Lab-1

> 作者：**Alex**
>
> 联系方式：`wang-zx23@mails.tsinghua.edu.cn`

数理逻辑与集合论是计算机科学最基本的数学基础，其抽象程度非常之高（x）；尤其是集合论中的关系、等价类等概念非常抽象。笔者在学完贵系开设的离散（1）后对这个学科依然一头雾水，直到做完这个如下几个小lab才发现原来集合论原来可以这么有趣和具象。本笔记是对`头歌实践教学平台-离散数学实验`AC代码的一个总结，供大家参考。实验代码完整仓库在这里.

```
https://github.com/wannabeyourfriend/THU-CST-DiscreteMath-2024-2025/tree/main/Lab-1
```


- 集合论 - **SetLab**
  - 集合表示、性质、运算
  - 自然数
- 关系 - **RelationLab**

    - 定义、运算
    - 闭包
- 数理逻辑 - **LogicLab**
    - 命题逻辑
    - 一阶谓词逻辑
- 布尔运算 - **BoolLab**
    - 真值表
    - 布尔代数
    - 电路模拟

1 SetLab

setlab包括两个实验，代码实现在`number.py`和`set.py`两个文件中

1.1 Set

1.1.1 Set basic

Python提供了两种数据类型来建模数学中的集合：

- `set`：这是一个`mutable`非标量数据类型，用无序的方式组织一组有限的、可区分的、`immutable`（`hashable`）对象。
- `frozenset`：这是一个`immutable`、`hashable`的非标量数据类型，用无序的方式组织一组有限的、可区分的、`immutable`（`hashable`）对象。
- Python提供了预定义函数将任一迭代对象，如List、Tuple、Dictionary对象转换为一个Set对象: `set(iterable)`
- 集合可以用列表推导式创建，可以用来去除重复元素，还可以在for循环里面迭代集合元素

注意！`set`的元素不可以是`set`,但可以是`frozenset`

`set`上的方法有：

方法	描述	示例

add()	向集合对象添加一个元素	`>>>box = {'apple', 'orange', 'banana'}`
		`>>>box.add('apricot')`
		`>>>box`
		`{'orange', 'apple', 'banana', 'apricot'}`

clear()	删除集合对象中所有元素	`>>>box = {'apple', 'orange', 'banana'}`
		`>>>box.clear()`
		`set()`

copy()	返回集合对象的一个拷贝。因为 Set 也是 mutable 对象，通过赋值并不会创建一个新 Set 对象。	`>>>s1={1,2,3}`
		`>>>s2 = s1`
		`>>>s1.add(4)`
		`>>>s1`
		`{1,2,3,4}`
		`>>>s2`
		`{1,2,3,4}`
		`>>>s2=s1.copy()`
		`>>>s1.add(5)`
		`>>>s1`
		`{1,2,3,4,5}`
		`>>>s2`
		`{1,2,3,4}`

difference()	返回两个集合的差，s1.difference(s2)等同于 s1-s2。注意会新建一个集合，s1、s2 不会被修改	`>>>{1, 2, 3, 4}.difference({2, 3, 5})`
		`{1, 4}`
		`>>>{1, 2, 3, 4} - {2, 3, 5}`
		`{1, 4}`

difference_update()	得到两个集合的差，结果直接保存在 s1，即 s1.difference_update(s2) 效果等同于 s1 = s1.difference(s2)	`>>> s1 = {1, 2, 3, 4}`
		`>>> s2 = {2, 3, 5}`
		`>>> s1.difference_update(s2)`
		`>>> s1`
		`{1, 4}`
		`>>> s2`
		`{2, 3, 5}`

discard()	删除指定的元素，如被删除的元素不在集合中，不报错	`>>>s={1,2,3}`
		`>>>s.remove(1)`
		`>>>s`
		`{2, 3}`
		`>>>s.remove(4)`

intersection	返回两个集合的交，s1. intersection(s2)等同于 s1&s2。注意会新建一个集合，s1、s2 不会被修改	`>>> {1, 2, 3, 4, 5}.intersection({3, 4, 5, 6})`
		`{3, 4, 5}`
		`>>>{1, 2, 3, 4, 5} & {3, 4, 5, 6}`
		`{3, 4, 5}`

intersection_update()	得到两个集合的交，结果直接保存在 s1，即 s1. intersection_update(s2) 效果等同于 s1 = s1. intersection(s2)	`>>> s1 = {1, 2, 3, 4, 5}`
		`>>> s2 = {3, 4, 5, 6}`
		`>>> s1.intersection_update(s2)`
		`>>> s1`
		`{3, 4, 5}`
		`>>> s2`
		`{3, 4, 5, 6}`

isdisjoint()	判断两个集合是否相交	`>>>{1, 2}.isdisjoint({3, 4})`
		`True`
		`>>>{1, 2}.isdisjoint({1, 4})`
		`False`

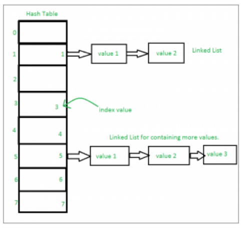
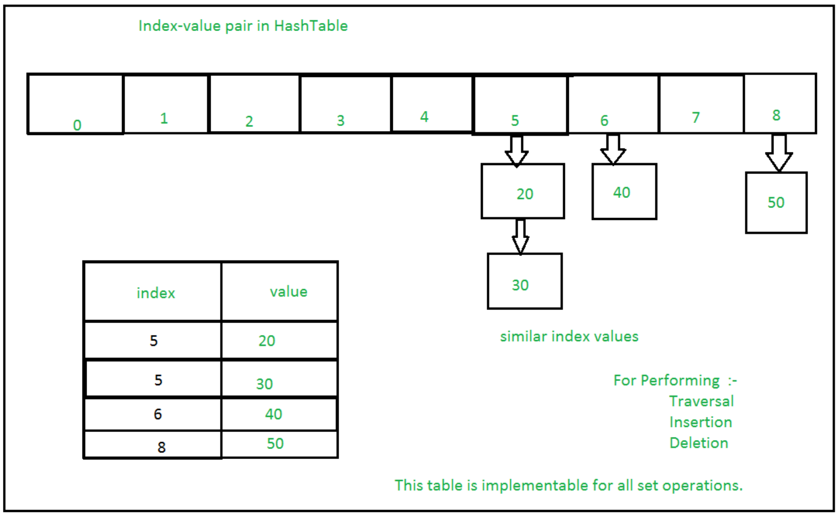
issubset()	判断该集合是否为另一集合的子集，与<=运算符等效	>>>{1, 2}.issubset({1, 2, 3}) True >>>{1, 2} <= {1, 2, 3} True
issuperset()	判断该集合是否包含另一集合，与>=运算符等效	>>>{1, 2}.isdisjoint({3, 4}) True >>>{1, 2}.isdisjoint({1, 4}) False
pop()	删除并返回集合中任意一个元素，但由于 Set 对象是无序的，所以不知道将会删除哪个。	>>>box = {'apple', 'orange', 'banana'} >>>box.pop() 'banana'
remove()	删除指定的元素，如被删除的元素不在集合中，将抛出一个异常。	>>>s={1,2,3} >>>s.remove(1) >>>s {2, 3} >>>s.remove(4) Traceback (most recent call last): File "<stdin>", line 1, in <module> KeyError: 4
symmetric_difference()	返回两个集合的对称差，s1.symmetric_difference(s2)等同于 s1^s2。注意会新建一个集合，s1、s2 不会被修改	>>>{1,2,3,4}.symmetric_difference({2,3,5}) {1, 4, 5} >>>{1,2,3,4} ^ {2, 3, 5} {1, 4, 5}

symmetric_difference_update()	得到两个集合的对称差，结果直接保存在 s1，即 s1.symmetric_difference_update(s2)效果等同于 s1 = s1.symmetric_difference(s2)	>>> s1 = {1,2,3,4}
		>>> s2 = {2,3,5}
		>>> s1.symmetric_difference_update(s2)
		>>> s1
		{1, 4, 5}
		>>> s2
		{2, 3, 5}

union()	返回两个集合的并，s1. intersection(s2)等同于 s1|s2。注意会新建一个集合，s1、s2 不会被修改	>>>{1,2,3,4,5}.union({3,4,5,6}) {1,2,3,4,5,6} >>>{1,2,3,4,5} | {3,4,5,6} {1,2,3,4,5,6}
update()	用本集合与另一集合的并运算结果更新本集合，通常用于向集合中一次加入多个元素	>>>box = {'apple', 'orange', 'banana'} >>>box.update(['apricot', 'mango', 'grapefruit']) >>>box {'orange', 'apple', 'mango', 'banana', 'apricot', 'grapefruit'}
in	用于判断某个元素是否属于某个集合	>>>2 in {1, 2, 3} True >>>4 in {1, 2, 3} False
len、min、max、sum 等	其他可用于非标量数据类型的操作仍可用于 Set 对象	>>>box={'apple', 'orange', 'apple', 'pear', 'orange', 'banana'} >>>len(box) 4 >>>max(box) 'pear' >>>min(box) 'apple' >>>sum({1, 2, 3}) 6

1.1.2 Python Set implementions

In Python Sets are implemented using a dictionary with dummy variables, where key beings the members set with greater optimizations to the time complexity.

<table>
<tr><th>1</th><th>2</th></tr>
<tr><td></td><td>
</td></tr>
</table>

Operation	Average case	Worst Case	notes
x in s	O(1)	O(n)	
Union s|t	O(len(s)+len(t))		
Intersection s&t	O(min(len(s), len(t))	O(len(s) * len(t))	replace “min” with “max” if t is not a set
Multiple intersection s1&s2&..&sn		(n-1)*O(l) where l is max(len(s1),..,len(sn))	
Difference s-t	O(len(s))		

1.1.3 Set Operations

Operators	Notes
key in s	containment check
key not in s	non-containment check
s1 == s2	s1 is equivalent to s2
s1 != s2	s1 is not equivalent to s2
s1 <= s2	s1 is subset of s2
s1 < s2	s1 is proper subset of s2
s1 >= s2	s1 is superset of s2
s1 > s2	s1 is proper superset of s2
s1 | s2	the union of s1 and s2
s1 & s2	the intersection of s1 and s2

Operators	Notes
s1 – s2	the set of elements in s1 but not s2
s1 ˆ s2	the set of elements in precisely one of s1 or s2

1.1.4 Set PAs

> PA1-T3:实现幂集

```
def powSet(S):
    if not S:
        return {frozenset()}
    element = S.pop()
    subsets = powSet(S)
    new_subsets = {subset | frozenset([element]) for subset in subsets}
    return subsets | new_subsets
```

> PA1-T4: 实现n个有限集合的笛卡尔乘积

```
"""
实际上有现成的计算笛卡尔乘积的包
from itertools import product
def DescartesProduct(*args):
    # 使用 itertools.product 计算笛卡尔积
    return set(product(*args))
"""
def DescartesProduct(*args):
    a = []
    for s in args:
        a.append([x for x in s])
    a = DescartesProduct2([], a)
    b = set()
    for i in a:
        b.add(tuple(i))
    return b

def DescartesProduct2(list1, list2):
    if len(list2) == 0:
        return list1
    if len(list1) == 0:
        for x in list2[0]:
            list1.append([x])
        return DescartesProduct2(list1, list2[1:])
    nlist = []
    for i in list2[0]:
        for x in list1:
            a = [j for j in x]
            a.append(i)
            nlist.append(a)
    return DescartesProduct2(nlist, list2[1:])
```

2.2 number

2.2.1 $\mathbb{N}$ Constructions

Peano公理（又称佩亚诺公理）是一组定义自然数及其基本性质的公理系统，通常用于构建算术的基础。它由意大利数学家Giuseppe Peano在1889年提出。Peano公理的目的是从一些最基本的假设出发，推导出自然数的所有性质。

1.零是一个自然数： 0 是自然数。
2.每个自然数都有一个后继数： 对于每个自然数n，都有一个唯一的后继数（记作 S(n)）。
3.零不是任何自然数的后继数： 0 不是任何自然数的后继数。
4.后继数是唯一的： 如果两个自然数的后继数相同，那么这两个自然数本身是相同的。
5.归纳原理： 如果某个属性对于0成立，并且假设它对一个自然数n成立时，它对n的后继数S(n)也成立，那么这个属性对于所有自然数都成立。

这些公理构建了自然数的基本性质，如加法和乘法等操作都可以通过这些公理推导出来。

对于任意的集合 A, 定义$A^+ = A \cup \{A\}$为集合A的后继.集合$0 = \emptyset$是一个自然数.根据这个定义，可以得到各个自然数:

$$
\begin{aligned}
0 &= \emptyset \\
1 = 0^+ &= 0 \cup \{0\} = \{0\} \\
2 = 1^+ &= 1 \cup 1 = \{0, 1\} \\
&\cdots
\end{aligned}
$$

使用如下代码构造自然数

```
class NaturalNumber(object):
    def __init__(self, pre):
        self.pre = pre
```

在代码`number.py`文件中实现了自然数的输出、加法、乘法、矩阵转换等算子:

```
class NaturalNumber(object):
    def __init__(self, pre):
        self.pre = pre

    def __str__(self):
        result = ''
        if self.pre == None:
            result = "Zero"
        elif self.pre.pre == None:
            result = "Succ Zero"
        else:
            pre = self.pre
            result = "Succ(" + pre.__str__() + ")"
        return result

    def __add__(self, other):
        # a + zero = a
        # a + succ(b) = succ(a + b)
        if other.pre == None:
            return self
        else:
```

```
            return NaturalNumber(self + other.pre)

    def __mul__(self, other):
        # a * zero = zero
        # a * succ(b) =  a * b + a
        if other.pre == None:
            return NaturalNumber(None)
        else:
            return self * other.pre + self

    def toNumber(self):
        if self.pre == None:
            return 0
        else:
            return self.pre.toNumber() + 1

def succ(n):
    return NaturalNumber(n)
```

2.2.2 $\mathbb{N}$ Isomorphic Sequences via Functional Operators

实际上，自然数还可以用函数来定义：$n = foldn(zero, succ, n)$,我们称foldn是自然数域上的叠加操作，其中succ是自然数上的函数，n是叠加的次数

> 在`foldn`函数的基础上，我们进一步定义`f(n)=foldn2(init,h)(n)`

$(+m) = foldn2(m, succ)$
描述了将自然数n增加m的操作，将它依次作用到自然数上可以产生和自然数同构的序列$m, m+1, m+2, ..., n+m, ...$

$(\cdot m) = foldn2(0, (+m))$
描述了将自然数n乘以m的操作，将它依次作用到自然数上可以产生和自然数同构的序列$0, m, 2m, 3m, ..., nm, ...$

$m^{()} = foldn(1, (\cdot m))$
描述了对自然数m取n次幂的操作，将它依次作用到自然数上可以产生和自然数同构 的序列$1, m, m^2, m^3, ..., m^n, ...$

笔者的理解是$foldn$是传入的函数h上的泛函，将其作用到自然数上可以产生一列与自然数同构的序列。实际上不懂泛函为何物（x）请大家多多指正。

```
def foldn2(init, h):
    def f(n: NaturalNumber):
        if n.pre is None:
            return init
        else:
            return h(foldn2(init, h)(n.pre))
    return f
```

2 FunctionLab

2.1 Max Set Theory

> 在集合论和映射理论中，固定点、不动点以及自映射是重要的基本概念。考虑一个从集合 A 到集合 A 的函数 $f: A \to A$，我们希望通过该映射寻找一个特殊的子集 S，该子集具有某种性质：通过递归过程和逐步移除元素，最终形成一个满足特定映射关系的子集。

> 1. **闭包运算**：如传递闭包、依赖闭包等问题中，我们需要找出在映射下不可进一步简化的集合。这一过程本质上是求解一个“最大子集”，该子集在映射下保持封闭性。
> 2. **图论问题**：如强连通分量的检测，或者在图的遍历过程中，寻找与映射关系相关的最大子图（例如，满足某些映射不变性的子图）。
> 3. **递归依赖关系分析**：例如，在数据库中对依赖关系进行分析，寻找稳定的或最大依赖关系集，这有助于优化查询性能和确保数据一致性。
>
> 最大子集问题：

```
"""
伪代码提示
Mapping(A, f)算法
输入：
集合A；
集合A到集合A的映射f；
输出：
满足条件的子集S；
"""
 if 集合A只有1个元素 then
    return A
 else if 能找到一个没有其他元素映射到其上的元素，设为k then
    A = A - {k};
    f =从f中删除所有包含k的映射关系；
    return Mapping(A, f);
 else
    return A;
 end if
```

解答

```
def findNomap(A, f):
    for a in A:
        mapped = False
        for m in f:
            if m[1] == a:
                mapped = True
                break
        if mapped == False:
            return a
    return None

def mapping(A, f):
    if len(A) == 0:
        return None
    if len(A) == 1:
        return A
    unmapped_element = findNomap(A, f)
    if unmapped_element is not None:
        for m in f:
            if m[0] == unmapped_element:
                f.remove(m)
        A.remove(unmapped_element)
        return mapping(A, f)
```

```
    return A
```

2.2 Selection Sort

> `selectsort`

```
def SelectSort(seq, i):
    if i == 0:
        return
    max_j = i
    for j in range(i):
        if seq[j] > seq[max_j]:
            max_j = j
    seq[i], seq[max_j] = seq[max_j], seq[i]
    SelectSort(seq, i - 1)
```

3 RelationLab

3.1 Relation modeling

3.1.1 Data structure

这个Lab要求使用**OOP**的编程思想定义集合上的二元关系。回顾关系的定义：对集合A和集合B, $A \times B$的任意子集称为$A \to B$的一个二元关系R。若$< x, y > \in R$,记作xRy.我们采用二元序偶来建模关系。

```
import functools

class Relation(object):
    def __init__(self, sets, rel):
        #rel为sets上的二元关系
        assert not(len(sets)==0 and len(rel) > 0) #不允许sets为空而rel不为空
        assert sets.issuperset(set([x[0] for x in rel]) | set([x[1] for x in 
rel])) #不允许rel中出现非sets中的元素
        self.rel = rel
        self.sets = sets

    def __str__(self):
        relstr = '{}'
        setstr = '{}'
        if len(self.rel) > 0:
            relstr = str(self.rel)
        if len(self.sets) > 0:
            setstr = str(self.sets)
        return 'Relation: ' + relstr + ' on Set: ' + setstr

    def __eq__(self, other):
        return self.sets == other.sets and self.rel == other.rel
```

3.1.2 Relation Operations

- 实现恒等关系I_A
- 实现关系的合成运算$R: X \to Y \quad S: Y \to Z$的合成关系为$T = S \circ R: X \to Z$,但在这里，我们在同一个集合$A$上实现关系的合成.
- 实现关系的幂运算 $R^n = R \circ R \cdots \circ R \qquad \text{where} \quad n \geq -1$
- 实现关系矩阵
- 判断关系的性质
  - 自反：$\forall a \in A, (a, a) \in R$
  - 反自反：$\forall a \in A, (a, a) \notin R$
  - 对称：$\forall a, b \in A,\ (a, b) \in R \implies (b, a) \in R$
  - 反对称：$\forall a, b \in A,\ ((a, b) \in R \wedge (b, a) \in R) \implies a = b$
  - 传递：$\forall a, b, c \in A,\ ((a, b) \in R \wedge (b, c) \in R) \implies (a, c) \in R$
- 等价关系：满足自反性、对称性和传递性的关系
- 等价类：A上的等价关系R,则集合中的任意元素$a \in A$，等价类为$[a]_R = \{x \in A \mid (a, x) \in R\}$，$A = \bigcup_{[a]_R \in \mathcal{P}} [a]_R$, 其中$\mathcal{P}$是所有等价类的集合.

以上代码实现均在`relation.py`中:

```
class Relation(object):
    def __init__(self, sets, rel):
        #rel为sets上的二元关系
        assert not(len(sets)==0 and len(rel) > 0) #不允许sets为空而rel不为空
        assert sets.issuperset(set([x[0] for x in rel]) | set([x[1] for x in 
rel])) #不允许rel中出现非sets中的元素
        self.rel = rel
        self.sets = sets

    def __str__(self):
        relstr = '{}'
        setstr = '{}'
        if len(self.rel) > 0:
            relstr = str(self.rel)
        if len(self.sets) > 0:
            setstr = str(self.sets)
        return 'Relation: ' + relstr + ' on Set: ' + setstr

    def __eq__(self, other):
        #判断两个Relation对象是否相等，关系及集合都要相等
        return self.sets == other.sets and self.rel == other.rel

    def diagonalRelation(self):
        #返回代表IA的Relation对象
        return Relation(self.sets, set([(a, a) for a in self.sets]))

    def __mul__(self, other):
        assert self.sets == other.sets
        #实现两个关系的合成，即self*other表示other合成self。请注意是先看other的序偶
        #返回合成的结果，为一个Relation对象
```

```
        return Relation(self.sets, set([(x, z) for (x, y1) in other.rel for (y2,
z) in self.rel if y1 == y2]))

    def __pow__(self, power, modulo=None):
        assert power >= -1
        # 实现同一关系的多次合成，重载**运算符，即self*self*self=self**3
        # 在每个分支中返回对应的结果，结果是一个Relation对象
        if power == -1:
            return Relation(self.sets, set([(x[1], x[0]) for x in self.rel]))
        elif power == 0:
            return self.diagonalRelation()
        else:
            return self**(power-1) * self

    def __add__(self, other):
        assert self.sets == other.sets
        #实现两个关系的并运算，重载+运算符，即self+other表示self并other
        #请注意，是Relation对象rel成员的并返回结果为一个Relation对象
        return Relation(self.sets, self.rel.union(other.rel))

    def toMatrix(self):
        #将序偶集合形式的关系转换为矩阵。
        #为保证矩阵的唯一性，需对self.sets中的元素先排序
        matrix = []
        elems = sorted(list(self.sets))
        line = [0]*len(self.sets)
        for elem in elems:
            #实现转换为矩阵的功能
            tups = [x for x in self.rel if x[0] == elem]
            for item in tups:
                line[elems.index(item[1])] = 1
            matrix.append(line)
            line = [0]*len(self.sets)
        return matrix

    def isReflexive(self):
        #判断self是否为自反关系，是则返回True，否则返回False
        for a in self.sets:
            if not((a, a) in self.rel):
                return False
        return True

    def isIrreflexive(self):
        # 判断self是否为反自反关系，是则返回True，否则返回False
        for a in self.sets:
            if (a, a) in self.rel:
                return False
        return True

    def isSymmetric(self):
        # 判断self是否为对称关系，是则返回True，否则返回False
        for (a, b) in self.rel:
            if not ((b, a) in self.rel):
                return False
        return True
```

```
    def isAsymmetric(self):
        # 判断self是否为非对称关系，是则返回True，否则返回False
        for (a, b) in self.rel:
            if (b, a) in self.rel:
                return False
        return True

    def isAntiSymmetric(self):
        # 判断self是否为反对称关系，是则返回True，否则返回False
        for (a, b) in self.rel:
            if (b, a) in self.rel:
                if not (a == b):
                    return False
        return True

    def isTransitive(self):
        # 判断self是否为传递关系，是则返回True，否则返回False
        for (a, b) in self.rel:
            tempR = [(x, y) for (x, y) in self.rel if x == b]
            if len(tempR) > 0:
                for (b, c) in tempR:
                    if not ((a, c) in self.rel):
                        return False
        return True

    def reflexiveClosure(self):
        #求self的自反闭包，注意使用前面已经重载过的运算符
        #返回一个Relation对象，为self的自反闭包
        return self + self.diagonalRelation()

    def symmetricClosure(self):
        # 求self的对称闭包，注意使用前面已经重载过的运算符
        # 返回一个Relation对象，为self的对称闭包
        return self + self**-1

    def transitiveClosure(self):
        closure = self
        # 求self的传递闭包，注意使用前面已经重载过的运算符
        # 该方法实现的算法：严格按照传递闭包计算公式求传递闭包
        for power in range(2, len(self.sets) + 1):
            closure = closure + self ** power
        return closure

    def transitiveClosure3(self):
        #该方法利用Roy-Warshall计算传递闭包
        #现将关系转换为矩阵，再调用__warshall函数
        m = self.toMatrix()
        return self.__warshall(m)
```

3.1.4 Order Properties of Relations

- 偏序/半序/弱偏序：满足反对称，自反，传递，部分元素之间的关系可以不在其中
- 拟序/强偏序：非自反、反对称、传递
- 全序：偏序且要求对所有元素皆可以比较

3.1.3 Algorithms

PA1: Warshall algorithm for transitive closure

Warshall-Roy算法

> 对于关系R，返回它的传递闭包R'

- **Formulation:**

$$
\begin{aligned}
&Step1: A^{(0)} = A \\
&Step2: A^{(k)} = A^{(k-1)} \cup (A^{(k-1)}[i][k] \wedge A^{(k-1)}[k][j]) \\
&or: a_{ij}^{(k)} = a_{ij}^{(k-1)} \vee (a_{ik}^{(k-1)} \wedge a_{kj}^{(k-1)}) \\
&Step3: A^{(n)}
\end{aligned}
$$

- **Time complexity:** $\mathcal{O}(n^3)$
- **code**

```python
def __warshall(self, a):
        assert (len(row) == len(a) for row in a)
        n = len(a)
        #参数a：为一个关系矩阵
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    a[i][j] = a[i][j] or (a[i][k] and a[k][j])
        return a
```

PA2: Generate equivalence

> 给定集合A，返回A在关系R下的商集

```python
def isEquivalenceRelation(rel):
    if rel.isReflexive() and rel.isSymmetric() and rel.isTransitive():
        return True
    else:
        return False

def createPartition(rel):
    if not isEquivalenceRelation(rel):
        print("The given relation is not an Equivalence Relation")
        return set([])
    partition = set([])
    for a in rel.sets:
        partition.add(frozenset(y for (x, y) in rel.rel if x == a))
    return partition
```

PA3: Generate relation from equivalence

> 给定等价类$[a]_R$，返回集合A上的关系R

```
def createEquivalenceRelation(partition, A):
    #对给定的集合A，以及A上的一个划分partition
    #生成由该划分决定的等价关系
    assert functools.reduce(lambda x, y: x.union(y), partition) == A
    return Relation(A, set([(a,b) for p in partition for a in p for b in p]))
```

PA4: Relation matrix operation operator

- `join`算子
- `meet`算子
- `logic_mul`算子

3.2 Relational Database Implemention

PA1 Definition

任务描述

本关任务：在看懂本实训定义的`relDB`类代码及其用法的基础上，完成函数`defineTables`的编写，该函数通过实例化`relDB`对象，构建并返回三个关系数据表。

1. 部门表：`relDB`类的实例，对象名为`dept`。该表有三个属性，分别为`DNO`、`DNAME`、`BUDGET`。表中数据有三组，对应的值分别为`("D1", "Marketing", "10M")`、`("D2", "Development", "12M")`、`("D3", "Research", "5M")`。
2. 雇员表1：`relDB`类的实例，对象名为`emp`。该表有四个属性，分别为`ENO`、`ENAME`、`DNO`与`SALARY`。表中数据有三组，对应的值分别为`("E1", "Lopez", "D1", "40K")`、`("E2", "Cheng", "D1", "42K")`、`("E3", "Finzi", "D2", "30K")`。
3. 雇员表2：`relDB`类的实例，对象名为`emp2`。该表有四个属性，分别为`ENO`、`ENAME`、`DNO`与`SALARY`。表中数据有2组，对应的值分别为`("E3", "Finzi", "D2", "30K")`、`("E4", "Saito", "D2", "35K")`。

PA2 Projection

任务描述

本关任务：编程实现关系数据表上的 `projection` 运算，并用第一关创建的三个数据表进行实验。

对有n个属性的关系数据表，`projection` 运算定义如下：
$A = A_1 \times A_2 \times \cdots \times A_n$ ，令 $i_k = (i_1, i_2, \cdots, i_m)$ ，且对所有的 $1 \le k \le m$ 有 $1 \le i_k \le n$ ，则n元序偶上的 `project` 运算定义为 $P_{\{i_k\}} : A_1 \times A_2 \times \cdots \times A_n \to A_{i_1} \times A_{i_2} \times \cdots \times A_{i_m}$ ，且 $P_{\{i_k\}}(a_1, a_2, \cdots, a_n) = (a_{i_1}, a_{i_2}, \cdots, a_{i_m})$

例如，对具有4个属性的数据表 `A={Place, Seats, BoardType, Computer}`，假设表中有4组数据 `{ (Fan171, 80, Chalk, Yes), (Lws210, 25, No,  Yes), (Nek138, 50, Chalk, No), (Agr212, 200, No, Yes), …}`，则 `project(R, {Place, Seats})` 将返回一个只包含 `Place` 和 `Seats` 2个属性的数据表，且其数据为 `{(Fan171, 80), (Lws210, 25), (Nek138, 50), (Agr212, 200), …}`。

任务：

1. 编程实现函数 `project(orig_dict, attributes)`，该函数接受2个参数，其中 `orig_dict` 是一个字典，用于表示数据表中的一组数据，例如 `{'Place':'Fan171', 'Seats':80, 'BoardType':'Chalk', 'Computer':'Yes'}`，`attributes` 是将被投影到的属性集合，例如上述示例中的 `{Place, Seats}`。该函数将返回一个字典，是 `orig_dict` 数据在 `attributes` 上的投影。
2. 编程实现函数 `PROJECT(orig_rel, attributes)`，该函数接受2个参数，`orig_rel` 为一个 `relDB` 对象，`attributes` 是将被投影到的属性集合。该函数返回一个 `relDB` 对象，为 `project` 运算后的结果。**请注意**该函数会调用 `project` 函数。

```
def project(orig_dict, attributes):
    return {attr: orig_dict[attr] for attr in attributes if attr in orig_dict}
def PROJECT(orig_rel, attributes):
    projected_data = [project(tup, attributes) for tup in orig_rel.tuples()]
    return RelDB(attributes, projected_data)
```

PA3 Selection

任务描述

本关任务：编程实现关系数据表上的 `select` 运算，并用第一关创建的三个数据表进行实验。

对有n个属性的关系数据表，`selection` 运算定义如下：
令 $A = A_1 \times A_2 \times \cdots \times A_n$ ，令 $C : A \to \{True, False\}$ 为集合A元素上的一个条件（谓词），则n元序偶上的 `select` 运算S_C定义为将A上关系R中所有满足条件C的n元序偶挑选出来，即 $\forall R \subseteq A, S_C(R) = \{a \in R | S_C(a) = True\}$。

例如，对具有4个属性的数据表 `A={Place, Seats, BoardType, Computer}`，假设表中有4组数据 `R={ (Fan171, 80, Chalk, Yes), (Lws210, 25, No,  Yes), (Nek138, 50, Chalk, No), (Agr212, 200, No, Yes), …}`，$C_1 : BoardType = "Chalk"$，则 `select(R,` C_1`)` 将返回 `{(Fan171,80,Chalk,Yes),(Nek138,50,Chalk,No),…}`。

任务：

1. 编程实现函数 `SELECT(orig_rel, restriction)`，该函数接受2个参数，`orig_rel` 为一个 `relDB` 对象，`restriction` 是一个函数，用于判断 `orig_rel` 中数据是否满足挑选条件，如 `lambda tup: tup["SALARY"] <= "40K"`。`SELECT` 函数将返回一个 `relDB` 对象，为 `SELECT` 运算后的结果。

PA4 Join

任务描述

本关任务：编程实现关系数据表上的 `join` 运算，并用第一关创建的三个数据表进行实验。

对有n个属性的关系数据表，`join` 运算用于将2个关系表组合出某种合成关系表，定义如下：
假如序偶$(A, B) \in R_1$，序偶$(B, C) \in R_2$，则序偶$(A, B, C) \in J(R_1, R_2)$，其中$J(R_1, R_2)$即为两个关系表 `join` 后的新关系表。请注意$A$、$B$和$C$并不只是代表一个属性上的数据，而是代表一系列属性上的数据。

`join` 运算示例如下图所示：

Prof.	Course
John	CS202
Brian	CS340
Kenny	CS220
Kemal	CS215

R1

Course	Class	Time
CS215	FNR1326	11:00
CS220	FNR2332	9:00
CS340	Quig101	2:30
CS202	LWS202	3:30

R2

Prof.	Course	Class	Time
Kemal	CS215	FNR1326	11:00
Kenny	CS220	FNR2332	9:00
Brian	CS340	Quig101	2:30
John	CS202	LWS202	3:30

J(R1, R2)

任务：

1. 编程实现函数 `JOIN(rel_1, rel_2)`，该函数接受2个参数，`rel_1` 和 `rel_2` 都为 `relDB` 对象。`JOIN` 函数将返回一个 `relDB` 对象，为 `JOIN` 运算后的结果。

```
def JOIN(rel_1, rel_2):
    assert not(set(rel_1.attributes()) & set(rel_2.attributes()) == set())
    common_attrs = set(rel_1.attributes()) & set(rel_2.attributes())
    new_attrs = list(rel_1.attributes()) + [attr for attr in rel_2.attributes()
if attr not in common_attrs]
    joined_tuples = set()
    for t1 in rel_1.tuples():
        for t2 in rel_2.tuples():
            if all(t1[attr] == t2[attr] for attr in common_attrs):
                new_tuple = tuple(t1[attr] for attr in rel_1.attributes()) +
tuple(t2[attr] for attr in rel_2.attributes() if attr not in common_attrs)
                joined_tuples.add(new_tuple)

    return RelDB(new_attrs, joined_tuples)
```

4 LogicLab

4.1 Formulation

按照BNF范式定义命题逻辑合式公式语法元素对应语义对象的建模

```
FOL BNF
    FORMULA     ::= PROPOSITION
                  |  '(' FORMULA CONNECTIVE FORMULA ')'
                  |  'not' FORMULA
                  |  '(' FORMULA ')'
                  |  'T'
                  |  'F'
    CONNECTIVE  ::= 'implies' | 'equiv' | 'and' | 'or'
    PROPOSITION ::= [A-Z]-[T, F]\w
```

为生成给定命题逻辑公式的真值表，需在基于上述语法的抽象语法树上应用命题逻辑联结词的语义规则，因此，首先需为上述语法元素建立对应的Python类。

共有7个语法元素需建立Python类：命题常量、命题词、非、合取、析取、蕴含、等值。每个类的属性及方法描述如下：

- Proposition类：对应命题词，有两个属性 `name` 和 `value`，初始时 `name` 为命题词的名字，`value` 为 `None`。
- BoolConstant类：对应命题常量，有两个属性 `name` 和 `value`，初始时 `name` 为命题常量对应的字母，为 `T` 或 `F`，`value` 为根据命题常量的 `name` 分别为 `True` 或 `False`。
- Not类：对应 `not` 逻辑联结词，只有一个属性，为 `formula`，对应为非联结词后的命题逻辑公式。
- And、Or、Implies、Equiv类：分别对应 `and`、`or`、`implies` 和 `equiv` 4个逻辑联结词，有2个属性，分别为 `formula_a` 和 `formula_b`，分别对应联结词左边和右边的命题逻辑公式。

在建立了上述类后，可将命题逻辑合式公式转换为对应的语义模型，例如公式 `P/\Q`、$\neg R$对应的语义模型分别为：

```
1. print(And(Proposition('P'), Proposition('Q')))
2. print(Not(Proposition('R')))
```

输出为：

```
1. (P /\ Q)
2. ~R
```

4.2 命题逻辑编译器

在类命题和基本算子的基础上，利用python中的内置的语义分析包，实现命题逻辑编译器.