjoined where P_m is the face lattice of an m -gon with its minimum and maximum removed.

Let ℓ_m be the m th letter of the alphabet. When $m \leq 26$ the set is $\{0, 1, \dots, m-1, A, B, \dots, \ell_m\}^n$ and otherwise is $\{0, \dots, m-1, *0, \dots, *[m-1]\}^n$. The order relation is componentwise where $0 < A, \ell_m$ $1 < A, B \dots m-1 < \ell_{m-1}, \ell_m$ for $m \leq 26$, and $0 < *1, *2 \dots m-1 < *[m-1], *0$ for $m > 26$.



The poset $\text{Torus}(2,2)$.

```
def Uncrossing(t, upper=False, weak=False, E_only=False, zerohat=True)
```

Returns either a lower interval $[\widehat{0}, t]$ or the upper interval $[t, \widehat{1}]$ in the uncrossing poset.

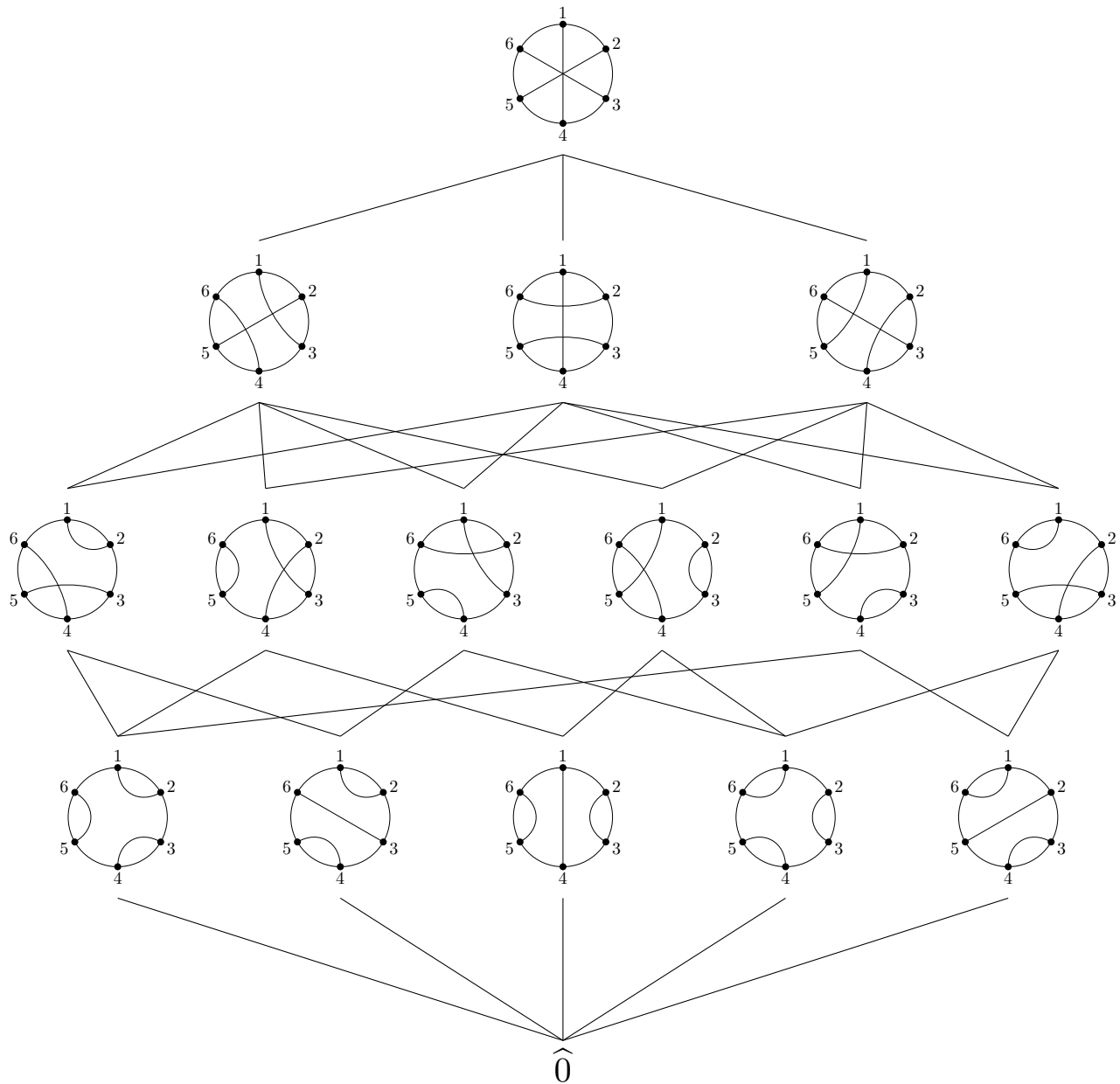
The parameter \mathbf{t} should be either a pairing encoded as a list $[\mathbf{s}_1, \mathbf{t}_1, \dots, \mathbf{s}_n, \mathbf{t}_n]$ where \mathbf{s}_i is paired to \mathbf{t}_i or an integer greater than 1. If \mathbf{t} is an integer the entire uncrossing poset of rank $\binom{t}{2} + 1$ is returned.

Covers in the uncrossing poset are of the form $\sigma < \tau$ where σ is obtained from τ by swapping points i and j to remove a crossing. If **weak** is **True** then the weak subposet is returned that has cover relations $\sigma < \tau$ when σ is obtained from τ by removing a single crossing via swapping two adjacent points. If **E_only** is **True** only swaps (i, j) such that the pairing τ satisfies $\tau(i) < i$ and $\tau(j) < j$ are used. These two flags are provided because this function acts as a backend to **Bruhat**. Calling **Uncrossing**(\mathbf{n} , **E_only**=**True**) constructs the Bruhat order on \mathfrak{S}_n and adding **weak**=**True** constructs the weak order on \mathfrak{S}_n .

If **zerohat** is **False** then no minimum is adjoined.

Raises a **ValueError** when \mathbf{t} is an integer less than 2.

For more info on the uncrossing poset see [7].

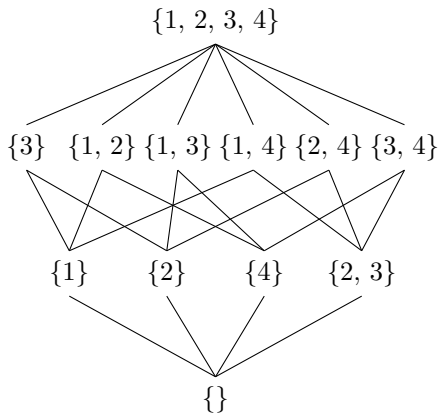


The poset $\text{Uncrossing}(3) == \text{Uncrossing}([1,4,2,5,3,6])$.

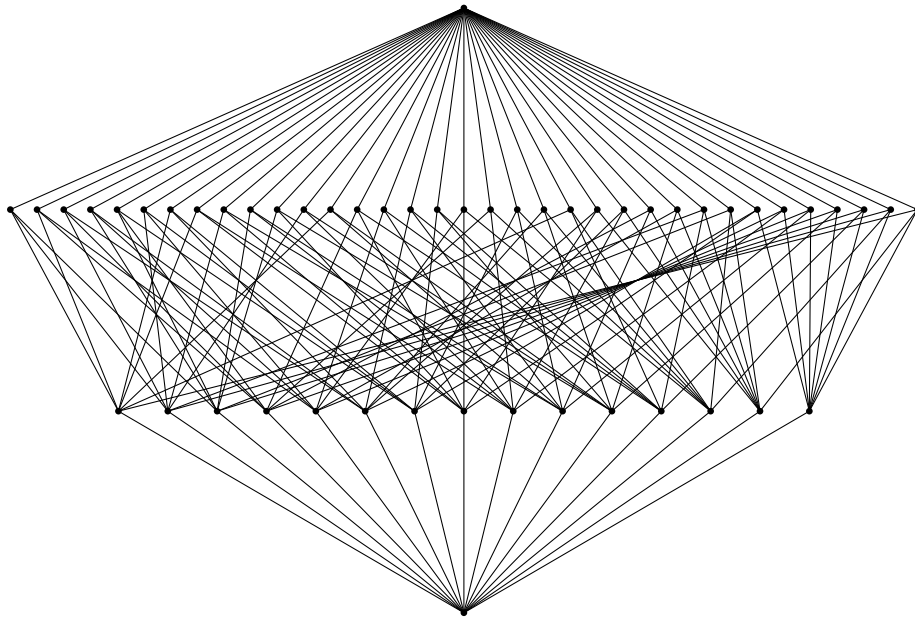
```
def UniformMatroid(n=3,r=3,q=1)
```

Returns the lattice of flats of the uniform (q) -matroid of rank r on n elements.

Currently only implemented for $q=1$ or a prime. Raises an instance of `NotImplementedError` if q is neither 1 nor prime.



The poset `UniformMatroid(4,3)`.



The poset `UniformMatroid(4,3,2)`.

Polynomial

```
class Polynomial
```

A class encoding polynomials in noncommutative variables (used by the `Poset` class to compute the `cd`-index).

This class is basically a wrapper around a dictionary representation for polynomials (e.g. $3\mathbf{ab} + 2\mathbf{bb}$ is encoded as `{'ab':3, 'bb':2}`). The class provides methods for basic arithmetic with polynomials, to substitute a polynomial for a variable in another polynomial and to convert `ab`-polynomials to `cd`-polynomials (when possible) and vice versa. You can also get and set coefficients as if a polynomial were a dictionary.

```
def __add__(*args)
```

Polynomial addition.

Raises `NotImplementedError` if the coefficients can't be added.

```
def __bool__(this)
```

```
def __eq__(this, that)
```

```
def __floordiv__(this, that)
```

```
def __ge__(this, that)
```

Returns `True` if `this` is coefficientwise greater than or equal to `that`.

```
def __getitem__(this, i)
```

```
def __gt__(this, that)
```

Returns `True` if `this` is greater than or equal to `that` coefficientwise and `this` is not equal to `that`.

```
def __init__(this, data=None)
```

Returns a `Polynomial` given a dictionary.

The keys in `data` are the monomials, encoded as strings, and the values are the coefficients. Coefficients can be any class that supports addition and multiplication and such that comparing to 0 returns a boolean.

If `data` is `None` or an empty dictionary then the zero polynomial is returned.

```
def __iter__(this)
```

```
def __le__(this, that)
```

Returns True if `this` is coefficientwise less or equal to `that`.

```
def __len__(this)
```

Returns the number of coefficients.

```
def __lt__(this, that)
```

Returns True if `this` is coefficientwise less or equal to `that` and `this` and `that` or not equal.

```
def __mod__(this,that)
```

```
def __mul__(*args)
```

Noncommutative polynomial multiplication.

```
def __neg__(this)
```

```
def __pow__(this,x)
```

Polynomial exponentiation by non-negative integers.

Raises `NotImplementedError` if either `x` is not an integer or `x<0`.

```
def __add__(*args)
```

Polynomial addition.

Raises `NotImplementedError` if the coefficients can't be added.

```
def __repr__(this)
```

```
def __mul__(*args)
```

Noncommutative polynomial multiplication.

```
def __setitem__(this,i,value)
```

```
def __str__(this)
```

```
def __sub__(this, that)
```

Polynomial subtraction

```
def __truediv__(this,that)
```

```
def _coeff_str(c)
```

```
def _monom_str(m)
```

```
def _poly_add_prepoly(p, q)
```


Internal backend for `__add__`.

```
def _prepoly_mul_poly(q, p)
```

Internal backend for `__mul__`.

```
def abToCd(this)
```

Given an **ab**-polynomial returns the corresponding **cd**-polynomial if possible and the given polynomial if not.

```
def cdToAb(this)
```

Given a **cd**-polynomial returns the corresponding **ab**-polynomial.

```
def strip(this)
```

Removes any zero terms from a polynomial in place and returns it.

```
def sub(this, poly, monom)
```

Returns the polynomial obtained by substituting the polynomial `poly` for the monomial `m` (given as a string) in `this`.

TriangularArray

```
class TriangularArray
```

A class encoding a triangular array.

This class is used to encode the zeta function of a poset.

Constructor arguments:

data – An iterable specifying the entries in the upper diagonal; may be either a flat list or an iterable of iterables.

flat – Whether the data is in flat form or not.

Constructor raises `ValueError` if the number of entries of `data` is not a triangle number.

```
def __eq__(this,that)
```

```
def __getitem__(this, x)
```

Zero based indexing (i,j) gives the element in row i and column j .

The argument x must be a tuple of integers such that $0 \leq x_0 \leq x_1 < n$ where n is the size of the triangular array.

```
def __init__(this, data, flat=True)
```

```
def __repr__(this)
```

```
def __setitem__(this, idx, x)
```

```
def __str__(this)
```

```
def col(this, j)
```

Generator for the i th column.

```
def inverse(this)
```

Returns the inverse of the triangular array considered as an upper triangular matrix.

Raises `ZeroDivisionError` if a diagonal entry of the array is zero.

```
def revtranspose(this)
```

Returns a new instance of `TriangularArray` obtained by transposing and reversing all columns and rows.

```
def row(this, i)
```

Returns the i th row as a list.

```
def subarray(this, S)
```

Returns a sub-triangular array by selecting the rows and columns indexed by `S`.

ZetaHasseDiagram

```
class ZetaHasseDiagram(SubposetsHasseDiagram)
```

Class to draw the Hasse diagram of a poset as principal filters (or ideals) labeled by the zeta function values.

This is a convenience class that merely passes appropriate options to `SubposetsHasseDiagram`. When `latex` is called this class produces latex code for the Hasse diagram of the given poset P with each element p as the principal filter $\{q \in P : q \geq p\}$ (or optionally the principal ideal) with each element q in the filter labeled by $\zeta(p, q)$.

This class is intended for representing quasigraded posets, those with a zeta function taking values other than 0 and 1.

Constructor arguments:

filters – Whether to represent elements by the associated principal filter or alternatively as ideals. The default value is `True` which will use filters.

keep_ranks – Whether to use the same rank values for elements in the filters/ideals drawn as in the given poset. If this argument is `False` then a new poset is created with rank function the standard length function as returned by `Poset.make_ranks`.

See `SubposetsHasseDiagram` for details on other arguments. Note, the argument `prefix` to `SubposetsHasseDiagram` defaults to `'V'`.

Note, if `V_width` (or `V_height`) is not provided (assuming the default value `'V'` for `prefix`) it is set to one fifth of `width` (or `height`). If `V_nodyscale` is not provided it is set to 0.5.

```
def __init__(this, P, filters=True, prefix='V', keep_ranks=True, func_args=None,
**kwargs)
```

See `ZetaHasseDiagram`.

References

- [1] Margaret M Bayer. The cd -index: a survey. In *Polytopes and discrete geometry*, volume 764 of *Contemp. Math.*, pages 1–19. Amer. Math. Soc., Providence, RI, 2021.
- [2] Louis J Billera and Richard Ehrenborg. Monotonicity of the cd -index for polytopes. *Math. Z.*, 233(3):421–441, 2000.
- [3] Garrett Birkhoff. *Lattice theory*. American Mathematical Society Colloquium Publications, Vol. XXV. American Mathematical Society, Providence, R.I., third edition, 1967.
- [4] Richard Ehrenborg, Mark Goresky, and Margaret Readdy. Euler flag enumeration of whitney stratified spaces. *Adv. Math. (N. Y.)*, 268:85–128, January 2015.
- [5] William Gustafson. Lattice minors and eulerian posets. *Theses and Dissertations–Mathematics*, 96, 2023.
- [6] Martina Juhnke-Kubitzke, José Alejandro Samper, and Lorenzo Venturello. The cd -index of semi-eulerian posets. *arXiv [math.CO]*, May 2024.
- [7] Thomas Lam. The uncrossing partial order on matchings is eulerian. *J. Combin. Theory Ser. A*, 135:105–111, 2015.
- [8] Richard P Stanley. *Enumerative combinatorics. Volume 1*, volume 49 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, second edition, 2012.