离散数学速通

written by SJTU-XHW

Reference: 清华大学出版社《数理逻辑与集合论（第二版）》《图论与代数结构》

未经授权，禁止传播

Chapter 1 命题逻辑

1.1 重要概念汇总

- 命题的定义：能判断真假的陈述句；（真假性：命题的值）
- 命题分类：原子命题、复合命题（成分命题 + 联结词）
- 真值联结词：非（否定式）、且（合取式）、或（析取式）、如果则（蕴含式）、等价（等值式）；
- 命题变元：以真假为变域的变元；
- 真值函数：以真假为定义域、真假为值域的函数；
- 命题演算公式的定义：（通俗）由命题变元利用真值联结词构成的式子；

> 命题变元是、公式的否定式是、两个公式的合取、析取式是；仅限于此；

- 变元组：n元公式 α 含有不同命题变元 p1、p2、……、pn，称（p1, p2, ..., pn）为 α 的变元组；
- 完全指派：变元组任一组确定值都称为 α 关于这个变元组的一个 ~；
- 部分指派：仅对变元组部分变元赋以确定值，另外一部分不赋值（X），称……；

> 完全、部分指派举例：(p, q, r, s) = (T, T, F, T)；(p, q, r, s) = (T, X, X, F)

- 成真指派、成假指派；

- [x] 一个公式的性质

- 重言式：该公式的所有完全指派都是成真指派；
- 矛盾式：该公式的所有完全指派都是成假指派；
- 可满足公式：该公式存在成真指派；

> 又可以分为：仅可满足公式、重言式；

- 非永真式：该公式存在成假指派；

> 又可分为：仅可满足公式、矛盾式；

- [x] 两个公式的关系

- 逻辑等值（**同真假**）：两个公式在它们的合成变元组的任何完全指派下，有完全相同的真假性；

  > 相对地，有**互相矛盾性**，~都取不同的真假值；

  > 推论：α = β ⟺ α ↔ β 为永真式；

  需要记忆的同真假式变换：

  - 交换律、结合律：**蕴含词不满足**；
  - 分配律：**等值词不满足、蕴含词分配后不变：**$p \rightarrow (q \rightarrow r) = (p \rightarrow q) \rightarrow (p \rightarrow r)$；
  - 摩根律、双重否定律、幂等律（与自身运算）、同一律和零律；
  - **联结词化归结论：**$p \rightarrow q = \overline{p} \vee q$、$p \leftrightarrow q = p \wedge q \vee \overline{p} \wedge \overline{q}$；

- 同永真性：若 α 永真当且仅当 β 永真，则称~；
- 同可满足性

> 推论：α、β 同真假 ⟹ α、β 既同永真性，又同可满足性；

- 对偶式：将任一**不含蕴含词、等价词**的 α 中所有 ∧、∨ 互换得到的公式称为 α 的对偶式，记为 α^*；

> 注：永真（T）、永假（F）看作 $p \vee \overline{p}$、$p \wedge \overline{p}$；

- 内否式：将任一 α 中各变元（**不能是子公式**）的所有肯定形式和否定形式互换，得到的公式称为 α 的内否式，记为 α^-；

$$
\begin{aligned}
\alpha &= (\overline{p \wedge q} \ \vee \ \overline{p \wedge r}) \ \wedge \ (p \vee (\overline{q} \wedge \overline{r})) \\
\alpha^* &= (\overline{p \vee q} \wedge \overline{p \vee r}) \ \vee \ (p \wedge (\overline{q} \vee \overline{r})) \\
\alpha^- &= (\overline{\overline{p} \wedge \overline{q}} \vee \overline{\overline{p} \wedge \overline{r}}) \ \wedge \ (\overline{p} \wedge (q \vee r))
\end{aligned}
$$

常用性质：

- $\alpha^- \wedge \beta^- = (\alpha \wedge \beta)^-$（对各种真值联结词适应，甚至蕴含词）
- $(P \vee Q \wedge R)^* = P \wedge (Q \vee R)$（变换后不改变原来运算次序）
- $\overline{\alpha^*} = \overline{\alpha}^*$、$\overline{\alpha^-} = \overline{\alpha}^-$（对偶、内否都可与否定交换运算顺序）
- $\overline{\alpha} = \alpha^{*-}$（内否、对偶同时取即为取反）

推论：

- α 与 α^- 既同永真性，又同可满足性（不一定同真假）；
- $\overline{\alpha}$ 与 α^* 既同永真性，又同可满足性；
- 【对偶定理】$\alpha \to \beta$ 与 $\beta^* \to \alpha^*$ 既同永真性，又同可满足性、$\alpha \leftrightarrow \beta$ 与 $\alpha^* \leftrightarrow \beta^*$ 既同永真性，又同可满足性；

- **置换规则和代入规则的异同**：
  - 置换规则：对公式的子公式，用与之等值的公式进行替换，得到的新公式与原公式同真假；

    > 可以只对某一个子公式替换；

  - 代入规则：对公式中出现的所有同一个命题变元代入一个公式，得到的新公式与原公式同真假；

    > 必须处处代入、必须对变元代入；

- 范式：一个公式到真假指派的**逆过程**——如何从真假指派构造出公式
- 简单合取式：命题变元，或（可兼得）其否定，或（可兼得）由它们利用联结词 $\wedge$ 组成的公式；

  简单析取式：……联结词 $\vee$ ……；

- 实合取式、虚合取式：某一变元的否定及其自身**最多仅出现一次**的简单合取式称为**实合取式**，否则为**虚合取式（即：永假合取式）**；

  实析取式、虚析取式：……简单析取式……（**永真析取式**）；

- （极大项极小项）唯一指派定理：对一个确定的变元组而言，任一实合取式有且仅有一个成真指派、任一个实析取式有且仅有一个成假指派（回顾数电中的极大项、极小项）；

> 变元组 (p, q, r) = (T, X, F) 是一个成真指派 对应：$p \wedge \overline{q}$；
>
> 变元组 (p, q, r) = (F, F, T) 是一个成假指派 对应：$p \vee q \vee \overline{r}$；

- 范式表示定理：**任一公式 α 恒可以表示为简单合取式的 析取，称为 α 的析取范式（即逻辑代数中的 SOP）**，也可以表示为**简单析取式的 合取，称为 α 的合取范式（POS）**；

> 合取范式和析取范式不唯一；

- 主析取范式、主合取范式：析合范式、合析范式中的简单合取式、简单析取式全为最小项、最大项；（标准SOP和标准POS）

  主析/合取范式定理：任一个 n 元公式，都存在一个唯一的与之等值的、恰含这 n 个变元的主析/合取范式；

  最小项的表示：m_i 指 使最小项为真的各变元组成的二进制数代表 i 的那个组合；

  最大项的表示：M_i 指 使最大项为真（这里和主流教材不一样）的各变元组成的二进制数代表 i 的那个组合；

  最小项的性质：

  - n 元公式的最小项共 2^n 种；
  - 在公式的变元组任意完全指派中，有且仅有一种最小项 为真；
  - 全体最小项析取为1，任意两个最小项合取为0；

  最大项的性质：

  - n 元公式的最大项共 2^n 种；
  - 在公式的变元组任意完全指派中，有且仅有一个最大项 为假；
  - 全体最大项合取为0，任意两个最大项析取为1；

- 范式表示定理的延伸：联结词的完备集；

1.2 命题逻辑的推理形式

- 判断A 和 B是否重言蕴含：**判断 $A \to B$ 是否永真，即在 A 所有的成真指派下，B是否都为真**；
- 重言蕴含的性质
  - 若 $A \Rightarrow B$，则：若 A 永真，则 B 永真；
  - 若 $A \Rightarrow B$、 $B \Rightarrow A$，则 A、B 等价；
  - 若 $A \Rightarrow B$ 且 $A \Rightarrow C$，则 $A \Rightarrow (B \wedge C)$（归并合取）；
  - 若 $A \Rightarrow C$ 且 $B \Rightarrow C$，则 $(A \vee B) \Rightarrow C$（归并析取）；
- 基本推理公式
  - $P \wedge Q \Rightarrow P$（从严格到一般）；
  - $P \Rightarrow (P \vee Q)$（从一般到模糊）；
  - $P \wedge (P \to Q) \Rightarrow Q$（分离规则）；

    $\overline{Q} \wedge (P \to Q) \Rightarrow \overline{P}$（分离规则的逆否命题）；
  - $(P \to Q)$ $\wedge$ $(Q \to R) \Rightarrow P \to R$（三段论）；

    $(P \leftrightarrow Q)$ $\wedge$ $(Q \leftrightarrow R) \Rightarrow P \leftrightarrow R$
- 如何证明推理公式
  - 可以使用**重言蕴含**的定义（$A \to B$ 为永真式）——永真推理 / 假设推理；
  - 使用**重言蕴含**定义的反面（$A \wedge \overline{B}$ 为永假式）——归结推理；
  - 证明推理公式本身的**逆否命题**；
  - 按三段论拆分；
  - 真值表法；
- 推理演算规则
  - 前提引入：推理过程中可以随时引入前提；
  - 结论引入：推论过程中获得的结论可以作为后续推理的前提；
  - 代入规则、置换规则

    > **代入：对命题变元、代入的必须是重言式、所有出现都要代入**；
    >
    > **置换：子公式，置换部分需要等值，可以仅一部分**；

  - 分离规则；
  - 条件证明规则：$A_1 \wedge A_2 \Rightarrow B \Longleftrightarrow A_1 \Rightarrow A_2 \to B$（将条件移动到另一边）；
- 归结推理法

  为证明 $A \to B$ 重言，从 $A \wedge \overline{B}$ 开始：

  step 1. 将 $A \wedge \overline{B}$ 化为**合取范式**，并由其中的子句（析取式+）构成子句集 S；

  step 2. 对 S 中的子句做归结：**消互补对（$C_1 = L \vee C_1'$ 和 $C_2 = \overline{L} \vee C_2'$ 归结为 $R(C_1, C_2) = C_1' \vee C_2'$）**；

  step 3. 重复step2，直至得到空子句（矛盾）；

1.3 考点和常见题型

- 命题的判断

  > 1. 必定为陈述句，否则没有真假；
  > 2. 必须可以辨别真假；
  > 3. 未证明的定理算，因为最终是有真假的；
  > 4. 悖论不算："这句话是假话"；

- 真值联结词和自然语言的差异
  - 并列"和"、并且"和"；
  - 可兼得"或"、不可兼得"或"；

- 不关心因果的“如果……则……”；
- 波兰表达式（前缀表达式）、逆波兰表达式（后缀表达式）、中缀表达式的转换

  前缀式书写思路：找到当前最后运算的运算符，先放到当前区块的最前面，以此为分割，分别处理左右边的式子；

- 给定一个公式，确定其成真指派、成假指派

  step 1. 应用等值运算将否定词深入到变元上；

  step 2. 随机指定一个变元（建议选出现次数多、对它指派能轻松得到结果的变元），对其分别做 T、 F 指派（分类讨论），得到两个不含该变元，但有真假值的公式；

  step 3. 化简这两个公式，如果仍存在变元，重复上面的操作，直至能够直接判断真假为止；

  > e.g., 判定 $\overline{(p \vee \overline{r}) \to ((p \to q) \leftrightarrow \overline{(p \wedge \overline{(q \leftrightarrow r)})})}$ 的永真性和可满足性；

- 判断联结词的完备集、联结词相互转化
  - 否定、析取、合取：$\neg$ 和 $\vee$ 、$\neg$ 和 $\wedge$ 构成**最小完备集**，但 $\wedge$ 和 $\vee$ 不行！！！
  - 与非（$\uparrow$）、或非（$\downarrow$）：各自独立构成**最小完备集**；
  - **否定和蕴含（$\to$）构成一个最小完备集**，但否定和等价**甚至连完备集都不是**！！！
  - 除了以上提到的最小完备集，及其扩充，其他任何联结词组成的集合都不是完备集；

    > e.g., $\{\wedge, \leftrightarrow\}, \{\vee, \to\}, \{\neg, \leftrightarrow\}$ 等，都不是完备集；

- n 元联结词的数量：2^{2^n} 个；
- 将一个公式转换为对应的主析取范式、主合取范式、最简析合范式、最简合析范式；

  > e.g.1, 将 主析取范式 $A = \bigvee_{0,1,4,5,7}$ 转换为主合取范式；
  >
  > e.g.2, 将 合取范式$P \vee Q$ 转换为析取范式；

- 证明推理公式：例题略
- 判断推理式是否正确：**从重言蕴含化为蕴含运算，看是否为永真式**；
- 自然语言到命题公式的形式化和证明

  ⚠️**极易出错点：只有……才、除非……否则、或者……或者的形式化**

  - “只有P-才Q”：只有满足P，才满足Q，**言外之意，P 是 Q 的必要不充分条件！（如果不满足P，那么并不影响 Q）**，所以翻译为：$Q \to P$；
  - “除非P-否则Q”：等价于只有满足 P，才满足 $\neg Q$，所以翻译为 $\neg Q \to P$；
  - “或者P-或者Q”：**有两种理解方式：可兼得或、不可兼得或**，视语境而定；可兼得或，和“……可以，……也可以”的意义相同，这个时候翻译成 $P \vee Q$；不可兼得或，和“要么……要么……”的意义相同，这个时候翻译成 $P \oplus Q = P \wedge \neg Q \vee \neg P \wedge Q$；

1.4 错题

- 形式化语句：如果水是清的，那么或者张三能见到池底或者他是个近视眼；

  错因：读题问题

  “或者……或者……”在这里是不可兼得或，表示“要么……要么……”

  > 设 P: 水清；Q: 张三能见池底；R: 张三是近视眼；
  >
  > $P \to (Q \wedge \overline{R} \vee \overline{Q} \wedge R)$

- 将公式写为波兰表达式、逆波兰表达式：$P \to Q \vee R \vee S$、$\neg\neg P \vee (W \wedge R) \vee \neg Q$、$P \wedge \neg R \leftrightarrow P \vee R$；

  错因：运算符优先级判断错误（$\neg > \wedge > \vee > \to > \leftrightarrow$）

  $\to P \vee \vee QRS$、$PQR \vee S \vee$、$\leftrightarrow \wedge P \neg R \vee PQ$（其他略）

Chapter 2 谓词逻辑（一阶）

> 将原子命题进一步分析为**个体**和**谓词**；

2.1 重要概念集合

- 个体：可以指具体/抽象的独立存在的对象；

  > 如：字母R，XX公司，1，2，-1，张三等；

- 个体域：由个体组成的集合；

  全总个体域：所有个体聚合在一起所组成的集合；

- 个体变元：以某个个体域 I 中的个体为变域的变元，称为**个体域 I 上的个体变元**；
- 谓词：个体、命题所具有的性质，或者若干个体、若干命题之间的关系；
  - 一元谓词：指明个体/命题的性质；
  - 二元谓词：指明两个个体/一个个体和一个命题间的关系；

  谓词实际上是一个以个体/命题（目前仅讨论以个体为变域的一阶谓词）为变域、命题为值的函数；

  > 真值函数是个以命题为变目、命题为值的函数，所以也是谓词；（真值联结词同理）

  ⚠**注意：谓词与个体域紧密相关，例如一元谓词"是质数"相对于自然数域而言**；

- 谓词变元：以谓词为变域的变元；

  > 约定：小写字母为（pqr）命题变元、大写字母（PQR）为命题、小写字母（abc）为个体变元、大写字母（ABC）为特定谓词、大写字母（XYZ）为谓词变元；

- 谓词填式：将谓词后面填以**个体变元**所得的式子；

  > 谓词和谓词填式是两个完全不同的概念；

- 命名变元：为了了解谓词的元数，一般在谓词后写**命名变元**来说明元数（类似函数形参）；
- 谓词命名式：谓词后填以**命名变元**的式子（类似函数声明）；

  > **谓词命名式 等价于 谓词**，所以谓词命名式和谓词填式完全不同；

- 谓词的严格定义：以某个个体域 I 为定义域，以真假为值域的谓词，称为**个体域 I 上的谓词**；

  谓词变元的严格定义：以个体域 I 上的谓词为变域的变元，称为**个体域 I 上的谓词变元**；

  结论：k 个个体组成的个体域 I 上的 m 元谓词共有 2^{k^m} 个；

- 量词：仅有个体、谓词，还是无法描述一些问题--->**措施：使用新的变元 + 约定新的变元的个体域 + 规定新的变元的性质 来处理**

  新的变元：
  - 全称性变元：表示**任意一个**的变元，都称~；
  - 存在性变元：表示**确定的一个，但现在可能不知道/无需指明的一个**，都称~；

  引入谓词约束新的个体域：
  - 谓词配合全称性量词+蕴含前件约束：

    > 所有实数，要么大于0，要么小于0，要么等于0
    >
    > $Ay \to (y > 0 \vee y = 0 \vee y < 0)$，y是**以全总个体域为变域的全称性变元**，A表"是实数"；

  - 谓词配合存在性量词+合取约束：

    > 一个中国人来了（Ae表示e来了）
    >
    > $Bu \wedge Au$，u是**以全总个体域为变域的存在性变元**，B表"是中国人"；

引入**量词**规定新的变元的性质：$\forall x$（全称量词）、$\exists x$（存在量词）；

- 细小概念：对两种公式：$\forall x\alpha(x)$，$\exists x\alpha(x)$
  - $\forall x$、$\exists x$ 中的 x 是相应量词的**指导变元**；
  - $\alpha(x)$ 为相应量词的**作用域（或辖域）**；
  - 在作用域中，但不与指导变元同名的其他变元称为**参数**；
- **谓词演算公式（定义和主流教材不一样！有争议，应该不考定义，这里写主流教材的）**：由命题变元、谓词填式利用真值联结词和量词作成的如下式子：
  - 命题公式是公式；
  - 填以个体变元的谓词填式也是公式；
  - 公式的否定是公式；
  - 两个公式的合取、析取、蕴含、等价也是公式；
  - 若 α 是公式，x 是个体变元，则 $\forall x\alpha$，$\exists x\alpha$ 也是公式；

> 提示，本教程中不主流，认为$\forall xF(x) \wedge G(x)$ 不是公式；

- 自由出现和约束出现：设 α 为一个谓词演算公式，Qxβ是 α 的一个子公式（Q为量词的一种，x 为变元，β 为公式），则在Qxβ 中：变元 x 的一切出现都称为 x 在 α 中的**约束出现**；而 α 中的 x 除了约束出现以外的一切出现都称 x 在 α 中的**自由出现**；

> $\forall x((Xxy \wedge \exists yYy) \rightarrow \exists y(\overline{X}xy \leftrightarrow \exists y\overline{X}y))$中，
>
> 第一个 y 是公式中 y 的自由出现，第2、3、4、5、6、7都是公式中 y 的约束出现；

- 约束关系：各个量词和变元的出现间的约束上的关系；

  确定约束关系的过程：确定"公式中哪个变元的哪些出现受哪个量词的约束"的过程；
- 自由变元、约束变元；

  同一变元在一个公式中可能既是自由变元，又是约束变元；

  在一阶谓词逻辑中，认为命题变元、谓词变元都是自由变元，不受量词约束；
- 改名：将一个变元改为另一个变元，并要求改名后的式子与原来的式子语义相同
  - 改名**针对约束变元**而言，不能对自由变元改名；
  - 改名必须同时对原式中该变元的一切受**某个量词约束**的约束出现**均**改名（**同约束全替换**）；
  - 改名后的名字不允许和作用域中其他自由变元同名；（会改变约束关系）
  - 改名后，**一般**不与作用域中约束变元同名，因为容易混；
- 代入：将一个变元代以式子（式子的变域与变元变域要相同），要求代入后的式子是原式的特例；（不要求等值）
  - 代入**针对自由变元**而言，约束变元不允许代入；
  - 代入式当中的变量名不能与原式作用域中的约束变量同名；
  - 同一自由变元出现**全部替换**；

> 可以的：命题变元代入公式、个体变元代入项、谓词变元代入同元谓词；

总结：1、只要没改变约束关系的、不引发歧义的改名、代入都行；

2、要改全改，要代全代；

> $\forall y(p \rightarrow Ay)$ 可以将 p（命题变元）代入为 $\exists yBxy$，因为没有歧义、没有改变约束关系；

- [x] 一个公式的关系

- 谓词演算公式的永真性、可满足性

  【POINT 1】**谓词演算公式的真假与：个体域、自由变元（自由个体变元、谓词变元、命题变元）有关**，和约束变元无关；
  - 谓词演算公式的成真指派、成假指派、有缺指派

    一个一阶谓词演算公式 α，其自由个体变元 $x_1, x_2, ..., x_h$，命题变元 $p_1, p_2, ..., p_k$，谓词变元 $X_1, X_2, ..., X_r$，则将 α 表示为：$\alpha(x_1, x_2, ..., x_h;\ p_1, p_2, ..., p_k;\ X_1, X_2, ..., X_r)$；

    若对其个体域 I 的指派为 I_0（约束变元的全总个体域也变为 I_0），

对 $x_1,x_2,\ldots,x_h$ 指派为：$\alpha_1,\alpha_2,\ldots,\alpha_h$、对 $p_1,p_2,\ldots,p_k$ 指派为 $p_1^0,p_2^0,\ldots,p_k^0$、对 $X_1,X_2,\ldots,X_r$ 指派为 $A_1,A_2,\ldots,A_r$，则称对 α 作一个个体域 I_0 上的指派 $(\alpha_1,\alpha_2,\ldots,\alpha_h;\ p_1^0,p_2^0,\ldots,p_k^0;\ A_1,A_2,\ldots,A_r)$；

在一个确定的个体域、自由个体变元、命题变元、谓词变元的指派下，公式 α 的真假值确定；

若该指派下公式的值为真，则称为成真指派，否则为成假指派；

若仅对 α 中的部分自由变元指派，则称该指派为 α 的一个有缺指派（有缺指派下公式不一定有确定的真假值）；

【POINT 2】谓词演算公式的真假值确定的**关键**在于**确定 $\forall x\alpha(x)$ 和 $\exists x\alpha(x)$ 的真假**；

> $\forall x\alpha(x)$ 为真，当且仅当 I 中每个个体都使公式为真；
>
> $\exists x\alpha(x)$ 为真，当且仅当 I 中至少有一个个体使公式为真；

- **永真性（普遍有效性）、可满足性**：若公式 α 对于个体域 I 中任何指派均取真值，则称 α 在 I 中永真；若 α 在每一个非空个体域中均永真，则称 α **永真（普遍有效公式）**；

  > 1. 可满足性的定义同理；
  > 2. 此处许多定理，例如 α、β 同真假 $\Longleftrightarrow$ α $\leftrightarrow$ β 永真，和命题逻辑类似，略去；

  重要定理：有限域上，公式的永真性、可满足性仅取决于个体域中的个体数；

  > 感性理解：如果公式的永真性取决于个体域中的不同个体的**特征**的话，那么就不是永真了；

  因此，将 $\{1,2,3,\ldots,k\}$ 作为具有 k 个个体的个体域的代表（仅与数量有关，所以元素是啥都行），命名为 “k-域”，α 在 k 域上永真称为**k-永真**；

- 同永真性、同可满足性：α 永真当且仅当 β 永真，则称 α、β **同永真性**；

* 有限域下的公式表示法：有限域 $D=\{1,2,3,\ldots,k\}$ 不失一般性；

$$\forall xP(x)=P(1)\wedge P(2)\wedge\cdots\wedge P(k)$$
$$\exists xP(x)=P(1)\vee P(2)\vee\cdots\vee P(k)$$

> **当发现某些谓词演算公式难以理解时，尝试在 {1, 2} 域下展开**；

* 全称封闭式、存在封闭式：将公式中的一切自由个体变元，构造出对应的全称量词并置于全式之前，则称得到的新公式为 α 的全称封闭式，记作$\Delta\alpha$；（存在封闭式同理）

  重要定理：α 与它的全称封闭式在每个域中同永真性、与它的存在封闭式在每个域中同可满足性；

  > 感性理解：一个公式一旦确定为永真，那么他的所有自由个体变元的所有取值都不影响它的真，所以它的全称封闭式也为真；

  引入存在封闭式，可以将公理50 $\Delta(\forall x\alpha(x)\to\alpha(\xi))$、公理51 $\Delta(\alpha(\xi)\to\exists x\alpha(x))$ 没有歧义地表述，不容易造成误解和错用；

- [x] 两个公式的关系

* 谓词演算公式的同真假性、在 I 上同真假性（或者说**等值**）：设有公式 α、β，若对个体域 I 的每一指派，α、β 均取相同真假值，则称 α、β **在 I 上同真假**；若 α、β 在每个非空域上同真假，则称二者**同真假**；

  谓词演算公式同真假公式记忆：

  - $\overline{\forall}x\alpha(x)=\exists x\overline{\alpha}(x)$、$\overline{\exists}x\alpha(x)=\forall x\overline{\alpha}(x)$；（**量词否定的同真假变换**）
  - $\overline{\forall}x\exists y\alpha(x)=\exists x\forall y\overline{\alpha}(x)$（**对多个量词的否定同真假变换，多少个量词都一样**）
  - $\forall x(\alpha(x)\vee\gamma)=\forall x\alpha(x)\vee\gamma$、$\forall x(\alpha(x)\wedge\gamma)=\forall x\alpha(x)\wedge\gamma$

    $\exists x(\alpha(x)\vee\gamma)=\exists x\alpha(x)\vee\gamma$、$\exists x(\alpha(x)\wedge\gamma)=\exists x\alpha(x)\wedge\gamma$；（**量词与”自由公式“的结合律**）

    （γ 为**不含自由变元 x** 的公式）

    请注意：如果涉及蕴含符号，请自行推导，结论是不同的！

  第二组重要的等值公式：相关约束分配律

  - $\forall x(P(x)\wedge Q(x))=\forall xP(x)\wedge\forall xQ(x)$

$\exists x(P(x) \vee Q(x)) = \exists xP(x) \vee \exists xQ(x)$

分开个体不必相同，原因在于：理解为有限域下公式表示，是否展开为连续的 $\wedge$ 和 $\vee$；

- $(\forall x)P(x) \vee (\forall x)Q(x) = \forall x\forall y(P(x) \vee Q(y))$

  $(\exists x)P(x) \wedge (\exists x)Q(x) = \exists x\exists y(P(x) \wedge Q(y))$

  分开不一定相同，一般情况是 $(\forall x)P(x) \vee (\forall x)Q(x)$、$\exists x(P(x) \wedge Q(x))$ **更强一些：**

  $\forall xP(x) \vee \forall xQ(x) \Rightarrow \forall x(P(x) \vee Q(x))$

  $\exists x(P(x) \wedge Q(x)) \Rightarrow \exists xP(x) \wedge \exists xQ(x)$

  记忆方法可以采用**"唱歌跳舞解释法"**：例如，班里所有人都会唱歌 或 班里所有人都会跳舞 **强于** 班里所有人都会唱歌或跳舞（因为可能恰好一部分人只会唱歌、另一部分人只会跳舞，这样满足后者，却不满足前者）

- 前束型公式、准前束型公式：一个公式的量词都在开头，它们的作用域延伸到全式的末尾，称这样的公式为**前束型公式**；由前束型公式利用真值联结词作成的公式称为**准前束型公式**；

  > $\forall x\exists y\forall z(Xxy \to Yyz)$ 为前束型公式，$\forall x\exists yXxy \wedge \forall y\exists zYyz$ 为准前束型公式；

- **前束范式定理：任何一个谓词演算公式都与一个前束型公式同真假（称此前束型公式为前束范式）**

  > **前束范式不唯一！**

  ⚠**完全不建议、甚至禁止将蕴含词、等价词留在式中，因为考虑否定深入，需要将这些属性的词展开，有时甚至不展开是错误的！因为量词在进出这些运算符时可能多了一层否定**

  将公式转化为前束范式的思路：

  step 1. 否定深入，将量词上的否定转移到作用域内部（利用前面两条同真假公式）；

  step 2. 适当改名：规避将量词提出时的重名错误；（考虑约束关系会不会在提出后有歧义或改变）

  step 3. 根据同真假公式，依此提出内层的量词开头的公式；

- Skolem 标准型：一阶谓词逻辑的任一公式 α，若：

  1. 其前束范式中所有的存在量词都在全称量词的左边（∃ **前束范式**），
  2. 或者，仅保留全称量词而消去存在量词（∀ **前束范式**）

  则得到的 α 的新公式称为 **skolem 标准型**；

  ⚠**skolem 标准型一定无法像前束范式一样，与原式保持完全等值的关系，只能保持一定意义上的"关系"**

  - ∃ 前束范式：$\exists x_1\exists x_2\cdots\exists x_i\forall x_{i+1}\cdots\forall x_n M(x_1, x_2, \ldots, x_n)$，保证 i ≥ 1且 M 内**不含量词和自由个体变元**；

    ∃ 前束范式定理：FOL（一阶谓词逻辑）的任一公式 α 都能写成与之对应的（不一定等值）∃ 前束范式，且 <u>α 和 其 ∃ 前束范式同永真性</u>

    > 所以 一般当 α 永真时，或者想要进行**同永真转换**时，才使用 ∃ 前束范式；

  - ∀ 前束范式：仅保留全称量词的前束范式

    ∀ 前束范式定理：FOL 的任一公式 α 都能写成与之对应的（不一定等值）∀ 前束范式，且 <u>α 和 其 ∀ 前束范式同可满足性</u>

    > 所以 一般当 α 永假时，或者想要进行**同不可满足转换（通常是谓词演算公式的归结推理）**时，才使用 ∀ 前束范式；

2.2 谓词演算公式的推理演算

- 较之命题演算新增 4 个规则：
  1. 全称量词消去（公理50）$\frac{\forall xP(x)}{\therefore P(y)}$ 和 $\frac{\forall xP(x)}{\therefore P(c)}$
     - 要求取代的自由变元不能在原式中约束出现（否则改变约束关系，这个一般不会违反）
     - 必须取代所有的 该变元（消去全称量词的指导变元对应的变元，一般也不会违反）

     > 提示：全称量词的消去有两种使用方法，一种是左边的消去为全域的自由变量 y，另一种是消去为个体常量 c；

  2. 全称量词引入（全“0”规则）$\frac{P(y)}{\therefore \forall xP(x)}$
     - **要求左式 $P(y)$ 中的 y 是使公式为真的<u>自由个体变元</u>**

       > 具体是要求查找前面所有**假设**中，y 是否自由出现，如果是，一定**不满足**这个条件；

     - 取代 y 的 x 约束变元不能在 P(y) 中约束出现（否则改变约束关系，这个一般不违反）
  3. 存在量词消去（**存在假设**）$\frac{\exists xP(x)}{\therefore P(e)}$
     - 要求右式的 e 必须是使公式为真的**个体常项**，不在 P 中出现；
     - 要求 P **不能有自由个体变元**，会导致个体域混淆；
  4. 存在量词引入（公理51）$\frac{P(c)}{\therefore \exists xP(x)}$
     - 要求个体常项 c 不在 P 中其他地方出现；
- 归结推理：在命题逻辑处理的基础上，将“同不可满足公式”转换为 ∀ 前束范式，略去全称量词、∧ 以逗号相连，构成子句集，归结方法：$R(x)\vee Q(x)$ **和** $\overline{R}(a)\vee P(y)$ **归结为** $Q(a)\vee P(y)$（不是x 的原因是：在子句集中，看起来是自由变元的都是全称变元），直至空子句；

  为了方便起见，一般不完全写出 ∀ 前束范式，而是对由合取联结的几个部分分别求子句集，最后求并集；虽然与原子句集不一样，但它们的可满足性是一致的；
- 推理小结论
  - 相关约束分配律的大小关系
  - $\forall x\forall yP(x,y)$ 强于 $\exists x\forall yP(x,y)$ 强于 $\forall x\exists yP(x,y)$ 强于 $\exists x\exists yP(x,y)$
  - 一般情况下，量词**一定不具备对于等价词的分配律**：因为将等价词拆为双向蕴含词的合取，如果要分配，就意味着同时对 析取、合取 两种运算分配，这无论是全称量词还是存在量词都是不可接受的；

    > 例如：$\exists x(P(x)\leftrightarrow Q(x))\rightarrow(\exists xP(x)\leftrightarrow\exists xQ(x))$ 不是普遍有效的；
    >
    > 可以将 $P(x)\leftrightarrow Q(x)$ 拆开成 $(P(x)\rightarrow Q(x))\wedge(Q(x)\rightarrow P(x))$ 理解；

2.3 常见题型

- 自然语言的形式化

  注意：如果语句中含有“不”或者其他否定的字样，建议将其提出谓词的定义中，例如：

  > “Qe：e 溶于水” 的定义好于：“Qe：e 不溶于水”

- 谓词演算公式的改名和代入
- 给定一个指派，求谓词演算公式的真假值

  step 1. 初步代入指派（尽可能代入谓词变元、命题变元、自由个体变元）；

  step 2. **由内向外**逐层求**由量词带头的**子公式的值；**如果不能求出，则转化为同一语境下同真假的式子**

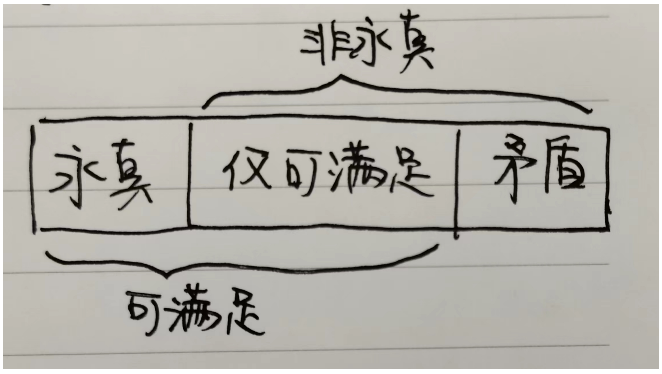
  > 在子公式内部，应该将与指导变元不同的变元暂时视作自由变元；

  step 3. 遇到多个量词和指导变元针对一个作用域时，按照**从左至右**的顺序理解，不能交换顺序；

- 判断公式的永真性（普遍有效性）、可满足性

> 但凡公式需要确认自由变元 才能判断真假的，都是仅可满足的；

只要将公式写出，那么它必然处于：永真性、仅可满足、不可满足（矛盾式）这三者特性之一！



- 想要证伪，则一般在 {1, 2} 域上列表格，“唱歌跳舞分析法”来查看，判断什么条件下能够生成反例，尤其对于蕴含式，设立一个前件正确、后件错误的才行；
- 想要证明，建议先看看证伪，发现无法证伪再来证明

⚠易错点1：有些公式默认了自由变元的全总个体域：$\forall x P(x) \to P(y)$ 是普遍有效公式；

⚠易错点2：有些公式没有直接方法化简到 F/T，而且你记得它不是普遍有效公式，也不能直接判断，因为它可能在某些情况下成立，例如：$(\exists x P(x) \wedge \exists x Q(x)) \to \exists x(P(x) \wedge Q(x))$ 是**1-永真**的，所以是可满足公式；

- 拓展：谓词演算公式的可判定性（**针对一个演算系统而言，并非是对一个公式而言**）

- 求公式对应的前束范式
- 求公式对应的 $\forall$ 前束范式 / $\exists$ 前束范式

> 考试时注意：**本教材（非主流）中除非强调，默认“skolem标准型”就是 $\forall$ 前束范式，不指 $\exists$ 前束范式**；

转换为 $\exists$ 前束范式的基本思路：

1. 将公式转换为前束范式；
2. 以 $\exists x_1 \exists x_2 \cdots \exists x_i \forall x_{i+1} \exists x_{i+2} \cdots \exists x_n M(x_1, x_2, \ldots, x_n)$ 为例（因为都可以转化为：存在量词中间夹一个全称量词的情况）：

   引入 $\forall x_{i+1}$ 前的所有指导变元（**默认从左到右消除全称量词**），包括自己，作为新的谓词（没有出现过）的填式：$S(x_1, x_2, \ldots, x_i, x_{i+1})$，其中 S 是自由变元，可以取个体域上所有谓词；这样：

$$
\begin{aligned}
&\exists x_1 \exists x_2 \cdots \exists x_i \forall x_{i+1} \exists x_{i+2} \cdots \exists x_n M(x_1, x_2, \ldots, x_n) \\
=&\exists x_1 \exists x_2 \cdots \exists x_i \exists x_{i+1} \exists x_{i+2} \cdots \exists x_n (M(x_1, x_2, \ldots, x_n) \wedge \overline{S}(x_1, x_2, \ldots, x_{i+1}) \vee \forall y S(x_1, x_2, \ldots, x_i, y)) \\
=&\exists x_1 \exists x_2 \cdots \exists x_i \exists x_{i+1} \exists x_{i+2} \cdots \exists x_n \forall y (M(x_1, \ldots, x_n) \wedge \overline{S}(x_1, \ldots, x_{i+1}) \vee S(x_1, \ldots, x_i, y))
\end{aligned}
$$

3. 如上面公式的最后一步，将任意的 y 利用结合律提出，循环此过程直至变为 $\exists$ 前束范式；

转换为 $\forall$ 前束范式的基本思路：

1. 将公式转换为前束范式；
2. 以 $\forall x_1 \forall x_2 \cdots \forall x_i \exists x_{i+1} \forall x_{i+2} \cdots \forall x_n M(x_1, x_2, \ldots, x_n)$ 为例，因为都可以转化为：全称量词中间夹一个存在量词的情况）：
   - 如果 x_{i+1} 左边没有（全称）量词（**也默认从左到右消除**），即 i = 0，那么只要在作用域中将所有 x_{i+1} （一定是自由出现）代以一个从未出现的自由变元名，例如 y：$M(y, x_2, \ldots, x_n)$，直接删除 $\exists x_{i+1}$ 即可；
   - 如果 x_{i+1} 左边有（全称）量词，则将 x_{i+1} **替换为**左边出现的所有全称量词指导变元构成的自由变元-函词填式（从未出现过该名字，例如 f）：$f(x_1, x_2, \ldots, x_i)$，构成：$M(x_1, x_2, \ldots, x_i, f(x_1, x_2, \ldots, x_i), x_{i+2}, \ldots, x_n)$，直接删除 $\exists x_{i+1}$ 即可；
3. 循环上面的过程直至变为 $\forall$ 前束范式；

- 分析谓词演算公式证明的正确性
- 谓词演算公式的推理证明

> 推理中如果使用到了推理结论，例如：$P(x) \land Q(x) \Rightarrow P(x)$，在原因一栏写“重言蕴含”，这对于**命题逻辑**的推理也是这样的！

Chapter 3 集合论

3.1 重要概念集合

- 集合定义（确定性、互异性、无序性）、属于不属于定义；
- ⚠集合元素不能是其自身（罗素悖论）=> 抽象公理错误，不能任给一个性质就能构造一个对应的集合；
- 集合表示法：字母约定、外延表示法（穷举）、内涵表示法（谓词描述性质）；
- 集合间的关系
  - **外延公理：一个集合由它的元素完全确定**；
  - 相等关系：由外延公理可以写出定义，下略；
  - 子集（包含关系）、否定包含关系、真子集（真包含关系）；
    - **相等包含定理（证明两集合相等时常用）**：$A = B \Longleftrightarrow (A \subseteq B) \land (B \subseteq A)$；
    - 包含关系的三条性质：自反、反对称、传递；（属于关系没有传递的性质）
  - 相交关系：两集合是否有公共元素；没有称**不相交的**；
- 特殊集合
  - 空集：不含任何元素的集合，记 ϕ；
    - 性质1：包含于**任意集合**；
    - 性质2：**空集的唯一性**（外延定理说明）；
  - 全集：在**给定的问题**中，所考虑所有事物的集合；
- 集合运算
  - 并集、交集、差集（“差集”相对于“被差集”的相对补集）、余集（高中的“补集”）、对称差集（异或）；
  - 广义交集、广义并集（**针对集合的元素全是集合的集合**）；

    > 规定 $\bigcup \phi = \phi$，$\bigcap \phi$ 没有意义；

  - 幂集：集合的幂集是该集合所有子集组成的集合，记 $P(A)$
    - 性质1：幂集的元素全部是集合，无论 A 有没有元素；
    - 性质2：$\phi \in P(A)$、$A \in P(A)$；
  - 笛卡尔积

    > 定义“有序对”：两个元素 x、y（允许 x = y）按给定次序排列组成的二元组合称为一个~；
    >
    > **有序对的集合定义**：$\langle x, y \rangle = \{\{x\}, \{x, y\}\}$；
    >
    > 有序对性质定理：
    >
    > $x \neq y \ \Rightarrow \ \langle x, y \rangle \neq \langle y, x \rangle$
    >
    > $\langle x, y \rangle = \langle u, v \rangle \ \Rightarrow \ x = u \land y = u$

    定义：集合 A 与 B 的笛卡尔积 $A \times B = \{z \mid x \in A \land y \in B \land z = \langle x, y \rangle\}$；

    > A = B 时，可以写作：A^2；

    以上可以推广到 n 维空间；

    注意：无论多少阶，都不存在“一个集合中选多个元素作为 n 元组”的情况，参考坐标点；
- 运算符优先级：大致是 **集合运算符 > 真值联结词 > 逻辑关系符**；
- 集合的图形表示法：韦恩图（交并补）、哈斯图（幂集）、笛卡尔坐标系图示法；
- **集合重要运算性质（记忆）**

- 运算律：基本运算由联结词定义，所以性质相似，此处仅补充少见的运算律

$$A-(B\cup C)=(A-B)\cap(A-C)$$
$$A-(B\cap C)=(A-B)\cup(A-C)$$

- 运算性质：以差集为例

$$A-B=A-(A\cap B)$$
$$A\cap(B-C)=(A\cap B)-C$$
$$A-B=A\cap -B$$

最后一条极其重要，常用于差集运算符消去；

- 基本关系运算性质

$$A\cup B=B\iff A\subseteq B\iff A\cap B=A\iff A-B=\phi$$

最后一步启示我们：在证明式中含有 ϕ 时，建议将式中的差运算符留到最后；

例如证明：$(A-B)\oplus(A-C)=\phi\iff A-B=A-C$ 的时候；

- 幂集性质
  - $A\subseteq B\iff P(A)\subseteq P(B)$;
  - $A=B\iff P(A)=P(B)$;
  - $P(A)\in P(B)\implies A\in B$，逆命题不成立（A={Φ}，B={{Φ}}）；
  - $P(A)\cap P(B)=P(A\cap B)$;
  - $P(A)\cup P(B)\subseteq P(A\cup B)$;
  - $P(A-B)\subseteq(P(A)-P(B))\cup\{\phi\}$;
  - $\bigcup P(A)=A$；（广义并集和幂集互为逆运算）

> 传递集合的通俗定义：满足以下 2 点的集合是传递集合
>
> 1. A 的元素都是集合；
> 2. 如果有的话，A 的元素的元素都是 A 的元素；
>
> 外延定理的描述：$A\text{为传递集合}\iff\forall x\forall y((x\in y\wedge y\in A)\to x\in A)$;
>
> - **性质1：**$A\text{为传递集合}\iff A\subseteq P(A)$**;**
> - **性质2：**$A\text{为传递集合}\iff P(A)\text{也为传递集合}$**;**

- 笛卡尔积性质：不满足交换律、结合律，但满足对 $\cap$ 和 $\cup$ 的分配律；
  - $x\in A,\ y\in A\Rightarrow\langle x,y\rangle\in PP(A)$
  - $A\subseteq B\Leftrightarrow(A\times C\subseteq B\times C)\Leftrightarrow(C\times A\subseteq C\times B),\ where\ C\neq\phi$
  - $(A\times B\subseteq C\times D)\Leftrightarrow(A\subseteq C\wedge B\subseteq D),\ where\ A,B,C,D\neq\phi$

- 集合的基数（讨论有限集合）：规定 $|\phi|=0$

3.2 常见题型

- 写出给定集合的幂集
- 判断传递集合
- 给实际应用场景，求某个部分的基数

> 例如：参加 A 的有 x 人，参加 B 的有 y 人……，有 z 个人同时参加了……，诸如此类；

思路：Venn 图，或者概率统计的加法公式（把求概率改成求基数即可）：
$P(\bigcup\limits_{i=1}^{n}A_i)=\sum\limits_{i=1}^{n}P(A_i)-\sum\limits_{1\le i\le j\le n}P(A_iA_j)+\cdots+(-1)^{n-1}P(A_1A_2\cdots A_n)$;

- 以谓词演算公式和集合定理，证明集合公式

技巧总结

  - 过于基本的式子，比如用三段论一步解决的：$A\in B\wedge B\subseteq C\Rightarrow A\in C$;

> 建议直接推，如果需要改成谓词，就该，例如上面的题目的证明：
>
> $A \in B \wedge B \subseteq C \Leftrightarrow A \in B \wedge \forall x(x \in B \rightarrow x \in C) \Rightarrow A \in C;$

- 对于非定理式的、基本运算式的证明，一般采用：
  1. **等式证明：集合等值记忆式**；
  2. 等价证明：集合常用等价关系（**相等-包含关系、基本关系运算性质、幂集的性质**）
  3. 重言蕴含证明：集合常用的推导方式（例如：等值运算、**外延定理分解为谓词逻辑**）

  e.g., 证明以下运算式

  $$\begin{aligned}&1.\ A \cap (B \oplus C) = (A \cap B) \oplus (A \cap C)\\&2.\ (A-B) \oplus (A-C) = \phi \Longleftrightarrow A-B = A-C\\&3.\ A \cup B = A \cup C \text{ 且 } A \cap B = A \cap C \Longrightarrow B = C\\&4.\ P(A) \in P(B) \Longrightarrow A \in B\end{aligned}$$

- 对于定理式的证明（上一则技巧无法解决），一般采用：
  1. 等式 / 重言蕴含的证明：使用外延定理分解为谓词逻辑，记得先声明 $\forall x$ ，从等式左边定义等价 / 重言蕴含推导，中途有必要可以引入新的约束变元，但量词要写在里面；（如果是关于笛卡尔积、二元组的证明，可以写 $\forall\langle x, y\rangle$）
  2. 等价证明：一般拆成双侧重言蕴含；如果是较为简单的、一眼看出的，可以一直等价到底；

  e.g., 证明下列各式：

  $$\begin{aligned}&1.\ A-B = A-(A \cap B)\\&2.\ A-B = A \cap -B\\&3.\ A \subseteq B \Longleftrightarrow P(A) \subseteq P(B)\\&4.\ A\text{为传递集合} \Longleftrightarrow P(A)\text{为传递集合}\end{aligned}$$

- 对于仅由集合运算符连接的式子的证明，要么一直使用等价变换，要么使用外延定理定义转化为谓词逻辑证明；

  e.g., 证明以下式子：

  $$\begin{aligned}&1.\ P(A) \cup P(B) \subseteq P(A \cup B)\\&2.\ P(A-B) \subseteq (P(A)-P(B)) \cup \{\phi\}\end{aligned}$$

3.3 错题

- 证明：$A \subseteq P(\bigcup A)$

  错因：忘记考虑 $A=\phi$ 的情况需要分开讨论；

Chapter 4 关系

4.1 重要概念集合

- 二元关系（**有序对的集合**）：若一个集合满足以下两个条件之一：
  1. 集合非空，且它的元素均为有序对；
  2. 集合为空集；

  则称该集合为一个二元关系；记为 R，简称**关系**；
- A 至 B 的二元关系：设 A，B 为集合，则 A × B 的**任一子集**所定义的二元关系，称为 **A 到 B 的二元关系**；

  $R \subseteq \{\langle x, y\rangle \mid x \in A \wedge y \in A\}$（可以推广至 n 元关系）
- 特殊关系
  - 恒等关系：$I_A=\{\langle x, x\rangle \mid x \in A\}$
  - 全域关系：$E_A=A \times A$
  - 空关系：$\phi \subseteq A \times A$
- 关系的定义域 $dom(R)$、值域 $ran(R)$、域 $fld(R)$（定义域和值域的并）
- 关系的表示

- 关系矩阵：对于集合 $X=\{x_1,...,x_m\}$，$Y=\{y_1,...,y_n\}$，若 R 为从 X 到 Y 上的一个关系，则 R 的关系矩阵（bool 矩阵）为 $M(R)=(r_{ij})_{m\times n}$（如果有序对 $\langle x,y\rangle$ 在 $X\times Y$ 中，则 $r_{ij}=1$，否则为0）；

  > 起始集合元素为**行**，目的集合元素为**列**；

- 关系图：使用**有向边**表示两个集合元素点的关系；

- 关系的运算（设 R 为 X 到 Y 的关系，S 为 Y 到 Z 的关系）
  - 逆：$R^{-1}=\{\langle x,y\rangle|\ \langle y,x\rangle\in R\}$
  - 合成：$S\circ R=\{\langle x,y\rangle|\ \exists z(\langle x,z\rangle\in R\ \wedge\langle z,y\rangle)\in S\}$

    > 遵循和函数复合一样的含义，内层先运算；

  - A 在 R 下的象：$R[A]=\{y\mid\exists x(x\in A\ \wedge\langle x,y\rangle\in R)\}$
  - R 在 A 下的限制：$R\uparrow A=\{\langle x,y\rangle|\ \langle x,y\rangle\in R\wedge x\in A\}$

    > 象和限制可以理解为生物学上的"**平板影印**"，象是将 存在关系 R 中的元素影印在 A 上，限制是将 存在于 A 中的元素影印到 R 上；没有被影印到的元素就被抛弃；

  补充：关系矩阵和定义的联系：第一部分

  1. 关系的逆**相当于关系矩阵的转置**：$M(R^{-1})=M^T(R)$
  2. 关系的复合**相当于关系矩阵逻辑乘法**：$M(S\circ R)=M(R)\cdot M(S)$

- 关系的运算性质
  - 逆的性质
    1. 关系的逆会使定义域和值域对调；
    2. $(R^{-1})^{-1}=R$
    3. $(S\circ R)^{-1}=R^{-1}\circ S^{-1}$
  - 关系合成结合律：$(R\circ S)\circ Q=R\circ(S\circ Q)$

    关系合成不符合交换律：$R\circ S\neq S\circ R$;

  - 关系合成对**并运算**满足分配律：

    $R_1\circ(R_2\cup R_3)=R_1\circ R_2\cup R_1\circ R_3$

    $(R_1\cup R_2)\circ R_3=R_1\circ R_3\cup R_2\circ R_3$

    关系合成对 交运算 不满足分配律：

    $R_1\circ(R_2\cap R_3)\subseteq R_1\circ R_2\cap R_1\circ R_3$

    $(R_1\cap R_2)\circ R_3\subseteq R_1\circ R_3\cap R_2\circ R_3$

- 关系的性质（设 R 为集合 A 上的关系）

  第一组：自反性

  - 自反性：$R\text{在}A\text{上自反}\iff\forall x(x\in A\to\langle x,x\rangle\in R)$

    反自反性（在有些教材里被称为非自反性）：$R\text{在}A\text{上反自反}\iff\forall x(x\in A\to\langle x,x\rangle\notin R)$

    非自反性（不常用）：**即自反性的反面，"不自反的都算非自反"**，下面将不再赘述；

  - **补充：关系矩阵和定义的关系**
    1. R 自反 当且仅当：M(R) **全部对角线元素=1**；
    2. R 反自反 当且仅当：M(R)**全部对角线元素=0**；
    3. R 非自反 当且仅当：M(R)对角线元素**不全为1**；

    > 注：对于关系图而言，图上有自环；

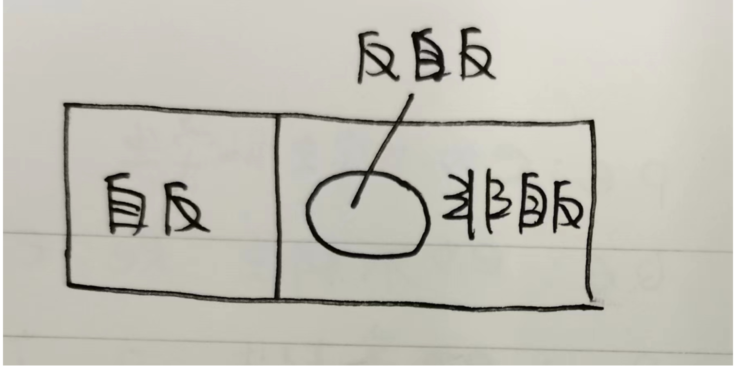
  - **自反性的重要结论**：
    1. $R\text{是自反的}\iff I_A\subseteq R$（反自反可以同理）
    2. $R\text{是自反的}\iff R^{-1}\text{是自反的}$

       $R_1，R_2\text{是自反的}\iff R_1\cup R_2\text{和}R_1\cap R_2\text{是自反的}$

       （反自反完全相同）

3. R_1, R_2是自反的 $\Longleftrightarrow R_1 \circ R_2$ （**反自反不满足这个性质**）

- 自反关系的 Venn 图



第二组：对称性

- 对称性：R在A上对称 $\Longleftrightarrow \forall x \forall y(x, y \in A \rightarrow (xRy \rightarrow yRx))$

  反对称性：R在A上反对称 $\Longleftrightarrow \forall x \forall y(x, y \in A \rightarrow (xRy \rightarrow \neg yRx))$

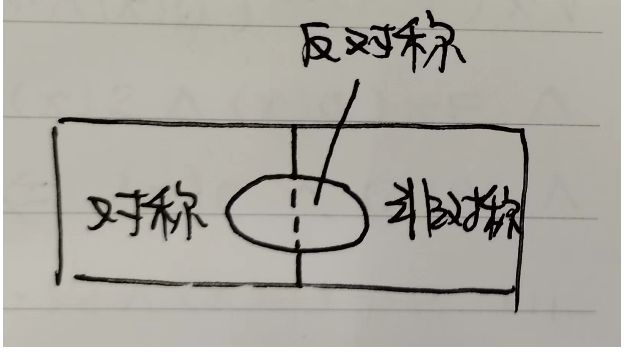
  非对称性：略；

  > 反对称性根据题目证明的要求，还可以写成：
  >
  > $\forall x \forall y(x, y \in A \wedge xRy \wedge yRx \rightarrow x = y)$
  >
  > $\forall x \forall y(x, y \in A \wedge xRy \wedge x \neq y \rightarrow \neg yRx)$
  >
  > 等

- **补充：关系矩阵和定义的关系**
  - R 对称 当且仅当：M(R)**为对称阵**；
  - R 反对称 当且仅当：**M(R) 的所有对称位不同时为1**； （可以同时为0）
  - R 非对称 当且仅当：M(R) **不为对称阵**；
- **对称性重要结论**：
  1. R为对称的 $\Longleftrightarrow R = R^{-1}$

     R_1，R_2是对称的 $\Longleftrightarrow R_1 \cup R_2$和$R_1 \cap R_2$是对称的
  2. **反对称有特殊**：$R_1 \cup R_2$不一定反对称

     R为反对称 $\Longrightarrow R \cap R^{-1} \subseteq I_A$
  3. **复合运算完全不行！**
- 对称关系的 Venn 图



第三组：传递性

- 传递性：R在A上传递 $\Longleftrightarrow \forall x, y, z \in A(xRy \wedge yRz \rightarrow xRz)$

  反传递性：R在A上反传递 $\Longleftrightarrow \forall x, y, z \in A(xRy \wedge yRz \rightarrow \neg xRz)$

  非传递性：略；
- **传递性重要性质**：
  1. 传递性和自反性、对称性一样，满足性质的关系的**逆**、**交**都是传递的，**但并运算不是**；
  2. **复合运算还是不行……**

- 关系性质的总结：如果 R 具有对应的关系那么......

(是否具有？)	自反性	对称性	传递性	反自反性	反对称性	反传递性
R的逆关系	✔	✔	✔	✔	✔	✔
R的并关系	✔	✔	✘	✔	✘	✘
R的交关系	✔	✔	✔	✔	✔	✘
R的复合关系	✔	✘	✘	✘	✘	✘

- 关系的闭包：某些关系不满足一些性质，为了方便研究，向关系集合中加入**尽可能少的**元素生成一个**满足该性质的超集合**，称为闭包；
  - 前置性质
    1. $R^0 = I_A$
    2. $R^{n+1} = R^n \circ R,\ R^m \circ R^n = R^{m+n},\ (R^m)^n = R^{mn}$
    3. 有限集上关系幂序列的周期性：若存在自然数 s、t 使得：$R^s = R^t$，则：

       $R^{s+k} = R^{t+k},\ R^{s+kp+i} = R^{s+i}$

       $B = \{R^0, R^1, \ldots, R^{t-1}\} \Rightarrow \forall q \in N,\ R^q \in B$（思考：为何是最大的t？）
  - 闭包定义：设 R 为非空集合 A 上的关系，若 A 上有另一关系 R'，满足：
    1. $R \subseteq R'$（**扩展得来**）
    2. R' 是具有对应性质的（扩展集具有性质）
    3. 对 A 上任何具有同样性质的关系 R'' 有：$R' \subseteq R''$（**扩展集最小**）
  - 自反闭包记为 $r(R)$，对称闭包记为 $s(R)$，传递闭包记为 $t(R)$
  - 闭包的性质
    1. **已有即当前**：$R\text{为自反的} \iff r(R) = R$（其他同理，下面不再赘述）
    2. 不同的关系可能多出“不在扩展中的元素”：$R_1 \subseteq R_2 \Rightarrow r(R_1) \subseteq r(R_2)$
    3. **仅对并运算 + 自反/对称成立分配律**：

       $r(R_1) \cup r(R_2) = r(R_1 \cup R_2)$

       $s(R_1) \cup s(R_2) = s(R_1 \cup R_2)$

       $t(R_1) \cup t(R_2) \subseteq t(R_1 \cup R_2)$
    4. **闭包叠加**：自反性“不受影响、不影响别人”，对称性“自己不受影响，只是会影响传递”，传递性“不影响别人，但会被对称性影响”

       $R\text{自反} \Longrightarrow s(R),\ t(R)\text{自反}$

       $R\text{对称} \Longrightarrow r(R),\ t(R)\text{对称}$

       $R\text{传递} \Longrightarrow r(R)\text{传递}$

       因此我们知道了，想要求叠加闭包，传递性质最脆弱，需要最后求传递闭包

       $rs(R) = sr(R),\ rt(R) = tr(R),\ st(R) \subseteq ts(R)$

       求“等价闭包”：$tsr(R)$
  - 闭包的构造：若 R 不满足对应的性质
    - $r(R) = R \cup R^0$
    - $s(R) = R \cup R^{-1}$
    - $t(R) = \bigcup_{i=1}^{\infty} R^i$，特别地，对于有限非空集 A，一定存在一个正整数 $k \leq |A|$，使得：

      $t(R) = R^+ = \bigcup_{i=1}^{k} R^i$

      > 这样的操作联想到图论中的**求两点间是否有路径**的算法，所以这里也可以使用 Warshell 算法

- 等价关系、等价类、商集、划分

  这一部分定义很长，这里仅仅说明一下对概念的理解：

  - 等价关系：**同时满足自反、对称、传递的关系**
  - 等价类：$[x]_R = \{y \mid y \in A \wedge xRy\}$，**A 中所有和元素 x（包含自身）满足 R 关系的元素组成的集合**；

    等价类的所有性质也可以理解为：相互等价的元素一定在一个集合（等价类）里，不相互等价的一定不在一个等价类里；
  - 商集：A 的所有等价类构成的集合（是集合的集合）
  - 划分：数学家们通过证明发现，用元素等价区分的方式刚好能够划分一个集合，也就是说，**一个等价类和一个划分一一对应。**数学上称为**诱导**，等价关系可以诱导出一个对应的划分，一个划分又可以诱导出一个对应的等价类；

    划分的性质：（数学形式描述请查阅资料）

    1. 分块全部来自 A、且每个分块不为空；
    2. 所有分块完全覆盖 A；
    3. **每个分块不重叠**；
- 相容关系、相容类、覆盖
  - 相容关系：**同时满足自反、对称（比等价关系缺少传递）的关系**；
  - 最大相容类性质：对任意不在最大相容类中的一个元素，总能找到类中的一个元素和它不满足相容关系；

  只要理解覆盖和划分的区别，覆盖是每个分块能够重叠；

  完全覆盖的唯一性；覆盖能够确定一个相容关系、相容关系能够确定一个完全覆盖；

  > 在关系图中，找相容关系相当于找 **所有的极大完全子图**（这个分块内每个元素都要有相互的关系），边可以重叠

  - **由覆盖构造相容关系**：给定非空集 A 上的一个覆盖 $\Omega = \{A_1, A_2, ..., A_n\}$，则由其确定的关系 $R = \bigcup_{i=1}^{n} A_i \times A_i$ 是 A 上的一个相容关系（不一定是最大相容关系）；

- 偏序关系：同时满足**自反、反对称、传递**性质的关系

  拟序关系：同时满足**反自反、传递**性质的关系（由这两个性质能推出**反对称性**）

  > 偏序关系能理解为抽象的 “≤” 关系
  >
  > 拟序关系能理解为抽象的 “ < ” 关系

  偏序关系和拟序关系仅在自反性上有差别，拟序和偏序讨论一个即可

  e.g.1, 对集合 A，在 P(A) 上的包含关系是偏序关系；在 P(A) 上的真包含关系是拟序关系；

  e.g.2, 可以通过增添/删减自反性来变换两者：$R \to R - R^0$ 或 $R \to R \cup R^0$
- 结构、偏序集

  结构：集合 A 及其上的关系 R 一起称为一个**结构**；

  偏序集：若结构中的关系 R 是偏序关系，称这个结构为**偏序集**，记作$\langle A, R \rangle$；

  > $\langle N, \leq \rangle$、$\langle P(A), \subseteq \rangle$ 都是偏序集

- 哈斯图：由于偏序关系自反（自环）、反对称（两顶点间最多一条有向边），所以表达偏序关系的图可以：①省略自环；②适当安排位置（默认偏序箭头向上）不画边的方向；③不画传递得到的边，这种图称为哈斯图（描述性定义）；

  预定义：盖住关系，对偏序关系 $\langle A, \preccurlyeq \rangle$，若 $x, y \in A$，$x \preccurlyeq y$，$x \neq y$，且不存在 $z \in A$ 使得 $x \preccurlyeq z \wedge z \preccurlyeq y$，则称 y 盖住 x；（理解为抽象的**直接后继关系**）

  定理：对所有偏序集$\langle A, \preccurlyeq \rangle$，A 上的盖住关系 cov A 唯一；

  画法：① 每个顶点代表 A 的一个元素；② 如果 $x \preccurlyeq y \wedge x \neq y$，将 y 置于 x 上方（后继在上）；③ 仅在 cov A 中有的关系才能在哈斯图上连接无向边（这保证了不画传递的边）；

- 可比：对于偏序集 $\langle A, \preccurlyeq\rangle$，若$\forall x, y \in A \Rightarrow x \preccurlyeq y \vee y \preccurlyeq x$，则称 x 和 y 可比；

  > **不可比不代表不能讨论 x 和 y 的关系，只是说两者双向的 $\preccurlyeq$ 都是假而已；**

- 偏序关系的上下界

  最小元的感性理解：必须**和其他所有元素构成偏序关系，且它是所有元素的"前驱"**；（最大元同理）；**因此最小/大元不一定存在（不一定与所有都可比），但如果存在一定唯一**；

  极小元的感性理解：**不能是某一个元素的"后继"就行**，可以与某些元素不可比；**因此极小元必然存在，可能不唯一**；

  上下界和上下确界：比较明显，意会一下~

- 全序关系的感性理解：一个偏序集 $\langle A, \preccurlyeq\rangle$ 中，所有元素间都可比，就称这个集为**全序集**，这个 $\preccurlyeq$ 关系称为**全序关系**；

  > 很容易理解，有限的全序集一定是有最大、最小元的；
  >
  > $\langle N, \leq\rangle$ 是全序集，$\langle P(A), \subseteq\rangle$ 不是；**因为自然数集上所有元素对于"小于等于"两两可比，但幂集的元素——集合可能两者都没有包含关系，即对于"包含关系"不可比**；

- 链的感性理解：将偏序集中相互可比的元素，构成一个子集，这个子集就是**链**，其中元素数称为**链长**；

  > 为什么叫链？**因为如果一系列元素可比，那么在哈斯图上呈现的是一条链，不会出现分支**；
  >
  > 这也是为什么全序关系又称为"线序关系"的原因；
  >
  > 找全序关系就是找贯穿所有元素的链；

- 良序关系：一个偏序集 $\langle A, \preccurlyeq\rangle$，A 的任意非空子集都有最小元，那么 $\preccurlyeq$ 就叫做**良序关系**，这个偏序集叫做**良序集**；

4.2 常见题型

- 求集合划分的种类数

  > 结论：贝尔数 B_n；[知识链接🔗]
  >
  > $B_n = C_{n-1}^0 B_{n-1} + C_{n-1}^1 B_{n-2} + \cdots + C_{n-1}^{n-1} B_0$；
  >
  > 可以用贝尔三角形每行第一个数来计算；
  >
  > 考到就仰仗自己的数学吧😱
  >
  > 帮你算几个：$B_0 = B_1 = 1,\ B_2 = 2,\ B_3 = 5,\ B_4 = 15,\ B_5 = 52,\ B_6 = 203$
  >
  > 考场上基数大于 6 的集合的划分数，如果出到，请好好地问候出题老师！

- 求各种闭包
- 求划分/等价类的元素和关系、由覆盖求对应的相容关系

  按照定义和性质，按部就班地来即可

- 证明某个关系的性质，证明关系符合的公式

  思路：一般情况下，关系的元素是有序对，所以证明时一般都先声明一个 $\forall\langle x, y\rangle$，然后依次向下推理即可；

  特别地，如果是已定义的关系，例如等价关系，那么按照定义来

  > e.g.1, 设 R, S, T 是 A 上的关系，证明：$R \circ (S \cup T) = (R \circ S) \cup (R \circ T)$
  >
  > e.g.2, 已知 $A = Z_+ \times Z_+$ 和 A 上的关系 $R = \{\langle\langle x, y\rangle, \langle u, v\rangle\rangle \mid xv = yu\}$；证明 R 是等价关系；
  >
  > e.g.3, 设 $\langle A, R_1\rangle$，$\langle B, R_2\rangle$ 是两个偏序集，定义 $A \times B$ 上的关系 R：对 $\forall a_1, a_2 \in A, b_1, b_2 \in B$，有 $\langle a_1, b_1\rangle R \langle a_2, b_2\rangle \Longleftrightarrow a_1 R_1 a_2 \wedge b_1 R_2 b_2$，证明 R 是 $A \times B$ 上的偏序关系；

- 给某个关于关系的公式找出反例

  思路：对于自反、对称而言，一般只需要找二阶关系矩阵就能发现矛盾；对于传递而言，需要从三阶关系矩阵开始找，先为每个矩阵设定一个传递关系，其他空不填，假设是0，看是否能成为反例；如果不行，在对空白处进行值的给定；

- 判断一个结构是不是偏序关系：按定义来
- 考察偏序关系最大/小元、极大/小元的定义
- 画出偏序关系的哈斯图

> e.g., 对下列集合上的整除关系画出哈斯图，并指出在这个关系下的极大元、极小元、最大元、最小元（如果有的话）
>
> `A = {1, 2, 3, 4, 6, 8, 12, 24}` 和 `B = {1, 2, 3, 4, 5, 6, 7, 8, 9}`
>
> 思考：从哈斯图上，我们能够如何更快地看出极大极小元？

- 找到某个特定集合的一个全序关系

> 找全序关系就是找一条贯穿集合所有元素的链
>
> e.g., 找出在集合 `{0, 1, 2, 3}` 上包含 $\langle 0, 3\rangle$ 和 $\langle 2, 1\rangle$ 的全序关系；
>
> **思路：将所有元素排成一条链，转化为排队问题**；
>
> 如果要求包含某个元素，等价于某个元素必须排在另一个元素的前面/后面；
>
> 提示：本例共有 $C_3^1 + C_3^2 = 6$ 种满足条件的全序关系；

Chapter 5 函数

5.1 基本概念

函数这章掌握几个概念和结论就行

- 定义：就是一种集合 A 到集合 B 间的特殊关系；

  > 注意：$dom(f) = A$，不然就不是函数了

- 部分函数：$dom(f) \subset A$，例如 $f : R \to R$ 的函数 $f(x) = \dfrac{1}{x}$，只有减去定义域中的 $\{0\}$ 或者在 $x = 0$ 处添加定义，才能变为函数；
- 所有函数的集合：$A_B = \{f \mid f : A \to B\}$，也写作 B^A，**基数** $|A_B| = |B|^{|A|}$；

  > 总数小于总关系数 $2^{|A||B|}$，**因为**：$A \neq \phi$，$B = \phi$ **不是函数**、$A = \phi$，$B \neq \phi$ **只对应一个函数：空函数**；

- 单射、满射、双射的定义
- 常用函数的定义：常函数、恒等函数、n 元运算、泛函、特征函数、典型映射；
- 函数的合成

  着重掌握特性：

  定理：若 f, g 满射，则$f \circ g$ 满射（单射、双射同理）

  逆定理：$f \circ g$ 满射则 f 满射；$f \circ g$ 单射则 g 单射；$f \circ g$ 双射，则 f 满射、g 单射；

  逆定理记忆：全部都是“外层满、内层单”

- 函数的逆

  着重掌握左逆、右逆的性质：

  定理：f 既有左逆、又有右逆，等价于 f 双射且左右逆相等；f 有左逆 $(g \circ f)$ 等价于 f 单射；f 存在右逆 $(f \circ g)$ 等价于 f 满射；

  记忆：还是“外层满、内层单”，函数在什么位置有逆，就是什么侧的满/单；

5.2 常见题型

- 判断单/满/双射
- 给定集合，要求构造集合间的 单射/满射/双射函数；

  一般情况都比较简单，有些不好想的需要记忆一下，例如：构造从 $N \times N$ 到 N 的双射函数；答案是在点阵中有序环绕：$f(\langle m, n\rangle) = \frac{1}{2}(m+n)(m+n+1)+m$；大部分其他的无穷集上的构造也多采用这种方法；

5.3 错题

- 关于 $A \to \phi$ 的函数,下列（）是正确的

  A．不存在

  B．有一个空函数 Φ

  C．仅当 A 非空时才能有函数

  D．仅当 A 为空时才能有函数

  事实上，$A \neq \phi$，$B = \phi$ **的函数不存在，但如果 A、B 都为空，那么可以是空函数**；

Chapter 6 图论

6.1 图论中的重要定义 I

> 图的起源：人们关心一类问题，给定的两点间是否有一条或多条连线的关系，而连接方式无关紧要。这类问题在数学上的抽象是**图**；

- 图的数学定义：一个图指**有序三元组**$(V(G), E(G), \psi_G)$，$V(G)$为**非空**（空图特殊，不参与讨论）顶点集，$E(G)$是不与$V(G)$相交的边集，ψ_G为关联函数；

  > 约定：对于图 G，一般用符号 $V(G)$ 表示顶点集、$E(G)$ 表示边集、$\nu(G)$ 表示顶点数，$\varepsilon(G)$ 表示边数；若上下文仅有一个图，则省略 “(G)”；
  >
  > 注：无向图可以表示为二元组，即 V 和 E；

- 顶点对的定义：ψ_G 使 G 的每条边对应于 G 的 无序的**顶点对**；
- 连接、端点的定义：若 e 为 G 的一条边，u、v 是使 $\psi_G(e) = (u, v)$ 的顶点，则称：e **连接** u、v，顶点 u、v 称为 e 的**端点**；
- 关联、相邻、自环：一条边的端点与这条边**关联**；与同一条边关联的两个顶点称为**相邻**；端点重合为一点的边称为**自环**；
- 平面图、非平面图：边仅在端点相交的图称为**平面图**，反之为非平面图；
- 平凡图、非平凡图：仅有一个顶点的图称为**平凡图**；
- 有向边、无向边：可由端点 v_i 和 v_j 表示 $e_k = (v_i, v_j)$ 的边称为**无向边**（vi、vj互为直接前驱、直接后继）；可由有序二元组 $e_k = \langle v_i, v_j\rangle$ 表示的边是**有向边**（vi是vj的直接前驱、vj是vi的直接后继）；
- 有向图、无向图：所有边都是有向边的图是**有向图**，反之是**无向图**，否则是混合图；

  > 可以将无向边视作双向有向边，因此此后不讨论混合图；

- 简单图、有向简单图：既没有**自环**，又没有**重边**的图，如果是无向图，称为**简单图**；如果是有向图，称为**有向简单图**；

  > 1. 重边的定义：有两条及以上条边连接同一对顶点，称这个边为重边；
  >
  >   ⚠️**易错警示：对有向边，A--->B 和 B--->A 组合不算重边！A--->B 和 A--->B 组合才是**；

> 2. 如果不强调“有向”，一般情况下“图”指“无向图”；不过在定义中，如果发现只描述无向图的，大概率对有向图也适用，不然就单独拎出来了；

- 完全图、有向完全图：每对不同顶点都有一条边相连的**简单**图称为**完全图**；有向完全图同理；

  特别地，将 n 个结点的完全图记作 K_n，但有向完全图没有这种记法；

  > **定理1：** $\varepsilon(K_n) = C_n^2$ **且对n 个结点的有向完全图G：** $\varepsilon(G) = A_n^2$；
  >
  > （因为A--->B 和 B--->A 组合不算重边）

- 偶图（或者称二部图）：一个图 G 的顶点集 $V(G)$ 可以分解为两个子集 X、Y，使得：每条边都有一个顶点在 X 中，另一个顶点在 Y 中；这样的一种分类 (X, Y) 称为 G 的一个**二分类**；

  > 理解：按分解的两个点集“切一刀”，所有边都被砍断的图；

- 子图：若 $V(H) \subseteq V(G)$，$E(H) \subseteq E(G)$，ψ_H 为 ψ_G 在 $E(H)$ 上的限制，则 H 为 G 的**子图**，记作：$H \subseteq G$；（真子图略）
- 母图：若 $H \subseteq G$，称 G 为 H 的**母图**；
- 生成子图（或称支撑子图，spanning sub-graph）：$H \subseteq G$ 且 $V(H) = V(G)$，称 H 为 G 的**生成子图**；
- 导出子图：有点抽象，一般用不到定义，想要形象地了解见：图的运算；
- 基础简单图：一个图 G 删去所有**“多余”的边**，使图中恰没有重边、自环，得到的这样的**简单生成子图**称为**基础简单图**；
- 赋权图（或称加权图）：若给图 G 的每条边都赋以实数 w_k 作为该边的权，称 G 为赋权图；
- 顶点的度：图 G 的顶点 v 的度记为 $d_G(v)$，指 G 中与 v 相关联的数目；
  - 约定：$\delta(G)$、$\Delta(G)$ 表示 G 的所有顶点的最小度、最大度；
  - 度为0的点称为**孤立点**；
  - **对于有向图，**$d(v) = d_+(v) + d_-(v)$，d_+ **为正度/入度，**d_- **为负度/出度**；
  - **自环贡献一个入度、一个出度**；

  > **定理2：（握手定理）**$\sum_{v \in V} d(v) = 2\varepsilon$ **（所有结点的度之和为边数的2倍，有向图也是）**；
  >
  > 推论：对任何图，度为奇数的点（称**奇点**）的个数为偶数；
  >
  > **定理3：有向图中，**$\sum d_- = \sum d_+ = \varepsilon$ **（入度和=出度和=边数，是入度出度平分的意思）**
  >
  > **定理4：非空简单图（**$\varepsilon > 1$**）一定存在度相同的结点**；

6.2 图的同构

> 注：图同构问题分为4类：精确图完全同构、精确子图同构、不精确图完全同构、不精确子图同构；现在学界已证明后三者是 NP 完全问题；计算机离散数学-图论、数据结构（包括下面的内容）讨论的是第一种问题；

- 恒等图的定义
- 同构图的定义：如果存在两个**一一映射（双射）**$\theta: V(G) \to V(H)$，$\phi: E(G) \to E(H)$，使 $\psi_G(e) = (u, v)$ 当且仅当 $\psi_H(\phi(e)) = \theta(u)\theta(v)$，则将这样的映射对 (θ, ϕ) 称为 G 和 H 间的一个同构；将 G 与 H 同构关系记为 $G \cong H$；

  > 理解：边边和点点必须一一相应；
  >
  > 就是在不添加边和点、不删除边和点的基础上**任意移动顶点的相对位置、为顶点和边改名**，所产生的不同形态的图；

- 关于同构的定理

  > **定理5：（同构的必要条件）两个同构图的结点度的非增序列相同**
  >
  > **定理6：（同构的必要条件）若 G1 与 G2 同构，则 G1 的任意导出子图都有 G2 的导出子图与其同构**；
  >
  > 其实还有一个必要条件过于明显，不作为定理：两同构图的顶点数、边数相等；

- 判断方法

    - 判断两图同构：**按定义，找到两个一一映射**；

      > 注：根据定义，可以得出一个显然的方法：**一个图的邻接矩阵经历有限次的行互换、列互换，能变成另一个图的邻接矩阵，那么这两个图同构**；

    - 判断两图不同构：使用定理5、6（必要条件），不满足必要条件的就不是；

6.3 图的存储实现

- 图的关联矩阵（**行是顶点，列是边**）：因为空间原因，不做存储图的方法；

  > 虽然不做存储方法，但在讨论**树、有向连通图、电路图的某些性质**时比较有用，感兴趣戳这里（不在初级数据结构要求范围内）；

    - 无向图的关联矩阵：可以由 bool 矩阵表示，1是有关联，0是没有关联；
    - 有向图的关联矩阵：+1表示该边离开该结点，即正度/出度；-1表示该边进入该结点，即负度/入度；0表示没有关联；
- 图的邻接矩阵表示法：对任意的图 G，对应一个 $\nu \times \nu$ 的邻接矩阵 $A(G)=[a_{ij}]$，其中 a_{ij} 为 v_i、v_j 的连接数目；（空间：$O(|V|^2)$）

  进一步，在数据结构中更常用的是"加权图的邻接矩阵"存储方法，可以兼顾非加权图：

  $$A[i][j]=\begin{cases}\omega, & <i,j,w>\in E\\ 0, & i=j\\ \infty, & otherwise\end{cases}$$

  C++代码实现统一放置于附录中，有需要可以前去查看，下同；
- 图的邻接表表示法：改进了邻接矩阵表示法在面对稀疏矩阵时浪费空间、不易维护的问题；

```
结点集(结点数组)：node1 {结点值，与该结点相邻的直接后继(有方向)结点索引链表头指针}
 |-------------------------------------------------------------|
 v
边集(单链表)：node2 {结点索引（不能放结点值，因为无法完成后面的遍历运算， 下一node2}

+ 保存边数、保存结点数
空间：O(|V|+|E|)
```

6.4 图的运算实现

- 图的基本运算
    - 差运算：**（要求 G_2 为 G_1 子图）** $G_1-G_2=(V_1,E_1-E_2)$；
    - 补运算：n 个结点的简单图的补图 $\overline{G}=K_n-G$；
    - 删去结点 v 及其关联的边：$G-v$

      > $G-v$ 为 G 的**导出子图**：有助于理解导出子图的意义；

    - 删去边 e：$G-e$

      > $G-e$ 为 G 的**生成子图**：有助于理解生成子图的意义；

    - 增加边 $e_{ij}=(v_i,v_j)$：$G+e_{ij}$
- 数据结构中图的基本运算：创建、判边、增删边、查点边数、遍历（后面分开讨论）；
- 图的运算实现：**对于不同的存储方式，图的运算时间复杂度有所不同**
    - 邻接矩阵表示的运算实现
    - 邻接表表示的运算实现

6.5 图论中的重要定义Ⅱ

- 道路和回路：在无向图 $G=(V,E)$ 中，若边点交替序列 $P=(v_{i1},e_{i1},v_{i2},e_{i2},\ldots,e_{iq-1},v_{iq})$ 满足：v_{ik}、v_{ik+1} 为 e_{ik} 的两个端点，则称 P 为 G 的一条**道路**；特别地，如果 $v_{i1}=v_{iq}$，那么称道路 P 为 G 的一条**回路**；
  - 如果 P 序列中没有重复的边，则 P 称为**简单道路（或称"迹"）、简单回路（或称"闭迹"）**；
  - 更特别地，如果 P 序列中结点也不重复（结点不重复是边不重复的充分不必要条件），则称 P 为 G 的**初级道路、初级回路**；
- 有向道路和有向回路：在有向图 $G=(V,E,\psi_G)$ 中，若边序列 $P=(e_{i1},e_{i2},\ldots,e_{iq})$，其中 $e_{ik}=(v_l,v_j)$，则称 P 为 G 的**有向道路**；若 e_{iq} 终点也是 e_{i1} 的始点，则称 P 为 G 的**有向回路**；
  - 同样有：简单有向道路、简单有向回路、初级有向道路、初级有向回路的概念；

> ⚠易错点：**平凡图一定是道路，但一定不是回路！**

- 连通性、强连通性、弱连通性、单向连通性
  - 无向图考虑"连通"：两结点间至少存在一条道路，则这两个结点间连通；
  - 有向图考虑：
    1. 两结点间存在一条从 v_i 到 v_j 的有向道路 **且** 存在另一条从 v_j 到 v_i 的有向道路，则称 v_i 和 v_j **强连通**；
    2. 两结点间**仅**存在一条从 v_i 到 v_j 的有向道路 **或 仅**存在另一条从 v_j 到 v_i 的有向道路，则称称 v_i 和 v_j **单向连通**；
    3. 两结点间 **不考虑所有道路的方向（称为"有向图的底图"）**，若这两个结点连通，则称 v_i 和 v_j **弱连通**；
- 连通图、连通分量（或称"连通支"）
  - 无向图 G 中任意两结点间都是连通的，则 G 为**连通图**；
  - G 的连通子图（子图且连通）H 不是 G 的任何其他连通子图的真子图，称 H 为 G 的一个**极大连通子图**，也称**连通分量**；

> 有些不严谨的题问有向图"是不是连通图"，就将它看成一个无向图（忽略方向）；

- 强连通图、强连通分量
  - 有向图 G 中任意两结点间都是强连通的，则 G 为**强连通图**；
  - G 的强连通子图 H 不是 G 的任何其他强连通子图的真子图，称 H 为 G 的一个**极大的强连通子图**，也称**强连通分量**；

> ⚠易错点1：**平凡图也单独算一个连通分量 / 强连通分量！**
>
> ⚠易错点2：因为无向边看作"双向边"，所以连通的无向图一定是强连通的；

> **推论：图 G 的每个连通分支都是其导出子图**；
>
> **小结论：图 G 对应关联矩阵记为 $M(G)$，则 G 的连通分支数为 $r(M(G))-1$**；

- 割边与非割边、割点与非割点：删去图中某个边 / 点，图的连通分支数（连通性）改变，则称该边 / 点为**割边 / 割点**；
- 欧拉道路、欧拉回路：**无向连通图** G 中的一条经过所有边的**简单道路/回路**称 G 的**欧拉道路/回路**；
  - 理解：不重复地遍历所有边，不管点的情况；
  - 注意：**有向图也能讨论欧拉回路的问题，不过要遵循有向的连通性**；

> **定理1：（欧拉回路充要条件）无向连通图 G 存在欧拉回路 $\Longleftrightarrow$ G 的各结点度数均为偶数**；
>
> **推论1-1：（欧拉道路充分条件）无向连通图 G 仅有2个奇点 $\Longrightarrow$ G 存在欧拉道路**；
>
> **推论1-2：（有向欧拉回路充分条件）有向连通图 G 的各结点的正、负度数相等 $\Longrightarrow$ G 存在有向欧拉回路**（侧面说明有向可能严格一些，不仅结点度全为偶数，而且要进出相等）；
>
> 定理2：连通图 G 有 k 个奇点（由部分 I 的定理可知，k为偶数），则 E(G) 可以划分为 $\frac{k}{2}$ 条简单道路；

- 哈密顿道路、哈密顿回路：**无向图** G 的一条经过全部结点的**初级道路/回路**称 G 的**哈密顿道路/回路（简称H道路/H回路）**；

- 理解：“不重复地遍历所有点”；
- 注意：H 道路 / 回路一般针对简单图，因为重边和自环对它没有什么影响，可以转换为简单图的问题；

> **很遗憾，目前 H 道路 / 回路的判定没有充要条件！一般遍历是 NP 问题……**
>
> **定理3：（H 回路充分条件）完全图 K_n 为 H 图**；
>
> **定理4：（H 回路充分条件）若简单图 G 每个结点度都大于 n/2，则 G 为 H 图**；
>
> 说明：平均每个点的度越大，越有可能有H道路、H回路；
>
> **推论4-1：（H 道路充分条件）若简单图 G 的任两结点 v_i, v_j 恒有 $d(v_i)+d(v_j)\ge n-1$，则 G 存在 H 道路；**
>
> 证明提示：有 H 道路一定连通，可以先证连通性；
>
> **推论4-2：（H 回路充分条件）若简单图 G 的任两结点 v_i, v_j 恒有 $d(v_i)+d(v_j)\ge n$，则 G 为 H 图**；
>
> **推论4-3：（H 回路的闭包等价关系）向图 G 中满足 “ $d(v_i)+d(v_j)\ge n$”的不相邻两结点 v_i, v_j 加边，直至无法找到这样的结点对为止，形成的新图称为 G 的闭包（记为$C(G)$）；那么有：** $G\text{ 为}H\text{图} \Longleftrightarrow C(G)\text{为}H\text{图}$；
>
> **推论4-4：（H回路闭包充分条件）若 $C(G)=K_n$，则 G 为 H 图**；
>
> **定理5：（可怜为数不多的 H 回路的必要条件）若 G 为 H 图，则对任意非空顶点集 S，有：** $\omega(G-S)\le|S|$；

- 补充：欧拉图、H 图的定义：有欧拉回路 / H 回路的图才叫~（只有欧拉道路 / H 道路的不是）；
- 判断一个图是 H 图：使用上面的充分条件/等价条件；
- 判断一个图不是 H 图：使用上面的必要条件；

  > 举例：证明 Peterson 图是极大非 H 图（有 H 道路，但没有 H 回路）
  >
  > 【问题：它满足定理5，能否判断一下为什么在删去任意4个顶点时，连通分支数一定小于等于3？】
  >
  > **定理6：（必要条件）若一个点在 H 回路中，那么必定有且仅有两个相连的相异道路**；

6.6 图的简单应用

- 【普通图】有 3L、5L、8L的三个没有刻度的量杯，现在8L的量杯装满了水，其他两个是空的；问如何操作（不撒不漏）可以让8L水分为两个4L水？
- 【二部图】人、狼、羊、菜过河问题

> 解决思路：**“状态转换图”**：将每一个状态抽象为一个顶点，先列出所有可能状态作为顶点，再用“一次能直接转换的关系”作为边连接，最后只需判断在起点（初态）和终点（末态）是否单向连通即可；

6.7 图论中的重要定义Ⅲ

> 提示：本章节不在初级数据结构要求范围内；

- 割边与非割边、割点与非割点：删去图中某个边 / 点，图的连通分支数（连通性）改变，则称该边 / 点为**割边 / 割点**；

  > **定理1：e 为割边，当且仅当 e 不属于 G 的任何回路**；

- 普通树的数学定义：不含任何回路的连通图称为**树**；

  > **定理2：“连通”、“无回路”、“有 n-1 条边”三个条件任取两个都可以作为树的定义**；
  >
  > 推论：“连通+全为割边”、“任意两点间有唯一道路”、“无回路+加一边就一回路” 这三个与树的定义等价；
  >
  > **定理3：树中一定有树叶结点（离散数学中没有空树的说法！只有空图）**

- 根树的定义：若树 T 是有向树，且 T 中存在某结点 v_0 的入度为0、其他结点入度为1，则称 **T 是以 v_0 为根的根树**（或外向树），用 $\vec{T}$ 表示；

> **根树才是数据结构中的“树”！**

- 生成树（或称“支撑树”）：图 G 的一个符合树定义的生成子图称为图 G 的**生成树**；

> 余树：给定图 G 的一棵生成树 T，定义余树 $\overline{T} = G - T$；一般情况下，余树不是树；

- 基本关联矩阵：**上接“关联矩阵存储”，虽然关联矩阵一般不作为存储方法，但有些情况讨论它的性质，可以更方便地解决某些问题**；

> 友情提醒1：这里和**电路理论的电路图研究**结合比较紧密；
> 友情提醒2：这里的讨论对象是**有向连通图**；

  - 定义：在**有向连通图** $G = (V, E)$ 的**关联矩阵** B 中，划去任意任意结点 v_k所对应的**一行**，得到 $(\nu - 1) \times \varepsilon$ 的矩阵 B_k，称为 **G 的一个基本关联矩阵**；
  - 相关定理

> **定理1：有向连通图 G 的关联矩阵 B 满足：**$r(B) = \nu - 1$；
>
> **定理2：有向连通图 G 的基本关联矩阵 B_k 满足：**$r(B_k) = \nu - 1$；
>
> 推论：n个结点树 T 的基本关联矩阵的秩为 $\nu - 1$；
>
> **定理3：有向连通图 G 如果存在回路 C，则 C 中各边所对应基本关联矩阵 B_k 的各列线性相关**；
>
> **定理4：有向连通图 G 的基本关联矩阵 B_k，有：**
> $B_k任意n-1阶子式M_{n-1} \neq 0 \iff M_{n-1}各列对应边构成G的一棵生成树$；
>
> 定理4说明了**可以由 B_k 的非零 n-1 阶子式的数目来代表 G 生成树的数目**；

- 回路矩阵和割集矩阵；

> 不说了亲，这边建议您好好复习电路理论课呢😅

- Huffman树（最优二叉树），详见“数据结构复习-第二部分”；

6.8 图的经典算法

6.8.1 图的遍历算法

- DFS 算法：类似树的前序遍历

  邻接表存储 $O(|V| + |E|)$

  邻接矩阵存储 $O(|V|^2)$

- BFS 算法：类似树的层次遍历

  邻接表存储 $O(|V| + |E|)$

  邻接矩阵存储 $O(|V|^2)$

6.8.2 两点间道路判定算法

> 这里介绍邻接矩阵表示的算法，比较常见；

- 引入：对于一个**非加权图的邻接矩阵（0&1）**，有 $P = (p_{ij})_{n\times n} = \sum\limits_{k=1}^{n} A^k$，则 p_{ij} 为从 v_i 到 v_j 的**道路数**；

  实际问题只关心**是否有道路**，所以可以改成逻辑运算提升速度：$P = (p_{ij})_{n\times n} = \bigvee\limits_{k=1}^{n} A^k$，时间复杂度 $O(\nu^4)$；

- Warshell算法 $O(\nu^3)$

```
P <- A
for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= n; ++j)
        for (int k = 1; k <= n; ++k)
            p_{jk} <- p_{jk} ∨ (p_{ji} ∧ p_{ik})
```

- DFS 和 BFS $O(\varepsilon)$：和图的遍历不一样的是，它比图的遍历更简单，只需从一个点出发（减少最外层循环），用visited数组和BFS/DFS整体寻找，如果遇到终点即停止并返回true，否则返回false；

6.8.3 有向图强连通分支判断算法

> 思路：先从图 G 任一点开始 DFS，如果 G 不是强连通图，则可能得到一个深度优先生成森林；对森林中的每棵树按照**生成次序**依此进行**后序遍历**，并按遍历顺序给每个结点编号（从小到大）；
>
> 然后使 G 的每条边逆向，得到 Gr，再从 Gr 编号最大的结点开始 DFS，得到新的深度优先遍历森林中的**每一棵树**就是 G 的一个强连通分量；

6.8.4 欧拉回路的构造算法

> 欧拉回路有明确的、好判断的充要条件，所以算法设计相对容易；
>
> 无论啥算法，最好先利用充要条件排除没有欧拉回路的图，能大大提高时间性能；
>
> 下面讨论**如果有欧拉回路，应该怎么找**的算法：

- 拼接法：DFS寻找回路（经过即删除），如果回路结束却仍然有未遍历的结点，则从新的未访问的结点开始遍历回路，**并拼接（"8"字原理）**，循环直到所有边已被访问；
- Floyd算法（非割边优先遍历）

6.8.5 欧拉回路的应用：中国邮递员问题（CPP）

> 中国邮递员问题：**走遍图中的所有边**后返回返回起点，要求**总路程最短**；

- 对于无向图 G 的结论
  - 如果 G 中所有结点个数都是偶数：该图的任一欧拉回路都是解；
  - 如果 G 中有且仅有 2 个奇点 v_i 和 v_j：找到 G 从 v_i 到 v_j **欧拉道路** E_{ij}，再找从 v_j 到 v_i 的**最短路径** P_{ji}，则回路 $E_{ij}+P_{ji}$ 就是问题的解；
  - 如果 G 中有两个以上，共2k个奇点（由前面图的性质推论，奇点必有偶数个）：

$$\text{图}G\text{有最佳邮路}L \iff \begin{cases} 1.\ L\text{的任一边最多重复一次} \\ 2.\ \text{对}G\text{中的任一回路}C\text{，}L\text{中在}C\text{上重复边的长度之和} \\ \quad\ \text{不超过}C\text{总长的一半}\ (\text{必须遍历所有包含重边的回路}) \end{cases}$$

    实际做法是：找出所有奇点，两两配对并依此为奇点间添加重复边（长度和原边相等），为它们配对成偶点，得到新图，也即邮路 L_x；再检查 L_x 是否满足以上两个条件；如果违反第一条则一次性删除两条多余重边，如果违反第二条则将 L_x 的该段道路改成与 C 互补的道路；

6.8.6 H 回路的应用：旅行商问题（TSP）

- 问题描述：给定一个正权完全图，求总权最小的 H 回路；

> NP 完全问题，只能寻找近似解；这里不介绍算法，仅介绍问题；提示：不建议使用贪心法，误差很大；

6.8.7 有向无环图、AOV网与拓扑排序

- 有向无环图（DAG）：不存在回路的有向图称为**有向无环图**；
- AOV网：**有向无环图**中的顶点表示活动，边表示活动间的先后关系，这样的图称为**AOV网**；
- 拓扑排序：将AOV网中的活动发生的先后次序排成一个序列（如果有一条从 u 到 v 的道路，那么 v 必须出现在 u 之后），称为**拓扑排序**，这个序列称为**拓扑序列**；

- 拓扑排序实现思路：类似于图的 BFS，但是**只有一个结点的所有直接前驱结点都已访问后，才能访问这个结点**；$O(|V|+|E|)$
  1. 计算每个结点的入度，保存在数组中；
  2. 检查入度数组中**入度为零（无依赖）**的对应结点索引，并将其入队；
  3. 当队伍非空时，循环出队并输出这个结点，在假设将这个结点删除，修正这个结点的所有直接结点的入度（减1），如此重复2、3步骤；

```
// 请自行实现私有函数 int _getInDegree(int) 获取入度；
// 使用到了之前的seqQueue类；
template <class VType, class EType>
int* adjListGraph<VType, EType>::topoSortIdx() const {
    int* ans = new int[this->vertixNum] {0}; int ansIdx = 0;
    int* inDegrees = new int[this->vertixNum] {0};
    seqQueue<int> preRequests;
    for (int i = 0; i < this->vertixNum; ++i) {
        inDegrees[i] = _getInDegree(i);
        if (!inDegrees[i]) preRequests.enQueue(i);
    }
    while (!preRequests.isempty()) {
        int cur = preRequests.deQueue();
        ans[ansIdx++] = cur;
        eNode* curEdge = vertices[cur].edge;
        while (curEdge) {
            if (--inDegrees[curEdge->end] == 0)
                preRequests.enQueue(curEdge->end);
            curEdge = curEdge->next;
        }
    }
    for (int i = 0; i < this->vertixNum; ++i)
        if (inDegrees[i]) {
            std::cout << "[ERROR] A non-DAG does not support topoSort().";
            delete[] ans; return 0;
        }
    return ans;
}

// Usage
template <class VType, class EType>
void adjListGraph<VType, EType>::topoSort() const {
    int* seq = topoSortIdx();
    if (!seq) return;
    for (int i = 0; i < this->vertixNum; ++i)
        std::cout << vertices[seq[i]].data << ' ';
    std::cout << '\n';
    delete[] seq;
}
```

6.8.8 AOE网与关键路径

- AOE网络：活动定义在边上（持续时间），事件定义在顶点上；
- AOE网络的重要两点：源点（入度为0，工程“起点”）、汇点（出度为0，工程“终点”）；
- AOE网络解决的问题：完成整项任务的最少时间、哪些活动是影响工程进度的关键；
- 关键路径：从源点到汇点的**最长**路径称为**关键路径**；
- 关键活动：关键路径上的活动。**推迟关键活动必定影响项目进度**；
- 最早发生时间：用“从源点到该结点的**最长路径**”（因为和拓扑排序一样，只有该结点的所有直接前驱结点都访问过后，才能算访问了这个结点）表征；

- 最迟发生时间：用“关键路径长(定值) - 从汇点到该结点的**最短路径**”表征（因为是最迟，距离汇点最近才符合定义）；
- 时间余量：最迟发生时间 - 最早发生时间。**时间余量为0的活动是关键活动（第二定义）**；
- 找关键路径的思路：（用第二定义）**就是找每个顶点的最早、最迟发生时间，进而得到关键活动、关键路径**；
  1. 找出AOE网的任一拓扑序列；
  2. 从头至尾遍历一次拓扑序列，在遍历到 u 时，更新它的**所有**直接后继结点 v 的最早发生时间（如果当前ee值 < u 的值+路径长，那么更新v的ee值为更大的）；
  3. 再从尾至头遍历一次拓扑序列，在遍历到 u 时，更新它的**所有**直接后继结点 v 的最迟发生时间（如果后继结点le值 < v 的值+路径长，那么更新为u为更小的）；

     > 别问为啥不和第二步相对应，找直接前驱结点，问就是找前驱结点复杂度太大了；
     >
     > ⚠**记得更新最迟发生时间之前，要用第二步得到的关键路径长度（就是拓扑序列最后一个结点的ee值）填充最迟发生时间数组**；

  4. 找出所有“最早发生时间=最迟发生时间”的结点，按照拓扑序列的顺序依此输出，即为关键路径；

```
template <class VType, class EType>
int adjListGraph<VType, EType>::criticalPath(int* early, int* late) const {
    int* topoSeq = topoSortIdx();
    for (int i = 0; i < this->vertixNum; ++i) early[i] = 0;
    for (int i = 0; i < this->vertixNum; ++i) {
        eNode* curEdge = vertices[topoSeq[i]].edge;
        while (curEdge) {
            if (early[topoSeq[i]] + curEdge->weight > early[curEdge->end])
                early[curEdge->end] = early[topoSeq[i]] + curEdge->weight;
            curEdge = curEdge->next;
        }
    }
    int pLen = early[topoSeq[this->vertixNum - 1]];
    for (int i = 0; i < this->vertixNum; ++i) late[i] = len;
    for (int i = this->vertixNum - 1; i >= 0; --i) {
        eNode* curEdge = vertices[topoSeq[i]].edge;
        while (curEdge) {
            if (late[topoSeq[i]] > late[curEdge->end] - curEdge->weight)
                late[topoSeq[i]] = late[curEdge->end] - curEdge->weight;
            curEdge = curEdge->next;
        }
    }
    return pLen;
}

// Usage
template <class VType, class EType>
void adjListGraph<VType, EType>::criticalPath() const {
    int* ee = new int[this->vertixNum];
    int* le = new int[this->vertixNum];
    int pathLen = criticalPath(ee, le);
    std::cout << "[INFO] The length of the critical path: " << pathLen << '\n';
    std::cout << "[INFO] The critical path: \n";
    for (int i = 0; i < this->vertixNum; ++i)
        if (ee[i] == le[i]) std::cout << vertices[i].data << " -> ";
    std::cout << "[Fin]\n";
}
```

6.8.9 生成树的计数算法

> 原理：Binet-Cauchy 定理：两个矩阵 $A_{m\times n}$，$B_{n\times m}$ $(m \le n)$，则 $det(AB)=\sum_i A_iB_i$ 。其中A_i、B_i 分别是从 A 中任取 m 列、B 中任取 m 行构成的行列式；
>
> 虽然这样计算行列式有些麻烦，但它揭示了乘积矩阵行列式和各矩阵的子式之间的关系；
>
> **定理1：（有向连通图的普通生成树计数）设 B_k 为有向连通图 $G=(V,E)$ 的某一基本关联矩阵，则 G 中不同树的数目为** $det(B_kB_k^T)$；
>
> - 解题提示：如果要求**不含**某个边的生成树数目，只要求将该边删去后的生成子图对应生成树的数目；如果要求**必含**某个边的生成树数目，只要该边的起点终点合并为一点，求新图对应生成树的数目；
>
> **如果想求无向连通图的生成树个数，需要将其每条边指定一个任意方向转化为有向连通图**；
>
> - 推论证明：求证**完全图 K_n 的不同生成树的数目为** n^{n-2}；
>
> $$det(B_kB_k^T)=\begin{vmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{vmatrix}=n^{n-2}$$
>
> - ⚠易错警示：如果是求**完全图 K_n 不同构的生成树的数目**，和**“不同生成树”**不一样！和化学上**求同分异构体的做法类似**；例如 K_5 的不同构生成树数目为 3，对应有机化学戊烷的正戊烷、异戊烷、新戊烷的构型；
>
> **定理2：（有向连通图的根树生成树计数）设 $\overrightarrow{B_k}$ 表示将有向连通图 G 的<u>关于结点 k 的</u>关联矩阵 B_k 中所有的 1 元素换成 0 之后的矩阵，则 G 中<u>以 k 为根的</u>不同根树数目为** $det(\overrightarrow{B_k}B_k^T)$；
>
> - 解题提示：如果要求**不含**某个边的根树生成树数目，删去这个边再算；
> - ⚠易错警示：和普通生成树不同，如果要求**必含**某个边的根树生成树数目，需要先计算以 v0 为根的总根树数目，再减去不含这个边的生成树数目；**或者求** $G'=G-\{(t,v)|t\ne u\}$ 的根树生成树数目；

6.8.10 生成树的生成算法

> 不作介绍，有兴趣请查阅相关资料，例如《图论与代数结构》清华大学出版社 第3章 3.5节 支撑树的生成；

6.8.11 最小生成树算法

- Kruskal 算法

  > 思路：不断向初始化为空的根结点中加入当前未加入过的最短边，如果构成回路，一定是回路中的最长边，删除它；如果不构成回路则继续，直至达到 n-1 条边为止，此时 T 一定不含任何回路、n-1条边、包含所有图的顶点、所有权最小，在贪心法上是最小生成树；
  >
  > **如何证明这个贪心算法的正确性?**
  >
  > 可以证明定理：$T=(V,E')$ 是赋权连通图 $G=(V,E)$ 的最短树，当且仅当对任意的余树边 $e\in E-E'$，回路 $C^e(C^e\subseteq E'+e)$ 满足：其边权 $w(e)\ge w(a)$，$a\in C^e$ $(a\ne e)$；

  ```
  T <- Φ  // 树根结点初始化为空
  while (|T| < n - 1 && E(G) != Φ) {
      e <- E中最短边
      E <- E - e
      if (T + e 无回路) T <- T + e
  }
  if (|T| < n - 1) 输出非连通的信息
  else return T
  ```

  时间复杂度：$O(\varepsilon + p\ log\ \varepsilon)$，其中$p$为迭代次数；适用于稀疏图（当p不大时）；

- Prim 算法

> 思路：在结点集中任选一个结点 v0 构成集合 V'，从 V' 和 V-V' 中各选一个顶点 u（来自 V-V'）、v（来自 V'）使得 (u, v) 是满足条件的u、v中最短的边，将此边加入树 T，令 V'+=u，直至 V'=V；
>
> 感兴趣可以找一找定理的正确性证明；

```
t <- v0, T <- Φ, U <- {t}
while (U != V) {
    w(t, u) = min{w(t, v)} where v in (V - U)
    T <- T + e(t, u)
    U <- U + u
    for (v in V - U) w(t, v) <- min{w(t, v), w(u, v)}
}
```

时间复杂度：$O(\nu^2)$；适用于稠密图；

6.9 常见题型和易错点

⚠易错点

- 导航适合使用**有向多重图**表示；
- 简单道路/回路**可以针对非简单图**，意味着可以经过自环、重边，但仅能经过一次；
- 平凡图一定是简单道路、初级道路，一定不是回路；
- 一个点很重要，虽然不知道有啥用：**每个格雷码对应 n-cube Q_n 上的一条 H 回路**；
- n 个结点的**连通的简单平面图**的边数 $m \leq 3n-6$；
- 另一点涉及群论的知识，n 个结点组成的简单无向图的数目为 $2^{n(n-1)/2}$；

  这些图里面互不同构的图的数量又为 $|X/G| = \sum_b \frac{2^k}{\prod(b_i)\prod(c_i!)}$，其中 $k = \sum_{i=1}^{K} \lfloor \frac{b_i}{2} \rfloor + \sum_{i=1}^{K}\sum_{j=1}^{i-1} gcd(b_i, b_j)$，$K$ 为将置换群拆为循环的个数（DFS计数）；

  参考：A000088，计算结论：1，2，4，11，34，156，1044，……

ⓘ常见题型

- 根据图的所有定理，判断/证明一定数量的结点度、结点数、边数等数量间的关系；
- 根据原图指出生成子图、导出子图；
- 给定一个图，判断是否有欧拉道路/回路、哈密顿回路/道路；
- 给定一个图，判断道路/回路、简单道路/回路、初级道路/回路；
- 给定一个图，判断连通分量、强连通分量；
- 给定一个图，找出哈夫曼编码；
- 给定一个图，找出最小生成树；

附录A：部分C++代码实现

A.1 图的存储实现

1. 邻接矩阵存储法

```cpp
template <class VType, class EType>
class adjMatrixGraph {
private:
    VType* vertices;    // store the data of each vertex.
    EType** edges;      // store the data of each edge (in adjacent matrix form).
    EType noEdgeFlag;   // represent the no-edge area.
    int vertixNum;
```

```
    int edgeNum;
    int vertexIdx(const VType& v) const {
        for (int i = 0; i < vertixNum; ++i)
            if (vertices[i] == v) return i;
        throw vertexNotExists();
    }
    void dfs(int start, bool visited[]) const;
public:
    adjMatrixGraph(int vSize, const VType vers[], const EType& noEdge);
    adjMatrixGraph(const adjMatrixGraph<VType, EType>& cp) = delete;
    ~adjMatrixGraph();
    bool exist(const VType& v1, const VType& v2) const;
    void insert(const VType& v1, const VType& v2, const EType& w);
    void remove(const VType& v1, const VType& v2);

    void printAdjMatrix() const;
    void dfs() const;
    void dfs_nonRecur() const;
    void bfs() const;
};
```

2. 邻接表存储法

```
template <class VType, class EType>
class adjListGraph: public graph<VType, EType> {
private:
    struct eNode {
        int end;
        EType weight;
        eNode* next;
        eNode(): end(0), next(0) {}
        eNode(const EType& w, int e=0, eNode* nxt=0)
            : weight(w), end(e), next(nxt) {}
    };
    struct vNode {
        VType data;
        eNode* edge;
    };

    struct EulerNode {
        int nodeIdx;
        EulerNode* next;
        EulerNode(int idx=0, EulerNode* n=0)
            : nodeIdx(idx), next(n) {}
    };

    vNode* vertices;
    bool directed;
    int vertixNum;
    int edgeNum;

    int vertexIdx(const VType& v) const {
        for (int i = 0; i < vertixNum; ++i)
            if (vertices[i].data == v) return i;
        throw vertexNotExists();
    }
    typename adjListGraph<VType, EType>::vNode* cloneBase() const;
    void dfs(int start, bool visited[]) const;

    void _insert(int v1, int v2, const EType& w);
    void _remove(int v1, int v2);
    int _getInDegree(int v) const;
    int _getOutDegree(int v) const;
```

```
    int _getDegree(int v) const;
    void _EulerCircuit(int start, EulerNode*& begin, EulerNode*& end);
    int* topoSortIdx() const;
    int criticalPath(int* early, int* late) const;
    void printPath(int start, int end, int prev[]) const;
public:
    adjListGraph(int vSize, const VType vers[], bool direct=1);
    adjListGraph(const adjListGraph<VType, EType>& cp);
    adjListGraph(adjListGraph<VType, EType>&& tmp);
    ~adjListGraph();
    bool exist(const VType& v1, const VType& v2) const;
    void insert(const VType& v1, const VType& v2, const EType& w);
    void remove(const VType& v1, const VType& v2);

    int getDegree(const VType& v) const;
    int getInDegree(const VType& v) const;
    int getOutDegree(const VType& v) const;

    void printAdjList() const;
    void dfs() const;
    void dfs_nonRecur() const;
    void bfs() const;
    void print_dfs_tree(const VType& emptyFlag) const;
    void print_bfs_tree(const VType& emptyFlag) const;

    bool EulerCircuit(const VType& start);
    void topoSort() const;
    void criticalPath() const;
    // The shortest path for the graph: O(n^3)
    VType* dijkstra(const VType& start, const EType& noEdge, bool prompt=false)
const;
    VType* SPFA(const VType& start, const EType& noEdge) const;
};
```

A.2 图的运算实现

1. 邻接矩阵表示

```
template <class VType, class EType>
adjMatrixGraph<VType, EType>::adjMatrixGraph(
    int vSize, const VType vers[], const EType& noEdge) {
    this->vertixNum = vSize; this->edgeNum = 0; noEdgeFlag = noEdge;
    vertices = new VType[vSize];
    edges = new EType*[vSize];
    for (int i = 0; i < vSize; ++i) {
        vertices[i] = vers[i];
        edges[i] = new EType[vSize];
        for (int j = 0; j < vSize; ++j)
            edges[i][j] = noEdge;
        edges[i][i] = 0;
    }
}

template <class VType, class EType>
adjMatrixGraph<VType, EType>::~adjMatrixGraph() {
    delete[] vertices;
    for (int i = 0; i < this->vertixNum; ++i)
        delete[] edges[i];
    delete[] edges;
}

```

```
template <class VType, class EType>
bool adjMatrixGraph<VType, EType>::exist(const VType& v1, const VType& v2) const
{
    int u = vertexIdx(v1), v = vertexIdx(v2);
    return edges[u][v] != noEdgeFlag;
}

template <class VType, class EType>
void adjMatrixGraph<VType, EType>::insert(const VType& v1, const VType& v2, const
EType& w) {
    int u = vertexIdx(v1), v = vertexIdx(v2);
    if (edges[u][v] == noEdgeFlag) ++this->edgeNum;
    edges[u][v] = w;
}

template <class VType, class EType>
void adjMatrixGraph<VType, EType>::undirected_insert(const VType& v1, const
VType& v2, const EType& w) {
    int u = vertexIdx(v1), v = vertexIdx(v2);
    if (edges[u][v] == noEdgeFlag) ++this->edgeNum;
    if (edges[v][u] == noEdgeFlag) ++this->edgeNum;
    edges[u][v] = edges[v][u] = w;
}

template <class VType, class EType>
void adjMatrixGraph<VType, EType>::remove(const VType& v1, const VType& v2) {
    int u = vertexIdx(v1), v = vertexIdx(v2);
    if (edges[u][v] != noEdgeFlag) --this->edgeNum;
    edges[u][v] = noEdgeFlag;
}

template <class VType, class EType>
void adjMatrixGraph<VType, EType>::printAdjMatrix() const {
    for (int i = 0; i < this->vertixNum; ++i) {
        for (int j = 0; j < this->vertixNum; ++j)
            std::cout << edges[i][j] << ' ';
        std::cout << '\n';
    }
}
```

2. 邻接表表示

```
template <class VType, class EType>
adjListGraph<VType, EType>::adjListGraph(int vSize, const VType vers[], bool
direct) {
    this->vertixNum = vSize; directed = direct;
    this->edgeNum = 0; vertices = new vNode[vSize];
    for (int i = 0; i < vSize; ++i) {
        vertices[i].data = vers[i];
        vertices[i].edge = nullptr;
    }
}

template <class VType, class EType>
adjListGraph<VType, EType>::~adjListGraph() {
    eNode* curEdge;
    for (int i = 0; i < this->vertixNum; ++i) {
        while (curEdge = vertices[i].edge) {
            vertices[i].edge = curEdge->next;
            delete curEdge;
        }
```

```cpp
    }
    if (vertices) delete[] vertices;
}

template <class VType, class EType>
typename adjListGraph<VType, EType>::vNode* adjListGraph<VType, EType>::cloneBase() const {
    vNode* newVers = new vNode[this->vertixNum];
    for (int i = 0; i < this->vertixNum; ++i) {
        newVers[i].data = vertices[i].data;
        newVers[i].edge = nullptr;
        eNode** curEdgeDst = &(newVers[i].edge);
        eNode* curEdgeSrc = vertices[i].edge;
        while (curEdgeSrc) {
            *curEdgeDst = new eNode(curEdgeSrc->weight, curEdgeSrc->end, 0);
            curEdgeDst = &((*curEdgeDst)->next);
            curEdgeSrc = curEdgeSrc->next;
        }
    }
    return newVers;
}

template <class VType, class EType>
adjListGraph<VType, EType>::adjListGraph(const adjListGraph<VType, EType>& cp) {
    vertices = cp.cloneBase(); this->edgeNum = cp.edgeNum;
    this->vertixNum = cp.vertixNum; directed = cp.directed;
}

template <class VType, class EType>
adjListGraph<VType, EType>::adjListGraph(adjListGraph<VType, EType>&& tmp) {
    this->edgeNum = tmp.edgeNum; this->vertixNum = tmp.vertixNum; directed = tmp.directed;
    tmp.edgeNum = tmp.vertixNum = 0; vertices = tmp.vertices; tmp.vertices = nullptr;
}

template <class VType, class EType>
bool adjListGraph<VType, EType>::exist(const VType& v1, const VType& v2) const {
    int u = vertexIdx(v1), v = vertexIdx(v2);
    const eNode* cur = vertices[u].edge;
    while (cur) {
        if (cur->end == v) return true;
        cur = cur->next;
    }
    return false;
}

template <class VType, class EType>
void adjListGraph<VType, EType>::_insert(int v1, int v2, const EType& w) {
    eNode** cur = &(vertices[v1].edge);
    while (*cur && (*cur)->end != v2) cur = &((*cur)->next);
    if (!(*cur)) { ++this->edgeNum; *cur = new eNode(w, v2); }
}

template <class VType, class EType>
void adjListGraph<VType, EType>::_remove(int v1, int v2) {
    eNode** cur = &(vertices[v1].edge);
    while (*cur && (*cur)->end != v2) cur = &((*cur)->next);
    if (*cur) {
        eNode* tmp = *cur; *cur = (*cur)->next;
        delete tmp; --this->edgeNum;
    }
}
```

```
template <class VType, class EType>
void adjListGraph<VType, EType>::insert(const VType& v1, const VType& v2, const EType& w) {
    int u = vertexIdx(v1), v = vertexIdx(v2);
    _insert(u, v, w);
    if (!directed) _insert(v, u, w);
}

template <class VType, class EType>
void adjListGraph<VType, EType>::remove(const VType& v1, const VType& v2) {
    int u = vertexIdx(v1), v = vertexIdx(v2);
    _remove(u, v);
    if (!directed) _remove(v, u);
}

template <class VType, class EType>
int adjListGraph<VType, EType>::_getInDegree(int v) const {
    int ans = 0;
    for (int i = 0; i < this->vertixNum; ++i) {
        if (i == v) continue;
        eNode* curEdge = vertices[i].edge;
        while (curEdge) {
            if (curEdge->end == v) ++ans;
            curEdge = curEdge->next;
        }
    }
    return ans;
}

template <class VType, class EType>
int adjListGraph<VType, EType>::_getOutDegree(int v) const {
    int ans = 0;
    eNode* target = vertices[v].edge;
    while (target) { ++ans; target = target->next; }
    return ans;
}

template <class VType, class EType>
int adjListGraph<VType, EType>::_getDegree(int v) const {
    if (directed) return _getInDegree(v) + _getOutDegree(v);
    else return _getOutDegree(v);
}

template <class VType, class EType>
int adjListGraph<VType, EType>::getDegree(const VType& v) const {
    return _getDegree(vertexIdx(v));
}

template <class VType, class EType>
int adjListGraph<VType, EType>::getInDegree(const VType& v) const {
    return _getInDegree(vertexIdx(v));
}

template <class VType, class EType>
int adjListGraph<VType, EType>::getOutDegree(const VType& v) const {
    return _getOutDegree(vertexIdx(v));
}

template <class VType, class EType>
void adjListGraph<VType, EType>::printAdjList() const {
    for (int i = 0; i < this->vertixNum; ++i) {
        std::cout << "(" << i << ") " << vertices[i].data << ' ';
```

```
        const eNode* cur = vertices[i].edge;
        while (cur) {
            std::cout << "|-w=" << cur->weight
                << "->(" << cur->end << ") ";
            cur = cur->next;
        }
        std::cout << '\n';
    }
}
```

A.3 图的遍历算法

1. DFS：邻接表表示

```
template <class VType, class EType>
void adjListGraph<VType, EType>::dfs(int start, bool visited[]) const {
    eNode* curEdge = vertices[start].edge;
    std::cout << vertices[start].data << ' ';
    visited[start] = 1;
    while (curEdge) {
        if (!visited[curEdge->end]) dfs(curEdge->end, visited);
        curEdge = curEdge->next;
    }
}

template <class VType, class EType>
void adjListGraph<VType, EType>::dfs() const {
    bool* visited = new bool[this->vertixNum] {0};
    for (int i = 0; i < this->vertixNum; ++i) {
        if (visited[i]) continue;
        dfs(i, visited);
        std::cout << '\n';
    }
}

template <class VType, class EType>
void adjListGraph<VType, EType>::dfs_nonRecur() const {
    seqStack<int> tasks;
    bool* visited = new bool[this->vertixNum] {0};
    for (int i = 0; i < this->vertixNum; ++i) {
        if (visited[i]) continue;
        tasks.push(i);
        while (!tasks.isempty()) {
            int tmp = tasks.pop();
            if (visited[tmp]) continue;     // Necessary when doing non-recursive op.
            std::cout << vertices[tmp].data << ' ';
            visited[tmp] = 1;
            eNode* curEdge = vertices[tmp].edge;
            while (curEdge) {
                if (!visited[curEdge->end])
                    tasks.push(curEdge->end);
                curEdge = curEdge->next;
            }
        }
        std::cout << '\n';
    }
    delete[] visited;
}
```

2. DFS：邻接矩阵表示

```cpp
template <class VType, class EType>
void adjMatrixGraph<VType, EType>::dfs(int start, bool visited[]) const {
    std::cout << vertices[start] << ' ';
    visited[start] = 1;
    for (int i = start + 1; i < this->vertixNum; ++i) {
        if (!visited[i] && edges[start][i] != noEdgeFlag)
            dfs(i, visited);
    }
}

template <class VType, class EType>
void adjMatrixGraph<VType, EType>::dfs() const {
    bool* visited = new bool[this->vertixNum] {0};
    for (int i = 0; i < this->vertixNum; ++i) {
        if (visited[i]) continue;
        dfs(i, visited);
        std::cout << '\n';
    }
    delete[] visited;
}

template <class VType, class EType>
void adjMatrixGraph<VType, EType>::dfs_nonRecur() const {
    seqStack<int> tasks;
    bool* visited = new bool[this->vertixNum] {0};
    for (int i = 0; i < this->vertixNum; ++i) {
        if (visited[i]) continue;
        tasks.push(i);
        while (!tasks.isempty()) {
            int tmp = tasks.pop();
            if (visited[tmp]) continue;
            std::cout << vertices[tmp] << ' ';
            visited[tmp] = 1;
            for (int j = tmp + 1; j < this->vertixNum; ++j) {
                if (!visited[j] && edges[tmp][j] != noEdgeFlag)
                    tasks.push(j);
            }
        }
        std::cout << '\n';
    }
    delete[] visited;
}
```

3. BFS：邻接表表示

```cpp
template <class VType, class EType>
void adjListGraph<VType, EType>::bfs() const {
    seqQueue<int> taskQ;
    bool* visited = new bool[this->vertixNum] {0};
    for (int i = 0; i < this->vertixNum; ++i) {
        if (visited[i]) continue;
        taskQ.enQueue(i);
        while (!taskQ.isempty()) {
            int tmp = taskQ.deQueue();
            if (visited[tmp]) continue;     // necessary.
            std::cout << vertices[tmp].data << ' ';
            visited[tmp] = 1;

```

```
            eNode* curEdge = vertices[tmp].edge;
            while (curEdge) {
                if (!visited[curEdge->end])
                    taskQ.enQueue(curEdge->end);
                curEdge = curEdge->next;
            }
        }
    }
    delete[] visited;
}
```

4. BFS：邻接矩阵表示

```
template <class VType, class EType>
void adjMatrixGraph<VType, EType>::bfs() const {
    seqQueue<int> taskQ;
    bool* visited = new bool[this->vertixNum] {0};
    for (int i = 0; i < this->vertixNum; ++i) {
        if (visited[i]) continue;
        taskQ.enQueue(i);
        while (!taskQ.isempty()) {
            int tmp = taskQ.deQueue();
            if (visited[tmp]) continue;     // necessary when doing no-recursive op.
            std::cout << vertices[tmp] << ' ';
            visited[tmp] = 1;
            for (int j = tmp + 1; j < this->vertixNum; ++j) {
                if (!visited[j] && edges[tmp][j] != noEdgeFlag)
                    taskQ.enQueue(j);
            }
        }
        std::cout << '\n';
    }

    delete[] visited;
}
```