第一板块：算法基础与效率分析

1. 算法概论 (LEC1)

- **什么是算法 (Algorithm)?**
  - 为了求解问题而给出的一系列明确的、可执行的指令序列 。
  - 非正式定义：任何定义明确的计算过程，接收一些值或集合作为输入，并产生一些值或集合作为输出 。
  - **简答题考点**：算法的定义。
  - **注意**：程序只是算法的一种实现方式 。算法可以用自然语言、伪代码、程序设计语言甚至硬件来描述 。
- **算法的五个重要特性**：
  - **正确性 (Correctness)**：对于任何合法的输入，都能得到正确的输出。
    - 例子：选票统计中，方法2（分组计票）不一定完全正确，但可能获得很高的正确概率 。
  - **有效性 (Effectiveness/Well-definedness)**：算法的每一步都能被有效执行，并得到预期的结果。
  - **确定性 (Determinism)**：算法的每一步执行后，都有确定的下一步指令。
  - **有穷性 (Finiteness)**：算法在执行有限步骤后必须终止。
  - **简答题考点**：列举并解释算法的特性。
- **解决问题的模式**：输入 (Input) -> 算法 (Algorithm) -> 输出 (Output) 。
  - 算法在解决问题层面上具有“黑盒”特征 。
- **简单问题 vs. 复杂问题**：
  - 一些问题直观上简单（如10以内加法），但其背后实现的算法（如加法器硬件设计）可能复杂 。
  - 一些问题看似复杂（如在DNA序列中寻找匹配基因），但可以通过高效算法解决 。

2. 算法效率分析基础 (LEC1)

- **为什么要分析效率?**
  - 衡量算法好坏的重要标准。
  - 有效地利用计算资源（如时间、空间） 。
- **计算资源**：主要是指算法执行所需的**时间 (Time)** 和 **空间 (Space)** 。
- **时空资源折衷原理**：对于同一问题，通常存在多种算法。为了改善时间开销，往往可以牺牲空间开销；反之亦然 。
  - **简答题考点**：解释时空折衷。
- **资源开销与输入的关系**：
  - 通常与**输入规模 (Input Size)** 相关，记为 n。
  - 也可能与输入的具体**组织结构 (Input Structure)** 有关。
  - 我们关心的是随输入规模变化的资源开销函数 $T(n)$ 。
- **渐进分析 (Asymptotic Analysis)**：
  - **目的**：评估算法在输入规模 n 趋向无穷大时的行为，从而比较算法的效率 。
  - **关注点**：运行时间或存储需求的增长率/阶 (Order of Growth) 。
  - **优点**：
    - 忽略常数因子和低阶项，使得分析更简洁，关注核心效率。
    - 独立于具体的机器和实现技术，是对算法本身的度量。

    - 主要关心大规模输入的情况。
  - **RAM模型假设**：一种简化的计算模型，假设基本操作（加、减、乘、除、比较、访存等）花费常数时间。
- **渐进记号 (Asymptotic Notations)** (LEC1, pp. 47-51,)：
  - **O (大O符号，上界)**：$f(n) = O(g(n))$ 表示当 n 足够大时，$f(n)$ 的增长速度**不快于** $g(n)$ 的常数倍。即存在正常数 c 和 n_0，使得对所有 $n \geq n_0$，有 $0 \leq f(n) \leq c \cdot g(n)$。
    - 常用于表示算法的**最坏情况**运行时间。
  - **Ω (大欧米伽符号，下界)**：$f(n) = \Omega(g(n))$ 表示当 n 足够大时，$f(n)$ 的增长速度**不慢于** $g(n)$ 的常数倍。即存在正常数 c 和 n_0，使得对所有 $n \geq n_0$，有 $0 \leq c \cdot g(n) \leq f(n)$。
    - 常用于表示算法的**最好情况**运行时间，或问题的**难度下界**。
  - **Θ (大西塔符号，紧确界)**：$f(n) = \Theta(g(n))$ 表示当 n 足够大时，$f(n)$ 的增长速度**与** $g(n)$ 的常数倍**相同**。即 $f(n) = O(g(n))$ 且 $f(n) = \Omega(g(n))$。存在正常数 c_1, c_2 和 n_0，使得对所有 $n \geq n_0$，有 $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$。
    - 给出了算法运行时间的一个精确的阶。
  - **o (小o符号，非紧确上界)**：$f(n) = o(g(n))$ 表示当 n 足够大时，$f(n)$ 的增长速度**远快于** $g(n)$ 的任何常数倍是不可能的，即 $f(n)$ 相对于 $g(n)$ 是渐进可忽略的。对任意正常数 c，存在 n_0，使得对所有 $n \geq n_0$，有 $0 \leq f(n) < c \cdot g(n)$。等价于 $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$。
  - **ω (小欧米伽符号，非紧确下界)**：$f(n) = \omega(g(n))$ 表示当 n 足够大时，$f(n)$ 的增长速度**远慢于** $g(n)$ 的任何常数倍是不可能的。对任意正常数 c，存在 n_0，使得对所有 $n \geq n_0$，有 $0 \leq c \cdot g(n) < f(n)$。等价于 $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ (PPT中未直接给出极限形式，但与o符号对应)。
  - **简答题考点**：解释O, Ω, Θ的定义，并能根据定义判断函数关系。
  - **分析题基础**：能够用这些符号表达算法的复杂度。
- **常见函数的增长阶** (从慢到快)：
  - $O(1)$ (常数)
  - $O(\log n)$ (对数)
  - $O(n)$ (线性)
  - $O(n \log n)$ (线性对数)
  - $O(n^2)$ (平方)
  - $O(n^c), c > 1$ (多项式)
  - $O(c^n), c > 1$ (指数)
  - $O(n!)$ (阶乘)
- **渐进记号的性质**：
  - 自反性：$f(n) = O(f(n)), f(n) = \Omega(f(n)), f(n) = \Theta(f(n))$。
  - 传递性：若 $f(n) = O(g(n))$ 且 $g(n) = O(h(n))$, 则 $f(n) = O(h(n))$ (Ω, Θ类似)。
  - 对称性 (Θ独有，O和Ω不完全对称)：$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$。
  - 转换关系：$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$；$f(n) = o(g(n)) \iff g(n) = \omega(f(n))$。
  - 和的关系：若 $f_1(n) = O(g_1(n))$ 且 $f_2(n) = O(g_2(n))$, 则 $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$，通常简化为 $O(\max(g_1(n), g_2(n)))$。
    - $f(n) = O(g(n)) \Rightarrow f(n) + g(n) = O(g(n))$ (应为 $\Theta(g(n))$ 或 $O(g(n))$ 取决于 $f(n)$的具体情况，PPT中写的是 $O(g(n))$，更准确地说是 $f(n) + g(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $g(n)$ dominates $f(n)$ or $f(n) = \Theta(g(n))$; if $f(n) = o(g(n))$, then $f(n) + g(n) = \Theta(g(n))$)。
- **最好、最坏和平均情况分析 (Best, Worst, Average Case Analysis)**：

- **最坏情况 (Worst Case)**：算法运行时间最长的情况。通常是我们最关心的，因为它给出了运行时间的一个上界。
- **最好情况 (Best Case)**：算法运行时间最短的情况。意义不大，除非最好情况经常发生。
- **平均情况 (Average Case)**：算法在所有可能输入（按某种概率分布）下的期望运行时间。分析通常比最坏情况复杂，且需要对输入分布做假设 。
- PPT指出，最坏情况通常比较接近于平均情况 。
- **简答题考点**：解释三种情况分析的含义及最坏情况分析的优点（简单、独立于输入假设、通常接近平均情况） 。
- **例 (LEC1, p.27,)**：插入排序
  - 最好情况 (输入已排序)：$T(n) = an + b = O(n)$
  - 最坏情况 (输入逆序)：$T(n) = an^2 + bn + c = O(n^2)$

3. 递归关系求解 (LEC2)

- **递归关系式 (Recurrence Relation)**：一个等式或不等式，它通过函数在更小输入上的值来描述该函数 。许多分治算法的运行时间可以用递归关系式表示 。
- **求解递归关系式的方法**：
  - **置换法 (Substitution Method)**：
    a. **猜想**解的形式。
    b. 用数学归纳法**证明**猜想的正确性（包括验证边界条件和归纳步骤）。
    - **如何获得好的猜想**：经验、画递归树、尝试更换变元、有时需要先证明一个较松的界然后收紧它。
    - **例**：$T(n) = 2T(\lfloor n/2 \rfloor) + n$。猜想 $T(n) = O(n \log n)$。
      证明：假设 $T(k) \le ck \log k$ 对 $k < n$ 成立。
      $T(n) \le 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n$
      $\le cn \log(n/2) + n = cn \log n - cn \log 2 + n = cn \log n - cn + n.$
      要使 $cn \log n - cn + n \le cn \log n$，需要 $-cn + n \le 0$，即 $c \ge 1$ 。
      需要处理边界条件，并可能需要调整常数或低阶项以使归纳成立。
    - **更换变元技巧**：将复杂的递归式转换为已知的形式。
      例：$T(n) = 2T(n^{1/2}) + \log n$。
      令 $m = \log n$，则 $n = 2^m$，$n^{1/2} = 2^{m/2}$。
      令 $S(m) = T(2^m)$。则 $S(m) = 2S(m/2) + m$。
      解得 $S(m) = O(m \log m)$。
      换回 $T(n)$：$T(n) = O(\log n \log(\log n))$ 。
    - **分析题考点**：使用置换法证明递归式的解。
  - **递归树法 (Recursion Tree Method)**：
    a. 将递归式展开成一棵树，树的每个节点代表该次递归调用的代价。
    b. 将每层的代价加起来得到每层的总代价。
    c. 将所有层的代价加起来得到总代价。
    - 通常用于**启发猜想**，然后用置换法证明。
    - **例**：$T(n) = T(n/3) + T(2n/3) + cn$ 。
      树的高度不规则，但最短路径到叶子是 $\log_{3/2} n$，最长路径是 $\log_3 n$。
      每层代价都是 cn (或接近 cn)。总代价大约是 $cn \times (\text{高度})$。
      此例的解为 $O(n \log n)$ 。

- **分析题考点**：使用递归树法推导递归式的解。

- **迭代法 (Iteration Method)**：
  a. 不断将递归式展开，直到呈现出某种求和模式。
  b. 利用代数运算和求和公式计算出结果。
  - **例1**：$s(n) = c + s(n-1), s(0) = 0$。
    $s(n) = c + c + s(n-2) = \cdots = kc + s(n-k)$。令 $k = n, s(n) = cn + s(0) = cn = O(n)$。
  - **例2**：$s(n) = n + s(n-1), s(0) = 0$。
    $s(n) = n + (n-1) + s(n-2) = \cdots = n + (n-1) + \cdots + (n-k+1) + s(n-k)$。
    令 $k = n, s(n) = \sum_{i=1}^{n} i + s(0) = n(n+1)/2 = O(n^2)$。
  - **例3**：$T(n) = 2T(n/2) + c, T(1) = c$ (假设 $n = 2^k$)。
    $T(n) = 2(2T(n/4) + c) + c = 2^2T(n/2^2) + 2c + c$
    $= 2^kT(n/2^k) + (2^{k-1} + \cdots + 2^1 + 2^0)c$
    $= 2^kT(1) + (2^k - 1)c = n \cdot c + (n-1)c = (2n-1)c = O(n)$。
  - **例4**：$T(n) = aT(n/b) + cn, T(1) = c$ (假设 $n = b^k$)。
    展开得到 $T(n) = a^kT(1) + cn\sum_{j=0}^{k-1}(a/b)^j$
    $= ca^k + cn\sum_{j=0}^{k-1}(a/b)^j$
    $= cn^{\log_b a} + cn\sum_{j=0}^{k-1}(a/b)^j$ (因为 $a^k = a^{\log_b n} = n^{\log_b a}$)。
    然后分三种情况讨论求和项：
    a. $a < b \Rightarrow a/b < 1$: $\sum(a/b)^j$ 是收敛的几何级数，为 $O(1)$。$T(n) = \Theta(n^{\log_b a} + n) = \Theta(n)$ (因为 $\log_b a < 1$)。
    b. $a = b \Rightarrow a/b = 1$: $\sum 1 = k = \log_b n$。$T(n) = \Theta(n + n\log_b n) = \Theta(n \log n)$。
    c. $a > b \Rightarrow a/b > 1$: $\sum(a/b)^j = \frac{(a/b)^k - 1}{(a/b) - 1} = \Theta((a/b)^k)$。$T(n) = \Theta(n^{\log_b a} + cn \cdot (a/b)^{\log_b n}) = \Theta(n^{\log_b a} + cn \cdot a^k/b^k) = \Theta(n^{\log_b a} + cn \cdot n^{\log_b a}/n) = \Theta(n^{\log_b a})$。
  - **分析题考点**：使用迭代法求解递归式。

- **主方法 (Master Method)**：
  - 提供了一种“菜谱式”的方法来求解形式为 $T(n) = aT(n/b) + f(n)$ 的递归式，其中 $a \geq 1, b > 1$ 是常数，$f(n)$ 是一个渐进正函数。
  - 需要比较 $f(n)$ 与 $n^{\log_b a}$ 的大小。
  - **三种情况**：
    a. 若 $f(n) = O(n^{\log_b a - \epsilon})$ 对某个常数 $\epsilon > 0$ 成立，则 $T(n) = \Theta(n^{\log_b a})$。
    ($f(n)$ 被 $n^{\log_b a}$ 多项式地压制)
    b. 若 $f(n) = \Theta(n^{\log_b a})$，则 $T(n) = \Theta(n^{\log_b a} \log n)$。
    ($f(n)$ 与 $n^{\log_b a}$ 阶数相同)
    c. 若 $f(n) = \Omega(n^{\log_b a + \epsilon})$ 对某个常数 $\epsilon > 0$ 成立，并且对某个常数 $c < 1$ 和所有足够大的 n，有 $af(n/b) \leq cf(n)$ (正则性条件)，则 $T(n) = \Theta(f(n))$。
    ($f(n)$ 多项式地支配 $n^{\log_b a}$，且满足正则性条件)
  - **例**：$T(n) = 9T(n/3) + n$。
    $a = 9, b = 3, f(n) = n$。
    $n^{\log_b a} = n^{\log_3 9} = n^2$。
    比较 $f(n) = n$ 和 n^2。

$n = O(n^{2-1}) = O(n^{\log_3 9-\epsilon})$，其中 $\epsilon = 1$。
符合情况1，所以 $T(n) = \Theta(n^2)$ 。
- **分析题考点**：使用主方法快速判断递归式的复杂度。注意主方法不适用于所有形式的递归式。

4. 算法效率比较与改进初步 (LEC3)

- **效率比较**：如果算法A的运行时间 $T_A(n) = o(T_B(n))$，则算法A优于算法B 。
  - 例如，对于排序问题，插入排序是 $O(n^2)$，而合并排序是 $O(n \log n)$。因为 $n \log n = o(n^2)$，所以合并排序在大规模输入下通常优于插入排序 。

本板块小结及备考提示：

- **简答题**：
  - 算法的定义及五个特性。
  - 为什么要进行算法效率分析？什么是时空折衷？
  - 渐进记号 O, Ω, Θ 的定义和含义。
  - 最好、最坏、平均情况分析的含义。
  - 主方法的三个情况描述（可能不会直接考，但理解很重要）。
- **算法分析题**：
  - **核心内容**：熟练使用置换法、递归树法、迭代法求解递归关系式的时间复杂度。
  - 主方法是快速判断的工具，但对于不能直接套用主方法或需要严格证明的情况，前三种方法是基础。
  - 能够分析给定代码片段的时间复杂度（如 LEC1 中的 `matrix_addition` 和 `Insertion Sort` 的分析过程）。
- **设计证明题**：
  - 本板块主要侧重分析，但理解渐进记号的严格定义对于后续证明算法性质（如贪心算法的正确性，动态规划的最优子结构等）至关重要。
  - 置换法中的归纳证明是一种重要的证明技巧。

建议：务必掌握各种渐进记号的精确定义，并多练习不同类型的递归式的求解。理解递归树如何帮助建立直观认识，并用置换法进行严格证明。

排序算法详解之一：插入排序 (Insertion Sort) (LEC3)

1. 基本思想

插入排序是一种简单直观的排序算法。它的工作方式类似于我们平时打扑克牌时整理手中的牌：

1. **分区**：将待排序的数组在逻辑上分为两个部分：“已排序区” 和 “未排序区”。
2. **初始状态**：在开始时，已排序区仅包含数组的第一个元素（或者可以视为空，从第一个元素开始取）。
3. **迭代插入**：依次从未排序区中取出第一个元素。
4. **比较与移动**：将这个取出的元素（称为“当前元素”或“关键字”）与已排序区中的元素从后向前逐个比较。

- 如果已排序区的元素大于当前元素，则将该元素向后移动一位，为当前元素腾出空间。
- 重复此过程，直到找到一个小于或等于当前元素的已排序区元素，或者已到达已排序区的最前端。

5. **插入**：将当前元素插入到找到的合适位置。
6. **重复**：重复步骤3-5，直到未排序区为空，整个数组排序完成。

可以想象成，每次从未排序部分拿起一张牌，然后在手中已排好序的牌中找到合适位置插入。

2. 伪代码及解释 (LEC3, p.1)

以下是PPT中提供的插入排序伪代码，通常用于对数组 `A` 的 `n` 个元素进行排序。假设数组下标从1开始， `A[1]` 是第一个元素。

```
IS(A, n) { // A是要排序的数组，n是元素个数
    // 外层循环控制从未排序区取元素
    // i 指向当前从未排序区取出的元素，从第二个元素开始 (A[2])
    // A[1...i-1] 是已排序区，A[i...n] 是未排序区
    for i = 2 to n {
        key = A[i]      // key 是当前需要插入到已排序区的元素

        // 内层循环负责在已排序区 A[1...i-1] 中为 key 找到正确的位置
        // j 指向已排序区的最后一个元素 A[i-1]
        j = i - 1

        // 将已排序区中大于 key 的元素向后移动
        // 条件：j > 0 (防止数组越界到A[0]) 且 A[j] > key
        while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]   // 将A[j]向后移动一位
            j = j - 1       // 继续向前比较
        }
        // 循环结束时，j 指向的位置是 key 应该插入位置的前一个，或者 j=0
        // 所以 key 应该插入到 A[j+1]
        A[j+1] = key
    }
}
```

逐步解释 `PARTITION` 逻辑 (以 `A[r]` 为主元)：

假设数组为 `A` ，要划分的范围是 `p` 到 `r` 。

1. `x = A[r]` ： 选择子数组的最后一个元素作为主元 `x` 。
2. `i = p - 1` ： `i` 用来标记“小于等于主元”区域的右边界。初始时，这个区域为空。
3. `for j = p to r - 1` ： `j` 是当前扫描的元素指针，从子数组的第一个元素开始，到主元的前一个元素结束。
   - `if A[j] <= x` ： 如果当前元素 `A[j]` 小于或等于主元 `x` ：
     - `i = i + 1` ： 扩展“小于等于主元”区域的边界。
     - `exchange A[i] with A[j]` ： 将 `A[j]` 交换到“小于等于主元”区域的末尾。

4. `exchange A[i+1] with A[r]`：循环结束后，`A[p..i]` 中的所有元素都小于等于 `x`，`A[i+1..j-1]`（在循环中j已经到了r）中的元素都大于 `x`。此时将主元 `A[r]` 与 `A[i+1]` 交换，主元就放到了其最终排序后的正确位置。
5. `return i + 1`：返回主元的新下标。

例子 (LEC3, p.3 的图示解释):
假设 `A = [2, 8, 7, 1, 3, 5, 6, 4]`， `p=0, r=7` (假设下标从0开始)，主元 `x = A[7] = 4`。

- `i = -1`
- `j=0, A[0]=2 <= 4`：`i=0`，`swap(A[0], A[0]) -> A=[2, 8, 7, 1, 3, 5, 6, 4]`
- `j=1, A[1]=8 > 4`
- `j=2, A[2]=7 > 4`
- `j=3, A[3]=1 <= 4`：`i=1`，`swap(A[1], A[3]) -> A=[2, 1, 7, 8, 3, 5, 6, 4]`
- `j=4, A[4]=3 <= 4`：`i=2`，`swap(A[2], A[4]) -> A=[2, 1, 3, 8, 7, 5, 6, 4]`
- `j=5, A[5]=5 > 4`
- `j=6, A[6]=6 > 4`
- 循环结束。`swap(A[i+1], A[r])` 即 `swap(A[3], A[7]) -> A=[2, 1, 3, 4, 7, 5, 6, 8]`。
- 返回 `i+1 = 3`。此时 `A[3]=4`，左边元素都 ≤ 4，右边元素都 > 4。

3. 效率分析

- **时间复杂度**:
  - **最坏情况 (Worst Case)**：当输入数组已经排序（升序或降序）或主元选择策略导致每次划分都极不平衡时（例如，每次都选到最小或最大元素作为主元）。
    - 此时，`PARTITION` 后一个子问题大小为 0，另一个为 $n-1$。
    - 递归式为：$T(n) = T(n-1) + \Theta(n)$ （$\Theta(n)$ 是 `PARTITION` 的开销）。
    - 解得 $T(n) = \Theta(n^2)$。
  - **最好情况 (Best Case)**：当每次 `PARTITION` 都恰好将问题划分为大小几乎相等的两个子问题（即主元是中位数）。
    - 递归式为：$T(n) = 2T(n/2) + \Theta(n)$。
    - 根据主方法，解得 $T(n) = \Theta(n \log n)$。
  - **平均情况 (Average Case)**：可以证明，对于随机输入（或使用随机化主元选择），快速排序的平均时间复杂度为 $\Theta(n \log n)$。PPT (LEC3, p.4) 中提到 "平均情况接近最好情况"。
- **空间复杂度**:
  - 取决于递归调用的栈深度。
  - **最好情况**（完美平衡划分）：递归深度为 $O(\log n)$。
  - **最坏情况**（极不平衡划分）：递归深度为 $O(n)$。
  - 通过一些优化（如对较短的子数组优先递归），可以保证最坏情况下的空间复杂度为 $O(\log n)$，但这指的是尾递归优化后的情况或特定实现。一般教科书分析的原地版本最坏栈空间是 $O(n)$。

4. 稳定性

- 快速排序的 `PARTITION` 操作通常是**不稳定**的。在交换元素时，相等元素的相对顺序可能会改变。

- 例如，对于 `A = [3a, 1, 2, 3b]` （ `3a` 和 `3b` 值相等但为了区分标记了 `a` 和 `b` ），若以第一个 `3a` 为主元， `3b` 可能会被交换到 `3a` 的前面。

5. 备考要点

- **简答题考点**:
  - 快速排序的基本思想（分治、主元、划分）。
  - 解释快速排序最坏情况发生的原因及其时间复杂度。
  - 解释快速排序最好情况发生的原因及其时间复杂度。
  - 快速排序是否稳定？为什么？
  - 相比合并排序，快速排序的优点（通常平均性能更好，原地性更好）和缺点（最坏情况性能差，不稳定）。
- **算法分析题考点**:
  - 分析给定 `PARTITION` 伪代码的运行时间。
  - 写出快速排序在最坏/最好情况下的时间复杂度递归式并求解。
  - 可能会结合随机化快速排序（LEC5）考察期望时间复杂度（虽然这部分更侧重于LEC5，但快速排序本身是基础）。
- **设计证明题考点**:
  - 可能会要求你描述 `PARTITION` 过程或 `QUICKSORT` 算法。
  - 证明 `PARTITION` 算法的正确性（即循环不变量）。

6. 总结

快速排序是一种非常高效的排序算法，平均性能优越，是实践中应用最广泛的排序算法之一。其核心在于 `PARTITION` 操作。理解主元的选择对性能的影响以及如何通过随机化来避免最坏情况是非常重要的。

排序算法详解之二：合并排序 (MergeSort) (LEC3)

1. 基本思想 (分治策略)

合并排序是分治策略的一个经典应用 。其核心思想是将一个大问题分解成小问题分别解决，然后将小问题的解合并起来得到大问题的解 。

1. **分解 (Divide)**：将包含 n 个元素的待排序序列平均分成两个各含 $n/2$ 个元素的子序列 。如果序列只有一个元素或为空，则认为它已经有序，无需再分解 。
2. **解决 (Conquer)**：递归地调用合并排序算法，对这两个子序列分别进行排序 。
3. **合并 (Combine)**：将两个已经排好序的子序列合并成一个单一的、完整的有序序列 。这个合并步骤是算法的关键。

可以想象成，要整理一堆杂乱的卡片，你先把它们分成两堆，再把每一小堆也分成两堆，直到每一堆只有一张卡片（自然有序）。然后，你开始把相邻的两小堆卡片按照顺序合并成一个有序的堆，不断重复这个合并过程，直到所有卡片都合并成一个完全有序的大堆。

2. 伪代码及解释 (LEC3, p.1)

合并排序通常由两个主要函数组成： `MergeSort` （负责递归分解）和 `Merge` （负责合并子序列）。

`MergeSort(A, p, r)`： 对数组 `A` 的子数组 `A[p..r]` 进行排序。

```
MergeSort(A, p, r) {
    // 1. 基本情况：如果子数组只有一个元素或为空 (p >= r)，则它已经有序
    if p < r then {
        // 2. 分解：计算子数组的中间点 q
        q = floor((p+r)/2)

        // 3. 解决：递归排序左右两个子数组
        MergeSort(A, p, q)      // 排序左半部分 A[p..q]
        MergeSort(A, q+1, r)    // 排序右半部分 A[q+1..r]

        // 4. 合并：将已排序的两个子数组 A[p..q] 和 A[q+1..r] 合并
        Merge(A, p, q, r)
    }
}
```

`Merge(A, p, q, r)`： 假设子数组 `A[p..q]` 和 `A[q+1..r]` 均已排序，此函数将它们合并成一个有序的子数组，并放回 `A[p..r]` 。

```
Merge(A, p, q, r) {
    // 1. 计算左右子数组的长度
    n1 = q - p + 1  // 左子数组 A[p..q] 的长度
    n2 = r - q      // 右子数组 A[q+1..r] 的长度

    // 2. 创建临时数组 L[1..n1+1] 和 R[1..n2+1]
    //    多一个位置用于存放哨兵值 (sentinel value)，如无穷大
    Let L[1..n1] and R[1..n2] be new arrays

    // 3. 将 A[p..q] 复制到 L[1..n1]
    for i = 1 to n1 {
        L[i] = A[p + i - 1]
    }
    // 4. 将 A[q+1..r] 复制到 R[1..n2]
    for j = 1 to n2 {
        R[j] = A[q + j]
    }

    // 5. 设置哨兵值，简化比较逻辑，避免每次检查是否已到数组末尾
    L[n1 + 1] = ∞  // 无穷大
    R[n2 + 1] = ∞  // 无穷大

    // 6. 合并过程
    i = 1  // L 的当前下标
    j = 1  // R 的当前下标

    // k 是 A 中当前填充位置的下标，从 p 开始到 r 结束
    for k = p to r {
        if L[i] <= R[j] then {
            A[k] = L[i]
            i = i + 1
        } else {
            A[k] = R[j]
            j = j + 1
        }
    }
}
```

- PPT中的 `Merge` 过程使用了哨兵值（如 ∞）来简化边界检查 。
- **合并步骤**：创建两个临时数组 `L` 和 `R`，分别存放待合并的两个已排序子数组 `A[p..q]` 和 `A[q+1..r]` 。然后，比较 `L` 和 `R` 的首元素，将较小的元素复制回原数组 `A` 的相应位置，并移动该较小元素所在数组的指针。重复此过程，直到一个临时数组的元素全部被复制完毕，再将另一个临时数组的剩余部分直接复制到 `A` 的末尾（使用哨兵值可以使这个过程更统一）。
- `Merge` 操作的时间复杂度为 $\Theta(n)$，其中 $n = r - p + 1$ 是待合并元素的总数，因为它对 n 个元素中的每一个都执行常数次操作（复制到临时数组，比较，复制回原数组）。

例子 (LEC3, p.2):
PPT中给出了一个合并排序的例子，将 `[51, 13, 10, 64, 34, 5, 32, 21]` 排序的过程：

1. **分解**:
   - `[51, 13, 10, 64]` 和 `[34, 5, 32, 21]`
   - `[51, 13]`，`[10, 64]` 和 `[34, 5]`，`[32, 21]`
   - `[51]`，`[13]`，`[10]`，`[64]` 和 `[34]`，`[5]`，`[32]`，`[21]` (基本情况，已排序)
2. **合并**:
   - `[13, 51]`，`[10, 64]` 和 `[5, 34]`，`[21, 32]`
   - `[10, 13, 51, 64]` 和 `[5, 21, 32, 34]`
   - 最终 `[5, 10, 13, 21, 32, 34, 51, 64]`

3. 效率分析

- **时间复杂度 (LEC3, p.1)**:
  - `MergeSort` 的运行时间 $T(n)$ 可以用递归关系式描述：
    - $T(n) = \Theta(1)$ 如果 $n = 1$ (基本情况)
    - $T(n) = 2T(n/2) + \Theta(n)$ 如果 $n > 1$ (两次递归调用处理一半规模的问题，加上 $\Theta(n)$ 的合并时间)
  - 根据主定理 (Master Theorem) 的情况2（因为 $f(n) = \Theta(n)$ 且 $n^{\log_b a} = n^{\log_2 2} = n^1 = n$），可以解得 $T(n) = \Theta(n \log n)$。
  - 这个时间复杂度对于**最好情况、最坏情况和平均情况都是** $\Theta(n \log n)$，因为无论输入数据如何，分解和合并的步骤都是固定的。
- **空间复杂度**:
  - `Merge` 操作需要额外的临时数组来存放待合并的子序列。在合并 n 个元素时，临时数组的大小为 $O(n)$。
  - 因此，合并排序的空间复杂度为 $O(n)$。它不是原地排序算法。

4. 稳定性

- 合并排序可以实现为**稳定**的排序算法。
- 在 `Merge` 函数中，当比较 `L[i]` 和 `R[j]` 时，如果 `L[i] == R[j]`，我们**优先选择左边子数组 `L` 中的元素**（即 `A[k] = L[i]; i = i + 1;`），这样就能保持相等元素的原始相对顺序。PPT中的伪代码 `if L[i] <= R[j]` 保证了这一点。

5. 备考要点

- **简答题考点**:
  - 合并排序的基本思想（分治策略）。
  - 解释合并排序的时间复杂度为什么是 $\Theta(n \log n)$。
  - 合并排序是否稳定？为什么？
  - 合并排序的空间复杂度及其原因。
  - 与快速排序等算法比较其优缺点（例如，合并排序时间复杂度稳定，但空间开销大）。
- **算法分析题考点**:
  - 写出并求解合并排序的时间复杂度递归关系式。
  - 分析 `Merge` 过程的时间和空间复杂度。

    - 可能会给定一个序列，要求手动模拟合并排序的分解和合并过程。
- **设计证明题考点**：
    - 描述 `MergeSort` 或 `Merge` 算法的伪代码。
    - 证明 `Merge` 过程的正确性或其时间复杂度。

6. 总结

合并排序是一种高效且稳定的排序算法，其在各种情况下的时间复杂度均为 $\Theta(n \log n)$，这使其成为一个可靠的选择。主要的缺点是需要 $O(n)$ 的额外存储空间。它清晰地展示了分治思想的威力。

排序算法详解之三：快速排序 (QuickSort) (LEC3, LEC5)

快速排序是另一种采用分治策略的排序算法，由C.A.R. Hoare在1960年提出。它通常比合并排序更快，因为它的常数因子较小，并且在很多情况下可以实现原地排序（或接近原地排序，空间复杂度为 $O(\log n)$）。

1. 基本思想 (分治策略) (LEC3, p.2)

1. **分解 (Divide)**：
    - 从数组中选择一个元素作为 **主元 (pivot)**（也称基准元）。
    - **划分 (Partition)** 数组：重新排列数组中的元素，使得所有小于或等于主元的元素都移动到主元的前面（左边），所有大于主元的元素都移动到主元的后面（右边）。在这个过程中，主元本身会被放置到其最终排序后的正确位置上。划分操作是快速排序的核心。
2. **解决 (Conquer)**：
    - 递归地调用快速排序算法，分别对主元左边的子数组和主元右边的子数组进行排序。
3. **合并 (Combine)**：
    - 这个步骤是平凡的，因为子数组是原地排序的。当两个子数组都有序后，整个数组自然就有序了，无需额外的合并操作。这也是快速排序“快速”的原因之一。

2. 伪代码及解释 (LEC3, pp.2-3)

快速排序主要由 `QUICKSORT` 过程和 `PARTITION` 过程组成。

`QUICKSORT(A, p, r)`：对数组 `A` 的子数组 `A[p..r]` 进行排序。

```
QUICKSORT(A, p, r) { // A是要排序的数组，p是起始下标，r是结束下标
    // 1. 基本情况：如果子数组只有一个元素或为空 (p >= r)，则它已经有序
    if p < r then {
        // 2. 分解：调用 PARTITION 对 A[p..r] 进行划分，q 是主元划分后所处的位置
        q = PARTITION(A, p, r)

        // 3. 解决：递归排序主元左边的子数组和右边的子数组
        QUICKSORT(A, p, q-1)  // 排序左半部分 A[p..q-1]
        QUICKSORT(A, q+1, r)  // 排序右半部分 A[q+1..r]
    }
    // 4. 合并：无需操作
}
```

PARTITION(A, p, r)：对子数组 A[p..r] 进行划分。有多种划分策略，PPT中 (LEC3, p.3) 展示了一种常用的 (Lomuto 划分方案的变体，通常选择 A[r] 作为主元）。

```
PARTITION(A, p, r) {
    // 1. 选择子数组的最后一个元素 A[r] 作为主元 x
    x = A[r]

    // 2. i 用于标记“小于等于主元”区域的右边界。
    //    初始时，i 指向 p-1，表示这个区域为空。
    //    A[p..i] 将是小于等于主元的元素。
    i = p - 1

    // 3. j 是当前扫描的元素指针，从子数组的第一个元素 A[p] 开始，
    //    到主元的前一个元素 A[r-1] 结束。
    //    A[i+1..j-1] 将是大于主元的元素。
    //    A[j..r-1] 是当前还未处理的元素。
    for j = p to r - 1 {
        // 4. 如果当前元素 A[j] 小于或等于主元 x
        if A[j] <= x then {
            i = i + 1               // 扩展“小于等于主元”区域
            exchange A[i] with A[j] // 将 A[j] 交换到这个区域的末尾
        }
    }

    // 5. 循环结束后，A[p..i] 中的所有元素都小于等于 x。
    //    将主元 x (即原来的 A[r]) 与 A[i+1] 交换，
    //    使得主元位于其最终排序后的正确位置。
    exchange A[i+1] with A[r]

    // 6. 返回主元的新下标
    return i + 1
}
```

`PARTITION` **过程图解 (LEC3, p.3, Figure 7.2, 7.3)**:
该图示很好地解释了 `PARTITION` 过程中四个区域的维护:

1. `A[p..i]`: 已处理过的，所有元素 $\leq x$ (主元)。
2. `A[i+1..j-1]`: 已处理过的，所有元素 $> x$。
3. `A[j..r-1]`: 当前未处理的元素区域。
4. `A[r]`: 主元 x。

当 `A[j] <= x` 时， `i` 右移， `A[i]` 和 `A[j]` 交换，然后 `j` 右移。这保持了循环不变量: `A[p..i] <= x < A[i+1..j-1]` 。
当 `A[j] > x` 时，仅 `j` 右移，扩展了大于 `x` 的区域。

3. 效率分析 (LEC3, pp.3-4)

快速排序的性能高度依赖于 `PARTITION` 操作划分的平衡性，即主元选择的好坏。

- **最坏情况 (Worst Case)** (LEC3, p.3):
  - **发生时机**: 当 `PARTITION` 每次都产生极不平衡的划分时。例如，如果输入数组已经有序或逆序，并且总是选择第一个或最后一个元素作为主元，那么一次划分会将问题分解为一个包含 $n-1$ 个元素的子问题和一个空子问题。
  - **递归式**: $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$ ($\Theta(n)$ 是 `PARTITION` 的时间开销)。
  - **时间复杂度**: $T(n) = \Theta(n^2)$。
- **最好情况 (Best Case)** (LEC3, p.4):
  - **发生时机**: 当 `PARTITION` 每次都能将问题划分为大小几乎相等的两个子问题时。理想情况下，主元恰好是待排序元素的中位数。
  - **递归式**: $T(n) = 2T(n/2) + \Theta(n)$。
  - **时间复杂度**: 根据主方法情况2， $T(n) = \Theta(n \log n)$。
- **平均情况 (Average Case)**:
  - 可以证明，即使划分不总是完美的，只要划分是“比较好”的（例如，总是按某个固定比例划分，如1:9），时间复杂度仍然是 $\Theta(n \log n)$。
  - 对于随机输入，或者通过**随机化主元选择**（例如，随机选取 `A[p..r]` 中的一个元素作为主元，或者先将 `A[r]` 与 `A[p..r]` 中随机一个元素交换），可以使得快速排序的**期望运行时间为** $\Theta(n \log n)$ (LEC5, pp.5-6)。 随机化使得出现最坏情况的概率变得极小。

4. 空间复杂度

- `PARTITION` 操作是原地的。
- 快速排序的空间复杂度主要来自递归调用所产生的栈空间。
  - **最好情况** (完美平衡划分): 递归深度为 $O(\log n)$。
  - **最坏情况** (极不平衡划分): 递归深度为 $O(n)$。
  - 通过对递归进行优化，例如总是先递归处理较短的子数组，可以将最坏情况下的栈空间限制在 $O(\log n)$。

5. 稳定性

- 快速排序的 `PARTITION` 操作（如Lomuto或Hoare划分）通常是**不稳定**的。在元素交换过程中，具有相同值的元素的原始相对顺序可能会被打乱。

6. 备考要点

- **简答题考点**：
  - 快速排序的基本思想（分治、主元选择、划分）。
  - `PARTITION` 操作的作用和过程。
  - 解释快速排序最坏情况发生的原因（例如，输入已排序且主元选择不当）及其时间复杂度。
  - 解释快速排序最好情况发生的原因（例如，主元总是中位数）及其时间复杂度。
  - 快速排序的平均时间复杂度是多少？如何达到这个平均性能（随机化的作用）？
  - 快速排序是否稳定？为什么？
  - 与合并排序相比，快速排序的优缺点（例如，平均情况下更快，空间效率更高，但不稳定且有最坏情况风险）。
- **算法分析题考点**：
  - 给定一个数组和主元选择策略（如选择第一个、最后一个或中间元素），手动模拟一次或多次 `PARTITION` 操作。
  - 分析特定 `PARTITION` 实现的时间复杂度。
  - 写出并求解快速排序在最坏、最好情况下的时间复杂度递归式。
  - （结合LEC5）随机化快速排序的期望运行时间分析可能作为更深入的题目。
- **设计证明题考点**：
  - 描述 `QUICKSORT` 或 `PARTITION` 算法的伪代码。
  - 证明 `PARTITION` 算法的正确性（通常使用循环不变量）。
  - 讨论不同的主元选择策略及其对性能的影响。

7. 总结

快速排序因其出色的平均性能（$\Theta(n \log n)$）和相对较好的空间效率（平均 $O(\log n)$ 栈空间，且常被认为是原地排序）而被广泛使用。其性能的关键在于 `PARTITION` 步骤能否产生较为平衡的划分。随机化是确保良好平均性能的常用技巧。尽管它是不稳定的且存在 $\Theta(n^2)$ 的最坏情况，但在实践中通常表现优异。

排序算法探讨：基于比较的排序算法的理论下界 (LEC3)

1. 什么是基于比较的排序 (Comparison Sort)？ (LEC3, p.4)

- **定义**：这类排序算法通过反复比较输入序列中元素对 (a_i, a_j) 的大小关系（如 $a_i < a_j, a_i \leq a_j, a_i = a_j, a_i \geq a_j$, 或 $a_i > a_j$）来确定它们之间的相对顺序，进而将整个序列排列成有序状态 。
- 算法在排序过程中，除了比较操作外，还可能进行数据的移动或交换，但决策的依据完全依赖于比较的结果。
- 我们熟悉的很多排序算法都属于这一类，例如：插入排序、选择排序、冒泡排序、希尔排序、合并排序、快速排序、堆排序等。

2. 决策树模型 (Decision Tree Model) (LEC3, p.4)

为了分析基于比较的排序算法的性能下界，我们引入决策树模型。

- **概念**:
  - 对于一个给定的输入规模 n（即有 n 个元素需要排序），任何一个基于比较的排序算法都可以被抽象地表示为一棵**二叉决策树**。
  - **内部节点 (Internal Nodes)**：树中的每一个内部节点代表一次元素之间的比较，例如比较 `A[i]` 和 `A[j]`。每个内部节点有两个子节点，分别对应比较结果的两种可能性（例如，`A[i] <= A[j]` 或 `A[i] > A[j]`）。
  - **叶子节点 (Leaf Nodes)**：树中的每一个叶子节点代表一种可能的排序结果，即输入元素的一种**排列 (permutation)**。对于 n 个互不相同的元素，总共有 n种可能的排列。
  - **执行路径 (Execution Trace)**：算法对特定输入的一次执行过程对应于决策树中从根节点到某个叶子节点的一条路径。这条路径的长度（即经过的内部节点数）就代表了该次执行所进行的比较次数。
- **重要特性** (LEC3, p.4):
  - 决策树必须能够覆盖所有可能的输入情况，因此它**至少要有 n个叶子节点**，因为每一种输入排列都必须导向一个唯一的、正确的排序结果。
  - 对于一个特定的基于比较的排序算法，其决策树是固定的（针对特定输入规模 n）。
  - 算法在**最坏情况下的比较次数**就等于这棵决策树的**高度 h**（即从根到最远叶子节点的路径长度）。

3. 下界证明 (Lower Bound Proof) (LEC3, p.4, 5)

目标是证明任何基于比较的排序算法在最坏情况下至少需要 $\Omega(n \log n)$ 次比较。

1. **引理 (Lemma)**：任何高度为 h 的二叉树至多有 2^h 个叶子节点 。
   - **证明 (归纳法)** (LEC3, p.4):
     - **基础 (Basis)**：当 $h = 0$ 时，树只有一个节点（根即叶子），叶子数为 $1 = 2^0$。引理成立。
     - **归纳步骤 (Inductive step)**：假设对于高度为 $h-1$ 的二叉树，引理成立，即其至多有 2^{h-1} 个叶子。当树的高度增加到 h 时，原来的叶子节点都可能成为新的内部节点，每个这样的节点最多可以有两个子节点（新的叶子）。因此，高度为 h 的树的叶子节点数至多是高度为 $h-1$ 的树的叶子节点数的两倍，即 $2 \times 2^{h-1} = 2^h$。引理成立。
2. **下界推导** (LEC3, p.5):
   - 我们已经知道，一个基于比较的排序算法的决策树必须至少有 $L = n$个叶子节点，以区分所有可能的输入排列。
   - 设该决策树的高度为 h（即最坏情况下的比较次数）。根据上述引理，这棵树最多有 2^h 个叶子节点。
   - 因此，我们有 $n \leq L \leq 2^h$。
   - 对不等式 $n \leq 2^h$ 两边取以2为底的对数，得到:
     $\log_2(n) \leq \log_2(2^h)$
     $\log_2(n) \leq h$
   - 所以，$h \geq \log_2(n)$。
   - 接下来需要估计 $\log_2(n)$ 的下界。我们知道 $n = n \times (n-1) \times \cdots \times 1$。
     $\log(n) = \sum_{i=1}^{n} \log i$
     可以使用积分近似或者更简单的放缩:
     $n! > (n/2)^{n/2}$ (因为 n中至少有 $n/2$ 个因子大于 $n/2$)

$\log_2(n) > \log_2((n/2)^{n/2}) = (n/2)\log_2(n/2) = (n/2)(\log_2 n - \log_2 2) = (n/2)(\log_2 n - 1)$
因此，$\log_2(n) = \Omega(n \log n)$。
更精确地，由斯特林近似公式 $n \approx \sqrt{2\pi n}(n/e)^n$，可知 $\log(n) \approx n \log n - n \log e + O(\log n)$，所以 $\log_2(n) = \Theta(n \log n)$。

- **结论**：$h = \Omega(n \log n)$。这意味着任何基于比较的排序算法在最坏情况下所需要的比较次数的增长阶至少是 $n \log n$ 。

4. 意义与影响

- 这个$\Omega(n \log n)$ 的下界告诉我们，不存在一个基于比较的排序算法能够在最坏情况下比 $n \log n$ 快（就比较次数而言）。
- 像合并排序、堆排序这样的算法，它们的最坏情况时间复杂度都是 $\Theta(n \log n)$，因此从渐进意义上讲，它们是最优的基于比较的排序算法。
- 快速排序的平均情况时间复杂度是 $\Theta(n \log n)$，也是渐进最优的。
- 如果一个排序算法的时间复杂度低于 $\Omega(n \log n)$（例如线性时间排序算法如计数排序、基数排序、桶排序），那么它一定**不是**基于比较的排序算法，而是利用了输入数据的一些特殊性质（如取值范围、分布等）。

5. 备考要点

- **简答题考点**：
  - 什么是基于比较的排序算法?
  - 解释决策树模型如何用于表示基于比较的排序算法。
  - 基于比较的排序算法在最坏情况下的时间复杂度下界是多少？为什么？（简述证明思路）
  - $\Omega(n \log n)$ 这个下界对我们设计和选择排序算法有何启示?
- **算法分析题考点**：
  - 可能会要求画出小规模输入（如 $n = 3$）的决策树。
  - 理解证明过程中的关键步骤，如叶子节点数与 n的关系，树高与叶子节点数的关系。
- **设计证明题考点**：
  - 虽然直接要求完整证明下界的可能性不大，但理解其核心逻辑（利用决策树叶子数量和树高关系）是很重要的。
  - 可能会问到，如果一个算法声称在最坏情况下用 $O(n)$ 时间完成任意 n 个数的排序，它是否可能是一个基于比较的排序算法？（答：不可能，因为它违反了 $\Omega(n \log n)$ 的下界）。

排序算法详解之四：计数排序 (Counting Sort) (LEC3)

计数排序是一种**非基于比较**的排序算法，它利用了输入元素的一些特殊性质（通常是整数且范围不大）来实现线性时间的排序。

1. 适用条件 (LEC3, p.5)

- 输入元素必须是**整数**。
- 这些整数必须在一个已知的、相对较小的**范围**内。例如，所有元素的值都在 0 到 k 之间，其中 k 是一个整数。如果 k 的值相对于元素个数 n 非常大，计数排序的效率会降低（空间开销大）。

2. 基本思想 (LEC3, p.5)

计数排序的核心思想是统计每个不同元素出现的次数，然后根据这些统计信息直接确定每个元素在排序后输出数组中的最终位置。

1. **统计频率**：创建一个辅助数组（称为计数数组 `C`），其大小为输入整数的范围（例如 $k+1$）。遍历输入数组 `A`，对于 `A` 中的每个元素 `x`，将 `C[x]` 的值加1。完成此步骤后，`C[i]` 存储的是输入数组中等于 `i` 的元素的个数。
2. **计算位置信息**：修改计数数组 `C`，使其每个位置 `C[i]` 存储的是输入数组中小于或等于 `i` 的元素的总个数。这可以通过从前往后累加 `C` 数组中的值来实现：`C[i] = C[i] + C[i-1]`。完成此步骤后，`C[i]` 就指明了值为 `i` 的元素在排好序的输出数组 `B` 中应该放置的最右边的位置（或者说是小于等于 `i` 的元素有多少个，从而可以推断出位置）。
3. **放置元素**：创建一个输出数组 `B`，与输入数组 `A` 大小相同。**从后向前**遍历输入数组 `A`。对于 `A` 中的每个元素 `A[j]`：
   - 将其放置到输出数组 `B` 的 `C[A[j]]` 位置上。
   - 然后将 `C[A[j]]` 的值减1（这是为了当存在重复元素时，下一个相同的元素能被正确放置到其前一个位置，从而保证稳定性）。
   - 从后向前遍历是为了保证排序的**稳定性**。

3. 伪代码及解释 (LEC3, p.5)

以下是PPT中提供的计数排序伪代码：

```
COUNTING-SORT(A, B, n, k)
// A: 输入数组 (假设下标从1到n)
// B: 输出数组 (大小与A相同)
// n: 输入数组 A 的元素个数
// k: 输入元素的最大值 (假设元素范围是 0 到 k，或 1 到 k，根据实际调整C数组大小和循环)
// PPT中C数组的下标似乎是从0到k (或者1到k)，这里我们假设元素值在0到k之间，则C大小为k+1

1. let C[0..k] be a new array        // 创建计数数组C，大小为 k+1
2. for i = 0 to k do
3.     C[i] = 0                    // 初始化计数数组C为0

4. for j = 1 to n do                 // 遍历输入数组A
5.     C[A[j]] = C[A[j]] + 1        // 统计元素A[j]的出现次数
   // 此时 C[i] 包含等于 i 的元素的个数

6. for i = 1 to k do                 // 遍历计数数组C (从1开始)
7.     C[i] = C[i] + C[i-1]          // 修改C，使C[i]包含小于或等于i的元素的个数
   // 此时 C[i] 包含小于或等于 i 的元素的个数 (即元素i在输出数组B中的最后位置)

8. for j = n downto 1 do             // 从后向前遍历输入数组A (保证稳定性)
9.     B[C[A[j]]] = A[j]             // 将元素A[j]放到输出数组B的正确位置上
10.    C[A[j]] = C[A[j]] - 1           // 对应计数值减1，为下一个相同元素准备位置
```

例子 (LEC3, p.5):
假设输入数组 `A = [2, 5, 3, 0, 2, 3, 0, 3]`，元素个数 `n=8`，最大值 `k=5`。

1. 初始化 `C[0..5] = [0, 0, 0, 0, 0, 0]`。
2. 统计频率（第4-5行）：
   - `A[1]=2 -> C[2]=1`
   - `A[2]=5 -> C[5]=1`
   - `A[3]=3 -> C[3]=1`
   - `A[4]=0 -> C[0]=1`
   - `A[5]=2 -> C[2]=2`
   - `A[6]=3 -> C[3]=2`
   - `A[7]=0 -> C[0]=2`
   - `A[8]=3 -> C[3]=3`
   - 此时 `C = [2, 0, 2, 3, 0, 1]` （`C[0]=2` 个0，`C[1]=0` 个1，`C[2]=2` 个2，`C[3]=3` 个3，`C[4]=0` 个4，`C[5]=1` 个5）。
3. 计算位置信息（第6-7行）：
   - `C[1] = C[1] + C[0] = 0 + 2 = 2`
   - `C[2] = C[2] + C[1] = 2 + 2 = 4`
   - `C[3] = C[3] + C[2] = 3 + 4 = 7`
   - `C[4] = C[4] + C[3] = 0 + 7 = 7`
   - `C[5] = C[5] + C[4] = 1 + 7 = 8`
   - 此时 `C = [2, 2, 4, 7, 7, 8]` （表示0的最后位置是2，1的最后位置是2，2的最后位置是4，等等）。
4. 放置元素（第8-10行），从 `j=8` 向下到 `j=1`：
   - `j=8, A[8]=3`： `B[C[3]] = B[7] = 3_c` (假设这是第三个3)。 `C[3]=6`。
   - `j=7, A[7]=0`： `B[C[0]] = B[2] = 0_b` (假设这是第二个0)。 `C[0]=1`。
   - `j=6, A[6]=3`： `B[C[3]] = B[6] = 3_b` (假设这是第二个3)。 `C[3]=5`。
   - `j=5, A[5]=2`： `B[C[2]] = B[4] = 2_b` (假设这是第二个2)。 `C[2]=3`。
   - `j=4, A[4]=0`： `B[C[0]] = B[1] = 0_a` (假设这是第一个0)。 `C[0]=0`。
   - `j=3, A[3]=3`： `B[C[3]] = B[5] = 3_a` (假设这是第一个3)。 `C[3]=4`。
   - `j=2, A[2]=5`： `B[C[5]] = B[8] = 5_a`。 `C[5]=7`。
   - `j=1, A[1]=2`： `B[C[2]] = B[3] = 2_a` (假设这是第一个2)。 `C[2]=2`。
   - 最终 `B = [0_a, 0_b, 2_a, 2_b, 3_a, 3_b, 3_c, 5_a]`。

4. 效率分析 (LEC3, p.6)

- **时间复杂度**:
  - 第2-3行（初始化C）：$\Theta(k)$。
  - 第4-5行（统计频率）：$\Theta(n)$。
  - 第6-7行（计算位置）：$\Theta(k)$。
  - 第8-10行（放置元素）：$\Theta(n)$。
  - 总时间复杂度：$\Theta(n+k)$。
  - 当 $k = O(n)$ 时（即最大元素与元素个数成正比），计数排序的时间复杂度为 $\Theta(n)$，即线性时间。
  - 如果 k 非常大，例如 $k = n^2$ 或 $k = 2^n$，则计数排序的效率会很低。

- **空间复杂度**:
  - 需要一个大小为 $k+1$ 的计数数组 `C` 和一个大小为 n 的输出数组 `B`。
  - 总空间复杂度：$\Theta(n+k)$。

5. 稳定性 (LEC3, p.6)

- 计数排序是**稳定**的排序算法。
- 稳定性是由伪代码的第8-10行保证的：通过**从后向前**遍历输入数组 `A`，并将元素放入输出数组 `B` 中由 `C` 数组指示的位置，可以确保具有相同值的元素在输出数组中的相对顺序与它们在输入数组中的相对顺序一致。如果从前往后遍历，则排序结果虽然正确，但会失去稳定性。

6. 备考要点

- **简答题考点**:
  - 计数排序的适用条件（整数，范围已知且较小）。
  - 计数排序的基本思想和关键步骤。
  - 计数排序的时间复杂度和空间复杂度，以及它们与 k 的关系。
  - 计数排序为什么是稳定的？哪个步骤保证了其稳定性？
  - 计数排序的优缺点（优点：特定条件下线性时间；缺点：对输入数据类型和范围有要求，空间开销可能较大）。
- **算法分析题考点**:
  - 给定一个输入序列和 k 值，手动模拟计数排序的整个过程，展示 `C` 数组和 `B` 数组在关键步骤后的状态。
  - 分析计数排序中各个循环的时间开销。
- **设计证明题考点**:
  - 可能会要求写出计数排序的伪代码。
  - 解释为什么计数排序可以突破 $\Omega(n \log n)$ 的下界（因为它不依赖于元素间的比较，而是利用了元素的绝对值信息）。

7. 总结

计数排序是一种非常高效的线性时间排序算法，但其应用场景受限于输入数据的类型和范围。当条件满足时，它通常是整数排序的首选方法之一，并且由于其稳定性，它常被用作更复杂排序算法（如基数排序）的子过程。

排序算法详解之五：基数排序 (Radix Sort) (LEC3)

基数排序也是一种**非基于比较**的排序算法，它通过逐位比较数字（或其他可按“位”分解的数据）来进行排序。它通常与一种稳定的子排序算法（如计数排序）结合使用。

1. 适用条件 (LEC3, p.6)

- 输入元素是可以被分解为多个“数字”或“位”的数据。常见的例子是整数，也可以是字符串（按字符排序）。
- 每一“位”的取值范围（即基数 k）不能太大，以便于子排序算法（如计数排序）高效执行。

2. 基本思想 (LEC3, p.6)

基数排序的核心思想是按照元素的各个“位”的优先级，从低到高（LSD - Least Significant Digit first）或从高到低（MSD - Most Significant Digit first）进行多次排序。最常用的是LSD基数排序。

LSD (Least Significant Digit) 基数排序过程：

1. **确定位数**：找出待排序元素中具有最多“位数”的元素，以确定需要进行多少轮排序（设为 d 轮）。对于不等长的元素，较短的元素可以在高位补0（或其它不影响比较的默认值）。
2. **逐位排序**：从最低有效位（第1位）开始，到最高有效位（第 d 位）结束，依次对所有元素按照当前处理的“位”进行排序。
3. **使用稳定排序**：在每一位的排序过程中，**必须使用一个稳定的排序算法**（例如计数排序）。稳定性是基数排序能够正确工作的关键。

为什么需要稳定排序?
因为当对某一位进行排序时，我们希望那些在该位上具有相同值的元素能够保持它们在前一位排序后已经确立的相对顺序。如果内部排序不稳定，那么之前位的排序结果可能会被打乱，导致最终结果错误。

例如，排序两位数 (27, 17, 23, 13):

- **按个位数排序 (稳定)**:
  - 2**3**, 1**3** (假设13在前)
  - 2**7**, 1**7** (假设17在前)
  - 得到：[23, 13, 27, 17] (注意：13在23之前，17在27之前，因为它们的个位数相同，保持了原输入中3和7的相对顺序——虽然这里不明显，但如果原输入是(13, 23)，则排序后是(13,23)，不是(23,13))。更准确的例子：原序列 `[..., 17, ..., 27, ...]`，个位都是7，稳定排序后它们的相对顺序不变。
  - 实际例子：按个位排 `[27, 17, 23, 13]`，假设计数排序后个位小的在前：`[23_1, 13_2, 27_3, 17_4]`（假设13在23之后输入，17在27之后输入)。
  - 更清晰的例子：排 `[17, 12, 27]`。按个位：`[12, 17, 27]` (7和2)。假设排 `[17, 27, 12]` 按个位稳定排序是 `[12, 17, 27]`。
- **按十位数排序 (稳定，基于上一轮结果)**:
  - 对 `[12, 17, 27]` (例子数据修正) 按十位排：
  - **1**2, **1**7 (17在12之后，因为个位排序时7>2，所以17在12之后，这里按十位排，十位都是1，保持17在12后的相对顺序)
  - **2**7
  - 得到：`[12, 17, 27]`。
  - 如果上一轮是 `[23, 13, 27, 17]`，按十位排：`[13, 17, 23, 27]`。因为13, 17的十位是1，23, 27的十位是2。在十位为1的组内，13在17前因为上一轮个位排序的结果；十位为2的组内，23在27前因为上一轮个位排序的结果。

3. 伪代码及解释 (LEC3, p.6)

```
RADIX-SORT(A, d)
// A: 输入数组
// d: 元素的位数 (例如，对于3位十进制数，d=3)

1. for i = 1 to d do  // 从最低有效位 (第1位) 到最高有效位 (第d位)
2.     use a stable sort to sort array A on digit i
       // 使用一个稳定的排序算法 (如计数排序) 按照当前处理的第 i 位对数组 A 进行排序
```

解释:

- `digit i` 指的是元素的第 i 个有效位。例如，对于数字 `345` ，如果从右到左数，第1位是5，第2位是4，第3位是3。
- 在每次迭代中，整个数组 `A` 都会根据当前位的值被重新排列。由于使用了稳定排序，之前位的排序成果得以保留。

4. 效率分析 (LEC3, p.6)

- 设 n 是待排序元素的个数。
- 设 d 是元素的位数（或关键字的个数）。
- 设 k 是每一位上可能的取值个数（即基数，例如对于十进制数，$k = 10$；对于二进制数，$k = 2$；对于ASCII字符，$k = 128$ 或 $k = 256$）。
- 如果在每一轮中使用的稳定排序算法是计数排序，则该轮排序的时间复杂度为 $\Theta(n + k)$ 。
- 由于总共有 d 轮排序，所以基数排序的总时间复杂度为 $\Theta(d(n + k))$ 。
- **何时达到线性时间?**
  - 如果 d 是一个常数（即位数固定或者较小），并且 k（基数）也是一个常数或者 $k = O(n)$，那么基数排序的时间复杂度可以达到 $\Theta(n)$。
  - 例如，对 n 个范围在 0 到 $n^c - 1$（c 是常数）之间的整数进行排序。这些数可以看作是 c 位的 n 进制数（$d = c, k = n$），或者 $c \log_b n$ 位的 b 进制数（$d = c \log_b n, k = b$）。如果选择基数 $b = n$，则 $d \approx c$，每轮时间 $\Theta(n + n) = \Theta(n)$，总时间 $\Theta(c \cdot n) = \Theta(n)$。如果选择固定基数（如 $b = 10$ 或 $b = 2$），则 $d = O(\log_b N_{max})$（其中 N_{max} 是最大数的值）。如果 N_{max} 是 n 的多项式级别，如 n^c，则 $d = O(c \log_b n)$，此时每轮 $\Theta(n + k)$，总时间 $\Theta(\log_b(n^c) \cdot (n + k)) = \Theta(d(n + k))$。

5. 空间复杂度

- 基数排序的空间复杂度主要取决于其内部使用的稳定排序算法。
- 如果使用计数排序作为子排序算法，则每一轮需要 $\Theta(n + k)$ 的额外空间（计数数组和输出数组，如果输出数组可以复用输入数组，则主要是计数数组 $\Theta(k)$ 和一个临时输出 $\Theta(n)$）。
- 因此，基数排序的空间复杂度通常是 $\Theta(n + k)$。

6. 稳定性 (LEC3, p.6)

- 基数排序是**稳定**的，前提是其内部使用的子排序算法是稳定的 。这是基数排序正确性的关键保证。

7. 备考要点

- **简答题考点**:
  - 基数排序的基本思想（LSD优先，逐位使用稳定排序）。
  - 基数排序的适用条件。
  - 为什么基数排序需要使用稳定的子排序算法?
  - 基数排序的时间复杂度和空间复杂度，以及它们与位数 d 和基数 k 的关系。
  - 基数排序如何能达到线性时间?
- **算法分析题考点**:
  - 给定一个整数序列，手动模拟LSD基数排序的过程，展示每一轮排序后的中间结果。
  - 分析在特定 d 和 k 的情况下，基数排序的效率。
- **设计证明题考点**:
  - 可能会要求描述基数排序的算法。
  - 通过归纳法证明基数排序的正确性（核心在于证明第 i 轮排序后，元素按最低 i 位有序，且依赖于第 $i-1$ 轮的稳定性）。

8. 总结

基数排序是一种巧妙的非比较排序算法，它通过将排序问题分解为对元素各个数位的多次简单排序来解决。当元素的位数 d 和每位的基数 k 控制得当时，它可以实现优于 $\Theta(n \log n)$ 的性能，甚至达到线性时间。其正确性高度依赖于所用子排序算法的稳定性。

排序算法详解之六：桶排序 (Bucket Sort) (LEC3)

桶排序是另一种非基于比较的排序算法，它假设输入数据服从均匀分布，并利用这个特性来达到平均情况下的线性时间效率。

1. 适用条件/假设 (LEC3, p.6)

- 输入数据是由一个**随机过程**生成的，该过程将元素**均匀且独立地分布**在某个已知的区间上，例如 $[0, 1)$ 。
- 如果输入数据不满足均匀分布的假设，桶排序的性能可能会显著下降。

2. 基本思想 (LEC3, p.6)

1. **设置桶 (Buckets)**：根据输入数据的范围和数量，创建 n 个（n 是输入元素的数量）大小相等的子区间，这些子区间称为“桶”。例如，如果输入数据在 $[0, 1)$ 区间内，可以将该区间划分为 n 个桶，每个桶的范围是 $[i/n, (i+1)/n)$，其中 $i = 0, 1, \ldots, n-1$。
2. **分发元素 (Distribution)**：遍历输入数组中的每个元素，根据元素的值将其放入相应的桶中。通常，每个桶使用链表来存储落入其中的元素。
3. **桶内排序 (Sort Buckets)**：对每个非空的桶中的元素分别进行排序。由于假设元素均匀分布，每个桶中元素的期望数量较少，因此可以使用简单且对于小规模数据高效的排序算法，如**插入排序** 。
4. **连接桶 (Concatenation)**：按照桶的顺序（即它们所代表的区间的顺序），依次遍历每个桶中的已排序元素，并将它们连接起来，形成最终的有序输出序列。

3. 伪代码及解释 (LEC3, p.7)

以下是桶排序算法的伪代码描述：

```
BUCKET-SORT(A, n) // A 是输入数组，n 是元素个数 (假设A中元素在 [0,1) 之间)
1. let B[0..n-1] be a new array of n empty lists (buckets) // 创建n个空桶(链表)
2. for i = 1 to n do                                  // 遍历输入数组 A (假设A下标从1到n)
3.     insert A[i] into list B[floor(n * A[i])]  // 将元素 A[i] 放入对应的桶 B[j] 中
                                                 // 注意下标计算：n*A[i] 将 [0,1) 的数映射到 [0,n)，取floor得
4. for i = 0 to n-1 do                              // 遍历每个桶
5.     sort list B[i] with insertion sort           // 使用插入排序对桶 B[i] 内的元素进行排序
6. concatenate the lists B[0], B[1], ..., B[n-1] together in order // 按顺序连接所有桶中的元素
7. return the concatenated list
```

例子 (LEC3, p.7):
PPT中给出了一个例子，输入数组 `A = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]`，这里 $n = 10$。

1. 创建10个桶 `B[0]` 到 `B[9]`。
   - `B[0]` 对应区间 `[0, 0.1)`
   - `B[1]` 对应区间 `[0.1, 0.2)`
   - ...
   - `B[9]` 对应区间 `[0.9, 1.0)`
2. 分发元素：
   - `0.78 -> B[floor(10*0.78)] = B[7]`
   - `0.17 -> B[floor(10*0.17)] = B[1]`
   - `0.39 -> B[3]`
   - `0.26 -> B[2]`
   - `0.72 -> B[7]`
   - `0.94 -> B[9]`
   - `0.21 -> B[2]`
   - `0.12 -> B[1]`
   - `0.23 -> B[2]`
   - `0.68 -> B[6]`

   此时各桶内容（未排序）：
   - `B[0]: []`
   - `B[1]: [0.17, 0.12]`
   - `B[2]: [0.26, 0.21, 0.23]`
   - `B[3]: [0.39]`
   - `B[4]: []`
   - `B[5]: []`
   - `B[6]: [0.68]`
   - `B[7]: [0.78, 0.72]`
   - `B[8]: []`

- B[9]: [0.94]

3. 桶内排序（例如用插入排序）：
   - B[1]: [0.12, 0.17]
   - B[2]: [0.21, 0.23, 0.26]
   - B[7]: [0.72, 0.78]
   - 其他桶不变或已是单个元素。
4. 连接各桶：

```
[0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94]
```

4. 效率分析 (LEC3, p.7)

- **时间复杂度**：
  - 步骤1（创建空桶）：$\Theta(n)$。
  - 步骤2（分发元素）：$\Theta(n)$，因为每个元素计算桶索引和插入链表头（或尾）是 $O(1)$。
  - 步骤3（桶内排序）：这是分析的关键。设 n_i 是桶 $B[i]$ 中的元素个数。如果使用插入排序，对桶 $B[i]$ 排序的时间是 $O(n_i^2)$。所有桶的总排序时间是 $\sum_{i=0}^{n-1} O(n_i^2)$。
    - **平均情况**：假设输入均匀分布，可以证明所有桶中元素数量的平方和的期望值是 $E[\sum_{i=0}^{n-1} n_i^2] = \Theta(n)$。因此，桶内排序的平均总时间是 $\Theta(n)$。
    - 步骤4（连接桶）：$\Theta(n)$，因为需要遍历所有元素一次。
    - 综合起来，桶排序的**平均时间复杂度是** $\Theta(n)$ 。
  - **最坏情况**：如果所有元素都落入同一个桶中（例如，输入数据分布极不均匀），则步骤3的时间将退化为所用桶内排序算法对 n 个元素排序的时间。如果桶内使用插入排序，最坏情况是 $O(n^2)$。
- **空间复杂度**：
  - 需要 $\Theta(n)$ 的空间来存储 n 个桶（的链表头）。
  - 还需要 $\Theta(n)$ 的空间来存储输入元素本身（如果桶是链表实现的，元素会分布在这些链表中）。
  - 总空间复杂度为 $\Theta(n)$。

5. 稳定性

- 桶排序的稳定性取决于其内部使用的桶内排序算法是否稳定。
- 如果在第5步中使用的插入排序是稳定的，并且在第2步中将元素插入到桶的链表时，新元素总是插入到链表的末尾（或者以保持原始顺序的方式插入），那么桶排序可以是**稳定**的。PPT中仅提到用插入排序，而标准插入排序是稳定的。

6. 备考要点

- **简答题考点**：
  - 桶排序的适用条件，尤其是对输入数据分布的假设。
  - 桶排序的基本思想和步骤。
  - 桶排序的平均时间复杂度及其原因（为什么能达到线性）。
  - 桶排序的最坏情况时间复杂度及其原因。
  - 桶排序是否稳定？（取决于桶内排序算法）
- **算法分析题考点**：

    - 给定一个输入序列（通常会说明其符合均匀分布在某个区间），手动模拟桶排序的过程，展示元素如何分桶以及桶内排序和最终连接的结果。
    - 分析为什么在均匀分布的假设下，桶内排序的总期望时间是 $\Theta(n)$。
- **设计证明题考点**：
    - 可能会要求描述桶排序的算法。
    - 讨论输入数据分布对桶排序性能的影响。

7. 总结

桶排序是一种在特定条件下（输入数据均匀分布）非常高效的排序算法，其平均时间复杂度可以达到线性。它的核心思想是利用“分桶”来减少后续排序的规模，从而提高整体效率。理解其对输入分布的依赖性是掌握该算法的关键。

分治策略详解之一：基本思想与开销分析 (LEC4)

1. 分治法的定义和策略思想 (LEC4, p.1)

- **定义**：分治法（Divide and Conquer）是一种重要的算法设计策略，用于解决可以将自身分解为若干个规模较小、与原问题性质相同或相似的独立子问题的问题 。
- **核心思想**：
    i. **分解 (Divide)**：将一个难以直接解决的大问题，分割（或分解）成若干个（通常是两个或更多）规模较小的、相互独立、且与原问题形式相同的子问题。
    ii. **解决 (Conquer)**：若子问题规模已足够小，则直接求解。否则，递归地调用分治算法来解决这些子问题。
    iii. **合并 (Combine)**：将各个子问题的解合并（或组合）起来，从而得到原问题的解。
- **通常形式**：分治算法通常以递归的形式出现 。

2. 分治法的适用前提/假设 (LEC4, p.1)

要有效地使用分治策略，问题通常需要满足以下一些前提条件：

- **可分解性**：原始问题的解能够通过其子问题的解来构建 。
- **子问题独立性**：各个子问题应该是相互独立的，即一个子问题的求解不依赖于另一个子问题的求解。这是与动态规划的一个重要区别。
- **子问题同质性**：分解出的子问题与原问题的性质相同，只是规模更小，从而使得递归调用成为可能。
- **子问题易解性**：子问题在规模缩小到一定程度后，必须能够很容易地直接求解（即递归的终止条件） 。

3. 分治法的算法描述框架 (LEC4, p.1)

一个典型的分治算法可以用如下的递归过程来描述：

```
Solve(I) { // I 代表问题的输入实例
    n = size(I) // 获取问题规模 n

    // 1. 判断是否达到递归基准情况
    if (n < smallsize) then { // smallsize 是一个阈值，表示问题规模小到可以直接求解
        solution = directly_solve(I) //直接解决小规模问题
    } else {
        // 2. 分解 (Divide)
        // 将问题 I 分解成 k 个子问题 I1, I2, ..., Ik
        divide I into I1, I2, ..., Ik

        // 3. 解决 (Conquer)
        // 递归地解决每个子问题
        for each i in {1, ..., k} {
            Si = Solve(Ii) // Si 是子问题 Ii 的解
        }

        // 4. 合并 (Combine)
        // 将子问题的解 S1, S2, ..., Sk 合并成原问题的解
        solution = combine(S1, S2, ..., Sk)
    }
    return solution
}
```

4. 采用分治法的优点 (LEC4, p.1)

- **简化问题**：将复杂的大问题分解为若干个结构相同但规模较小的子问题，使得问题更易于理解和处理 。
- **提高效率**：对于某些问题，分治法可以设计出比其他方法更高效的算法 。
  - 典型的例子包括合并排序 (MergeSort) 和快速排序 (QuickSort)，它们的时间复杂度通常优于简单的平方级排序算法 。
- **天然适合并行计算**：由于子问题通常是相互独立的，它们可以被分配到不同的处理器上并行求解，从而显著提高计算速度。
- **注意**：并非所有采用分治策略的算法都一定更有效。有时其效率可能与其它方法相同甚至更差，因此必须对分治算法的代价进行具体分析 。

5. 分治法的开销分析 (LEC4, p.1)

一个分治算法的总运行时间 $T(n)$（其中 n 是问题规模）通常由三部分组成：

1. **分解开销 (Divide Cost, $D(n)$)**：将原问题分解成子问题所需的时间。
2. **子问题解决开销 (Conquer Cost)**：解决各个子问题所需的时间。如果原问题被分解为 a 个规模为 n/b 的子问题（为简化，假设规模均匀划分），并且解决每个子问题的时间为 $T(n/b)$，则这部分开销为 $aT(n/b)$。
3. **合并开销 (Combine Cost, $C(n)$)**：将子问题的解合并成原问题的解所需的时间。

因此，分治算法的运行时间通常可以用一个**递归关系式**来表示：
$T(n) = D(n) + \sum_{i=1}^{k} T(\text{size}(I_i)) + C(n)$

如果问题被分解为 a 个规模为 n/b 的子问题，且分解和合并的总开销为 $f(n)$（即 $f(n) = D(n) + C(n)$），则递归式通常写为：
$T(n) = aT(n/b) + f(n)$

这个形式的递归式可以使用之前讨论过的主方法 (Master Method)、递归树法或迭代法来求解。

6. 备考要点

- **简答题考点**：
  - 解释分治策略的基本步骤（分解、解决、合并）。
  - 分治策略适用于解决什么类型的问题？（满足可分解性、子问题独立性、同质性、小规模易解等）
  - 采用分治法有哪些优点?
  - 写出分治算法时间复杂度的通用递归表达式。
- **算法分析题考点**：
  - 对于一个具体的分治算法（如后续将要学习的二分搜索、合并排序等），能够写出其时间复杂度的递归关系式，并求解。
  - 分析算法中分解步骤和合并步骤的时间复杂度。
- **设计证明题考点**：
  - 当遇到一个新问题时，思考是否可以用分治策略来设计算法。
  - 如果设计了分治算法，需要能够清晰地描述分解、解决（递归基准）、合并的过程。

分治策略详解之二：二分搜索 (Binary Search) (LEC4)

二分搜索（也称折半查找）是一种在**有序数组**中查找特定元素的高效算法。它完美地体现了分治策略的思想。

1. 基本思想 (LEC4, p.2)

1. **前提**：二分搜索算法要求待搜索的序列必须是**有序的**（通常是升序排列）。
2. **核心步骤**：
   - **分解 (Divide)**：将当前搜索区间的中间位置的元素与目标值进行比较。
     - 如果中间元素等于目标值，则查找成功。
     - 如果中间元素大于目标值，则说明目标值（如果存在）必定在当前区间的左半部分。
     - 如果中间元素小于目标值，则说明目标值（如果存在）必定在当前区间的右半部分。
   - **解决 (Conquer)**：根据比较结果，将搜索范围缩小到左半部分或右半部分，然后对新的、更小的搜索区间递归地（或迭代地）应用二分搜索。
   - **合并 (Combine)**：这一步在二分搜索中是隐式的，因为一旦找到元素或确定元素不存在（搜索区间为空），问题就解决了，不需要额外的合并操作。

2. 算法描述 (LEC4, p.2)

假设我们要在已排序的数组 `A` 的子数组 `A[low...high]` 中查找目标值 `X`。

递归版本伪代码：

```
BinarySearch_Recursive(A, X, low, high) {
    // 1. 基本情况：如果搜索区间无效 (low > high)，则元素不存在
    if low > high then {
        return NOT_FOUND // 或者返回一个特殊值，如 -1
    }

    // 2. 分解：计算中间位置的下标
    mid = floor((low + high) / 2)

    // 3. 解决与判断
    if A[mid] == X then {
        return mid // 找到元素，返回其下标
    } else if A[mid] > X then {
        // 目标值在左半部分，递归搜索 A[low...mid-1]
        return BinarySearch_Recursive(A, X, low, mid - 1)
    } else { // A[mid] < X
        // 目标值在右半部分，递归搜索 A[mid+1...high]
        return BinarySearch_Recursive(A, X, mid + 1, high)
    }
}
```

迭代版本伪代码:

```
BinarySearch_Iterative(A, X, n) { // A是数组，X是目标值，n是数组元素个数
    low = 0 // 假设数组下标从0开始
    high = n - 1

    while low <= high {
        mid = floor((low + high) / 2)

        if A[mid] == X then {
            return mid // 找到元素，返回其下标
        } else if A[mid] > X then {
            high = mid - 1 // 目标值在左半部分
        } else { // A[mid] < X
            low = mid + 1  // 目标值在右半部分
        }
    }
    return NOT_FOUND // 循环结束仍未找到，元素不存在
}
```

3. 例子 (LEC4, p.2)

PPT (LEC4, p.2) 中给出了一个例子：在一个有序序列 `[3, 7, 11, 12, 15, 19, 24, 33, 41, 55]` 中查找 `X = 24` 。

1. 初始区间 `[3, ..., 55]` (`low=0, high=9`)
    - `mid = floor((0+9)/2) = 4` 。 `A[4] = 15` 。

- `15 < 24`，所以目标值在右半部分。新的搜索区间 `A[mid+1...high]` 即 `A[5...9]`（`low=5, high=9`）。
- 当前序列：`[19, 24, 33, 41, 55]`

2. 搜索区间 `[19, ..., 55]`（`low=5, high=9`）
   - `mid = floor((5+9)/2) = 7`。`A[7] = 33`。
   - `33 > 24`，所以目标值在左半部分。新的搜索区间 `A[low...mid-1]` 即 `A[5...6]`（`low=5, high=6`）。
   - 当前序列：`[19, 24]`
3. 搜索区间 `[19, 24]`（`low=5, high=6`）
   - `mid = floor((5+6)/2) = 5`。`A[5] = 19`。
   - `19 < 24`，所以目标值在右半部分。新的搜索区间 `A[mid+1...high]` 即 `A[6...6]`（`low=6, high=6`）。
   - 当前序列：`[24]`
4. 搜索区间 `[24]`（`low=6, high=6`）
   - `mid = floor((6+6)/2) = 6`。`A[6] = 24`。
   - `24 == 24`，找到目标值，返回下标 `6`。

4. 效率分析 (LEC4, p.2)

- **时间复杂度**：
  - 每次比较后，搜索区间的规模大约减半。
  - 设问题规模为 n（数组元素个数）。
  - 递归关系式可以表示为：$T(n) = T(n/2) + \Theta(1)$（一次比较和一些算术运算是常数时间 $\Theta(1)$）。
  - 根据主方法 (Master Theorem) 的情况2（因为 $f(n) = \Theta(1) = \Theta(n^0)$ 且 $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$），解为 $T(n) = \Theta(\log n)$。
  - 因此，二分搜索的时间复杂度在**最好情况、最坏情况和平均情况都是 $\Theta(\log n)$**。
    - 最好情况：第一次就找到中间元素，比较1次，$\Theta(1)$，但这是指找到元素的操作次数，整个算法的增长阶仍然是 $\log n$ 的。更准确地说，找到元素的操作次数在最好情况下是1次，最坏是 $\log n$ 次。
- **空间复杂度**：
  - **迭代版本**：只需要常数个额外变量（`low`，`high`，`mid`），所以空间复杂度是 $O(1)$。
  - **递归版本**：需要额外的栈空间来存储递归调用的信息。栈的深度在最坏情况下是 $O(\log n)$，所以空间复杂度是 $O(\log n)$。

5. 备考要点

- **简答题考点**：
  - 二分搜索的前提条件是什么？（数组必须有序）
  - 描述二分搜索的基本思想（分治：比较中间，缩小范围）。
  - 二分搜索的时间复杂度是多少？为什么？
  - 递归实现和迭代实现的二分搜索在空间复杂度上有什么区别？
- **算法分析题考点**：
  - 给定一个有序数组和目标值，手动模拟二分搜索的查找过程，列出每一步的 `low`，`high`，`mid` 以及比较结果。
  - 写出二分搜索的时间复杂度递归式并求解。
  - 可能会问一些边界条件的处理，例如当 `low` 和 `high` 如何更新，循环或递归的终止条件。
- **设计证明题考点**：
  - 写出二分搜索的递归或迭代伪代码。

- 证明二分搜索算法的正确性（通常使用循环不变量或递归归纳）。
- 讨论二分搜索在查找不存在元素时的行为。

6. 总结

二分搜索是一种非常基础且重要的查找算法，它利用了数据的有序性，通过分治策略将查找时间显著降低到对数级别。理解其工作原理、前提条件以及递归和迭代两种实现方式都非常关键。

分治策略详解之三 (A)：标准定义与朴素分治矩阵乘法 (LEC4)

1. 标准矩阵乘法：定义与复杂度 (LEC4, p.2)

- **问题描述**：给定两个 $n \times n$ 的方阵 X 和 Y，计算它们的乘积矩阵 $Z = XY$ 。
- **数学公式**：结果矩阵 Z 中的任意一个元素 Z_{ij}（位于第 i 行，第 j 列）是通过将 X 矩阵的第 i 行与 Y 矩阵的第 j 列的对应元素相乘后再求和得到的 。
  $Z_{ij} = \sum_{k=1}^{n} X_{ik}Y_{kj}$
  这个公式适用于 $1 \le i, j \le n$。
- **计算开销分析**：
  - **计算单个元素 Z_{ij}**：
    - 需要进行 n 次标量乘法 ($X_{ik} \times Y_{kj}$，其中 k 从1到 n) 。
    - 需要进行 $n-1$ 次标量加法 (将 n 个乘积加起来)。
  - **计算整个矩阵 Z**：
    - 结果矩阵 Z 共有 $n \times n = n^2$ 个元素。
    - **总乘法次数**：n^2 (元素个数) $\times$ n (每次计算的乘法数) $= n^3$ 次。
    - **总加法次数**：n^2 (元素个数) $\times$ $(n-1)$ (每次计算的加法数) $= n^3 - n^2$ 次。
  - **时间复杂度**：由于主导操作是 n^3 次乘法（或加法），因此标准（或称为暴力法）矩阵乘法的时间复杂度为 $\Theta(n^3)$ 。
  - **一般情况**：对于一个 $a \times m$ 的矩阵与一个 $m \times q$ 的矩阵相乘，其开销是 $a \cdot m \cdot q$ 次乘法 。

2. 基于分治思想的朴素矩阵乘法 (LEC4, p.2)

我们可以尝试应用分治策略来计算两个 $n \times n$ 矩阵 X 和 Y 的乘积 Z。为了简化分析，通常假设 n 是2的幂（如果不是，可以通过向矩阵中填充0行和0列使其维度达到最接近的2的幂次，但这会使实际操作更复杂，所以理论分析时常做此假设）。

- **步骤1：分解 (Divide)**
  - 将输入的 $n \times n$ 矩阵 X 和 Y 以及输出矩阵 Z 均均匀地划分成4个 $(n/2) \times (n/2)$ 的子矩阵 。
    $X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}, Z = \begin{pmatrix} I & J \\ K & L \end{pmatrix}$
  - 其中 A, B, C, D 是 X 的子矩阵，E, F, G, H 是 Y 的子矩阵，I, J, K, L 是 Z 的子矩阵，它们都是 $(n/2) \times (n/2)$ 规模的。这个分解步骤本身只需要常数时间（概念上的划分）。
- **步骤2：解决 (Conquer)**
  - 根据矩阵乘法的分块规则，结果矩阵 Z 的各个子矩阵可以通过以下公式计算：

        - $I = AE + BG$
        - $J = AF + BH$
        - $K = CE + DG$
        - $L = CF + DH$
    - 观察这些公式，我们会发现计算 I, J, K, L 共需要：
        - **8 次** $(n/2) \times (n/2)$ 规模的**矩阵乘法**（例如 $AE, BG, AF, BH, CE, DG, CF, DH$）。这些乘法将通过递归调用分治算法来解决。
        - **4 次** $(n/2) \times (n/2)$ 规模的**矩阵加法**（例如 $AE + BG$ 中的加法）。一次 $(n/2) \times (n/2)$ 矩阵的加法需要 $(n/2)^2 = n^2/4$ 次标量加法，因此其时间复杂度为 $\Theta(n^2)$。
- **步骤3：合并 (Combine)**
    - 将通过上述公式计算得到的子矩阵 I, J, K, L 组合（或者说直接赋值）到结果矩阵 Z 的相应位置。这个步骤的开销主要就是4次矩阵加法的开销，即 $\Theta(n^2)$。
- **效率分析 (LEC4, p.2)**
    - 令 $T(n)$ 表示计算两个 $n \times n$ 矩阵相乘所需的时间。
    - **分解**步骤的开销：$\Theta(1)$。
    - **解决**步骤的开销：
        - 8次递归调用，每次处理规模为 $n/2$ 的问题，因此是 $8T(n/2)$。
    - **合并**步骤（包括计算子矩阵时的加法）的开销：4次 $(n/2) \times (n/2)$ 矩阵加法，每次 $\Theta((n/2)^2) = \Theta(n^2)$，总共是 $4 \times \Theta(n^2/4) = \Theta(n^2)$。
    - 因此，该分治算法的运行时间递归关系式为：
      $T(n) = 8T(n/2) + \Theta(n^2)$
    - **使用主方法 (Master Theorem) 求解**：
        - $a = 8$ (子问题数量)
        - $b = 2$ (子问题规模是原问题规模的 $1/b$)
        - $f(n) = \Theta(n^2)$ (分解和合并步骤的开销)
        - 计算 $n^{\log_b a} = n^{\log_2 8} = n^3$。
        - 比较 $f(n) = n^2$ 与 $n^{\log_b a} = n^3$。
        - 我们发现 $f(n) = n^2 = O(n^{3-\epsilon})$，其中 $\epsilon = 1 > 0$。这符合主方法的**情况1**。
    - 根据主方法情况1，该递归式的解为 $T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$。
    - **结论**：这种基于分块的朴素分治算法，其时间复杂度仍然是 $\Theta(n^3)$，与标准的迭代算法相比并没有实现渐进意义上的改进 。它只是将问题递归地分解了，但子问题的数量（8个）相对于问题规模的缩小程度（$n/2$）来说还是太多了。

3. 备考要点（针对此部分）

- **简答题考点**：
    - 标准矩阵乘法的元素计算公式。
    - 标准矩阵乘法的时间复杂度。
    - 描述基于分治思想的朴素矩阵乘法的分解和合并步骤。
    - 为什么朴素的分治矩阵乘法算法没有改进时间复杂度？（因为有8次递归调用）
- **算法分析题考点**：
    - 写出朴素分治矩阵乘法的时间复杂度递归关系式。
    - 使用主方法求解该递归关系式。

- **设计证明题考点**:
  - 可能会要求描述分块矩阵乘法的公式。

分治策略详解之三 (B)：Strassen 矩阵乘法算法 (LEC4)

Strassen 算法通过一种巧妙的代数技巧，减少了分治过程中子矩阵乘法的次数，从而获得了比 $\Theta(n^3)$ 更好的时间复杂度。

1. 背景与动机

正如我们上一部分分析的，朴素的分治矩阵乘法需要进行8次规模为 $(n/2) \times (n/2)$ 的子矩阵乘法，这导致其时间复杂度仍然是 $\Theta(n^3)$。Strassen 在1969年发现，可以通过增加一些矩阵加减法的次数，来换取子矩阵乘法次数的减少，具体来说，是将8次乘法减少到7次。

2. Strassen 算法的步骤 (LEC4, p.3)

假设我们要计算 $Z = XY$，其中 X, Y, Z 都是 $n \times n$ 的矩阵 (为简化，仍假设 n 是2的幂)。

- **步骤1：分解 (Divide)**
  - 与朴素分治法相同，将 X, Y, Z 均划分为4个 $(n/2) \times (n/2)$ 的子矩阵:
    $X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}, Z = \begin{pmatrix} I & J \\ K & L \end{pmatrix}$
  - 这个分解步骤的时间开销是 $\Theta(1)$。
- **步骤2：构造中间量 (Create Intermediate Matrices)**
  - 这是 Strassen 算法的核心。它定义了10个 $(n/2) \times (n/2)$ 的中间矩阵 $T_1, \ldots, T_{10}$（PPT中没有直接用 T_i 命名，而是直接给出了 S_i 的构成部分，这里我们整合一下），它们是通过对 X 和 Y 的子矩阵进行加减运算得到的。这一步需要 **10次** $(n/2) \times (n/2)$ 规模的矩阵加法或减法。
    - $T_1 = F - H$
    - $T_2 = A + B$
    - $T_3 = C + D$
    - $T_4 = G - E$
    - $T_5 = A + D$
    - $T_6 = E + H$
    - $T_7 = B - D$
    - $T_8 = G + H$
    - $T_9 = A - C$
    - $T_{10} = E + F$
  - 然后，基于这些 T_i（或者说直接基于 $A, B, ..., H$ 的加减组合），定义7个关键的乘积项 $S_1, \ldots, S_7$，这些 S_i 也是 $(n/2) \times (n/2)$ 的矩阵:
    - $S_1 = A \cdot T_1 = A(F - H)$
    - $S_2 = T_2 \cdot H = (A + B)H$
    - $S_3 = T_3 \cdot E = (C + D)E$
    - $S_4 = D \cdot T_4 = D(G - E)$

        - $S_5 = T_5 \cdot T_6 = (A+D)(E+H)$
        - $S_6 = T_7 \cdot T_8 = (B-D)(G+H)$
        - $S_7 = T_9 \cdot T_{10} = (A-C)(E+F)$
    - 计算这7个 S_i 矩阵需要 **7次递归的** $(n/2) \times (n/2)$ **矩阵乘法。**
    - 创建这些 T_i 和 S_i 的总加减法次数是固定的（10次用于 T_i 的计算，如果 S_i 的定义直接用原始子块，则这些加减法分布在 S_i 的计算中）。每个 $(n/2) \times (n/2)$ 矩阵的加减法需要 $\Theta((n/2)^2) = \Theta(n^2)$ 的时间。因此，这部分的加减法总开销是 $\Theta(n^2)$。
- **步骤3：解决 (Conquer)**
    - 递归地调用 Strassen 算法来计算上述7个乘积 $S_1, S_2, \ldots, S_7$。
- **步骤4：合并 (Combine)**
    - 用这些 S_i 矩阵通过加减运算组合出结果矩阵 Z 的四个子块 I, J, K, L:
        - $I = S_5 + S_6 + S_4 - S_2$
        - $J = S_1 + S_2$
        - $K = S_3 + S_4$
        - $L = S_1 - S_7 - S_3 + S_5$
    - 这一步需要额外的 **8次** $(n/2) \times (n/2)$ 规模的矩阵加法或减法，其总开销也是 $\Theta(n^2)$。

3. 效率分析 (LEC4, p.3)

- 令 $T(n)$ 表示使用 Strassen 算法计算两个 $n \times n$ 矩阵相乘所需的时间。
- **分解**：$\Theta(1)$。
- **构造中间量与解决**:
    - 10次 $(n/2) \times (n/2)$ 矩阵加/减法：$10 \cdot \Theta((n/2)^2) = \Theta(n^2)$。
    - 7次递归调用，每次处理规模为 $n/2$ 的问题：$7T(n/2)$。
- **合并**:
    - 8次 $(n/2) \times (n/2)$ 矩阵加/减法：$8 \cdot \Theta((n/2)^2) = \Theta(n^2)$。
- 因此，总的加减法开销是 $10 \cdot \Theta(n^2/4) + 8 \cdot \Theta(n^2/4) = \Theta(n^2)$。
- 递归关系式为:
  $T(n) = 7T(n/2) + \Theta(n^2)$
- **使用主方法 (Master Theorem) 求解**:
    - $a = 7$ (子问题数量)
    - $b = 2$ (子问题规模是原问题规模的 $1/b$)
    - $f(n) = \Theta(n^2)$ (分解和合并步骤中的非递归开销，主要是矩阵加减法)
    - 计算 $n^{\log_b a} = n^{\log_2 7}$。
    - 我们知道 $\log_2 4 = 2$ 且 $\log_2 8 = 3$，所以 $2 < \log_2 7 < 3$。具体地，$\log_2 7 \approx 2.80735$。
    - 比较 $f(n) = n^2$ 与 $n^{\log_2 7}$。
    - 由于 $2 < \log_2 7$，我们可以找到一个常数 $\epsilon = \log_2 7 - 2 > 0$，使得 $f(n) = n^2 = O(n^{\log_2 7 - \epsilon})$。这符合主方法的**情况1**。
- 根据主方法情况1，该递归式的解为 $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7})$。
- **时间复杂度**：$T(n) \approx \Theta(n^{2.808})$。
- **结论**：Strassen 算法的时间复杂度 $\Theta(n^{\log_2 7})$ 渐进地优于标准算法和朴素分治算法的 $\Theta(n^3)$。

4. 实际应用中的考虑

- **常数因子**：虽然 Strassen 算法在渐进意义上更快，但其递归结构和涉及的多次矩阵加减操作使得其常数因子比标准算法大。这意味着对于较小的 n，标准算法可能实际运行得更快。通常存在一个交叉点，只有当 n 大于这个交叉点时，Strassen 算法才开始显现优势。这个交叉点的值依赖于具体的实现和硬件平台，可能在 $n = 32$ 到 $n = 128$ 或更大。
- **数值稳定性**：Strassen 算法涉及更多的加减运算，这可能在浮点数运算中引入更大的累积误差，导致其数值稳定性不如标准算法。
- **实现复杂度**：Strassen 算法的实现比标准算法更复杂，尤其是在处理递归基准情况和非2的幂次维度的矩阵时（通常需要填充）。
- **空间复杂度**：Strassen 算法在每次递归时需要为中间矩阵（T_i 和 S_i）分配空间。如果不进行非常仔细的内存管理（例如原地计算某些加减法或复用空间），其空间复杂度可能会比朴素分治的 $\Theta(n^2)$（用于结果和临时子块）或 $O(\log n)$ 栈空间（如果子块计算能优化）要高。通常教科书分析其空间复杂度也为 $\Theta(n^2)$（如果考虑存储中间结果所需的空间）。

5. 备考要点

- **简答题考点**：
  - Strassen 算法相比朴素分治矩阵乘法的主要改进是什么？（将8次子矩阵乘法减少到7次）
  - Strassen 算法的时间复杂度是多少？它与 n^3 相比如何？
  - Strassen 算法在实际应用中可能有哪些局限性或缺点？（常数因子大，数值稳定性，实现复杂）
- **算法分析题考点**：
  - 写出 Strassen 算法的时间复杂度递归关系式。
  - 使用主方法求解该递归关系式，并解释为什么 $n^{\log_2 7}$ 优于 n^3。
  - 可能会问到 Strassen 算法中 S_i 的具体表达式（完全默写的可能性不大，但理解其是通过7次乘法和多次加减法构造的是关键）。
- **设计证明题考点**：
  - 核心在于验证 Strassen 算法中如何用7个乘积项 $S_1, \ldots, S_7$ 和若干加减法来正确地构造出结果矩阵的四个子块 I, J, K, L。例如，证明 $I = S_5 + S_6 + S_4 - S_2 = (A + D)(E + H) + (B - D)(G + H) + D(G - E) - (A + B)H$ 展开后确实等于 $AE + BG$。这需要一定的代数推导。

6. 总结

Strassen 算法是算法设计中一个里程碑式的成果，它展示了通过巧妙的代数变换可以突破看似固有的计算复杂度。尽管它有实际应用中的一些限制，但它启发了后续更多关于快速矩阵乘法的研究，这些研究已经将理论上的时间复杂度进一步降低到接近 $\Theta(n^{2.37})$ 的水平。

分治策略详解之四：大整数乘法 (Large Integer Multiplication) (LEC4)

1. 问题背景与标准算法复杂度

- **问题**：计算两个 n 位大整数 X 和 Y 的乘积。
- **标准算法（类似小学竖式乘法）**：

- 如果将一个 n 位数与另一个 n 位数相乘，大致需要进行 n^2 次个位数乘法和类似数量级的加法。
- 因此，标准算法的时间复杂度是 $\Theta(n^2)$ 。

2. 基于分治思想的朴素大整数乘法 (LEC4, p.3)

我们可以尝试用分治法来解决大整数乘法。假设两个 n 位整数 X 和 Y（为简化，设 n 是偶数，可以将每个数分为高位和低位，每部分 $n/2$ 位）。

1. **分解 (Divide)**:
   - 将 n 位整数 X 分为高位 a 和低位 b，即 $X = a \cdot 2^{n/2} + b$。
   - 将 n 位整数 Y 分为高位 c 和低位 d，即 $Y = c \cdot 2^{n/2} + d$。
   - 其中 a, b, c, d 都是 $n/2$ 位的整数。
   - $2^{n/2}$ 表示乘以2的 $n/2$ 次方，相当于左移 $n/2$ 位（在二进制表示下）。
2. **解决 (Conquer)**:
   - 计算它们的乘积 XY:
     $XY = (a \cdot 2^{n/2} + b)(c \cdot 2^{n/2} + d)$
     $XY = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd$
   - 这个表达式需要计算 **4 个** $n/2$ 位整数的乘积：ac, ad, bc, bd。这些乘积可以通过递归调用大整数乘法算法来完成。
   - 此外，还需要进行几次加法运算和移位运算（乘以 2^n 或 $2^{n/2}$）。加法和移位操作相对于乘法来说，其复杂度较低（对于 n 位数加法是 $\Theta(n)$，移位也是 $\Theta(n)$）。

   伪代码 (MULT(X,Y), LEC4, p.3):

```
MULT(X,Y) {
    if |X| = |Y| = 1 then // 如果是一位数
        return X * Y    // 直接相乘
    else
        // a, b 是 X 的高低位; c, d 是 Y 的高低位
        // n 是 X (或 Y) 的位数
        P1 = MULT(a,c)
        P2 = MULT(a,d)
        P3 = MULT(b,c)
        P4 = MULT(b,d)
        return P1 * 2^n + (P2 + P3) * 2^(n/2) + P4
}
```

3. **合并 (Combine)**:
   - 将计算得到的 ac, ad, bc, bd 按照上述公式通过移位（乘以 2^n 和 $2^{n/2}$）和加法组合起来。
   - 3次 $O(n)$ 规模的加法和2次 $O(n)$ 规模的移位操作，总共是 $\Theta(n)$ 的开销。

- **效率分析 (LEC4, p.4)**:
  - 设 $T(n)$ 是计算两个 n 位整数相乘的时间。
  - 递归关系式为：$T(n) = 4T(n/2) + \Theta(n)$（4次递归调用，加上 $\Theta(n)$ 的加法和移位开销）。
  - 根据主方法 (Master Theorem)：$a = 4, b = 2, f(n) = \Theta(n)$。
    $n^{\log_b a} = n^{\log_2 4} = n^2$。

由于 $f(n) = n = O(n^{2-\epsilon})$（例如 $\epsilon = 1$），这符合主方法的情况1。
- 因此，$T(n) = \Theta(n^2)$ 。
- **结论**：这种朴素的分治大整数乘法算法并没有改进标准算法的时间复杂度，仍然是 $\Theta(n^2)$ 。

3. Karatsuba 算法 (更优的分治算法) (LEC4, p.4)

Karatsuba 在1960年（PPT中写为1962年 ）发现了一种方法，可以将上述4次 $n/2$ 位整数的乘法减少到3次，从而改进算法的整体效率。

1. **分解 (Divide)**：同上，将 $X = a \cdot 2^{n/2} + b$ 和 $Y = c \cdot 2^{n/2} + d$。
2. **利用高斯等式 (Gauss's Trick)**：
   - 原始的乘积展开式为 $XY = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd$。
   - 关键在于中间项 $ad + bc$ 的计算。Karatsuba 注意到：
     $(a + b)(c + d) = ac + ad + bc + bd$
   - 因此，可以得到：
     $ad + bc = (a + b)(c + d) - ac - bd$
   - 这样，我们只需要计算以下 **3 个** $n/2$ 位（或 $n/2 + 1$ 位，因为 $a + b$ 可能多一位）整数的乘积：
     - $P_1 = ac$
     - $P_2 = bd$
     - $P_3 = (a + b)(c + d)$
   - 然后，$XY = P_1 \cdot 2^n + (P_3 - P_1 - P_2) \cdot 2^{n/2} + P_2$。
3. **解决 (Conquer)**：
   - 递归地计算 $P_1 = \text{MULT}(a, c)$。
   - 递归地计算 $P_2 = \text{MULT}(b, d)$。
   - 计算 $a + b$ 和 $c + d$ (这需要 $O(n)$ 的加法)。
   - 递归地计算 $P_3 = \text{MULT}(a + b, c + d)$。

   伪代码 (Karatsuba - MULT(X,Y), LEC4, p.4):

```
MULT(X,Y) {
    if |X| = |Y| = 1 then
        return X * Y
    else
        // a, b 是 X 的高低位; c, d 是 Y 的高低位
        // n 是 X (或 Y) 的位数
        A1 = MULT(a,c)      // P1
        A2 = MULT(b,d)      // P2
        // 计算 (a+b) 和 (c+d) 需要 O(n) 的加法
        A3 = MULT((a+b), (c+d)) // P3
        // 计算 (A3 - A1 - A2) 需要 O(n) 的减法
        return A1 * 2^n + (A3 - A1 - A2) * 2^(n/2) + A2
}
```

4. **合并 (Combine)**：
   - 执行移位操作和若干次加减法操作。具体来说：
     - 计算 $A_3 - A_1 - A_2$ 需要两次减法。

    - 最终组合需要两次移位和两次加法。
  - 这些操作的开销都是 $\Theta(n)$。

- **效率分析 (LEC4, p.4)**:
  - 递归关系式为：$T(n) = 3T(n/2) + \Theta(n)$（3次规模为 $n/2$ 的递归调用，加上 $\Theta(n)$ 的加减法和移位开销）。
  - 根据主方法 (Master Theorem)：$a = 3, b = 2, f(n) = \Theta(n)$。
    $n^{\log_b a} = n^{\log_2 3}$。
    我们知道 $\log_2 3 \approx 1.58496$。
    比较 $f(n) = n$ 与 $n^{\log_2 3}$。
    由于 $1 < \log_2 3$，我们可以找到一个常数 $\epsilon = \log_2 3 - 1 > 0$，使得 $f(n) = n = O(n^{\log_2 3 - \epsilon})$。这符合主方法的**情况1**。
  - 因此，$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$ 。
  - **结论**：Karatsuba 算法的时间复杂度 $\Theta(n^{\log_2 3})$ 渐进地优于标准算法和朴素分治算法的 $\Theta(n^2)$。

4. 备考要点

- **简答题考点**:
  - 传统大整数乘法（小学乘法）的时间复杂度。
  - 描述朴素分治大整数乘法的分解方式及其为何没有改进复杂度。
  - Karatsuba 算法的核心思想是什么？（利用高斯等式将4次乘法降为3次）
  - Karatsuba 算法的时间复杂度是多少?
- **算法分析题考点**:
  - 写出朴素分治大整数乘法或 Karatsuba 算法的时间复杂度递归关系式。
  - 使用主方法求解这两个递归关系式。
  - 可能会要求手动模拟 Karatsuba 算法计算两个较小（例如2位或4位）整数的乘积，展示 P_1, P_2, P_3 的计算过程。
- **设计证明题考点**:
  - 证明 Karatsuba 算法中 $XY = P_1 \cdot 2^n + (P_3 - P_1 - P_2) \cdot 2^{n/2} + P_2$ 的正确性，即展开验证其等价于 $ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd$。

5. 总结

大整数乘法问题展示了分治策略如何通过巧妙的代数变换（如Karatsuba算法中的高斯等式）来减少关键递归步骤的次数，从而实现算法复杂度的显著降低。Karatsuba 算法是将复杂度从 $\Theta(n^2)$ 降至 $\Theta(n^{\log_2 3})$ 的一个经典例子，对于非常大的整数，这种改进是非常可观的。更高级的算法如 Toom-Cook 和 Schönhage-Strassen 算法可以进一步降低大整数乘法的复杂度。

分治策略详解之五：最近点对问题 (Closest Pair Problem) (LEC4)

最近点对问题是计算几何中的一个基本问题，目的是在给定的 n 个点的集合中，找到欧几里得距离最小的两个点。分治法为此问题提供了一个高效的解决方案。

1. 问题描述与距离计算 (LEC4, p.4)

- **输入 (Input)**：二维平面上的 n 个点，每个点 P_i 由其坐标 (x_i, y_i) 给出 [cite: 105]。
- **输出 (Output)**：找出这 n 个点中，距离最小的一对点（或它们之间的最小距离） [cite: 105]。
- **距离**：点 $P_1(x_1, y_1)$ 和 $P_2(x_2, y_2)$ 之间的欧几里得距离定义为：
  $d(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ [cite: 107]。
  在算法实现中，为了避免开方运算带来的精度问题和计算开销，通常比较距离的平方 $d^2(P_1, P_2) = (x_1 - x_2)^2 + (y_1 - y_2)^2$。

2. 蛮力法 (Brute-Force Approach)

最直接的方法是计算所有 $C(n, 2) = n(n-1)/2$ 对点之间的距离，然后找出最小值。此方法的时间复杂度为 $\Theta(n^2)$ [cite: 107]。当 n 很大时，这种方法的效率较低。

3. 一维情况的启示 (LEC4, p.5)

如果所有点都位于一条直线上（一维空间），问题可以简化：

1. 将所有点按其坐标排序（例如，升序）。这需要 $\Theta(n \log n)$ 时间 [cite: 107]。
2. 然后，线性扫描排序后的点，计算每对相邻点之间的距离，并找出最小值。这需要 $\Theta(n)$ 时间 [cite: 107]。
   因此，一维最近点对问题可以在 $\Theta(n \log n)$ 时间内解决 [cite: 107]。这提示我们预排序可能对二维问题也有帮助。

4. 基于分治策略的二维最近点对算法

该算法遵循分治法的三个主要步骤：分解、解决、合并。

- **A. 预处理 (Preprocessing)** (LEC4, p.7)
  i. 创建两个数组（或列表）：
    - P_x：包含所有 n 个点，并按照它们的 x 坐标升序排列。
    - P_y：包含所有 n 个点，并按照它们的 y 坐标升序排列。
  ii. 这个预排序步骤需要 $\Theta(n \log n)$ 时间。这些排好序的列表将在后续的递归调用中被有效地使用，以避免在每个子问题中都进行重复排序。
- **B. 算法核心** `Closest-Pair(P_x, P_y)`
  这个递归函数接收按 x 坐标排序的 P_x 和按 y 坐标排序的 P_y（它们包含相同的点集）。
  i. **分解 (Divide)** (LEC4, p.5, 7):
    - **基准情况**：如果 P_x 中的点数 n 小于或等于某个小常数（例如，$n \leq 3$），则直接使用蛮力法计算这些点之间的最近点对距离，并返回该距离和对应的点对。
    - **分割**：否则，找到 P_x 中的中位数点（根据 x 坐标）。设此点的 x 坐标为 x_{mid}。这条垂直线 L：$x = x_{mid}$ 将点集 P 分为两个子集：
      - P_L：P_x 中 x 坐标小于或等于 x_{mid} 的点。
      - P_R：P_x 中 x 坐标大于 x_{mid} 的点。
    - 根据 P_L 和 P_R 来构建四个新的排序列表，供递归调用使用：
      - P_{Lx}：P_L 中的点按 x 坐标排序（即 P_x 的前半部分）。
      - P_{Ly}：P_L 中的点按 y 坐标排序。

- P_{Rx}: P_R 中的点按 x 坐标排序（即 P_x 的后半部分）。
- P_{Ry}: P_R 中的点按 y 坐标排序。
  这四个列表可以在 $\Theta(n)$ 时间内通过线性扫描 P_x 和 P_y 得到。例如，遍历 P_y 中的每个点，如果其 x 坐标 $\leq x_{mid}$，则加入 P_{Ly}，否则加入 P_{Ry}。P_{Lx} 和 P_{Rx} 可以直接从 P_x 划分得到。

ii. **解决 (Conquer)** (LEC4, p.5, 7):
- 递归地调用 `Closest-Pair` 解决两个子问题:
  - $(\delta_L, \text{pair}_L) = \text{Closest-Pair}(P_{Lx}, P_{Ly})$
  - $(\delta_R, \text{pair}_R) = \text{Closest-Pair}(P_{Rx}, P_{Ry})$
- 令 $\delta = \min(\delta_L, \delta_R)$。当前找到的最小距离是 δ，对应的点对是 δ_L 和 δ_R 中较小者对应的点对。

iii. **合并 (Combine)** (LEC4, p.6, 7):
- 现在需要考虑是否存在一对点 (p_l, p_r)，其中 $p_l \in P_L$ 且 $p_r \in P_R$，使得它们之间的距离小于当前已知的最小距离 δ。
- **关键观察1：带状区域**: 如果这样的点对 (p_l, p_r) 存在，那么 p_l 和 p_r 都必须位于以分割线 $L(x = x_{mid})$ 为中心，宽度为 2δ 的垂直带状区域内。即，它们的 x 坐标必须满足 $x_{mid} - \delta < x(p_l) \leq x_{mid}$ 和 $x_{mid} < x(p_r) < x_{mid} + \delta$ [cite: 107]。
- **构建 S_y 列表**: 创建一个新的列表 S_y，包含所有 x 坐标在区间 $(x_{mid} - \delta, x_{mid} + \delta)$ 内的点，并且这些点在 S_y 中仍然是按照 y 坐标升序排列的。这可以通过线性扫描原始的 P_y 列表（在 $\Theta(n)$ 时间内）来高效完成。
- **关键观察2：有限比较**: 对于 S_y 中的每个点 p，我们不需要将其与 S_y 中的所有其他点进行比较。一个重要的几何结论是（LEC4, p.6, Figure 33.11），对于 S_y 中的任意点 p，任何可能与 p 形成距离小于 δ 的点 q（假设 q 在 S_y 中位于 p 之后，即 $y(q) \geq y(p)$），必定满足 $y(q) - y(p) < \delta$。进一步地，可以证明这样的点 q 在 S_y 列表中必定位于 p 之后的**常数个**位置之内（PPT中提到的是最多检查后续的7个点）[cite: 108, 109]。
  - *证明概要*: 考虑一个以点 p 为参考，在 2δ 宽的带状区域内，高度为 δ 的矩形（即一个 $2\delta \times \delta$ 的区域，其中 y 坐标范围是 $[y(p), y(p) + \delta)$）。任何与 p 距离小于 δ 的点 q 必须落在这个矩形内。这个 $2\delta \times \delta$ 的矩形可以被划分为两个 $\delta \times \delta$ 的正方形。在一个 $\delta \times \delta$ 的正方形区域内，如果所有点对之间的距离都至少为 δ，那么这个正方形内最多只能有4个点 [cite: 109]。由于点 q 必须在 p 的同侧或另一侧（相对于分割线 L），且 $y(q) - y(p) < \delta$，通过更细致的"鸽巢原理"分析，可以得出我们只需要检查 p 之后的固定少数几个点。
- **扫描 S_y**: 遍历 S_y 中的每个点 p。对于每个点 p，只检查其在 S_y 列表中后续的至多7个点 q。计算 $d(p, q)$。如果 $d(p, q) < \delta$，则更新 $\delta = d(p, q)$，并记录下新的最近点对。
- 由于 S_y 的长度最多为 n，且每个点只与常数个点比较，所以合并步骤的时间复杂度是 $\Theta(n)$。

5. 效率分析 (LEC4, p.7)

- **预处理排序**: $\Theta(n \log n)$。
- **递归函数 `Closest-Pair` 的运行时间 $T(n)$** (不包括初始预排序):
  - 分解: $\Theta(n)$ (主要是为了构建 P_{Ly} 和 P_{Ry})。
  - 解决: $2T(n/2)$。
  - 合并: $\Theta(n)$ (构建 S_y 和扫描 S_y)。
  - 递归关系式: $T(n) = 2T(n/2) + \Theta(n)$。
- 根据主方法 (Master Theorem) 情况2，解为 $T(n) = \Theta(n \log n)$。
- **总时间复杂度**: 由于预处理也是 $\Theta(n \log n)$，所以整个最近点对算法的时间复杂度是 $\Theta(n \log n)$ [cite: 111]。

6. 备考要点

- **简答题考点**:
  - 描述二维最近点对问题的分治算法的基本步骤。
  - 在合并步骤中，为什么只需要考虑一个宽度为 2δ 的带状区域内的点？[cite: 107]
  - 在带状区域 S_y 中，对于每个点 p，为什么只需要检查其后（按y坐标排序）固定数量（常数个）的点？简述其几何原理。
  - 最近点对分治算法的时间复杂度是多少？
- **算法分析题考点**:
  - 写出最近点对分治算法的时间复杂度递归关系式并求解。
  - 分析算法中各个主要步骤（如预处理、构建子问题的输入列表 P_{Ly}, P_{Ry}、构建 S_y、扫描 S_y 中的点对）的时间复杂度。
- **设计证明题考点**:
  - 描述最近点对算法的伪代码，特别是合并步骤的详细逻辑。
  - 可能会要求对“为何在 S_y 中每个点只需比较后续常数个点”这一结论进行更详细的解释或基于给定图示的推导。

7. 总结

最近点对算法是分治策略在解决几何问题时的一个优雅且高效的范例。它通过将问题规模减半递归求解，并在合并步骤中巧妙地利用几何特性将跨区域的比较限制在線性时间内完成，最终实现了 $\Theta(n \log n)$ 的整体时间复杂度，远优于蛮力法的 $\Theta(n^2)$。算法成功的关键在于有效的预排序以及合并步骤中对带状区域内点对比较的优化。

随机化算法详解之一：概览与雇佣问题 (LEC5)

1. 随机化算法简介 (LEC5, p.1)

- **确定性算法的平均情况分析**:
  - 算法本身是确定的 。
  - 平均情况分析通常需要对输入的分布做出假设（例如，假设所有可能的输入排列等概率出现）。
  - 这种分析的缺点是:
    - 可能存在明确的、会导致最坏情况性能的输入 。
    - 如果输入分布的假设不成立，分析结果可能不准确 。
    - “对手”或不良数据可能利用算法的确定性来构造导致最坏情况的输入 。
- **算法随机化 (Randomization)**:
  - 通过在算法执行过程中引入随机决策，使得算法对于任何特定的输入，其行为都具有随机性。
  - **目标**：消除或减少“坏”输入的影响，使得算法的期望运行时间（对于任何输入）都比较好。
  - **优点**:
    - 不存在明确的导致最坏情况的输入（因为算法的行为是随机的）。
    - 削弱了对手通过构造特定输入来攻击算法的能力 。
    - 通常分析的是**期望运行时间**，这个期望是针对算法内部的随机选择而言的，而不是针对输入分布 。

2. 雇佣问题 (The Hiring Problem) (LEC5, p.1)

这是一个经典的用来说明随机化算法优势的问题。

- **问题描述** (LEC5, p.1):
  - 你需要通过一个中介机构招聘一名新的办公室助理。
  - 中介每天向你推荐一位候选人。你每天面试一位。
  - 如果你面试的候选人比你当前雇佣的助理更优秀，你就会解雇当前的助理，并雇佣这位新的、更优秀的候选人。
  - 面试和雇佣都有成本。
- **代价模型** (LEC5, p.1):
  - **面试代价 (c_i)**：面试每位候选人需要支付给中介的费用（假设这个费用不高）。
  - **雇佣代价 (c_h)**：每次雇佣一位新助理需要支付的额外费用（例如给中介的额外佣金、培训成本等，假设这个费用远高于面试代价 $c_h \gg c_i$）。
  - 目标：在雇佣到最佳人选的前提下，最小化总的雇佣代价（主要是指由多次雇佣行为产生的 c_h 代价）。
- **算法 `Hire-Assistant(A)`** (LEC5, p.1):

```
Hire-Assistant(A) { // A是候选人序列，按面试顺序排列
    current_assistant = an infinitely useless dummy assistant // 初始时有一个虚拟的最差助理
    number_of_hires = 0
    for i = 1 to n { // n是候选人总数
        interview candidate A[i]
        if A[i] is better qualified than current_assistant then {
            current_assistant = A[i]
            hire A[i]
            number_of_hires = number_of_hires + 1
        }
    }
    return current_assistant (and number_of_hires)
}
```

- **代价分析 (确定性算法)** (LEC5, p.2):
  - 总面试代价是固定的：$n \cdot c_i$（因为所有候选人都要面试）。
  - 总雇佣代价：$m \cdot c_h$，其中 m 是雇佣的总次数。
  - 总支出：$n \cdot c_i + m \cdot c_h$。
  - 我们关心的是期望的雇佣次数 $E[m]$。
  - **最好情况**：第一个面试的候选人就是最好的。只雇佣1次。总代价 $n \cdot c_i + 1 \cdot c_h$。
  - **最坏情况**：候选人按能力严格递增的顺序出现。每次面试都会导致一次新的雇佣。共雇佣 n 次。总代价 $n \cdot c_i + n \cdot c_h$。这在 c_h 很大时代价非常高。
  - **平均情况分析（假设输入随机排列）**：
    - 为了分析期望的雇佣次数 $E[m]$，我们需要对输入的排列做出假设。
    - 假设所有 $n!$ 种候选人能力排名排列都是等概率出现的（一致性随机排列 Uniform Random Permutation）。
    - **指示变量 (Indicator Random Variable)**：
      - 定义事件 E_j 为“第 j 个面试的候选人被雇佣”。

    - 指示变量 $X_j = I(E_j)$，如果事件 E_j 发生则 $X_j = 1$，否则 $X_j = 0$。
    - 总雇佣次数 $m = X = \sum_{j=1}^{n} X_j$。
    - $E[X] = E[\sum_{j=1}^{n} X_j] = \sum_{j=1}^{n} E[X_j]$。
    - $E[X_j] = P(\text{candidate } j \text{ is hired})$。
    - 第 j 个候选人被雇佣的条件是：在他之前的 $j-1$ 个候选人中，他的能力是最强的。由于是随机排列，在前 j 个人中，任何一个人是这 j 个人中最优秀的概率都是 $1/j$。
    - 所以，$P(\text{candidate } j \text{ is hired}) = 1/j$ 。
    - 因此，$E[X_j] = 1/j$。
    - $E[m] = \sum_{j=1}^{n} (1/j) = H_n$ (调和级数第 n 项)。
    - 我们知道 $H_n \approx \ln n + \gamma$ （γ 是欧拉常数），所以 $E[m] = \Theta(\ln n)$ （PPT中写为 $O(\log n)$，通常自然对数和以2为底的对数在渐进分析中只差一个常数因子）。
  - **期望总代价**：$n \cdot c_i + (\ln n) \cdot c_h = \Theta(c_i n + c_h \ln n)$ 。这个结果比最坏情况的 $n \cdot c_h$ 好得多。

3. 随机化雇佣算法 (Randomized Hiring Algorithm) (LEC5, p.3)

上述平均情况分析依赖于输入是随机排列的假设。如果输入不是随机的，我们仍然可能遇到最坏情况。随机化算法通过在算法内部引入随机性来打破这种依赖。

- **解决办法 - 随机化** (LEC5, p.3)：
  i. 从中介那里一次性获得所有 n 位候选人的名单。
  ii. 在算法内部，将这 n 位候选人进行一次**一致性随机排列 (Uniform Random Permutation)**。
  iii. 然后按照这个随机产生的顺序依次面试这些候选人，并执行与之前相同的雇佣策略。
- **改进的雇佣算法 (Randomized-Hire-Assistant)**：

```
Randomized-Hire-Assistant(A_original_list) {
    n = A_original_list.length
    A_permuted = Randomly_Permute(A_original_list) // 产生一个随机排列
    // 接下来的步骤与 Hire-Assistant(A_permuted) 相同
    current_assistant = an infinitely useless dummy assistant
    number_of_hires = 0
    for i = 1 to n {
        interview candidate A_permuted[i]
        if A_permuted[i] is better qualified than current_assistant then {
            current_assistant = A_permuted[i]
            hire A_permuted[i]
            number_of_hires = number_of_hires + 1
        }
    }
    return current_assistant (and number_of_hires)
}
```

- **期望代价分析** (LEC5, p.3)：
  - 由于算法内部自己制造了随机排列，所以无论原始输入的候选人名单是怎样的，面试的顺序都是随机的。
  - 因此，之前基于随机排列的平均情况分析现在适用于这个随机化算法的**期望情况分析**。
  - 期望雇佣次数仍然是 $\Theta(\ln n)$。
  - 期望总代价是 $\Theta(c_i n + c_h \ln n)$。

- **关键区别**：这个期望值不再依赖于输入是否有某种特定的分布，而是算法自身随机行为的结果。对于任何输入，其期望代价都是这个值。

4. 备考要点

- **简答题考点**：
  - 什么是随机化算法？它与确定性算法的平均情况分析有何不同？
  - 随机化算法有哪些优点？（例如，避免最坏情况输入，削弱对手）
  - 描述雇佣问题的场景和代价模型。
  - 在确定性的雇佣算法中，最好和最坏的输入序列分别是什么样的？对应的雇佣次数是多少？
  - 如何通过随机化改进雇佣算法？随机化后的期望雇佣次数是多少？
  - 指示变量在分析随机化算法中的作用。
- **算法分析题考点**：
  - 使用指示变量推导在随机排列输入下，雇佣问题的期望雇佣次数为 $H_n = \Theta(\ln n)$。
  - 分析随机化雇佣算法的期望总代价。
- **设计证明题考点**：
  - 可能会要求描述随机化版本的雇佣算法。

随机化算法详解之二：随机排列生成 (LEC5)

PPT (LEC5, p.3) 中提到了两种生成随机排列的方法：

1. 排序法排列 (Permuting by Sorting / `Permute-By-Sorting`) (LEC5, p.3)

- **思想**：
  i. 为原始数组 `A` (长度为 n) 中的每个元素 $A[i]$ 生成一个随机的优先级 $P[i]$。
  ii. 根据这些随机生成的优先级对数组 `A` 进行排序。
  iii. 排序后的数组 `A` 即为一个随机排列。
- **伪代码** (`Permute-By-Sorting(A)`) (LEC5, p.3)：

```
Permute-By-Sorting(A) {
    n = A.length
    Let P[1..n] be a new array // 存储随机优先级

    // 1. 为每个元素生成随机优先级
    for i = 1 to n {
        P[i] = RANDOM(1, n^3) // 生成一个在 [1, n^3] 范围内的随机整数作为优先级
    }

    // 2. 根据优先级 P 对数组 A 进行排序
    //    例如，可以将 (A[i], P[i]) 作为一对进行排序，以 P[i] 为键
    Sort array A, using array P as sort keys
}
```

- **为什么选择优先级范围为 $[1, n^3]$?**
  - 选择这个范围是为了确保所有优先级**几乎肯定**是唯一的。如果有两个或多个元素获得了相同的随机优先级，那么它们之间的相对顺序将由排序算法的稳定性（或不稳定性）决定，这可能会导致某些排列出现的概率略有不同。
  - 如果优先级都在 $[1, n^3]$ 范围内随机选择，那么两个优先级相同的概率大约是 $1/n^3$。对于 n 个元素，所有优先级都唯一的概率非常高。
  - PPT (LEC5, p.3) 给出了一个定理：**排序法排列产生的序列是一致性随机排列**（假设所有优先级唯一）。
- **效率分析** (LEC5, p.3):
  - 生成 n 个随机优先级需要 $\Theta(n)$ 时间。
  - 根据优先级对数组 `A` 进行排序，如果使用基于比较的排序算法（如合并排序或堆排序），需要 $\Theta(n \log n)$ 时间。
  - 因此， `Permute-By-Sorting` 的总时间复杂度是 $\Theta(n \log n)$。

2. 换位法排列 (Permuting In-Place / `Permute-In-Place`) (LEC5, p.4)

这是一种更常用且更高效的原地生成随机排列的方法，也称为 Fisher-Yates shuffle (或 Knuth shuffle)。

- **思想**:
  i. 迭代 n 次（或 $n-1$ 次，取决于实现）。
  ii. 在第 i 次迭代时（i 从 1 到 n），从数组的第 i 个元素到最后一个元素（即 $A[i \dots n]$）的范围内随机选择一个元素。
  iii. 将选中的随机元素与 $A[i]$ 进行交换。
  - 这样，在第一次迭代后，$A[1]$ 就被随机确定了；第二次迭代后，$A[2]$ 在剩余元素中被随机确定，以此类推。
- **伪代码** (`Permute-In-Place(A)`) (LEC5, p.4):

```
Permute-In-Place(A) {
    n = A.length
    // 循环从第一个元素到倒数第二个元素 (或者到最后一个元素，取决于随机范围)
    // PPT中的版本是到 n-1，随机范围是 [i, n]
    for i = 1 to n { // 或者 for i = 1 to n-1
        // 从 A[i] 到 A[n] 的范围内随机选择一个下标 j
        j = RANDOM(i, n)
        // 交换 A[i] 和 A[j]
        swap A[i] with A[j]
    }
}
```

  - **`RANDOM(i, n)`**：表示生成一个在 i 和 n 之间（包含 i 和 n）的随机整数。
- **正确性证明概要（循环不变量）**：
  - 可以证明，在 `for` 循环的每次迭代开始之前（对于 i），子数组 $A[1 \dots i-1]$ 包含 $i-1$ 个元素，它们是原始数组中 $i-1$ 个元素的一个均匀随机排列。
  - **初始化**：当 $i = 1$ 时，子数组 $A[1 \dots 0]$ 为空，不变量成立。
  - **保持**：假设在第 i 次迭代开始前，$A[1 \dots i-1]$ 是前 $i-1$ 个元素的一个均匀随机排列。在第 i 次迭代中，算法从 $A[i \dots n]$ 中随机选择一个元素与 $A[i]$ 交换。这使得 $A[i]$ 位置上的元素是以 $1/(n-i+1)$

的概率从剩余的 $n-i+1$ 个元素中选出的。结合归纳假设，可以证明 $A[1\ldots i]$ 在迭代后成为 i 个元素的均匀随机排列。
    - **终止**：当 $i=n+1$（或循环结束）时，$A[1\ldots n]$ 包含原始数组 n 个元素的均匀随机排列。
    - PPT (LEC5, p.4) 给出了一个定理：**换位法排列产生的序列是一致性随机排列**。
- **效率分析** (LEC5, p.4)：
    - 循环执行 n 次（或 $n-1$ 次）。
    - 在每次迭代中，`RANDOM` 函数调用和 `swap` 操作都花费 $\Theta(1)$ 时间。
    - 因此，`Permute-In-Place` 的总时间复杂度是 $\Theta(n)$。这比排序法排列的 $\Theta(n\log n)$ 更优。

3. 随机数生成器 `RANDOM(a,b)`

- 两种方法都依赖于一个能够生成指定范围内均匀随机整数的函数 `RANDOM(a,b)`。
- 在实践中，通常使用伪随机数生成器 (PRNG)，它们能产生在统计上接近随机的序列。

4. 备考要点

- **简答题考点**：
    - 生成一个均匀随机排列意味着什么？（每种排列出现的概率相等，为 $1/n!$）
    - 描述“排序法排列”（Permute-By-Sorting）的基本思想和时间复杂度。
    - 描述“换位法排列”（Permute-In-Place / Fisher-Yates shuffle）的基本思想和时间复杂度。
    - 为什么“换位法排列”通常优于“排序法排列”？（时间复杂度更低）
- **算法分析题考点**：
    - 分析 `Permute-By-Sorting` 和 `Permute-In-Place` 的时间复杂度。
    - 可能会要求手动模拟 `Permute-In-Place` 对一个小数组的随机排列过程（给定随机数序列）。
- **设计证明题考点**：
    - 可能会要求写出其中一种随机排列算法的伪代码。
    - 理解 `Permute-In-Place` 算法正确性证明的关键思想（循环不变量）。

随机化算法详解之三：随机化快速排序 (Randomized QuickSort) (LEC5)

1. 快速排序回顾 (LEC5, p.4)

在讨论随机化版本之前，我们先简要回顾一下标准快速排序（在之前的排序算法板块已详细介绍）：

- **分治策略**：
    i. **分解 (Divide)**：选择一个主元，将数组 `A[p..r]` 划分为两个（可能为空的）子数组 `A[p..q-1]` 和 `A[q+1..r]`，使得 `A[p..q-1]` 中的所有元素都小于等于 `A[q]`（主元），而 `A[q+1..r]` 中的所有元素都大于等于 `A[q]`。下标 `q` 是主元在划分后的位置。
    ii. **解决 (Conquer)**：递归地调用快速排序算法对子数组 `A[p..q-1]` 和 `A[q+1..r]` 进行排序。
    iii. **合并 (Combine)**：无需操作，因为子数组是原地排序的。
- **`PARTITION(A, p, r)` 过程** (LEC5, p.4)（通常选择 `A[r]` 作为主元）：
    i. `x = A[r]` (主元)
    ii. `i = p - 1`

iii. `for j = p to r - 1`

iv. `do if A[j] <= x`

v. `then i = i + 1`

vi. `exchange A[i] with A[j]`

vii. `exchange A[i+1] with A[r]`

viii. `return i + 1`

- **性能分析回顾** (LEC5, p.4):
  - **最坏情况**：当主元总是选择到当前子数组中的最小或最大元素时，划分产生一个空子数组和一个大小为 $n-1$ 的子数组。 递归式为 $T(n) = T(n-1) + \Theta(n)$，解为 $T(n) = \Theta(n^2)$。
  - **最好情况**：当主元总是选择到当前子数组的中位数时，划分产生两个大小近似为 $n/2$ 的子数组。递归式为 $T(n) = 2T(n/2) + \Theta(n)$，解为 $T(n) = \Theta(n \log n)$。
  - **平均情况**：对于一般输入，快速排序的平均时间复杂度为 $\Theta(n \log n)$。

2. 随机化快速排序的思想 (LEC5, p.5)

为了避免固定主元选择策略（如总是选择最后一个元素）可能导致的在特定输入（如已排序数组）上的最坏情况性能，随机化快速排序引入了随机性来选择主元。

- **两种常见的随机化策略**:
  i. **随机选择主元**：在 `PARTITION(A, p, r)` 过程中，不再固定选择 `A[r]` 作为主元，而是从子数组 `A[p..r]` 中随机选择一个元素作为主元。然后，将这个随机选出的主元与 `A[r]` （或 `A[p]`，取决于 `PARTITION` 的具体实现）交换，再执行原来的 `PARTITION` 逻辑。
  ii. **随机打乱输入**：在排序开始之前，对整个输入数组 `A` 进行一次均匀随机排列（如使用前面讨论的 `Permute-In-Place` 算法）。之后再运行确定性版本的快速排序算法（例如，总是选择最后一个元素作为主元）。

  这两种策略都能达到相似的效果：使得主元的选择（相对于当前子数组中的元素值）是随机的，从而使得划分的平衡性在期望意义上得到保证。PPT (LEC5, p.5) 中展示的随机化快速排序伪代码采用了第一种策略（随机交换后进行划分）。
- **伪代码** **`Randomized-Quicksort(A, p, r)`** (LEC5, p.5，略有调整以匹配其描述):

```
Randomized-Quicksort(A, p, r) {
    if p < r then {
        // 1. 随机选择一个主元：从 A[p..r] 中随机选择一个下标 i_rand
        //    然后将 A[i_rand] 与 A[r] 交换 (这样 PARTITION 仍然可以使用 A[r] 作为主元)
        i_rand = RANDOM(p, r)
        swap A[i_rand] with A[r] // PPT中的示意图是A[r]与A[random(p,r)]交换

        // 2. 进行划分
        q = PARTITION(A, p, r) // PARTITION 过程与之前相同

        // 3. 递归排序
        Randomized-Quicksort(A, p, q-1)
        Randomized-Quicksort(A, q+1, r)
    }
}
```

PPT (LEC5, p.5) 的图示中， `swap A[r] with A[random(p,r)]` 是在 `Partition` 之前执行的，这是一个常见的随机化方式。

3. 随机化快速排序的期望运行时间分析 (LEC5, pp.5-6)

随机化使得算法的运行时间不再仅仅依赖于输入数据，而是也依赖于随机数生成器的输出。我们分析的是其**期望运行时间**。

- **核心思想**：由于主元的选择是随机的，那么任何一个元素都有相同的概率被选为下一轮划分的主元。可以证明，这种随机性使得极不平衡的划分出现的概率很小，而比较平衡的划分出现的概率较大。
- **运行时间与比较次数** (LEC5, p.5)：
  - 快速排序的运行时间主要由 `PARTITION` 中的比较操作决定。
  - 设随机变量 X 表示在整个 `Randomized-Quicksort` 执行过程中所做的比较总次数。
  - 则算法的运行时间为 $O(n + X)$ (n 是数组大小，用于其他开销）。
  - 我们的目标是计算 $E[X]$。
- **期望比较次数 $E[X]$ 的分析** (LEC5, pp.5-6)：
  i. 设排序后的元素序列为 $z_1 < z_2 < \cdots < z_n$。
  ii. 定义指示变量 $X_{ij} = I\{z_i \text{ is compared to } z_j\}$，即如果元素 z_i 和 z_j 在算法执行过程中被比较了，则 $X_{ij} = 1$，否则为0。
  iii. 总比较次数 $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$。
  iv. $E[X] = E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$。
  v. $E[X_{ij}] = P(z_i \text{ is compared to } z_j)$。
  vi. **关键洞察**：两个元素 z_i 和 z_j (假设 $z_i < z_j$) 会被比较，当且仅当在集合 $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ 中，第一个被选为**主元**的元素恰好是 z_i 或 z_j。
    - 如果 Z_{ij} 中某个元素 z_k (其中 $i < k < j$) 首先被选为这一区段的主元，那么 z_i 和 z_j 将会被划分到 z_k 的不同两侧（或一侧但 z_k 介于它们之间），它们之后就不会再被相互比较了。
    - 如果 z_i 首先被选为 Z_{ij} 集合中的主元，那么它会与 Z_{ij} 中所有其他元素（包括 z_j）进行比较。
    - 如果 z_j 首先被选为 Z_{ij} 集合中的主元，那么它会与 Z_{ij} 中所有其他元素（包括 z_i）进行比较。

vii. 由于主元是随机选择的，集合 Z_{ij}（包含 $j-i+1$ 个元素）中的任何一个元素都有相同的概率 $(1/(j-i+1))$ 成为该集合中第一个被选为主元的元素。
viii. 因此，$P(z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}) = \frac{2}{j-i+1}$ 。
ix. 所以，$E[X_{ij}] = \frac{2}{j-i+1}$。
x. $E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$。
xi. 通过代换 $k = j-i+1$ (或者直接分析这个和式)，可以证明 $E[X] = O(n \log n)$ 。
    - $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k}$。
    - 对于固定的 i，内层和 $\sum_{k=2}^{n-i+1} \frac{1}{k} \le \sum_{k=1}^{n} \frac{1}{k} = H_n = O(\log n)$。
    - 所以 $E[X] \le \sum_{i=1}^{n-1} 2 \cdot O(\log n) = O(n \log n)$。
- **结论**：随机化快速排序的期望运行时间为 $O(n \log n)$。

4. 备考要点

- **简答题考点**：
    - 为什么需要随机化快速排序？（为了避免确定性快速排序在特定输入下的最坏情况性能）
    - 随机化快速排序的两种主要策略是什么？
    - 随机化快速排序的期望时间复杂度是多少？
    - 随机化是否改变了快速排序的最坏情况时间复杂度？（不改变，最坏情况仍然是 $O(n^2)$，但发生的概率极小）
- **算法分析题考点**：
    - 描述随机化快速排序的期望比较次数分析过程中的关键步骤和指示变量的定义。
    - 解释为什么任意两个元素 z_i 和 z_j 被比较的概率是 $2/(j-i+1)$。
    - 能够推导出期望总比较次数 $E[X] = O(n \log n)$ 的求和式。
- **设计证明题考点**：
    - 描述随机化快速排序的伪代码（特别是随机选择主元的部分）。
    - 可能会要求对期望比较次数分析的某个步骤进行更详细的论证。

5. 总结

随机化快速排序通过在主元选择中引入随机性，使得算法对于任何输入都能有很大概率表现出接近最优的 $\Theta(n \log n)$ 的性能，其期望时间复杂度为 $\Theta(n \log n)$。这使得它在实践中成为一种非常高效和常用的排序算法，因为它通常比其他 $\Theta(n \log n)$ 算法（如合并排序、堆排序）具有更小的常数因子，并且可以实现良好的原地性。

随机化算法详解之四：随机算法分类及实例

1. 随机算法分类 (LEC5, p.6)

- **Las Vegas 算法 (Las Vegas Algorithms)**：
    - 这类算法**总是给出正确的结果**，或者在无法给出正确结果时（虽然这种情况很少见或可以通过增加运行时间来避免）会报告失败（无解）。
    - 其**运行时间是随机变量**。对于相同的输入，每次运行的时间可能不同，因为算法内部的随机选择不同。
    - 我们通常分析其**期望运行时间**。

    - **例子**：随机化快速排序可以被看作是 Las Vegas 算法，因为它总是产生正确的排序结果，但其运行时间（比较次数）依赖于随机的主元选择，期望运行时间是 $\Theta(n \log n)$，最坏情况是 $\Theta(n^2)$（但发生概率极小）。
- **Monte Carlo 算法 (Monte Carlo Algorithms)**:
    - 这类算法的**运行时间通常是确定的**（或者是有一个确定的上界）。
    - 但是，它们可能会产生**不正确的结果**，尽管产生错误结果的**概率很小**。
    - 这个错误概率通常可以通过重复执行算法多次或调整算法参数来降低到任意期望的程度。
    - **例子**：用于测试大数是否为素数的 Miller-Rabin 算法（PPT中未直接提及，但属于此类）。PPT中将要讨论的"测试串相等性"算法也是一个典型的 Monte Carlo 算法。

简答题考点:

- 比较 Las Vegas 算法和 Monte Carlo 算法的特点（在结果确定性、运行时间确定性、错误概率方面）。
- 各举一个例子（如随机化快速排序是Las Vegas，串相等性测试是Monte Carlo）。

2. 实例：测试串的相等性 (Testing String Equality) (LEC5, p.6)

- **问题描述** (LEC5, p.6):
    - 假设有两个通信方 A 和 B。A 有一个很长的比特串 x，B 有一个很长的比特串 y。
    - 目标：判断 x 是否等于 y ($x = y$?)。
- **传统确定性方法** (LEC5, p.6):
    - A 将整个串 x 发送给 B，B 比较接收到的 x 与自己的 y 是否相等。
    - 或者 B 将 y 发送给 A，A 进行比较。
    - **缺点**：如果比特串非常长，传输整个串的代价（时间、带宽）可能非常高，造成资源浪费 。
- **随机化方法：基于"指纹"(Fingerprinting) 的 Monte Carlo 算法** (LEC5, p.6):
    - **核心思想**：不直接比较整个长串，而是为每个长串计算一个短得多的"指纹"（checksum 或 hash value）。然后只比较这两个指纹。
        - 如果指纹不同，则可以断定原串一定不同。
        - 如果指纹相同，则**很可能**原串相同，但存在一个（希望很小的）概率是原串不同但指纹恰好碰撞了（即所谓的"哈希碰撞"）。
    - **特点**：节约了资源（传输指纹比传输原串快得多），但理论上不能100%保证结果的正确性 。
- **"指纹"生成方法** (LEC5, p.6):
    i. 将比特串 w 解释为一个（非常大的）整数 $I(w)$。例如，如果 $w = w_k w_{k-1} \dots w_1 w_0$，则 $I(w) = \sum_{i=0}^{k} w_i 2^i$。
    ii. 选择一个素数 p。
    iii. 通过取模运算生成指纹：$I_p(w) = I(w) \pmod p$。
- **测试串相等性算法步骤** (LEC5, p.6):
    i. 通信方 A 从一个预先确定的、小于某个上界 M 的素数集合中**随机选择一个素数** p。
    ii. A 计算 $I_p(x) = I(x) \pmod p$。
    iii. A 将选择的素数 p 和计算出的指纹 $I_p(x)$ 发送给 B。
    iv. B 收到 p 和 $I_p(x)$后，计算自己串 y 的指纹 $I_p(y) = I(y) \pmod p$。
    v. B 比较 $I_p(x)$ 和 $I_p(y)$:

    - 如果 $I_p(x) \neq I_p(y)$，则 B 确定 $x \neq y$。
    - 如果 $I_p(x) = I_p(y)$，则 B 认为 $x = y$（但存在错误的可能性）。
- **失败概率分析 (Error Probability Analysis)** (LEC5, p.7):
    - 算法失败（即 $x \neq y$ 但 $I_p(x) = I_p(y)$）的条件是：$I(x) \neq I(y) \pmod p$ 不成立，即 $I(x) \equiv I(y) \pmod p$，或者说 p 整除 $|I(x) - I(y)|$。
    - 令 $N = |I(x) - I(y)|$。如果 $x \neq y$，则 $N > 0$。
    - 假设 x 和 y 是 n 位的比特串，那么 $I(x)$ 和 $I(y)$ 的值都小于 2^n。因此，$N = |I(x) - I(y)| < 2^n$。
    - 一个数 $N < 2^n$ 最多有多少个不同的素因子?
        - 即使 N 是由最小的素数 $2, 3, 5, \ldots$ 相乘得到，由于 $2 \cdot 3 \cdot 5 \cdot \cdots \cdot p_k < 2^n$，素因子的数量 k 也不会太多。
        - 一个更宽松的上界是：N 最多有 n 个不同的素因子（因为 2^n 是 N 的一个上界，而 2^n 只有 n 个素因子2）。实际上，一个小于 2^n 的数，其不同素因子的个数远小于 n。一个更紧的界是，不同素因子的数量 k 满足 $2^k \leq N < 2^n$，所以 $k < n$。PPT中给出的结论是基于数论中关于一个整数 N 的不同素因子个数 $\omega(N)$ 的上界。$\omega(N) \leq \log_2 N$。所以如果 $N < 2^n$，则 $\omega(N) < \log_2(2^n) = n$。
        - PPT中使用了 $\pi_N(N)$ 来表示 N 的不同素因子的数量（这里 π_N 不是指素数计数函数 $\pi(N)$）。更常见的记号是 $\omega(N)$。
    - 设 $\pi(M)$ 是小于 M 的不同素数的总个数（这是标准的素数计数函数）。
    - 算法失败的概率是 $P(\text{error}) = \frac{\text{number of prime divisors of } N \text{ that are } < M}{\text{total number of primes } < M}$。
    - PPT (LEC5, p.7) 中给出的失败概率上界是：
      $P(\text{error}) \leq \frac{\omega(N)}{\pi(M)} \leq \frac{n}{\pi(M)}$ (这里用 n 作为 $\omega(N)$ 的一个非常宽松的上界，因为 $N < 2^n$)。
      更准确地，如果 $N = |I(x) - I(y)| \neq 0$，那么它最多有 $\log N \approx n \ln 2$ 个不同的素因子。
      根据素数定理，$\pi(M) \approx M / \ln M$。
    - 如果我们选择 M 足够大（例如 $M = n^2 \ln(n^2)$），使得 $\pi(M)$ 远大于 n，那么这个错误概率就可以做得很小。例如，如果 M 取得足够大，使得 $\pi(M) > n^c$ for some $c > 1$，则错误概率可以降到 $O(1/n^{c-1})$。
- **如何降低错误概率**:
    - 选择一个更大的范围 M 来挑选素数 p。
    - 独立地重复该过程 k 次，使用 k 个不同的随机素数。只有当所有 k 次指纹都相同时才判断 $x = y$。此时，如果 $x \neq y$，则 k 次都出错的概率是 $(P(\text{single error}))^k$，可以降到非常小。

3. 备考要点

- **简答题考点**:
    - Las Vegas 算法和 Monte Carlo 算法的主要区别和特点。
    - 描述测试串相等性的随机化（指纹）方法的基本思想。
    - 为什么基于指纹的串相等性测试属于 Monte Carlo 算法? （因为它可能出错）
    - 在串相等性测试中，选择素数 p 的作用是什么？错误可能在什么情况下发生?
    - 如何降低 Monte Carlo 算法（如串相等性测试）的错误概率?
- **算法分析题考点**:
    - 理解串相等性测试中错误概率的来源（即 $I(x) \neq I(y)$ 但 $I(x) \equiv I(y) \pmod p$）。
    - 对错误概率的上限分析（LEC5, p.7）可能不会要求完整推导，但应理解其大致思路：错误是由于随机选择的素数 p 恰好是 $|I(x) - I(y)|$ 的一个素因子。
- **设计证明题考点**:

- 描述测试串相等性的随机化算法步骤。

随机化算法详解之四：随机算法分类及实例 (续)

接续上一部分，我们已经讨论了 Las Vegas 算法和 Monte Carlo 算法的分类，并详细介绍了使用指纹法进行串相等性测试作为 Monte Carlo 算法的一个实例。这里对串相等性测试的错误概率分析做进一步的说明和总结。

指纹法串相等性测试的进一步说明

- **核心挑战**：当 $I(x) \neq I(y)$ 时，我们希望 $I_p(x) \neq I_p(y)$。算法出错当且仅当 $I(x) \neq I(y)$ 但 $I_p(x) = I_p(y)$。这等价于 $I(x) - I(y) \neq 0$ 且 $I(x) - I(y) \equiv 0 \pmod{p}$，也就是说，随机选择的素数 p 恰好是 $N = |I(x) - I(y)|$ 的一个素因子。
- **错误概率的控制**：
  - 令 $N = |I(x) - I(y)|$。如果 $x \neq y$，则 $N > 0$。
  - 假设 x 和 y 是长度为 L 的比特串，则 $I(x) < 2^L$ 且 $I(y) < 2^L$，因此 $N < 2^L$。
  - 一个正整数 N 最多有 $\log_2 N$ 个不同的素因子（因为如果 $N = p_1^{a_1} \dots p_k^{a_k}$，其中 p_i 是不同的素数且 $a_i \geq 1$，那么 $N \geq p_1 \dots p_k \geq 2^k$，所以 $k \leq \log_2 N$）。
  - 因此，如果 $N < 2^L$，则 N 最多有 L 个不同的素因子。
  - 假设我们从一个包含 $\pi(M)$ 个小于 M 的素数的集合中随机选择一个素数 p。
  - 如果 $x \neq y$，算法出错的概率是 $P(\text{error}) = \frac{\text{number of distinct prime factors of } N \text{ in the chosen set of primes}}{\text{total number of primes in the chosen set (i.e., } \pi(M))}$。
  - $P(\text{error}) \leq \frac{\omega(N)}{\pi(M)} \leq \frac{L}{\pi(M)}$。 (这里 $\omega(N)$ 是 N 的不同素因子个数， L 是 N 的位数的一个非常宽松的上界)。
  - 根据素数定理，$\pi(M) \approx M / \ln M$。如果我们选择 M 使得 $\pi(M)$ 远大于 L（例如，选择 M 使得 $M / \ln M \approx L^c$ for some $c > 1$），那么错误概率就可以变得非常小。例如，如果选择 M 足够大，使得 $\pi(M) \geq 2L^2/\epsilon$，则错误概率可以控制在 $\epsilon/(2L)$ 以下。
- **实践中的选择**：
  - 在实际应用中，通常会选择一个足够大的素数 p（或一个素数范围 M），使得碰撞概率（即错误概率）低于某个可接受的阈值。
  - 例如，如果使用一个64位的随机指纹（相当于模一个非常大的数，或者使用多项式指纹模一个不可约多项式），碰撞概率会非常低。

总结与备考提示 (针对串相等性测试)

- **Monte Carlo 特性**：串相等性测试是一个典型的 Monte Carlo 算法，因为它速度快（只传输和比较短指纹），但有（可控的）错误概率。
- **错误来源**：错误发生在 $x \neq y$ 但它们的指纹 $I_p(x)$ 和 $I_p(y)$ 恰好相同时，即 p 是 $|I(x) - I(y)|$ 的一个素因子。
- **降低错误率**：可以通过选择更大的素数范围 M（从而有更多素数可选，降低 p 恰好是特定数因子的概率）或多次独立测试（使用不同的随机素数 p）来显著降低错误概率。
- **简答题**：
  - 描述基于指纹的串相等性测试方法。
  - 解释为何这种方法是 Monte Carlo 算法。

    - 错误是如何产生的？如何降低错误概率？
- **分析题**：
    - 可能不会要求严格的数论证明，但应理解错误概率与 $|I(x) - I(y)|$ 的素因子数量以及所选素数范围大小的关系。

选择与统计算法详解之一：求序列中最大和最小值 (LEC6)

1. 问题描述

- **输入**：一个包含 n 个（通常是互不相同的）元素的序列 A。
- **输出**：序列中的最小值和/或最大值。

2. 求最小值 (或最大值) (LEC6, p.2)

- **算法 `Minimum(A)`** (LEC6, p.2)：

```
Minimum(A) {
    n = A.length
    min_val = A[1] // 假设数组下标从1开始
    for i = 2 to n {
        if A[i] < min_val then {
            min_val = A[i]
        }
    }
    return min_val
}
```

- **思想**：初始化一个 `min_val` 为序列的第一个元素，然后遍历序列的其余部分，如果遇到比当前 `min_val` 更小的元素，则更新 `min_val` 。
- **比较次数**：在最坏情况下（例如，序列是降序排列的），需要进行 $n-1$ 次比较。在最好情况下（例如，序列是升序排列的，或者第一个元素就是最小的），也需要 $n-1$ 次比较来确认。因此，总比较次数是 $n-1$ 次。
- **时间复杂度**：$\Theta(n)$。
- 求最大值的算法与此类似，只需将比较符 `<` 改为 `>` 。

3. 同时求最大值和最小值 (LEC6, pp.2-3)

一个直接的方法是分别调用 `Minimum(A)` 和 `Maximum(A)` ，这将需要 $(n-1) + (n-1) = 2n-2$ 次比较。PPT (LEC6, p.2) 中也展示了这种思路，先将数组元素两两比较分成较小组 B 和较大组 C，然后分别在 B 中找最小，C 中找最大。

- **简单方法 (LEC6, p.2)**：
    i. 将数组 A 中的元素两两配对进行比较。对于每对 $(A[2i-1], A[2i])$：

    - 如果 $A[2i-1] \leq A[2i]$，则将 $A[2i-1]$ 放入候选最小值集合 B，将 $A[2i]$ 放入候选最大值集合 C。
    - 否则，将 $A[2i]$ 放入 B，将 $A[2i-1]$ 放入 C。
  ii. 这一步需要 $n/2$ 次比较（假设 n 是偶数）。
  iii. 然后在集合 B (大小为 $n/2$) 中找到最小值，需要 $n/2-1$ 次比较。
  iv. 在集合 C (大小为 $n/2$) 中找到最大值，需要 $n/2-1$ 次比较。
  v. 总比较次数为 $n/2 + (n/2-1) + (n/2-1) = 3n/2 - 2$ 次比较。 (如果 n 是奇数，第一个元素可以同时作为初始的最小和最大值，然后处理剩下的 $n-1$ 个元素，比较次数为 $3(n-1)/2$。)
- **能否更快?** $\Theta(n)$ 是显然的下界，因为每个元素至少要被"看"一遍。$3n/2-2$ 次比较已经是一个很好的结果了。
- **更优化的同时求最大最小值的算法思路**:
  - 目标：减少比较次数到接近 $3n/2$。
  - **方法**:
    a. 如果 n 是奇数，将第一个元素同时设为当前的 `min_val` 和 `max_val`。然后从第二个元素开始，成对处理。
    b. 如果 n 是偶数，比较前两个元素，将较小的设为 `min_val`，较大的设为 `max_val`。然后从第三个元素开始，成对处理。
    c. 对于后续成对的元素 $(A[i], A[i+1])$:
      - **首先比较 $A[i]$ 和 $A[i+1]$** (1次比较)。
      - 然后将较小者与当前的 `min_val` 比较 (1次比较)。
      - 将较大者与当前的 `max_val` 比较 (1次比较)。
      - 这样，每处理一对元素，需要进行3次比较。
  - **总比较次数**:
    - 如果 n 是奇数：初始比较0次。处理 $(n-1)/2$ 对元素，每对3次比较。总共 $3(n-1)/2$ 次。
    - 如果 n 是偶数：初始比较1次。处理 $(n-2)/2$ 对元素，每对3次比较。总共 $1 + 3(n-2)/2 = (2+3n-6)/2 = (3n-4)/2$ 次。
    - 两种情况都可以概括为大约 $3n/2$ 次比较。例如，对于 n 个元素，比较次数是 $\lceil 3n/2 \rceil - 2$。
  - **PPT (LEC6, p.3)** 中的 `Min-Max(A)` 伪代码实际上是之前描述的将元素分到B、C两组再分别找最小最大的方法，其比较次数也是 $n/2 + (n/2-1) + (n/2-1) = 3n/2 - 2$。

4. 备考要点

- **简答题考点**:
  - 描述从一个序列中找到最小（或最大）元素的基本算法及其比较次数。
  - 描述一种同时找到一个序列中最大和最小元素的高效方法，并说明其比较次数。
  - 为什么同时找最大最小值的比较次数可以少于分别找再相加的次数？（因为成对比较后，一个元素只需要和当前最大值比，另一个只需要和当前最小值比，节省了比较）。
- **算法分析题考点**:
  - 分析给定伪代码（如 `Minimum(A)` 或 `Min-Max(A)`）的比较次数和时间复杂度。
- **设计证明题考点**:
  - 可能会要求设计一个算法来同时找到最大值和最小值，并分析其效率。

选择与统计算法详解之二：选择第k小元素

选择问题，也称为顺序统计量问题，是指在一个包含 n 个（通常不排序的）元素的集合中，找出第 k 小的元素。其中 $1 \leq k \leq n$。

- 当 $k = 1$ 时，是找最小值。
- 当 $k = n$ 时，是找最大值。
- 当 $k = \lfloor (n+1)/2 \rfloor$（下中位数）或 $k = \lceil (n+1)/2 \rceil$（上中位数）时，是找中位数。

1. 基于排序的简单方法

最直接的方法是先对整个序列进行排序（例如使用合并排序或堆排序，时间复杂度为 $\Theta(n \log n)$），然后直接取出排序后数组中第 k 个位置的元素。这种方法虽然简单，但效率不是最优的。我们的目标是找到比 $\Theta(n \log n)$ 更快的方法，最好是线性时间 $\Theta(n)$。

2. 随机选择算法（`RANDOMIZED-SELECT`）(LEC6, p.3)

这种算法的思想类似于随机化快速排序。它通过随机选择一个主元，并利用快速排序中的 `PARTITION` 过程来缩小查找范围。

- **核心思想**：
  i. **分解 (Divide)**：从待查找的子数组 `A[p..r]` 中随机选择一个元素作为主元，并调用 `PARTITION`（或其随机化版本 `RANDOMIZED-PARTITION`）将子数组围绕主元进行划分。设主元划分后位于下标 `q`。
  ii. **解决 (Conquer)**：
    - 比较我们想要查找的第 i 小元素（相对于原始数组而言的第 i 小，或者是在当前子数组 `A[p..r]` 中相对于 `p` 的第 $k' = i - p + 1$ 小）与主元的位置关系。
    - 令主元是当前子数组 `A[p..r]` 中的第 $k_{pivot} = q - p + 1$ 小的元素。
    - **情况一**：如果我们要找的第 i 小元素恰好是主元本身（即 i (相对于整个数组的秩) 等于 q (主元在整个数组中的下标，假设下标从1开始)，或者说，在子数组 `A[p..r]` 中要找的秩 k' 等于主元在子数组中的秩 k_{pivot}），那么就找到了，直接返回 `A[q]`。
    - **情况二**：如果要找的第 i 小元素比主元小（即 $i < q$ 或 $k' < k_{pivot}$），则递归地在主元左边的子数组 `A[p..q-1]` 中查找第 i 小（或第 k' 小）元素。
    - **情况三**：如果要找的第 i 小元素比主元大（即 $i > q$ 或 $k' > k_{pivot}$），则递归地在主元右边的子数组 `A[q+1..r]` 中查找第 $i - q$ 小（或第 $k' - k_{pivot}$ 小，相对于新子数组的起始位置）元素。
  iii. **合并 (Combine)**：无需操作，因为递归调用直接返回结果。
- **伪代码** `RANDOMIZED-SELECT(A, p, r, i)` (LEC6, p.3) (查找数组A中从下标p到r范围内第i小的元素，这里的i是指在整个数组A中的第i小，但需要在递归中调整)
  - **PPT中 `i` 的含义**：PPT的伪代码中 `i` 代表的是“在当前子数组 `A[p..r]` 中，我们要找的是第 `i` 小的元素”。 这个 `i` 在递归调用时会变化。

```
RANDOMIZED-SELECT(A, p, r, i) { // 找 A[p..r] 中的第 i 小元素 (1 <= i <= r-p+1)
    1. if p == r then // 如果子数组只有一个元素
    2.     return A[p]

    3. // 调用 RANDOMIZED-PARTITION 将 A[p..r] 划分，并返回主元的下标 q
    // RANDOMIZED-PARTITION 会随机选择一个主元与 A[r] 交换，然后执行标准的 PARTITION
    q = RANDOMIZED-PARTITION(A, p, r)

    4. k = q - p + 1 // k 是主元 A[q] 在子数组 A[p..r] 中的秩 (即第 k 小)

    5. if i == k then // 如果要找的第 i 小恰好是主元
    6.     return A[q]
    7. else if i < k then // 如果要找的第 i 小在主元左边
    8.     return RANDOMIZED-SELECT(A, p, q-1, i) // 在左子数组中继续找第 i 小
    9. else // i > k, 如果要找的第 i 小在主元右边
    10.    return RANDOMIZED-SELECT(A, q+1, r, i-k) // 在右子数组中找第 (i-k) 小
}
```

  - **注意**： `RANDOMIZED-PARTITION` 与快速排序中的类似，它首先在 `A[p..r]` 中随机选择一个元素，将其与 `A[r]` 交换，然后调用标准的 `PARTITION(A,p,r)` 过程。
- **效率分析** (LEC6, p.3):
  - **最坏情况**：尽管随机化使得持续选到最差主元的概率很低，但理论上最坏情况仍然可能发生（例如，每次随机选择都恰好是当前子数组的最小或最大元素）。此时，问题规模每次只减1， `PARTITION` 操作需要 $\Theta(n)$ 时间。
    - $T(n) = T(n-1) + \Theta(n)$，解为 $T(n) = \Theta(n^2)$。
  - **期望情况 (Average/Expected Case)**：可以证明， `RANDOMIZED-SELECT` 的期望运行时间是线性的，即 $E[T(n)] = \Theta(n)$。
    - **直观理解**：由于主元是随机选择的，平均来说，它会将数组划分得比较均衡。即使不完全均衡，例如每次都按 1/10 和 9/10 的比例划分，递归式也会是 $T(n) \approx T(9n/10) + \Theta(n)$，根据主方法情况3（需要验证正则条件，但通常对于这类分治是满足的），其解也是 $\Theta(n)$。
    - PPT (LEC6, p.3) 中给出了期望分析的递归式 $E[T(n)] \le \frac{2}{n}\sum_{k=0}^{n-1} E[T(k)] + O(n)$ (这里的求和上界应为 $n-1$ 或者考虑最坏分割后子问题大小为 $\max(k-1, n-k)$，然后对 k 求期望)，并指出其解为 $\Theta(n)$。
    - **关键在于**：与快速排序不同， `RANDOMIZED-SELECT` 只需要在划分后的一侧进行递归，而不是两侧都递归。

3. 最坏情况线性时间选择算法 (`SELECT`) (LEC6, p.3)

为了确保在最坏情况下也能达到线性时间，Blum、Floyd、Pratt、Rivest和Tarjan在1973年提出了一个更复杂的算法，通常称为 "BFPRT" 算法或中位数的中位数算法。

- **核心思想**：通过一种巧妙的方法来选择主元 (pivot)，以保证 `PARTITION` 过程能够做出一个“足够好”的划分，使得最坏情况下的递归规模得到有效控制。
- **算法步骤 `SELECT(A, p, r, i)`** (LEC6, p.4) (查找数组A中从下标p到r范围内，全局第i小的元素):

i. **分组 (Divide into groups)**：将输入的 $n = r - p + 1$ 个元素划分为 $\lceil n/5 \rceil$ 组，每组5个元素（最后一组可能少于5个）。此步骤时间 $\Theta(n)$。

ii. **找每组的中位数 (Find medians of groups)**：对这 $\lceil n/5 \rceil$ 组中的每一组分别进行排序（例如用插入排序，因为每组只有5个元素，排序时间是常数），然后找到每一组的中位数。收集所有这些中位数。此步骤时间 $\Theta(n)$（$\lceil n/5 \rceil$ 组 $\times O(1)$ 时间/组）。

iii. **递归找中位数的中位数 (Find median of medians)**：递归调用 `SELECT` 算法，找到上一步收集到的 $\lceil n/5 \rceil$ 个中位数中的中位数，称之为 x (主元)。
   - 递归调用是 $T(\lceil n/5 \rceil)$。

iv. **划分 (Partition around x)**：使用上一步找到的中位数的中位数 x 作为主元，调用（修改版的） `PARTITION` 过程将整个数组 `A[p..r]` 围绕 x 进行划分。设 x 划分后位于下标 q。此步骤时间 $\Theta(n)$。
   - 令 $k = q - p + 1$ 为 x 在子数组 $A[p..r]$ 中的秩。

v. **解决 (Conquer by recursion or return)**：
   - 如果我们要找的第 i 小（相对于整个数组的第 i 小，需要在递归中调整 `i` 的含义，或者如PPT中， `i` 就是在当前子数组 `A[p..r]` 中要找的秩）恰好是主元 x（即 $i = k$），则返回 x。
   - 如果要找的第 i 小元素比 x 小（即 $i < k$），则递归地在主元左边的子数组 `A[p..q-1]` 中查找第 i 小元素。
   - 如果要找的第 i 小元素比 x 大（即 $i > k$），则递归地在主元右边的子数组 `A[q+1..r]` 中查找第 $i - k$ 小元素。

- **为什么选择5个元素一组？主元 x 的保证**：
  - 至少有一半的组（即至少 $\lceil \frac{1}{2} \lceil n/5 \rceil \rceil$ 组）的中位数是小于或等于 x 的。
  - 在这些组中，每组至少有3个元素是小于或等于该组中位数的（中位数本身，以及该组中比中位数小的2个元素，除非该组元素少于3个）。
  - 因此，整个数组中至少有 $3\left(\lceil \frac{1}{2} \lceil n/5 \rceil \rceil - 1\right) \approx 3n/10$ 个元素小于或等于 x。（减1是因为包含 x 本身的那一组可能只有1或2个其他元素小于等于x，如果是包含中位数的中位数的那组）。
  - 类似地，可以证明至少有约 $3n/10$ 个元素大于或等于 x。
  - 这意味着，以 x 为主元进行划分后，每一边的子数组大小**最多**约为 $n - 3n/10 = 7n/10$。
  - (PPT (LEC6, p.4) 中给出的递归子问题规模是 $T(7n/10 + 6)$，+6 是为了处理取整和边界的小常数项。)
- **效率分析 (LEC6, p.4)**：
  - 步骤1 (分组)：$\Theta(n)$。
  - 步骤2 (找组内中位数)：$\Theta(n)$。
  - 步骤3 (递归找中位数的中位数)：$T(\lceil n/5 \rceil)$。
  - 步骤4 (划分)：$\Theta(n)$。
  - 步骤5 (递归解决一侧)：最坏情况下，递归调用处理的子问题大小不超过 $7n/10 + 6$（对于足够大的 n）。所以是 $T(7n/10 + 6)$。
  - 递归关系式为：$T(n) \le T(\lceil n/5 \rceil) + T(7n/10 + 6) + \Theta(n)$。
  - 可以使用代入法证明 $T(n) = \Theta(n)$。
    - 假设 $T(k) \le ck$ 对 $k < n$ 成立。
    - $T(n) \le c(n/5 + 1) + c(7n/10 + 6) + dn$ (其中 dn 是 $\Theta(n)$ 项)
    - $T(n) \le cn/5 + c + 7cn/10 + 6c + dn = c(n/5 + 7n/10) + 7c + dn = c(9n/10) + 7c + dn = 0.9cn + 7c + dn$。
    - 我们要使 $0.9cn + 7c + dn \le cn$。
    - $7c + dn \le 0.1cn$。

        - $7c/n + d \leq 0.1c$。
        - 当 n 足够大时，$7c/n$ 趋于0。所以需要 $d \leq 0.1c$，即 $c \geq 10d$。
        - 只要选择足够大的 c (例如 $c = 20d$，并处理好足够大的 n 的边界)，就可以使归纳成立。
    - **结论**：该算法的最坏情况时间复杂度是 $\Theta(n)$。

4. 备考要点

- **简答题考点**：
    - 什么是选择问题（或第 i 个顺序统计量问题）？
    - 描述 `RANDOMIZED-SELECT` 算法的基本思想。其期望时间复杂度和最坏情况时间复杂度是多少?
    - 描述最坏情况线性时间选择算法 (`SELECT` 或 BFPRT) 的核心思想是什么？（如何保证选出一个好的主元）
    - 在 `SELECT` 算法中，为什么将元素分为5个一组？（为了保证划分的平衡性，使得递归子问题规模得到有效控制，5是满足条件的最小奇数，能保证递归式收敛到线性）。
    - `SELECT` 算法的最坏情况时间复杂度是多少?
- **算法分析题考点**：
    - 分析 `RANDOMIZED-SELECT` 的期望线性时间（可能需要理解其与快速排序分析的相似与不同之处）。
    - 写出 `SELECT` 算法的时间复杂度递归关系式 $T(n) \leq T(n/5) + T(7n/10) + \Theta(n)$。
    - 解释为什么 `SELECT` 算法中，递归子问题的规模最大约为 $7n/10$。
    - (较难) 使用代入法证明 `SELECT` 算法的递归式解为 $\Theta(n)$。
- **设计证明题考点**：
    - 描述 `RANDOMIZED-SELECT` 或 `SELECT` 算法的伪代码。
    - 可能会要求解释 `SELECT` 算法中选择主元（中位数的中位数）的过程。

5. 总结

选择问题有多种解决方案。基于排序的方法简单但效率不高 ($\Theta(n \log n)$)。 `RANDOMIZED-SELECT` 算法在期望情况下可以达到线性时间 $\Theta(n)$，非常实用。而 `SELECT` (BFPRT) 算法则保证了在最坏情况下也是线性时间 $\Theta(n)$，尽管其常数因子较大且实现更复杂，但在理论上非常重要，因为它证明了选择问题可以在线性时间内确定性地解决。

动态规划详解之一：基本思想与步骤 (LEC7)

1. 动态规划与分治法的比较 (LEC7, p.1)

- **相似之处**：
    - 两者都是通过组合子问题的解来求解原问题 。
- **不同之处**：
    - **分治法 (Divide and Conquer)**：
        - 将问题划分为**相互独立**的子问题 。
        - 递归地求解这些子问题 。
        - 然后合并子问题的解得到原问题的解 。
        - 由于子问题独立，分治法可能会重复解决相同的子问题（例如，在朴素的斐波那契数列递归实现中）。

- **动态规划 (Dynamic Programming)**:
  - 适用于子问题**不独立**，即子问题之间存在**重叠**的情况 。
  - 动态规划通过记录（通常是填表）并复用已解决子问题的解，来避免对重叠子问题的重复计算 。
  - 通常以自底向上（bottom-up）的方式计算子问题的解，从小规模的子问题开始，逐步构建到原问题的解 。

2. 动态规划适用的问题特征 (LEC7, p.1)

动态规划通常适用于求解**优化问题 (Optimization Problems)**，这些问题可能有很多个可行解，目标是找到其中具有最优值（最大值或最小值）的解 。要使用动态规划，问题一般需要具备以下两个重要性质:

1. **最优子结构 (Optimal Substructure)**:
   - 一个问题的最优解包含其子问题的最优解。换句话说，如果我们将原问题分解为子问题，并通过对子问题进行某种选择来得到原问题的解，那么这个选择必须是基于子问题的最优解才能导出原问题的最优解。
   - **例子 (LEC7, p.2，装配线调度)**: 如果从起点到装配线1的第 j 个站台 $S_{1,j}$ 的最快路径是通过其前一个站台 $S_{1,j-1}$ 到达的，那么从起点到 $S_{1,j-1}$ 的这段路径也必须是从起点到 $S_{1,j-1}$ 的最快路径 。
2. **重叠子问题 (Overlapping Subproblems)**:
   - 在问题的递归求解过程中，某些相同的子问题会被多次重复地遇到和求解。
   - 动态规划通过计算每个子问题一次，并将结果存储在一个表格中（通常是数组或哈希表），之后需要时直接查表获取，从而避免了重复计算，提高了效率。
   - **例子 (LEC7, p.3，装配线调度递归解的图示)**: 在自顶向下递归计算 $f_1[j]$ 和 $f_2[j]$ 时，许多 $f_i[k]$ (其中 $k < j$) 会被多次计算 。

3. 动态规划算法的设计步骤 (LEC7, p.1)

设计一个动态规划算法通常遵循以下四个步骤:

1. **描述最优解的结构特征 (Characterize the structure of an optimal solution)**:
   - 分析问题的最优解是如何由其子问题的最优解构成的，即证明问题具有最优子结构性质。
   - 例如，在装配线调度问题中，要找到通过第 j 个站台的最快路径，需要依赖于通过第 $j-1$ 个站台的最快路径 。
2. **递归地定义最优解的值 (Recursively define the value of an optimal solution)**:
   - 根据最优子结构，建立一个递归关系式（也称为状态转移方程），用来计算原问题最优解的值与子问题最优解的值之间的关系。
   - 例如，在装配线调度问题中，$f_1[j]$（通过装配线1第 j 个站台的最快时间）可以表示为 $min(f_1[j-1]+a_{1,j}, f_2[j-1]+t_{2,j-1}+a_{1,j})$ 。
3. **以自底向上的方式计算最优解的值 (Compute the value of an optimal solution, typically in a bottom-up fashion)**:
   - 通常使用一个表格（例如一维或二维数组）来存储子问题的解。
   - 按照子问题规模从小到大的顺序，或者按照递归关系式中依赖关系的顺序，依次计算并填充表格中的每一项。
   - 这种自底向上的计算方式避免了递归带来的额外开销，并确保在计算某个子问题的解时，它所依赖的更小子问题的解已经被计算出来并存储好了。
4. **根据计算出的信息构造一个最优解 (Construct an optimal solution from computed information)**:

- 在计算最优解的值的过程中，通常会额外记录一些信息，这些信息指示了在每一步是如何做出选择才达到了最优值的（例如，在装配线调度中，记录是从哪条线的上一个站台转移过来的）。
- 在最优解的值计算完毕后，可以根据这些记录的信息，从后向前（或从前向后）回溯，构造出实际的最优解路径或方案。

4. 备考要点

- **简答题考点**：
  - 动态规划与分治法的主要区别是什么？（子问题是否独立，是否重复计算）
  - 动态规划适用于解决哪类问题？需要具备哪两个重要性质？（优化问题；最优子结构，重叠子问题）
  - 简述动态规划算法设计的四个主要步骤。
  - 解释什么是“最优子结构”和“重叠子问题”。
- **算法分析题考点**：
  - 对于给定的问题，判断其是否具有最优子结构和重叠子问题性质，从而判断是否适合使用动态规划。
  - 能够写出问题的状态定义和状态转移方程（递归关系式）。
- **设计证明题考点**：
  - 针对具体问题，按照动态规划的四个步骤设计算法。
  - 证明所设计算法的正确性（通常包括证明最优子结构性质）。
  - 分析动态规划算法的时间和空间复杂度。

动态规划详解之二：装配线调度问题 (Assembly Line Scheduling) (LEC7)

装配线调度问题是一个经典的优化问题，旨在找出在一系列具有不同工序时间的装配站之间选择一条路径，使得通过整个装配线的总时间最短。

1. 问题描述 (LEC7, p.1)

- 我们有一个汽车厂，其中有**两条装配线**，编号为1和2。
- 每条装配线都有 n **个工序站台 (stations)**。装配线 i 的第 j 个站台表示为 $S_{i,j}$。
- 在装配线 i 的站台 $S_{i,j}$ 上完成作业所需的时间为 $a_{i,j}$。
- 汽车进入装配线 i 的初始时间（**上线时间**）为 e_i。
- 汽车从装配线 i 的第 n 个站台完成作业后离开（**下线时间**）为 x_i。
- 汽车在完成装配线 i 的第 j 个站台 $S_{i,j}$ 的作业后，可以有两种选择：
  i. 继续在**同一条装配线**上进入下一个站台 $S_{i,j+1}$。这个转移时间通常认为是0（或者已经包含在 $a_{i,j+1}$ 中）。
  ii. **转移到另一条装配线**的下一个站台 $S_{k,j+1}$ (其中 $k \neq i$)。从装配线 i 的第 j 个站台 $S_{i,j}$ 转移到装配线 k 的第 $j+1$ 个站台 $S_{k,j+1}$ 所需的时间为 $t_{i,j}$。
- **目标**：找出一条通过工厂的路径（即一系列站台的选择），使得组装一辆汽车的总时间最短。

图示 (参考 LEC7, p.1, "汽车厂两条装配线"图示):
该图示清晰地展示了两条装配线、每个站台的加工时间、上线/下线时间以及跨线转移时间。

2. 蛮力法 (Brute-Force Approach) (LEC7, p.2)

- 我们可以枚举所有可能的通过工厂的路径。
- 对于每个站台 j（从1到 n），我们都可以选择在装配线1或装配线2上。因此，共有 2^n 条不同的路径。
- 如果 n 很大，计算 2^n 条路径的代价然后比较，这种方法是不可接受的 。

3. 动态规划求解步骤

步骤1：描述最优解的结构特征 (Optimal Substructure) (LEC7, p.2)

- 假设通过工厂的最快路径经过站台 $S_{1,j}$。那么，从起点（上线）到站台 $S_{1,j}$ 的这段路径也必须是从起点到站台 $S_{1,j}$ 的最快路径。
  - **证明（反证法）**：如果存在一条从起点到 $S_{1,j}$ 的更快路径，那么我们可以用这条更快的子路径替换原最快路径中到达 $S_{1,j}$ 的部分，从而得到一条通过工厂的、比原“最快路径”更快的路径，这与原路径是最快的假设矛盾 。
- 类似地，如果最快路径经过站台 $S_{2,j}$，那么从起点到 $S_{2,j}$ 的这段子路径也必须是最快的。
- 这个性质表明问题具有**最优子结构**。

步骤2：递归地定义最优解的值 (Recursive Definition) (LEC7, p.3)

- 设 f^* 为通过整个工厂的最快时间。
- 设 $f_i[j]$ 为从起点开始，通过装配线 i 的第 j 个站台 $S_{i,j}$ 的最快时间。
- **最终解 f^* 的计算**：
  汽车在完成第 n 个站台后下线，所以：
  $f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$
- **$f_i[j]$ 的递归定义**：
  - **基本情况 (j=1)**：汽车刚进入装配线。
    $f_1[1] = e_1 + a_{1,1}$
    $f_2[1] = e_2 + a_{2,1}$
  - **递归步骤 (j $\geq$ 2)**：
    要到达装配线1的第 j 个站台 $S_{1,j}$，有两种可能的前一个站台：
    a. 从同一条线（装配线1）的第 $j-1$ 个站台 $S_{1,j-1}$ 直接过来。时间为 $f_1[j-1] + a_{1,j}$。
    b. 从另一条线（装配线2）的第 $j-1$ 个站台 $S_{2,j-1}$ 转移过来。时间为 $f_2[j-1] + t_{2,j-1} + a_{1,j}$ ($t_{2,j-1}$ 是从线2的站台 $j-1$ 转移到线1的站台 j 的时间)。
    因此，$f_1[j] = \min(f_1[j-1] + a_{1,j}, \quad f_2[j-1] + t_{2,j-1} + a_{1,j})$ 。
    同理，对于装配线2的第 j 个站台 $S_{2,j}$：
    $f_2[j] = \min(f_2[j-1] + a_{2,j}, \quad f_1[j-1] + t_{1,j-1} + a_{2,j})$。
- **重叠子问题**：如果直接使用上述递归定义自顶向下计算，会发现很多 $f_i[k]$ (其中 $k < j$) 会被重复计算多次，如 LEC7, p.3 的图示所示。这表明问题具有重叠子问题性质，适合动态规划的自底向上计算。

步骤3：以自底向上的方式计算最优解的值 (Bottom-up Computation) (LEC7, p.4)

我们可以使用两个一维数组 `f1[1..n]` 和 `f2[1..n]` 来分别存储 $f_1[j]$ 和 $f_2[j]$ 的值。我们从 $j = 1$ 开始计算，逐步到 $j = n$。

1. **初始化 (j=1)**:
   `f1[1] = e1 + a[1][1]` (这里用 `a[i][j]` 表示 $a_{i,j}$)
   `f2[1] = e2 + a[2][1]`
2. **迭代计算 (j = 2 to n)**:

```
for j = 2 to n {
    // 计算 f1[j]
    val1_stay = f1[j-1] + a[1][j]
    val1_switch = f2[j-1] + t[2][j-1] + a[1][j] // t[k][j-1] 表示从线k站台j-1转出
    f1[j] = min(val1_stay, val1_switch)

    // 计算 f2[j]
    val2_stay = f2[j-1] + a[2][j]
    val2_switch = f1[j-1] + t[1][j-1] + a[2][j]
    f2[j] = min(val2_stay, val2_switch)
}
```

3. **计算最终结果 f^***:
   `f_star = min(f1[n] + x1, f2[n] + x2)`

步骤4：根据计算出的信息构造一个最优解 (Constructing an Optimal Solution) (LEC7, p.4)

为了能够构造出实际的最快路径（即选择了哪些站台），我们需要在计算 $f_i[j]$ 的过程中记录下导致这个最小值的选择。

- 我们引入两个额外的数组 $l_1[2..n]$ 和 $l_2[2..n]$。
- $l_i[j]$ 存储的是：要使得到达装配线 i 的第 j 个站台 $S_{i,j}$ 的时间 $f_i[j]$ 最短，汽车在第 $j-1$ 个站台时应该在哪条装配线上。值可以是1或2。
  - 例如，在计算 $f_1[j]$ 时：
    - 如果 $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$，则 $l_1[j] = 1$ (表示从线1的 $S_{1,j-1}$ 过来)。
    - 否则，$l_1[j] = 2$ (表示从线2的 $S_{2,j-1}$ 转移过来)。
- 我们还需要一个变量 l^* 来记录最后一步（从第 n 个站台到下线）是在哪条装配线上使得总时间 f^* 最短。
  - 如果 $f_1[n] + x_1 \leq f_2[n] + x_2$，则 $l^* = 1$。
  - 否则，$l^* = 2$。
- **算法 `FASTEST-WAY(a, t, e, x, n)`** (LEC7, p.4):
  该伪代码整合了计算 $f_i[j]$ 和 $l_i[j]$ 以及最终 f^* 和 l^* 的过程。

```
FASTEST-WAY(a, t, e, x, n) {
    f1[1] = e[1] + a[1][1]
    f2[1] = e[2] + a[2][1]

    for j = 2 to n {
        if f1[j-1] + a[1][j] <= f2[j-1] + t[2][j-1] + a[1][j] then {
            f1[j] = f1[j-1] + a[1][j]
            l1[j] = 1
        } else {
            f1[j] = f2[j-1] + t[2][j-1] + a[1][j]
            l1[j] = 2
        }

        if f2[j-1] + a[2][j] <= f1[j-1] + t[1][j-1] + a[2][j] then {
            f2[j] = f2[j-1] + a[2][j]
            l2[j] = 2
        } else {
            f2[j] = f1[j-1] + t[1][j-1] + a[2][j]
            l2[j] = 1
        }
    }

    if f1[n] + x[1] <= f2[n] + x[2] then {
        f_star = f1[n] + x[1]
        l_star = 1
    } else {
        f_star = f2[n] + x[2]
        l_star = 2
    }
    // return f_star, l_star, l1, l2 (l1, l2用于路径回溯)
}
```

- **打印最优路径** **`PRINT-STATIONS(l, l_star, n)`** (LEC7, p.4):
  一旦我们有了 l_1, l_2 和 l^*，就可以从后向前追溯并打印出最优路径上的每个站台。

```
PRINT-STATIONS(l1, l2, l_star, n) { // l_star是最后在第n站台时所在的线
    current_line = l_star
    print "line " + current_line + ", station " + n

    for j = n downto 2 {
        if current_line == 1 then {
            current_line = l1[j] // 查看从哪条线到达的 line 1, station j
        } else { // current_line == 2
            current_line = l2[j] // 查看从哪条线到达的 line 2, station j
        }
        print "line " + current_line + ", station " + (j-1)
    }
}
```

LEC7, p.4 的图示 "line 1, station 5", "line 1, station 4", "line 1, station 3", "line 2, station 2", "line 1, station 1" 就是一个回溯打印的例子。

4. 效率分析

- **时间复杂度**:
  - 初始化 $f_1[1], f_2[1]$ 需要 $\Theta(1)$ 时间。
  - 主循环 `for j = 2 to n` 执行 $n-1$ 次。循环体内部进行常数次算术运算和比较。所以循环部分的时间是 $\Theta(n)$。
  - 计算最终 f^* 和 l^* 需要 $\Theta(1)$ 时间。
  - 因此，计算最优值 f^* 和记录选择信息的总时间复杂度是 $\Theta(n)$。
  - 如果需要打印路径， `PRINT-STATIONS` 的时间复杂度也是 $\Theta(n)$。
- **空间复杂度**:
  - 需要数组 f_1, f_2 各自存储 n 个值，即 $\Theta(n)$。
  - 需要数组 l_1, l_2 各自存储 $n-1$ 个值（或 n 个，取决于实现），即 $\Theta(n)$。
  - 因此，总空间复杂度是 $\Theta(n)$。

5. 备考要点

- **简答题考点**:
  - 描述装配线调度问题。
  - 解释为什么装配线调度问题具有最优子结构和重叠子问题性质。
  - 动态规划解决此问题的状态定义是什么？（$f_i[j]$ 的含义）
  - 写出 $f_i[j]$ 的递归关系式（状态转移方程）。
- **算法分析题考点**:
  - 给定具体的装配线参数（$a_{i,j}, t_{i,j}, e_i, x_i$），手动填表计算出所有的 $f_i[j]$ 和 $l_i[j]$ 值，并找出 f^* 和最优路径。 (LEC7, p.4 右下角的表格就是一个例子)
  - 分析 `FASTEST-WAY` 算法的时间和空间复杂度。
- **设计证明题考点**:
  - 完整描述使用动态规划解决装配线调度问题的算法步骤（包括状态定义、递归式、填表顺序、路径回溯）。
  - 证明该问题的最优子结构性质。

动态规划详解之三：矩阵链相乘问题 (Matrix-Chain Multiplication) (LEC7)

1. 问题描述 (LEC7, p.5)

- **输入**：一个包含 n 个矩阵的序列（或链）$\langle A_1, A_2, \ldots, A_n \rangle$。每个矩阵 A_i 的维度为 $p_{i-1} \times p_i$。
- **矩阵相乘的条件**：两个矩阵 A (维度 $r_A \times c_A$) 和 B (维度 $r_B \times c_B$) 能够相乘，当且仅当 A 的列数等于 B 的行数 ($c_A = r_B$)。乘积 $C = AB$ 的维度是 $r_A \times c_B$。
- **相乘代价**：计算一个 $p \times q$ 维矩阵和一个 $q \times r$ 维矩阵的乘积，所需的标量乘法次数为 $p \cdot q \cdot r$。

- **矩阵乘法的结合律**：矩阵乘法满足结合律，例如 $(A_1A_2)A_3 = A_1(A_2A_3)$。这意味着我们可以通过不同的加括号方式来计算一个矩阵链的乘积，但最终结果矩阵是相同的。
- **目标**：找到一种加括号的顺序（即一种完全的括号化方案），使得计算整个矩阵链乘积 $A_1A_2 \dots A_n$ 所需的**总标量乘法次数最少**。

2. 不同乘法顺序的代价差异 (LEC7, p.5)

矩阵链相乘的顺序对计算代价（标量乘法次数）有显著影响。

- **例子** (LEC7, p.5)：给定三个矩阵 $A_1 : 10 \times 100$, $A_2 : 100 \times 5$, $A_3 : 5 \times 50$。
  i. **顺序** $((A_1A_2)A_3)$：
    - 计算 $A_{12} = A_1A_2$：需要 $10 \times 100 \times 5 = 5,000$ 次标量乘法。A_{12} 的维度是 10×5。
    - 计算 $(A_{12})A_3$：需要 $10 \times 5 \times 50 = 2,500$ 次标量乘法。
    - 总代价：$5,000 + 2,500 = 7,500$ 次标量乘法。
  ii. **顺序** $(A_1(A_2A_3))$：
    - 计算 $A_{23} = A_2A_3$：需要 $100 \times 5 \times 50 = 25,000$ 次标量乘法。A_{23} 的维度是 100×50。
    - 计算 $A_1(A_{23})$：需要 $10 \times 100 \times 50 = 50,000$ 次标量乘法。
    - 总代价：$25,000 + 50,000 = 75,000$ 次标量乘法。
  - 这个例子表明，不同的加括号顺序导致的计算代价差异巨大。

3. 蛮力法 (Brute-Force Approach) (LEC7, p.6)

- 我们可以尝试枚举所有可能的加括号方式，计算每种方式的代价，然后选出最小的。
- 设 $P(n)$ 为 n 个矩阵链相乘的不同括号化方案的数量。这是一个**卡特兰数 (Catalan number)** 的问题（严格来说，卡特兰数 C_n 是对 $n+1$ 个叶子的二叉树计数，对应 n 次二元运算）。
- $P(n)$ 的递归关系为：
  $$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$
  这个和式表示，最后一次乘法可以将链在 A_k 和 A_{k+1} 之间分开，即 $(A_1 \dots A_k)(A_{k+1} \dots A_n)$。
- $P(n)$ 的增长速度是指数级的，$P(n) = \Omega(4^n/n^{3/2})$。 因此，蛮力法是不可行的。

4. 动态规划求解步骤

步骤1：描述最优解的结构特征 (Optimal Substructure) (LEC7, p.6)

- 考虑计算矩阵链 $A_{i \dots j} = A_iA_{i+1} \dots A_j$ 的最优括号化方案。
- 假设这个最优方案在 A_k 和 A_{k+1} 之间进行最后一次乘法（即 $(A_i \dots A_k)(A_{k+1} \dots A_j)$），其中 $i \leq k < j$。
- 那么，子链 $A_{i \dots k}$ 的括号化方案和子链 $A_{k+1 \dots j}$ 的括号化方案，也必须分别是它们各自的最优括号化方案。
  - **证明（反证法）**：如果 $A_{i \dots k}$ 的方案不是最优的，那么我们可以用一个更优的方案来替换它，从而得到一个比原最优方案更好的计算 $A_{i \dots j}$ 的方案，这与原方案是最优的假设矛盾。对 $A_{k+1 \dots j}$ 同理。
- 因此，矩阵链相乘问题具有**最优子结构**性质。这使得我们可以将原问题分解为求解子链的最优括号化问题。

步骤2：递归地定义最优解的值 (Recursive Definition) (LEC7, p.6)

- 设 $m[i, j]$ 为计算矩阵子链 $A_{i \dots j}$ 所需的最小标量乘法次数。我们的目标是计算 $m[1, n]$。

- **维度信息**：设矩阵 A_k 的维度是 $p_{k-1} \times p_k$。序列 $p_0, p_1, \ldots, p_n$ 给出了所有 n 个矩阵的维度。
- **基本情况**：
  - 当 $i = j$ 时，子链只包含一个矩阵 A_i，不需要进行任何乘法。所以 $m[i, i] = 0$。
- **递归步骤**：
  - 当 $i < j$ 时，我们需要找到一个分割点 k ($i \le k < j$)，使得在 A_k 和 A_{k+1} 之间进行最后一次乘法，其总代价最小。
  - 如果最后一次乘法是 $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$：
    - 计算 $A_{i \ldots k}$ 的代价是 $m[i, k]$。
    - 计算 $A_{k+1 \ldots j}$ 的代价是 $m[k+1, j]$。
    - 将这两个结果矩阵相乘（第一个结果矩阵维度是 $p_{i-1} \times p_k$，第二个是 $p_k \times p_j$）的代价是 $p_{i-1} \cdot p_k \cdot p_j$。
  - 因此，我们需要选择使这三项之和最小的 k：
    $m[i, j] = \min_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\}$
    (LEC7, p.6 的公式符号略有不同，它使用了 $A_{i,k}$ 和 $A_{k+1,j}$ 来表示子链，并说 $A_{i,k}$ 的维度是 $p_{i-1} \times p_k$，$A_{k+1,j}$ 的维度是 $p_k \times p_j$)

步骤3：以自底向上的方式计算最优解的值 (Bottom-up Computation) (LEC7, p.7)

- 我们使用一个二维表 `m[1..n, 1..n]` 来存储 $m[i, j]$ 的值。
- 还需要一个二维表 `s[1..n-1, 2..n]` 来存储导致最优代价的分割点 k 的值，即 $s[i, j]$ 存储的是计算 $A_{i \ldots j}$ 时最优分割位置 k。
- **计算顺序**：
  - 表 `m` 是按照链的长度 l 来填充的，从 $l = 1$ 到 $l = n$。
  - 当 $l = 1$ 时，$m[i, i] = 0$ for $i = 1, \ldots, n$。
  - 当 $l = 2, \ldots, n$ 时，对于所有 $i = 1, \ldots, n - l + 1$：
    - 令 $j = i + l - 1$。
    - 计算 $m[i, j] = \min_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\}$。
    - 并记录使 $m[i, j]$ 最小的那个 k 值到 $s[i, j]$。
- **算法** `Matrix-Chain-Order(p)` (LEC7, p.7)：

```
Matrix-Chain-Order(p) { // p是维度数组 p[0...n]
    n = p.length - 1 // 矩阵个数

    let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables

    // 1. 初始化链长度为1的情况 (m[i,i] = 0)
    for i = 1 to n {
        m[i,i] = 0
    }

    // 2. 迭代计算链长度从 l=2 到 l=n
    for l = 2 to n { // l is the chain length
        for i = 1 to n - l + 1 { // i是链的起始矩阵下标
            j = i + l - 1 // j是链的结束矩阵下标
            m[i,j] = infinity // 初始化为一个大值
            for k = i to j - 1 { // k是分割点，在i和j-1之间
                // q = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j] (这里p是维度数组)
                q = m[i,k] + m[k+1,j] + p[i-1] * p[k] * p[j]
                if q < m[i,j] then {
                    m[i,j] = q
                    s[i,j] = k // 记录最优分割点 k
                }
            }
        }
    }
    return m and s // 返回代价表 m 和分割点表 s
}
```

- **时间复杂度**：算法有三个嵌套循环。外层循环 `l` 从2到 n（约 n 次）。第二层循环 `i` 从1到 $n-l+1$（约 n 次）。最内层循环 `k` 从 i 到 $j-1=i+l-2$（约 l 次，最多 n 次）。因此，总时间复杂度是 $O(n^3)$。
- **空间复杂度**：需要 $\Theta(n^2)$ 的空间来存储表 `m` 和表 `s`。

步骤4：根据计算出的信息构造一个最优解 (Constructing an Optimal Solution) (LEC7, p.7)

- 表 `s` 中记录了最优分割点。我们可以使用一个递归过程来打印出最优的括号化方案。
- **算法** `Print-Optimal-Parens(s, i, j)` (或类似 `Matrix-Chain-Multiply` 的递归构造，LEC7, p.7):

```
Print-Optimal-Parens(s, i, j) {
    if i == j then {
        print "A" + i
    } else {
        print "("
        Print-Optimal-Parens(s, i, s[i,j]) // 递归打印左子链
        Print-Optimal-Parens(s, s[i,j] + 1, j) // 递归打印右子链
        print ")"
    }
}
```

初始调用为 `Print-Optimal-Parens(s, 1, n)`。

5. 备考要点

- **简答题考点**:
  - 描述矩阵链相乘问题（目标是什么）。
  - 解释为什么矩阵链相乘的顺序很重要（用例子说明代价差异）。
  - 矩阵链相乘问题是否具有最优子结构性质？如何体现？
  - 动态规划解决此问题的状态定义是什么？（$m[i, j]$ 的含义）
  - 写出 $m[i, j]$ 的递归关系式（状态转移方程）。
- **算法分析题考点**:
  - 给定一个矩阵维度序列，手动填表计算出 $m[i, j]$ 和 $s[i, j]$ 的值，并找出最优代价 $m[1, n]$。 (LEC7, p.7 右侧的表格就是一个例子)
  - 分析 `Matrix-Chain-Order` 算法的时间和空间复杂度。
- **设计证明题考点**:
  - 描述使用动态规划解决矩阵链相乘问题的完整算法步骤。
  - 证明该问题的最优子结构性质。
  - 根据计算出的 s 表，构造并打印出最优的括号化方案。

动态规划详解之四：最长公共子序列问题 (Longest Common Subsequence - LCS) (LEC7)

1. 问题描述与定义 (LEC7, p.7)

- **子序列 (Subsequence)**：给定一个序列 $X = \langle x_1, x_2, \ldots, x_m \rangle$，另一个序列 $Z = \langle z_1, z_2, \ldots, z_k \rangle$ 是 X 的一个子序列，如果存在 X 的一个严格递增的下标序列 $\langle i_1, i_2, \ldots, i_k \rangle$ (即 $1 \le i_1 < i_2 < \cdots < i_k \le m$)，使得对于所有的 $j = 1, 2, \ldots, k$，都有 $x_{i_j} = z_j$。
  - 简单来说，子序列是通过从原序列中删除零个或多个元素（不改变其余元素的相对顺序）得到的序列。
  - **例子**：如果 $X = \langle A, B, C, B, D, A, B \rangle$，那么 $Z = \langle B, C, A \rangle$ 是 X 的一个子序列，对应的下标序列可以是 $\langle 2, 3, 6 \rangle$。

- **公共子序列 (Common Subsequence)**：给定两个序列 X 和 Y，如果序列 Z 既是 X 的子序列，又是 Y 的子序列，则称 Z 是 X 和 Y 的一个公共子序列。
  - **例子 (LEC7, p.7)**：
    - $X = \langle A, B, C, B, D, A, B \rangle$
    - $Y = \langle B, D, C, A, B, A \rangle$
    - $Z = \langle B, C, A \rangle$ 是 X 和 Y 的一个公共子序列。
- **最长公共子序列 (Longest Common Subsequence - LCS)**：在两个序列 X 和 Y 的所有公共子序列中，长度最长的那个（或那些）子序列称为 X 和 Y 的最长公共子序列。LCS 可能不唯一。
  - **例子 (LEC7, p.7)**：对于上述 X 和 Y，$Z' = \langle B, D, A, B \rangle$ 是一个长度为4的LCS。另一个LCS可能是 $\langle B, C, B, A \rangle$。
  - **例子 (DNA 测序, LEC7, p.7)**：
    - `S1 = ACCGGTCGAGATGCAG`
    - `S2 = GTCGTTCGGAATGCAT` (PPT中S2为空，这里假设一个S2)
    - 它们的LCS可以用于衡量它们的相似度。

2. 蛮力法 (Brute-Force Approach) (LEC7, p.7)

- 我们可以枚举序列 X 的所有 2^m 个子序列（m 是 X 的长度）。
- 对于每个子序列，检查它是否也是 Y 的子序列（这需要 $O(n)$ 时间，其中 n 是 Y 的长度）。
- 然后从所有公共子序列中选出最长的。
- 这种方法的时间复杂度大约是 $O(m2^m)$ 或 $O(n2^m)$，是指数级的，效率非常低。

3. 动态规划求解步骤

步骤1：描述最优解的结构特征 (Optimal Substructure) (LEC7, p.8)

- 设 $X = \langle x_1, x_2, \ldots, x_m \rangle$ 和 $Y = \langle y_1, y_2, \ldots, y_n \rangle$ 是两个序列。
- 设 $Z = \langle z_1, z_2, \ldots, z_k \rangle$ 是 X 和 Y 的一个LCS。
- 我们考虑 X 的最后一个元素 x_m 和 Y 的最后一个元素 y_n：
  i. **如果 $x_m = y_n$**：那么这个共同的元素 $x_m (= y_n)$ 必然是 Z 的最后一个元素 z_k。并且，序列 $Z_{k-1} = \langle z_1, \ldots, z_{k-1} \rangle$ 必须是 $X_{m-1} = \langle x_1, \ldots, x_{m-1} \rangle$ 和 $Y_{n-1} = \langle y_1, \ldots, y_{n-1} \rangle$ 的一个LCS。
     - **证明**：如果 $z_k \neq x_m$，那么我们可以将 $x_m (= y_n)$ 追加到 Z 的末尾，得到一个比 Z 更长的公共子序列，与 Z 是LCS矛盾。如果 Z_{k-1} 不是 X_{m-1} 和 Y_{n-1} 的LCS，设 W 是 X_{m-1} 和 Y_{n-1} 的一个更长的LCS，那么将 x_m 追加到 W 的末尾会得到一个比 Z 更长的 X 和 Y 的公共子序列，也与 Z 是LCS矛盾。
  ii. **如果 $x_m \neq y_n$**：那么 x_m 和 y_n 不可能同时是 Z 的最后一个元素（实际上，它们中至少有一个不是 Z 的一部分，或者都不是）。
     - **情况 2a**: 如果 $z_k \neq x_m$ (即 x_m 不是LCS Z 的一部分)，那么 Z 必须是 X_{m-1} 和 Y 的一个LCS。
     - **情况 2b**: 如果 $z_k \neq y_n$ (即 y_n 不是LCS Z 的一部分)，那么 Z 必须是 X 和 Y_{n-1} 的一个LCS。
     - 因此，如果 $x_m \neq y_n$，那么 Z 要么是 X_{m-1} 和 Y 的LCS，要么是 X 和 Y_{n-1} 的LCS，取决于哪个能得到更长的公共子序列。
- 这表明LCS问题具有**最优子结构**性质。

步骤2：递归地定义最优解的值 (Recursive Definition) (LEC7, p.8)

- 设 $c[i, j]$ 为序列 $X_i = \langle x_1, \ldots, x_i \rangle$ 和 $Y_j = \langle y_1, \ldots, y_j \rangle$ 的LCS的**长度**。我们的目标是计算 $c[m, n]$。
- 根据上述最优子结构分析，我们可以得到 $c[i, j]$ 的递归定义：
  - **基本情况**：如果 $i = 0$ 或 $j = 0$（即其中一个序列为空），则LCS的长度为0。
    $c[i, j] = 0 \quad \text{if } i = 0 \text{ or } j = 0$
  - **递归步骤** (对于 $i, j > 0$)：
    - **如果** $x_i = y_j$：那么 $x_i (= y_j)$ 是LCS的一个元素，我们需要找到 X_{i-1} 和 Y_{j-1} 的LCS的长度，然后加1。
      $c[i, j] = c[i-1, j-1] + 1 \quad \text{if } x_i = y_j$
    - **如果** $x_i \neq y_j$：那么 x_i 和 y_j 不能同时作为LCS的最后一个匹配字符。我们需要在两种可能中取较大者：
      a. LCS是 X_{i-1} 和 Y_j 的LCS (即不考虑 x_i)。
      b. LCS是 X_i 和 Y_{j-1} 的LCS (即不考虑 y_j)。
      $c[i, j] = \max(c[i, j-1], c[i-1, j]) \quad \text{if } x_i \neq y_j$

步骤3：以自底向上的方式计算最优解的值 (Bottom-up Computation) (LEC7, p.8)

- 我们使用一个二维表 `c[0..m, 0..n]` 来存储 $c[i, j]$ 的值。
- 还需要一个辅助表 `b[1..m, 1..n]` 来记录构造最优解的选择（即 $c[i, j]$ 的值是由哪个子问题导出的），这有助于后续回溯构造LCS本身。
  - 如果 $x_i = y_j$，则 $b[i, j]$ 记为 "↖" (表示来自 $c[i-1, j-1]$)。
  - 如果 $x_i \neq y_j$：
    - 若 $c[i-1, j] \geq c[i, j-1]$，则 $b[i, j]$ 记为 "↑" (表示来自 $c[i-1, j]$)。
    - 否则 ($c[i-1, j] < c[i, j-1]$)，则 $b[i, j]$ 记为 "←" (表示来自 $c[i, j-1]$)。
- **算法** `LCS-Length(X, Y)` (LEC7, p.8)：

```
LCS-Length(X, Y) {
    m = X.length
    n = Y.length
    let b[1..m, 1..n] and c[0..m, 0..n] be new tables

    // 初始化基本情况 (第0行和第0列为0)
    for i = 0 to m { c[i,0] = 0 }
    for j = 0 to n { c[0,j] = 0 }

    // 按行主序或列主序填表
    for i = 1 to m {
        for j = 1 to n {
            if X[i] == Y[j] then { // X[i] 和 Y[j] 是序列中第i和第j个字符
                c[i,j] = c[i-1,j-1] + 1
                b[i,j] = "↖" // Diagonal arrow
            } else if c[i-1,j] >= c[i,j-1] then {
                c[i,j] = c[i-1,j]
                b[i,j] = "↑" // Up arrow
            } else {
                c[i,j] = c[i,j-1]
                b[i,j] = "←" // Left arrow
            }
        }
    }
    return c and b // c[m,n] 包含LCS的长度，b用于构造LCS
}
```

- **时间复杂度**：初始化需要 $O(m+n)$。主体是两个嵌套循环，内层是常数时间操作。所以总时间复杂度是 $O(mn)$。
- **空间复杂度**：需要 $O(mn)$ 来存储表 `c` 和表 `b`。

步骤4：根据计算出的信息构造一个最优解 (Constructing an LCS)

- 有了表 `b` (或仅通过表 `c` 的值也可以推断)，我们可以从 $b[m,n]$ 开始，沿着箭头反向回溯到 $b[0,0]$ (或 $c[0,0]$) 来构造LCS。
- **算法 `Print-LCS(b, X, i, j)`** (递归版本)：

```
Print-LCS(b, X, i, j) {
    if i == 0 or j == 0 then {
        return
    }
    if b[i,j] == "↖" then {
        Print-LCS(b, X, i-1, j-1)
        print X[i] // 打印出LCS的一个字符
    } else if b[i,j] == "↑" then {
        Print-LCS(b, X, i-1, j)
    } else { // b[i,j] == "←"
        Print-LCS(b, X, i, j-1)
    }
}
```

初始调用为 `Print-LCS(b, X, m, n)`。这个过程的时间复杂度是 $O(m+n)$，因为它每次递归调用 i 或 j (或两者) 至少减1。

5. 例子 (LEC7, pp.8-9)

PPT (LEC7, pp.8-9) 中给出了一个详细的例子，计算序列 $X = \langle s, p, a, n, k, i, n, g \rangle$ 和 $Y = \langle a, m, p, u, t, a, t, i, o, n \rangle$ 的LCS。
表格 `c` 和 `b` 会被逐步填充。例如：

- $c[0,j] = 0, c[i,0] = 0$。
- 当计算 $c[1,1]$ 时，$X[1] =' s', Y[1] =' a'$。$X[1] \neq Y[1]$。
  $c[0,1] = 0, c[1,0] = 0$。$\max(0,0) = 0$。所以 $c[1,1] = 0$。$b[1,1]$ 为 "↑" 或 "←"。
- 当计算 $c[3,1]$ 时，$X[3] =' a', Y[1] =' a'$。$X[3] = Y[1]$。
  $c[3,1] = c[2,0] + 1 = 0 + 1 = 1$。$b[3,1]$ 为 "↖"。
  最终 $c[m,n]$ (表中右下角的值) 就是LCS的长度。然后从 $b[m,n]$ 开始根据箭头回溯，遇到 "↖" 时对应的 $X[i]$ (或 $Y[j]$) 就是LCS中的一个字符。

6. 备考要点

- **简答题考点**：
  - 什么是子序列？什么是公共子序列？什么是LCS？
  - 解释LCS问题如何体现最优子结构性质。
  - 动态规划解决LCS问题的状态定义是什么？（$c[i,j]$ 的含义）
  - 写出 $c[i,j]$ 的递归关系式（状态转移方程）。
- **算法分析题考点**：
  - 给定两个短序列 X 和 Y，手动填表计算出 $c[i,j]$ 和 $b[i,j]$ 的值。
  - 分析 `LCS-Length` 算法的时间和空间复杂度。
  - 根据计算出的 b 表，回溯构造出一个LCS。
- **设计证明题考点**：
  - 描述使用动态规划解决LCS问题的完整算法步骤。
  - 证明LCS问题的最优子结构性质。

- 描述如何从计算出的辅助信息中构造LCS序列。

贪心算法详解之一：基本思想与动态规划比较 (LEC8)

1. 贪心算法的基本思想 (LEC8, p.1)

- **核心概念**：贪心算法在解决问题时，总是做出当前看起来是最佳的选择。也就是说，它期望通过一系列的**局部最优选择**来达到**全局最优解** 。
- **决策过程**：
  i. 将优化问题分解为若干个步骤。
  ii. 在每个步骤中，都面临一个选择。
  iii. 贪心算法会做出在当前看来“最好”的选择，而不考虑这个选择对后续步骤可能产生的影响。
  iv. 一旦做出选择，就不能撤销（没有回溯）。
- **适用性**：贪心算法并不总是能得到全局最优解。对于某些特定类型的问题（例如具有贪心选择性质和最优子结构性质的问题），贪心策略才能保证得到全局最优解。

2. 贪心算法与动态规划的比较 (LEC8, p.4)

特性	动态规划 (Dynamic Programming)	贪心算法 (Greedy Algorithms)
每步选择	每一步的选择依赖于子问题的解， 通常需要解决所有相关的子问题才能做出当前选择。	每一步都做出当前看起来最优的局部选择（贪心选择）。
子问题依赖	子问题的解会被存储起来，用于解决更大的问题（通常是自底向上）。	当前的贪心选择可能会简化问题， 产生一个唯一的、规模更小的子问题。
决策顺序	通常是自底向上， 从小规模子问题逐步求解到大规模问题。	通常是自顶向下，做出一次贪心选择后， 递归地解决剩余的子问题。
全局最优	保证能找到全局最优解 （如果问题具有最优子结构和重叠子问题性质）。	不一定能得到全局最优解， 需要证明贪心选择能导向全局最优。
信息利用	利用子问题的解来指导当前步骤的选择。	当前的选择只基于局部信息， 不依赖于未来子问题的解。
回溯	本质上不回溯 （因为它已经考虑了所有子问题的最优解）。	不回溯，一旦做出选择就不能更改。

- **共同点**：两者通常都要求问题具有**最优子结构**性质。
- **关键区别**：
  - **贪心选择性质 (Greedy Choice Property)**：可以通过做出局部最优（贪心）选择来构造全局最优解。做出贪心选择后，原问题简化为一个更小的子问题。动态规划通常需要考察多个子问题的解才能做出选择。
  - **动态规划**通常在每一步需要求解多个子问题，并从中选择一个能导出最优解的决策。而**贪心算法**在每一步都直接做出一个看起来最优的决策，然后只需求解由这个决策导出的那个子问题。

3. 贪心算法的设计步骤 (LEC8, p.4)

设计一个贪心算法并证明其正确性通常涉及以下步骤：

1. **确定问题的优化结构 (Optimal Substructure)**：证明问题的最优解包含其子问题的最优解。这是贪心算法和动态规划都能使用的基础。
2. **设计贪心策略 (Greedy Strategy)**：确定一种在每一步都能做出局部最优选择的标准或规则。
3. **证明贪心选择性质 (Greedy Choice Property)**：证明通过所选的贪心策略做出的局部最优选择，能够导向全局最优解。这是最关键的一步，通常通过“交换论证”（exchange argument）来证明：假设存在一个不同于贪心选择的最优解，然后证明可以通过修改这个最优解（将其中的某个选择替换为贪心选择）而不损害其最优性，甚至得到与贪心解一致的最优解。
4. **证明最优子结构 (结合贪心选择)**：证明做出贪心选择后，原问题简化为一个规模更小的、性质相同的子问题，且原问题的最优解加上这个贪心选择就构成了对简化后子问题的最优解的扩展。
5. **用递归或迭代实现贪心策略 (Recursive or Iterative Implementation)**：将贪心策略转化为算法。贪心算法通常是自顶向下的，做出选择，然后解决子问题。

4. 备考要点

- **简答题考点**：
  - 描述贪心算法的基本思想。
  - 贪心算法与动态规划的主要区别和联系是什么？
  - 什么是“贪心选择性质”？它在贪心算法设计中的作用是什么？
  - 贪心算法一定能得到最优解吗？为什么？
  - 简述设计一个贪心算法并证明其正确性的一般步骤。
- **算法分析题考点**：
  - 对于给定的问题，判断是否适合使用贪心策略。
  - 如果一个贪心策略不能得到最优解，能够举出反例。
- **设计证明题考点**：
  - 针对具体问题，设计贪心策略。
  - 证明所设计的贪心策略的正确性（尤其是贪心选择性质和最优子结构）。
  - 描述贪心算法的实现。

贪心算法详解之二：作业选择问题 (Activity Selection Problem) (LEC8)

作业选择问题是贪心算法的一个经典应用。目标是从一组有起始和结束时间的作业中，选出尽可能多的互不冲突的作业。

1. 问题描述 (LEC8, p.1)

- **输入**：
  - 一个包含 n 个作业的集合 $S = \{a_1, a_2, \ldots, a_n\}$。
  - 每个作业 a_i 都有一个起始时间 s_i 和一个完成时间 f_i，其中 $0 \leq s_i < f_i$。
  - 这些作业在执行期间需要共享某个共同的资源，该资源在同一时间只能被一个作业使用。

- **作业冲突**：如果两个作业 a_i 和 a_j 的执行时间区间 $[s_i, f_i)$ 和 $[s_j, f_j)$ 有重叠，则它们是冲突的。
- **兼容作业**：如果两个作业 a_i 和 a_j 的时间区间不重叠，则称它们是兼容的（或不冲突的）。这满足以下条件之一：
  - $f_i \leq s_j$ (作业 a_i 在作业 a_j 开始前结束)
  - $f_j \leq s_i$ (作业 a_j 在作业 a_i 开始前结束)
- **目标**：从集合 S 中选出一个**最大兼容作业子集**，即选出数量最多的、两两之间互不冲突的作业。

例子 (LEC8, p.1)：
PPT中给出了11个作业，每个作业有起始时间和完成时间。

作业 i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

- 不冲突的作业集合例子：$\{a_3, a_9, a_{11}\}$。
- 数目最多的不冲突作业集合例子：$\{a_1, a_4, a_8, a_{11}\}$ 或 $\{a_2, a_4, a_9, a_{11}\}$。

2. 动态规划的思路 (LEC8, p.1, 2)

在考虑贪心策略之前，我们先看看动态规划如何解决这个问题。这有助于理解为何贪心策略在此问题上有效。

- **预处理**：首先，将所有作业按照**完成时间 f_i 非递减（升序）排序**。这是一个非常关键的步骤，很多贪心策略都依赖于此。假设排序后 $f_1 \leq f_2 \leq \cdots \leq f_n$。
- **定义子问题空间** (LEC8, p.1)：
  - 设 $S_{ij} = \{a_k \in S \mid f_i \leq s_k \text{ and } f_k \leq s_j\}$。这表示在作业 a_i 完成之后开始，并且在作业 a_j 开始之前完成的所有作业的集合。
  - 为了方便定义边界，引入两个虚拟作业：a_0 (完成时间 $f_0 = -\infty$ 或 0) 和 a_{n+1} (开始时间 $s_{n+1} = \infty$)。
  - 那么，完整的问题空间就是 $S_{0,n+1}$。
- **最优子结构** (LEC8, p.2)：
  - 假设 A_{ij} 是子问题 S_{ij} 的一个最优解（即 S_{ij} 中最大的兼容作业集）。
  - 如果 A_{ij} 非空，设 a_k 是 A_{ij} 中的一个作业。那么 A_{ij} 可以分解为：$A_{ik} \cup \{a_k\} \cup A_{kj}$ (其中 A_{ik} 是 S_{ik} 的最优解，A_{kj} 是 S_{kj} 的最优解)。
  - 最优解 A_{ij} 的大小等于 $|A_{ik}| + 1 + |A_{kj}|$。
  - 这表明问题具有最优子结构。
- **递归解** (LEC8, p.2)：
  - 设 $c[i, j]$ 是集合 S_{ij} 中兼容作业的最大数目。
  - 如果 $S_{ij} = \emptyset$ (即 $i \geq j$)，则 $c[i, j] = 0$。
  - 如果 $S_{ij} \neq \emptyset$，则我们需要选择一个作业 $a_k \in S_{ij}$ 作为解的一部分，然后递归求解子问题 S_{ik} 和 S_{kj}。$c[i, j] = \max_{a_k \in S_{ij}}\{c[i, k] + c[k, j] + 1\}$ (这里 k 的范围是 $i+1, \ldots, j-1$，并且 a_k 必须与 a_i 和 a_j 兼容)。
  - 这种动态规划方法需要填充一个二维表，时间复杂度较高（可能是 $O(n^3)$ 或 $O(n^2)$ 取决于如何实现 $\max$）。

3. 贪心策略的设计与证明 (LEC8, p.2, 3)

动态规划提供了一个解决思路，但我们可以做得更好。关键在于找到一个有效的贪心选择。

- **贪心选择**：在所有可供选择的作业中（即那些与已选作业不冲突的作业），选择**完成时间最早**的那个作业。
  - **前提**：再次强调，所有作业必须**预先按照完成时间 f_i 非递减排序**。
- **定理 (LEC8, p.2)**：对于非空的作业集 S_{ij}（已按完成时间排序），设 a_m 是 S_{ij} 中具有最早完成时间的作业，即 $f_m = \min\{f_k \mid a_k \in S_{ij}\}$。则：
  i. 作业 a_m 必然是 S_{ij} 的某个最大兼容作业子集中的一个作业（即 a_m 是一个“安全”的贪心选择）。
  ii. 做出贪心选择 a_m 后，原问题 S_{ij} 简化为**一个**非空的子问题 S_{mj}（即寻找在 a_m 完成之后开始，并在 a_j 开始之前完成的作业）。子问题 S_{im} 是空集，因为 a_m 是 S_{ij} 中完成时间最早的，不可能有作业在 a_i 之后完成、在 a_m 之前开始且在 a_m 之前完成。
- **证明第1点 (贪心选择性质 - 交换论证)** (LEC8, p.3)：
  - 设 A_{ij} 是 S_{ij} 的一个最大兼容作业子集，并且假设 A_{ij} 中的作业已按完成时间排序。
  - 设 a_k 是 A_{ij} 中的第一个作业（即完成时间最早的作业）。
  - 如果 $a_k = a_m$，则贪心选择已在最优解中，证明完毕。
  - 如果 $a_k \neq a_m$，那么由于 a_m 是 S_{ij} 中具有最早完成时间的作业，所以 $f_m \leq f_k$。
  - 构造一个新的作业集合 $A'_{ij} = (A_{ij} - \{a_k\}) \cup \{a_m\}$。
  - 由于 $f_m \leq f_k$，且 a_k 与 $A_{ij} - \{a_k\}$ 中的所有作业都兼容，所以 a_m 也与 $A_{ij} - \{a_k\}$ 中的所有作业兼容（因为 a_m 完成得更早或同时完成，不会与那些在 a_k 之后开始的作业冲突）。
  - 因此，A'_{ij} 也是一个兼容作业子集，且 $|A'_{ij}| = |A_{ij}|$，所以 A'_{ij} 也是一个最大兼容作业子集。
  - 这表明，总存在一个最优解包含了贪心选择的作业 a_m。
- **证明第2点 (最优子结构结合贪心选择)** (LEC8, p.3)：
  - 我们选择了 a_m。我们需要证明，最优解 A_{ij} (包含 a_m) 等于 $\{a_m\}$ 加上 S_{mj} 的一个最优解 A_{mj}。
  - $A_{ij} = \{a_m\} \cup A_{mj}$。
  - 如果存在 S_{mj} 的一个解 A'_{mj} 使得 $|A'_{mj}| > |A_{mj}|$，那么 $\{a_m\} \cup A'_{mj}$ 将是 S_{ij} 的一个比 A_{ij} 更优的解，这与 A_{ij} 是最优解矛盾。
  - 同时，如前所述，S_{im} 是空集。因为如果存在 $a_k \in S_{im}$，则 $f_i \leq s_k < f_k \leq s_m$。这意味着 $f_k < f_m$，这与 a_m 是 S_{ij} 中具有最早完成时间的作业的前提相矛盾。
- **贪心策略的优势** (LEC8, p.3)：
  - 动态规划需要考虑多个子问题组合（S_{ik} 和 S_{kj}，对于所有可能的 k）。
  - 贪心策略在做出选择后，只留下**一个**非空子问题 (S_{mj})。这使得算法可以更直接地（通常是迭代地）进行。

4. 贪心算法实现 (LEC8, p.4)

基于上述贪心策略（选择完成时间最早的作业），可以设计出迭代和递归两种版本的算法。

- **递归贪心算法 `REC-ACT-SEL(s, f, i, n)`** (LEC8, p.4)：
  - (假设作业已按完成时间 f_k 排序， `s` 和 `f` 是起始和完成时间数组)
  - `i` 是上一个选入最优解的作业的下标（初始调用时可以是虚拟作业 a_0 的下标0）。
  - `n` 是作业总数。
  - 算法找到第一个在作业 a_i 完成之后开始的作业 a_m (即 $s_m \geq f_i$)。
  - 如果找到了这样的 a_m，则将 a_m 加入解集，并递归调用 `REC-ACT-SEL(s, f, m, n)`。

```
REC-ACT-SEL(s, f, i, n_total) { // 查找与作业a_i兼容的活动，a_i是上一个被选中的活动
    m = i + 1
    // 找到第一个与 a_i 完成时间 f[i] 不冲突的活动 a_m
    // (即 s[m] >= f[i])
    while m <= n_total and s[m] < f[i] {
        m = m + 1
    }
    if m <= n_total then {
        // a_m 是找到的第一个兼容活动
        return {a_m} U REC-ACT-SEL(s, f, m, n_total)
    } else {
        return {} // 没有更多兼容活动
    }
}
// 初始调用: REC-ACT-SEL(s, f, 0, n)  (假设 f[0] = 0 或 -infinity)
```

  - **时间复杂度**：如果作业已排序，每次递归调用都会使问题规模减小，并且 `while` 循环在整个递归过程的各次调用中，指针 `m` 总共只会从1扫描到 n 一次。因此，总时间是 $\Theta(n)$（不包括初始排序）。
- **迭代贪心算法 `GREEDY-ACTIVITY-SELECTOR(s, f, n)`** (LEC8, p.4)：
  - (假设作业已按完成时间 f_k 排序)

```
GREEDY-ACTIVITY-SELECTOR(s, f, n) {
    A = {a_1} // 第一个作业 (完成时间最早) 总是被选入
    last_selected_idx = 1 // 记录最后一个选入解集的作业的下标

    for m = 2 to n { // 考察其余作业
        // 如果作业 a_m 与最后一个选入的作业 a_last_selected_idx 兼容
        // (即 a_m 的开始时间晚于或等于 a_last_selected_idx 的完成时间)
        if s[m] >= f[last_selected_idx] then {
            A = A U {a_m}
            last_selected_idx = m
        }
    }
    return A
}
```

  - **时间复杂度**：
    - 初始排序作业按完成时间：$\Theta(n \log n)$。
    - 迭代过程：一个简单的 `for` 循环，执行 $n-1$ 次，每次迭代内部是常数时间操作。所以迭代部分是 $\Theta(n)$。
    - 总时间复杂度由排序主导，为 $\Theta(n \log n)$。如果输入已经按完成时间排序，则算法本身是 $\Theta(n)$。

5. 备考要点

- **简答题考点**：
  - 描述作业选择问题。

- 解决作业选择问题的贪心策略是什么？（选择完成时间最早的兼容作业）
- 为什么在应用该贪心策略前需要对作业按完成时间排序?
- 简述证明该贪心策略正确性的“交换论证”思想。

- **算法分析题考点**:
  - 给定一组作业，手动模拟贪心算法选择过程，并给出结果。
  - 分析迭代贪心算法的时间复杂度（包括排序和选择部分）。
- **设计证明题考点**:
  - 描述解决作业选择问题的贪心算法（迭代或递归版本）。
  - 证明作业选择问题的贪心选择性质（即选择最早完成的作业总能导向某个最优解）。
  - 证明作业选择问题具有最优子结构（在做出贪心选择后）。

贪心算法详解之三 (修正版)：Huffman 编码 (Huffman Coding) (LEC8)

Huffman 编码是一种用于无损数据压缩的贪心算法，它根据字符出现的频率来构造最优的前缀码。

1. 问题背景：数据压缩 (LEC8, p.4)

- **目标**：减少存储文件或传输数据所需的位数 。
- **编码方式**:
  - **等长编码 (Fixed-length code)**：将每个字符用相同长度的二进制串表示 。例如，ASCII码用7位或8位表示一个字符。
    - 如果字符集大小为 $|C|$，则等长编码至少需要 $\lceil \log_2 |C| \rceil$ 位。
    - **缺点**：没有利用字符出现频率的不同。频繁出现的字符和不频繁出现的字符使用相同的编码长度，可能导致空间浪费 。
  - **变长编码 (Variable-length code)**：允许不同字符使用不同长度的二进制串表示 。
    - **思想**：给频率高的字符赋予较短的编码，给频率低的字符赋予较长的编码，从而达到压缩的目的 。

2. 前缀码 (Prefix Codes) (LEC8, p.5)

- **定义**：在前缀码中，任何字符的编码都**不能**是另一个字符编码的前缀 。
  - 例如，如果 'a' 的编码是 "01"，那么任何其他字符的编码都不能以 "01" 开头（如 "010" 或 "0110"）。
- **为什么要用前缀码?** (LEC8, p.5)
  - **无歧义解码**：前缀码的主要优点是它们可以唯一地、无歧义地进行解码 。当读取编码后的比特流时，一旦遇到一个与某个字符编码匹配的序列，就可以立即确定该字符，而无需向后查看更多比特来区分 。
- **前缀码的表示：编码树 (Binary Tree Representation)** (LEC8, p.5)
  - 前缀码可以用一个**满二叉树**（full binary tree，即每个非叶节点都有两个孩子）来表示 。
  - **字符**：树的**叶子节点**对应于字符集中的字符 。
  - **编码**：从根节点到某个字符叶子节点的路径定义了该字符的编码 。通常约定，从父节点到左孩子的边标记为 "0"，到右孩子的边标记为 "1" (反之亦可，但需一致) 。
  - **路径长度**：字符的编码长度等于从根到其对应叶子节点的路径长度（即深度） 。
  - **前缀码性质的体现**：由于每个字符都在叶子节点，所以没有哪个字符的路径会是另一个字符路径的前缀 。

3. 最优编码 (Optimal Coding) (LEC8, p.6)

- **编码代价**：给定一个字符集 C，其中每个字符 $c \in C$ 的出现频率为 $f(c)$（或 $f[c]$ 如PPT中所示），并且在一个编码树 T 中，字符 c 的编码长度（即叶节点 c 的深度）为 $d_T(c)$。那么，使用编码树 T 对一个包含这些字符的文件进行编码所需的总位数（或文件的总代价）为：
  $B(T) = \sum_{c \in C} f(c)d_T(c)$
- **目标**：找到一个前缀码（即一棵编码树 T），使得 $B(T)$ 最小 。这样的编码称为最优前缀码，对应的树称为最优编码树。
- **最优编码的特征** (LEC8, p.6)：
  - 最优编码总是通过**满二叉树**来表示（即每个非叶节点都有两个子节点） 。
  - 频率高的字符应该具有较短的编码长度（深度较浅），频率低的字符应该具有较长的编码长度（深度较深） 。

4. Huffman 算法 (Greedy Approach) (LEC8, p.6)

Huffman 提出了一种贪心算法，用于构造最优前缀码。

- **基本思想 (自底向上构建编码树)**：
  i. 将字符集中的每个字符视为一个单独的叶子节点，其权重为其出现频率 $f(c)$。
  ii. 重复以下步骤，直到只剩下一个节点（即树的根节点）：
    - 从当前所有节点中，选择两个**频率最低**的节点（设为 x 和 y）。
    - 创建一个新的内部节点 z，使其左右孩子分别为 x 和 y。
    - 新节点 z 的频率设为其两个孩子频率之和：$f[z] = f[x] + f[y]$。
    - 将 x 和 y 从待合并节点集合中移除，并将新节点 z 加入该集合。
- **数据结构**：通常使用**最小优先队列 (min-priority queue)** 来高效地管理和选择频率最低的节点 。优先队列中的元素是树的节点，其键值为节点的频率。
- **算法 `HUFFMAN(C)`** (LEC8, p.6)：

```
HUFFMAN(C) { // C 是字符集，每个字符 c 带有频率 f[c]
    n = |C|                                 // 字符集大小
    Q = C                                   // 将所有字符（视为叶节点）放入最小优先队列 Q，按频率排序

    for i = 1 to n - 1 {                    // 循环 n-1 次
        allocate a new node z
        z.left = x = EXTRACT-MIN(Q)         // 提取频率最小的节点 x
        z.right = y = EXTRACT-MIN(Q)        // 提取频率次小的节点 y
        z.freq = x.freq + y.freq            // 新节点 z 的频率是 x 和 y 的频率之和
                                            // (PPT中用 f[z] <- f[x] + f[y])
        INSERT(Q, z)                        // 将新节点 z 插回优先队列
    }
    return EXTRACT-MIN(Q)                   // 返回队列中剩下的唯一节点，即 Huffman 树的根
}
```

- **效率分析** (LEC8, p.6)：
  - 构建初始优先队列：如果用二叉堆实现，需要 $O(n)$ 时间（$n = |C|$） 。
  - 循环执行 $n-1$ 次 。

- `EXTRACT-MIN` 操作：在二叉堆中是 $O(\log n)$ 。
- `INSERT` 操作：在二叉堆中是 $O(\log n)$ 。
- 循环总时间：$(n-1) \times (O(\log n) + O(\log n)) = O(n \log n)$ 。
- 因此，Huffman 算法的总时间复杂度是 $O(n \log n)$ 。

5. Huffman 编码的正确性（贪心选择性质与最优子结构）(LEC8, p.6, 7)

- **贪心选择性质 (Greedy Choice Property)** (LEC8, p.6, Theorem)：
  - 设 C 是一个字符集，其中每个字符 $c \in C$ 的频率为 $f[c]$。设 x 和 y 是 C 中具有最低频率的两个字符。那么，**存在** C 的一个最优前缀码方案，在该方案中，字符 x 和 y 的编码具有相同的长度，并且只有最后一位不同（即它们是兄弟节点） 。
  - **证明思路 (交换论证)** (LEC8, p.7)：
    a. 取任意一棵表示最优前缀码的树 T。设 a 和 b 是树 T 中具有最大深度的两个兄弟叶节点。不失一般性，假设 $f[a] \le f[b]$ 。
    b. 由于 x 和 y 是频率最低的两个字符，必然有 $f[x] \le f[a]$ 和 $f[y] \le f[b]$ 。同时，深度最大的叶子节点对应的编码长度最长。
    c. 通过交换字符的位置（先将 x 与 a 交换得到 T'，再将 y 与 b 交换得到 T''）可以构造一棵新树 T''，使得 x 和 y 成为具有与原 a, b 相同深度的兄弟节点，并且新树的总代价 $B(T'')$ 不会比原最优树的代价 $B(T)$ 更差。
       - 代价变化分析： $B(T) - B(T') = (f[a] - f[x])(d_T(a) - d_T(x))$ 。
       - 由于 $f[a] \ge f[x]$ 且 $d_T(a) \ge d_T(x)$ (因为 a 是深度最大的叶子之一， x 的频率是最低的，如果 x 不在最深层，则 $d_T(a) > d_T(x)$，如果 x 也在最深层，则 $d_T(a) = d_T(x)$)，所以 $B(T) - B(T') \ge 0$，即 $B(T') \le B(T)$ 。
       - 类似地， $B(T'') \le B(T')$。因此 $B(T'') \le B(T)$。由于 T 是最优的，必有 $B(T) \le B(T'')$，故 $B(T) = B(T'')$ 。
    d. 这表明，总存在一棵最优编码树，其中频率最低的两个字符 x 和 y 是兄弟节点。
    e. 这意味着，将频率最低的两个字符 x, y 合并为一个新的“复合字符”（其频率为 $f[x] + f[y]$），并将此复合字符与其他字符一起考虑，是构造最优解的一步。
- **最优子结构 (Optimal Substructure)**：
  - 设 x 和 y 是频率最低的两个字符，我们将它们合并成一个新节点 z，其频率为 $f[z] = f[x] + f[y]$。
  - 考虑原字符集 C 去掉 x, y 并加入 z 得到的新字符集 $C' = (C - \{x, y\}) \cup \{z\}$。
  - 如果树 T' 是对字符集 C' 的一个最优编码树，那么将 T' 中代表 z 的叶节点替换为一个以 z 为根（其频率为 $f[z]$）、以 x 和 y 为孩子（频率分别为 $f[x], f[y]$）的子树，所得到的树 T 就是对原字符集 C 的一个最优编码树。
  - **证明思路**：设 T 是通过上述方法从 T' 构造出来的树。其代价关系为 $B(T) = B(T') + f[x] + f[y]$ (因为 $d_T(x) = d_{T'}(z) + 1, d_T(y) = d_{T'}(z) + 1$, 而 $f[z] = f[x] + f[y]$)。如果 T 不是 C 的最优编码树，设 T_{opt} 是 C 的最优编码树，且 $B(T_{opt}) < B(T)$。根据贪心选择性质，T_{opt} 中 x, y 可以是兄弟节点。将 T_{opt} 中 x, y 及其父节点替换为代表 z 的叶节点，得到 C' 的一棵编码树 T'_{opt}，则 $B(T'_{opt}) = B(T_{opt}) - f[x] - f[y]$。那么 $B(T'_{opt}) < B(T) - f[x] - f[y] = B(T')$。这与 T' 是 C' 的最优编码树矛盾。
  - 因此，问题具有最优子结构。

6. 备考要点

- **简答题考点**:
  - 什么是前缀码？为什么它对数据压缩很重要?
  - Huffman 编码的目标是什么?
  - 描述 Huffman 算法的贪心策略 。
  - Huffman 算法的时间复杂度是多少？主要由什么操作决定?
  - 简述 Huffman 编码正确性的依据（贪心选择性质和最优子结构） 。
- **算法分析题考点**:
  - 给定一组字符及其频率，手动模拟 Huffman 算法构建编码树的过程，并写出每个字符的 Huffman 编码 。
  - 计算给定 Huffman 编码的总位数或平均编码长度 。
- **设计证明题考点**:
  - 描述 Huffman 算法的伪代码 。
  - 证明 Huffman 编码的贪心选择性质（如交换论证） 。
  - 证明 Huffman 编码具有最优子结构。

贪心算法详解之四：背包问题 (Knapsack Problem) (LEC8)

1. 问题概述

背包问题通常描述为：有一个固定容量的背包，以及一组具有各自价值和重量（或体积）的物品。目标是在不超过背包容量的前提下，选择装入背包的物品，使得装入物品的总价值最大。

2. 背包问题的两种主要类型 (LEC8, p.7)

1. **0/1 背包问题 (0/1 Knapsack Problem)**:
   - 对于每件物品，要么**完全拿走**（选择放入背包），要么**完全放弃**（不放入背包）。不能只拿走物品的一部分。
   - 设物品有 n 件，第 i 件物品的价值为 v_i，重量为 w_i。背包的容量为 W。
   - 需要确定一个选择向量 $x = \langle x_1, x_2, \ldots, x_n \rangle$，其中 $x_i \in \{0, 1\}$ ($x_i = 1$ 表示选择物品 i，$x_i = 0$ 表示不选择)。
   - 目标是最大化总价值 $\sum_{i=1}^{n} x_i v_i$，约束条件是总重量不超过背包容量 $\sum_{i=1}^{n} x_i w_i \leq W$ 。
2. **部分背包问题 (Fractional Knapsack Problem)**:
   - 与0/1背包问题不同，这种情况下可以拿走物品的一部分。
   - 设物品有 n 件，第 i 件物品的价值为 v_i，重量为 w_i。背包的容量为 W。
   - 需要确定每种物品拿走的比例 x_i，其中 $0 \leq x_i \leq 1$。
   - 目标是最大化总价值 $\sum_{i=1}^{n} x_i v_i$，约束条件是总重量不超过背包容量 $\sum_{i=1}^{n} x_i w_i \leq W$ 。

3. 0/1 背包问题与贪心策略 (LEC8, p.8)

- **贪心策略尝试**：对于0/1背包问题，可以尝试几种贪心策略：
  i. **选择价值最高的物品**：每次都选择当前剩余物品中价值最高的那个，只要它能装入背包。
  ii. **选择重量最轻的物品**：每次都选择当前剩余物品中重量最轻的那个。

iii. **选择单位重量价值最高的物品 (性价比最高)**：计算每件物品的单位重量价值 v_i/w_i，每次都选择当前剩余物品中单位重量价值最高的那个，只要它能装入背包。

- **贪心策略在0/1背包问题上的局限性**：
  - 上述这些贪心策略**都不能保证**为0/1背包问题找到最优解。
  - **例子 (LEC8, p.8)**：
    - 背包容量 $W = 50$。
    - 物品1: $v_1 = 60, w_1 = 10$ (单位价值 $60/10 = 6$)
    - 物品2: $v_2 = 100, w_2 = 20$ (单位价值 $100/20 = 5$)
    - 物品3: $v_3 = 120, w_3 = 30$ (单位价值 $120/30 = 4$)
    - **如果采用单位重量价值最高的贪心策略**：
      a. 选择物品1 ($v_1/w_1 = 6$)：放入背包。剩余容量 $50 - 10 = 40$。当前价值 60。
      b. 选择物品2 ($v_2/w_2 = 5$)：放入背包。剩余容量 $40 - 20 = 20$。当前价值 $60 + 100 = 160$。
      c. 物品3 ($w_3 = 30$) 无法放入剩余容量为20的背包。
      - 贪心解的总价值为 160。
    - **最优解**：
      - 选择物品2和物品3：总重量 $20 + 30 = 50 \leq 50$。总价值 $100 + 120 = 220$。
      - 这比贪心解 160 要好。
  - 这个例子表明，对于0/1背包问题，局部最优（选择当前性价比最高的）并不一定能导向全局最优。因为0/1的限制（要么全拿要么不拿）可能导致在选择了某个物品后，剩余空间无法有效利用。
- **0/1背包问题的解决方法**：0/1背包问题通常使用**动态规划**来求解，可以得到最优解。其动态规划的状态转移方程大致为：
  $P(i, w) = \max\{v_i + P(i-1, w-w_i), P(i-1, w)\}$
  其中 $P(i, w)$ 表示在前 i 件物品中选择，背包容量为 w 时的最大价值。

4. 部分背包问题与贪心策略 (LEC8, p.8)

- 对于部分背包问题，**选择单位重量价值最高的物品**的贪心策略是**可以得到最优解**的。
- **贪心算法步骤** `Fractional-Knapsack(W, v[n], w[n])` (LEC8, p.8)：
  i. 计算每件物品的单位重量价值 $p_i = v_i/w_i$。
  ii. 将所有物品按照单位重量价值 p_i **从高到低排序**。
  iii. 遍历排序后的物品：
    - `current_weight = 0`
    - `total_value = 0`
    - `for each item i in sorted order:`
      - `if current_weight + w[i] <= W then:` (如果整个物品可以装下)
        - `take all of item i (x_i = 1)`
        - `total_value = total_value + v[i]`
        - `current_weight = current_weight + w[i]`
      - `else:` (如果物品只能装一部分)
        - `remaining_capacity = W - current_weight`
        - `take fraction x_i = remaining_capacity / w[i] of item i`
        - `total_value = total_value + x_i * v[i]`
        - `current_weight = current_weight + x_i * w[i]`

        - break (背包已满)
    - PPT中的描述 (LEC8, p.8):
        a. While w > 0 and as long as there are items remaining (背包有容量且还有物品)
        b. `pick item with maximum (v_i / w_i)` (选择单位重量价值最高的物品)
        c. `x_i = min(1, w / w_i)` (要么全拿，要么只拿填满背包剩余容量的部分，其中 `w` 在此上下
        d. `remove item i from list` (或将该物品标记为已处理)
        e. `w = w - x_i * w_i` (更新背包剩余容量)

        (这里PPT的 w 似乎同时表示了总容量和剩余容量，理解其逻辑即可)
- **正确性证明思路 (交换论证)**:
    - 假设贪心算法得到的解不是最优的，那么存在一个最优解。
    - 比较贪心解和这个假设的最优解。找到第一个不同的物品选择。
    - 贪心算法总是选择当前单位重量价值最高的物品。如果最优解在某一步没有选择当前可选的单位重量价值最高的物品 g，而是选择了单位重量价值较低的物品 o（或者没有填满 g 就去选了 o），那么我们可以尝试从最优解中“取出一部分”物品 o，用等重量的物品 g 来“替换”。由于 g 的单位价值更高，这样的替换会使得总价值增加（或至少不减少），从而得到一个不差于原最优解的新解，并且这个新解更接近于贪心算法的选择。
    - 通过一系列这样的交换，可以证明贪心算法得到的解至少和任何最优解一样好，因此它就是最优的。
- **例子 (LEC8, p.9)** (与0/1背包例子相同的数据):
    - 背包容量 $W = 50$。
    - 物品1: $v_1 = 60, w_1 = 10$ (单位价值 6)
    - 物品2: $v_2 = 100, w_2 = 20$ (单位价值 5)
    - 物品3: $v_3 = 120, w_3 = 30$ (单位价值 4)
    - 按单位价值排序：物品1 > 物品2 > 物品3。
    - **贪心选择过程**:
        a. 选择物品1 (单位价值6)：全部拿走 ($x_1 = 1$)。
            - 已用重量: 10。价值: 60。剩余容量: $50 - 10 = 40$。
        b. 选择物品2 (单位价值5)：全部拿走 ($x_2 = 1$)。
            - 已用重量: $10 + 20 = 30$。价值: $60 + 100 = 160$。剩余容量: $40 - 20 = 20$。
        c. 选择物品3 (单位价值4)：只能拿一部分。可拿重量为剩余容量20。
            - 拿走比例 $x_3 = 20/30 = 2/3$。
            - 增加价值: $(2/3) \times 120 = 80$。
            - 已用重量: $30 + 20 = 50$。价值: $160 + 80 = 240$。剩余容量: 0。
    - 贪心解的总价值为 240。这对于部分背包问题是最优的。
- **时间复杂度**:
    - 计算所有物品的单位重量价值：$\Theta(n)$。
    - 根据单位重量价值排序物品：$\Theta(n \log n)$。

- 遍历排序后的物品并填充背包：$\Theta(n)$。
- 总时间复杂度由排序主导，为 $\Theta(n \log n)$。

5. 备考要点

- **简答题考点**：
  - 区分0/1背包问题和部分背包问题。
  - 对于0/1背包问题，常见的贪心策略（如按单位重量价值最高）是否能得到最优解？为什么？（举反例说明）
  - 0/1背包问题通常用什么方法求解最优解？（动态规划）
  - 对于部分背包问题，有效的贪心策略是什么？它能得到最优解吗？
- **算法分析题考点**：
  - 给定一组物品和背包容量，分别使用针对0/1背包的贪心策略（并指出其非最优性）和针对部分背包的贪心策略进行求解，并给出结果。
  - 分析部分背包问题贪心算法的时间复杂度。
- **设计证明题考点**：
  - 描述解决部分背包问题的贪心算法。
  - 证明部分背包问题的贪心策略（按单位重量价值最高者优先）的正确性（通常使用交换论证）。

图算法详解之 8.1：图的基础 (Graph Fundamentals) (LEC9)

这一部分主要介绍图的基本概念、不同类型的图，以及在计算机中表示图的常用方法。

1. 图的基本概念 (LEC9, p.1, 2)

- **图 (Graph)**：图是由一组**顶点 (vertices)**（也称为节点，nodes）和一组连接这些顶点的**边 (edges)**（也称为链接，links）组成的结构 。
- **形式化表示**：一个图 G 通常表示为 $G = (V, E)$，其中：
  - V是顶点的集合 。
  - $|V|$ 通常用 n 表示，代表图中顶点的数量 。
  - E是边的集合 。
  - $|E|$ 通常用 m 表示，代表图中边的数量 。
- **应用场景** (LEC9, p.1)：图可以用来模型化各种现实世界中的关系和网络，例如：
  - 地图（城市为顶点，道路为边） 。
  - 日程表（任务为顶点，依赖关系为边）。
  - 计算机网络（计算机或路由器为顶点，网络连接为边） 。
  - 超文本链接（网页为顶点，链接为边） 。
  - 电路板（元件或引脚为顶点，导线为边） 。

2. 图的类型 (LEC9, p.1, 2)

根据边的方向性、是否有回路等特性，图可以分为多种类型：

- **有向图 (Directed Graph / Digraph)** (LEC9, p.1):
  - 边是**有方向的**，从一个顶点指向另一个顶点。
  - 边 (u, v) 表示从顶点 u 到顶点 v 的一条有向连接。此时，u 称为边的起点（tail），v 称为边的终点（head）。
- **无向图 (Undirected Graph)** (LEC9, p.1):
  - 边是**没有方向的**。
  - 边 (u, v) 表示顶点 u 和顶点 v 之间存在一条连接，它等同于边 (v, u)。
- **连通图 (Connected Graph)** (LEC9, p.2):
  - 指一个**无向图**中，任意两个不同的顶点之间都存在一条路径 。
  - 如果一个无向图不是连通的，它会由若干个连通分量 (connected components) 组成。
- **有向无环图 (Directed Acyclic Graph - DAG)**:
  - 一个没有有向环路（cycle）的有向图。
  - DAG 在表示任务依赖关系（如拓扑排序）、状态转换等方面非常有用。
  - PPT (LEC9, p.1) 中展示了一个 "Acyclic Graph (无回路图)" 的例子 。
- **二分图 (Bipartite Graph)** (LEC9, p.2):
  - 指一个**无向图** $G = (V, E)$，其顶点集 V 可以被划分为两个互不相交的子集 V_1 和 V_2 ($V = V_1 \cup V_2$ 且 $V_1 \cap V_2 = \emptyset$)，使得图中的每一条边都连接 V_1 中的一个顶点和 V_2 中的一个顶点 。也就是说，在 V_1 内部或 V_2 内部不存在边。
  - 例如，可以用二分图来表示工人和工作之间的匹配关系，其中 V_1 代表工人，V_2 代表工作，边表示某个工人能做某项工作 。
- **加权图 (Weighted Graph)** (LEC9, p.9):
  - 图中的每条边 (u, v) 都被赋予了一个数值，称为边的**权重 (weight)** 或成本 (cost)，记为 $w(u, v)$ 。
  - 权重可以表示距离、时间、费用、容量等。
  - 权重函数 $w : E \to R$ (权重是实数) 。

3. 图的表示方法 (Graph Representation) (LEC9, pp.2-9)

在计算机中存储和操作图，主要有两种常用的表示方法：邻接表和邻接矩阵。

a. 邻接表 (Adjacency List) (LEC9, p.2)

- **结构**:
  - 对于图 $G = (V, E)$，邻接表表示法包含一个由 $|V|$ 个链表组成的数组（或类似结构，如Python中的列表的列表或字典的列表） 。
  - 数组的每个索引 u (对应一个顶点) 对应一个链表 `Adj[u]` 。
  - 链表 `Adj[u]` 中存储了所有与顶点 u **邻接 (adjacent)** 的顶点 v（即存在边 $(u, v) \in E$） 。
  - `Adj[u]` 中的顶点顺序可以是任意的 。
- **适用性**:
  - 既可以用于表示有向图，也可以用于表示无向图 。
    - **有向图**：如果存在边 (u, v)，则 v 出现在 `Adj[u]` 的链表中。边 (u, v) 只出现一次 。
    - **无向图**：如果存在边 (u, v)，则 v 出现在 `Adj[u]` 的链表中，并且 u 也出现在 `Adj[v]` 的链表中。边 (u, v) 会被表示两次 。
- **特性** (LEC9, p.2, 3):

- **空间复杂度**：$\Theta(V+E)$ (或 $\Theta(n+m)$)。因为它需要 $\Theta(V)$ 的空间存储顶点（数组的头部或链表头指针），以及 $\Theta(E)$ 的空间存储所有的邻接关系（在有向图中，每条边在邻接表中出现一次；在无向图中，每条边出现两次，所以是 $\Theta(2E)=\Theta(E)$）。
- **优点**：对于**稀疏图 (sparse graphs)**（即边的数量 $|E|$ 远小于 $|V|^2$ 的图，例如 $|E| \ll |V|^2$）非常节省空间。
- **缺点**：判断两个顶点 u 和 v 之间是否存在边 (u,v) 比较慢。需要扫描 `Adj[u]` 链表，最坏情况下时间复杂度为 $O(\text{degree}(u))$（$\text{degree}(u)$ 是顶点 u 的度，即与 u 相连的边的数量）。
- **列出与 u 邻接的所有顶点的时间**：$\Theta(\text{degree}(u))$。

b. 邻接矩阵 (Adjacency Matrix) (LEC9, p.8)

- **结构**：
  - 假设图 $G=(V,E)$ 的顶点被编号为 $1,2,\ldots,|V|$ (或 $0,\ldots,|V|-1$)。
  - 邻接矩阵是一个 $|V| \times |V|$ 的矩阵 $A=(a_{ij})$，其中：
    - $a_{ij}=1$ 如果边 $(i,j) \in E$。
    - $a_{ij}=0$ 如果边 $(i,j) \notin E$。
- **适用性**：
  - **无向图**：邻接矩阵是对称的，即 $a_{ij}=a_{ji}$。因此 $A=A^T$。
  - **有向图**：邻接矩阵可能是不对称的。
- **特性** (LEC9, p.9)：
  - **空间复杂度**：$\Theta(V^2)$ (或 $\Theta(n^2)$)，与图中边的数量无关。
  - **优点**：
    - 判断两个顶点 u 和 v 之间是否存在边 (u,v) 非常快，只需要检查 $A[u,v]$ 的值，时间复杂度为 $\Theta(1)$。
    - 对于**稠密图 (dense graphs)**（即边的数量 $|E|$ 接近 $|V|^2$ 的图）比较适合。
  - **缺点**：
    - 对于稀疏图，空间浪费严重。
    - 列出与顶点 u 邻接的所有顶点需要遍历矩阵的第 u 行（或第 u 列），时间复杂度为 $\Theta(V)$。

c. 加权图的表示 (LEC9, p.9)

- **邻接表**：可以在存储邻接顶点 v 的同时，存储边 (u,v) 的权重 $w(u,v)$。例如，`Adj[u]` 中的每个节点可以是一个包含顶点编号和对应边权重的对象或元组。
- **邻接矩阵**：可以直接在矩阵元素 $A[u,v]$ 中存储边 (u,v) 的权重 $w(u,v)$。如果边不存在，可以用一个特殊值（如 ∞ 或0，取决于上下文）来表示。如果 $u=v$，通常 $A[u,u]=0$。

4. 备考要点

- **简答题考点**：
  - 图的定义及其基本组成部分（顶点、边）。
  - 区分有向图和无向图。什么是连通图？什么是二分图？什么是加权图？
  - 描述邻接表表示法的结构和特点（空间复杂度、优缺点）。
  - 描述邻接矩阵表示法的结构和特点（空间复杂度、优缺点）。
  - 对于什么类型的图（稀疏/稠密）分别适合使用邻接表和邻接矩阵？为什么？
  - 如何在邻接表和邻接矩阵中表示加权图的权重？

- **算法分析题考点**:
  - 给定一个图的图形表示，画出其邻接表和邻接矩阵。
  - 分析使用邻接表或邻接矩阵时，执行特定操作（如判断边是否存在、列出邻接点）的时间复杂度。
- **设计证明题考点**:
  - （较少见）可能会要求比较两种表示方法的效率。

图算法详解之 8.2：图的遍历 (Graph Traversal) (LEC9)

A. 广度优先搜索 (Breadth-First Search - BFS) (LEC9, pp.10-22)

BFS 从给定的源顶点 s 开始，逐层向外扩展，探索图中的顶点。它首先访问所有与源顶点 s 直接相邻的顶点（距离为1），然后是与这些顶点相邻的、尚未访问过的顶点（距离为2），以此类推。

1. **基本思想与目标** (LEC9, p.11, 12):
   - **输入**：一个图 $G = (V, E)$（可以是有向图或无向图）和一个源顶点 $s \in V$ 。
   - **目标**:
     - 探索图的边，以“发现”从源顶点 s 可以到达的每一个顶点 。
     - 按照离源顶点 s 的距离（边的数量）**由近及远**的顺序访问顶点 。
     - 计算从 s 到每个可达顶点的**最短路径距离**（以边的数量计）。
     - 构建一棵以 s 为根的**广度优先树 (Breadth-First Tree)**，该树包含了所有从 s 可达的顶点 。
2. **算法过程与数据结构** (LEC9, p.13, 15, 16):
   - **颜色标记**：为了跟踪顶点的状态，每个顶点被赋予一种颜色：
     - **WHITE (白色)**：顶点尚未被发现 。
     - **GRAY (灰色)**：顶点已被发现，但其邻接顶点尚未全部被检查 。灰色顶点构成一个“边界”或“前沿”。
     - **BLACK (黑色)**：顶点已被发现，并且其所有邻接顶点都已被检查 。
   - **FIFO 队列 (Queue)**：使用一个先进先出 (FIFO) 队列 Q 来存储所有已被发现但其邻接顶点尚未完全探索的灰色顶点 。
   - **附加数据结构** (LEC9, p.16):
     - `color[u]`： 顶点 u 的颜色 。
     - `d[u]`： 从源顶点 s 到顶点 u 的距离（最短路径上的边数）。
     - `π[u]`： 顶点 u 在广度优先树中的前驱（父节点）。如果 $u = s$ 或者是尚未发现的顶点，则 `π[u] = NIL` 。
3. **BFS 算法伪代码** `BFS(V, E, s)` (LEC9, p.17, 18):

```
BFS(G, s) { // G=(V,E) 是图，s是源顶点
    // 1. 初始化 (LEC9, p.17)
    for each vertex u in V - {s} {
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
    }
    color[s] = GRAY
    d[s] = 0
    π[s] = NIL

    // 2. 初始化队列 Q 并将源顶点 s 入队 (LEC9, p.17)
    Q = EMPTY_QUEUE()
    ENQUEUE(Q, s)

    // 3. 主循环 (LEC9, p.18)
    while Q is not empty {
        u = DEQUEUE(Q) // 取出队首顶点 u
        for each vertex v in Adj[u] { // 遍历 u 的所有邻接顶点 v
            if color[v] == WHITE then { // 如果 v 是白色（未被发现）
                color[v] = GRAY
                d[v] = d[u] + 1
                π[v] = u
                ENQUEUE(Q, v) // 将 v 入队
            }
        }
        color[u] = BLACK // u 的所有邻接点都已检查完毕，将 u 设为黑色
    }
}
```

4. **广度优先树 (Breadth-First Tree)** (LEC9, p.15):
   - BFS 算法在执行过程中会隐式地构建一棵广度优先树 。
   - 树的根是源顶点 s 。
   - 当在扫描顶点 u 的邻接表时发现了一个白色顶点 v，边 (u, v) 就被加入到广度优先树中，并且 u 成为 v 的父节点（即 $\pi[v] = u$）。
   - 由于每个顶点只被发现一次（颜色从白色变为灰色时），所以它在广度优先树中最多只有一个父节点 。
   - 从源点 s 到任何可达顶点 v 在广度优先树中的路径，就是从 s 到 v 的一条最短路径（以边的数量衡量）。
5. **效率分析** (LEC9, p.20, 21):
   - **初始化**：$\Theta(V)$ 时间 。
   - **队列操作**：每个顶点最多入队一次 (ENQUEUE) 和出队一次 (DEQUEUE)。每次操作是 $\Theta(1)$。总共 $\Theta(V)$ 时间 。
   - **邻接表扫描**：每个顶点的邻接表最多只被扫描一次（当该顶点从队列中取出时）。扫描所有顶点的邻接表的总时间，对于有向图是 $\sum_{u \in V} |\text{Adj}[u]| = \Theta(E)$；对于无向图是 $\sum_{u \in V} |\text{Adj}[u]| = \Theta(2E) = \Theta(E)$ 。
   - **总时间复杂度**：$\Theta(V + E)$ 。这是基于图采用邻接表表示的情况。
6. **BFS 的特性**:

- 能够找到从源点 s 到所有可达顶点的最短路径（以边的数量计）。这是 `d[v]` 值的含义。
- 如果图是非连通的（对于无向图）或从 s 无法到达某些顶点（对于有向图），那么这些不可达顶点的 `d` 值将保持为 ∞，颜色保持为 WHITE。

7. **例子** (LEC9, p.19):
   PPT 中有一个详细的图示，展示了BFS从源顶点 s 开始，如何逐层发现顶点 w, r, t, x, v, u, y，以及队列 Q、各顶点的颜色、距离 d 和前驱 π 的变化过程。

B. 深度优先搜索 (Depth-First Search - DFS) (LEC9, pp.23-36)

DFS 尽可能深地探索图的分支。它从一个顶点开始，沿着一条路径访问顶点，直到到达一个没有未访问邻接点的顶点，然后回溯到前一个顶点，尝试其他未访问的路径。

1. **基本思想与目标** (LEC9, p.23, 24):
   - **输入**：一个图 $G = (V, E)$（可以是有向图或无向图）。DFS 不一定需要指定源顶点，它可以从任意未访问的顶点开始，并可能形成一个深度优先森林 。
   - **目标**:
     - 尽可能深地探索图的边，以发现图中的每一个顶点 。
     - 当一个顶点的所有出边都已被探索后，算法会“回溯”(backtrack) 到该顶点的父节点，继续探索其他分支 。
     - 如果图中仍有未被发现的顶点，则选择其中一个作为新的起点，重复搜索过程 。
   - **输出** (LEC9, p.23):
     - 为每个顶点 v 记录两个**时间戳 (timestamps)**:
       - `d[v]`：发现时间 (discovery time)，即 v 第一次被访问（颜色变为灰色）的时刻。
       - `f[v]`：完成时间 (finishing time)，即 v 的邻接表扫描完毕（颜色变为黑色）的时刻。
     - 构建一个**深度优先森林 (Depth-First Forest)**，由若干棵深度优先树组成。
2. **算法过程与数据结构** (LEC9, p.25):
   - **颜色标记**：与BFS类似，使用 WHITE, GRAY, BLACK 三种颜色 。
     - WHITE：顶点尚未被发现。
     - GRAY：顶点已被发现，但其邻接顶点尚未全部被探索完（即该顶点仍在递归栈中）。
     - BLACK：顶点已被发现，并且其所有邻接顶点都已被探索完（即对该顶点的递归调用已返回）。
   - **时间戳**：使用一个全局变量 `time`，从0开始计数。每当一个顶点被发现或完成时，`time` 递增 。
     - `d[u]`：记录顶点 u 从白色变为灰色时的 `time` 值。
     - `f[u]`：记录顶点 u 从灰色变为黑色时的 `time` 值。
     - 对于每个顶点 u，有 $1 \le d[u] < f[u] \le 2|V|$ 。
   - `π[u]`：顶点 u 在深度优先森林中的前驱（父节点）。
3. **DFS 算法伪代码** (LEC9, p.26, 27):
   DFS 通常由一个主过程 `DFS(G)` 和一个递归的辅助过程 `DFS-VISIT(u)` 组成。

```
DFS(G) { // G=(V,E)
    // 1. 初始化 (LEC9, p.26)
    for each vertex u in V {
        color[u] = WHITE
        π[u] = NIL
    }
    time = 0 // 全局时间戳

    // 2. 主循环，遍历所有顶点 (LEC9, p.26)
    for each vertex u in V {
        if color[u] == WHITE then {
            DFS-VISIT(u) // 如果 u 未被访问，则从 u 开始一次新的深度优先搜索
        }
    }
}

DFS-VISIT(u) { // 访问顶点 u (LEC9, p.27)
    // 1. 标记 u 为已发现 (灰色)，记录发现时间
    color[u] = GRAY
    time = time + 1
    d[u] = time

    // 2. 探索 u 的所有邻接顶点 v
    for each vertex v in Adj[u] {
        if color[v] == WHITE then { // 如果 v 未被发现
            π[v] = u
            DFS-VISIT(v) // 递归访问 v
        }
        // 如果 color[v] == GRAY，则 (u,v) 是一条反向边 (back edge)
        // 如果 color[v] == BLACK，则 (u,v) 是一条前向边或交叉边
    }

    // 3. u 的邻接表已扫描完毕，标记 u 为已完成 (黑色)，记录完成时间
    color[u] = BLACK
    time = time + 1
    f[u] = time
}
```

4. **深度优先森林 (Depth-First Forest)**:
    - `DFS` 主过程中的循环确保了图中的每个顶点都会被访问到。
    - 每次调用 `DFS-VISIT(u)` 时，如果 `u` 是白色的，那么 `u` 就成为深度优先森林中一棵新树的根 。
    - 边 (u, v) 如果在 `DFS-VISIT(u)` 中导致对 `DFS-VISIT(v)` 的递归调用（即 v 是白色的），则 (u, v) 是一条**树边 (tree edge)**，它构成了深度优先森林的一部分。
5. **DFS 的特性与边的分类** (LEC9, pp.28-31, 33-36):
    - **括号定理 (Parenthesis Theorem)** (LEC9, p.35)：在一个图 G 的任何深度优先搜索中，对于任意两个顶点 u 和 v，它们的发现时间和完成时间区间 $[d[u], f[u]]$ 和 $[d[v], f[v]]$ 之间的关系只有以下三种情况之一：
        a. 两个区间完全不相交，且 u 和 v 互不为对方的后代。

    b. 区间 $[d[v], f[v]]$ 完全包含在区间 $[d[u], f[u]]$ 之内，此时 v 是 u 在深度优先树中的后代。
    c. 区间 $[d[u], f[u]]$ 完全包含在区间 $[d[v], f[v]]$ 之内，此时 u 是 v 在深度优先树中的后代。
    - 这个定理说明了DFS过程中顶点访问和完成的嵌套结构。
  - **推论 (后代关系)** (LEC9, p.36)：顶点 v 是顶点 u 在深度优先森林中的一个真后代，当且仅当 $d[u] < d[v] < f[v] < f[u]$ 。
  - **白路径定理 (White-Path Theorem)** (LEC9, p.36)：在图 G 的深度优先森林中，顶点 v 是顶点 u 的后代，当且仅当在时刻 $d[u]$ （即 u 被发现时），存在一条从 u 到 v 的路径，该路径完全由白色顶点组成（v 除外，它在被发现前也是白色的）。
  - **边的分类** (LEC9, p.28, 31)：DFS 探索过程中遇到的边可以根据被探索顶点 v (当从 u 探索到 v 时) 的颜色进行分类：
    a. **树边 (Tree edge)**：v 是白色的。边 (u, v) 成为深度优先森林的一部分 。
    b. **反向边 (Back edge)**：v 是灰色的。边 (u, v) 连接顶点 u 到其在深度优先树中的一个祖先 v 。在有向图中，自环也是反向边 。反向边的存在意味着图中有环。
    c. **前向边 (Forward edge)**：v 是黑色的，并且 $d[u] < d[v]$ 。边 (u, v) 连接顶点 u 到其在深度优先树中的一个后代 v (非树边) 。
    d. **交叉边 (Cross edge)**：v 是黑色的，并且 $d[u] > d[v]$ 。边 (u, v) 可以连接同一棵深度优先树中没有祖先后代关系的两个顶点，或者连接不同深度优先树中的顶点 。
6. **效率分析** (LEC9, p.32, 33)：
  - `DFS(G)` **主过程**：初始化循环 $\Theta(V)$ 。第二个 `for` 循环对每个顶点调用一次，不包括 `DFS-VISIT` 的时间也是 $\Theta(V)$ 。
  - `DFS-VISIT(u)`：对每个顶点 u 只调用一次（当它还是白色时）。在 `DFS-VISIT(u)` 内部，`for each v in Adj[u]` 循环会遍历 u 的所有邻接边。因此，所有 `DFS-VISIT` 调用中，这个循环的总执行次数与图中边的数量成正比。
  - **总时间复杂度**：$\sum_{u \in V}(1 + |\mathrm{Adj}[u]|) = \Theta(V) + \Theta(E) = \Theta(V + E)$ 。 这是基于图采用邻接表表示的情况。
7. **例子** (LEC9, p.29, 30)：
  PPT 中给出了一个详细的图示，展示了DFS的执行过程，包括顶点的发现时间 `d` 和完成时间 `f` 的标记，以及形成的深度优先森林。

8.2 小结及备考提示 (BFS & DFS)

- **简答题考点**：
  - 描述 BFS 的基本工作原理和特性（逐层、最短路径）。
  - 描述 DFS 的基本工作原理和特性（深入、回溯、时间戳）。
  - BFS 和 DFS 在探索顺序上的主要区别是什么?
  - BFS 使用什么数据结构来管理待访问节点？DFS 呢？（BFS用队列，DFS隐式用栈/递归栈）
  - 解释 BFS/DFS 中顶点颜色的含义。
  - 解释 DFS 中的时间戳 $d[u]$ 和 $f[u]$ 的含义以及括号定理。
  - 列举 DFS 中可能的边的类型（树边、反向边、前向边、交叉边）并解释其含义。
- **算法分析题考点**：

    - 给定一个图和起始顶点，手动模拟 BFS 或 DFS 的过程，画出广度/深度优先树（森林），并标出距离 `d` （BFS中）或时间戳 `d, f` （DFS中）以及前驱 `π` 。
    - 分析 BFS 和 DFS 的时间复杂度，并解释为什么是 $\Theta(V+E)$。
    - 根据 DFS 的时间戳判断两个顶点之间是否存在祖先-后代关系。
    - 根据 DFS 过程识别图中的边属于哪种类型。
- **设计证明题考点**:
    - 写出 BFS 或 DFS (包括 `DFS-VISIT`) 的伪代码。
    - 证明 BFS 找到的是最短路径（按边数计）。
    - 证明 DFS 的括号定理或白路径定理。

图算法详解之 8.3 (A)：拓扑排序 (Topological Sort) (LEC9)

拓扑排序是对**有向无环图 (Directed Acyclic Graph - DAG)** 的顶点进行线性排序，使得对于图中任意一条有向边 (u, v)，顶点 u 在排序中都出现在顶点 v 之前。这种排序在表示事件的先后顺序或任务的依赖关系时非常有用。

1. 问题描述与应用 (LEC9, p.37)

- **定义**：有向无环图 $G=(V,E)$ 的拓扑排序是 V 中所有顶点的一个线性序列，满足：如果图 G 包含一条边 (u,v)，则在该序列中 u 出现在 v 之前 。
- **应用场景**:
    - 课程安排：某些课程是其他课程的先修课程。
    - 项目管理：任务之间存在依赖关系，某些任务必须在其他任务开始前完成。
    - 编译依赖：源代码文件之间的编译顺序。
    - PPT中举例 (LEC9, p.38)：穿衣顺序（例如，必须先穿袜子再穿鞋子）。
- 一个DAG的拓扑排序可能不唯一。
- **关键特性**：只有DAG才有拓扑排序。如果一个有向图包含环，则无法进行拓扑排序，因为环中的顶点无法满足“一个在另一个之前”的线性顺序要求。

2. 基于深度优先搜索 (DFS) 的拓扑排序算法 (LEC9, p.39)

一种常用的拓扑排序算法是基于深度优先搜索 (DFS):

1. **执行DFS**：对输入的DAG $G=(V,E)$ 调用 `DFS(G)` 算法，计算每个顶点 v 的**完成时间** $f[v]$。
2. **按完成时间逆序排列**：当每个顶点完成时（即其颜色变为黑色，所有后代都已被访问完毕），将其插入到一个链表的前端。
3. **输出结果**：最终得到的链表中的顶点顺序即为一个拓扑排序。也就是说，拓扑排序的结果是顶点按照其DFS完成时间的降序排列。

- **算法** `TOPOLOGICAL-SORT(V, E)` (LEC9, p.39):

```
TOPOLOGICAL-SORT(G) {
    1. Call DFS(G) to compute finishing times f[v] for each vertex v.
    2. As each vertex is finished (in DFS-VISIT, just before setting color to BLACK),
       insert it onto the front of a linked list.
    3. Return the linked list of vertices.
}
```

- **例子 (LEC9, p.39)**:
  PPT中给出了一个穿衣顺序的DAG，并标出了DFS过程中每个顶点的发现时间/完成时间 (d/f)。
  例如:
  - undershorts: 11/16
  - socks: 17/18
  - pants: 12/15
  - shoes: 13/14
  - watch: 9/10
  - shirt: 1/8
  - belt: 6/7
  - tie: 2/5
  - jacket: 3/4

  按照完成时间从大到小（即插入链表前端的顺序）排列:
  socks (18), undershorts (16), pants (15), shoes (14), watch (10), shirt (8), belt (7), tie (5), jacket (4)。
  这就是一个拓扑排序结果。

3. 算法正确性 (LEC9, p.39, 40)

- **引理1 (LEC9, p.40)**：一个有向图 G 是无环的 (DAG) 当且仅当对其进行深度优先搜索不产生任何**反向边 (back edges)**。
  - **证明思路 (LEC9, p.40)**:
    - **($\Rightarrow$) 无环 $\Rightarrow$ 无反向边**：如果图是无环的，那么在DFS访问过程中，当我们从顶点 u 探索到其邻接点 v 时，如果 v 是灰色的（即 v 是 u 的祖先且仍在递归栈中），则意味着存在一条从 v 到 u 的路径（树边构成），再加上边 (u, v) 就构成了一个环，这与图无环矛盾。所以不会有反向边。
    - **($\Leftarrow$) 无反向边 $\Rightarrow$ 无环**：假设图有环。设 v 是环中第一个被DFS发现的顶点，设 (u, v) 是环中指向 v 的边。在时刻 $d[v]$，从 v 到 u 的路径（环中除去边 (u, v) 的部分）上的所有顶点都是白色的（除了 v 自身刚变灰）。根据白路径定理， u 将成为 v 在深度优先树中的后代。因此，当DFS从 u 探索到边 (u, v) 时，v 仍然是灰色的（因为 v 是 u 的祖先），所以边 (u, v) 会被分类为反向边。这与假设无反向边矛盾。所以图必然无环。
- **拓扑排序算法正确性证明**:
  我们需要证明，如果算法输出一个序列 $u_1, u_2, \ldots, u_n$，那么对于图中任意边 (u_i, u_j)，都有 $i < j$（即 u_i 在 u_j 之前）。
  考虑图中任意一条边 (u, v)。当DFS探索这条边时，根据DFS的性质和边的分类:
  i. 如果 v 是白色的，则 v 成为 u 的后代，$DFS - VISIT(v)$ 会在 $DFS - VISIT(u)$ 完成前完成。因此，$f[v] < f[u]$。
  ii. 如果 v 是灰色的，则 (u, v) 是一条反向边。根据上述引理，这意味着图中存在环，但我们处理的是DAG，所以这种情况不会发生。

iii. 如果 v 是黑色的，则 v 已经被访问并完成。此时 $f[v]$ 已经确定。由于 v 是从 u 可达的，并且 v 已经完成，所以 v 不可能是 u 的祖先（否则 v 应该是灰色的）。因此，要么 v 是 u 的后代（但 u 还在探索，这意味着 (u, v) 是一条前向边，此时 $f[v] < f[u]$），要么 u 和 v 没有祖先后代关系（(u, v) 是交叉边，此时也必有 $f[v] < f[u]$，因为如果 $f[v] > f[u]$，则意味着当访问 v 时 u 是白色的，那么 u 会成为 v 的后代，导致 (u, v) 不会是 v 探索时遇到的边，除非是 v 到 u 的反向边，这与边是 (u, v) 矛盾）。
    - 更简洁的论证：对于DAG中的任意边 (u, v)，必有 $f[v] < f[u]$。否则，如果 $f[v] > f[u]$，那么当DFS访问 u 时，v 要么是白色（则 v 成为 u 的后代，$f[v]$ 会在 $f[u]$ 之前结束，矛盾），要么是灰色（反向边，DAG中不允许），要么是黑色（v 已完成，但如果 $f[v] > f[u]$，这意味着当处理 u 时 v 已经完成，且 (u, v) 存在，这通常指向 v 不是 u 的后代，此时 $f[v]$ 的值在 $f[u]$ 之前就固定了）。
    - 关键在于，对于DAG中的边 (u, v)，当 $DFS - VISIT(u)$ 正在进行并探索到 v 时，如果 v 是灰色，则有环。如果 v 是白色，则 v 会在 u 之前完成 ($f[v] < f[u]$)。如果 v 是黑色，则 v 已经完成了，所以 $f[v]$ 已经确定，并且 $f[v]$ 也必然小于 $f[u]$ （因为如果 $f[v] > f[u]$，当 v 完成时 u 还没有完成，这意味着 (u, v) 不可能是一条从 u 到已完成的 v 的边，除非 u 是 v 的祖先，但那样 u 会在 v 之后完成）。
    - **标准结论**：对于DAG中的任意边 (u, v)，在DFS完成后，总有 $f[u] > f[v]$。
      因此，当我们将顶点按照完成时间的降序排列（即将完成早的顶点放在链表后面，完成晚的放在前面），对于任何边 (u, v)，u (完成时间 $f[u]$ 大) 会在 v (完成时间 $f[v]$ 小) 之前出现在链表中。

4. 效率分析 (LEC9, p.39)

- 调用 `DFS(G)` 的时间复杂度为 $\Theta(V + E)$。
- 在每个顶点完成时将其插入链表前端的操作是 $\Theta(1)$。总共有 $|V|$ 个顶点，所以这部分是 $\Theta(V)$。
- 因此，拓扑排序算法的总时间复杂度是 $\Theta(V + E)$。

5. 备考要点

- **简答题考点**：
    - 什么是拓扑排序？它适用于什么类型的图？（DAG）
    - 拓扑排序在哪些实际问题中有应用？
    - 描述基于DFS的拓扑排序算法的基本步骤。
    - 为什么有环图不能进行拓扑排序？（或者说，DFS如何检测环并说明与拓扑排序的关系：反向边的存在）
- **算法分析题考点**：
    - 给定一个DAG，手动模拟DFS过程，标出发现时间和完成时间，并给出拓扑排序的结果。
    - 分析基于DFS的拓扑排序算法的时间复杂度。
- **设计证明题考点**：
    - 描述基于DFS的拓扑排序算法的伪代码。
    - 证明基于DFS的拓扑排序算法的正确性（核心是证明对于DAG中的任意边 (u, v)，有 $f[u] > f[v]$）。
    - 证明一个有向图是DAG当且仅当其DFS不产生反向边。

图算法详解之 8.3 (B)：强连通分量 (Strongly Connected Components - SCC) (LEC9)

强连通分量是针对**有向图**的一个重要概念。它将图中的顶点划分为若干个子集，在每个子集内部，任意两个顶点都是相互可达的。

1. 问题描述与定义 (LEC9, p.42)

- **输入**：一个有向图 $G = (V, E)$。
- **强连通 (Strongly Connected)**：在有向图 G 中，如果对于两个顶点 u 和 v，既存在从 u 到 v 的路径，也存在从 v 到 u 的路径，则称 u 和 v 是强连通的。
- **强连通分量 (Strongly Connected Component - SCC)**：有向图 G 的一个强连通分量是其顶点集 V 的一个**极大**子集 $C \subseteq V$，使得对于 C 中的任意一对顶点 $u, v \in C$， u 和 v 都是强连通的。
  - “极大”意味着如果再加入 C 之外的任何顶点到 C 中，这个新的集合就不再是强连通的了。
- **性质**：
  - 图 G 的顶点集 V 可以被唯一地划分为若干个不相交的强连通分量。
  - 即使图中存在环，也可以找到其SCC。

例子 (LEC9, p.42)：
PPT中给出了一个有向图，并标出了其SCC。例如，顶点 $\{a, b, e\}$ 构成一个SCC，$\{c, d\}$ 构成一个SCC，$\{f, g\}$ 构成一个SCC，$\{h\}$ 单独构成一个SCC。

2. 分量图 (Component Graph) (LEC9, p.45)

- 我们可以将原图 G 的每个强连通分量看作一个“超级顶点”，从而构造出一个新的图，称为**分量图** $G^{SCC} = (V^{SCC}, E^{SCC})$。
  - $V^{SCC} = \{v_1, v_2, \ldots, v_k\}$，其中 v_i 对应原图 G 的一个SCC C_i。
  - 如果原图 G 中存在一条边 (x, y)，其中 $x \in C_i$ 且 $y \in C_j$ $(i \neq j)$，则在分量图 G^{SCC} 中存在一条从 v_i 到 v_j 的边 $(v_i, v_j) \in E^{SCC}$。
- **重要性质**：分量图 G^{SCC} **总是一个有向无环图 (DAG)** 。
  - **证明思路**：如果 G^{SCC} 中存在一个环，例如 $v_i \to v_j \to \cdots \to v_i$，这意味着在原图 G 中，从 C_i 中的某个点出发可以到达 C_j 中的某个点，再从 C_j 中的点可以到达下一个分量的点，最终可以回到 C_i 中的某个点。这表明 $C_i, C_j, \ldots$ 中的所有顶点实际上都应该属于同一个更大的强连通分量，这与它们是独立的、极大的SCC的定义矛盾。

3. 基于两次DFS的SCC算法 (Kosaraju-Sharir Algorithm) (LEC9, p.44)

这是一个经典的、基于两次深度优先搜索来找到有向图中所有强连通分量的算法。

- **图的转置 (Transpose of a Graph, G^T)** (LEC9, p.43)：
  - 给定有向图 $G = (V, E)$，其转置图 $G^T = (V, E^T)$ 拥有与 G 相同的顶点集 V。
  - G^T 的边集 E^T 是将 G 中所有边的方向反向得到的：$E^T = \{(u, v) \mid (v, u) \in E\}$。
  - 如果 G 用邻接表表示，可以在 $\Theta(V + E)$ 时间内计算出 G^T 的邻接表。
  - **重要性质**：图 G 和其转置图 G^T **具有完全相同的强连通分量** 。因为如果 u 和 v 在 G 中是强连通的（即 $u \rightsquigarrow v$ 且 $v \rightsquigarrow u$），那么在 G^T 中，由于边反向，仍然有 $u \rightsquigarrow v$ （对应 G 中的 $v \rightsquigarrow u$）和 $v \rightsquigarrow u$

(对应 G 中的 $u \rightsquigarrow v$)。

- **算法步骤** `STRONGLY-CONNECTED-COMPONENTS(G)` (LEC9, p.44):
  i. **第一次DFS**: 对原图 G 调用 `DFS(G)`，计算每个顶点 u 的**完成时间** $f[u]$。
  ii. **计算转置图**: 计算 G^T。
  iii. **第二次DFS**: 对转置图 G^T 调用 `DFS(G^T)`。在这次DFS的主循环中，**必须按照第一步计算出的顶点完成时间 $f[u]$ 的降序来依次选择顶点作为DFS的起点**。即，优先从 $f[u]$ 值最大的未访问顶点开始新的深度优先搜索树。
  iv. **输出结果**: 在第二次DFS（在 G^T 上进行）所生成的深度优先森林中，**每一棵树的顶点恰好构成原图 G 的一个强连通分量**。
- **例子 (LEC9, p.46-47)**:
  - PPT中给出了一个图 G。
  - **第一次DFS (on G)** (LEC9, p.46): 计算出所有顶点的 $f[u]$。例如，顶点 b 的 $f[b]$ 可能最大。
  - **计算 G^T**。
  - **第二次DFS (on G^T)** (LEC9, p.47): 按照 $f[u]$ 降序处理顶点。
    - 假设 $f[b]$ 最大，从 b 开始在 G^T 上DFS，访问到的顶点 $\{b, a, e\}$ 构成一个SCC。
    - 然后在剩余未访问顶点中找 $f[u]$ 次大的，例如是 c，从 c 开始在 G^T 上DFS，访问到的 $\{c, d\}$ 构成一个SCC。
    - 依此类推，直到所有顶点都被访问。

4. 算法正确性直观理解 (为何两次DFS有效？)

- **第一次DFS的目的**: 计算完成时间 $f[u]$。$f[u]$ 较大的顶点，在某种意义上，是图结构中"较早完成探索其后代"的顶点，或者说是分量图 G^{SCC} 中"汇点分量"（没有出边指向其他分量的SCC）中的顶点，或者是能到达这些汇点分量的分量中的顶点，它们的完成时间会相对较高。
- **第二次DFS的目的**: 在 G^T 中，边的方向反转了。如果我们按照 $f[u]$ 的降序在 G^T 中启动DFS，第一个启动点 u (具有最大 $f[u]$ 值) 必定属于 G^{SCC} 中的一个"源点分量"（即在 G^{SCC} 中没有入边的SCC，对应于原图 G 中的一个"汇点分量"）。从这个 u 出发在 G^T 中进行DFS，能够访问到的所有顶点，恰好就是原图 G 中与 u 属于同一个强连通分量的所有顶点。
  - 这是因为，如果 u 属于某个SCC C_1，并且在 G 中存在从 C_1 到另一个SCC C_2 的边，那么在 G^T 中就存在从 C_2 到 C_1 的边。由于 u 的 $f[u]$ 值最高（或在当前未访问顶点中最高），它所在的 C_1 是 G^{SCC} 中的一个源点分量（或在剩余图的 G^{SCC} 中的源点分量）。因此，从 u 开始的DFS无法通过 G^T 中的边跳出到那些在 G 中"指向" C_1 的分量。它只能遍历到 C_1 内部的所有顶点。
- **符号标记 $d(U)$ 和 $f(U)$** (LEC9, p.48):
  - 对于顶点集合 $U \subseteq V$:
    - $d(U) = \min_{u \in U}\{d[u]\}$ (U中顶点的最早发现时间)
    - $f(U) = \max_{u \in U}\{f[u]\}$ (U中顶点的最晚完成时间)
  - 一个关键的引理是（未在PPT中直接编号，但属于正确性证明的一部分）：如果 C 和 C' 是 G 的两个不同的强连通分量，并且在 G 中存在一条边 (u, v) 其中 $u \in C, v \in C'$，那么在第一次DFS (on G) 后，必有 $f(C) > f(C')$。这意味着分量图的拓扑序与 f 值的降序是一致的。

5. 效率分析

- 第一次DFS: $\Theta(V + E)$。
- 计算 G^T: $\Theta(V + E)$ (如果用邻接表)。

- 第二次DFS：$\Theta(V+E)$。
- 因此，寻找所有强连通分量的总时间复杂度为 $\Theta(V+E)$。

6. 备考要点

- **简答题考点**：
  - 什么是强连通？什么是强连通分量 (SCC)?
  - 描述分量图 G^{SCC} 的概念及其性质（DAG）。
  - 简述基于两次DFS的SCC算法 (Kosaraju算法) 的主要步骤。
  - 在第二次DFS中，为什么需要按照第一次DFS完成时间的降序来选择起点?
  - 图 G 和其转置图 G^T 在SCC方面有什么关系?
- **算法分析题考点**：
  - 给定一个有向图，手动模拟Kosaraju算法的过程：
    a. 执行第一次DFS，得到所有顶点的 $f[u]$。
    b. 画出转置图 G^T。
    c. 在 G^T 上按 $f[u]$ 降序执行第二次DFS，找出所有的SCC。
  - 分析该算法的时间复杂度。
- **设计证明题考点**：
  - 描述Kosaraju算法的伪代码。
  - 证明分量图 G^{SCC} 是一个DAG。
  - (较难) 证明Kosaraju算法的正确性，特别是第二次DFS的起始顺序的重要性。

图算法详解之 8.4 (A)：最小生成树 (Minimum Spanning Trees - MST) - 基本概念与通用算法 (LEC10)

最小生成树 (MST) 问题是在一个连通的、无向的、加权图中找到一棵包含图中所有顶点且总权重最小的生成树。

1. 问题描述与背景 (LEC10, p.1)

- **场景**：想象一个城镇有多座房屋，房屋之间可以通过修建道路来连接。每条潜在的道路都有一个修建或维修的成本 。
- **目标**：选择修建（或保留）一部分道路，使得：
  i. 所有房屋保持连通（即从任何一座房屋都可以到达其他任何一座房屋）。
  ii. 修建（或保留）的道路总成本最低 。
- **图模型** (LEC10, p.1)：
  - 这是一个连通的、无向的图 $G=(V,E)$ 。
  - 顶点 (Vertices) V 代表房屋 。
  - 边 (Edges) E 代表潜在的道路 。
  - 每条边 $(u,v)\in E$ 都有一个权重 $w(u,v)$，代表修建或维修该道路的成本 。
- **任务**：找到边集 E 的一个子集 $T\subseteq E$，使得：
  i. T 连接了图中的所有顶点（即 (V,T) 是一棵生成树）。
  ii. T 的总权重 $w(T)=\sum_{(u,v)\in T}w(u,v)$ 最小化 。

- 这样的子集 T 构成一棵**生成树 (Spanning Tree)** 。
- 在所有可能的生成树中，总权重最小的那棵（或那些）树被称为**最小生成树 (Minimum Spanning Tree - MST)** 。

2. 最小生成树的特性 (LEC10, p.1)

- **不唯一性 (Not Unique)**：一个图的最小生成树可能不是唯一的 。例如，如果图中存在两条权重相同且可以互相替换而不影响连通性和总权重的边，那么就可能存在多个不同的MST 。
- **无环性 (No Cycles)**：最小生成树（以及任何生成树）中都不能包含环路 。如果生成树中存在环路，我们可以移除环上的一条边，仍然保持所有顶点连通，但总权重会减少（或者不变，如果移除的是权重最大的边或环上权重非负），这与MST的最小权重定义矛盾（除非所有环上边的权重都为0，但通常我们考虑正权重，或者说移除任意边都可以减少“边数”这个次要目标）。
- **边的数量 (Number of Edges)**：对于一个包含 $|V|$ 个顶点的图，其任何一棵生成树（包括MST）都恰好有 $|V|-1$ 条边 。

3. 构造MST的通用方法 (Generic MST Algorithm) (LEC10, p.1)

许多MST算法都遵循一个通用的贪心策略来逐步构建MST。

- **基本思想** (LEC10, p.1):
  i. 维护一个边的集合 A，初始时为空 ($A = \emptyset$) 。
  ii. 逐步向集合 A 中添加边，每次添加的边都必须是“安全”的，即添加该边后，A 仍然是某个MST的子集 。
  iii. 当 A 中的边恰好构成一棵生成树时（即包含 $|V|-1$ 条边且连接所有顶点），算法结束。
- **安全边 (Safe Edge)** (LEC10, p.1)：一条边 (u, v) 对于当前的边集合 A 是安全的，当且仅当 $A \cup \{(u, v)\}$ 也是某个最小生成树的子集 。通用MST算法只会向 A 中添加安全边 。
- **通用算法框架 `GENERIC-MST(G, w)`** (LEC10, p.2):

```
GENERIC-MST(G, w) {
    A = {} // 初始化为空集

    while A does not form a spanning tree { // 当 A 中的边数少于 |V|-1 或 A 不连通时
        Find an edge (u,v) that is safe for A
        A = A U {(u,v)}
    }
    return A
}
```

这个框架的关键在于如何有效地找到一条安全边。

4. 如何找到安全边？切割 (Cut) 和轻量级边 (Light Edge) (LEC10, p.2)

为了具体化寻找安全边的方法，需要引入一些定义：

- **切割 (Cut)** (LEC10, p.2)：图 $G = (V, E)$ 的一个切割 $(S, V-S)$ 是对顶点集 V 的一种划分，将其分为两个互不相交的非空子集 S 和 $V-S$ 。

- **横跨切割的边 (Edge Crossing a Cut)** (LEC10, p.2)：如果一条边 $(u, v) \in E$ 的一个端点在集合 S 中，而另一个端点在集合 $V - S$ 中，则称这条边横跨切割 $(S, V - S)$ 。
- **切割尊重集合A (Cut Respects Set A)** (LEC10, p.2)：如果集合 A 中没有任何一条边横跨切割 $(S, V - S)$，则称该切割尊重集合 A 。
- **轻量级边 (Light Edge)** (LEC10, p.2)：在所有横跨某个特定切割 $(S, V - S)$ 的边中，权重最小的那条边（或那些边之一）被称为横跨该切割的一条轻量级边 。一个切割可能有多条权重相同的轻量级边 。
- **安全边的识别定理 (Theorem 23.1 in CLRS, LEC10, p.2)**:
  设 A 是图 G 的某个最小生成树 T_{MST} 的一个子集。设 $(S, V - S)$ 是图 G 中任意一个尊重集合 A 的切割。如果边 (u, v) 是横跨切割 $(S, V - S)$ 的一条轻量级边，那么边 (u, v) 对于集合 A 是安全的 。
  - **证明思路 (LEC10, pp.2-3)**:
    a. 假设MST T 包含 A 。如果 $(u, v) \in T$，则 (u, v) 对 A 安全。
    b. 如果 $(u, v) \notin T$ 。在 T 中添加边 (u, v) 会形成一个唯一的环路 。由于 $u \in S$ 和 $v \in V - S$ (或者反之)，这条环路必定会再次横跨切割 $(S, V - S)$，设这条边为 (x, y)，其中 $(x, y) \in T$ 。注意，(x, y) 不能属于 A，因为切割 $(S, V - S)$ 尊重 A。
    c. 由于 (u, v) 是横跨切割 $(S, V - S)$ 的轻量级边，所以 $w(u, v) \leq w(x, y)$ 。
    d. 构造一棵新的生成树 $T' = T - \{(x, y)\} \cup \{(u, v)\}$ 。T' 仍然是一棵生成树。
    e. T' 的总权重 $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$ 。
    f. 由于 T 是一棵MST，所以 $w(T) \leq w(T')$。因此，必有 $w(T') = w(T)$，所以 T' 也是一棵MST 。
    g. 因为 $A \subseteq T$ 且 $(x, y) \notin A$，所以 $A \subseteq T - \{(x, y)\}$。因此 $A \cup \{(u, v)\} \subseteq T'$ 。
    h. 由于 T' 是一棵MST，且 $A \cup \{(u, v)\}$ 是 T' 的子集，所以边 (u, v) 对于 A 是安全的 。
- **通用MST算法的推论 (LEC10, p.3)**:
  - 在算法的任何时刻，集合 A 中的边构成一个森林（包含若干棵树，即连通分量）。初始时，每个顶点自身构成一个独立的连通分量 。
  - 每当找到一条安全边并将其加入 A 时，这条边必然连接了森林中两个不同的连通分量，并将它们合并为一个新的连通分量 。
  - 因为MST恰好有 $|V| - 1$ 条边，所以通用算法在迭代 $|V| - 1$ 次后，森林将只包含一个连通分量，即所求的MST 。

5. 备考要点 (针对此部分)

- **简答题考点**:
  - 什么是最小生成树 (MST)？它的主要特性有哪些（无环，边数）？
  - 描述构造MST的通用贪心策略的基本思想。
  - 什么是“安全边”?
  - 定义图的“切割”、“横跨切割的边”、“尊重集合A的切割”以及“轻量级边”。
  - 陈述识别安全边的关键定理（即连接集合 A 和某个尊重 A 的切割的轻量级边是安全的）。
- **算法分析题考点**:
  - 理解安全边定理的证明思路，特别是交换论证的使用。
- **设计证明题考点**:
  - （较少直接考证明定理本身，但理解是关键）可能会问到通用MST算法的框架。

图算法详解之 8.4 (B)：Kruskal 算法 (LEC10)

Kruskal 算法是求解最小生成树 (MST) 的一种经典贪心算法。它遵循通用MST算法的框架，通过一种特定的方式来选择安全边。

1. 算法思想 (LEC10, p.3)

- **核心策略**：按照边的权重**从小到大**的顺序来考虑图中的所有边。
- **决策过程**：
  i. 初始时，将图中的每个顶点都看作一个独立的连通分量（即森林中的一棵独立的树）。
  ii. 依次检查按权重排好序的边。对于当前检查的边 (u, v)：
    - 如果顶点 u 和顶点 v 已经属于同一个连通分量（即添加边 (u, v) 会形成环路），则舍弃这条边。
    - 如果顶点 u 和顶点 v 属于不同的连通分量，则将这条边 (u, v) 加入到MST的边集合 A 中，并将这两个连通分量合并为一个。
  iii. 重复此过程，直到加入了 $|V| - 1$ 条边，此时所有顶点构成一个单一的连通分量，即MST。
- **如何判断顶点是否属于同一连通分量?**
  - Kruskal 算法通常使用**不相交集合数据结构 (Disjoint-Set Data Structure)**（也称为并查集，Union-Find）来高效地维护和查询顶点的连通分量信息。

2. 不相交集合数据结构的操作 (LEC10, p.3)

不相交集合数据结构维护了一组互不相交的动态集合。每个集合都有一个代表元（representative），用于标识该集合。它支持以下主要操作：

- **`MAKE-SET(u)`**：创建一个新的集合，其唯一成员是顶点 u。这个集合的代表元通常是 u 自身。
- **`FIND-SET(u)`**：返回包含顶点 u 的集合的代表元。如果 u 和 v 属于同一个集合，则 `FIND-SET(u)` 和 `FIND-SET(v)` 返回相同的代表元。
- **`UNION(u, v)`**：合并包含顶点 u 的集合和包含顶点 v 的集合（假设它们原先属于不同的集合）。通常是将一个集合的代表元指向另一个集合的代表元。

3. Kruskal 算法伪代码 `KRUSKAL(V, E, w)` (LEC10, p.3)

```
KRUSKAL(G, w) { // G=(V,E) 是图，w是权重函数
    A = {} // 初始化MST的边集合为空

    // 1. 为每个顶点创建一个不相交集合
    for each vertex v in V {
        MAKE-SET(v) //
    }

    // 2. 将图中所有边 E 按权重 w 非递减（从小到大）排序
    Sort the edges of E into non-decreasing order by weight w

    // 3. 遍历排序后的边
    for each edge (u,v) in E, taken in non-decreasing order by weight {
        // 4. 如果边 (u,v) 连接的两个顶点不在同一个连通分量中
        //    (即它们的代表元不同)
        if FIND-SET(u) != FIND-SET(v) then { //
            A = A U {(u,v)}     // 将边 (u,v) 加入到 A 中
            UNION(u,v)          // 合并 u 和 v 所在的集合
        }
    }
    return A
}
```

4. Kruskal 算法的正确性

Kruskal 算法选择边的策略符合通用MST算法中识别安全边的方法。

- **贪心选择**：算法总是选择当前所有尚未考虑的、且不会形成环路的边中权重最小的那条边。
- **安全性证明**：
  - 设 A 是算法在某个阶段已经选择的边的集合，这些边构成一个森林。
  - 设 (u, v) 是根据Kruskal策略接下来要选择的边，即 (u, v) 是所有连接 A 中不同连通分量的边中权重最小的一条。
  - 考虑一个切割 $(S, V - S)$，其中 S 是包含顶点 u 的连通分量中的所有顶点，而 $V - S$ 包含图中的其余所有顶点（包括顶点 v 所在的连通分量）。这个切割尊重 A（因为 A 中的边不连接不同连通分量，除非是刚要加入的边）。
  - 边 (u, v) 横跨这个切割。由于我们是按权重从小到大处理边的，并且 (u, v) 是第一条连接 S 和 $V - S$ (即 u 和 v 所在的不同连通分量) 的边，所以 (u, v) 必然是横跨切割 $(S, V - S)$ 的一条轻量级边。
  - 根据安全边的识别定理（LEC10, p.2），边 (u, v) 对于 A 是安全的。
  - 因此，Kruskal 算法每一步加入的边都是安全的，最终形成的边集合 A 是一个MST。

5. 效率分析 (LEC10, p.3)

- **初始化不相交集合**：对 $|V|$ 个顶点执行 `MAKE-SET` 操作。如果使用优化的不相交集合实现（如按秩合并和路径压缩），这部分总共是 $O(V)$ 或接近 $O(V)$。

- **边排序**：对 $|E|$ 条边按权重排序，使用基于比较的排序算法（如合并排序或堆排序），时间复杂度为 $O(E \log E)$。由于 $|E| \leq |V|^2$，所以 $\log E \leq \log(V^2) = 2 \log V$，因此 $O(E \log E)$ 通常也写作 $O(E \log V)$。
- **遍历排序后的边并执行不相交集合操作**：
  - 循环最多执行 $|E|$ 次。
  - 在循环内部，执行两次 `FIND-SET` 操作和（可能一次） `UNION` 操作。
  - 使用路径压缩和按秩（或按大小）合并的优化，一系列 m' 次不相交集合操作（包括 `MAKE-SET`，`FIND-SET`，`UNION`）的总时间几乎是线性的，可以认为是 $O(m'\alpha(V))$，其中 $\alpha(V)$ 是阿克曼函数的一个非常缓慢增长的反函数，对于所有实际的 $|V|$ 值，$\alpha(V)$ 不会超过5，因此可以近似看作常数。
  - 在Kruskal算法中，我们有 $|V|$ 次 `MAKE-SET`，最多 $2|E|$ 次 `FIND-SET`，以及最多 $|V|-1$ 次 `UNION`。总的不相交集合操作次数是 $O(E)$。因此这部分的时间复杂度接近 $O(E\alpha(V))$。
- **总时间复杂度**：主要由边的排序决定，即 $O(E \log E)$ 或 $O(E \log V)$。如果边的数量 $|E|$ 非常接近 $|V|^2$，则 $O(E \log V)$。如果 $|E|$ 接近 $|V|$（例如稀疏图），则 $O(V \log V)$（因为排序 $O(E \log E)$）。

PPT (LEC10, p.3) 中给出的运行时间是 "O(E lgV)-dependent on the implementation of the disjoint-set data structure"。通常我们认为排序是瓶颈。

6. 例子 (LEC10, p.4)

PPT中给出了一个详细的例子，展示了如何逐步选择边并合并连通分量：

1. 初始时，每个顶点 $\{a\}, \{b\}, \ldots, \{i\}$ 是一个独立的集合。
2. 按权重从小到大选择边：
   - (h,g) - weight 1: $A = \{(h,g)\}$, 合并 $\{h\}, \{g\}$ 为 $\{g,h\}$。
   - (c,i) - weight 2: $A = \{(h,g), (c,i)\}$, 合并 $\{c\}, \{i\}$ 为 $\{c,i\}$。
   - (g,f) - weight 2: $A = \{\ldots, (g,f)\}$, 合并 $\{g,h\}, \{f\}$ 为 $\{f,g,h\}$。
   - (a,b) - weight 4: $A = \{\ldots, (a,b)\}$, 合并 $\{a\}, \{b\}$ 为 $\{a,b\}$。
   - (c,f) - weight 4: $A = \{\ldots, (c,f)\}$, 合并 $\{c,i\}, \{f,g,h\}$ 为 $\{c,f,g,h,i\}$。
   - (i,g) - weight 6: $FIND-SET(i)$ 和 $FIND-SET(g)$ 返回相同代表元（因为它们已在 $\{c,f,g,h,i\}$ 中），舍弃。
   - (c,d) - weight 7: $A = \{\ldots, (c,d)\}$, 合并 $\{c,f,g,h,i\}, \{d\}$ 为 $\{c,d,f,g,h,i\}$。
   - (i,h) - weight 7: $FIND-SET(i)$ 和 $FIND-SET(h)$ 相同，舍弃。
   - (a,h) - weight 8: $A = \{\ldots, (a,h)\}$, 合并 $\{a,b\}, \{c,d,f,g,h,i\}$ 为 $\{a,b,c,d,f,g,h,i\}$。
   - (b,c) - weight 8: $FIND-SET(b)$ 和 $FIND-SET(c)$ 相同，舍弃。
   - (d,e) - weight 9: $A = \{\ldots, (d,e)\}$, 合并 $\{a,b,c,d,f,g,h,i\}, \{e\}$ 为 $\{a,b,c,d,e,f,g,h,i\}$。此时已有 $|V|-1 = 9-1 = 8$ 条边，MST形成。
   - 后续的边如 (e,f), (b,h), (d,f) 都会因为连接已在同一连通分量中的顶点而被舍弃。

7. 备考要点

- **简答题考点**：
  - 描述 Kruskal 算法的基本思想和步骤。
  - Kruskal 算法使用什么数据结构来判断添加一条边是否会形成环路？（不相交集合数据结构）

    - Kruskal 算法如何保证找到的是最小生成树？（它总是选择当前权重最小且不形成环的边，这条边是安全的）
- **算法分析题考点**：
    - 给定一个带权无向图，手动模拟 Kruskal 算法的执行过程，列出边的选择顺序和最终得到的MST。
    - 分析 Kruskal 算法的时间复杂度，并说明哪个步骤是主要的性能瓶颈。
- **设计证明题考点**：
    - 描述 Kruskal 算法的伪代码。
    - 证明 Kruskal 算法的正确性（通过论证其每一步选择的边都是安全的）。

图算法详解之 8.4 (C)：Prim 算法 (LEC10)

Prim 算法是另一种求解最小生成树 (MST) 的经典贪心算法。与 Kruskal 算法不同，Prim 算法从一个初始顶点开始，逐步扩展一棵树，直到包含图中所有顶点。

1. 算法思想 (LEC10, p.4)

- **核心策略**：算法维护一个不断增长的树 A（初始时只包含一个任意选择的起始顶点）。在每一步中，Prim 算法找到一条连接树 A 中的一个顶点与树 A 外的一个顶点的边，并且这条边是所有这类连接边中权重最小的。然后将这条权重最小的边和它连接的那个树外顶点加入到树 A 中。
- **决策过程**：
    i. 选择一个任意的起始顶点 r（作为树根），将其加入到集合 V_A（表示已在树中的顶点集合），MST的边集合 A 初始为空。
    ii. 重复以下步骤，直到 V_A 包含了图中所有的顶点 V（或者说，直到加入了 $|V|-1$ 条边）：
        - 找到一条**轻量级边 (light edge)** (u, v)，它横跨切割 $(V_A, V - V_A)$（即 $u \in V_A$ 且 $v \in V - V_A$）。也就是说，在所有一端在树中、另一端在树外的边中，选择权重最小的那条边。
        - 将这条轻量级边 (u, v) 加入到集合 A 中，并将顶点 v 加入到集合 V_A 中。
- **贪心特性** (LEC10, p.4)：在每一步，Prim 算法都选择一条使得当前树的权重增长最小的边来扩展树。

2. 如何高效地找到轻量级边？(LEC10, p.4)

为了高效地找到横跨切割 $(V_A, V - V_A)$ 的轻量级边，Prim 算法通常使用一个**最小优先队列 (min-priority queue)** Q。

- **优先队列 Q 的内容**：
    - Q 包含所有当前**不在树中**的顶点，即 $V - V_A$。
    - 对于 Q 中的每个顶点 v，我们关联一个**键值** `key[v]`。
    - `key[v]` **的含义** (LEC10, p.4)：表示连接顶点 v (在 Q 中) 与当前树 V_A 中的某个顶点的所有边中，权重最小的那条边的权重。如果 v 与 V_A 中的任何顶点都没有边直接相连，则 `key[v] = ∞`。
    - 我们还需要一个数组 `π[v]` 来存储当 `key[v]` 更新时，使得 `key[v]` 达到最小值的树中顶点 u（即边 (u, v) 是当前的候选轻量级边，$\pi[v] = u$）。
- **算法步骤使用优先队列**：

i. 初始化：对于起始顶点 r，设 `key[r] = 0$`，对于所有其他顶点 `$u \in V-\{r\}$`，设 key[u] = ∞ 。所有顶点的 π[u] = NIL`。将所有顶点加入最小优先队列 Q。

ii. 当 Q 不为空时，重复：

- **`u = EXTRACT-MIN(Q)`**：从 Q 中提取具有最小 `key` 值的顶点 u。这个顶点 u 和边 $(\pi[u], u)$（如果 $\pi[u] \neq NIL$）就被加入到MST中（即 u 加入 V_A）。
- 对于 u 的每个邻接顶点 $v \in \mathrm{Adj}[u]$:
  - 如果 v 仍然在队列 Q 中（即 $v \in V - V_A$）并且边 (u, v) 的权重 $w(u, v)$ 小于当前记录的 `key[v]`，则更新 `key[v] = w(u,v)$` 并设置 `$\pi[v] = u$`。这个操作相当于在优先队列中对元素 `$v$` 执行 DECREASE-KEY` 操作。

3. Prim 算法伪代码 `PRIM(V, E, w, r)` (LEC10, p.4)

```
PRIM(G, w, r) { // G=(V,E) 是图，w是权重函数，r是起始顶点
    // 1. 初始化
    for each vertex u in V {
        key[u] = ∞
        π[u] = NIL
    }
    key[r] = 0 // 起始顶点的key值为0，这样它会被第一个提取出来

    // 2. 将所有顶点加入最小优先队列 Q，以 key 值作为优先级
    Q = V // ( conceptually, Q contains all vertices of V )

    // 3. 主循环
    while Q is not empty {
        u = EXTRACT-MIN(Q) // 从Q中提取key值最小的顶点u

        // 将u加入到MST的顶点集合中 (隐式地通过从Q中移除)
        // 如果 u != r, 边 (π[u], u) 是MST的一条边

        // 4. 更新 u 的邻接顶点 v 的 key 值和 π 值
        for each vertex v in Adj[u] {
            if v is in Q and w(u,v) < key[v] then { // 如果v还在队列中且找到了更短的连接方式
                π[v] = u
                key[v] = w(u,v)
                // 通常这里会有一个 DECREASE-KEY(Q, v, key[v]) 操作
            }
        }
    }
}
```

- **注意**：在实际的优先队列实现中（如二叉堆或斐波那契堆），当 `key[v]` 更新时，需要调用 `DECREASE-KEY` 操作来调整 v 在队列中的位置。

4. Prim 算法的正确性

Prim 算法每一步选择的边都是连接当前已构建树 V_A 与树外顶点 $v \in V - V_A$ 的权重最小的边。这条边正是横跨切割 $(V_A, V - V_A)$ 的一条轻量级边。

- 在算法的每一步，集合 A（由已经选择的边 $(\pi[u], u)$ 构成，其中 u 是刚从 Q 中提取出来的顶点且 $u \neq r$）是某个MST的子集。
- 切割 $(V_A, V - V_A)$（其中 V_A 是已经从 Q 中提取出的顶点集合）尊重当前的 A。
- `EXTRACT-MIN(Q)` 选出的顶点 u 和对应的边 $(\pi[u], u)$ 就是横跨切割 $(V_A, V - V_A)$ 的轻量级边。
- 根据安全边的识别定理（LEC10, p.2），这条边对于 A 是安全的。
- 因此，Prim 算法能够正确地构造出一个MST。

5. 效率分析 (LEC10, p.5)

Prim 算法的效率依赖于最小优先队列 Q 的具体实现。设 $|V| = n, |E| = m$。

- **初始化**：设置 `key` 和 `π` 数组需要 $O(V)$。构建初始优先队列（如果所有顶点都加入）：
  - 如果用**二叉堆 (Binary Heap)** 实现 Q：建堆 $O(V)$。
- **主循环** `while Q is not empty`：执行 $|V|$ 次。
  - `EXTRACT-MIN(Q)`：
    - 二叉堆：$O(\log V)$。总共 $|V|$ 次，所以是 $O(V \log V)$。
  - **内层** `for each v in Adj[u]` **循环**：
    - 这个循环的总执行次数（对所有 u 而言）是 $\sum_{u \in V} \text{degree}(u) = 2|E|$（对于无向图）。
    - `if v is in Q`：检查 v 是否在队列中，如果用辅助布尔数组标记，是 $O(1)$。
    - `w(u,v) < key[v]`：比较是 $O(1)$。
    - **更新** `π[v]` **和** `key[v]`：$O(1)$。
    - `DECREASE-KEY(Q, v, key[v])` **操作**（当 `key[v]` 更新时）：
      - 二叉堆：$O(\log V)$。
- **总时间复杂度**：
  - **使用二叉堆**：ParseError: KaTeX parse error: Expected 'EOF', got '_' at position 17: …(V \text{ (init_Q)}) + O(V \log…。
    由于 $|E| \geq |V| - 1$ (对于连通图)，通常 $O(E \log V)$ 会主导 $O(V \log V)$。
    所以，时间复杂度为 $O((V + E) \log V)$，通常简化为 $O(E \log V)$ (因为 E 通常至少是 $V - 1$)。
    PPT (LEC10, p.5) 给出的总时间是 $O(V \log V + E \log V) = O(E \log V)$。
  - **使用斐波那契堆 (Fibonacci Heap)**：
    - `EXTRACT-MIN` 的摊销时间是 $O(\log V)$。总共 $O(V \log V)$。
    - `DECREASE-KEY` 的摊销时间是 $O(1)$。总共 $O(E)$ 次 `DECREASE-KEY` 操作，总时间 $O(E)$。
    - 建堆 $O(V)$。
    - 总时间复杂度为 $O(E + V \log V)$。这在边稠密的图中比二叉堆实现要好。

6. 例子 (LEC10, p.4, 5)

PPT中从顶点 'a' 开始，逐步展示了 `key` 值和 `π` 值的更新，以及顶点被 `EXTRACT-MIN` 的顺序：

1. $Q = \{a : 0, b : \infty, c : \infty, d : \infty, e : \infty, f : \infty, g : \infty, h : \infty, i : \infty\}, V_A = \emptyset$

2. `Extract a (key=0)` . $V_A = \{a\}$.
   Update neighbors: $key[b] = 4, \pi[b] = a$; $key[h] = 8, \pi[h] = a$.
   $Q = \{b:4, c:\infty, d:\infty, e:\infty, f:\infty, g:\infty, h:8, i:\infty\}$
3. `Extract b (key=4)` . $V_A = \{a, b\}$. (Edge (a,b) added to MST)
   Update neighbors of b: $key[c] = 8, \pi[c] = b$; $key[h]$ (curr 8 via a) vs $w(b,h) = 11$, no change.
   $Q = \{c:8, d:\infty, e:\infty, f:\infty, g:\infty, h:8, i:\infty\}$
4. `Extract c (key=8)` . $V_A = \{a, b, c\}$. (Edge (b,c) added to MST)
   Update neighbors of c: $key[d] = 7, \pi[d] = c$; $key[f] = 4, \pi[f] = c$; $key[i] = 2, \pi[i] = c$.
   $Q = \{d:7, e:\infty, f:4, g:\infty, h:8, i:2\}$
5. `Extract i (key=2)` . $V_A = \{a, b, c, i\}$. (Edge (c,i) added to MST)
   Update neighbors of i: $key[h]$ (curr 8 via a) vs $w(i,h) = 7$, $key[h] = 7, \pi[h] = i$; $key[g]$ (curr ∞) vs $w(i,g) = 6$, $key[g] = 6, \pi[g] = i$.
   $Q = \{d:7, e:\infty, f:4, g:6, h:7\}$
   ...以此类推。

7. 备考要点

- **简答题考点**:
  - 描述 Prim 算法的基本思想和步骤（如何逐步构建MST）。
  - Prim 算法使用什么数据结构来高效地选择下一个要加入树的边/顶点？（最小优先队列）
  - 在 Prim 算法中，优先队列中存储的顶点的 `key` 值代表什么？ `π` 值代表什么？
  - Prim 算法如何保证找到的是最小生成树？（它总是选择连接当前树和树外顶点的轻量级边）
- **算法分析题考点**:
  - 给定一个带权无向图和起始顶点，手动模拟 Prim 算法的执行过程，列出每一步优先队列的状态、 `key` 和 `π` 值的变化，以及最终得到的MST。
  - 分析 Prim 算法的时间复杂度，并说明其如何依赖于优先队列的实现（例如，二叉堆 vs 斐波那契堆）。
- **设计证明题考点**:
  - 描述 Prim 算法的伪代码。
  - 证明 Prim 算法的正确性（通过论证其每一步选择的边都是安全的，符合通用MST框架）。
  - 比较 Prim 算法和 Kruskal 算法的异同点（例如，Kruskal处理边，Prim处理顶点；Prim的树总是一个连通块，Kruskal的森林可能有多个）。

图算法详解之 8.5 (A)：单源最短路径问题 - 概览与基础 (LEC11)

单源最短路径问题是图论中一个非常基础且重要的问题，目标是找到从一个指定的源顶点到图中所有其他顶点的最短路径。

1. 问题描述 (LEC11, p.1)

- **输入**:
  - 一个**有向图** $G = (V, E)$。虽然很多算法也可以适用于无向图（通常将无向边视为两条方向相反的有向边），但基本定义和一些算法（如Bellman-Ford）更关注有向图。

    - 一个**权重函数** $w : E \to \mathbb{R}$，为图中的每条边 (u, v) 赋予一个实数值权重 $w(u, v)$。这个权重可以表示距离、时间、成本等。
    - 一个指定的**源顶点** $s \in V$。
- **路径的权重 (Weight of a path)**：一条路径 $p = \langle v_0, v_1, \ldots, v_k \rangle$ 的权重 $w(p)$ 是构成该路径的所有边的权重之和：
  $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$ 。
- **最短路径权重 (Shortest-path weight)**：从顶点 u 到顶点 v 的最短路径权重 $\delta(u, v)$ 定义为：
  $$\delta(u, v) = \begin{cases} \min\{w(p) \mid u \rightsquigarrow p \rightsquigarrow v\} & \text{如果存在从 } u \text{ 到 } v \text{ 的路径} \\ \infty & \text{否则} \end{cases}$$
- **最短路径 (Shortest path)**：从 u 到 v 的任何一条权重为 $\delta(u, v)$ 的路径都称为最短路径 。最短路径可能不唯一。
- **单源最短路径问题目标**：给定 G, w, s，对于所有 $v \in V$，找出 $\delta(s, v)$ 以及一条从 s 到 v 的最短路径。

2. 最短路径问题的不同形式 (Variants of Shortest Paths) (LEC11, p.1)

- **单源最短路径 (Single-Source Shortest Path - SSSP)**：从给定的源顶点 s 到图中其他所有顶点 $v \in V$ 的最短路径 。这是本节主要讨论的问题。
- **单目的地最短路径 (Single-Destination Shortest Path)**：找到从图中所有顶点 $v \in V$ 到某个给定的目的顶点 t 的最短路径 。可以通过在图的转置图 G^T（所有边反向）上运行从 t 出发的单源最短路径算法来解决 。
- **单点对最短路径 (Single-Pair Shortest Path)**：找到给定顶点 u 和 v 之间的最短路径 。通常，解决这个问题并没有比解决从 u 出发的单源最短路径问题在最坏情况下效率更高 。
- **所有点对最短路径 (All-Pairs Shortest Paths - APSP)**：找到图中每对顶点 (u, v) 之间的最短路径 。可以通过对每个顶点运行一次单源最短路径算法来解决，或者使用专门的APSP算法（如Floyd-Warshall）。

3. 最优子结构 (Optimal Substructure of Shortest Paths) (LEC11, p.1)

最短路径问题具有最优子结构性质，这是动态规划和贪心算法能够应用的基础。

- **定理**：一条最短路径的任何子路径也必定是其端点之间的最短路径 。
    - **证明 (反证法)** (LEC11, p.1)：设路径 $p = \langle v_1, v_2, \ldots, v_k \rangle$ 是从 v_1 到 v_k 的一条最短路径 。考虑其任意子路径 $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ (其中 $1 \le i \le j \le k$) 。如果存在一条从 v_i 到 v_j 的路径 p'_{ij} 其权重 $w(p'_{ij}) < w(p_{ij})$，那么我们可以用 p'_{ij} 替换 p 中的 p_{ij} 来构造一条从 v_1 到 v_k 的新路径 $p' = \langle v_1, \ldots, v_i \rangle \rightsquigarrow p'_{ij} \rightsquigarrow \langle v_j, \ldots, v_k \rangle$。这条新路径 p' 的权重 $w(p') = w(p) - w(p_{ij}) + w(p'_{ij}) < w(p)$。这就与 p 是从 v_1 到 v_k 的最短路径的假设相矛盾。因此，p_{ij} 必须是从 v_i 到 v_j 的最短路径 。

4. 负权边与环路 (Negative-Weight Edges and Cycles) (LEC11, pp.1-2)

- **负权边 (Negative-Weight Edges)**：边可以具有负权重。
    - 如果图中不存在从源点 s 可达的**负权环路 (negative-weight cycle)**，那么从 s 到所有可达顶点的最短路径权重仍然是明确定义的 。
    - 如果图中存在一个从 s 可达的负权环路（例如，环路 c 的权重 $w(c) < 0$），那么对于该环路上的任何顶点 v（以及从 s 经此环路可达的其他顶点），最短路径权重 $\delta(s, v)$ 将是 $-\infty$ 。这是因为我们可以无限次地遍历这个负权环路，每次都会使路径总权重减少。在这种情况下，最短路径通常被认为是不存在的（或无定义的）。

    - **例子** (LEC11, p.2)：图中 (e,f,e) 构成一个权重为 $3 + (-6) = -3$ 的负权环路。如果这个环路从源点 s 可达，则 $\delta(s, e) = -\infty, \delta(s, f) = -\infty, \delta(s, g) = -\infty$ 。
- **环路与最短路径** (LEC11, p.2)：
    - **最短路径能否包含环路?**
        - **负权环路**：不能。如果包含，可以通过多次遍历使路径权重任意小，导致最短路径无定义（$-\infty$） 。
        - **正权环路**：不能。如果最短路径包含正权环路，移除该环路会得到一条权重更小的路径，与最短路径的定义矛盾 。
        - **零权环路**：可以移除而不改变路径权重。因此，我们可以假设我们寻找的最短路径是**简单路径 (simple paths)**，即不包含重复顶点的路径（自然也就不包含环路） 。
    - 所以，我们通常寻找的是**不包含环路的最短路径**。

5. 最短路径的表示与初始化 (LEC11, p.2)

- 对于每个顶点 $v \in V$，我们维护两个属性：
    - `d[v]`：从源点 s 到顶点 v 的**最短路径权重的估计值 (shortest-path estimate)**。它总是 $\delta(s, v)$ 的一个上界，即 $d[v] \geq \delta(s, v)$。随着算法的进行，$d[v]$ 的值会逐渐减小并收敛到 $\delta(s, v)$。
    - `π[v]`：顶点 v 在从 s 到 v 的当前计算出的最短路径上的**前驱顶点 (predecessor)**。
- **前驱子图 (Predecessor Subgraph)**：由所有顶点的 π 值构成的边集 $E_\pi = \{(\pi[v], v) \mid v \in V \text{ and } \pi[v] \neq \text{NIL}\}$ 会形成一棵以 s 为根的**最短路径树 (shortest-path tree)**，前提是图中没有从 s 可达的负权环路。
- **初始化算法 `INITIALIZE-SINGLE-SOURCE(V, s)`** (LEC11, p.2)：

```
INITIALIZE-SINGLE-SOURCE(G, s) {
    for each vertex v in V {
        d[v] = ∞
        π[v] = NIL
    }
    d[s] = 0 // 源点到自身的距离为0
}
```

所有单源最短路径算法都以此初始化开始 。

6. 松弛操作 (Relaxation) (LEC11, p.2)

松弛是大多数最短路径算法的核心操作。对边 (u, v) 进行松弛操作是指：检查是否可以通过顶点 u 来改进当前已知的到达顶点 v 的最短路径。

- **`RELAX(u, v, w)` 过程** (LEC11, p.3)：

```
RELAX(u, v, w) { // w 是边 (u,v) 的权重函数
    if d[v] > d[u] + w(u,v) then { // 如果通过 u 到达 v 更近
        d[v] = d[u] + w(u,v)   // 更新 v 的最短路径估计
        π[v] = u                 // 将 u 设置为 v 的前驱
    }
}
```

- 松弛操作可能会减小 $d[v]$ 的值并更新 $\pi[v]$。

- 不同的最短路径算法的区别在于它们以何种顺序、以及对每条边执行多少次松弛操作 。

7. 最短路径的性质 (LEC11, p.4)

这些性质是理解和证明最短路径算法正确性的基础：

- **三角不等式 (Triangle inequality)**：对于任何边 $(u, v) \in E$，总有 $\delta(s, v) \leq \delta(s, u) + w(u, v)$ 。
- **上界性质 (Upper-bound property)**：对于所有顶点 v，我们总是有 $d[v] \geq \delta(s, v)$。一旦 $d[v]$ 达到了 $\delta(s, v)$，它将不再改变（因为松弛操作只会降低 $d[v]$ 的值，而它不能低于真正的最短路径权重） 。
- **无路径性质 (No-path property)**：如果从 s 到 v 没有路径，则算法结束后（或在过程中）总是有 $d[v] = \infty$（且 $\delta(s, v) = \infty$） 。
- **收敛性质 (Convergence property)**：如果 $s \rightsquigarrow u \rightarrow v$ 是一条最短路径（即 (u, v) 是从 s 到 v 的某条最短路径上的最后一条边），并且在对边 (u, v) 进行松弛操作之前的任何时刻有 $d[u] = \delta(s, u)$，那么在松弛操作之后以及此后的任何时刻，都有 $d[v] = \delta(s, v)$ 。
- **路径松弛性质 (Path-relaxation property)**：如果 $p = \langle v_0, v_1, \ldots, v_k \rangle$ 是从源点 $s = v_0$ 到 v_k 的一条最短路径，并且我们按照路径上边的顺序 $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ 对这些边进行松弛（即使中间穿插了对其他边的松弛操作），那么在这些松弛操作完成之后，必有 $d[v_k] = \delta(s, v_k)$。这个性质对某些算法（如 Bellman-Ford）的正确性至关重要 。

8. 备考要点 (针对此概览部分)

- **简答题考点**：
  - 定义单源最短路径问题。
  - 解释最短路径的“最优子结构”性质。
  - 负权边对最短路径问题有什么影响？什么是负权环路？它如何影响最短路径的定义？
  - 解释松弛操作 `RELAX(u,v,w)` 的过程和目的。
  - 列举并解释最短路径的几个重要性质（如三角不等式、上界性质等）。
- **算法分析题考点**：
  - 对于给定的图（可能包含负权边或负权环路），判断某些顶点间的最短路径是否存在或其权重。
- **设计证明题考点**：
  - 证明最短路径的最优子结构性质。
  - 证明三角不等式。

图算法详解之 8.5 (B)：Bellman-Ford 算法 (LEC11)

Bellman-Ford 算法是解决单源最短路径问题的一种算法，它的主要特点是能够处理**带有负权重边**的图。同时，它还能检测出图中是否存在从源点可达的负权环路。

1. 算法特点与适用性 (LEC11, p.3)

- **解决问题**：单源最短路径问题。
- **输入**：一个有向图 $G = (V, E)$，权重函数 w，源顶点 s。
- **输出**：

- 如果不存在从源点 s 可达的负权环路，则算法返回 `TRUE`，并计算出从 s 到所有其他顶点的最短路径权重 `d[v]` 和前驱 `π[v]`。
- 如果存在从源点 s 可达的负权环路，则算法返回 `FALSE`，表明无法计算出明确的最短路径（因为路径权重可以趋向 $-\infty$）。

- **允许负权边**：这是 Bellman-Ford 算法与 Dijkstra 算法的一个主要区别。

2. 算法思想 (LEC11, p.3)

- Bellman-Ford 算法的核心思想是**对图中的所有边进行多次松弛 (relaxation) 操作**。
- 它迭代地减少从源点 s 到所有其他顶点的路径权重的估计值 `d[v]`，直到找到实际的最短路径。
- **主要步骤**：
  i. 初始化所有顶点的 `d` 值为 ∞（源点 s 的 `d[s]` 为 0），`π` 值为 `NIL`。
  ii. 对图中的**每一条边**执行松弛操作。这个过程重复进行 $|V|-1$ 次。
    - 之所以是 $|V|-1$ 次，是因为一条不包含环路的最短路径最多包含 $|V|-1$ 条边。根据路径松弛性质 (Path-relaxation property)，如果在一条最短路径 $p=\langle v_0, v_1, \ldots, v_k\rangle$ 上按顺序松弛其边，即使中间穿插其他松弛，最终 $d[v_k]$ 也会收敛到 $\delta(s, v_k)$。经过 $|V|-1$ 轮对所有边的松弛，可以保证所有不长于 $|V|-1$ 条边的最短路径都被找到。
  iii. 完成上述 $|V|-1$ 轮松弛后，再对所有边进行一次检查。如果在这次检查中，某条边 (u, v) 仍然可以被松弛（即 $d[v] > d[u] + w(u, v)$），则说明图中存在从源点可达的负权环路。

3. Bellman-Ford 算法伪代码 `BELLMAN-FORD(V, E, w, s)` (LEC11, p.3)

```
BELLMAN-FORD(G, w, s) { // G=(V,E), w是权重函数, s是源顶点
    // 1. 初始化 (LEC11, p.2)
    INITIALIZE-SINGLE-SOURCE(G, s) // d[s]=0, 其他d[v]=∞, 所有π[v]=NIL

    // 2. 对所有边进行 |V|-1 轮松弛 (LEC11, p.3)
    for i = 1 to |V| - 1 {
        for each edge (u,v) in E { // 对于图中的每一条边
            RELAX(u, v, w) // 调用松弛操作 (LEC11, p.3)
        }
    }

    // 3. 检查是否存在负权环路 (LEC11, p.3)
    for each edge (u,v) in E {
        if d[v] > d[u] + w(u,v) then {
            return FALSE // 存在负权环路
        }
    }

    return TRUE // 不存在负权环路，最短路径已找到
}
```

4. 例子 (LEC11, p.3)

PPT (LEC11, p.3) 中给出了一个包含5个顶点 (s, t, x, y, z) 和10条边的图的例子，并展示了 Bellman-Ford 算法的执行过程。

- 边集 E: (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)。
- **初始化**： `d[s]=0` ， `d[t]=d[x]=d[y]=d[z]=∞` 。
- **Pass 1 (第一轮对所有边松弛)**:
  - `RELAX(s,t,w(s,t))` $\Rightarrow$ `d[t]` 更新，`π[t]=s` 。
  - `RELAX(s,y,w(s,y))` $\Rightarrow$ `d[y]` 更新，`π[y]=s` 。
  - ...（按边的顺序逐个松弛）
  - PPT中展示了每轮松弛后各顶点 `d` 值的变化（图中用顶点内的数字表示 `d` 值，箭头指向前驱 `π` ）。
- **Pass 2, Pass 3, Pass 4 ($|V|-1=5-1=4$ 轮)**:
  - 继续对所有边进行松弛， `d` 值和 `π` 值会逐步更新，收敛到最短路径。
  - 例如，在Pass 2之后， `d[x]` 可能通过路径 $s \to y \to x$ 得到更新。
- **检查负权环路** (LEC11, p.3, "回路检测"):
  - 在4轮松弛后，如果再对所有边检查一次，发现仍有边可以松弛，则说明存在负权环路。PPT中的例子，如果边 (s,b) 满足 $d[b] > d[s] + w(s,b)$，则说明存在问题。 (这里的 (s,b) 是一个泛指的例子，并非特指之前图中的边)。

5. 效率分析 (LEC11, p.3)

- **初始化** `INITIALIZE-SINGLE-SOURCE`： $\Theta(V)$ 时间。
- **主循环（$|V|-1$ 轮松弛）**：
  - 外层循环执行 $|V|-1$ 次 ($O(V)$ 次)。
  - 内层循环遍历所有 $|E|$ 条边 ($O(E)$ 次)。
  - `RELAX` 操作是 $\Theta(1)$ 时间。
  - 因此，这部分的总体时间复杂度是 $O(V) \times O(E) = O(VE)$。
- **检查负权环路**：遍历所有 $|E|$ 条边，每次检查是 $\Theta(1)$。总共 $O(E)$ 时间。
- **总时间复杂度**：$O(VE)$。

6. 正确性概要

- **无负权环路的情况**:
  - 经过 i 轮对所有边的松弛后，对于任何从源点 s 出发，最多包含 i 条边的路径到达的顶点 v，其 $d[v]$ 的值必定小于或等于该路径的实际权重。
  - 由于简单最短路径最多包含 $|V|-1$ 条边，所以在 $|V|-1$ 轮松弛之后，对于所有可达顶点 v， $d[v]$ 已经收敛到 $\delta(s,v)$（根据路径松弛性质）。
- **存在负权环路的情况**:
  - 如果在 $|V|-1$ 轮松弛之后，仍然可以对某条边 (u,v) 进行松弛（即 $d[v] > d[u] + w(u,v)$），这表明存在一条从 s 到 u 的路径，然后通过边 (u,v) 到达 v，这条路径比当前记录的 $d[v]$ 更短。
  - 如果 $d[u]$ 和 $d[v]$ 在 $|V|-1$ 轮后已经是最短路径权重（假设没有负权环），那么 $d[v] \le d[u] + w(u,v)$（三角不等式）应该成立。如果还能松弛，说明 $d[u]$ 或 $d[v]$ （或两者）还可以通过负权环路无限减小，因此 $d[v]$ 不会是真正的最短路径权重，而是 $-\infty$。

- 因此，第3步的检查可以有效地检测出从源点可达的负权环路。

7. 备考要点

- **简答题考点**:
  - Bellman-Ford 算法主要用于解决什么类型的最短路径问题？（单源，允许负权边）
  - Bellman-Ford 算法如何检测负权环路?
  - 为什么 Bellman-Ford 算法的主循环需要执行 $|V|-1$ 次?
- **算法分析题考点**:
  - 给定一个带权有向图（可能包含负权边）和源顶点，手动模拟 Bellman-Ford 算法的执行过程，展示每一轮松弛后各顶点 `d` 值和 `π` 值的变化。
  - 判断给定的图是否存在从源点可达的负权环路。
  - 分析 Bellman-Ford 算法的时间复杂度。
- **设计证明题考点**:
  - 描述 Bellman-Ford 算法的伪代码。
  - 证明 Bellman-Ford 算法的正确性（在无负权环路时能找到最短路径，在有负权环路时能检测出来）。这通常涉及到对路径松弛性质的应用和对第 k 轮迭代后 $d[v]$ 意义的理解。

图算法详解之 8.5 (C)：有向无环图 (DAG) 中的单源最短路径 (LEC11)

对于有向无环图 (DAG)，我们可以利用其拓扑结构来更高效地解决单源最短路径问题，即使图中存在负权重边。由于DAG中不存在环路，自然也就不存在负权环路，因此最短路径总是明确定义的。

1. 算法思想 (LEC11, p.4)

1. **拓扑排序 (Topological Sort)**：首先对DAG中的所有顶点进行拓扑排序。拓扑排序会给出一个顶点的线性序列，使得对于图中任意一条有向边 (u,v)，顶点 u 在序列中都出现在顶点 v 之前。
2. **按拓扑序松弛边 (Relax Edges in Topological Order)**：按照拓扑排序得到的顶点顺序，依次处理每个顶点。对于当前处理的顶点 u，对其所有出边 (u,v) 执行松弛操作 `RELAX(u,v,w)`。

- **为什么这个顺序有效?**
  - 当我们按照拓扑序处理顶点 u 并松弛其出边 (u,v) 时，根据拓扑排序的性质，所有可能到达 u 的路径（即 u 的所有前驱顶点）都已经在 u 之前被处理过了。
  - 这意味着，在处理顶点 u 时，其最短路径估计 $d[u]$ 已经达到了其最终的最短路径权重 $\delta(s,u)$（假设源点 s 也在 u 之前或就是 u）。
  - 因此，当松弛边 (u,v) 时，我们使用的是 $d[u]=\delta(s,u)$ 的值，这保证了如果 $s \rightsquigarrow u \rightarrow v$ 是一条最短路径，那么 $d[v]$ 将被正确更新为 $\delta(s,v)$。
  - 每一条边只会被松弛一次。

2. DAG 最短路径算法伪代码 `DAG-SHORTEST-PATHS(G, w, s)` (LEC11, p.5)

```
DAG-SHORTEST-PATHS(G, w, s) { // G=(V,E) 是DAG, w是权重函数, s是源顶点
    // 1. 对顶点进行拓扑排序
    Topologically sort the vertices of G

    // 2. 初始化 (LEC11, p.2)
    INITIALIZE-SINGLE-SOURCE(G, s) // d[s]=0, 其他d[v]=∞, 所有π[v]=NIL

    // 3. 按照拓扑排序的顺序处理每个顶点
    for each vertex u, taken in topologically sorted order {
        // 4. 对从 u 出发的所有边进行松弛
        for each vertex v in Adj[u] {
            RELAX(u, v, w)
        }
    }
}
```

3. 例子 (LEC11, p.5)

PPT (LEC11, p.5) 中给出了一个包含6个顶点 (r, s, t, x, y, z) 的DAG，并展示了算法的执行过程。

- **拓扑排序结果**：例如，一个可能的拓扑序是 $\langle r, s, t, x, y, z \rangle$ (假设源点是 s 或在 s 之前的 r)。
- **初始化**：`d[s]=0` (如果s是源点)，其他顶点的 `d` 值为 ∞。
- **按拓扑序松弛**：
  - 处理 r：松弛 r 的出边（例如 $(r, s), (r, t)$）。
  - 处理 s（假设此时 $d[s]$ 已确定为0）：松弛 s 的出边（例如 $(s, t), (s, x)$）。`d[t]` 和 `d[x]` 会被更新。
  - 处理 t：松弛 t 的出边（例如 $(t, x), (t, y), (t, z)$）。此时使用的是已经更新过的 $d[t]$。`d[x]`，`d[y]`，`d[z]` 可能会根据 $d[t]$ 进一步更新。
  - ...以此类推，直到所有顶点处理完毕。
- 由于是按照拓扑序处理，当我们松弛边 (u, v) 时，可以保证 $d[u]$ 已经达到了 $\delta(s, u)$。

4. 效率分析 (LEC11, p.5)

- **拓扑排序**：使用基于DFS的拓扑排序算法，时间复杂度为 $\Theta(V + E)$。
- **初始化** `INITIALIZE-SINGLE-SOURCE`：$\Theta(V)$ 时间。
- **主循环（遍历所有顶点并松弛其出边）**：
  - 外层循环对每个顶点执行一次，共 $|V|$ 次。
  - 内层循环对每个顶点的所有出边执行一次松弛操作。所有顶点的出边总和是 $|E|$。
  - `RELAX` 操作是 $\Theta(1)$ 时间。
  - 因此，这部分的总时间复杂度是 $\Theta(V + E)$（因为每个顶点和每条边都只被处理常数次）。
- **总时间复杂度**：$\Theta(V + E) + \Theta(V) + \Theta(V + E) = \Theta(V + E)$。
- 这比通用的 Bellman-Ford 算法 ($O(VE)$) 要快得多，也比在一般图上使用基于二叉堆的 Dijkstra 算法 ($O((V + E) \log V)$ 或 $O(E \log V)$) 要快（尤其是在稠密图上，或者当不需要优先队列的开销时）。

5. 备考要点

- **简答题考点**:
  - 为什么可以在DAG上高效地求解单源最短路径问题，即使存在负权边？（因为DAG无环，所以无负权环路）
  - 描述在DAG上求解单源最短路径的算法思想。（拓扑排序 + 按序松弛）
  - 为什么按拓扑序松弛边是有效的?
- **算法分析题考点**:
  - 给定一个DAG、权重和源顶点，手动模拟DAG最短路径算法的执行过程，包括拓扑排序的结果和每一步松弛后的 `d` 和 `π` 值。
  - 分析DAG最短路径算法的时间复杂度。
- **设计证明题考点**:
  - 描述DAG最短路径算法的伪代码。
  - 证明该算法的正确性（关键在于证明当松弛边 (u, v) 时，$d[u] = \delta(s, u)$ 已经成立）。

图算法详解之 8.5 (D)：Dijkstra 算法 (LEC11)

Dijkstra 算法是解决单源最短路径问题的另一种著名贪心算法。与 Bellman-Ford 算法不同，Dijkstra 算法要求图中**所有边的权重必须是非负的** ($w(u, v) \geq 0$ for all $(u, v) \in E$)。如果存在负权边，Dijkstra 算法可能无法给出正确的结果。

1. 算法思想 (LEC11, p.5)

- **核心策略**：Dijkstra 算法维护一个顶点集合 S，其中包含所有其最终最短路径权重已经确定的顶点。算法重复地从 $V - S$（即尚未确定最终最短路径的顶点集合）中选择一个最短路径估计 $d[u]$ 最小的顶点 u，将 u 加入到 S 中，然后对所有从 u 出发的边 (u, v) 进行松弛操作。
- **数据结构**:
  - 集合 S：初始为空。
  - 最小优先队列 Q：存储 $V - S$ 中的顶点，以它们的 d 值（最短路径估计）作为键 。
- **贪心选择**：每次从 Q 中选择（即 `EXTRACT-MIN` ）具有最小 d 值的顶点。这个选择是贪心的，因为它选择了当前看来离源点最近的未确定顶点。

2. Dijkstra 算法伪代码 `Dijkstra(G, w, s)` (LEC11, p.5)

```
Dijkstra(G, w, s) { // G=(V,E), w是权重函数 (非负), s是源顶点
    // 1. 初始化 (LEC11, p.2)
    INITIALIZE-SINGLE-SOURCE(G, s) // d[s]=0, 其他d[v]=∞, 所有π[v]=NIL

    // 2. 初始化集合 S 和优先队列 Q (LEC11, p.5)
    S = {} // S 是已确定最短路径的顶点集合，初始为空
    Q = V  // 优先队列 Q 包含所有顶点，以 d[v] 作为键值

    // 3. 主循环 (LEC11, p.5)
    while Q is not empty {
        u = EXTRACT-MIN(Q) // 从 Q 中提取 d[u] 最小的顶点 u
        S = S U {u}        // 将 u 加入集合 S

        // 4. 对从 u 出发的所有边 (u,v) 进行松弛 (LEC11, p.5)
        for each vertex v in Adj[u] {
            // 如果通过 u 到达 v 可以缩短路径，则更新 d[v] 和 π[v]
            // 并且如果 v 仍在 Q 中，需要更新其在队列中的键值
            RELAX(u, v, w)
            // (RELAX内部会判断 if d[v] > d[u] + w(u,v) then d[v]=..., π[v]=...)
            // 如果 d[v] 被更新，且 v 仍在 Q 中，优先队列需要执行 DECREASE-KEY(Q, v, d[v])
        }
    }
}
```

- **注意 `RELAX` 和 `DECREASE-KEY`**：在 `RELAX(u,v,w)` 中，如果 `d[v]` 被更新了，并且顶点 `v` 仍然在优先队列 `Q` 中，那么需要调用一个 `DECREASE-KEY(Q, v, d[v])` 操作来调整 `v` 在队列中的位置以反映其新的、更小的 `d[v]` 值。

3. 例子 (LEC11, pp.5-6)

PPT (LEC11, pp.5-6) 中给出了一个包含5个顶点 (s, t, x, y, z) 和非负权边的图，并逐步展示了 Dijkstra 算法的执行过程。

- **初始**：`d[s]=0`，`d[t,x,y,z]=∞`。`S={}`。$Q = \{s:0, t:\infty, x:\infty, y:\infty, z:\infty\}$.
- **迭代1**：`EXTRACT-MIN(Q) -> s`。`S={s}`。
  - `RELAX(s,t)`：`d[t] = min(∞, 0+10) = 10`，`π[t]=s`。
  - `RELAX(s,y)`：`d[y] = min(∞, 0+5) = 5`，`π[y]=s`。
  - $Q = \{y:5, t:10, x:\infty, z:\infty\}$.
- **迭代2**：`EXTRACT-MIN(Q) -> y`。`S={s,y}`。
  - `RELAX(y,t)`：`d[t] = min(10, d[y]+w(y,t)=5+3=8)`，`π[t]=y`。
  - `RELAX(y,x)`：`d[x] = min(∞, d[y]+w(y,x)=5+9=14)`，`π[x]=y`。
  - `RELAX(y,z)`：`d[z] = min(∞, d[y]+w(y,z)=5+2=7)`，`π[z]=y`。
  - $Q = \{z:7, t:8, x:14\}$.
- **迭代3**：`EXTRACT-MIN(Q) -> z`。`S={s,y,z}`。

- `RELAX(z,s)`：(假设有边(z,s)) $d[s]$ 不会改变因为 $d[s]=0$ 且是最小值。
- `RELAX(z,x)`：`d[x] = min(14, d[z]+w(z,x)=7+6=13)`，`π[x]=z`。
- $Q=\{t:8,x:13\}$.

- **迭代4**：`EXTRACT-MIN(Q) -> t`。`S={s,y,z,t}`。
  - `RELAX(t,x)`：`d[x] = min(13, d[t]+w(t,x)=8+1=9)`，`π[x]=t`。
  - `RELAX(t,y)`：$y\notin Q$.
  - $Q=\{x:9\}$.
- **迭代5**：`EXTRACT-MIN(Q) -> x`。`S={s,y,z,t,x}`。
  - `RELAX(x,z)`：$z\notin Q$.
  - $Q=\{\}$. 循环结束。

4. 效率分析

Dijkstra 算法的效率依赖于最小优先队列 Q 的实现。设 $|V|=n,|E|=m$。

- **初始化** `INITIALIZE-SINGLE-SOURCE`：$\Theta(V)$。
- **构建初始优先队列** Q：
  - 如果用**数组**实现（每次查找最小需要 $O(V)$）：总的查找最小操作 $O(V^2)$。更新 d 值 $O(1)$。总时间 $O(V^2+E)=O(V^2)$ (因为对每条边松弛一次)。
  - 如果用**二叉堆 (Binary Heap)** 实现：
    - 建堆 (将所有顶点加入)：$O(V)$。
    - 主循环执行 $|V|$ 次 `EXTRACT-MIN` 操作，每次 $O(\log V)$。总共 $O(V\log V)$。
    - 最多有 $|E|$ 次 `DECREASE-KEY` 操作（每次松弛成功时），每次 $O(\log V)$。总共 $O(E\log V)$。
    - 总时间复杂度：$O(V+V\log V+E\log V)=O((V+E)\log V)$ 或通常简写为 $O(E\log V)$（假设图连通，$E\geq V-1$）。
  - 如果用**斐波那契堆 (Fibonacci Heap)** 实现：
    - 建堆 $O(V)$。
    - $|V|$ 次 `EXTRACT-MIN` 操作，摊销时间 $O(\log V)$。总共 $O(V\log V)$。
    - $|E|$ 次 `DECREASE-KEY` 操作，摊销时间 $O(1)$。总共 $O(E)$。
    - 总时间复杂度：$O(V\log V+E)$。这对于稠密图 ($E\approx V^2$) 优于二叉堆实现。

5. 正确性概要（贪心选择的正确性）

Dijkstra 算法的正确性依赖于其贪心选择的正确性：当它将一个顶点 u 从 Q 中取出并加入到 S 中时，此时的 $d[u]$ 必定等于 $\delta(s,u)$ (即 u 的最短路径权重已经最终确定)。

- **证明思路 (反证法)**：
  i. 假设 u 是第一个被加入 S 但 $d[u]\neq\delta(s,u)$ 的顶点。由于 $d[u]\geq\delta(s,u)$ (上界性质) 且 $d[u]\neq\delta(s,u)$，所以必有 $d[u]>\delta(s,u)$。
  ii. 由于 $s\in S$ 且 $d[s]=\delta(s,s)=0$，所以 $u\neq s$。
  iii. 因为存在从 s 到 u 的路径（否则 $\delta(s,u)=\infty$，而 $d[u]$ 有限），所以存在一条从 s 到 u 的最短路径 p。设 $p=s\rightsquigarrow y\to x\rightsquigarrow u$，其中 (y,x) 是 p 上第一条使得 $y\in S$ 而 $x\in V-S$ (即 $x\in Q$) 的边。
  iv. 当 y 被加入 S 时，我们有 $d[y]=\delta(s,y)$ (因为 u 是第一个不满足此条件的)。此时，边 (y,x) 被松弛，所以 $d[x]\leq d[y]+w(y,x)=\delta(s,y)+w(y,x)$。

v. 由于 x 在最短路径 p 上位于 u 之前（或者 $x = u$），且 p 的子路径 $s \rightsquigarrow x$ 也是最短路径，所以 $\delta(s, x) = \delta(s, y) + w(y, x)$。
vi. 因此，$d[x] \leq \delta(s, x)$。结合上界性质 $d[x] \geq \delta(s, x)$，我们有 $d[x] = \delta(s, x)$。
vii. 由于 x 在 s 到 u 的最短路径上（或者 $x = u$），且所有边权重非负，所以 $\delta(s, x) \leq \delta(s, u)$。
viii. 因此，$d[x] = \delta(s, x) \leq \delta(s, u) < d[u]$ (由步骤1的假设)。
ix. 但是，如果 $d[x] < d[u]$，那么 Dijkstra 算法在选择 u 之前应该先选择 x 从 Q 中取出（因为 x 也在 Q 中，且 $d[x]$ 更小），这与 u 是第一个被错误加入 S 的顶点（或者说，是当前 Q 中 d 值最小而被选出的）矛盾。
x. 因此，初始假设 $d[u] > \delta(s, u)$ 不成立。所以 $d[u] = \delta(s, u)$。

- **关键点**：非负权边保证了 $\delta(s, x) \leq \delta(s, u)$，从而确保了 $d[x] < d[u]$，导出矛盾。如果存在负权边，这个不等式关系可能不成立，导致算法出错。

6. 备考要点

- **简答题考点**：
  - Dijkstra 算法解决什么问题？其对边的权重有什么要求？（单源最短路径，非负权边）
  - 描述 Dijkstra 算法的基本思想和步骤（维护集合S，使用优先队列Q，贪心选择）。
  - Dijkstra 算法的贪心选择是什么?
  - 为什么 Dijkstra 算法不适用于有负权边的图？（举例说明或解释其正确性证明如何失效）
- **算法分析题考点**：
  - 给定一个带非负权重的有向图（或无向图）和源顶点，手动模拟 Dijkstra 算法的执行过程，展示每一步集合 S、优先队列 Q 的状态、各顶点 `d` 值和 `π` 值的变化，以及最终的最短路径树。
  - 分析 Dijkstra 算法的时间复杂度，并说明其如何依赖于优先队列的实现（数组、二叉堆、斐波那契堆）。
- **设计证明题考点**：
  - 描述 Dijkstra 算法的伪代码。
  - 证明 Dijkstra 算法的正确性（核心是证明当一个顶点 u 被从 Q 中取出加入 S 时，$d[u] = \delta(s, u)$）。

图算法详解之 8.5 (E)：所有点对最短路径 (All-Pairs Shortest Paths - APSP) (LEC12)

所有点对最短路径问题 (APSP) 的目标是计算一个图中每对顶点 (u, v) 之间的最短路径权重。

1. 问题描述与输入输出 (LEC12, p.1)

- **输入**：
  - 一个有向图 $G = (V, E)$。
  - 一个权重函数 $w : E \to \mathbb{R}$。
- **输出**：
  - 一个 $n \times n$ 的矩阵 $D = (d_{ij})$，其中 $d_{ij} = \delta(i, j)$ 是从顶点 i 到顶点 j 的最短路径的权重。
  - 通常还会输出一个前驱矩阵 $\Pi = (\pi_{ij})$，其中 π_{ij} 是从 i 到 j 的某条最短路径上 j 的前驱顶点。
- **图的表示**: 通常假设图以邻接矩阵的形式给出权重 $W = (w_{ij})$，其中:
  - $w_{ij} = 0$ 如果 $i = j$。

- $w_{ij} = \text{weight of edge } (i,j)$ 如果 $i \neq j$ 且 $(i,j) \in E$。
- $w_{ij} = \infty$ 如果 $i \neq j$ 且 $(i,j) \notin E$。

2. 使用单源最短路径算法解决 APSP (LEC12, p.1)

一种直接的方法是多次运行单源最短路径 (SSSP) 算法：

- **使用 Bellman-Ford 算法**:
  - 从每个顶点 $v \in V$ 作为源点运行一次 Bellman-Ford 算法。
  - Bellman-Ford 算法的时间复杂度是 $O(VE)$。
  - 总时间复杂度为 $O(V \cdot VE) = O(V^2E)$。
  - 如果图是稠密的 ($E = \Theta(V^2)$)，则复杂度为 $O(V^4)$。
  - 这种方法可以处理负权边和检测负权环路。
- **使用 Dijkstra 算法 (如果无负权边)**:
  - 如果图中所有边的权重都是非负的，可以从每个顶点作为源点运行一次 Dijkstra 算法。
  - 使用二叉堆实现的 Dijkstra 算法复杂度为 $O((V+E)\log V)$ 或 $O(E \log V)$。
  - 总时间复杂度为 $O(V(E+V)\log V)$ 或 $O(VE \log V)$。
  - 如果图是稠密的，复杂度可能为 $O(V^3 \log V)$。

接下来我们将讨论专门为 APSP 设计的、通常时间复杂度为 $\Theta(V^3)$ 的算法。

3. 基于矩阵乘法的 APSP 算法 (LEC12, pp.1-3)

这种方法利用了最短路径的最优子结构性质，通过逐步扩展路径中允许的边数来找到所有点对的最短路径。

- **最短路径的结构** (LEC12, p.1):
  - 最短路径的所有子路径也都是最短路径。
  - 设 $l_{ij}^{(m)}$ 是从顶点 i 到顶点 j 的路径中，包含**至多 m 条边**的最短路径的权重。
- **递归解** (LEC12, p.1):
  - $m = 0$:
    - $l_{ij}^{(0)} = 0$ 如果 $i = j$。
    - $l_{ij}^{(0)} = \infty$ 如果 $i \neq j$。
  - $m \geq 1$:
    - $l_{ij}^{(m)} = \min_{1 \leq k \leq n} \{l_{ik}^{(m-1)} + w_{kj}\}$。
    - 这个公式的含义是：从 i 到 j 包含至多 m 条边的最短路径，要么是已经找到的从 i 到 j 包含至多 $m-1$ 条边的最短路径（如果这条路径不需要第 m 条边，即 $l_{ij}^{(m-1)}$），要么是通过某条路径到达某个中间顶点 k (该路径至多 $m-1$ 条边，权重 $l_{ik}^{(m-1)}$)，然后再从 k 通过一条边 (k,j) (权重 w_{kj}) 到达 j。
    - 实际上，PPT (LEC12, p.1, slide 6) 中给出的递归是 $l_{ij}^{(m)} = \min(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\{l_{ik}^{(m-1)} + w_{kj}\})$。这里 $\min_{1 \leq k \leq n}\{l_{ik}^{(m-1)} + w_{kj}\}$ 已经考虑了所有可能的倒数第二跳是 k 的情况。如果 $l_{ij}^{(m-1)}$ 本身就是这个最小值（即路径不多于 $m-1$ 条边），那么它自然会被包含。更简洁的写法是

$l_{ij}^{(m)} = \min_{1 \le k \le n}\{l_{ik}^{(m-1)} + w_{kj}\}$，其中允许 $k = j$ 并且 $w_{jj} = 0$，这样就包含了路径长度小于 m 的情况。

- PPT (LEC12, p.1, slide 6) 的公式 $l_{ij}^{(m)} = \min_{1 \le k \le n}\{l_{ik}^{(m-1)} + w_{kj}\}$ 是正确的，它意味着考虑所有可能的最后一条边是 (k, j)，而从 i 到 k 的路径至多有 $m-1$ 条边。

- **计算过程** (LEC12, p.2):
  - $L^{(1)} = W$ (即 $l_{ij}^{(1)} = w_{ij}$，只包含一条边的最短路径权重就是边的直接权重)。
  - 可以迭代计算 $L^{(2)}, L^{(3)}, \ldots, L^{(n-1)}$。
  - 如果图中没有负权环路，那么所有简单最短路径最多包含 $n-1$ 条边。因此，$L^{(n-1)}$ 就包含了所有点对的最短路径权重，即 $\delta(i, j) = l_{ij}^{(n-1)}$。
  - 计算 $L^{(m)}$ 从 $L^{(m-1)}$ 和 W 的过程类似于矩阵乘法 (LEC12, p.2, slide 8):
    如果将矩阵乘法 $C = A \cdot B$ 定义为 $c_{ij} = \sum_k a_{ik} b_{kj}$，那么这里 "扩展最短路径" 的操作可以定义为:
    $l_{ij}^{(m)} = \min_k (l_{ik}^{(m-1)} + w_{kj})$
    这个操作被称为 "min-plus" 矩阵乘法或 "distance product"。

- **慢速算法** `SLOW-ALL-PAIRS-SHORTEST-PATHS(W, n)` (LEC12, p.2):
  i. $L^{(1)} \leftarrow W$。
  ii. `for m = 2 to n-1`:
  iii. `do` $L^{(m)} \leftarrow \text{EXTEND-SHORTEST-PATHS}(L^{(m-1)}, W)$。
  iv. `return` $L^{(n-1)}$。
  其中 `EXTEND-SHORTEST-PATHS(L', W)` 计算 $L''_{ij} = \min_k (L'_{ik} + w_{kj})$，需要 $\Theta(n^3)$ 时间。
  由于循环 $n-2$ 次，总时间复杂度为 $O(n \cdot n^3) = O(n^4)$。

- **改进的快速算法** `FASTER-APSP(W, n)` (LEC12, p.3):
  - 注意到 "min-plus" 乘法也具有类似标准矩阵乘法的结合律。我们可以通过重复平方的方法来加速计算 $L^{(n-1)}$。
  - 我们想计算 W^{n-1} (在 "min-plus" 意义下)。
  - 计算序列: $L^{(1)} = W$, $L^{(2)} = W \otimes W$, $L^{(4)} = L^{(2)} \otimes L^{(2)}$, $L^{(8)} = L^{(4)} \otimes L^{(4)}$, ..., 直到 $L^{(2^k)}$ 其中 $2^k \ge n-1$。
  - 这个过程只需要 $\lceil \log_2(n-1) \rceil$ 次 "min-plus" 矩阵乘法。
  - 每次 "min-plus" 矩阵乘法需要 $\Theta(n^3)$ 时间。
  - 总时间复杂度为 $O(n^3 \log n)$。

```
FASTER-APSP(W, n) {
    L_current = W            // L^(1)
    m = 1
    while m < n - 1 {
        L_next = EXTEND-SHORTEST-PATHS(L_current, L_current) // L^(2m) = L^(m) min-plus L^(m)
        L_current = L_next
        m = 2 * m
    }
    return L_current         // 最终 L_current 是 L^(m) 其中 m >= n-1
}
```

  - PPT (LEC12, p.3, slide 13) 中的伪代码 `FASTER-APSP(W,n)` 是正确的，它通过 `m = 2*m` 来实现重复平方。

4. Floyd-Warshall 算法 (LEC12, pp.3-4)

Floyd-Warshall 算法是另一种解决 APSP 问题的经典动态规划算法，其时间复杂度为 $\Theta(n^3)$，并且允许图中存在负权边（但不能有负权环路）。

- **最短路径的结构 (另一种角度)** (LEC12, p.3):
  - 考虑从顶点 i 到顶点 j 的一条最短路径 p。
  - 路径 p 上的**中间顶点 (intermediate vertex)** 是指路径上除了起点 i 和终点 j 之外的任何顶点 。
  - 设 $d_{ij}^{(k)}$ 是从顶点 i 到顶点 j 的所有路径中，其**中间顶点全部取自集合** $\{1, 2, \ldots, k\}$ 的一条最短路径的权重 。
- **递归解** (LEC12, pp.3-4):
  - $k = 0$: $d_{ij}^{(0)} = w_{ij}$ (没有中间顶点，路径只包含边 (i, j) 或者 $i = j$ 时为0) 。
  - $k \geq 1$: 考虑从 i 到 j，中间顶点只来自 $\{1, \ldots, k\}$ 的最短路径。这条路径有两种可能:
    a. **不经过顶点 k 作为中间顶点**：那么这条路径的最短权重与中间顶点只来自 $\{1, \ldots, k-1\}$ 的最短路径权重相同，即 $d_{ij}^{(k-1)}$ 。
    b. **经过顶点 k 作为中间顶点**：那么这条路径可以分解为从 i 到 k (中间顶点来自 $\{1, \ldots, k-1\}$) 的最短路径，和从 k 到 j (中间顶点也来自 $\{1, \ldots, k-1\}$) 的最短路径的拼接。其权重为 $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ 。
  - 因此，递归式为:
    $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 。
  - 最终的解是 $D^{(n)} = (d_{ij}^{(n)})$，其中 $d_{ij}^{(n)} = \delta(i, j)$ 。
- **Floyd-Warshall 算法伪代码** `FLOYD-WARSHALL(W)` (LEC12, p.4):

```
FLOYD-WARSHALL(W) { // W 是 n x n 的邻接矩阵
    n = rows[W]
    D_prev = W // D^(0)

    for k = 1 to n { // 允许中间顶点从 {1,...,k} 中选择
        let D_curr[1..n, 1..n] be a new matrix
        for i = 1 to n {
            for j = 1 to n {
                D_curr[i,j] = min(D_prev[i,j], D_prev[i,k] + D_prev[k,j])
            }
        }
        D_prev = D_curr // D_prev 现在是 D^(k)
    }
    return D_prev // 最终是 D^(n)
}
```

  - 可以优化空间，只使用一个二维数组 D，因为计算 $D^{(k)}$ 时，$D^{(k-1)}[i, k]$ 和 $D^{(k-1)}[k, j]$ 的值在内层循环中不会被 $D^{(k)}$ 的新值覆盖。

```
FLOYD-WARSHALL-OPTIMIZED(W) {
    n = rows[W]
    D = W

    for k = 1 to n {
        for i = 1 to n {
            for j = 1 to n {
                D[i,j] = min(D[i,j], D[i,k] + D[k,j])
            }
        }
    }
    return D
}
```

- **效率分析** (LEC12, p.4):
  - 算法有三个嵌套的 `for` 循环，每个循环从1到 n。
  - 内部操作是常数时间。
  - 总时间复杂度为 $\Theta(n^3)$ 。
- **构造最短路径**:
  - 可以维护一个前驱矩阵 $\Pi^{(k)}$，其中 $\pi_{ij}^{(k)}$ 是从 i 到 j (中间顶点来自 $\{1, \ldots, k\}$) 的最短路径上 j 的前驱。
  - 如果 $d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$，则 $\pi_{ij}^{(k)} = \pi_{ij}^{(k-1)}$。
  - 否则，$\pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$ (路径经过 k，所以 j 的前驱与从 k 到 j 的路径上 j 的前驱相同)。
  - 初始化 $\pi_{ij}^{(0)}$：如果 $i = j$ 或 $w_{ij} = \infty$，则 $\pi_{ij}^{(0)} = \text{NIL}$；否则 $\pi_{ij}^{(0)} = i$。

5. 检测负权环路

- **基于矩阵乘法的方法**：如果在计算 $L^{(m)}$ 的过程中，某个对角线元素 $l_{ii}^{(m)}$ 变为负数，则表明存在一个从 i 出发回到 i 的负权环路。
- **Floyd-Warshall 算法**：在算法执行完毕后，如果矩阵 $D^{(n)}$ 的对角线上出现负值 $d_{ii}^{(n)} < 0$，则说明从顶点 i 可达一个负权环路（且 i 也在这个环路上或可达这个环路并返回）。

6. 备考要点

- **简答题考点**:
  - 描述解决所有点对最短路径问题的几种思路（多次SSSP vs 专用APSP算法）。
  - 解释基于矩阵乘法（min-plus）的APSP算法中 $l_{ij}^{(m)}$ 的含义及其递归定义。
  - 描述Floyd-Warshall算法中 $d_{ij}^{(k)}$ 的含义及其递归定义。
  - Floyd-Warshall算法的时间复杂度是多少？它是否能处理负权边？是否能检测负权环路?
- **算法分析题考点**:
  - 给定一个带权图的邻接矩阵，手动模拟Floyd-Warshall算法的几轮迭代，计算 $D^{(k)}$ 矩阵。
  - 分析基于矩阵乘法的APSP算法（慢速和快速版本）的时间复杂度。
- **设计证明题考点**:
  - 描述基于矩阵乘法的APSP算法或Floyd-Warshall算法的伪代码。
  - 证明Floyd-Warshall算法的递归关系 $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 的正确性。

- 解释如何使用Floyd-Warshall算法或基于矩阵乘法的方法来检测负权环路。

图算法详解之 8.6 (A)：最大网络流 - 基本概念 (LEC13)

最大网络流问题是在一个有容量限制的网络中，找到从一个指定的源点到一个指定的汇点能够传输的最大流量。这个问题在物流、通信网络、电路分析等领域有广泛应用。

1. 流网络 (Flow Networks) (LEC13, p.1)

- **定义**：一个流网络是一个**有向图** $G = (V, E)$ 。
- **边的容量 (Capacity)**：每条边 $(u, v) \in E$都有一个非负的**容量** $c(u, v) \geq 0$ 。如果图中不存在边 (u, v)，则可以认为 $c(u, v) = 0$。
- **源点 (Source) 和汇点 (Sink)**：网络中有两个特殊的顶点：
  - **源点** s：物质产生的起点 。
  - **汇点** t：物质消耗的终点 。
- **路径假设**：对于网络中的任何其他顶点 v，都存在一条从 s 到 v 再到 t 的路径 。
- **中间顶点流量守恒**：对于除了源点 s 和汇点 t 之外的所有顶点，流入该顶点的总流量必须等于流出该顶点的总流量（即“所入等于所出”）。

2. 流 (Flow) (LEC13, p.1)

- **定义**：流网络中的一个流是一个函数 $f : V \times V \to \mathbb{R}$，它为每对顶点 (u, v)（可以看作是图中的每条潜在边）赋予一个实数值，并满足以下三个性质：
  i. **容量约束 (Capacity Constraint)**：对于所有的顶点 $u, v \in V$，从 u 到 v 的流 $f(u, v)$ 不能超过该边的容量 $c(u, v)$ 。
  $f(u, v) \leq c(u, v)$
  ii. **斜对称性 (Skew Symmetry)**：从顶点 u 到顶点 v 的流，是其反向流的负值 。
  $f(u, v) = -f(v, u)$
    - 这意味着如果实际存在从 u 到 v 的正向流，那么 $f(v, u)$ 将是一个负值。这种表示主要是为了符号和计算上的方便，例如在剩余网络中。
    - **流的抵消 (Cancellation of flows)** (LEC13, p.1)：实际中，我们不希望在两个顶点间同时存在方向相反的正流量，因为它们会（部分）抵消 。斜对称性是一种符号约定 。
  iii. **流守恒性 (Flow Conservation)**：对于所有不包括源点 s 和汇点 t 的顶点 $u \in V - \{s, t\}$，流入顶点 u 的总流量等于流出顶点 u 的总流量 。
  $\sum_{v \in V} f(v, u) = 0$ (净流入为0)
  或者可以写成：
  $\sum_{v \in V} f(u, v) = 0$ (净流出为0)
  这等价于 $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$ (总流入 = 总流出)。
- **流的值 (Value of the flow)** (LEC13, p.1)：一个流 f 的值 $|f|$ 定义为从源点 s 流出的总净流量（或者流入汇点 t 的总净流量）。

$|f| = \sum_{v \in V} f(s, v) = f(s, V)$
由于流守恒，也可以表示为流入汇点的净流量：
$|f| = \sum_{v \in V} f(v, t) = f(V, t)$ （这里应该是 $f(V, t)$ 指所有流入 t 的流量，或者使用斜对称性 $f(V, t) = -\sum_{v \in V} f(t, v)$ 表示从 t 流出的净流量为负）。
PPT中的表达 $f(V, t)$ 指的是从所有顶点到 t 的净流量和，根据流守恒，这个值也等于从源点 s 出发的净流量。

- **最大流问题 (Maximum Flow Problem)**：目标是找到一个流 f，使得其值 $|f|$ 最大化 。

例子 (LEC13, p.1)：
PPT中展示了一个流网络的例子，其中边上标注了 "flow/capacity"。例如，$v_1 \to v_2$ 的边标注为 $8/12$，表示当前流为8，容量为12。源点 s 流出的总流量为 $f(s, v_1) + f(s, v_2) = 11 + 8 = 19$。

3. 备考要点 (针对此基础概念部分)

- **简答题考点**：
  - 什么是流网络？它包含哪些基本元素（源点、汇点、边的容量）？
  - 定义流的三个性质：容量约束、斜对称性、流守恒性。并解释它们的含义。
  - 如何定义一个流的值?
  - 最大流问题的目标是什么?
- **算法分析题考点**：
  - 给定一个网络和一组边的流量，判断它是否是一个合法的流（即是否满足三个性质）。
  - 计算给定流的值。

图算法详解之 8.6 (B)：Ford-Fulkerson 方法 (LEC13)

Ford-Fulkerson 方法不是一个具体的算法，而是一种解决最大流问题的**通用框架**。它通过不断地在“剩余网络”中寻找“增广路径”来逐步增加流的值，直到找不到增广路径为止，此时就达到了最大流。

1. 核心思想 (LEC13, p.2)

- **迭代改进**：从一个初始流（通常是零流）开始，在每次迭代中，尝试找到一种方法来增加从源点 s 到汇点 t 的总流量。
- **增广路径 (Augmenting Path)**：如果在当前流的基础上，还存在一条从 s 到 t 的路径，可以沿着这条路径发送更多的流量，那么这条路径就称为增广路径。
- **剩余容量与剩余网络**：这两个概念是寻找增广路径的关键。

2. 剩余网络 (Residual Network) (LEC13, p.2)

- **剩余容量 (Residual Capacity)**：对于流网络 $G = (V, E)$ 中的任意一对顶点 u, v，在当前流 f 下，从 u 到 v 的剩余容量 $c_f(u, v)$ 定义为：
  $c_f(u, v) = c(u, v) - f(u, v)$

    - 这表示在不超过边 (u, v) 的原始容量的前提下，还可以从 u 向 v 推送多少额外的流量。
    - 注意，如果 $f(u, v)$ 是正的，那么根据斜对称性 $f(v, u) = -f(u, v)$ 是负的。此时，从 v 到 u 的剩余容量 $c_f(v, u) = c(v, u) - f(v, u) = c(v, u) - (-f(u, v)) = c(v, u) + f(u, v)$。如果原始网络中没有边 (v, u)（即 $c(v, u) = 0$），那么 $c_f(v, u) = f(u, v)$。这代表了可以从 v "推回"给 u 的流量，即减少原来从 u 到 v 的流量。
- **剩余网络 (Residual Network)**：给定流网络 $G = (V, E)$ 和一个流 f，其对应的剩余网络 $G_f = (V, E_f)$ 具有与 G 相同的顶点集 V。边集 E_f 由所有满足 $c_f(u, v) > 0$ 的顶点对 (u, v) 组成。
    - 剩余网络中的边 (u, v) 的权重（或容量）就是剩余容量 $c_f(u, v)$。
    - **观察** (LEC13, p.2)：E_f 中的边要么是原图 E 中的边（称为**前向边**，对应于 $c(u, v) - f(u, v) > 0$），要么是 E 中边的反向（称为**后向边**，对应于 $f(v, u) > 0$，即 $c_f(u, v) = c(u, v) - f(u, v) = 0 - (-f(v, u)) = f(v, u) > 0$）。
    - 因此，剩余网络中的边数 $|E_f|$ 最多是原图边数的两倍，即 $|E_f| \leq 2|E|$。
- **例子 (LEC13, p.2)**：
    - PPT中展示了一个原始网络及其对应的剩余网络。
    - 例如，如果原始网络中边 (s, v_1) 有 $f(s, v_1)/c(s, v_1) = 11/16$，则在剩余网络 G_f 中：
        - 存在前向边 (s, v_1)，其剩余容量 $c_f(s, v_1) = 16 - 11 = 5$。
        - 存在后向边 (v_1, s)，其剩余容量 $c_f(v_1, s) = f(s, v_1) = 11$（假设原图中没有边 (v_1, s)）。

3. 增广路径 (Augmenting Paths) (LEC13, p.2)

- **定义**：剩余网络 G_f 中一条从源点 s 到汇点 t 的**简单路径** p 称为一条增广路径。
- **意义**：如果存在增广路径，意味着我们可以沿着这条路径增加整个网络的流值。
- **增广路径的剩余容量 (Residual capacity of an augmenting path)**：增广路径 p 的剩余容量 $c_f(p)$ 定义为路径 p 上所有边的剩余容量的最小值。
  $c_f(p) = \min\{c_f(u, v) \mid (u, v) \text{ is on path } p\}$
  这个值表示我们可以沿着路径 p 额外推送的最大流量。
- **增流操作 (Augmentation)**：
    - 找到一条增广路径 p 及其剩余容量 $c_f(p)$。
    - 对于路径 p 上的每一条边 (u, v)：
        - 将流 $f(u, v)$ 增加 $c_f(p)$：$f(u, v) \leftarrow f(u, v) + c_f(p)$。
        - 根据斜对称性，将反向流 $f(v, u)$ 减少 $c_f(p)$：$f(v, u) \leftarrow f(v, u) - c_f(p)$ (或者 $f(v, u) \leftarrow -f(u, v)$)。
    - 经过增流操作后，整个网络的流值 $|f|$ 增加了 $c_f(p)$。
    - 新的流仍然满足容量约束、斜对称性和流守恒性 。
- **例子 (LEC13, p.2, "Augmenting path" 图示)**：
    - 原始网络流为19。
    - 在其剩余网络中，找到了一条增广路径 $s \to v_2 \to v_1 \to v_3 \to t$ (路径是 $s - v_2 - v_1 - v_3 - t$)。
    - 路径上的剩余容量分别为 $c_f(s, v_2) = 5$, $c_f(v_2, v_1) = 4$, $c_f(v_1, v_3) = 1$, $c_f(v_3, t) = 5$。
    - 因此，这条增广路径的剩余容量 $c_f(p) = \min(5, 4, 1, 5) = 1$。
    - 沿着这条路径增加流量1，总流量变为 $19 + 1 = 20$。 (PPT中最终流值为23，这里只是一步的例子)

4. Ford-Fulkerson 方法框架 (LEC13, p.2)

```
FORD-FULKERSON-METHOD(G, s, t) {
    1. Initialize flow f to 0 for all edges (u,v) in E.
       (即 f(u,v) = 0 对所有 u,v)
    2. while there exists an augmenting path p from s to t in the residual network G_f {
    3.     c_f_p = residual capacity of p  // c_f(p) = min{c_f(u,v) : (u,v) is on p}
    4.     for each edge (u,v) on p {
    5.         f(u,v) = f(u,v) + c_f_p
    6.         f(v,u) = f(v,u) - c_f_p  // 或者 f(v,u) = -f(u,v)
    7.     }
    8. }
    9. return f
}
```

PPT (LEC13, p.2) 的伪代码更简洁：

```
FORD-FULKERSON-METHOD (G, s, t)
1 initialize flow f to 0
2 while there exists an augmenting path p
3     do augment flow f along p
4 return f
```

- **终止性**：如果所有边的容量都是整数，那么每次增广至少使流值增加1。由于最大流的值以图中所有从 s 出发的边的容量之和为上界，所以算法必然终止。
- **具体实现**：Ford-Fulkerson 方法本身没有规定如何寻找增广路径。不同的寻找策略会导致不同的具体算法（如 Edmonds-Karp 算法）。

5. 例子演示 (LEC13, p.3)

PPT (LEC13, p.3) 通过一系列图示展示了Ford-Fulkerson方法的执行过程：

- **Flow(0)**: 初始零流，剩余网络即为原网络。找到增广路径 $s \to v_1 \to v_3 \to t$，瓶颈容量为12。
- **Flow(1)**: 流值为12。更新剩余网络。找到增广路径 $s \to v_1 \to v_2 \to v_3 \to t$ (这个路径在PPT中可能不同，它选择 $s \to v_2 \to v_4 \to t$ 瓶颈为4)。假设选择 $s \to v_2 \to v_4 \to t$，瓶颈容量为4。
- **Flow(2)**: 流值为 $12 + 4 = 16$。更新剩余网络。找到增广路径 $s \to v_2 \to v_1 \to v_3 \to t$ (或者是 $s \to v_1 \to v_2 \to v_4 \to v_3 \to t$ 等)。
- ... 这个过程一直持续，直到剩余网络中不再存在从 s 到 t 的路径。
- **最终 (LEC13, p.4, "No more augmenting paths max flow attained")**：PPT例子中，流值逐步增加：
  - (a) 初始网络
  - (b) 第一条增广路径 (s, v_1, v_3, t)，流量4。总流量4。
  - (c) 第二条增广路径 (s, v_1, v_2, v_4, t)，流量7。总流量11。
  - (d) 第三条增广路径 (s, v_2, v_4, t)，流量4。总流量19。
  - (e) 第四条增广路径 (s, v_2, v_1, v_3, t)，流量4。总流量23。

- (f) 最终剩余网络中 s 和 t 不再连通，算法终止，最大流为23。
  (注意：PPT中第4页的例子与第3页的例子流程和数值有出入，这里参照第4页的详细步骤。第4页的 (b) $s - v_1 - v_3 - t$ 的瓶颈是 $min(16, 12, 20) = 12$ 才对，但图示写的是4。这里按图示的流量增加值。第4页的(c) $s - v_2 - v_4 - t$ 瓶颈是 $min(13, 12, 4) = 4$，流量从4增加到 $4 + 7 = 11$ (增量7)。这说明图示的路径和瓶颈选择与文字描述的简单路径可能不完全一致，或者PPT中的图示是经过多步增广得到的结果。关键是理解增广的过程。)
- **参照PPT Page 4的图例 (算法设计LEC13.pdf, Page 4)**:
  a. **(a) 初始网络.**
  b. **(b) 增广路径 $s - v_1 - v_3 - t$。** 瓶颈容量 = $\min(c(s, v_1), c(v_1, v_3), c(v_3, t)) = \min(16, 12, 20) = 12$。流量增加12。当前总流量 = 12。
  c. **(c) 增广路径 $s - v_2 - v_4 - t$。** (此时的剩余容量) $c_f(s, v_2) = 13, c_f(v_2, v_4) = 12, c_f(v_4, t) = 4$。瓶颈容量 = $\min(13, 12, 4) = 4$。流量增加4。当前总流量 = $12 + 4 = 16$。
  d. **(d) 增广路径 $s - v_1 - v_2 - v_3 - t$。** (此时的剩余容量) $c_f(s, v_1) = 16 - 12 = 4, c_f(v_1, v_2) = \text{not shown directly, assume } c_f(v_1, v_2) = 9, c_f(v_2, v_3) = 7, c_f(v_3, t) = 20 - 12 = 8$。这条路径可能复杂，PPT中直接给出了流量增加值。
  - **一个更清晰的例子来自 LEC13, p.3 的图演变，最终达到23。**
    - 初始流0。
    - (1) 路径 $s \to v_1 \to v_3 \to t$，瓶颈 $\min(16, 12, 20) = 12$。流 $= 12$。
    - (2) 剩余网络中，路径 $s \to v_2 \to v_4 \to t$，瓶颈 $\min(13, 12, 4) = 4$。流 $= 12 + 4 = 16$。
    - (3) 剩余网络中，路径 $s \to v_2 \to v_3 \to t$ (注意此时 $v_2 \to v_3$ 的边可能在原图没有，是在剩余网络中产生的反向边的效果，或者原图就有 $v_2 \to v_3$ capacity 7)，假设瓶颈为 $\min(13 - 4, 7, 20 - 12) = \min(9, 7, 8) = 7$。流 $= 16 + 7 = 23$。
    - (4) 检查剩余网络，如LEC13 Page 4 (e) 的 Residual Network。此时 s 到 t 不再有增广路径。所以最大流为23。
      (PPT的例子在数值的逐步演进上存在一些不一致或跳跃，但核心思想是找到增广路径并增广)

6. 备考要点

- **简答题考点**:
  - 描述 Ford-Fulkerson 方法的基本思想。
  - 什么是剩余网络？如何根据当前流和原网络容量计算边的剩余容量（包括前向边和后向边）？
  - 什么是增广路径？如何计算增广路径的剩余容量？
  - 增流操作是如何进行的？
  - Ford-Fulkerson 方法的终止条件是什么？
- **算法分析题考点**:
  - 给定一个流网络和当前流，画出其剩余网络。
  - 在剩余网络中找出一条增广路径，并计算其剩余容量。
  - 执行一次增流操作，更新网络中的流。
  - 手动模拟 Ford-Fulkerson 方法的完整执行过程，直到找到最大流。
- **设计证明题考点**:
  - 描述 Ford-Fulkerson 方法的伪代码。
  - 证明一次增流操作后，新的流仍然满足容量约束、斜对称性和流守恒性。
  - 论证当所有容量为整数时，Ford-Fulkerson 方法的终止性。

图算法详解之 8.6 (C)：切割 (Cuts) 与最大流最小割定理 (LEC13)

切割的概念在流网络中非常重要，它帮助我们理解流量的上限，并引出了最大流问题的一个核心定理——最大流最小割定理。

1. 切割 (Cuts) (LEC13, p.3)

- **定义**：在流网络 $G = (V, E)$ 中（源点为 s，汇点为 t），一个 (s, t)**-切割**（简称切割）是将顶点集 V 划分为两个不相交的子集 S 和 T ($T = V - S$)，使得源点 $s \in S$ 且汇点 $t \in T$ 。
- **切割的净流量 (Net flow across a cut)** (LEC13, p.3)：对于一个流 f 和一个切割 (S, T)，从 S 到 T 的**净流量** $f(S, T)$ 定义为所有从 S 中的顶点流向 T 中的顶点的流量之和，减去所有从 T 中的顶点流向 S 中的顶点的流量之和。
  $f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$
  由于斜对称性 $f(v, u) = -f(u, v)$，上式可以简化为：
  $f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v)$ (PPT中的定义关注的是从 S 指向 T 的边的总流量)。
  - **更准确的净流量定义**：$f(S, T) = \sum_{u \in S, v \in T, (u,v) \in E} f(u, v) - \sum_{v \in T, u \in S, (v,u) \in E} f(v, u)$。
  - PPT (LEC13, p.3) 的图示中 "The net flow (f(S,T)) through the cut is the sum of flows f(u,v), where u e S and v e T" 更侧重于从 S 到 T 的总的正向流。
- **切割的容量 (Capacity of a cut)** (LEC13, p.3)：切割 (S, T) 的容量 $c(S, T)$ 定义为所有从 S 中的顶点到 T 中的顶点的**边的容量之和**。
  $c(S, T) = \sum_{u \in S} \sum_{v \in T, (u,v) \in E} c(u, v)$
  注意：这里只计算从 S 指向 T 的边的容量，不考虑从 T 指向 S 的边。
- **最小切割 (Minimum Cut)** (LEC13, p.3)：一个图中所有可能的 (s, t)-切割中，容量最小的那个切割称为最小切割 。其容量称为最小切割容量。

例子 (LEC13, p.3):
PPT中展示了一个切割，其中 $S = \{s, v_1, v_2\}$，$T = \{v_3, v_4, t\}$。

- **切割的容量** $c(S, T)$ 是边 (v_1, v_3)、(v_2, v_3)、(v_2, v_4) 的容量之和 (假设这些是仅有的从S到T的边)。
- **切割的净流量** $f(S, T)$ 是 $f(v_1, v_3) + f(v_2, v_3) + f(v_2, v_4) - f(v_3, v_1) - \dots$ (如果存在反向流)。

2. 切割的性质

- **引理 (LEC13, p.3, Lemma 7.6)**：对于任何流 f 和任何切割 (S, T)，流经切割的净流量 $f(S, T)$ 等于流的值 $|f|$ 。
  $f(S, T) = |f|$
  - **证明思路**：
    $|f| = f(s, V) = \sum_{u \in S} (f(u, V) - f(V, u))$ (因为对于 $u \in S, u \neq s$，其净流出为0)。
    $|f| = \sum_{u \in S} \sum_{v \in V} f(u, v) - \sum_{u \in S} \sum_{v \in V} f(v, u)$
    将 V 拆分为 S 和 T:
    $|f| = \sum_{u \in S} (\sum_{v \in S} f(u, v) + \sum_{v \in T} f(u, v)) - \sum_{u \in S} (\sum_{v \in S} f(v, u) + \sum_{v \in T} f(v, u))$
    $\sum_{u \in S} \sum_{v \in S} f(u, v) - \sum_{u \in S} \sum_{v \in S} f(v, u) = 0$ (S内部的流相互抵消或不计入跨切割的净流)。
    所以 $|f| = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) = f(S, T)$。

- **推论 (LEC13, p.3, Corollary)**：任何流 f 的值 $|f|$ 都不能超过任何切割 (S,T) 的容量 $c(S,T)$。
  $|f| \leq c(S,T)$
  - **证明**：
    $|f| = f(S,T) = \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u)$
    由于 $f(v,u) \leq c(v,u)$ 且 $f(v,u)$ 是从 T 到 S 的流，所以 $\sum f(v,u) \geq 0$ (如果只考虑正向流) 或者说 $f(v,u)$ 如果为负值（代表从 S 到 T 的"推回"），其绝对值不超过 $c(u,v)$。
    更直接地：$|f| = \sum_{u\in S,v\in T} f(u,v) - \sum_{v\in T,u\in S} f(v,u)$
    $\leq \sum_{u\in S,v\in T} f(u,v)$ (因为 $\sum_{v\in T,u\in S} f(v,u) \geq 0$ 如果只考虑实际正向流，或者说这项为负时会使得 $|f|$ 更大)。
    $\leq \sum_{u\in S,v\in T} c(u,v)$ (根据容量约束 $f(u,v) \leq c(u,v)$)
    $= c(S,T)$。

3. 最大流最小割定理 (Max-Flow Min-Cut Theorem) (LEC13, p.3, Theorem 7.9)

这是流理论中的一个核心定理，它将最大流问题与最小切割问题联系起来。

- **定理陈述**：对于一个流网络 $G = (V,E)$（源点为 s，汇点为 t）中的一个流 f，以下三个条件是等价的：
  i. f 是图 G 中的一个**最大流**。
  ii. 流 f 对应的**剩余网络 G_f 中不包含任何增广路径**。
  iii. 对于某个切割 (S,T)（这个切割实际上是一个最小切割），流的值等于该切割的容量，即 $|f| = c(S,T)$。
- **意义**：
  - 该定理表明，网络中的最大流值恰好等于该网络的最小切割容量。
  - 它为 Ford-Fulkerson 方法的正确性提供了理论基础：当算法终止时（即剩余网络中没有增广路径），此时得到的流就是最大流，并且其值等于某个最小切割的容量。
  - 如何找到这个最小切割？当 Ford-Fulkerson 方法终止时，设 S 是在最终的剩余网络 G_f 中从源点 s 可达的所有顶点的集合，而 $T = V - S$ 是其余顶点的集合。这个 (S,T) 就是一个最小切割，并且 $|f_{max}| = c(S,T)$。

4. Ford-Fulkerson 算法的实现细节 (LEC13, p.3)

回顾 `FORD-FULKERSON (G,s,t)` 算法（基本框架）：

```
FORD-FULKERSON(G,s,t)
1 for each edge (u,v) in E[G]
2     do f[u,v] = 0
3        f[v,u] = 0
4 while there exists a path p from s to t in the residual network G_f
5     do c_f(p) = min {c_f(u,v) : (u,v) is in p}  // 增广路径的容量
6        for each edge (u,v) in p
7            do f[u,v] = f[u,v] + c_f(p)
8               f[v,u] = -f[u,v] // 或 f[v,u] = f[v,u] - c_f(p)
```

(LEC13, p.3, "The Basic Ford-Fulkerson Algorithm")

- **第4行**：如何找到增广路径 p？这取决于具体的实现。可以使用 BFS (如 Edmonds-Karp 算法) 或 DFS。
- **算法的终止和正确性**：基于最大流最小割定理。当循环终止时，意味着 G_f 中没有增广路径，因此当前流 f 是最大流。

5. 备考要点

- **简答题考点**：
  - 定义网络中的 (s,t)-切割、切割的净流量和切割的容量。
  - 什么是最小切割?
  - 陈述最大流最小割定理（三个等价条件）。
  - Ford-Fulkerson 方法如何利用最大流最小割定理来确保找到最大流？（当找不到增广路径时，流达到最大，等于最小割）。
  - 在 Ford-Fulkerson 算法终止后，如何找到一个最小切割?
- **算法分析题考点**：
  - 给定一个网络和流，以及一个切割 (S,T)，计算该切割的净流量和容量。
  - 验证对于给定的流和切割，$|f| \leq c(S,T)$ 是否成立。
  - （结合Ford-Fulkerson）当算法终止时，识别出对应的最小切割。
- **设计证明题考点**：
  - 证明引理：$f(S,T) = |f|$。
  - 证明推论：$|f| \leq c(S,T)$。
  - (较难) 证明最大流最小割定理的等价性（例如，(2) $\Rightarrow$ (3) 的构造性证明，即通过 G_f 中 s 可达的点集构造出切割 S）。

图算法详解之 8.6 (D)：Edmonds-Karp 算法与 Ford-Fulkerson 分析 (LEC13)

Ford-Fulkerson 方法的效率取决于如何选择增广路径。如果增广路径选择不当，算法的运行时间可能不是多项式的。Edmonds-Karp 算法是对 Ford-Fulkerson 方法的一个改进，它规定了如何选择增广路径，从而保证了多项式时间复杂度。

1. Ford-Fulkerson 方法的分析与潜在问题 (LEC13, p.5)

- **整数容量**：如果所有边的容量都是整数，那么每次成功增广，流的值 $|f|$ 至少增加1 。
- **运行时间 (依赖于最大流值)**：如果最大流的值为 $|f^*|$，那么 Ford-Fulkerson 方法最多需要 $|f^*|$ 次增广 。如果每次寻找增广路径的时间（例如使用BFS或DFS）为 $O(E)$，则总时间复杂度为 $O(E|f^*|)$ 。
- **非多项式时间问题**：这个运行时间 $O(E|f^*|)$ **不是输入规模的多项式函数**，因为它依赖于最大流的值 $|f^*|$，而 $|f^*|$ 可能非常大，与 $|V|$ 或 $|E|$ 的关系不是多项式的 。
  - **例子 (LEC13, p.5, "The basic Ford-Fulkerson Algorithm" 图示)**：
    一个包含4个顶点 s,u,v,t 的网络。边 (s,u) 和 (v,t) 的容量为 1,000,000。边 (s,v) 和 (u,t) 的容量也为 1,000,000。边 (u,v) 的容量为 1。
    - 如果算法不幸地总是选择包含边 (u,v) 作为瓶颈的增广路径（例如，路径 $s \to u \to v \to t$），每次增广流量只增加1。

    - 最大流是 2,000,000（例如，通过路径 $s \to u \to t$ 和 $s \to v \to t$）。
    - 这种情况下，算法可能需要进行 2,000,000 次增广，性能非常差 。
- **有理数容量**：如果容量是有理数，可以通过乘以一个共同的倍数将它们转换为整数，然后应用上述分析 。
- **无理数容量**：如果容量是无理数，Ford-Fulkerson 方法甚至可能**永远不会终止** 。

2. Edmonds-Karp 算法 (LEC13, p.6)

Edmonds-Karp 算法是对 Ford-Fulkerson 方法的一个特定实现，它通过指定寻找增广路径的策略来改进效率。

- **核心改进** (LEC13, p.6)：在 Ford-Fulkerson 方法的第4行（或通用框架的第2步）中，**总是选择在剩余网络 G_f 中从 s 到 t 的一条最短增广路径**（其中路径长度是指边的数量，即每条边的权重视为1）。
- **实现方式**：可以使用**广度优先搜索 (BFS)** 来寻找剩余网络 G_f 中的最短增广路径，因为BFS能够找到按边数计的最短路径 。BFS的运行时间是 $O(E)$（因为 $|E_f| \leq 2|E|$）。
- **算法框架 (与Ford-Fulkerson类似，但明确了路径选择)**：
  i. 初始化流 $f = 0$。
  ii. `while` 在剩余网络 G_f 中**通过BFS**可以找到一条从 s 到 t 的增广路径 p:
  a. 计算路径 p 的剩余容量 $c_f(p)$。
  b. 沿着 p 增广流量 $c_f(p)$。
  iii. 返回流 f。
- **例子 (LEC13, p.5, "Run Ford-Fulkerson on this example")**：对于之前提到的可能导致 Ford-Fulkerson 性能差的例子，Edmonds-Karp 算法只需要两次迭代（增广）就能找到最大流 。
  i. 第一次BFS会找到路径 $s \to u \to t$（2条边）或 $s \to v \to t$（2条边）。假设找到 $s \to u \to t$，瓶颈容量1,000,000。流量增加1,000,000。
  ii. 第二次BFS会在剩余网络中找到另一条路径，例如 $s \to v \to t$，瓶颈容量1,000,000。流量再增加1,000,000。
  总流量达到2,000,000，算法终止。

3. Edmonds-Karp 算法的效率分析 (LEC13, p.6)

- **关键引理**：在 Edmonds-Karp 算法的执行过程中，对于所有顶点 $v \in V - \{s, t\}$，从源点 s 到 v 在剩余网络 G_f 中的最短路径距离 $\delta_f(s, v)$（按边的数量计）是单调不减的。
- **增广次数的上界**：可以证明 Edmonds-Karp 算法总共进行的增广操作次数不超过 $O(VE)$ 次。
  - 证明思路比较复杂，涉及到分析每次增广后，至少有一条边的剩余容量变为0，并且这条边是当前最短增广路径上的“瓶颈边”。可以论证这样的瓶颈边在整个算法执行过程中，其作为特定距离最短路径上的瓶颈边的次数是有限的。
- **总时间复杂度** (LEC13, p.6)：
  - 每次寻找最短增广路径使用BFS，时间复杂度为 $O(E)$。
  - 总共进行的增广次数为 $O(VE)$。
  - 因此，Edmonds-Karp 算法的总时间复杂度为 $O(VE^2)$。
  - 这是一个多项式时间复杂度，与流的值 $|f^*|$ 无关。

4. 其他更优的改进算法 (LEC13, p.6)

虽然 Edmonds-Karp 算法是多项式的，但存在更高效的最大流算法：

- **Push-relabel 算法 (推送-重贴标签算法)**：一种常见的版本是 $O(V^2E)$。
- **Relabel-to-front 算法 (前置重贴标签算法)**：是 Push-relabel 的一种改进，时间复杂度为 $O(V^3)$ 。
- **Scaling Max-Flow 算法 (比例缩放最大流算法)**：时间复杂度为 $O(E^2 \log C)$，其中 C 是网络中边的最大整数容量 。
  这些算法通常更为复杂。

5. 备考要点

- **简答题考点**：
  - 为什么朴素的 Ford-Fulkerson 方法可能不是多项式时间的？（依赖于 $|f^*|$ 和增广路径的选择）
  - Edmonds-Karp 算法对 Ford-Fulkerson 方法做了什么关键改进？（总是选择最短的增广路径，用BFS实现）
  - Edmonds-Karp 算法的时间复杂度是多少？它是否依赖于流的值？
- **算法分析题考点**：
  - 对于给定的网络，如果使用 Edmonds-Karp 算法，指出其选择的第一条（或前几条）增广路径会是什么。
  - 理解 Edmonds-Karp 算法的增广次数界 $O(VE)$（可能不会要求证明，但应知道这个结论）。
- **设计证明题考点**：
  - 描述 Edmonds-Karp 算法的伪代码或其与 Ford-Fulkerson 的主要区别。

图算法详解之 8.6 (E)：最大二分匹配 (Maximum Bipartite Matching) (LEC13)

最大二分匹配问题是在二分图中找到一个包含边数最多的匹配。这个问题可以通过将其转化为最大流问题来解决。

1. 二分图与匹配的定义 (LEC13, p.6)

- **二分图 (Bipartite Graph)**：一个无向图 $G = (V, E)$，其顶点集 V 可以被划分为两个不相交的子集 L 和 R ($V = L \cup R, L \cap R = \emptyset$)，使得图中的每一条边都连接 L 中的一个顶点和 R 中的一个顶点 。也就是说，L 内部或 R 内部不存在边 。
  - 例如，可以将 L 看作一组工人，R 看作一组工作，边 (u, v) 表示工人 $u \in L$ 可以胜任工作 $v \in R$ 。
- **匹配 (Matching)**：在图 $G = (V, E)$ 中，一个匹配 M 是边集 E 的一个子集，使得对于 V 中的所有顶点 v ，最多只有一条 M 中的边与 v 相关联（即 M 中的任意两条边都没有公共顶点）。
- **最大匹配 (Maximum Matching)**：一个图中包含边数最多的匹配称为最大匹配 。其包含的边数称为匹配的**基数 (cardinality)** 。
  - 在工人和工作的例子中，最大匹配意味着为尽可能多的人提供工作 。

例子 (LEC13, p.7)：
PPT中展示了“not maximum”和“maximum”匹配的图示 。一个最大匹配不一定唯一。

2. 将最大二分匹配问题转化为最大流问题 (LEC13, p.7)

我们可以通过构造一个相应的流网络 G'，将二分图 $G = (L \cup R, E)$ 的最大二分匹配问题转化为 G' 上的最大流问题 。

- **构造相应的流网络 $G' = (V', E')$** (LEC13, p.7)：
  i. **创建源点 s 和汇点 t**：在原二分图的基础上，添加一个新的源点 s 和一个新的汇点 t。
  ii. **从 s 到 L 的边**：对于 L 中的每一个顶点 $u \in L$，添加一条从源点 s 到 u 的有向边 (s, u)，并将其容量 $c(s, u)$ 设为1 。
  iii. **从 L 到 R 的边**：对于原二分图 G 中的每一条边 (u, v)（其中 $u \in L, v \in R$），在流网络 G' 中添加一条从 u 到 v 的有向边 (u, v)，并将其容量 $c(u, v)$ 设为1 。
  iv. **从 R 到 t 的边**：对于 R 中的每一个顶点 $v \in R$，添加一条从 v 到汇点 t 的有向边 (v, t)，并将其容量 $c(v, t)$ 设为1 。
- **关键结论** (LEC13, p.7)：在这个构造出来的流网络 G' 中计算得到的**最大流的值**，等于原二分图 G 中**最大匹配的基数**（即最大匹配包含的边数）。

3. 为什么最大流对应最大匹配?

- **从匹配到流**：如果二分图 G 中有一个匹配 M，我们可以构造一个流 f。对于 M 中的每条边 (u, v)（$u \in L, v \in R$），我们设置流 $f(s, u) = 1, f(u, v) = 1, f(v, t) = 1$。对于不在 M 中的边，流量为0。这个流的值等于 $|M|$。由于所有容量都是1，这个流是合法的。
- **从流到匹配**：如果在流网络 G' 中找到了一个整数值的流 f（由于所有容量都是整数，Ford-Fulkerson 方法会找到整数流），那么我们可以从中构造出一个匹配 M。匹配 M 包含所有满足 $f(u, v) = 1$ 的边 (u, v)，其中 $u \in L, v \in R$。
  - 由于从 s 到每个 $u \in L$ 的边的容量为1，所以 $f(s, u)$ 只能是0或1。
  - 由于从每个 $v \in R$ 到 t 的边的容量为1，所以 $f(v, t)$ 只能是0或1。
  - 根据流守恒性，对于每个 $u \in L$，如果 $f(s, u) = 1$，那么它流向 R 的总流量也必须是1（通过某条边 (u, v) 流出）。
  - 类似地，对于每个 $v \in R$，如果它有流入，那么它流向 t 的总流量也等于该流入量。
  - 这意味着每个 L 中的顶点最多与一条流为1的 (L, R) 边相连，每个 R 中的顶点也最多与一条流为1的 (L, R) 边相连。因此，这些流为1的 (L, R) 边构成一个匹配。匹配的大小等于流的值。
- **最大流最小割定理的应用**：可以更形式化地通过最大流最小割定理来证明这个对应关系。构造的流网络中的切割与二分图中的顶点覆盖（vertex cover）有关（Kőnig 定理：二分图中的最大匹配数等于其最小顶点覆盖数）。

4. 求解方法 (LEC13, p.7)

1. 根据给定的二分图 G 构造出对应的流网络 G'（如上所述，所有边的容量设为1）。
2. 使用任何最大流算法（例如 Ford-Fulkerson 方法或其具体实现如 Edmonds-Karp 算法）在 G' 中计算从 s 到 t 的最大流 f^*。
3. 最大流的值 $|f^*|$ 就是原二分图 G 中最大匹配的基数。
4. 最大匹配 M 由那些在 G 中对应于 G' 中流量 $f(u, v) = 1$（其中 $u \in L, v \in R$）的边组成。

- **时间复杂度**：如果使用 Edmonds-Karp 算法计算最大流，其复杂度为 $O(V'E'^2)$，其中 V' 和 E' 是流网络 G' 的顶点数和边数。
  - 在构造的流网络 G' 中：
    - $V' = |V| + 2 = |L| + |R| + 2$。
    - $E' = |E| + |L| + |R|$。

- 因此，求解最大二分匹配的时间复杂度通常由所选的最大流算法决定。对于 $|V|$ 个顶点和 $|E|$ 条边的二分图，如果使用 Hopcroft-Karp 算法（专门为二分匹配设计，比一般最大流算法更快），时间复杂度可以达到 $O(E\sqrt{V})$。如果使用基于最大流的方法，如 Dinic 算法，复杂度也可以很好。对于 Edmonds-Karp，是 $O(VE^2)$（这里 V, E 指的是原二分图的）。

例子 (LEC13, p.7):
PPT中给出了一个二分图，并展示了其对应的流网络以及一个最小割。最大流的值（例如3）等于最大匹配的边数。

5. 备考要点

- **简答题考点**:
  - 什么是二分图？什么是图中的匹配？什么是最大匹配？
  - 描述如何将最大二分匹配问题转化为最大流问题（即如何构造相应的流网络 G'，包括边的方向和容量设置）。
  - 在构造的流网络中，最大流的值与原二分图的最大匹配基数之间有什么关系？
- **算法分析题考点**:
  - 给定一个二分图，画出其对应的流网络。
  - 在构造的流网络上，（可能结合一个简单的最大流算法）找出最大流，并据此确定原二分图的一个最大匹配。
- **设计证明题考点**:
  - 描述将最大二分匹配问题转化为最大流问题的构造方法。
  - 简要解释为什么这个转化是有效的（即为什么最大流值等于最大匹配数）。