CS2045M: 高级数据结构与算法分析 **2025-2026 学年秋冬学期**

Lecture 8: 动态规划

编写人：吴一航 yhwu_is@zju.edu.cn

我想这毫无疑问将是期中之前这门课最令人感到噩梦的一章，因为动态规划并不是一个很 trivial 的方法，不像回溯、分治以及之后的贪心自然直观。但我相信，经过本讲的讲解与推导，以及诸位自己的仔细思考，至少对这门课的考试、未来的研究和工作都能打下坚实的基础。

8.1 动态规划的背景

8.1.1 动态规划的来由

1966 年，Dynamic Programming 登上国际顶级期刊《Science》，理查德 · 贝尔曼（Richard Bellman）为唯一作者，该论文被引 28421 次。贝尔曼在他的自传中写道：“1950 年秋季，我在兰德公司工作。我的第一个任务是为多阶段决策过程找到一个名称。”

“一个有趣的问题是，“动态规划” 这个名字是从哪里来的？20 世纪 50 年代不是搞数学研究的好年代。当时在华盛顿有一位非常有趣的人，他叫威尔逊 (Charles E. Wilson, 1953-1957 年任美国国防部长)，是当时的美国国防部长，他对 “研究” 这个词有病态的恐惧和憎恨。提到研究，他的脸会涨得通红，如果有人在他面前用 “研究” 这个词，他就会变得很暴力。你可以想象他对数学这个词的感受。兰德公司受雇于空军，而空军的老板基本上是威尔逊。因此，我觉得我必须做点什么来瞒过威尔逊和空军，不让他们知道我实际上是在兰德公司做数学研究。”

“所以，给这个研究起个什么名字好呢？首先，我对 planning，decision making 和 thinking 比较感兴趣。但是由于种种原因，planning 这个词并不合适。因此，我决定使用 programming 这个词。我想让大家明白，这是动态的，是多级的，是时变的——我想，这个名字可以起到一石二鸟的效果。作为形容词，dynamic 还有一个很有趣的特性，那就是 dynamic 这个词不可能有贬义。我们不可能想出一些可能使它具有贬义的组合。因此，我认为 dynamic programming 是个好名字。这是一件连国会议员都不会反对的事情。所以我把它作为我研究活动的 “保护伞”。”（以上内容摘自这个知乎文章）

8.1.2 基础问题：爬楼梯

首先看动态规划思想的最简单的应用，从中体会动态规划解决问题的基本思想。

【爬楼梯】假设你正在爬楼梯。需要爬 n 阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶，求有多少种

不同的方法爬到楼顶，要求算法在线性时间内解决这一问题。

现在尝试分析这一问题，有的同学可能会想这是一个计数问题，所以使用中学的排列组合思想即可解决——当然可以，但这里希望讨论算法设计，所以不会讨论这种直白的方法，当然感兴趣的读者应该不难想到如何使用排列组合解决这一问题（通过上 2 级台阶的次数分类是一种办法）。至于算法设计方面，首先会想到曾经学过的回溯和分治思想，回溯法解决这一问题的想法非常简单，或许与暴力枚举区别不大，时间复杂度显然不可能是线性的。如果本题要求列出所有的可能情况，那么回溯是合理的，但只需要计数，因此除非找不到更好的办法，否则不会采用效率低下的回溯法。

分治或许是一个降低复杂度的好方法，若 n 为偶数，那么想法可以是先爬前一半的台阶，再爬后一半的台阶，然后两种方法数相乘。然而这并不完整，因为一次可以爬两个台阶，因此第 $n/2$ 个台阶不一定是分界线，此时相当于先走了 $n/2-1$ 个台阶，然后一次跨两个台阶，再走后 $n/2-1$ 个台阶，因此有

$$f(n) = f(n/2)^2 + f(n/2-1)^2,$$

其中 $f(n)$ 表示爬 n 阶楼梯的方法数。因此偶数的情况分治只需要计算 $f(n/2)$ 和 $f(n/2-1)$ 即可，递归直到 base case，时间复杂度的递推式大致是 $T(n) = 2T(n/2) + O(1)$，看起来很美好，因为用上一节课的知识知道这是线性的复杂度，满足要求。然而如果考虑奇数，需要计算 $f(\lceil n/2 \rceil)$、$f(\lfloor n/2 \rfloor)$、$f(\lceil n/2 \rceil - 1)$ 和 $f(\lfloor n/2 \rfloor - 1)$，这样复杂度递推式就大致变成了 $T(n) = 4T(n/2) + O(1)$，这样看来复杂度是 $O(n^2)$ 的，显然不是我们想要的。

分治法的失败迫使我们思考新的解法，或许可以对这个问题的结构进行一些观察。可以看为什么这个问题分治法并没有之前学习的那些问题合适：实际上是因为问题对半分了之后还无法解决这一问题，还要递归解决很多近似于对半分的子问题才能将所有问题解决（比如还需要考虑 $n/2-1$ 的情况）——不像归并排序，对半分之后两边问题解决了，然后只需要解决一个线性合并问题即可。为了算法的效率性，利用子问题的组合来帮助解决最后的问题显然是一条正确的道路，既然对半分不适合，但这个问题的解应该是从小问题的解出发开始逐步得到最后整个问题的解。那么解的最后一步应该是一个好的出发点，因为它可能依赖于前面小问题的解，这样或许可以得到一些新的关于最终解的结构的观察。不难知道爬楼梯的方法中，最后一步有两种可能：

1. 最后一步爬 1 个台阶；
2. 最后一步爬 2 个台阶。

如果能知道这两种情况下爬楼梯的方法数，那么总的爬楼梯的方法数就是把以上两种情况的方法数求和。事实上很容易发现，如果最后一步爬 1 个台阶，那么爬楼梯的方法数就是爬 $n-1$ 阶楼梯的方法数；如果最后一步爬 2 个台阶，那么爬楼梯的方法数就是爬 $n-2$ 阶楼梯的方法数。由此就得到了一个递推关系：$f(n) = f(n-1) + f(n-2)$。这个递推关系是不是很熟悉？没错，这就是斐波那契数列的定义！于是可以怀着激动的心情，像 PPT 第 2 页那样写出一个一个美好的递归的解法，如果这是 PTA 的一个习题，我想邪恶的出题老师一定会让你感受到运行超时的绝望！为什么呢？事实上递归算法如此慢的原因在于算法模仿了递归。为了计算 F_N，存在一个对 F_{N-1} 和 F_{N-2} 的调用。然而，由于 F_{N-1}

递归地对 F_{N-2} 和 F_{N-3} 进行调用，因此存在两个单独的计算 F_{N-2} 的调用。如果探试整个算法，那么可以发现，F_{N-3} 被计算了 3 次，F_{N-4} 计算了 5 次，而 F_{N-5} 则是 8 次，等等，因此冗余计算的增长是爆炸性的。

因此需要一个实现上的技术来帮助减少冗余的计算。其实这一想法非常简单，只要用一个数组把已经计算出来的值存起来：首先初始化斐波那契数列的第 0 和 1 项，放在数组下标 0 和 1 的位置，然后基于此可以计算出第 2 项，放在数组下标 2 的位置，然后基于存储的第 1 和 2 项算出第 3 项放在数组下标 3 的位置，以此类推直到算出第 n 项。显然这一迭代的算法复杂度是线性的，但因为使用了数组存储，所以使用了额外的空间，所以是一种典型的 “空间换时间” 的算法（当然其实递归也很耗空间），这一重要的思想称其为 “记忆化”，非常形象，因为就是利用额外的空间记住曾经算过的结果，从而避免了重复计算的问题。换句话说，原先递归的方法实际上是一种 “自顶向下” 的方法，它符合直接思路：最后长度为 n 的问题需要长度为 $n-1$ 和 $n-2$ 的求和，那代码直接利用递归计算递归式显得非常自然。但是重复的计算迫使我们放弃这种自然的想法，转而采用自底向上从小问题逐步迭代构建原问题的解法。

总结一下上面的讨论：在爬楼梯这一简单而经典的问题中，通过对最后一步两种情况的分类，将整个问题的解转化为了两个子问题解的求和，即有一个从子问题的解到原问题的解的递推公式，然后通过记忆化的方式求解递推式。将上述特点抽象出来：一个问题，它的最优解可以表达为一些合适的子问题的最优解的递推关系，则称这一问题具有**最优子结构性质**（因为大问题的最优解可以直接依赖于小问题的最优解）。然后求解这一递推式，通过设置好 base case（这里也用 base case 指代最简单的情况，但注意这时不是递归了），然后通过记忆化的方法，使用迭代算法而非费时的递归算法避免冗余计算，得到一个时间复杂度令人满意的算法，这就是动态规划的基本想法。

更一般的来说，上面的问题只是借用了动态规划的简单思想，接下来的例子才是正式的开始。在开始旅程之前，先给出动态规划的一般范式。动态规划方法通常用来求解最优化问题（optimization problem）。这类问题可以有很多可行解，每个解都有一个值，我们希望寻找具有最优值（最小值或最大值）的解。称这样的解为问题的一个最优解（an optimal solution），而不是最优解（the optimal solution），因为可能有多个解都达到最优值。

通常按如下 4 个步骤来设计一个动态规划算法：

1. 刻画一个最优解的结构特征；
2. 递归地定义最优解的值；
3. 计算最优解的值，通常采用自底向上的方法；
4. 利用计算出的信息构造一个最优解。

步骤 1-3 是动态规划算法求解问题的基础。如果仅仅需要一个最优解的值，而非解本身，可以忽略步骤 4（上面的例子就没有这一步骤，当然上面的例子也不是最优化问题，但接下来马上就可以看到包含步骤 4 的例子）。如果确实要做步骤 4, 有时就需要在执行步骤 3 的过程中维护一些额外信息，以便用来构造一个最优解。

更精炼地，动态规划就是为一个具有所谓最优子结构性质（即原问题最优解可以由子问题最优解递推得到）的最优化问题寻找一个子问题到原问题的递推式，然后用记忆化方法求解，最后有时需要构造出这一最优解。接下来将开始正式的旅途，我们将会从易到难，观察一些经典的例子的解法。动态规划是一个经验和 trick 都很重要的算法设计思想，因此见识很多不同风格、不同特点的例子并加以自己的思考是非常重要的。

8.2 加权独立集合问题

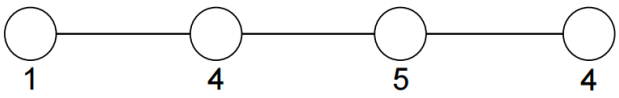
前面的爬楼梯问题只能说是一个简单的热身，引出了动态规划的基本想法，即最优子结构的递推以及记忆化。接下来将正式研究如何将动态规划应用到最优化问题中，展示上面给出的求解一般流程。

8.2.1 问题描述

第一个问题称之为加权独立集合问题。可以从一个带故事背景的问题出发：

【打家劫舍】你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。给定一个代表每个房屋存放金额的非负整数数组，计算你不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

将这一问题抽象出来，实际上就是：考虑一个无向图 G，其上所有点都在一条线上（这种图称其为路径图），每个点都有一个非负权重。称 G 的独立集合是指顶点互不相邻的子集（换句话说独立集合不会同时包含一条边上的两个点），然后要求解一个具有最大顶点权重和的独立集合。这一问题称为一维的加权独立集合问题，当然显然有对一般的图的版本，但那一问题之后会知道是很困难的，所以这里只介绍一维的。



上图是一个简单的例子，在这个图中有 8 个独立子集：空集、四个单点集、第 1 和第 3 个点、第 1 和第 4 个点、第 2 和第 4 个点三个两点集。其中最大的独立集合显然是第 2 和第 4 个点构成的集合，其权重和为 8，即小偷的最优选择是偷 2 和 4 两家，最大收益是 8。需要注意的是，题目的假设中每个顶点都有非负权重，所以最优解的顶点应当是越多越好。

8.2.2 最优子结构

开始尝试解决这一问题。在使用动态规划之前，当然可以先看看其它方法是否可以。回溯当然可行，但复杂度显然太高；分治法读者可以自己尝试一下，事实上又会落入前面爬楼梯那样子问题纠缠的情况，因此也并不是很合适。所以尝试动态规划，寻找最优子结构和递推关系。

为了找到这一递推关系，可以做一个简单的自顶向下的思维实验（这在之后的问题中经常用到）：如果你拿到一个路径图，那么你对于这一问题的解首先会想什么？你应该想知道最后一个点是不是在最优解中，因为只有知道了这个点在不在才能进一步向前继续构造出整个问题的解！更具体地说，设 $G = (V, E)$ 表示 n 个顶点的路径图，具有边 $(v_1, v_2), \ldots, (v_{n-1}, v_n)$，并且每个顶点 v_i 都有非负权重 w_i。根据前面的分析，应当考虑 v_n 是否在最优解 S 中，显然可以分为如下两种情况讨论：

1. 如果 v_n 不在最优解 S 中，那么 S 实际上可以看成前 $n-1$ 个点构成的路径图 G_{n-1} 组成的子问题的一个可行解。事实上 S 一定是 $n-1$ 个点的路径图的最大加权独立子集（即最优解），否则如果能找到更优的解 S^* 是 G_{n-1} 的最优加权独立集合，那么 S^* 作为子问题的最优解，是原问题的可行解，因此一定不会优于原问题的最优解 S，这就产生了矛盾：前面假设 S^* 优于 S，现在推出 S^* 不会优于 S；

2. 如果 v_n 在最优解 S 中，那么由于是独立集合问题，那么 v_{n-1} 就不能在最优解中，因为 v_{n-1} 和 v_n 相邻。所以类似于第一种情况的分析，$S - \{v_n\}$ 一定是前 $n-2$ 个点构成的路径图 G_{n-2} 的最优解，所以整个问题的最优解就是 G_{n-2} 的最优解加上 v_n。

由此，整个问题的最优解依赖于前 $n-1$ 和 $n-2$ 个点构成的子图的最优解，或者说，这两个子问题的最优是构成原问题最优的基础，这就是这一问题对应的最优子结构性质。设前 i 个点的最优加权独立集合的权重之和为 W_i，可以写出递推关系：

$$W_n = \max\{W_{n-1}, W_{n-2} + w_n\},$$

这里是最后一步的递推关系，事实上可以不断利用这一递推关系自顶向下向前构建出整个问题的解，即有更一般的表达

$$W_i = \max\{W_{i-1}, W_{i-2} + w_i\},$$

其中 $i = 2, 3, \cdots, n$，$W_0 = 0$。于是，到目前为止完成了动态规划四步框架的前两步，接下来需要写代码实现这一算法。

利用之前介绍的记忆化方法，从 base case 出发，自底向上构建出整个问题的解，否则自顶向下递归会重复计算太多的子问题。很显然需要一个循环，从开始的结点遍历到最后，用递推式逐步向后计算出截止于每个点的最优解，最后得到整个问题的解。但需要设置好 base case，否则将无从开始我们的迭代。事实上 base case 就是最 trivial 的情况，即 0 个点和 1 个点的情况，有了这两种情况，接下来利用递推式就可以算出所有子问题以及原问题的解了，伪代码如下图。很显然，这一算法的时间复杂度是 $O(n)$，是一个非常高效的算法。

$A :=$ length-$(n+1)$ array // subproblem solutions
$A[0] := 0$ // base case #1
$A[1] := w_1$ // base case #2
for $i = 2$ to n **do**
  // use recurrence from Corollary 16.2
  $A[i] := \max\{\underbrace{A[i-1]}_{\text{Case 1}}, \underbrace{A[i-2] + w_i}_{\text{Case 2}}\}$
return $A[n]$ // solution to largest subproblem

8.2.3 解的重构

现在还剩下动态规划框架的最后一步，即有时候可能不仅需要最优解的值，还需要最优解本身包含了哪些点，这就需要解的重构。事实上这一问题想要重构出解非常简单，只需要观察上面伪代码的数组 A 留下的线索，结合递推式即可。从数组 A 中，首先可以看出最后一个点是否在最优解中，因为

$$W_n = \max\{W_{n-1}, W_{n-2} + w_n\},$$

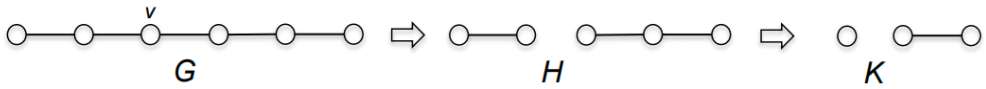
如果 $W_n = W_{n-1}$，那么最后一个点不在最优解中，此时回退一格，继续重建图 G_{n-1} 的最优解；否则最后一个点在最优解中，回退两格重建图 G_{n-2} 的最优解。具体伪代码实现中还有 base case 的考量，读者可以自己思考然后与下图中的参考答案比对：

$S := \emptyset$ // vertices in an MWIS
$i := n$
while $i \geq 2$ **do**
  if $A[i-1] \geq A[i-2] + w_i$ **then** // Case 1 wins
    $i := i - 1$ // exclude v_i
  else // Case 2 wins
    $S := S \cup \{v_i\}$ // include v_i
    $i := i - 2$ // exclude v_{i-1}
if $i = 1$ **then** // base case #2
  $S := S \cup \{v_1\}$
return S

总而言之，这是一个简单而经典的动态规划的例子，这一例子将之前给出的框架完整地展示了一遍。接下来是一些思考题，检验读者是否至少理解了本题的解法：

【思考:】本问题中每个顶点都有非负的权重，假设允许顶点的权重为负数，那么上面的解法是否仍然适用？如果不适用，你能否构造出一个不适用的图并给出一个新的解法？

【思考:】这个问题规划了一种方法，该方法解决比路径图更为复杂的图的加权独立集合问题。考虑一个任意的无向图 $G = (V, S)$，所有顶点的权重均不为负，并且任意一个顶点 $v \in V$ 具有权重 w_v。通过从 H 中删除 v 的相邻顶点以及相关联的边得到 K，如下图所示：



设 W_G、W_H 和 W_K 分别表示 G、H 和 K 的最大加权独立集合的总权重，并考虑下面这个公式：

$$W_G = \max\{W_H, W_K + w_v\},$$

下面哪些说法是正确的（选择所有正确的答案）：

1. 这个公式对于路径图并不总是正确的；
2. 这个公式对于路径图总是正确的，但对于树并不总是正确的；
3. 这个公式对于树总是正确的，但对于任意的图并不总是正确的；
4. 这个公式对于任意的图都是正确的；
5. 这个公式对于树的加权独立集合问题可以产生一种线性时间的算法；
6. 这个公式对于任意图的加权独立集合问题可以产生一个线性时间的算法。

【思考:】（最大子序列和问题）同上一讲分治的最大子序列和问题，要求写出递推关系，并且算法要给出这一最大子序列。

【思考:】（零钱兑换问题）给你一个整数数组 `coins`，表示不同面额的硬币；以及一个整数 `amount`，表示总金额。计算并返回可以凑成总金额所需的最少的硬币个数，以及最少硬币个数对应的兑换方式。如果没有任何一种硬币组合能组成总金额，返回 -1。你可以认为每种硬币的数量是无限的。

例如：`coins` $= [1, 2, 5]$，`amount` $= 11$，则答案为 3，最优兑换为 $11 = 5 + 5 + 1$。

8.3 背包问题

8.3.1 问题描述

背包问题是一个最最经典的动态规划问题，这里首先介绍最基础的背包问题，即 0-1 背包问题。这一问题的描述如下：有 n 个物品，每个物品的重量为 s_i，价值为 v_i，有一个容量为 C 的背包，希望找到一个最优的装载方案，使得背包中的物品总价值最大。这一问题的特点是每个物品你要么不放进包里，要么完整的 1 个放进去，因此称为 0-1 背包问题。

一个简单的例子如下：假设有 4 个物品，重量分别为 $s = [2, 1, 3, 2]$，价值分别为 $v = [12, 10, 20, 15]$，背包容量为 $C = 5$，那么最优的装载方案是将第 1、2、4 个物品放进背包，总价值为 $12 + 10 + 15 = 37$。

8.3.2 最优子结构：一维似乎不再适用

完全可以仿照着前面讨论的两个问题的解法来解决背包问题。首先考虑最后一个物品是否在最优解中，然后根据这一情况划分子问题，然后递推：

1. 如果第 n 个物品不在最优解 S 中，即最优方案排除了最后一件物品，因此它可以看成仅由前 $n-1$ 个物品组成的子问题的一种可行解决方案，并且 S 必定是这个子问题的最优解，否则可以找到一个更优的解 S^*，S^* 作为子问题的最优解，是原问题的可行解，因此一定不会优于原问题的最优解 S，这就产生了矛盾（事实上理由和前面加权独立集合问题一样）；
2. 如果第 n 个物品在最优解 S 中，这种情况只有在 $s_n \leqslant C$ 时才有意义。类似于加权独立集合问题，我们希望 $S-\{n\}$ 是前 $n-1$ 个物品组成的子问题的最优解，但这显然是错误的！如果 $S-\{n\}$ 就已经把 W 几乎占满，那最后 n 根本就放不进来了。因此需要对子问题的设置略做调整：$S-\{n\}$ 应当是前 n 个物品在背包容量为 $C-s_n$ 的情况下的最优解！因为我们知道 n 在解中，那么除去 n 之外的其它解的重量之和最多就是 $C-s_n$，因此 $S-\{n\}$ 应当是前 n 个物品在背包容量为 $C-s_n$ 的情况下的可行解，并且应当是最优解，否则又可以同前面情况的讨论构造出矛盾。

因此根据上述讨论，此时的子问题不再仅仅是只由单个参数（即是第几个输入）控制，还由背包容量控制，因此需要一个二维的数组来存储子问题的解。设 $V_{i,c}$ 表示总重量不超过 c 的前 i 件物品组成的子集的最大总价值，那么可以得到如下递推关系：

$$V_{i,c}=\begin{cases}V_{i-1,c}, & s_i > c;\\ \max\{V_{i-1,c}, V_{i-1,c-s_i}+v_i\}, & s_i \leqslant c.\end{cases}$$

所以分析完了最优子结构（原问题和子问题最优解的依赖关系），写出了递推式，接下来就是写代码实现这一算法，首先需要设置好 base case，然后利用递推式自底向上构建出整个问题的解。根据递推式，只要 $i-1$ 的所有情况都确定了，递推就可以往 i 推进，所以 base case 需要对 i 的最小情况（即 $i=0$）设置好就可以了，而 $i=0$ 时 $V_{i,c}$ 显然是 0，因为根本没有物品可以放。伪代码如下页图所示。一个显然的事情是，这一算法的时间复杂度是 $O(nC)$。

```
// subproblem solutions (indexed from 0)
A := (n+1) × (C+1) two-dimensional array
// base case (i = 0)
for c = 0 to C do
    A[0][c] = 0
// systematically solve all subproblems
for i = 1 to n do
    for c = 0 to C do
        // use recurrence from Corollary 16.5
        if s_i > c then
            A[i][c] := A[i-1][c]
        else
            A[i][c] :=
              max{A[i-1][c], A[i-1][c-s_i] + v_i}
                  Case 1          Case 2
return A[n][C] // solution to largest subproblem
```

8.3.3 解的重构

和前面的加权独立集合问题一样，还是根据递推式从数组中挖掘出最优解，建议读者自己思考，下面给出参考代码：

```
S := ∅              // items in an optimal solution
c := C                        // remaining capacity
for i = n downto 1 do
    if s_i ≤ c and A[i-1][c-s_i] + v_i ≥ A[i-1][c] then
        S := S ∪ {i}        // Case 2 wins, include i
        c := c - s_i          // reserve space for it
    // else skip i, capacity stays the same
return S
```

8.3.4 降维

事实上，对于这种多维的背包问题，经常可以考虑这样一种优化策略：即有没有可能数组的维数可以下降，例如这里是不是一维数组就能帮我们做好所有的事情。事实上如果不考虑解的重建是完全可以的（考虑解的重建可以参考后面矩阵乘法的方法，那样可能又要维护新的二维数组），因为事实上第 i 个物品对应的所有容量的解只依赖于第 $i-1$ 个物品的所有容量的解，因此事实上数组完全可以不用 i 这一维，每次循环对某个 $i-1$，算出所有容量的解，保存在只有容量这一个 index 的数组中，然后再算下一个 i 的所有容量的解覆盖掉前面的解。这样看起来很完美，但有一点需要注意，就是第 i 轮数组的

$A[c]$ 会依赖于更小的 c 在 $i-1$ 轮的解，所以如果 c 的遍历还是从小到大的，那么就可能提前把需要的 $i-1$ 轮的解覆盖，所以对 c 应当从大到小遍历，这样就不会出现错误的覆盖。

【思考:】（背包问题的变体）背包问题有大量有趣的变体，读者可以参考网上的资料进行学习，此处限于体量和时间无法展开讲解。

8.4 矩阵乘法的计算顺序

前面的问题似乎带给我们一些错觉：似乎动态规划就是看输入的最后一个元素是否在最优解中，然后就有两种最优解的构成形式，对应到两个子问题递推式取最优即可。事实上显然不是所有问题都这么简单的，下面这一问题将会展示子问题可能有很多种的情况。

8.4.1 问题描述

PPT 第 4 页描述了这一问题，简单而言就是要计算一系列矩阵链相乘的顺序，使得总的计算代价最小。PPT 上给出了一个非常极端的例子，表明最优和最差的计算顺序的差距是巨大的。

8.4.2 最优子结构：比较的子问题不一定只有两种

考虑矩阵链相乘 $M_1M_2\cdots M_n$，每个矩阵的大小为 $r_{i-1}\times r_i$。记

$$M_{ij}=M_iM_{i+1}\cdots M_j,$$

显然前面的三个问题给我们了不少的经验：应当考虑最后一个矩阵，然后依据这一矩阵可能的相乘方式划分最优解的可能性。很简单的，第一种情况对应的计算方式是 $M_{1,n-1}M_n$，即先用算前 $n-1$ 个矩阵相乘的最优方式计算前 $n-1$ 个矩阵的乘积，最后与 M_n 相乘；第二种情况的计算方式是 $M_{1,n-2}M_{n-1,n}$，即先用算前 $n-2$ 个矩阵相乘的最优方式计算前 $n-2$ 个矩阵的乘积，最后与 $M_{n-1,n}$ 的结果相乘。这是根据前面的问题的经验得到的，看起来很完美，但事实上真的只有这两种情况吗？显然不是的！注意，请回到分类的根源标准：此时分类的依据是 M_n 的计算方式，那么对于如下乘积形式：

$$M_{1i}M_{i+1,n-1}M_n,$$

只要是不同的 i，事实上都可能是不同的乘积方式（对于 M_n 而言，不同的 $M_{i+1,n-1}$ 与其相乘时如果各个 $M_{i+1,n-1}$ 的行数不一样，那就会带来不同的复杂度），因此情况远远没有我们想的那么简单。

所以不应当简单利用最后一个矩阵的相乘方式进行讨论，而是思考其它的子问题划分方式。应当停下来回到最开始讨论动态规划的想法，而非局限于一些过拟合的错误思路：进行自顶向下的思维实验，当你面对一个完全没有任何划分的矩阵链乘法时，你的一个很直接的想法就应当是首先确定这第一刀的

划分应该出现在哪里——正如面对一个加权独立集合问题时，需要首先知道最后一个点是否在最优解中一样，否则将无法自顶向下推进得到最优解的剩余部分是什么样的。于是考虑所有的

$$M_{1i}M_{i+1,n}, \quad 1 \leqslant i \leqslant n-1,$$

分解，那么整个问题的最优解就是所有上式中 $n-1$ 个分解的最优的那个对应的情况。因为是自顶向下的思考，那么在之前的步骤中应当已经得到了所有的 M_{1i} 和 $M_{i+1,n}$ 的最优乘法顺序，因此对于每一种分解，总的最小时间应当等于 M_{1i} 和 $M_{i+1,n}$ 的最小时间求和加上 M_{1i} 和 $M_{i+1,n}$ 相乘所需的时间（即 $r_0r_ir_n$），然后可以得到这 $n-1$ 种情况的最优时间，接下来只需取出它们中的最优值就是整个问题的最优解。或许说起来有点难懂，看下面这一递推式就会非常清楚：

$$m_{1n} = \min\{m_{1i} + m_{i+1,n} + r_0r_ir_n\},$$

其中 m_{ij} 表示 M_{ij} 的最优乘法时间。更一般地有

$$m_{ij} = \begin{cases} 0, & i = j; \\ \min\limits_{i \leqslant k < j}\{m_{ik} + m_{k+1,j} + r_{i-1}r_kr_j\}, & i < j. \end{cases}.$$

其中第一条是接下来代码中不可或缺的的 base case。从递推式中很容易看出这一问题的最优子结构，实际上就是对于 M_{ij}，其计算的最后一定是 M_{ik} 和 $M_{k+1,j}$ 相乘，因此所有分解的情况构成了需要比较的全体情况，然后必须取出每个分解的最优情况（否则最后可能得到的不是最优解），然后取出所有分解的最优情况中最好的就是大问题的最优解。因此 M_{ij} 的最优乘法顺序依赖于子问题 M_{ik} 和 $M_{k+1,j}$ 的最优值，这就是这一问题的最优子结构。

和之前的问题一致，这一递推式很适合于递归，但冗余计算很多，因此自底向上记忆化。这里的记忆化和之前也不太一样，因为思考递推式并不是以前那样按输入顺序每个输入是否是子问题解的一部分，那时的代码可以同理可以做一个遍历，根据输入顺序逐个判断当前输入是否应当是某个子问题的解的一部分。读者可以思考，如果按顺序遍历我们的代码一定会出现混乱，因为你会发现你还要依赖很多你当前并没算出来的情况。但是只要你想到这一点就可以：如果短的矩阵链最优解被构造，那么可以基于此计算更长的矩阵链的最优解，即应当按照矩阵链的长度从小到大的顺序计算最优解，即最外层循环应当是按 M_{ij} 中 $i-j$ 的大小递增的顺序。这样 base case 是矩阵链长度为 1（$i=j$）的全 0，然后是长度为 2 （$j-i=1$）的情况，实际上也只有一种乘法方式，所以也是 trivial 的，接下来长度为 3 的情况完全可以基于长度为 1 和 2 的情况得到，然后逐步迭代到最后的长度为 n 的情况。这样就得到了一个非常简单的动态规划算法，其时间复杂度是 $O(n^3)$，这里同样建议读者自己写一遍代码（可以参考 PPT 第六页，体会 base case 开始自底向上构建解的想法），可以加深理解。

8.4.3 解的重构

这一问题重构解的方法如果用和前面的问题一样简单直接的挖掘方法显然会比较耗时间，因为每次要比较很多种可能的情况才能重构出解，所以需要做一些优化，优化的方法也非常非常自然，实际上关键

就是用一个二维数组记住每个子问题

$$m_{ij} = \min_{i \leqslant k < j} \{m_{ik} + m_{k+1,j} + r_{i-1} r_k r_j\}$$

对应的最优的分点 k，然后当算出总问题 m_{1n} 的最优解及其对应的最优分点后，再找左右两半子问题的最优分点，以此类推，用中序遍历的思想将分点输出即可，只需线性时间。

【思考:】（钢条切割问题）Serling 公司购买长钢条，将其切割为短钢条出售。切割工序本身没有成本支出。公司管理层希望知道最佳的切割方案。假定已知 Serling 公司出售一段长度为 i 英寸的钢条的价格为 $p_i (i = 1, 2, \cdots, n)$，单位为美元）。钢条的长度均为整英寸。下图给出了一个价格表的样例。

长度i	1	2	3	4	5	6	7	8	9	10
价格p_i	1	5	8	9	10	17	17	20	24	30

钢条切割问题是这样的：给定一段长度为 n 英寸的钢条和一个价格表 $p_i (i = 1, 2, \cdots, n)$，求切割钢条方案，使得销售收益 r_n 最大。注意，如果长度为 n 英寸的钢条的价格 p_n 足够大，最优解可能就是完全不需要切割。

【思考:】（完全平方数的和）给你一个整数 n，返回和为 n 的完全平方数的最少数量。例如 $n = 13$，则 n 至少需要写成两个完全平方数相加的形式，即 $n = 4 + 9$。

8.5 最长公共子序列问题

8.5.1 问题描述

8.5.2 序列对齐问题

8.5.3 最优子结构：字符串问题的特点

8.5.4 解的重构

【思考:】（回文字符串）如果字符串的反序与原始字符串相同，则该字符串称为回文字符串。现在给你一个字符串 `s`，找到 `s` 中最长的回文子串。

【思考:】RNA

8.6 最优二叉搜索树

8.6.1 问题描述

这一问题给考虑下列输人：给定一列单词 $w_1, w_2, \cdots, w_n$（它们的顺序已经按字典序排好）和它们出现的固定的概率 $p_1, p_2, \cdots, p_n$。问题是要以一种方法在一棵二叉查找树中安放这些单词使得总的期望存取时间最小。在一棵二叉查找树中，访问深度 d 处的一个元素所需要的比较次数是 $d+1$，因此如果 w_i 被放在深度 d_i 上，就要将 $\sum_{i=1}^{N} p_i(1+d_i)$ 极小化。

8.6.2 算法描述

这一问题最直接的想法便是贪心或者使用平衡树，但 PPT 第 7 页中的例子告诉我们它们都是不一定可行的。一种也很自然的想法是分治法，可以将问题分为两个子问题来解决，然而分治法有一个很大的问题：这两半的子问题应该如何划分？如果提前知道最优的根结点是哪个单词，那很容易进行问题的划分，然而可惜的是我们没有预知未来的能力。但从这一特点能回忆到曾经熟悉的问题：在矩阵链乘法中，需要知道第一个分点在哪里才能构建后续的解，加权独立集合问题中，需要知道最后一个点是否在最优解中才能推出整个解。这一问题也是一样的，需要知道最优的根结点在哪里才能划分问题，然后可以得到两个子问题，然后我们可以得到最优解。并且知道最初给定的输入已经按字典序排好，所以如果知道根结点的最优解为 w_k，那么在整个问题的最优解中，其左子树必定是 $w_1, \cdots, w_{k-1}$ 对应的最优二叉搜索树，右子树是 $w_{k+1}, \cdots, w_n$ 对应的最优二叉搜索树。因此可以得到一个递推式（符号含义见 PPT 第 8 页）：

$$c_{ij} = \sum_{k=i}^{j} p_k + \min_{i \leqslant k \leqslant j} \{c_{i,k-1} + c_{k+1,j}\},\ i \leqslant j.$$

注意第一项的意义在于，因为根结点的存在，导致所有 i 到 j 的单词都加深一层，是两个子问题结合起来增加的代价。第二项则是前面讨论的左右子树子问题的情况。此时的 base case 没有写在递推式中，但读者仔细观察应当不难看出是 $j = i-1$ 时一定为 0（记住 base case 就是递推式的最小出发点，是最 trivial 的情况），当然也可以把 $i = j$ 的情况，即只有一个点的情况作为 base case，然后对上面的递推式略作修改。基于此，可以得到如下伪代码：

图中为代码的复杂度显然是立方级别的：尽管只写了两个循环，但显然最后取 min 的步骤也是要遍历的——这与前面的矩阵乘法问题完全一致。总结而言，这一问题的最优子结构也是不言自明的，即最优解依赖于每个分割得到的子树的的最优解。这一问题的解的重构也是非常简单的，只需记住每个子问题的最优分点即可，然后回溯建树即可，和矩阵乘法顺序问题也是一致的，只需线性时间。

或许这一问题从很多方面看来都与矩阵乘法顺序问题一致，但事实上可以为这一问题构建一个平方级别的算法，读者可以跟随下一思考题来实现这一解法：

```
// subproblems (i indexed from 1, j from 0)
A := (n+1) × (n+1) two-dimensional array
// base cases (i = j + 1)
for i = 1 to n + 1 do
    A[i][i − 1] := 0
// systematically solve all subproblems (i ≤ j)
for s = 0 to n − 1 do          // s=subproblem size-1
    for i = 1 to n − s do      // i + s plays role of j
        // use recurrence from Corollary 17.5
        A[i][i + s] :=
            ∑_{k=i}^{i+s} p_k + min_{r=i}^{i+s} {A[i][r−1] + A[r+1][i+s]}
                                                 (Case r)
return A[1][n] // solution to largest subproblem
```

【思考:】设

$$c_{ij} = \begin{cases} 0, & i = j; \\ w_{ij} + \min\limits_{i<k\leqslant j}\{c_{i,k-1} + c_{kj}\}, & i < j. \end{cases}$$

设 W 满足四边形不等式，即对所有的 $i \leqslant i' \leqslant j \leqslant j'$，有

$$w_{ij} + w_{i'j'} \leqslant w_{ij'} + w_{i'j}.$$

进一步假设 W 是单调的：如果 $i \leqslant i'$ 且 $j \leqslant j'$，则 $w_{ij} \leqslant w_{i'j'}$。

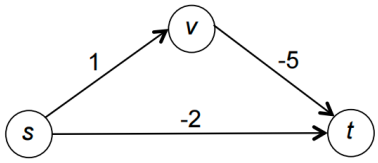
1. 证明：C 满足四边形不等式；
2. 令 R_{ij} 是使 $C_{i,k-1} + C_{kj}$ 最小值的最大的 k（即出现平局的情况选择最大的 k），证明：

$$R_{ij} \leqslant R_{i,j+1} \leqslant R_{i+1,j+1};$$

3. 证明：R 沿着每一行和列是非减的；
4. 证明：C 中所有项的计算可以在 $O(n^2)$ 时间内完成；
5. 用上述结论描述一个最优二叉搜索树的 $O(n^2)$ 时间算法。

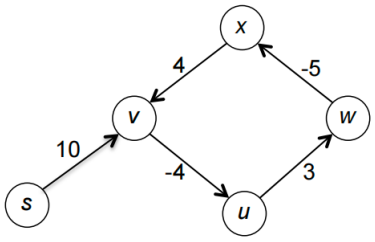
8.7 最短路问题再讨论

相信各位在 FDS 中已经对经典的单源（即只考虑一个顶点出发到任意其它顶点的距离）无负边最短路问题算法——Dijkstra 算法非常熟悉，但这一问题对于带负边的问题可能给不出正确的解，如下面的例子：



如果用 Dijkstra 算法求解这一问题，会得到最短路径长为 -2，但事实上显然是 -4，为什么 Dijkstra 算法在有负边时会失效则要回顾这一贪心算法的正确性证明，具体的读者可以回顾离散或 FDS 中学习过的证明，总体原因是正确性证明的确是依赖于边长非负这一条件才能实现的。

负边带来的更痛苦的事情是可能出现负环：



这样即使有一个能求出带负边的最短路的算法，但因为事实上的最短路应当在负环中无限循环直至负无穷，所以算法也会失效。当然有一种面对负环的解决方案，即计算只考虑无环路最短路径，但在负环存在时这一问题是 NP 困难的（之后的章节会介绍，总之是一类非常难解的问题）。看起来我们无路可走了，但还可以考虑另一种修订版本，即希望这一算法要么能计算出正确的最短路径长度，要么能正确地判断出输入图中包含了负环（只需要判断即可）。这就带来了下面的 Bellman-Ford 算法。

【注：本节的讲解很多内容参考拉夫加登的《算法详解》，感兴趣的同学可以读一下原书】

8.7.1 Bellman-Ford 算法

事实上大家在 FDS 中就已经见过这一算法（可以回过头去翻一翻 PPT），只是当时我相信老师并不会说这是一个动态规划算法。那么现在用动态规划的思想来重新设计和理解这一算法。

既然是动态规划，就需要找到最优子结构。但是对于一般的图而言，因为具有线性结构，所以定义子结构并不是很容易的事情（所以加权独立集合考虑的是路径图，这样点就有线性的先后顺序，子结构很好定义）。于是需要考虑一些特别的子问题、子结构定义方式。在 Bellman-Ford 算法中，子问题被巧妙地定义为：对于每个顶点 v，考虑从源点 s 到 v 的所有路径中，最多经过 k 条边的路径的最短路径长度。为什么这一子问题是合理的呢？来看递推关系的推导就知道了，即能否用最多经过 $k-1$ 条边的路径的最短路径长度推到最多经过 k 条边的路径的最短路径（记为 P）长度。事实上完全可以，并且显然地，递推依赖于如下两种情况取最小值：

1. P 还是只有 $k-1$ 条甚至更少的边，那么直接继承子问题的解即可；

2. P 有 k 条边，设 P' 为 P 中前 $k-1$ 条边，并且 (w,v) 是其最后一次的跳跃，那么因为 P 是 s 到 v 的 k 条边的最短路径，所以 P' 也是 s 到 w 的 $k-1$ 条边的最短路径，否则能取到一个更短的 P'' 从 s 到 w，那 P'' 接上 (w,v) 是比 P 更短的路径，显然矛盾。因此递推时只需要考虑所有能一步到达 v 的顶点 $w_1,\cdots,w_m$，然后取 s 到这些 $w_i(i=1,\ldots,m)$ 的最短路长度，再加上 (w_i,v) 的长度得到 m 条 s 到 v 的路径长度，然后取出这些长度的最小值即可。

记 $D^k[s][v]$ 为从 s 到 v 的最多经过 k 条边的最短路径长度，整个图为 $G=(V,E)$，如果 a 到 b 有一条边，那么这条边的长度记为 l_{ab}，那么根据上面的讨论有递推式：

$$D^k[s][v]=\min\begin{cases}D^{k-1}[s][v], & \text{case 1;}\\ \min\limits_{(w,v)\in E}\{D^{k-1}[s][w]+l_{wv}\}, & \text{case 2.}\end{cases}$$

以上递推式的正确性来源于考虑到了所有长度小于等于 $k+1$ 的 s 到 v 的路径的情况，最优子结构体现在长度至多为 k 的路径的最短路径长度一定是依赖于长度至多为 $k-1$ 的路径的最短路径长度的。这是最短路问题的一个非常重要的性质（即最短路径的子路径也是对应两点间的最短路径），通过定义如上的子问题 $D^k[s][v]$，恰好利用了这一重要性质，找到了合适的最优子结构，十分精妙。

似乎我们的工作已经做完了？先别高兴太早，还需要检测负环！可以看上面的递推式，事实上现在还没有讨论 k 上升到多少的时候停止递推，如果有负环，事实上上面的递推式可以无限递推下去，这显然不是我们想要的。因此现在考虑，上面的算法最多到哪一步停止，停止时是否可以判断出有无负环。事实上有下一重要引理：

引理 8.1 *设起点为 s，如果对于某个 $k\geqslant 0$，对于所有目标顶点 v 都有 $D^k[s][v]=D^{k+1}[s][v]$，那么*

1. *对于每个 $i\geqslant k$ 和任意的目标顶点 v，$D^i[s][v]=D^k[s][v]$ 都成立；*
2. *对于每个目标顶点 v，$D^k[s][v]$ 就是 s 到 v 的最短路径长度。*

证明： 引理第一条，只需看清前面的递推式即可。对于所有目标顶点 v 都有 $D^k[s][v]=D^{k+1}[s][v]$，那么当考虑 $D^{k+2}[s][v]$ 时，递推式面对的输入和 $D^{k+1}[s][v]$ 完全一致！所以显然 $D^{k+2}[s][v]=D^{k+1}[s][v]$，这样递推下去，就得到了第一条的结论。

对于第二条，用反证法非常显然，如 $D^k[s][v]$ 不是 s 到 v 的最短路径长度，但按定义这是 k 步之内的最短路，那么只能是存在一条更长的路径比 $D^k[s][v]$ 更短。但根据引理第一条，任意长度大于等于 k 的 s 到 v 的路径的最短路径长度都是 $D^k[s][v]$，显然矛盾。 ■

基于这一引理可以得到如下的判断是否包含负环的定理：

定理 8.2 *输入图 G 不包含负环当且仅当 $D^n[s][v]=D^{n+1}[s][v]$ 对所有的目标顶点 v 都成立。其中 n 是图 G 的顶点数。*

证明: 若 $D^n[s][v] = D^{n+1}[s][v]$ 对所有的目标顶点 v 都成立，根据引理知道对于每个 $i \geqslant n$ 和任意的目标顶点 v，$D^i[s][v] = D^n[s][v]$ 都成立，如果存在负环，那么显然 i 增大的时候某些顶点的最短路径长度会减小，矛盾!

反之，如果 G 不包含负环，那么从 s 出发到一个不同的点 v，最多也就是把所有顶点都遍历了一遍，所以最短路径最长也只能是 $n-1$（否则更长就有环，但此时环都是非负的）。换句话说，把边预算从 $n-1$ 上升到 n 对所有顶点的结果一定是没有影响的。 ■

接下来是代码实现，因为是单源最短路，所以 s 无需在数组索引中，数组形式为 $A[i][v]$，表示从 s 到 v 的边数最多为 i 的最短路长度这一子问题。下面设置 base case，很显然此时 base case 是 $i = 0$ 的情况，此时只有 s 到 s 最短路就是 0，其余都设置为无穷大，然后就可以用递推式进行计算了。如果到了某一步发现对于所有的 v 都有 $A[i][v] = A[i-1][v]$，那么根据引理第二条可知所有的最短路长度就出来了，如果知道 n 步后都无法满足这一点，根据定理则存在负环（因此需要维护一个 flag 表明是否达到稳定状态）。如果需要重构最短路径，可以存下每次取 min 的时候距离最小的那个点，最后做个 traceback 即可，并且最短路径的子路径也是最短路径，所以这一重建过程遍历一次就可以得到所有顶点的最短路径了。

下面分析一下 Bellman-Ford 算法的时间复杂度，设输入图的顶点数为 n，边数为 m，可以大致知道有两层循环，其中第一层 i 最多需要遍历 n 个值，第二层 n 个顶点也需要全部遍历，然后第二层循环内需要遍历对应的顶点 v 的所有 (w, v)，于是实际上整个第二层循环计算、比较了每个顶点的入度之和这个层级的量，而所有顶点入度之和在有向图中数量就是边数，所以总的复杂度就是 $O(mn)$。

Bellman-Ford 算法应用非常广泛，事实上早期 Internet 路由协议 RIP 就是使用的这一算法。为什么使用 Bellman-Ford 算法呢? 因为如果要考虑一个新的互联网中节点 w 和路由器的最短路，那么只需要知道和 w 直接相连的顶点的最短路就可以直接利用递推式推导即可，这就意味着 Bellman-Ford 算法即使对于像 Internet 这样规模的网络图也是可以实现的，每台计算机只需要与直接邻居通信、只执行本地计算即可得知需要的一切信息，而 Dijkstra 算法每台主机还要计算一个相当大的 Internet 规模的网络图的最短路，显然是不合适的。

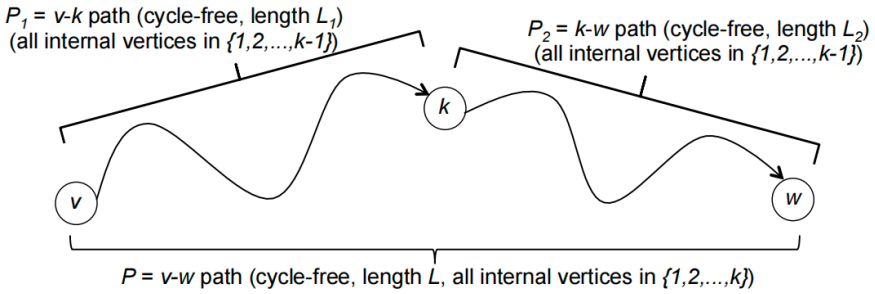
8.7.2 Floyd-Warshall 算法

在考虑了单源、找出最短路或负环的算法后，最后一种推广是考虑所有点对之间的最短路，也就不是单源，每个点都可能是源，并且有负环也需要指出。最简单的想法就是机械地调用 n 次 Bellman-Ford 算法，但这样复杂度就是 $O(n^2m)$，在稠密图上是四次方级别的，有点不可接受，所以考虑尝试新的动态规划思想来重新解决这一问题。

Floyd-Warshall 算法定义的子问题更加有趣（图上的算法总是让人觉得很巧妙）。首先需要给图上的每个顶点一个编号：$1, 2, \cdots, n$，保持记号 $D^k[i][j]$，但现在它的含义变成了：从点 i 出发到 j 的，只使用 $1, 2, \cdots, k$ 作为内部顶点的（内部顶点就是除起点和终点外的其它点，即从 i 到 j 中间只能经过这些点），不包含环的最短路径长度。

这一子问题的定义显然导致了以下递推的想法：当我们要从 $D^{k-1}[i][j]$ 推到 $D^k[i][j]$ 对应的路径 P 时，显然又是老调重弹的分类方法：要么 k 在 P 中，要么不在：

1. 若 k 不是 P 的内部顶点，那么最好的解就是继承 $D^{k-1}[i][j]$；
2. 若 k 是 P 的内部顶点，那么可以把 P 分成两段：$i \to k$ 和 $k \to j$，如图所示。即 P 的总长度就等于这两段的长度之和，即 $D^k[i][j] = D^{k-1}[i][k] + D^{k-1}[k][j]$。



有一个值得考虑的问题是，有没有可能这两段路径纠缠起来形成一个环？如果形成负环，无需关心这一情况，因为最终会检测出来；如果是正环，那么能画出一个不经过 k 的更短的路径，所以这种情况也不会在最终的最短路径中出现。因此下述递推式是合理的：

$$D^k[i][j] = \min \begin{cases} D^{k-1}[i][j], & \text{case 1;} \\ D^{k-1}[i][k] + D^{k-1}[k][j], & \text{case 2.} \end{cases}$$

接下来的问题和 Bellman-Ford 算法一样，需要考虑如何判断是否存在负环：

定理 8.3 *图 G 包含负环当且仅当存在一个顶点 v，使得 $D^n[v][v] < 0$。*

证明: 若图 G 不包含负环，根据前面的推理，Floyd-Warshall 算法一定会返回正确的最短路径长度，所以 $D^n[v][v] = 0$。若包含负环，那么根据负环的定义，存在一个顶点 v，使得从 v 出发回到自己的最短路径长度小于 0。所以在递推过程中一定会出现 $D^n[v][v] = D^{k-1}[v][k] + D^{k-1}[k][v] < 0$ 的情况，这时就会更新 v 到 v 的最小值，且之后这个值只能越来越小（因为一直在取 min），所以 $D^n[v][v] < 0$。 ■

基于以上讨论可以写出 Floyd-Warshall 算法的伪代码。最 naive 的方式就是用三维数组 $D[k][i][j]$ 表达 $D^k[i][j]$ 的含义，那么首先要讨论 base case，因为递推是 k 逐渐增大的过程，所以 base case 就是 $k = 0$ 的情况，接下来用三层循环构建全部的解即可，因此时间复杂度是 $O(n^3)$，相比于调用 n 次 Bellman-Ford 算法更好。PPT 第 11 页给出了一种降维的方式：其实 k 没有必要出现在下标，只需要在增大 k 的过程中更新 i 和 j 之间的最短路长度即可，因为随着 k 增大最短路一定是递减的，对所有的 k 维护一个值足矣（但 Bellman-Ford 算法不能这么降维，因为判断负边需要二维子问题）。至于重建最短路径，在循环的时候不断更新每对顶点 i 和 j 形成最短路径的最优的 k，因为这样最后一次 k

的更新就对应于 i 和 j 之间最短路的必经之路。然后可以回溯在线性时间内重建给定的两点之间的最短路径。

事实上关于最短路的问题的研究时至今日仍在进行，STOC'23，FOCS'24 和 STOC'25 仍然有相关的文章发表，这一领域的研究仍然是非常活跃的。事实上，所有顶点对的最短路径是否存在 $O(n^{2.99})$ 的算法仍然是一个很著名的开放性问题，留待后人解决。

8.8 总结

8.8.1 动态规划、贪心算法、运筹学

我想在看过这么多的例子之后，可以尝试总结一下动态规划解决问题的范式。实际上，使用动态规划解决的问题一般都有一个特点，即是希望求解在满足一定条件下某个问题的最优解，事实上这是数学规划的最经典的问题，也是运筹学的出发点。例如 0-1 背包问题，事实上可以建模成如下混合整数规划问题：

$$\begin{aligned} &\max \sum_{i=1}^{n} v_i x_i, \\ &\text{s.t.} \sum_{i=1}^{n} w_i x_i \leqslant W, \\ &x_i \in \{0,1\}. \end{aligned}$$

当然这里并非要讨论运筹学，毕竟这是一个更广且更深的话题，只是本讲和下一讲贪心算法中遇到的问题大都是运筹学问题，很多问题在运筹学中都有数学规划的 "通法" 解决，但这两章中遇到的问题在结构上具有一定的特殊性，使得我们能用更漂亮的办法解决。

8.8.2 动态规划与分治法

下一讲将讨论动态规划和贪心算法能解决的问题的结构上有什么区别，现在回到本讲开头遇到的一个情况，即爬楼梯问题不适合分治算法，因此这引出了一个分治法和动态规划方法比较的问题：两者似乎都是组合子问题的解，那么更具体的区别在哪里呢？

在拉夫加登的《算法详解》中，他做出了如下总结，诸位可以对照之前学过的例子（分治的经典就是归并排序）体会：

1. 在典型的分治算法中，每个递归调用只提供一种方法把输入划分为更小的子问题。在动态规划中，如果采用递归，每个递归调用的选择就比较开放，可以对更小子问题的多种定义方法进行考虑并从中选择最优的一种。

2. 由于动态规划算法的每个递归调用会对更小子问题的多个选择进行试验，因此子问题一般会在不同的递归调用中重复出现（或者说子问题之间有 overlapping）。因此，将子问题的解决方案进行缓存是一种可以无脑进行的优化。在大多数分治算法中，所有的子问题都是不同的，因此没有必要特它们的解决方案进行缓存。

3. 分治算法的大多数经典应用把任务的多项式杂度的简单算法替换为更快的分治版本。动态规划得力的应用是把需要指数级时间复杂度的任务（例如分举搜索）优化为多项式时间级的复杂度，因此是一种基于子问题的聪明的枚举法。

4. 相对来说，分治算法的子问题大小一般不会超过输入的某个比例（例如 50%）。动态规划的子问题只要小于输入就没有问题，只要满足正确性。

5. 分治算法可以看成动态规划的一种特殊情况，每个递归调用选择一个固定的子问题集合以递归的方式解决。作为一种更为复杂的算法范例，动态规划相比分治算法适用于范围更广的问题，但它的技术要求也更高（至少需要足够的实践）。

面临一个新问题时，应该选择哪种算法范例呢？如果能够发现一种分治算法解决方案，就应毫不犹豫地使用它。如果分治算法无能为力，尤其当失败的原因是组合步骤总需要大量从头开始的计算时，就可以尝试使用动态规划（例如之前用分治法分析过爬楼梯、加权独立集合、最优二叉搜索树等问题，它们子问题的划分要么会纠缠，要么会不知如何划分）。

8.8.3 最后的话

或许现在的讨论总是强调回溯法的效率低下以及分治算法的不可行性，这容易使得我们认为动态规划是一种 “万全之法”。当然事实上显然不是这样的，动态规划对于很多问题而言也是不适用的。PPT 第 17 页给出了两种不适用的情况，第一是没有最优子结构，例如出现了 History-dependency，那么你如果从 base case 出发迭代，当迭代到打叉的点时，会因为前面的最优路径经过了冲突的点而舍弃这条路，但你可能因此舍弃了一个很好的解。其二是子问题如果没有 overlapping，那么递推计算可能是不完备的，这时用分治算法可能会更好。

总结而言，动态规划重点在于两个步骤：其一是寻找问题的最优子结构，即原问题的最优解会依赖于一些子问题的最优解，其二是根据递推式的特点，选取合适的循环变量自底向上记忆化逐步构建出原问题的解。我想这两步都多少有些 trick 在：不同问题可能有不同的最优子结构，可能一维就能解决，可能二维甚至更高维才能解决，需要解决的子问题个数可能是 2 个、3 个或者很多个。对于一般的问题，都可以通过思维实验的方式思考解的构成：我首先需要判断什么，才能构建出接下来的解。当然这需要偶尔的灵光一闪以及一定的经验积累，但我相信同学们会在一些问题中感受到其中的乐趣。除此之外，代码实现也是不可缺少的进一步理解动态规划的方式，这是针对于第二步记忆化而言的，这里 base case 的选取，以及循环变量的选取（可能就是顺序遍历，也可能是子问题的长度等）都是在代码实现中不可或缺的。

总而言之，除了最后最短路这种图上的问题比较特别外，其它很多问题使用动态规划并不是很困难的事情。或许使得动态规划是一个让人觉得有点费解的方法的很大一部分理由其实是因为这个不明所以却在当年又迫不得已的名字！倘若动态规划改名递推法，我想从一开始的讨论诸位就会觉得上面的方法并没有多么特殊，对吗？这种方法会和回溯、分治、贪心一样自然，甚至比回溯和贪心更加有章可循。