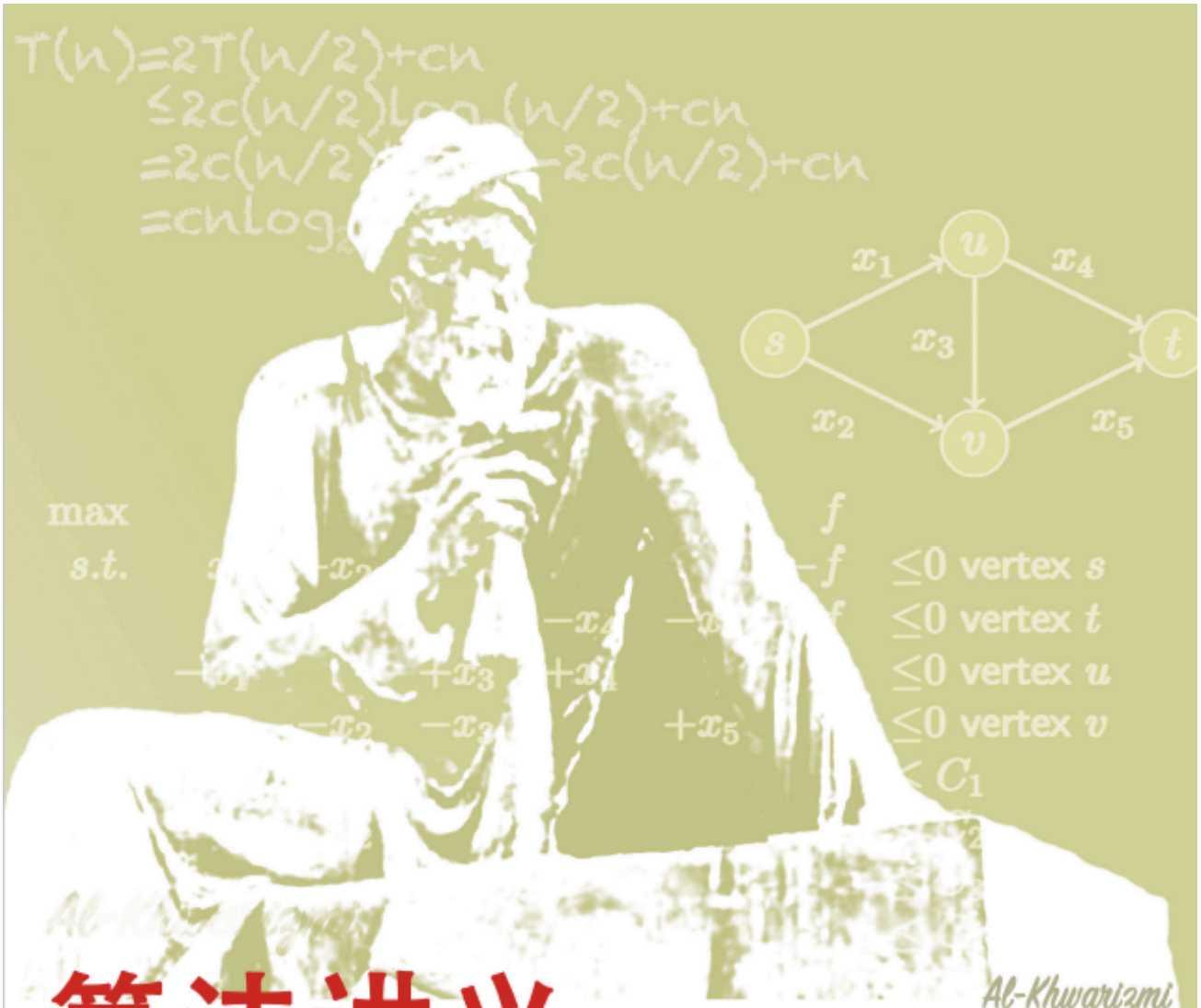
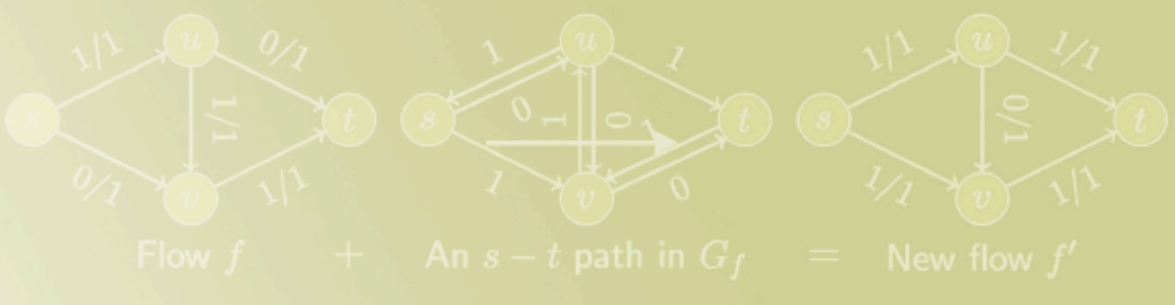


算法讲义

关于问题求解方法的十八讲

卜东波 张家琳 编著

LECTURES ON ALGORITHMS



高等教育出版社

献给我的父母—辛勤教书育人四十载的卜老师和蔡老师。
献给我的太太 Tine 和女儿 Martine，感谢你们的支持！

— 卜东波

献给我的先生杜老师和亲爱的儿子零零！

— 张家琳

序（一）

李国杰老师序

序（二）

白老师序

前　言

公元 9 世纪，波斯数学家 Muhammad ibn Musa al-Khwarizmi 写了一本名为 *The Compendious Book on Calculation by Completion and Balancing* 的书。

这本书记载了贸易、测量、遗产分配等方面的一些实际问题，以及从这些实际问题出发、抽象建模而成的数学问题—一些线性方程组和二次方程。更重要的是，这本书还介绍了求解上述方程的步骤：这些步骤描述清晰；无论是人还是机器，只需按步骤机械式地操作即可求解方程。这样的问题求解步骤被称作 “Algorithm”，中文译作*算法*。Algorithm 一词来源于 Al-Khawarizmi 名字的拉丁文译法，以表达对他的敬意。

无独有偶。中国古代的数学著作《九章算术》以及《九章算术注释》也是记载了许多实际问题以及相应的求解方法，甚至还提出“算”这一概念来表示“运算次数”，以比较不同求解方法的效率。中国古代数学历来重视解决来自现实生活的实际问题，而不像古希腊数学那样强调定理的证明。吴文俊先生称中国传统数学的特点是*高度的机械化和算法化*，并认为这一特点是和现代计算机科学密切相通的。

当面对一个数学问题时，从何处入手进行求解值得深入思考。数学家 G. Polya 如此描述他学生时代的困惑:“*是的，这个解答看来是可行的，也是正确的，但怎样才能想到这样一个解答呢?*”。不仅要理解解答本身，而且要理解如何做出这个解答，并尽力向别人解释清楚思考过程—这些动机最终驱使 G. Polya 写出了关于数学思维的名著《怎样解题》。

与上述情形非常类似，当面对一个算法问题时，从何处入手进行求解也是值得深入思考的事情。从另一个角度来讲，理解一个算法如何工作并不算是太困难的任务，但是要弄明白算法是怎样设计出来的，却是很困难的。当我们看到别人设计出的精妙算法时，在钦佩之余，往往也会有与 G. Polya 类似的困惑:“*这么精妙的算法是怎样设计出来的? 我为什么没想到这个算法呢?*”正是对这些困惑的思考促使我们撰写了这本讲义。

在这本讲义里，我们不打算只是简单地罗列现有的算法，而是试图揭示算法背后

的设计过程。为此目的，我们一方面尽力采用*历史途径法*描述算法设计者的思考过程，即：尽可能地从原著、访谈或者传记中还原出设计者当初的知识储备、遇到的困难、做过的尝试，以及最终克服这个困难的诀窍。在此方面，R. Bellman 提出动态规划算法的过程是一个绝佳示例。另一方面，我们采用*观察-尝试-迭代改进*的方式来实际设计算法，即：*首先观察问题的结构，然后设计一个初步的算法，接下来观察算法的行为，进而迭代改进算法*。我们强调*观察问题的结构*，强调*基于问题的结构进行算法设计*，强调*基于问题的结构和算法的行为迭代改进算法*—求解问题的过程不应当只是逐个尝试各个算法技术，也不是纯粹依赖于灵感，而是应该依赖于对问题结构的认识；我们对问题结构认识得越深入，越有助于求解算法的设计。

我们按照问题结构组织本篇讲义，将之分作四个大的部分：

(1) 当我们观察到待求解的问题能够归约成规模较小的子问题时，就可以尝试*基于归约原理的算法设计*。这一类算法组成讲义的第一部分，包括分而治之算法（第 2 章）、动态规划算法（第 3, 4 章）和贪心算法（第 5 章）。

(2) 当我们观察到问题不太容易归约成规模较小的子问题、但是能够观察到可行解之间的变换关系时，就可以尝试*逐步改进*类算法设计策略。这一类算法组成讲义的第二部分，包括线性规划及对偶理论（第 6, 7 章）、网络流算法及其应用（第 8, 9 章）。

(3) 当我们观察到解可以写成 $X = [x_1, x_2, \cdots, x_n], x_i = 0/1$ 的形式时（或者更一般地，解可以一步一步地逐渐构建出来时），就可以尝试*智能枚举*类算法设计策略—所谓“智能”，是指采用下界等信息对枚举树进行剪枝操作。这一类算法组成讲义的第三部分，包括分支限界算法（第 1 章）、回溯算法（第 13 章）；有些内容也会穿插在贪心算法中进行讲述（第 5 章）。

(4) 难度是问题的本质属性。当我们观察到问题之间的归约关系、并进而证明了问题是 NP-Hard 之后（第 10 章），就意味着只有放松要求才能设计出高效的算法。这一类算法组成讲义的第四部分，包括只要求能够得到近似解的近似算法（第 11 章）、允许算法中存在随机化行为的随机算法（第 12 章），以及实用的启发式算法（第 13 章）。我们把问题求解思路的详细描述放在第 1 章，作为整个讲义的总纲。

我们在中国科学院大学的研究生班上多次讲授《算法设计与分析》课程，通常需要讲授一个学期、共十八次课。这本讲义是教学过程的自然结晶；我们加上副标题“关

于问题求解方法的十八讲”，是想表达这本书和教学之间的紧密关系。在写作这本讲义时，我们想象中的读者是计算机专业的研究生或者高年级本科生；我们假定读者已经修习过《数据结构》和《计算机程序设计》，并熟练掌握至少一门计算机语言。

十年的教学统计表明：在修习算法课的中科院计算所博士班同学中，每年有大约 10% 的同学利用上课所学的知识，解决了自己研究工作中碰到的问题，并发表了有创见的学术论文。比如韦祎、王超同学使用线性规划设计了一种具有广泛适用范围的*逆向* Monte Carlo *采样*（Reverse sampling）技术，能够有效改善能量函数的平滑度，进而提高正向 Monte Carlo 搜索的效率和性能；杨飞雕同学使用神经网络表示动态规划中的递归函数，克服了 Held-Karp 算法指数级空间开销的缺陷，进而设计出了求解旅行商问题的全新算法。博士生大多是“带着问题来上课”，触类旁通，所在多有。

“科教融合”是国科大的教学特色之一：教学内容中有最新的科研成果；听讲者和讲授者在教学过程中有所感悟，并据此改进自己的研究。我们把同学们和我们自己的一些研究进展添加到讲义的正文之中，一方面反映了算法的最新发展，是与时俱进；另一方面也是对讲授者的反哺，是教学相长。除此之外，一些习题也是摘录自国科大博士生和硕士生的学位论文；因此可以说这本讲义是一本充满浓厚国科大色彩的书。

要想把复杂的算法讲得清楚明白，是需要一些技巧的。我在滑铁卢大学听过 Timothy Chan 教授讲授算法课，从中领会到了授课的技巧和方法；而在写作方面，我则偏爱于《费恩曼物理学讲义》的风格：要揭示复杂事物背后的直观思想，自然的笔触或许更有助于描述本质性的东西。这份讲义是采用这种写法的一次尝试—由中国科学院大学的同学们依据课堂录音整理成文；我们在此对乔扬、申世伟、邵益文、黄斌、闾泽军、乔晶、袁伟超、李飞、孔鲁鹏、吴步娇、张敬玮、罗纯龙、曹晓然、梁志鹏、江涛等记录者致以谢忱。

有很多同事、同学担任了《算法设计与分析》课程的助教，对这本讲义有直接的贡献。我在此对林宇、袁雄鹰、韦祎、邵明富、王超、张海仓、黄春林、李锦、黄琴、凌彬、王耀军、许情、张任玉、巩海娥、杨飞、王冰、高枫、李艳博、朱建伟、魏国正、鞠富松等同学表示诚挚的感谢。令我痛心的是，在本书尚待付印之时，林宇不幸英年早逝—林宇是我的第一位研究生，后在澳大利亚国立大学当教授，是基因组序列拼接算法、进化树构建领域的国际著名学者；我讲授算法课的脉络和很多细节，都是和他仔细讨论过的；在整数线性规划那一章，更是直接引述了他的成果。我想这本讲义的出版，也算是对林宇的一个纪念吧。

在讲授这门课程时，我们制作、使用了电子课件（包括幻灯片、算法演示，以及 On-

line Judge 编程题目);这些课件可以从 https://deltadbu.github.io/ UCAS_algorithm_course/下载。诸位读者在使用讲义和课件时所发现的错误,敬请发送至 errata.lnoa@gmail.com。

这本讲义的写作得到了李国杰、白硕、徐志伟等老师的鼓励。诸位师长奖掖后进;拳拳之心,殷殷之情,使人难生懈怠之意。我们谨以这本小书回馈他们的希冀。

如何在讲清楚直观思想的同时不丢失严谨性,如何在厘清脉络的同时又不限制创造性,是讲课和写作中最难把握的,也是最让我们困惑的。在这两点上,我们始终惴惴不安,只能静候读者诸君的反馈和指正。

卜东波

2018 年于中科院计算所、中国科学院大学

目录

序（一） **iii**

序（二） **iv**

前言 **i**

第一章 绪论 **1**

1.1 什么是算法问题? 1

1.2 怎样进行算法设计? 3

1.2.1 观察问题内在结构的途径 4

1.2.2 “分而治之”算法设计过程简介 6

1.2.3 “逐步改进”算法设计过程简介 9

1.2.4 智能枚举算法设计过程简介 12

1.3 算法的复杂度 16

1.3.1 时间复杂度与空间复杂度 21

1.3.2 大 O 记号 22

第二章 “分而治之”算法 **29**

2.1 引言 29

2.2 排序问题：对数组的归约 30

2.2.1 依据元素下标拆分数组：插入排序与归并排序 30

2.2.2 “分而治之”算法时间复杂度分析及 Master 定理 37

2.2.3 依据元素的值拆分数组：QuickSort 算法 39

2.3 一个密切相关的问题：数组中的逆序对计数 48

2.4 选择问题：对数组的归约 51

2.4.1 选择“分组中位数的中位数”作为中心元 52
2.4.2 依据随机样本的统计量确定中心元 55
2.4.3 采用随机选择的一个元素作中心元 57
2.5 整数乘法：对一对数组的归约 60
2.6 快速傅里叶变换：对数组的归约 67
2.7 矩阵乘法：对二维数组的归约 78
2.8 平面上最近点对寻找问题：对点集的归约 80
2.9 小结 84

第三章 动态规划算法 89

3.1 引言 89
3.2 矩阵序列的链式相乘问题：对序列的归约 90
3.2.1 算法设计：一个比较慢的初始版本 92
3.2.2 避免子问题的重复计算：“以存代算” 96
3.2.3 算法运行过程示例 98
3.2.4 时间复杂度分析 101
3.2.5 一些讨论 101
3.3 动态规划与多步决策过程 103
3.4 0-1 背包问题：对集合的归约 104
3.4.1 算法设计：一个比较慢的初始版本 105
3.4.2 算法设计：改进后的版本 107
3.4.3 算法运行过程示例 109
3.4.4 时间复杂度分析 109
3.4.5 一些讨论 110
3.5 RNA 二级结构预测：对字符串的归约 111
3.5.1 算法设计与描述 112
3.5.2 算法运行过程示例 115
3.5.3 时间复杂度分析 116
3.6 隐马尔可夫模型的解码问题：对序列的归约 116
3.6.1 算法设计与描述 119
3.6.2 算法运行过程示例 123
3.6.3 时间复杂度分析 124

3.7 字符串之间的编辑距离：对一对字符串的归约 124
3.7.1 算法设计与描述 127
3.7.2 算法运行过程示例 131
3.7.3 时间复杂度分析 132
3.7.4 一些讨论 132
3.8 字符串局部联配：对一对字符串的归约 133
3.8.1 算法设计与描述 134
3.8.2 算法运行过程实例 137
3.8.3 时间复杂度分析 138
3.8.4 一些讨论 139
3.9 树上的顶点覆盖问题：对树的归约 139
3.9.1 算法设计与描述 140
3.9.2 时间复杂度分析 141
3.9.3 算法设计：更“细”的子问题表述方式 141
3.10 有向无环图上的单源最短路径：对图的归约 142
3.10.1 算法设计与描述 144
3.10.2 时间复杂度分析 146
3.11 一般有向图上的单源最短路径：对图的归约 147
3.11.1 算法设计与描述 147
3.11.2 算法运行过程示例 150
3.11.3 时间复杂度分析 151
3.11.4 一些讨论 151
3.12 应用动态规划技术求解整数规划问题 153
3.12.1 算法设计与描述 154
3.12.2 算法运行过程示例 155
3.12.3 时间复杂度分析 156
3.12.4 一些讨论 157
3.13 动态规划与马尔可夫决策过程 158
3.13.1 算法设计与描述 160
3.13.2 运行过程示例 160
3.13.3 一些讨论 161

第四章 高级动态规划 173

4.1 引言 . . . 173
4.2 降低空间复杂度：以算代存 . . . 174
4.2.1 线性空间复杂度的字符串联配算法：一个不太成功的尝试 . . . 174
4.2.2 更高效的动态规划算法：从中间字符开始的多步决策过程 . . . 176
4.2.3 进一步的改进：采用递归策略计算最优决策项 . . . 181
4.3 降低空间复杂度：以“存函数”代替“存值” . . . 184
4.4 降低时间复杂度：利用子问题最优解值的稀疏性 . . . 188
4.4.1 利用稀疏性加速动态规划算法 . . . 189
4.4.2 只与边界点比较以进一步加速 . . . 192
4.4.3 一些讨论 . . . 196
4.5 降低时间复杂度：基于最优解估计值的去冗余技术 . . . 196
4.5.1 计算两序列编辑距离的 Ficket 算法 . . . 196
4.5.2 改进版本：条带型动态规划算法 . . . 198
4.6 降低时间复杂度：利用单调性的动态规划 . . . 199
4.7 降低时间复杂度：QI . . . 199

第五章 贪心算法 202

5.1 引言 . . . 202
5.2 依据优化目标直接设计贪心规则：作业规划问题 . . . 204
5.2.1 算法设计与描述 . . . 205
5.2.2 运行过程示例 . . . 207
5.2.3 时间复杂度分析 . . . 207
5.2.4 一些讨论 . . . 207
5.3 依据优化目标直接设计贪心规则：最短路径问题 . . . 208
5.3.1 算法设计与描述 . . . 208
5.3.2 运行过程示例 . . . 213
5.3.3 时间复杂度分析 . . . 213
5.3.4 一些讨论 . . . 215
5.4 将动态规划简化成贪心算法：区间调度问题 . . . 216
5.4.1 求解区间调度问题的动态规划算法 . . . 216
5.4.2 区间调度问题的特殊情形及贪心求解算法 . . . 218

5.4.3 关于贪心规则设计过程的一些讨论 222
5.5 将动态规划简化成贪心算法：再论最短路径问题 224
5.5.1 Bellman-Ford 算法中的第一类冗余计算 224
5.5.2 对 Bellman-Ford 算法的第一次简化 227
5.5.3 Bellman-Ford 算法中的第二类冗余计算 228
5.5.4 对 Bellman-Ford 算法的再次简化：Dijkstra 贪心算法 . . 228
5.6 最小加权集合覆盖问题：基于神经网络的贪心算法 229
5.7 贪心算法的理论基础：拟阵 229
5.8 贪心算法的理论基础：次模函数 229

第一章 优先队列及均摊分析 232

A.1 附录一：优先队列 232
A.1.1 引言 232
A.1.2 实现方式 1：数组和链表 232
A.1.3 实现方式 2：二叉堆 232
A.1.4 实现方式 3：二项式堆 233
A.1.5 实现方式 4：Fibonacci 堆 237

Appendices 232

第一章　绪论

所谓*算法*，是指求解给定问题的*操作步骤*—无论是人还是计算机，按步骤机械式地操作，即可求得给定问题的解。

那么对于一个给定的问题来说，怎样才能设计出求解问题的算法来呢？如果我们脱离对问题特性的认识、只是简单套用已知的算法技术的话，一般很难行得通—算法设计的关键在于深入观察待解决问题的特性；我们对问题特性认识得越多，可资使用的算法技术就越多，设计出的算法也越好。

好的算法不仅是正确的，还应是高效的。因此在设计出算法之后，我们还需要对其性能进行分析，以评估算法运行所需时间和空间的数量；在此基础上，我们可以比较不同算法的性能，进而改进算法的设计。

在本章里，我们首先介绍如何形式化描述算法问题；然后以旅行商问题 (Traveling Salesman Problem, TSP) 为例介绍算法设计的基本过程；最后以计算 Fibonacci 数为例介绍算法复杂度分析的基本思想。

1.1　什么是算法问题？

人们在工作或者日常生活中，常常需要求解一些*实际问题*（Practical problems），例如 *100 Years on the Road* [**?**] 一书中记载的下述真实问题：

1925 年，Page 种子公司的推销员 H. M. Cleveland 从公司出发，去 350 个城镇推销种子。他手里有一幅地图，能够知道每个城镇的地理位置，以及两两城镇之间的距离。在出发之前，Cleveland 面临的问题是：如何规划路线，使得走遍所有城镇后回到公司的总里程最短？

在使用计算技术求解上述实际问题时，公司名称、推销员姓名等信息是无关紧要的。为避免这些无关信息的干扰，我们将其剥离出去，只保留那些对求解来说必要的信息，最终形式化描述成如下的*算法问题* (Algorithmic problem)：

旅行商问题 (Traveling Salesman Problem, TSP)

输入：结点集合 $V = \{v_1, \cdots, v_n\}$，以及结点间距离矩阵 $D = (d_{i,j}) \in \mathbf{R}^{n\times n}$，其中 $d_{i,j}$ 表示结点 i 与结点 j 之间的距离；

输出：最短的环游路线 (Tour)，即以任一结点作出发点，经过每个结点一次且仅一次、最终返回出发点的里程最短的环游。

算法问题的形式化描述中包含输入和输出两个部分。所谓输入和输出，都是对求解算法而言的：输入部分描述问题的所有参数及其格式，表示我们已知的信息；而输出部分表示问题的“解”必须满足的条件，是算法工作的目标。

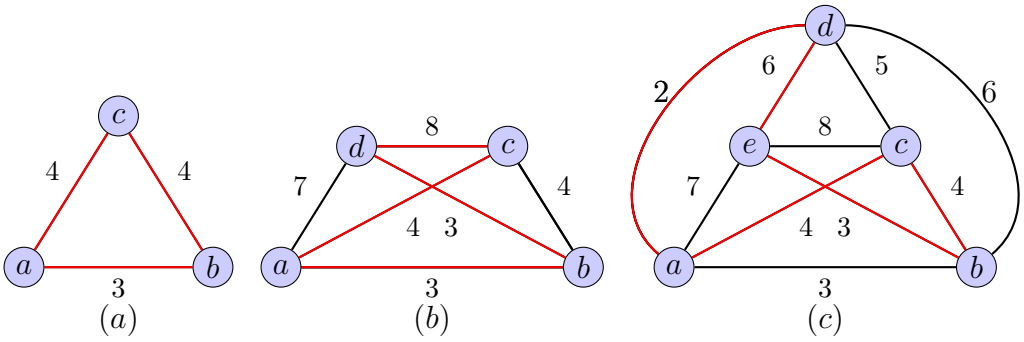


图 1.1: 旅行商问题的 3 个实例：$n = 3, 4, 5$，其中结点对之间的距离标注于相应的边上，最短环游路线使用红色示出。

在日常用语中，“问题”常常指特定的一个提问或者难题；但是在计算机算法中，“问题”指的是多个同类型提问的总称。换句话说，算法问题是一个**一般性的提问**：所谓一般性，是指“问题输入部分包含一些参数、其取值尚未确定”；当所有的参数取值都确定了之后，所指明的**特定的输入**被称作**问题的实例** (Problem instance)。以旅行商问题为例，参数是结点集合 V 以及距离矩阵 d；图 1.1所示实例包含的结点以及距离矩阵都明确给出。（熟悉面向对象程序设计的同学很容易联想到，“问题”和“实例”之间的关系非常类似于“类”和“对象”之间的关系。）当一个算法能够应用于问题的所有实例，并始终能够获得符合问题“输出部分”规定的解时，我们才称**此算法解答了这个问题**。

不同实例的求解难度也不同。图 1.1所示实例 (a) 包含 3 个结点，有 2 种环游路线且里程相同，因此求最短环游是非常简单的；实例 (b) 包含 4 个结点，共有 6 种环游路线（见表 1.1）；实例 (c) 包含 5 个结点，环游路线增加到 24 种，导致寻找最短环游的难度大大增加。

值得指出的是，在问题的形式化过程中，我们通常还要进行抽象。就 TSP 问题来说，我们将“城市”抽象表示成“结点”。这样处理的好处是：抽象出的算法问题不仅

表 1.1: TSP 问题实例 (b) 的所有环游路线及其总里程

环游路线	里程
$a \to b \to c \to e \to a$	22
$a \to e \to c \to b \to a$	22
$a \to c \to b \to e \to a$	18
$a \to e \to b \to c \to a$	18
$a \to b \to e \to c \to a$	18
$a \to c \to e \to b \to a$	18

可以表示 Cleveland 所面临的路线规划问题，还可以表示与之相似的一大类问题，从而具有推广的可能性。

1.2 怎样进行算法设计?

那么当给定一个算法问题时，怎样才能设计出求解此问题的算法呢?

在介绍算法设计的基本过程之前，我们不妨先来读一下 V. Vazirani 说过的具有指导意义的一段话 [?]:

“无论是设计算法还是描述算法，其背后的哲学都和米开朗基罗做雕塑的过程非常相似：在创作一件艺术品时，米开朗基罗的绝大部分的努力和时间，都花在寻找一块‘有内涵’的石头、并琢磨这块石头内蕴的天然结构上。一旦看清楚了这块石头所内蕴的天然结构之后，斧凿之功就是简单的事情了。

之所以说算法设计过程与米开朗基罗的创作过程很相似，是因为我们在设计算法时，绝大部分努力与时间都花在琢磨问题内蕴的组合结构上。一旦琢磨清楚问题的内在结构之后，设计合适的算法就是简单的事情了—依据问题的结构，设计求解问题的操作步骤，并尽量保持操作步骤的精炼与简单。”

或许反过来的思考能够更好地理解这段话：对于一个给定的问题，如果我们脱离了对问题特性的认识，只是简单地套用已知的算法技术，一般很难行得通—算法设计的关键在于深入观察待解决问题的特性；我们对问题特性认识得越多，可资使用的算法技术就越多，设计出的算法也就可能更好。

1.2.1 观察问题内在结构的途径

要想观察问题的内在结构，我们可以尝试做如下的思考：

- **观察一、问题的可分解性**：问题的*最简单实例*是什么？复杂的实例能否*分解*成简单的实例？
- **观察二、可行解的形式以及解之间的变换关系**：问题的*可行解的形式*是什么？可行解的总数有多少？可行解能否一步一步地逐步构建出来？我们能否对一个可行解施加小幅扰动，将之*变换成另一个可行解*？
- **观察三、类似的问题**：和给定问题*类似的问题*有哪些？解决类似问题的算法能否直接应用于解决当前的问题？如果不能，那又是什么因素造成了妨碍？能否想办法消除这些妨碍因素？
- **观察四、现有算法的不足**：现有的求解算法有哪些不足？比如：是否存在冗余计算？如果存在的话，能否想办法去除冗余计算？

依据对问题内在结构的观察，我们采用相应的策略设计算法：

“分而治之”策略：如果我们观察到问题具有*可归约性*，即问题的复杂实例可以分解成一些简单实例，而且反过来，简单实例的解可以用来组成原问题的解，那我们就可以尝试*基于归约原理的算法设计* [?]。这一类算法的典型代表是*分而治之* (Divide and Conquer)，即先把问题的复杂实例分解成一些简单实例，然后通过递归调用求解简单实例，最后用简单实例的解*组合*出复杂实例的解。这些简单实例具有和原始的复杂实例相同的形式，只是规模更小一些，也常常称作*子实例*（Sub-instance）；求解子实例则称为对应于原始问题的*子问题*（Sub-problem）。

在此基础上，我们做进一步的观察：如果待求解的问题是一个优化问题，而且我们能够观察到*最优子结构性质* (Optimal substructure)，那么我们可以尝试设计*动态规划* (Dynamic programming) 求解算法。所谓最优子结构性质，是指原始复杂实例的最优解能够由子实例的最优解组合而成。

再进一步地观察问题结构：如果待求解的问题不仅具有最优子结构性质，还具有*贪心选择性质* (Greedy selection)，那我们可以尝试设计贪心算法求解问题。所谓贪心选择性质，是指*局部最优决策* (Locally optimal decision)，可以直观地理解为做决策时的*短视*策略，即：问题的解可以一步一步地逐渐构造而成；在构造的每一步，无需考虑尚未求解的子问题，只依据已经构造出的部分解即可做出最优或者近乎最优的选

择。在这一点上，贪心算法与动态规划算法显著不同：动态规划算法中需要考虑子问题的解、通过回溯才能确定出最优解。

“逐步改进”策略：如上所述，能够应用*分而治之*策略进行算法设计的前提是我们观察到问题具有可分解性。但是对于不容易分解的问题（或者虽然能够分解但是我们并不去分解），又该如何设计求解算法呢?

在这种情形之下，我们可以尝试*逐步改进* (Improvement strategy) 策略，即首先构造出给定问题的一个粗糙的*完整可行解* (Complete solution)，然后逐步修正完整解以改进其质量，直到获得满意的解。值得指出的是，由于不进行问题的分解，也就不存在子问题以及子问题的解这些概念；我们只有问题的完整可行解可资使用，因此也只能逐步修正完整可行解以改进其质量。所谓的改进完整解，是指将一个完整解进行小的扰动，变成另一个完整解；其中所依赖的是对*完整解之间变换关系*的观察。

逐步改进类的典型算法有线性规划 (Linear programming)、非线性规划 (Non-linear programming)、计算最大流的网络流算法 (Network flow)、局部搜索 (Local search)、模拟退火 (Simulated annealing) 等。最大流问题为逐步改进策略提供了一个很好的注解：最大流问题不容易分解成子问题，因此到目前为止，尚未设计出求解最大流问题的分而治之类高效算法；事实上，当前最高效的算法都是采用逐步改进的策略 [**?**]。

智能枚举策略：除了观察问题是否能分解之外，我们还可以观察问题的*可行解的形式*。假如问题的可行解可以表示成如下形式：

$$X = [x_1, x_2, \cdots, x_n],\ x_i = 0/1$$

则我们可以尝试采用*枚举*（Enumeration）策略设计求解算法，即先产生出所有的可行解，然后从中找出符合要求的解。一般来说，可行解的数目较多，导致简单的枚举策略求解时间过长。为加快求解过程，通常采用*智能枚举*（Intelligent enumeration）策略，即对枚举树进行剪枝，忽略一些低质量的可行解，从而加速枚举过程 [**?**]。事实上，智能枚举策略的适用范围很广—只要可行解能够一步一步地，逐渐构建出来的问题，都可以考虑尝试智能枚举策略。

枚举类的典型算法有回溯法（Backtracking）、分支限界算法（Branch and bound）等。值得指出的是，贪心算法可以看做枚举策略的一个特例：贪心算法在构造解的每一步都依据贪心规则作出选择，最终获得解实质上是枚举树中的一条路径，因此可以看做是最极端的一种剪枝。

针对“难”的问题的算法设计策略：有一些问题比较难，迄今尚未找到快速的求解算法。难度是问题的本质属性；我们可以通过观察问题之间的*归约关系*来证明某个问题比另一个问题难度更高。证明了给定的问题是难的问题，就意味着只有放松要求才能设计出高效的算法：比如我们可以不再要求算法一定要求得最优解、只要求获得“足够好”的解；我们也可以不再要求算法的每一步操作都是确定性的，允许算法中存在随机化行为，仅要求算法输出错误结果的概率很小、或者期望运行时间很短；我们还可以不再要求算法对“最坏的问题实例”能够很快运行，仅仅要求算法在大部分现实情况下运行很快。值得指出的是：在放松了要求之后，我们依然要先观察问题结构，继而基于问题结构设计高效的求解算法 [**?**]。

下面我们以旅行商问题为例，说明如何对问题进行观察，以及在此观察基础上如何应用上述三类策略进行算法设计。

1.2.2 “分而治之”算法设计过程简介

在设计求解 TSP 问题的算法之前，我们首先*从最简单的实例入手*，画一些简单的实例以观察规律。如图 1.1(a) 所示，最简单的 TSP 实例是只有 3 个结点的情形，其最短环游非常容易计算。

接下来我们考察*复杂的实例能否分解成简单实例*、以及*能否使用简单实例的解组合出复杂实例的解*。然而不幸的是，虽然一个复杂的 TSP 实例能够比较容易地分解成简单实例，但是用简单实例的解组合出复杂实例的解却不太容易。如图 1.1所示，对于包含 5 个结点 a, b, c, d, e 的 TSP 实例 (c) 来说，我们去除结点 d 后，即获得一个只有 4 个结点 a, b, c, e 的 TSP 实例 (b)；但是由于要求解是一个环游，我们很难找到一个通用的构造规则，能够将实例 (b) 的解（由红色边示出的最短环游）转换成实例 (c) 的解（由红色边示出的最短环游)。

转换目标：求解一个辅助问题

既然难以设计出直接求解 TSP 问题的分而治之算法，我们就转换一下目标，求解一个*辅助问题*：计算从起始结点 s 出发、经过结点集合 S 中的结点一次且仅一次、最终到达目的结点 x 的*最短路径*，其长度记为 $M(s, S, x)$。研究这个辅助问题 $M(s, S, x)$ 有两点好处：

(1) 可以基于辅助问题的解构造出 TSP 问题的解；

(2) 易于设计一个分而治之算法计算 $M(s,S,x)$。

我们首先来看第一点。考察图 1.1(b) 所示的 TSP 实例：假设我们从结点 a 出发，则环游路线最终返回 a 时有三种情况，即：从 b 返回 a、从 c 返回 a，以及从 e 返回 a；因此，我们可以将最短环游路线的里程表示为

$$\begin{aligned}\min\{\quad & d_{b,a}+M(a,\{c,d\},b),\\ & d_{c,a}+M(a,\{b,d\},c),\\ & d_{e,a}+M(a,\{b,c\},e)\}.\end{aligned}$$

上述观察可以推广到任意的 TSP 实例：如果对任意的 $s\in V$, $x\in V$, $S\subseteq V$，都能够计算出 $M(s,S,x)$, 则可以求解原始的 TSP 问题。此算法称为 Bellman-Held-Karp 算法 [**?**, **?**]，其伪代码表示如下：

Algorithm 1 求解 TSP 的 Bellman-Held-Karp 算法

function Bellman-Held-Karp(V,D)

Require: $|V|\geq 3$

1: **if** $|V|=3$ **then**
2: **return** $d_{v_1,v_2}+d_{v_2,v_3}+d_{v_3,v_1}$;
3: **else**
4: **return** $\min_{x\in V,x\neq v_1}\{M(v_1,V\backslash\{x\},x)+d_{x,v_1}\}$;
5: **end if**

此处我们是以 V 中第一个结点 v_1 作为起始结点。

接下来我们看第二点：如何计算 $M(s,S,x)$。我们依然先从最简单的实例 $|S|=1$ 入手。如图 1.2(a) 所示，当 $S=\{c\}$ 时，$M(a,\{c\},b)$ 可以非常容易地计算出来：

$$M(a,\{c\},b)=d_{a,c}+d_{c,b}.$$

更重要的是，对于 $|S|\geq 2$ 的复杂实例，我们可以将其归约成简单实例。如图 1.2(b) 和 (c) 所示，我们可以如下计算 $M(a,\{c,e\},b)$：

$$M(a,\{c,e\},b)=\min\{d_{c,b}+M(a,\{e\},c),\ d_{e,b}+M(a,\{c\},e)\}.$$

因此我们可以设计如下的分而治之算法计算 $M(s,S,x)$：

Algorithm 2 计算 $\mathrm{M}(s, S, x)$ 的递归算法

function $\mathrm{M}(s, S, x)$

1: **if** $|S| = 1$ **then**
2: Let's represent S as $S = \{v\}$;
3: **return** $\mathrm{M}(s, S, x) = d_{s,v} + d_{v,x}$;
4: **end if**
5: **return** $\min_{v \in S,\ v \neq x} \mathrm{M}(s, S \backslash \{v\}, v) + d_{v,x}$;

值得注意的是，对 TSP 问题来说，由于解必须是环游这一约束，使得我们难以基于子问题的解构造出原问题的解。与之相反，$M(s, S, x)$ 的计算避开了这个困难：引入不同于出发点 s 的目的结点 x 之后，解不再是一个环游，而只是一条路径，从而使得我们能够基于子问题的解构造出原问题的解。这也是能够应用分而治之技术计算 $M(s, S, x)$ 的原因之所在。

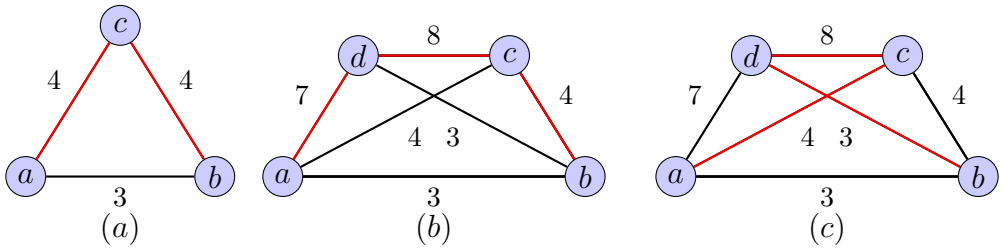


图 1.2: $M(s, S, x)$ 的计算过程示例。(a) 当 S 只包含一个结点 c 时，$M(a, \{c\}, b) = d_{a,c} + d_{c,b}$。$(b)$, (c) 当 S 包含 2 个结点 c, d 时，$M(a, \{c, d\}, b) = \min\{d_{c,b} + M(a, \{d\}, c), d_{d,b} + M(a, \{c\}, d)\}$

表 1.2展示对图 1.2(b) 所示 4 个结点的 TSP 实例计算出的 $M(a, S, x)$ 表格。依据这个表格，我们进而可以计算出最短环游长度是 $\min\{M(a, \{b, c\}, d) + d_{d,a}, M(a, \{b, d\}, c) + d_{c,a}, M(a, \{c, d\}, b) + d_{b,a}\} = 18$。

表 1.2: 对图 1.2(b) 所示 TSP 实例计算出的 $M(a,S,x)$ 表格

	到达结点 x		
中间结点集合 S	b	c	d
$\{b\}$	-	7	6
$\{c\}$	8	-	12
$\{d\}$	10	15	-
$\{b,c\}$	-	-	11
$\{b,d\}$	-	14	-
$\{c,d\}$	15	-	-

由于需要枚举所有的结点子集 S, 因此整个表格的大小是 $O(2^n n)$; 而且计算每个单元时都需要 n 次比较, 故 Bellman-Held-Karp 算法的时间复杂度是 $O(2^n n^2)$ 。

1.2.3 “逐步改进”算法设计过程简介

与*分而治之*算法的思想截然不同,“逐步改进”策略不考虑如何将原始问题分解成子问题、以及如何将子问题的解组合成原问题的解, 而是直接考虑*原问题的完整可行解*。以图 1.1(c) 中所示的 TSP 实例为例, $a \to b \to c \to d \to e \to a$ 是一条环游路线, 是这个实例的一个完整可行解。当然, 这个完整可行解的总里程是 25,“质量”不太高, 需要想办法进行改进。

“逐步改进”算法的*基本过程*是: 从问题的一个粗糙的、质量不太高的完整可行解开始, 不断进行改进, 直至获得满意的解为止; 在算法运行过程中, 考虑的都是完整可行解。

求解 TSP 问题的“逐步改进”算法的一般性框架可以描述如下:

Algorithm 3 求解 TSP 的“逐步改进”算法框架

function GENERIC-IMPROVEMENT(V, D)

```
1: Initialize a tour S;
2: while TRUE do
3:     Select a new tour S' from the neighbourhood of S;
4:     if S' is shorter than S then
5:         S = S';
6:     end if
7:     if STOPPING(S) then
8:         return S;
9:     end if
10: end while
```

上述的一般性框架中，有 3 处需要做进一步的明确规定：

(1) *初始可行解的选择*（第 1 行）：可以任意选择一个初始可行解，也可以采用启发式规则等选择高质量的初始点以加快改进过程。为简单起见，此处我们采用选择任意初始解的策略。

(2) *可行解的改进方法*（第 3 行）：将一个低质量的可行解进行修改，变成另一个质量更高的可行解，是“逐步改进”算法的核心。这依赖于我们对*可行解之间的变换关系*的定义，即需要规定对一个解 S 做一些小的“扰动”（Perturbation）之后，可以变换成哪些解 S'；所有可能的变换形成的解 S' 构成 S 的“邻域”（Neighborhood）。

此处我们采用 `2-Opt` 扰动，即从环游 S 中选择不相交的 2 条边，交换其端点，产生一个新的环游 S'；扰动前后的环游只在两条边上存在差异。以图 1.3所示的环游 $S = a \to b \to c \to d \to a$ 为例，我们首先选择顶点不相交的两条边 (a, d) 和 (b, c)，然后删除这两条边，并将 4 个端点交叉连接生成两条新的边 (以红色示出)，即 a 连接 c、b 连接 d，最终获得新的环游 $S' = a \to b \to d \to c \to a$。其他常用的解变换规则包括 `3-Opt`, Lin-Kernighan 启发式规则等 [**?**]。

(3) *算法的终止条件*（第 7 行）：由于算法是对可行解迭代进行改进，因此需要规定解满足何种条件时算法终止。常用的终止条件包括：i) 当前可行解无法

进一步改进；*ii*) 迭代次数超过预先定义的阈值；*iii*) 当前可行解质量超过预先定义的阈值。此处我们采用第一种方案。

图 1.4展示“逐步改进”算法运行的一个例子：算法运行的初始可行解是环游 $S = a \to b \to c \to d \to e \to a$（蓝色边组成的路径），总里程为 25；第一次迭代情况见图 1.4(a)：选择两条边 (a, e) 和 (c, d) 执行 `2-Opt` 操作，生成新的边 (a, d) 和 (c, e)（红色边），并相应地生成新的环游 $S = a \to b \to c \to e \to d \to a$，总里程降低为 23；第二次迭代情况见图 1.4$(b)$：选择两条边 (a, b) 和 (c, e) 执行 `2-Opt` 操作，生成新的边 (a, c) 和 (b, e)（红色边），并相应地生成新的环游 $S = a \to c \to b \to e \to d \to a$，总里程进一步降低至为 19（见图 1.4$(c)$）。此时无论选择哪两条边进行 `2-Opt` 操作，都不能产生更短的环游，因此算法结束，返回环游 $S = a \to c \to b \to e \to d \to a$ 作为最终结果。

需要指出的是：虽然在这个实例上“逐步改进”算法求得了最优解，但是在一般情况下，“逐步改进”算法不能保证获得最优解。

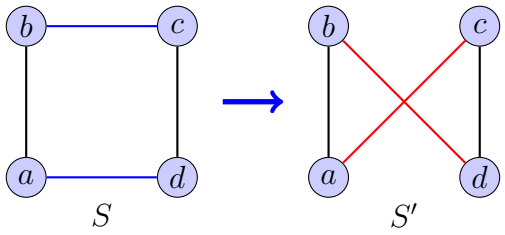


图 1.3: TSP 问题中解变换的 `2-Opt` 规则。采用此规则，将环游 $S = a \to b \to c \to d \to a$ 变换成新的环游 $S' = a \to b \to d \to c \to a$

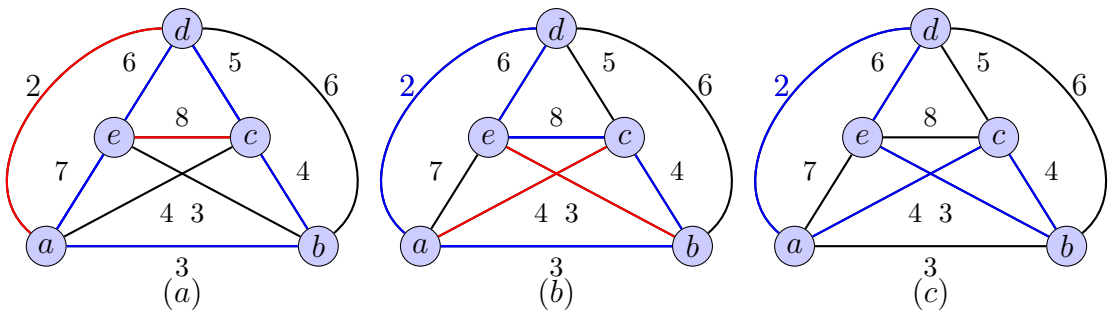


图 1.4: “逐步改进”算法求解旅行商问题的运行示例。(a) 和 (b) 显示算法执行的两次迭代改进过程，(c) 显示求得的最终解。在算法运行过程中，完整可行解用蓝色边示出，`2-Opt` 操作产生的新边用红色示出

1.2.4 智能枚举算法设计过程简介

除了观察问题的可分解性之外，我们还可以观察**可行解的形式**。首先看一个具体例子：图 1.5所示的实例中共有 10 条边，蓝色标识的环游包括 $e_1, e_4, e_5, e_8, e_{10}$ 共 5 条边，可以表示为

$$X = 1001100101,$$

其中 $x_i = 1$ 表示环游经过边 e_i，$x_i = 0$ 表示环游不经过边 e_i。

我们可以将上述观察推广到任意的 TSP 实例，即将其完整可行解表示成一个向量 $X = x_1x_2\cdots x_m$, 其中 $x_i = 0/1$ $(1 \le i \le m)$，共有 n 个 $x_i = 1$。此处的 n 表示结点的个数，m 表示边的条数。

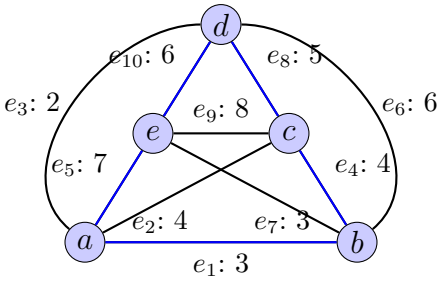


图 1.5: 用向量表示旅行商问题可行解。我们首先对所有边进行标号；蓝色标识的环游包含 5 条边，可以表示为 $X = 1001100101$，其中 $x_i = 1$ 表示环游经过边 e_i，$x_i = 0$ 表示环游不经过边 e_i

那怎样才能枚举出所有的长为 m、其中有 n 个 1 的 0-1 向量呢?

直接枚举出整个向量不容易，但是枚举一个分量是很容易的，因为分量只能取 0 和 1 两个值。因此，我们可以采用“逐次确定分量”的策略，即：每次选择一个分量，把它的值分别设置成 0 和 1。这样，在枚举过程中，有些分量的值是已确定的，有些分量的值尚未确定（我们用 ? 表示）。我们称所有分量都确定的向量为完整解，例如 $X = 1001100101$，表示环游 $a \to b \to c \to e \to d \to a$；我们称仅部分分量确定的向量为部分解，例如 $X = 1?????????$，表示仅已知环游使用了边 e_1，其他边是否使用尚未确定；一个极端情形是 $X =??????????$，表示所有的边都尚未确定是否被环游使用。

对于具有向量形式的可行解，我们可以将所有的可行解（包括完整解和部分解）组织成一棵树，称为部分解树（Partial Solution Tree [**?**, **?**]）。

用于枚举可行解的“部分解树”

如图 1.6所示，部分解树的根结点是 X =??????????；树的每个结点表示一个解：**内部结点**表示部分解，即只有一些分量 x_i 的取值已确定；**叶子结点**表示完整解，即所有分量 x_i 都已确定其取值。**树中的边**从一个部分解 X 指向另一个解 X'，其中 X' 比 X 中多了 1 个分量确定了取值。

我们还可以从另一个角度认识内部结点和部分解：一个叶子结点仅表示一个完整解，而一个内部结点则表示**一族完整解**，即以此内部结点为根的子树上所有叶子结点对应的完整解。

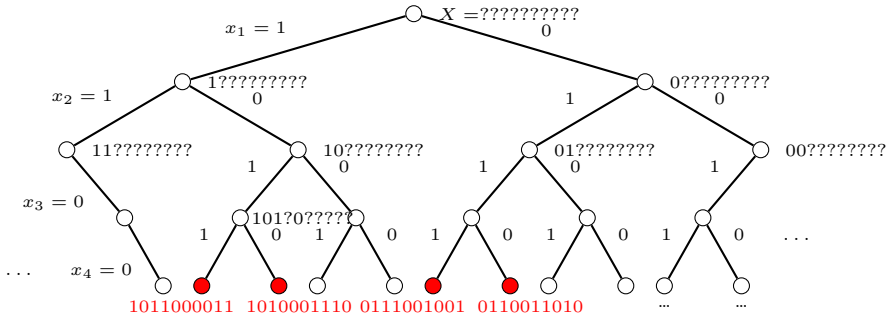


图 1.6: 枚举 TSP 可行解的部分解树。每个内部结点表示一个部分解，叶子结点表示一个完整解（即环游）；边表示设定某一个分量 x_i 的取值

朴素枚举算法

求解给定的 TSP 实例，可以通过枚举所有的可行解、然后选择里程最短的可行解来实现。我们可以采用如下策略来枚举所有的可行解：从根结点 X =?????????? 开始，逐步构建出部分解树；每一步都选择一个未确定取值的分量 x_i，分别尝试其两种取值 $x_i = 1$ 与 $x_i = 0$，从而扩展成两个新的结点；如此逐步增大能够确定取值的分量的个数，直至所有分量都确定取值，从而获得了完整可行解。这种朴素的枚举算法描述如下：

Algorithm 4 求解 TSP 的枚举算法框架

function GENERIC-ENUMERATION-FOR-TSP(V, D)

Initialize active set of unexplored nodes $A = \{X_0\}$. Here, X_0 =??...?
$best\_so\_far = \infty$;
while $A \neq \{\}$ **do**
  Choose and remove a node X from A, and select an undetermined item x_i from X;
  for $v \in \{0, 1\}$ **do**
    Expand X into node X' by setting $x_i = v$;
    if X' represents a complete solution **then**
      Update $best\_so\_far$ if X' has better objective function value;
    else
      Insert X' into A; //X' needs to be explored further;
    end if
  end for
end while
return $best\_so\_far$;

由于存在指数多个完整可行解，因此需要指数多的运行时间才能构建出完整的部分解树、枚举出所有的可行解。即便对于规模较小的 TSP 实例，这种朴素的枚举算法往往也难以求解。

为提高求解速度，一种常用的策略是对部分解树进行**剪枝**（Pruning），即：(i) 对于每个内部结点，评估与其对应的部分解 X 的**质量**；(ii) 在枚举中不扩展低质量的部分解，无需考虑由其扩展而成的完整解；直观上看，我们“剪掉”了以此结点为根的子树，从而降低了需要构建的部分解树的大小。这种带剪枝的枚举策略称作**智能枚举** (Intelligent enumeration [**?**])。

智能枚举算法

智能枚举算法的核心在于如何**评估部分解的质量**。对于一个完整解来说，我们可以精确计算出与其对应环游的里程；而对于一个部分解而言，我们仅仅知道环游的部分信息（如已知一些边被用、一些边肯定不会用），无法精确计算出环游的里程，因此只能估计部分解的质量。常用的估计量之一是部分解 X 所代表的**一族完整解的里程的下界**（Lower bound），记为 $LB(X)$，计算方法如下：

- 我们先考察特殊的部分解 X =??????????。我们依次考察 TSP 实例中的所有结

点，对每一个结点，从与其相邻接的 $n-1$ 条边中选择最短的两条边，则共可获得 $2n$ 条边。容易证明，此 $2n$ 条边的里程之和的一半，是最短环游里程的下界。以图 1.5所示实例为例，我们可以计算出下界：

$$LB(X) = \frac{1}{2}(5+6+8+7+9) = 17.5$$

- 对于一般的部分解 X，我们只需对上述过程做小幅修改：在选择最短的两条邻接边时，需要满足部分解 X 蕴含的约束。以部分解 $X = 10????????$ 为例，$x_1 = 1$ 表示边 e_1 已使用，$x_2 = 0$ 表示边 e_2 不可使用，因此对结点 c 来说，最短的两条邻接边只能使用 e_4 和 e_8，其和为 9；从而下界是：

$$LB(X) = \frac{1}{2}(5+6+9+7+9) = 18$$

利用部分解的下界信息进行剪枝的智能枚举算法称为**分支限界法**（Branch and bound），描述如下：

Algorithm 5 求解 TSP 的智能枚举算法

function INTELLIGENT-ENUMERATION-FOR-TSP(V, D)

```
1: Initialize active set of unexplored nodes A = {X_0}. Here, X_0 =??...?
2: best_so_far = ∞; //Store the best tour till now;
3: while A ≠ {} do
4:    Choose a node X ∈ A such that LB(X) ≤ best_so_far, and remove X from A;
5:    Select an undetermined item x_i from X;
6:    for v = 0 to 1 do
7:       Expand X into node X' by setting x_i = v;
8:       if X' represents a complete solution then
9:          Update best_so_far if X' has better objective function value;
10:      else if LB(X') ≤ best_so_far then
11:         Insert X' into A; //X' needs to be explored further;
12:      end if
13:   end for
14: end while
15: return best_so_far;
```

与朴素的枚举算法相比，智能枚举算法只在两个地方有所不同：(i) 当选择一个部分解 X 进行扩展时，增加了一个约束，要求 $LB(X)$ 必须小于当前已获得的最好环游 $best\_so\_far$（第 4 行）；(ii) 向待扩展结点集 A 增加部分解 X 时，也增加了相同

的约束（第 11 行）。换句话说，对于那些 $LB(X) > best\_so\_far$ 的部分解 X，与其对应的一族完整解不会优于现在已知的最好解 $best\_so\_far$，因此无需进行扩展，从而实现了“剪枝”。

对于图 1.5所示实例，算法 IntelligentEnumerationForTSP 共需迭代执行 7 轮，其中前 4 轮迭代后生成的部分解树见图 1.8。我们以第 4 轮迭代为例详细描述“剪枝”过程：算法选择部分解 $X_5 = 101?0?????$ 进行扩展，在分别设置 $x_4 = 1$ 与 $x_4 = 0$ 并进行一些推理之后，生成两个完整解 $X_7 = 1011000011$ 和 $X_8 = 1010001110$，相应的环游里程分布为 23 和 21，因而将 $best\_so\_far$ 更新为 21；由于部分解 X_6 的下界 $LB(X_6) = 23$ 大于 $best\_so\_far$，意味即使扩展 X_6 也不会产生优于 X_8 的环游，因此我们从 A 中去除 X_6。类似地，由于所有边上的距离都是整数，因此虽然 $LB(X_3) = 20.5$，我们依然可以断定扩展 X_3 也不会产生优于 X_8 的完整解，从而将 X_3 从 A 中去除。

当执行了 7 轮迭代之后（见图 1.8），所有的部分解要么被扩展，要么被剪枝；算法生成一棵仅有 15 个结点的部分解树，就求出了最优解 $X_4 = 0111001001$，对应的环游里程为 19。和朴素的枚举算法比较可见，智能枚举算法大大减少了所需考查的结点，从而显著提高了速度。

可行解的另一种向量表达形式：以顶点为分量

除了将环游表示成以边为分量的向量之外，我们还可以将可行解表示成以顶点为分量的向量。比如对于如图 1.5所示实例来说，环游 $a \to b \to c \to d \to e \to a$ 可以表示为向量 $X = abcd$（还剩一个结点，必定是 e，故省略）。上述观察可以推广到任意实例，即环游可以表示为：

$$X = x_1 x_2 \cdots x_{n-2}, \quad x_i \in V \quad (1 \le i \le n-2).$$

类似于上一小节所做的处理，在将可行解表示成向量形式之后，我们可以采用智能枚举算法构造部分解树，并最终获得最优解。如图 3.7所示，智能枚举算法使用了一棵仅包含 15 个结点的部分解树，即求出最优解 $X_{14} = adeb$，其对应的环游里程为 19。

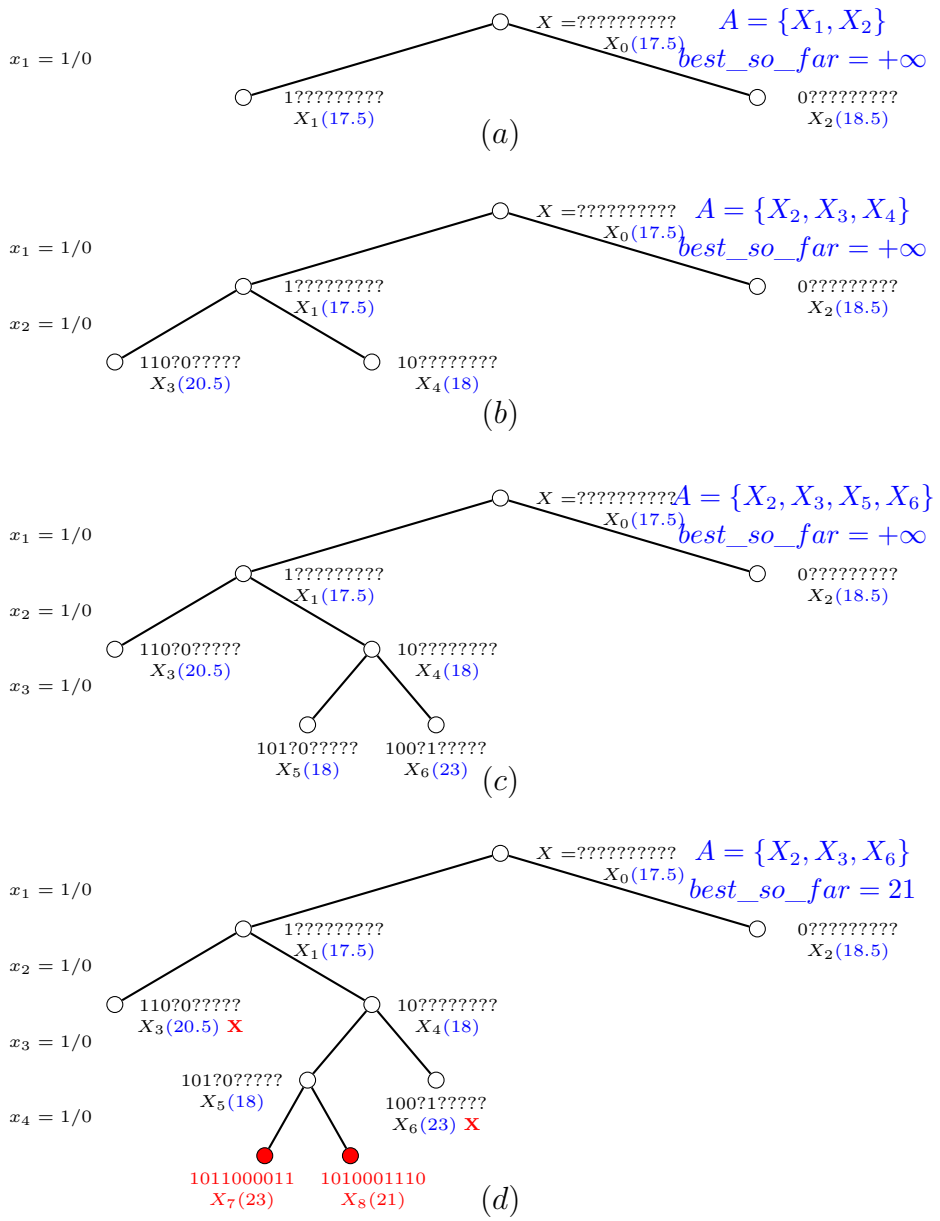


图 1.7: 算法 IntelligentEnumerationForTSP 迭代执行 1-4 轮产生的部分解树

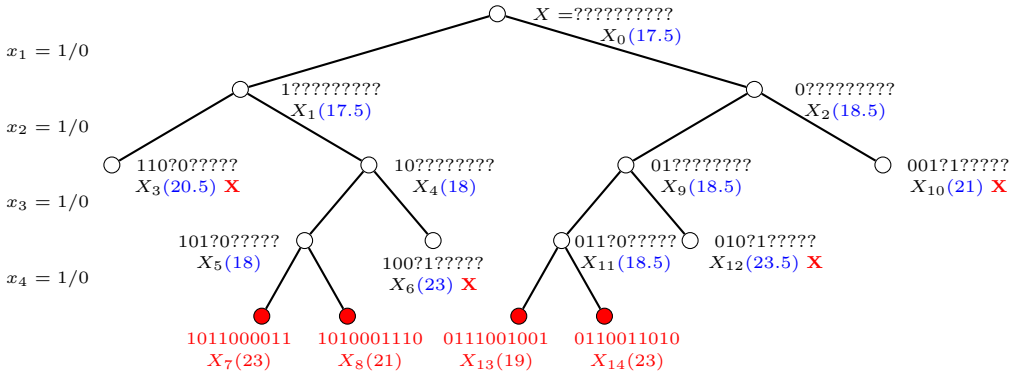


图 1.8: 算法 IntelligentEnumerationForTSP 迭代执行 7 轮后产生的部分解树，其中 $A = \{\}$，$best\_so\_far = 19$

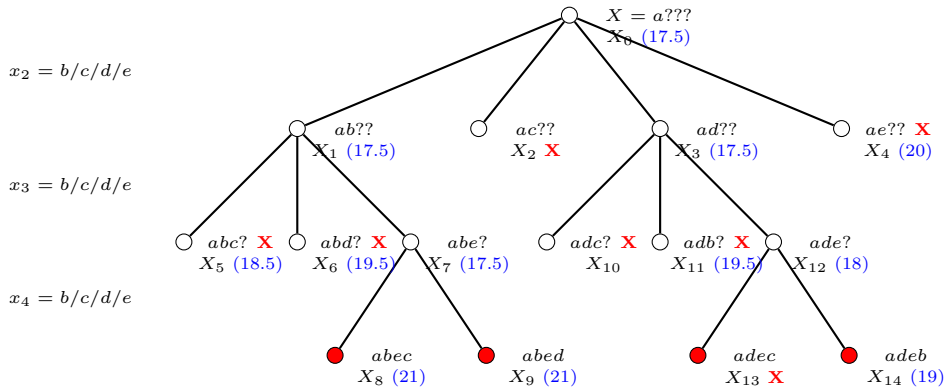


图 1.9: 算法 IntelligentEnumerationAlgoForTSP 迭代执行 6 轮后产生的部分解树，其中 $A = \{\}$，$best\_so\_far = 19$。为避免对同一条环游的正向和逆向重复搜索，算法仅考虑 b 出现在 c 之前的环游，故在 $x_2 = ac??$ 处进行剪枝

1.3 算法的复杂度

好的算法不仅是正确的，还应该是高效的。算法的效率可以从两个方面进行定量刻画：计算机运行算法求解问题时所需要的**运行时间长短**、以及**占用存储单元的多少**。

我们采用计时技术可以很容易地获得算法运行时间的精确数值，得到“对特定的实例，算法运行了多少秒”这样的详细信息。然而这种精确运行时间不仅与算法的优劣相关，还受实现算法所用的编程语言、CPU 性能、操作系统使用的缓冲策略等多种因素影响，导致在一台计算机得到的结论无法推广到另一台计算机上去。

为单纯地评价算法的优劣，我们脱离开运行所使用的计算机以及算法的实现等细节，只考虑算法执行过程中的**基本操作的次数**。我们假定所有的基本操作的用时都是 1 个时间单位，因此基本操作的次数就可以用来代表算法的运行时间（这个假定的合理性，我们将在第 9 章讲述）。至于哪些操作是基本操作，则是依赖于具体问题的，需要针对具体问题做具体的约定。在不做特殊声明时，我们假定如下操作是基本操作：两个整数的加、减、乘、除以及比较大小；判断两个整数是否相等；逻辑运算 AND、OR 和 NOT；变量的赋值、读和写。

下面我们从一个简单的例子—计算 Fibonacci 数—谈起。Fibonacci 数列是指这样的一列数：

$$0, 1, 1, 2, 3, 5, 8, 13, 21, ...$$

其中前两项是 0 和 1，后续项是其两个直接前项之和。Fibonacci 数列可以用递归的方式定义：

$$F_n = \begin{cases} 0 & \text{如果} n = 0 \\ 1 & \text{如果} n = 1 \\ F_{n-1} + F_{n-2} & \text{否则} \end{cases}$$

我们考虑如下问题：给定 n，计算 Fibonacci 数列中的第 n 项 F_n；形式化描述为：

Fibonacci 数计算问题

输入：自然数 $n \geq 0$；

输出：Fibonacci 数 F_n。

一种简单的思路是直接按照定义写成递归程序，描述如下：

Algorithm 6 计算 Fibonacci 数的递归算法

function F(n)

Require: $n \geq 0$

```
1: if n == 0 then
2:     return 0;
3: else if n == 1 then
4:     return 1;
5: else
6:     return F(n − 1)+F(n − 2);
7: end if
```

这个算法是对递归定义的忠实翻译，其正确性是毋庸置疑的。但是算法的效率却不是显而易见的。我们考虑计算 F_n 所需的**基本操作次数**，记为 $T(n)$。这里所谓的基本操作包括判断两个数是否相等（第 1 行、第 3 行）、两个数相加（第 6 行）。我们能够观察到如下两点：

(1) 当 $n \leq 1$ 时，上述算法执行最多两次基本操作，即：

$$T(n) \leq 2, \quad 当n \leq 1时。$$

(2) 当 $n > 1$ 时，上述算法需执行两次递归调用，以及 3 次基本操作，从而有：

$$T(n) = T(n-1) + T(n-2) + 3, \quad 当n > 1时。$$

很容易能够证明：$T(n) \geq F_n$。这意味着基本操作次数 $T(n)$ 随着 n 成指数式增长，比如当 $n = 20$ 时，$T(n) \geq 6765$；当 $n = 30$ 时，$T(n) \geq 832040$；当 $n = 40$ 时，$T(n) \geq 102334155$；当 $n = 50$ 时，$T(n) \geq 12586269025$。因此，除了 n 取非常小的数值之外，上述算法都会非常慢。

只需检查一个简单的例子就能够清楚看出，导致算法低效的原因在于重复计算。图 1.10展示了计算 $F(5)$ 的递归调用过程，其中 $F(3)$ 被重复计算 2 次，$F(2)$ 被重复计算 3 次。当 n 取更大的数值时，计算 $F(n)$ 时的重复计算就更多了。

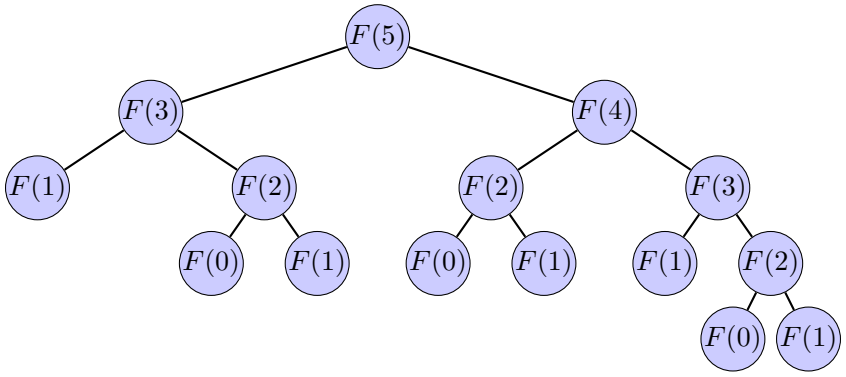


图 1.10: $F(5)$ 的递归计算过程

为避免重复计算，一种简单而有效的策略是将已经计算过的数值 F_k 存入一个表格中，以后再次计算 F_k 时直接查表即可。采用这种策略的算法描述如下：

Algorithm 7 计算 Fibonacci 数的递归算法

function FIBONACCI(n)

Require: $n \geq 0$

```
if n == 0 then
    return 0;
else if n == 1 then
    return 1;
end if
Allocate an array f[0..n];
f[0] = 0;
f[1] = 1;
for i = 2 to n do
    f[i] = f[i − 1] + f[i − 2];
end for
return f[n];
```

很明显可以看出，当 $n \geq 2$ 时，这个新的算法的基本操作次数 $T(n) = n + 5$，随 n 成线性增长。因此，我们可以说第二个算法比第一个算法更高效。

1.3.1 时间复杂度与空间复杂度

在约定了算法的基本操作之后，算法执行过程中的*总的基本操作次数*称为算法的*时间复杂度*，算法所使用的存储单元数目则称为算法的*空间复杂度*。我们很容易观察

到时间和空间复杂度受两方面因素的影响:

(1) 无论是时间复杂度还是空间复杂度，都和实例的规模密切相关：对规模较大的实例，通常需要较多的基本操作次数以及存储单元数目。例如在计算 Fibonacci 数 F_n 时，我们使用 n 来表示问题实例的规模；而对于旅行商问题而言，我们常常使用结点数目来表示实例的规模。（问题规模的精确定义涉及到实例的编码方式，是指在特定的编码方式下实例的编码长度，称为输入长度。自然，对于一个问题来说，通常可以有多种编码方式表示其实例；但是只要编码方案是合理的，则不同的编码方案下实例编码长度之间的差异至多是多项式的。我们将在第 9 章做详细描述。）

(2) 即使在规模相同的条件下，对于不同的实例，算法的时间复杂度和空间复杂度往往也会不同。以旅行商问题为例，当限定结点数 $n = 5$ 时，如果结点间距离不同，智能枚举算法的运行时间一般也会不同。

仅凭算法在一个或少数几个特定实例上的时间和空间复杂度，难以准确评估其性能。因此，常用的方式是考虑具有同等规模的所有实例，以在所有实例上基本操作次数的最大值作为时间复杂度，称为**最坏情况时间复杂度**（Worst-case time complexity）；以在所有实例上使用存储单元数目的最大值作为空间复杂度，称为**最坏情况空间复杂度**（Worst-case space complexity）。由于忽略了具体实例的影响，时间复杂度和空间复杂度只与实例规模相关，可以表示成**实例规模的函数**。以上一小节中的两个算法为例，第一个算法的时间复杂度 $T(n) \geq F(n)$，第二个算法的时间复杂度为 $T(n) = n + 5$。图 1.11展示了常见的时间复杂度函数及其之间的关系。

除了最大值之外，还有另外一种方式考虑算法在具有同等规模的所有实例上的表现：以在所有实例上基本操作次数的平均值作为时间复杂度，称为**平均情况时间复杂度**（Average-case time complexity）；以在所有实例上存储单元数目的平均值作为空间复杂度，称为**平均情况空间复杂度**（Average-case space complexity）。然而平均值的计算，需要事先已知问题实例的概率分布，从而导致这一方案很难行得通。因此在讨论时间和空间复杂度是，常常是指最坏情况时间和空间复杂度。

1.3.2 大 O 记号

当我们想比较两个算法的复杂度，或者想简化一个算法复杂度的表示时，我们可以使用大 O 符号。比如上面描述的计算 Fibonacci 数的两个算法，第一个算法

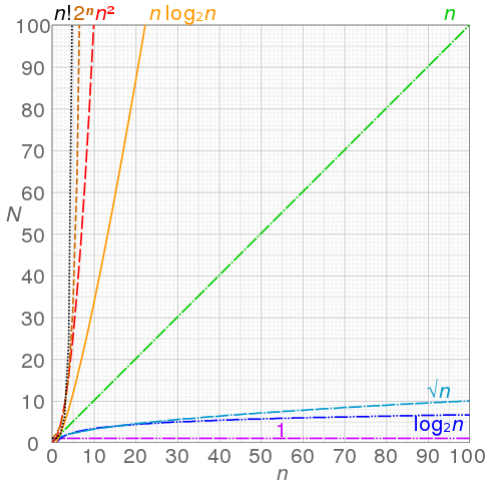


图 1.11: 常见的复杂度函数及其之间的关系。

的时间复杂度为 $T_1(n) \geq F_n$，第二个算法的时间复杂度为 $T_2(n) = n + 5$，我们说 $T_2(n) = O(T_1(n))$，其直观含义是第二个算法时间复杂度增加速度更慢，性能更好。大 O 符号的精确定义描述如下：

定义 1.3.1 (大 O 记号). 考虑两个函数 $f(n)$ 和 $g(n)$，其定义域是正整数，值域是正实数。如果存在一个正数 $c > 0$ 以及 $N > 0$，使得对任意的 $n > N$，总有 $f(n) \leq cg(n)$ 成立，则记为 $f(n) = O(g(n))$。

大 O 符号的含义类似于“$\leq$”，只是附加了两个限制：

(1) 不是简单地指 $f(n)$ 小于等于 $g(n)$，而是 $f(n)$ 小于等于 $g(n)$ 的某个常数倍；因此可以把大 O 符号看做一个隐含的常数 [?]。比如对 $f(n) = 10n$, $g(n) = n$，虽然 $10n > n$，但是我们能推导出 $10n = O(n)$。这使得我们可以忽略系数，只关注复杂度中的主要部分。

(2) 不等式只需要对充分大的 n 成立即可（也称为渐进估计）。比如 $f(n) = 10n$，$g(n) = n^2$，当 $n \leq 10$ 时，$10n > n^2$；而当 $n > 10$ 时，$10n \leq n^2$。之所以只关注 n 充分大的情况，其原因在于：当实例规模 n 很小时，算法一般都比较快，不同算法之间性能差异一般不太大，因此我们更关心当 n 增大时，复杂度的增长趋势的差异。

如同大 O 符号的含义类似于“$\leq$”，我们还可以使用类似于“$\geq$”的符号 Ω，以及类似于“$=$”的符号 Θ [?]：

定义 1.3.2 (Ω 记号和 Θ 记号). *考虑两个函数 $f(n)$ 和 $g(n)$，其定义域是正整数，值域是正实数。如果 $f(n) = O(g(n))$，则可以记为 $g(n) = \Omega(f(n))$。如果 $f(n) = O(g(n))$ 和 $g(n) = O(f(n))$ 同时成立，则可以记为 $f(n) = \Theta(g(n))$。*

下面我们先来看如何使用上述符号，比较不同算法的时间和空间复杂度。我们以下述三个算法的时间复杂度为例：

$$\begin{aligned} T_1(n) &= 100n^2 + n + 1, \\ T_2(n) &= n^2, \\ T_3(n) &= n^3. \end{aligned}$$

我们能够得到如下结论：

(1) $T_2(n) = O(T_1(n))$，表示算法 2 比算法 1 优越，但是其优越性只体现在常数倍的差异上；和 $T_1(n)$ 和 $T_3(n)$ 之间的差异比较而言，$T_1(n)$ 与 $T_2(n)$ 之间的差异微不足道。

(2) 我们同时还能推出 $T_1(n) = O(T_2(n))$，从而表示当忽略常数系数之后，$T_1(n)$ 和 $T_2(n)$ 具有相同的增长趋势，即 $T_1(n) = \Theta(T_2(n))$。

(3) $T_1(n) = O(T_3(n))$，表示算法 1 比算法 3 更高效。

接着我们来看如何使用大 O 符号来简化时间和空间复杂度的表达，获得复杂度的上界。我们可以使用如下的经验规则：

(1) 如果复杂度函数可以写成多项之和，我们可以只保留占支配地位的那一项。比如 $100n^2 + n + 1 = O(100n^2)$，其原因在于：当 n 比较大时，低次项 n 和 1 相对于高次项 $100n^2$ 来说很小，去除之后影响不大。类似地，我们有 $2^n + n^{100} = O(2^n)$，其原因在于：和 n^{100} 相比而言，2^n 占支配地位；此外，$n^{0.01} + \log n = O(n^{0.01})$，其原因在于：和 $\log n$ 相比而言，$n^{0.01}$ 占支配地位。

(2) 复杂度函数里占支配地位那一项的系数可以忽略，例如 $100n^2+n+1 = O(n^2)$。

算法时间复杂度函数多种多样，但是有一种简单分类得到普遍认可：*多项式时间算法*和*指数时间算法*。具体地，多项式时间算法定义为时间复杂度为 $O(p(n))$ 的算法，其中 $p(n)$ 表示输入长度 n 的一个多项式；时间复杂度不能表示为 $O(p(n))$ 的算法则称为指数时间算法（值得指出的是，这样定义的指数时间算法包含时间复杂度为 $O(n^{\log n})$ 的算法，虽然 $n^{\log n}$ 不是指数函数）[**?**]。

多项式时间算法和指数时间算法之间的区别，是问题的固有难度和 NP-完全问题的核心：J. Edmonds 把多项式时间算法等同于“好”的算法 [?]；如果一个问题不能用多项式时间算法求解，我们就称这个问题是“难”的。通常来说，指数时间算法常常是枚举法的变种；而往往只有在我们对问题结构有了深入了解之后，我们才能设计出多项式时间算法。我们将在第 9 章详细讨论这一点。

小结

延伸阅读

算法需求分析

1999 年，Steven S. Skiena 做了一项关于算法需求的“市场调研” [?]。他首先建立了 Stony Brook 算法库 (http://https://algorist.com/algorist.html)，涵盖 7 大类（数据结构、数值计算、组合问题、图论中的简单问题、图论中难的问题、计算几何学、集合与字符串相关问题）、75 个问题的求解算法及实现；然后他分析了访问记录，以了解访问者对哪些算法更感兴趣。统计数据表明：最短路径算法的访问量最多，旅行商问题的求解算法的访问量也很多，位列第 4 名。

新的趋势：用神经网络替代“专家经验”设计算法

在求解旅行商问题的“逐步改进”策略中，我们谈到了 2-OPT 边交换技术：挑选合适的两条边执行边交换，以逐步改善解的质量。然而选择哪些边执行边交换，始终是一个令人困扰的难题。过去基本上是凭借专家经验来选择；最近的一些研究表明，使用神经网络，结合强化学习技术，可以做出比 “专家经验”更优的选择 [?]；隋京言等进一步扩展到 3-OPT 边交换技术。

算法复杂度攻击：利用平均时间复杂度和最坏时间复杂度之间的差异

对于有些算法来说，其平均时间复杂度和最差时间复杂度情况之间存在显著差异，比如 QuickSort 算法的平均运行时间是 $O(n \log n)$，但在一些实例上可能退化为 $O(n^2)$。

算法复杂度攻击（Algorithmic complexity attack）就是利用这种差异进行攻击，即：构造一些最坏情况的实例，并设法让算法在这些实例上运行，使得算法效率显著下降，最终导致设备失能 (http://www.cs.dartmouth.edu/ doug/mdmspe.pdf) 。

算法演进 vs 硬件演进，孰快?

硬件的性能随着时间不断提升；1965 年，Intel 的创始人之一 Gordon E. Moore 受 Electronics Magazine 邀请预测未来 10 年硬件演进的趋势；为此目的，Moore 总结了 1962 年至 1965 年共 4 年的电路集成度数据，绘制了一条曲线，并用外插值技术预测未来发展趋势。Moore 发现：集成电路上可容纳的晶体管数目，大致每隔两年便会增加一倍（后被 David House 修正为每 18 个月晶体管数目翻一番，速度提升一倍）；这个经验性规律被称作摩尔定律（Moore's law）[?]。

一个有意思的问题是：既然硬件的性能呈指数级增长，那是否我们可以不改进算法，仅仅依靠硬件的进步即可解决所有的问题呢？2021 年，Yash Sherry 和 Neil C. Thompson 分析了 113 类算法 [?]，发现两点有趣的现象：

(1) 硬件的演进是平滑的（smoothly over time），而算法的进步常常是偶发的、跳跃性；

(2) 当问题规模很大时，算法进步的效果要显著高于硬件进步的效果。

因此，对于机器学习等大数据场景来说，算法的进步尤为重要。

智能枚举策略与数学定理证明

陶哲轩、波利亚

[?, ?]

习题

1. 比较如下的复杂度函数 $f(n)$ 和 $g(n)$，使用大 O 符号、Ω 记号或者 Θ 记号表示 $f(n)$ 和 $g(n)$ 之间的关系：

   (1) $f(n) = 2n^3 + 3n$, $g(n) = 100n^2 + 2n$

   (2) $f(n) = n \log n$, $g(n) = n^2$

   (3) $f(n) = \log n$, $g(n) = (\log n)^2$

   (4) $f(n) = \log n^{2 \log n}$, $g(n) = n^2$

   (5) $f(n) = n \log n$, $g(n) = n^2$

   (6) $f(n) = n!$, $g(n) = 2^n$

   (7) $f(n) = \log n$, $g(n) = \log \log n$

(8) $f(n) = n^{0.01}$, $g(n) = \log^2 n$

(9) $f(n) = n2^n$, $g(n) = 3^n$

2. 考虑求解最大公约数问题（Greatest Common Divisor, GCD）：

> **最大公约数计算问题**
>
> **输入**：两个整数 a 和 b，$a \geq b \geq 0$；
>
> **输出**：a 和 b 的最大公约数 $gcd(a, b)$。

采用 Euclid 辗转相除法，我们可以设计如下的求解算法：

Algorithm 8 Calculation of Greatest Common Divisor

function EUCLID(a, b)

Require: $a \geq b \geq 0$

1: **if** $b = 0$ **then**
2: **return** a;
3: **end if**
4: **return** EUCLID$(b, a \mod b)$;

试证明算法 EUCLID 的时间复杂度是 $O(n^3)$，此处 n 表示 a 的二进制表示中的比特数。算法的基本操作包括：检测一个数是否等于 0、一个比特位上的加法和减法。

3. 考虑如下的多项式求值问题：

> **多项式求值问题**
>
> **输入**：多项式系数 $a_0, a_1, \cdots, a_n$，实数 x；
>
> **输出**：多项式 $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ 的值。

试设计多项式求值算法，时间复杂度分别为：(i) $\Omega(n^2)$；(ii) $O(n)$。这里的基本操作包括：实数的加法和乘法。

/

第二章　“分而治之”算法

2.1　引言

给定一个问题，我们该从哪里入手设计求解算法呢?

一个朴素然而行之有效的思想是*从最简单的实例入手*，观察最简单的实例的规律，看是否可以求解。假如我们能够求解最简单的实例，那接下来求解大的实例的思考方向是：*能否将大的实例分解*（Divide）*成规模较小的实例*? *能否将小的实例的解“组合”*（Combine）*成大的实例的解*? 如果对这两个问题的回答都是“能”的话，我们就称*大的实例能够归约*（Reduction）*成小的实例*；通过连续执行归约操作，就能将原给定实例逐步归约成最简单的实例；然后再反方向操作，就能将最简单实例的解逐步“组合”成原给定实例的解。

那么，如何判断能否将大的实例分解成小的实例呢? 如果能的话，又该如何分解呢? 我们可以通过观察问题形式化描述中*输入部分的关键数据结构*来获得一些线索：一个字符串的一部分仍然是字符串、一个集合的一部分仍然是集合、一棵树去除根节点之后会形成若干子树，以及一个图的一部分是一个子图。

进一步地，如何判断能否将小实例的解“组合”成大实例的解呢? 如果能的话，又该如何组合呢? 我们可以通过观察问题形式化描述中*输出部分的形式和约束条件*来获得一些线索。

因此，要想准确地判断规模较大的实例能够归约成规模较小的实例，我们既要考虑问题描述中的“输入”部分，也要考虑“输出”部分。如果能够归约的话，我们就称这个问题具有*递归结构*，并可以设计*递归算法*进行求解。

基于归约思想的算法有很多，其典型代表是“分而治之”算法（Divide and conquer）。分而治之是一个军事术语，表示把“一个大的任务分解成小的任务”；算法领域中借用了这个名词，但是和传统用法不同的是：算法领域中的“分而治之”，要求小的实例和大的实例是同一类型的，因此通常用递归策略进行求解。

在本章里，我们将介绍“分而治之”算法的设计、正确性证明，以及时间复杂度分析。我们依据归约对象来组织本章内容，分作在数组（序列）、矩阵（二维序列）、集合等数据结构上的归约；对树和图的归约则放在下一章讲述。

值得强调的是，“分而治之”的思想适用范围很广；我们在第 5 章将会讲述如何应用“分而治之”思想计算动态规划算法的回溯路径。

2.2 排序问题：对数组的归约

在实际应用中，数组是常用的数据存储和组织方式；如何对数组中的数据进行排序，是常见的实际问题。我们以整数数组为例，对排序问题做如下的形式化描述：

> **排序问题（Sorting problem）**
> **输入**：一个包含 n 个元素的数组 $A[0..n-1]$，其中每个元素都是整数；
> **输出**：调整元素顺序后的数组 A，使得对任意的两个下标 $0 \le i < j \le n-1$，有 $A[i] \le A[j]$。

对排序问题来说，实例的规模可以用数组的大小 n 来刻画。我们先从最简单的实例入手：当 $n = 1$ 时，无需对数组 A 进行排序；当 $n = 2$ 时，我们只需对两个元素 $A[0]$ 和 $A[1]$ 进行 1 次比较、必要时执行 1 次交换操作，即可完成排序。

下面考虑规模更大的实例 $A[0..n-1]$（$n \ge 3$）。我们思考方向是如何把大的实例分解成小的实例；这些小的实例和原给定实例*形式完全相同、只是规模较小*，称为原给定实例的*子实例*（Sub-instance）。以子实例为“输入”的问题，称为原给定问题的*子问题*（Sub-problem）。对数组排序来说，子实例就是元素个数少于 n 的数组；因此将大的实例分解成子实例就是从一个大的数组中抽出一些元素，形成一个或多个小的数组。

我们可以采用两种方式将大数组拆分成小的数组：一种是基于*元素的下标*进行拆分，另一种是基于*元素的值*进行拆分。我们首先来看第一种方式。

2.2.1 依据元素下标拆分数组：插入排序与归并排序

即使是基于元素的下标，我们也有两种方案将大数组拆分成小数组，不同的拆分方案导致不同的算法，分别描述如下：

第一种拆分方案及插入排序算法

我们只需执行一个简单操作即可将数组 $A[0..n-1]$ 分解成两部分：前 $n-1$ 个元素 $A[0..n-2]$，以及单独一个元素 $A[n-1]$（见图 2.1）。前 $n-1$ 个元素组成一个小的数组，是原给定实例的子实例。

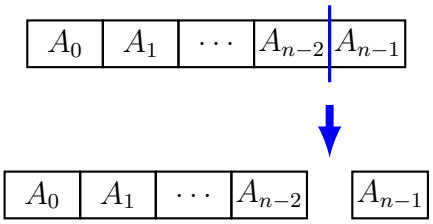


图 2.1: 依据元素下标将数组 $A[0..n-1]$ 分解成小的数组 $A[0..n-2]$ 和最后一个元素 $A[n-1]$

在将原给定实例拆分成子实例之后，我们假定子实例已被求解，下一步需要思考如何将子实例的解“组合”成原始给定实例的解。这里对子实例的求解就是“分而治之”里的“治”（Conquer），可以通过*递归调用*来完成。

对数组来说，所谓子实例的解就是已经排好序的小的数组 $A[0..n-2]$。要想完成对整个数组 $A[0..n-1]$ 的排序，我们只需将最后一个元素 $A[n-1]$ 和小数组 $A[0..n-2]$ 中的元素逐个比较，然后将 $A[n-1]$ 插入到合适的位置即可。连续应用递归调用，最终会到达*基始情形*：当 $n=1$ 时，数组 A 只有一个元素，此时无需排序，直接返回即可。

算法设计与描述

采用上述问题分解方案的算法称为*插入排序*（Insertion Sort），其伪代码描述如下：

Algorithm 9 InsertionSort algorithm

function Insertion-Sort(A, n)

```
1: if n == 1 then
2:     return ;
3: else
4:     InsertionSort(A, n − 1);
5:     key = A[n − 1];
6:     i = n − 1;
7:     while i ≥ 0 and A[i] > key do
8:         A[i + 1] = A[i];
9:         i − −;
10:    end while
11:    A[i + 1] = key;
12: end if
```

时间复杂度分析

图 2.2展示 InsertionSort 算法对 $n = 4$ 的一个数组的排序过程。由于初始数组中元素是从大到小排列的，因此在“组合”解时，共需要执行 6 次比较操作和赋值操作。我们以元素的比较作为基本操作，用 $T(n)$ 表示 InsertionSort 算法的时间复杂度，可得如下递归关系：

$$T(n) = \begin{cases} 1 & \text{如果} n = 1 \\ T(n-1) + O(n) & \text{否则} \end{cases}$$

其中 $O(n)$ 表示“组合”解的时间（在最坏情况下，第 $8-11$ 行的 `while` 循环需要执行 $O(n)$ 轮）。

为获得 $T(n)$ 上界的显式表达式，我们将上述递归表达式展开，得到如下结果：

$$\begin{aligned} T(n) &\leq T(n-1) + cn \\ &\leq T(n-2) + c(n-1) + cn \\ &\cdots\cdots \\ &\leq c + \cdots + c(n-1) + cn \\ &= O(n^2) \end{aligned}$$

此处的 c 是为了分析的方便而引入的一个常数。

InsertionSort 算法时间复杂度较高，对于大数组来说运行速度很慢，故而不

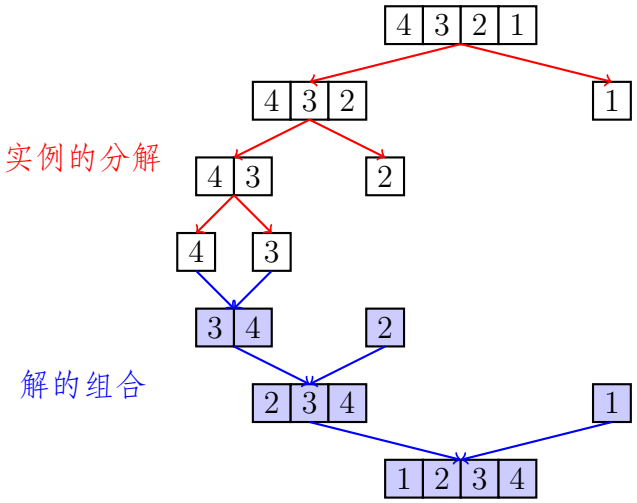


图 2.2: INSERTION-SORT 算法对 $n = 4$ 的一个数组的排序过程

实用。算法低效的原因是：在运行过程中，**问题规模是呈线性下降的**，即每次递归操作都是将规模为 n 的问题分解成一个规模为 $n-1$ 的子问题。

第二种拆分方案及归并排序算法

下面我们来看另外一种问题分解方法，能够使得**问题规模呈指数下降**。如图 2.3所示，我们可以将大的数组 $A[0..n-1]$ 按下标拆分成规模相同的两半，即 $A[0..\lceil \frac{n}{2} \rceil - 1]$ 和 $A[\lceil \frac{n}{2} \rceil..n-1]$；每一半依然是数组，形式相同，但是规模变小，因此是原给定实例的子实例。通过迭代执行分解操作，即可使得问题规模呈指数形式下降。

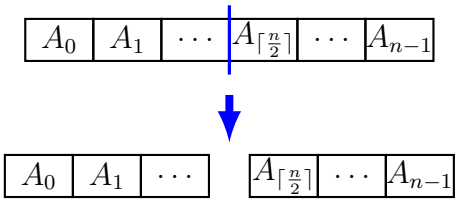


图 2.3: 依据元素下标将数组 $A[0..n-1]$ 拆分成两个小的数组：左一半 $A[0..\lceil \frac{n}{2} \rceil - 1]$ 和右一半 $A[\lceil \frac{n}{2} \rceil..n-1]$

在使用递归调用将小的数组排好序之后，我们只需依据这两个已排好序的小的数组，“归并”（Merge）出整个数组。这里的归并包括两层意思：合并、以及排序。

算法设计与描述

采用这种实例分解方式的排序算法称作“归并排序”（Merge sort），由 J. von Neumann 于 1945 年提出 [?]，其伪代码描述如下：

Algorithm 10 MergeSort 算法

function MergeSort(A, l, r)

1: **if** $l < r$ **then**
2: $m = \lceil (l + r)/2 \rceil$; //$m$ denotes the middle point
3: MergeSort(A, l, m);
4: MergeSort$(A, m + 1, r)$;
5: Merge(A, l, m, r); //Merge the sorted arrays
6: **else**
7: **return** ;
8: **end if**

上述函数是对数组 A 中下标 l 到 r 之间的元素进行排序；我们执行 MergeSort$(A, 0, n-1)$ 即可完成对整个数组的排序。连续应用递归调用，最终会到达基始情形：当 $l = r$ 时，数组 A 只有一个元素，此时无需排序，直接返回即可。归并操作由 Merge 函数完成，可以这样来直观理解：整个数组的最小元素要么是左一半 $A[l..m]$ 中的最小元素（由于左一半 $A[l..m]$ 已排好序，最小元素存放于 $A[l]$ 中），要么是右一半 $A[m+1..r]$ 中的最小元素（由于右一半 $A[m+1..r]$ 已排好序，最小元素存放于 $A[m+1]$ 中），因此只需将这两个最小元素 $A[l]$ 和 $A[m+1]$ 做比较即可。

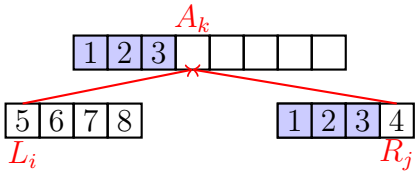


图 2.4: 归并过程：将已排好序的两个小数组 $A[l..m]$（简记为 L）和 $A[m+1..r]$（简记为 R）合并，并整理成有序数组 $A[0..n-1]$

归并过程见图 2.4，其伪代码表示如下：

Algorithm 11 对两个已排好序的子数列的归并算法

function MERGE(A, l, m, r)

Require: $A[l..m]$ (denoted as L) and $A[m+1..r]$ (denoted as R) have already been sorted

```
1: i = 0; j = 0;
2: for k = l to r do
3:     if L[i] > R[j] then
4:         A[k] = R[j];
5:         j++;
6:         if all elements in R have been copied then
7:             Copy the remainder elements from L into A;
8:             break;
9:         end if
10:    else
11:        A[k] = L[i];
12:        i++;
13:        if all elements in L have been copied then
14:            Copy the remainder elements from R into A;
15:            break;
16:        end if
17:    end if
18: end for
```

我们可以使用“循环不变量”技术（Loop invariant）证明：当 MERGE 函数中 `while` 循环结束时，数组 $A[0..n-1]$ 中所有元素都将按序排好。所谓循环不变量，是指关于程序行为的一个断言；这个断言在循环起始时成立，并且每执行一轮循环时都保持成立，因此可以推论出当循环结束时，断言必定成立。比如对于 MERGE 函数来说，我们可以验证“$A[l..k-1]$ 中包含 $A[l..r]$ 中最小的 $k-l$ 个数，且按序排好”是循环不变量。类似于循环不变量，我们还可以针对递归函数定义“归纳不变量”（Induction invariant）。通过设置合理的循环不变量或递归不变量，可以证明包含循环或递归操作的算法正确性 [**?**, **?**]。

时间复杂度分析

图 2.5展示了 MERGESORT 算法对 $n = 8$ 的一个数组的运行过程。我们注意到使用 MERGE 函数对整个数组 $A[0..n-1]$ 进行排序时，需要执行 n 次 `for` 循环（第

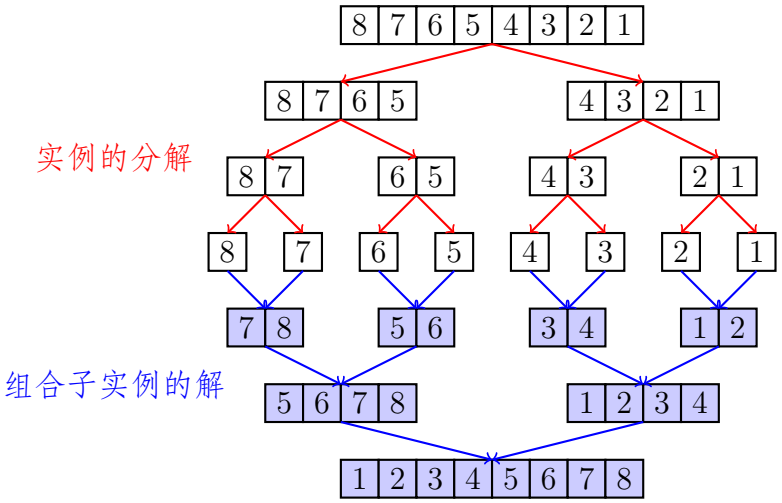


图 2.5: MergeSort 算法对 $n = 8$ 的一个数组的运行过程

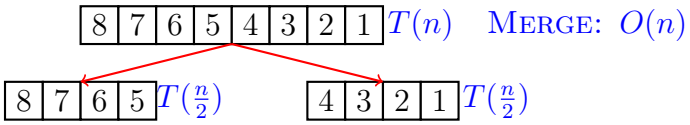


图 2.6: 归并排序算法的时间复杂度分析：以 $n = 8$ 的一个数组为例

2-10 行），需要 $O(n)$ 的时间。我们以 $T(n)$ 表示对数组 $A[0..n-1]$ 运行 MergeSort 的时间。由于两个小的数组的递归调用时间为 $2T(\frac{n}{2})$，归并时间不超过 cn（见图 2.6），从而得到如下递归表达式：

$$T(n) = \begin{cases} 1 & \text{如果} n = 1 \\ 2T(\frac{n}{2}) + O(n) & \text{否则} \end{cases}$$

为获得时间复杂度的显式表达式，我们连续应用递归表达式，生成一棵递归树。如图 2.7所示，树中每一个结点表示一个实例；每一层所有实例的归并时间累加后都是 cn，显示于右侧；树的叶子节点对应于分解到最后得到的最简单实例，累计时间为 n。由于树的高度为 $\log_2 n$，因此有：

$$T(n) = O(n \log n)$$

和插入排序算法相比，归并排序算法具有显著的优势；这种优势来源于归并时避免了一些冗余的比较和赋值操作。以图 2.5中最下面一行所示的归并为例：我们只需比较左一半的最小元 5 和右一半的最小元 1，即可知道总体的最小元为 1；由于左一半已经排好序，因此无需再将 1 和左一半的其他元素 $6, 7, 8$ 进行比较。与之形成对比的是插入排序算法的最后一次迭代，在确定了 1 的正确位置之后，还需要执行 n 次移动元素操作，才能把 1 放入正确位置。

对于基于元素比较的排序算法来说，在最坏情况下，至少要执行 $\Omega(n\log n)$ 次比较操作（如果对元素的操作不限于比较，则在某些限制之下可以在线性时间内完成排序，比如当给定 n 个整数，且最大整数 $k = n^{O(1)}$ 时，基数排序（Radix Sort）可以用 $O(n)$ 的时间完成排序）。因此，归并排序是理论上最优的基于比较的排序算法 [?]。

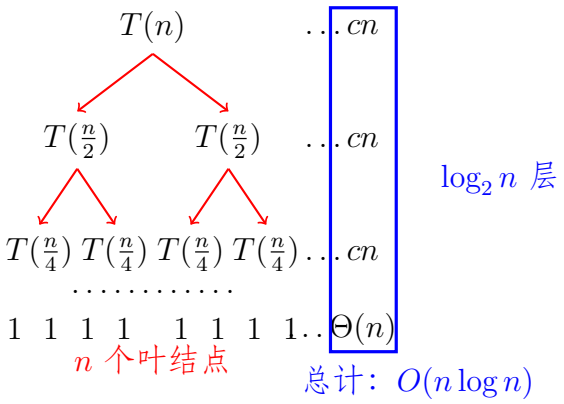


图 2.7: MergeSort 算法执行过程中的递归关系树

2.2.2 “分而治之”算法时间复杂度分析及 Master 定理

在分而治之算法中，一种常见的情况是将一个规模为 n 的实例归约成 a 个子实例，每个子实例规模都相同（设为 $\frac{n}{b}$）。假如“组合”子实例解的时间开销是 $O(n^d)$，则我们可将时间复杂度 $T(n)$ 的递归关系及基始情形表示如下：

$$T(n) = \begin{cases} 1 & \text{如果} n = 1 \\ aT(\frac{n}{b}) + O(n^d) & \text{否则} \end{cases}$$

为得到 $T(n)$ 上界的显式表达式，我们可以仿照上节所采用方式，画出递归关系树，尝试迭代展开几次，观察并总结规律，然后推导出显式的表达式。

对于子问题比较规整的情况，即每个子问题的规模都相同，$T(n)$ 上界的显式表达式已被总结成 Master 定理 [?, ?]，因此只需直接套用定理即可。Master 定理陈述如下：

定理 2.2.1. *考虑递归表达式 $T(n) = aT(\frac{n}{b}) + O(n^d)$，其中 $a > 1$, $b > 1$, $d > 0$，则 $T(n)$ 的上界可表示为：*

1. *当 $d < \log_b a$ 时，有 $T(n) = O(n^{\log_b a})$;*

2. *当 $d = \log_b a$ 时，有 $T(n) = O(n^{\log_b a}\log n)$;*

3. 当 $d > \log_b a$ 时, 有 $T(n) = O(n^d)$。

证明. 迭代应用上述递归表达式，可推导出：

$$\begin{aligned} T(n) &= aT(\tfrac{n}{b}) + O(n^d) \\ &\le aT(\tfrac{n}{b}) + cn^d \\ &\le a\big(aT(\tfrac{n}{b^2}) + c(\tfrac{n}{b})^d\big) + cn^d \\ &\le \cdots\cdots \\ &\le cn^d\big(1 + \tfrac{a}{b^d} + (\tfrac{a}{b^d})^2 + \ldots + (\tfrac{a}{b^d})^{\log_b n - 1}\big) + a^{\log_b n} \\ &= \begin{cases} O(n^{\log_b a}) & \text{如果} d < \log_b a \\ O(n^{\log_b a}\log n) & \text{如果} d = \log_b a \\ O(n^d) & \text{如果} d > \log_b a \end{cases} \end{aligned}$$

此处 c 表示一个大于 0 的常数。 □

上述证明的大意是这样的：$T(n)$ 由两项构成，一项表示递归调用时“组合”子问题解的用时，是一个等比数列之和；另一项是递归的基始情形的用时。这两项中哪一项占优，是由等比数列的比例因子 $\frac{a}{b^d}$ 决定的，分作三种情况：

(1) 当 $\frac{a}{b^d} < 1$ 时：

(2) 当 $\frac{a}{b^d} = 1$ 时：

(3) 当 $\frac{a}{b^d} > 1$ 时：

我们在此列出直接应用 Master 定理的两个例子：

- 从递归表达式 $T(n) = 3T(\frac{n}{2}) + O(n)$，可推出 $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$。
- 从递归表达式 $T(n) = 2T(\frac{n}{2}) + O(n^2)$，可推出 $T(n) = O(n^2)$。

值得指出的是，Master 定理仅适用于子实例规模都相等、且是原始实例规模的分数的情况，对于子实例规模不同等特殊情况则不适用。针对这种特殊情况，M. Akra 与 L. Bazzi 对 Master 定理进行了扩展，可以处理如下形式的递归表达式：

$$T(n) = \sum_{i=1}^{k} a_i T\big(b_i n + h_i(n)\big) + O(n^c)$$

其中 $h_i(n) = O(\frac{n}{\log^2 n})$ [**?**]。

对于规整的递归表达式，我们可以直接套用 Master 定理；而对于不那么规整的递归表达式，则可以使用“先尝试展开几级、观察总结规律”的策略，也可以采用“先猜测上界的形式、再加以证明”的策略。我们在第 3 章将会看到这种情况的例子。

2.2.3 依据元素的值拆分数组：QuickSort 算法

无论是 InsertionSort 还是 MergeSort 算法，都是依据元素的下标将大数组拆分成小数组，即选定一个元素，比这个元素下标小的元素组成一个小数组，比这个元素下标大的那些元素组成另一个小数组。

除了依据元素下标之外，我们还可以依据元素的数值将大数组拆分成小数组，即选定一个元素作“中心元”（Pivot），比中心元数值小的元素组成一个小数组，比中心元数值大的那些元素组成另一个小数组。采用这种分解方式的排序算法称为 QuickSort 算法 [?]。该算法由 C. A. R. Hoare 于 1961 年提出，其伪代码描述如下：

Algorithm 12 QuickSort algorithm

function QuickSort(A)

1: **if** $\|A\| == 1$ **then**
2: **return** ;
3: **end if**
4: Create two empty arrays S_- and S_+;
5: Choose an element $A[j]$ from A **uniformly at random** and use it as pivot;
6: **for** $i = 0$ to $\|A\| - 1$ **do**
7: Put $A[i]$ into S_- if $A[i] < A[j]$ and put $A[i]$ into S_+ otherwise;
8: **end for**
9: QuickSort(S_+);
10: QuickSort(S_-);
11: Return the concatenation of S_-, $A[j]$, and S_+ as A;

QuickSort 算法能够完成数组排序，这一点是显而易见的（第 12 行），但是时间复杂度分析却有些困难：InsertionSort 和 MergeSort 算法都依据元素下标进行分解，因此子实例的大小事先可以控制、在运行前就已知的。但是 QuickSort 算法中依据随机选择的中心元，所得到的子实例大小在算法运行前是并不知道的。

为描述方便起见，我们称排序后的数组 A 为 $\tilde{A}$，因此数组 A 的最小元是 $\tilde{A}[0]$，最大元是 $\tilde{A}[n-1]$，中位数是 $\tilde{A}[\lceil \frac{n}{2} \rceil]$。我们在选择中心元时可能面临如下两种情况：

(1) **选择数组中的最大元 $\tilde{A}[n-1]$/最小元 $\tilde{A}[0]$ 作为中心元**：这样只会生成一个子实例，规模减少了 1，呈线性降低。如果在每一次迭代都是如此选择的话，运行过程就与 InsertionSort 算法相同，时间复杂度为：

$$T(n) = T(n-1) + O(n) = O(n^2)$$

(2) **选择数组中的中位数 $\tilde{A}[\lceil \frac{n}{2} \rceil]$ 作为中心元**：这样会生成两个子实例，每个子实例的规模都是原来的一半，呈指数下降。如果在每一次迭代都是如此选择的话，运行过程就与 MergeSort 算法相同，时间复杂度为：

$$T(n) = 2T(\tfrac{n}{2}) + O(n) = O(n \log n)$$

由于 QuickSort 算法中是均匀随机地选择一个元素作为中心元，因此上述两种选择发生的概率都很小，都是 $\frac{1}{n}$。大概率的情况是既不会像第一种情况那么差，也不会像第二种情况那么好，而是和第二种情况差不多。

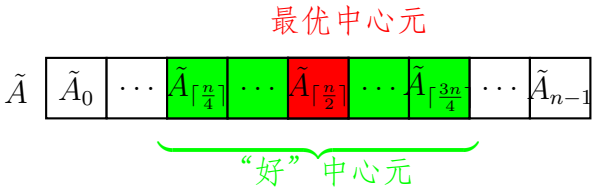


图 2.8: QuickSort 算法中的中心元的“最优选择”和“足够好”选择

详细地说，QuickSort 算法是一个随机算法，其运行过程中包含有随机行为，导致即使以同一个数组 A 作为输入，每次运行算法进行排序的执行时间也不是一个固定值，而是一个随机变量。我们将要证明**运行时间的期望值**依然是 $O(n \log n)$ 的。这个证明过程略显复杂，我们先来看一个易于分析的修正版 QuickSort 算法。

MODIFIED-QUICKSORT 算法：一个复杂度易于分析的版本

Algorithm 13 MODIFIED-QUICKSORT algorithm

function MODIFIED-QUICKSORT(A)

1: **if** $\|A\| == 1$ **then**
2: **return** ;
3: **end if**
4: **while** TRUE **do**
5: Create two empty arrays S_- and S_+;
6: Choose an element $A[j]$ from A **uniformly at random** and use it as pivot;
7: **for** $i = 0$ to $\|A\| - 1$ **do**
8: Put $A[i]$ into S_- if $A[i] < A[j]$ and put $A[i]$ into S_+ otherwise;
9: **end for**
10: **if** $\|S_+\| \geq \frac{n}{4}$ and $\|S_-\| \geq \frac{n}{4}$ **then**
11: break; //A fixed proportion of elements fall both below and above the pivot;
12: **end if**
13: **end while**
14: MODIFIED-QUICKSORT(S_+);
15: MODIFIED-QUICKSORT(S_-);
16: **return** the concatenation of S_-, $A[j]$, and S_+ as A;

和 QUICKSORT 算法相比，MODIFIED-QUICKSORT 算法只做了一点修改：随机选择一个元素做中心元之后，先检验一下这个中心元是否足够好（第 8-10 行）；如果足够好，则继续执行后续的比较和排序，否则重新选择一个元素做中心元。所谓的中心元足够好，是指它位于 $\tilde{A}$ 的中间区域，即 $\tilde{A}[\lceil \frac{n}{4} \rceil .. \lceil \frac{3n}{4} \rceil]$。直观上看，中位数 $\tilde{A}_{\lceil \frac{n}{2} \rceil}$ 是中心元的最佳选择，而 $\tilde{A}$ 中间区域的元素是中心元的足够好的选择（见图 2.8）。

之所以说 $\tilde{A}$ 中间区域的元素都是足够好的选择，是因为如下两个事实：

(1) 选中 $\tilde{A}$ 中间区域中某个元素的概率足够高：由于中间区域中共有 $\frac{n}{2}$ 个元素，因此第 3 行做一次随机选择时选中中间区域某个元素的概率是 $\frac{1}{2}$，进而可以推出算法中 `while` 循环期望执行 2 次（一个直观的类比是掷一枚均匀硬币，等待第一次掷出正面的期望次数是 2）。

注意到每次 `while` 循环里，都会将数组中的所有元素和中心元进行比较，因此递归调用之外的所有操作（即第 1-14 行）期望执行时间是 $2n$。

(2) 以 $\tilde{A}$ 中间区域中某个元素做中心元，生成子实例的规模呈指数下降：中心元的最优选择是中位数 $\tilde{A}_{\lceil \frac{n}{2} \rceil}$，能够产生两个规模都是 $\frac{n}{2}$ 的子实例；选择中间区域的元素作为中心元，所生成的子实例规模会大于 $\frac{n}{4}$，小于 $\frac{3n}{4}$；直观上看既不会太大，也不会太小。

我们用 $T(n)$ 表示期望运行时间，可以得到如下结论：

$$\begin{aligned} T(n) &\leq T(\frac{n}{4}) + T(\frac{3n}{4}) + 2n \\ &\leq \big(T(\frac{n}{16}) + T(\frac{3n}{16}) + 2\frac{n}{4}\big) + \big(T(\frac{3n}{16}) + T(\frac{9n}{16}) + 2\frac{3n}{4}\big) + 2n \\ &= \big(T(\frac{n}{16}) + T(\frac{3n}{16})\big) + \big(T(\frac{3n}{16}) + T(\frac{9n}{16})\big) + 2n + 2n \\ &\leq \cdots\cdots\cdots \\ &= O(n \log_{\frac{4}{3}} n) \end{aligned}$$

QuickSort 算法时间复杂度分析

刚才我们分析了改动后的 ModifiedQuickSort 算法的时间复杂度，接下来我们分析 QuickSort 算法的时间复杂度。在做具体的分析之前，我们先陈述关于运行时间的 3 点事实：

(1) *运行时间由比较次数界定*：如果把所有的递归操作都展开的话，很明显算法的总运行时间由第 7 行的比较操作的总次数确定。我们将比较操作的总次数记为 X。

值得指出的是，QuickSort 算法是个随机算法：第 5 行是随机选择一个元素 $A[j]$ 作为中心元；这种随机选择导致算法的两次执行过程可能不一样（虽然最终结果都把数组排好序）、算法的比较操作的总次数 X 也可能不一样。在这种情况下，我们只好估计 X 的最大值或者期望值；不过和 X 的最大值相比，期望值 $\mathbb{E}[X]$ 更具有实际意义。

那如何计算定义 X 的期望值 $\mathbb{E}[X]$ 呢？

我们先来尝试按照期望值的定义进行计算。由期望值的定义可知：

$$\mathbb{E}[X] = 1 \times \Pr[X = 1] + 2 \times \Pr[X = 2] + 3 \times \Pr[X = 3] + \cdots \qquad (2.2.1)$$

不过随机变量 X 比较负责，概率 $\Pr[X = 1]$，$\Pr[X = 2]$，$\Pr[X = 3]$，$\cdots$ 不好计算，导致这种方式不可行。因此，我们换一种方式：*把 X 拆分，表示成一些简单的随机变量之和*，具体拆分方式见下。

(2) **任意两个元素 $\tilde{A}[i]$ 和 $\tilde{A}[j]$ 最多只会比较一次**：如图 2.9所示，只有当 $\tilde{A}[i]$ 或 $\tilde{A}[j]$ 被选做中心元时，$\tilde{A}[i]$ 才会和 $\tilde{A}[j]$ 进行比较；一旦比较完成之后，$\tilde{A}[i]$ 和 $\tilde{A}[j]$ 不会同时出现在 S_- 或者 S_+ 中，因此不会再次进行比较。

因此我们可以定义如下的**指示变量**（Index variable）：

$$X_{ij} = \begin{cases} 1 & \text{如果元素}\tilde{A}[i]\text{和}\tilde{A}[j]\text{发生比较} \\ 0 & \text{否则} \end{cases}$$

进而将 X 拆分，表示成随机变量之和，即：$X = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}$。

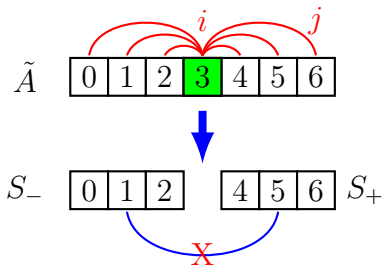


图 2.9: 在 QuickSort 算法执行过程中，$\tilde{A}[i]$ 和 $\tilde{A}[j]$ 最多只会比较一次

(3) **两个元素 $\tilde{A}[i]$ 和 $\tilde{A}[j]$ $(0 \le i < j \le n-1)$ 发生比较的概率是 $\frac{2}{j-i+1}$**：对这个事实的证明我们稍后陈述。

基于上述事实，我们立刻能够证明如下定理：

定理 2.2.2. QuickSort **算法的期望运行时间是** $\mathbb{E}[X] = O(n \log n)$。

怎样估计 $\mathbb{E}[X]$ 的大小呢？

证明. 由于 $X = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}$，我们有：

$$
\begin{aligned}
\mathbb{E}[X] &= \mathbb{E}[\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} X_{ij}] \\
&= \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} \mathbb{E}[X_{ij}] \\
&= \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} \Pr[\tilde{A}[i]\text{和}\tilde{A}[j]\text{进行比较}] \\
&= \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} \frac{2}{j-i+1} \\
&= \sum_{i=0}^{n-1}\sum_{k=1}^{n-i-1} \frac{2}{k+1} \\
&\leq \sum_{i=0}^{n-1}\sum_{k=1}^{n-1} \frac{2}{k+1} \\
&= O(n\log n)
\end{aligned}
$$

此处 k 定义为 $k=j-i$。 □

现在我们只遗留了一个疑问：为什么在 QuickSort 算法执行过程中，任意两个元素 $\tilde{A}[i]$ 和 $\tilde{A}[j]$ $(0\leq i<j\leq n-1)$ 发生比较的概率都是 $\frac{2}{j-i+1}$，并且和数组的大小无关呢？我们以 $i=0, j=2$ 为例，证明元素 $\tilde{A}[0]$ 和 $\tilde{A}[2]$ 比较的概率是 $\frac{2}{3}$。我们首先来看两个简单的情况：

(1) 数组只包含 3 个元素的情况：

如图 2.10所示，当 $\tilde{A}[0]$ 或 $\tilde{A}[2]$ 被选作中心元时，会发生 $\tilde{A}[0]$ 和 $\tilde{A}[2]$ 的比较，故有：

$$\Pr[\tilde{A}[0]\text{和}\tilde{A}[2]\text{进行比较}] = \frac{2}{3}$$

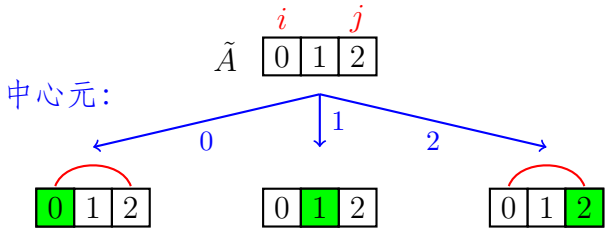


图 2.10: 对包含 3 个元素的数组 A 运行 QuickSort 算法排序时，当且仅当 $\tilde{A}[0]$ 或 $\tilde{A}[2]$ 被选做中心元时，会进行 $\tilde{A}[0]$ 和 $\tilde{A}[2]$ 的比较，因此进行比较的概率是 $\frac{2}{3}$

(2) 数组包含 4 个元素的情况：

如图 2.11所示，$\tilde{A}[0]$ 和 $\tilde{A}[2]$ 发生比较有两种可能：(i) 当 $\tilde{A}[0]$ 或 $\tilde{A}[2]$ 被选做中心元时，在本轮迭代即进行比较；(ii) 当 $\tilde{A}[3]$ 被选做中心元时，$\tilde{A}[0], \tilde{A}[1], \tilde{A}[2]$

被归入小数组 S_-；在对 S_- 执行下一轮迭代时，可能发生比较。我们已经证明，对于包含 3 个元素的数组 S_- 运行算法时，$\tilde{A}[0]$ 和 $\tilde{A}[2]$ 进行比较的概率是 $\frac{2}{3}$。

综合这两种可能，我们有：

$$\Pr[\tilde{A}[0]和\tilde{A}[2]进行比较] = \frac{1}{4} + \frac{1}{4} + \frac{1}{4} \times \frac{2}{3} = \frac{2}{3}.$$

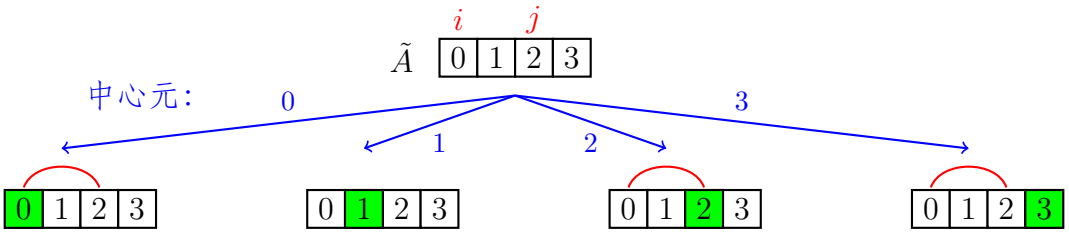


图 2.11: 对包含 4 个元素的数组 A 运行 QuickSort 算法排序时，$\tilde{A}[0]$ 和 $\tilde{A}[2]$ 发生比较有两种可能：(i) 当 $\tilde{A}[0]$ 或 $\tilde{A}[2]$ 被选做中心元时，在本轮迭代即进行比较；(ii) 当 $\tilde{A}[3]$ 被选做中心元时，$\tilde{A}[0], \tilde{A}[1], \tilde{A}[2]$ 被归入小数组 S_-，并对 S_- 执行下一轮迭代；在下一轮迭代时可能发生比较，发生概率是 $\frac{2}{3}$

上述观察完全可以推广至数组包含 n 个元素的一般情况，即：

$$\begin{aligned}
\Pr[\tilde{A}[i]与\tilde{A}[j]进行比较] &= \frac{1}{n} + \frac{1}{n} + \frac{n-(j-i+1)}{n} \times \frac{2}{j-i+1} \\
&= \left(\frac{j-i+1}{n} + \frac{n-(j-i+1)}{n}\right) \times \frac{2}{j-i+1} \\
&= \frac{2}{j-i+1}
\end{aligned}$$

从而证明了两个元素 $\tilde{A}[i]$ 和 $\tilde{A}[j]$ 发生比较的概率是 $\frac{2}{j-i+1}$。这个概率仅与这两个元素的“排名之差”相关，而和数组的大小 n 无关：两个元素的排名相差越大，发生比较的概率就越小；反之则越大。

一个有意思的特殊情况是：对于排名相邻的两个元素来说，它们之间必定会比较一次。图 2.12展示一个例子：$\tilde{A}[0]$ 和 $\tilde{A}[1]$ 之间比较的概率是：

$$\Pr[\tilde{A}[0]和\tilde{A}[1]进行比较] = \frac{1}{3} + \frac{1}{3} + \frac{1}{3} \times 1 = 1.$$

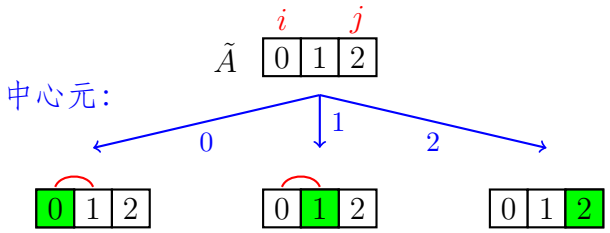


图 2.12: 对包含 3 个元素的数组 A 运行 QuickSort 算法排序时，排名相邻的两个元素，比如 $\tilde{A}[0]$ 与 $\tilde{A}[1]$，必定会被比较一次

降低 QuickSort 算法空间复杂度的努力：“原位”排序算法

上一小节所述的 QuickSort 算法的时间复杂度很小，但是需要创建两个辅助数组 S_- 和 S_+，这样一来，除了数组本身之外还要额外占用 n 个内存单元，导致当 n 比较大时，内存需求有时难以满足。因此，无需开辟额外的辅助空间、仅使用数组 A 所占的空间进行“原位”（In-place）排序，是很有实际价值的工作。

N. Lomuto 和 C. A. R. Hoare 各自提出了一种“原位”QuickSort 算法；我们在此仅描述 Lomuto 的方法 [?, ?]，Hoare 的方法请见文献 [?, ?, ?]。

为避免开辟辅助数组 S_- 和 S_+，Lomuto 算法直接将数组 A 的左半部分当做 S_-，存放比中心元小的元素；把数组 A 的右半部分当做 S_+，存放比中心元大的元素（见图 2.13）。

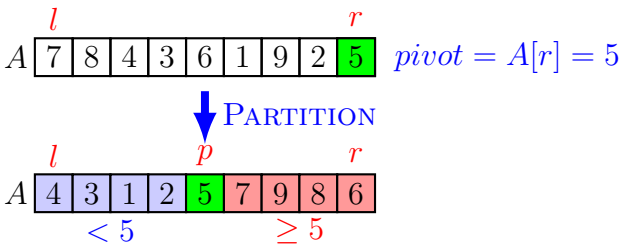


图 2.13: Lomuto 快速排序算法中使用的 Partition 函数运行结果示例

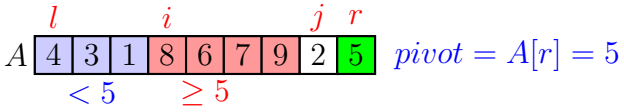


图 2.14: Lomuto 快速排序算法中使用的 Partition 函数运行过程示例：当发现 $A[j]$ 比中心元小时，则交换 $A[j]$ 与 $A[i]$。此处 i 记录比中心元大的元素所在区域的最小下标

算法设计与描述

数组分解这一步，Lomuto 算法是通过函数 Partition 来完成的，其关键操作是“交换”，即：以 $A[r]$ 作为中心元，比中心元小的元素放在 $A[l..i-1]$ 区域，比中心元大的元素放在 $A[i..j-1]$ 区域；下标 j 从 l 到 r，逐个检验元素 $A[j]$：若 $A[j]$ 比中心元小，则交换 $A[j]$ 和 $A[i]$；最后将中心元放入正确的位置。

Lomuto 算法的伪代码描述如下：

Algorithm 14 Lomuto-QuickSort algorithm

function Lomuto-QuickSort(A, l, r)

```
1: if l < r then
2:     p =Partition(A, l, r) //Use A[r] as pivot;
3:     Lomuto-QuickSort(A, l, p − 1);
4:     Lomuto-QuickSort(A, p + 1, r);
5: end if
```

function Partition(A, l, r)

```
1: pivot = A[r]; i = l;
2: for j = l to r − 1 do
3:     if A[j] < pivot then
4:         Swap A[i] with A[j];
5:         i + +;
6:     end if
7: end for
8: Swap A[i] with A[r]; //Put pivot in its correct position
9: return i;
```

算法性能比较

C. A. R. Hoare 做了一个实验，比较了 MergeSort 和 QuickSort 的性能。如表 2.1所示，对越大的数组，QuickSort 算法的性能优势越显著。由于 Hoare 使用的 405 型计算机内存大小有限，表格中的带 “*” 的数据是 Hoare 使用公式推算出来的。

还应该说明的是：迄今为止我们都是假设数组中的元素各不相同；当数组中存在重复元素时，上述算法性能不太好。为克服这种缺陷，一种简单的改进方式是在分解数组时，不是简单地分成两个小数组 S_- 和 S_+，而是分成三个小数组：比中心元小的放入 S_-，比中心元大的放入 S_+，以及和中心元相等的元素。此外，归并排序算法

表 2.1: MergeSort 算法和 QuickSort 算法性能比较 [?]

	运行时间	
数组大小 n	MergeSort 算法	QuickSort 算法
500	2 min 8 sec	1 min 21 sec
1000	4 min 48 sec	3 min 8 sec
1500	8 min 15 sec*	5 min 6 sec
2000	11 min 0 sec *	6 min 47 sec

的一个优势是其“稳定性”，即重复元素的原始顺序在归并操作中会被保留。

2.3 一个密切相关的问题：数组中的逆序对计数

和数组排序密切相关的一个问题是：如何对数组中的“逆序对”进行计数。所谓逆序对，是指数组中的两个元素 $A[i]$ 和 $A[j]$，其下标 $i < j$，但是考察元素的值，却有 $A[i] > A[j]$。比如数组 $A = [2, 4, 1, 3, 5]$ 中，共存在 3 个逆序对，即 2 和 1、4 和 1、4 和 3。逆序对计数问题可形式化描述如下：

逆序对计数问题（Inversion-counting problem）
输入：一个包含 n 个元素的数组 $A[0..n-1]$;
输出：数组中的逆序对的数目。

数组的逆序对计数是一项基本运算，有着广泛的应用。例如在衡量两个变量的相关程度时，可以使用 Spearman 系数 ρ 和 Kendall 系数 τ 来衡量“秩”相关程度（Rank correlation），其中 Kendall 系数 τ 可以归结为逆序数的计算 [?]。

依据逆序对的定义，我们使用两重 `for` 循环检查数组中所有的元素对，即可计算出逆序对的数目。这个算法的时间复杂度是 $O(n^2)$，当 n 比较大时速度较慢。下面我们介绍一种基于分而治之思想的高效算法。

我们依然从最简单的实例入手：如果数组 A 只有两个元素 $A[0]$ 和 $A[1]$，我们只需比较这两个元素，即可计算出逆序数。

接下来我们考虑如何求解规模更大的实例。对于一个包含 n 个元素的数组 A，我们可以很容易地依据下标将 A 分解成两个小的数组，即左一半 $A[0..\lceil \frac{n}{2} \rceil - 1]$（简记为 L）和右一半 $A[\lceil \frac{n}{2} \rceil ..n-1]$（简记为 R）。在将大的实例分解成子实例之后，我们可

以假定子实例已经求解，即使用递归调用分别求出两个元素都在 L 中的逆序对数目、以及两个元素都在 R 中的逆序对数目；因此只剩下最后一个困难：如何将子实例的解“组合”成原始给定实例的解。

要想组合出原始给定实例的解，我们需要对一个元素在 L、另一个元素在 R 中所构成的逆序对进行计数。一种直接的计数方法如图 2.15（左）所示：检查所有的一个在 L、一个在 R 的元素对，对其中出现的逆序对进行计数，需要做 $\frac{n^2}{4}$ 次比较，从而导致整个算法的时间复杂度是：

$$T(n) = 2T(\tfrac{n}{2}) + \tfrac{n^2}{4} = O(n^2)$$

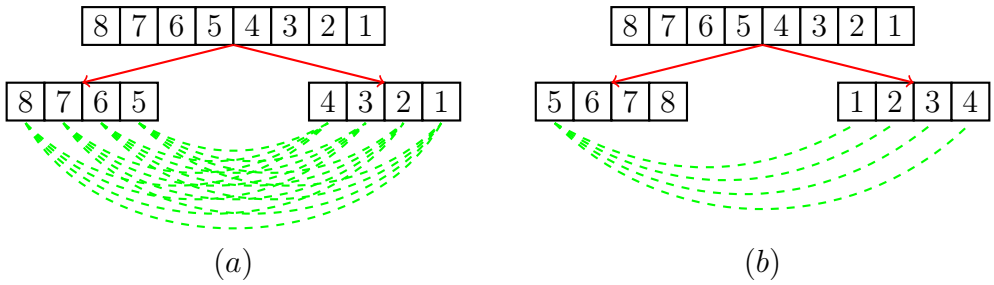


图 2.15: 逆序对计数算法中计算一个元素在左一半、另一个元素在右一半中的逆序对数目的两种策略：(a) 当 L 和 R 中元素无结构时，我们只能直接进行比较，共需 $\frac{n^2}{4}$ 次；(b) 当 L 和 R 已排好序之后，只需 $O(n)$ 次比较即可完成

那么能否更高效地组合出原始实例的解呢？这里的一个关键性的思考是：如果 L 和 R 中元素是任意的、没有任何结构的，我们没有别的办法，只能采用上述逐对检查策略；因此，要想实现高效的逆序对计数，我们必须在 L 和 R 中引入有利于逆序对计数的结构。

那么，引入哪些结构会有利于逆序对计数呢？我们注意到：如果 L 和 R 都已经排好序的话，只需执行 $O(n)$ 次比较即可完成逆序对的计数。如图 2.15所示，当 L 中元素 L_i 比 R 中元素 R_j 小时，意味着 L_i 比 $R_{j+1}, R_{j+2}, ..., R_{n-1}$ 都小，从而不构成逆序；反之，则意味着 $L_i, L_{i+1}, ..., L_{\frac{n}{2}-1}$ 都比 R_j 要大，从而构成 $\frac{n}{2} - i$ 个逆序对。

基于上述思想，我们可以设计如下的逆序对计数算法：

Algorithm 15 InversionCounting algorithm

function Sort-and-Count-Inversion(A)

```
if ||A|| == 1 then
    return ;
end if
Divide A into two sub-sequences L and R;
(RC_L, L) = Sort-and-Count-Inversion(L);
(RC_R, R) = Sort-and-Count-Inversion(R);
(C, A) = Merge-and-Count-Inversion(L, R);
return (RC = RC_L + RC_R + C, A);
```

function Merge-and-Count-Inversion (L, R)

```
RC = 0; i = 0; j = 0;
for k = 0 to ||L|| + ||R|| − 1 do
    if L[i] > R[j] then
        A[k] = R[j];
        j++;
        RC+ = (||L|| − i);
        if all elements in R have been copied then
            Copy the remainder elements from L into A;
            break;
        end if
    else
        A[k] = L[i];
        i++;
        if all elements in L have been copied then
            Copy the remainder elements from R into A;
            break;
        end if
    end if
end for
return (RC, A);
```

时间复杂度分析

由于归并步骤的时间复杂度是 $O(n)$ 的，因此整个算法的时间复杂度是：

$$T(n) = 2T(\tfrac{n}{2}) + O(n) = O(n \log n).$$

相对于简单的两两比较算法而言，采用了分而治之技术的算法的效率显著提升；其根本原因在于避免了大量的冗余比较：如图 2.15所示，当 L 和 R 中元素未排序时，我们不得不对所有的元素对进行比较，比如 R 中的元素 1 要和 L 中的所有元素 $5,6,7,8$ 进行比较；而当 L 和 R 中元素已经排好序之后，我们比较 R 中的 1 和 L 中的 5 之后，就无需再和 R 中其他元素 $6,7,8$ 进行比较，从而避免了冗余的比较操作。

2.4 选择问题：对数组的归约

在实际应用中，一个常见的问题是：如何从数组中快速找出中位数；扩展到一般的情形，是如何从数组中找出第 k 小的数。这个问题被称作*选择问题*（Selection problem），形式化描述如下：

选择问题（Selection problem）
输入：一个包含 n 个元素的数组 $A[0..n-1]$，以及一个整数 k，$(0 \leq k \leq n-1)$；
输出：数组 A 中第 k 小的元素。

求解选择问题的一种自然的思路是先对数组 A 的元素从小到大进行排序，即可找出第 k 小的元素。这个算法的时间复杂度是 $O(n \log n)$；而使用分而治之技术，能够设计出线性时间复杂度的算法。

应用*分而治之*技术的首要步骤是确定如何将大的数组拆分成小的数组。如果依据*元素下标*拆分数组的话，将面临如下困难：我们难以确定待寻找的目标数（整体第 k 小的数）会出现在拆分形成的哪个小数组里，更难以确定目标数是小数组里的第几小的数，从而无法进行递归。因此，我们只好采用依据*元素值*拆分数组的方案，用伪代码描述如下：

Algorithm 16 求解 SELECTION 问题的分而治之算法的通用框架

function SELECT(A, k)

```
1: Choose an element A_i from A as a pivot;
2: S_+ = [ ]; S_- = [ ];
3: for all element A_j in A except A_i do
4:     Put A[i] into S_- if A[i] < A[j] and put A[i] into S_+ otherwise;
5: end for
6: if ||S_-|| = k − 1 then
7:     return A_i;
8: else if ||S_-|| > k − 1 then
9:     return SELECT(S_-, k);
10: else
11:    return SELECT(S_+, k − ||S_-|| − 1);
12: end if
```

上述伪代码和 QUICKSORT 算法很类似，但是在递归调用时有所差异：QUICKSELECT 算法只对一个子实例进行递归调用，而 QUICKSORT 算法是对两个子实例都进行递归调用。

之所以说上述伪代码描述的是一般性框架，是因为第 1 行代码中未指定如何确定中心元。依据我们在数组排序问题中所获得的经验，一个“足够好”的中心元应该使得子实例的大小呈指数性减少，即：应该选择 A“中间区域”的元素作为中心元。

这样就导致一个“循环”：我们的目的是找到 A 中的第 k 小的元素（一个特例是找 A 的中位数），而完成这一目标依赖于找到 A 中的“中间区域”的元素作中心元。那么，如何打破这种循环呢？

打破这种循环的一种行之有效的策略是构造 A 的“一个近似”。我们将考察如下 3 种近似，以及相应的中心元选择方法：*i*) 先将 A 中的元素进行分组，然后以“分组的中位数”作为 A 的近似，进而选择“分组中位数的中位数”作为中心元；*ii*) 对 A 中元素进行随机采样，作为 A 的近似，进而依据随机样本的统计量来确定中心元；*iii*) 随机选择 A 中的一个元素作为中心元。

2.4.1 选择“分组中位数的中位数”作为中心元

首先我们来看构造 A 的近似的第一种策略：将数组 A 分组，然后以组中位数作为 A 的近似；进而以组中位数的中位数作为中心元。下一小节将要讲述以随机采样

作为 A 的近似；与之对比，这种“以**组中位数**作为 A 的近似”可以不严格地称为**确定性采样策略**。

之所以说**组中位数**可以作为 A 的一个良好近似，是因为组中位数的中位数和 A 的中位数非常接近。如表 2.2所示，对于一个包含 $n = 55$ 个元素的数组 A：

$$A = [51, 10, 24, 9, 5, 40, 30, 26, 25, 21, 15, 12, 7, 2, 0, 13, 11, 6, 28,$$
$$23, 43, 27, 45, 16, 3, 34, 37, 39, 31, 14, 32, 33, 53, 19, 17, 4, 35,$$
$$41, 47, 20, 8, 44, 18, 48, 52, 1, 36, 38, 50, 46, 22, 42, 54, 49, 29],$$

我们将每 5 个元素分作一组，共 11 组；各组中位数的中位数是 32，与整个数组 A 的中位数 27 非常接近。

进一步讲，以**组中位数的中位数**作为中心元，能够导致子实例的大小以指数形式下降。如表 2.2所示，比中心元大的元素至少有 $3\lceil\frac{n}{10}\rceil - 1 = 17$ 个（记为 S_+，标为红色），比中心元小的元素也至少有 $3\lceil\frac{n}{10}\rceil - 1 = 17$ 个（记为 S_-，标为蓝色）。进一步可以推出，S_+ 和 S_- 都至多包含 $7\lfloor\frac{n}{10}\rfloor + 2$ 个元素；因此，下一步无论是对 S_+ 还是 S_- 进行递归，都能使得子问题规模以指数形式下降。

表 2.2: BFPRT 算法中的元素分组，以及“分组中位数的中位数”的计算过程示例。为直观起见，我们按组中位数的大小排放各组

组 3	组 1	组 4	组 2	组 5	组 7	组 6	组 8	组 10	组 11	组 9
0	5	6	21	3	17	14	4	1	22	8
2	9	11	25	16	19	31	20	36	29	18
7	10	13	26	27	**32**	34	35	38	42	44
12	24	23	30	43	33	37	41	46	49	48
15	51	28	40	45	53	39	47	50	54	52

值得指出的是，R. C. Singleton 采用非常类似的思想选择中心元：以数组的第一个元素 A_0、最后一个元素 A_{n-1}，和中间元素 $A_{\lceil\frac{n}{2}\rceil}$ 作为样本，以这 3 个元素的中位数作中心元。这种中心元选择方法被称作“**三元取中法**”(Median of three)。实验结果表明采用这种策略，能够将比较次数减少 14%，但是需要额外的找中位数的时间开销 [?, ?]。

算法设计与描述

采用上述中心元选择策略的算法由 M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest 和 R. E. Tarjan 于 1973 年提出，被称作 BFPRT 算法 [?]（伪代码描述见算法 17）。

Algorithm 17 求解选择问题的 BFPRT 算法

function SELECT(A, k)

1: Line up elements in groups of 5 elements;
2: Find the median of each group; //Cost $\frac{6}{5}n$ time
3: Find the median of medians (denoted as M) through recursively running SELECT over the group medians; //$T(\frac{n}{5})$ time
4: Use M as pivot to partition A into S_- and S_+; //$O(n)$ time
5: **if** $|S_-| = k - 1$ **then**
6: **return** M;
7: **else if** $|S_-| > k - 1$ **then**
8: **return** SELECT(S_-, k); //at most $T(\frac{7}{10}n)$ time
9: **else**
10: **return** SELECT($S_+, k - |S_-| - 1$); //at most $T(\frac{7}{10}n)$ time
11: **end if**

时间复杂度分析

算法中第 3 行采用递归调用计算“分组中位数的中位数”，时间复杂度是 $T(\frac{n}{5})$；第 8 行和第 10 行的递归调用只会执行一个，时间复杂度是 $T(7\lfloor\frac{n}{10}\rfloor)$。因此，整个算法的时间复杂度是：

$$T(n) \leq T(\tfrac{n}{5}) + T(\tfrac{7n}{10}) + O(n) = O(n).$$

更细致的分析表明，算法总共需要的比较次数不超过 $24n$ 次。

一个很自然的问题是：BFPRT 算法中每 5 个元素分作一组，那么分组大小能否设置成其他值呢？很容易证明，每 7 个元素分作一组，也能得线性时间复杂度的算法；但是当每 3 个元素分作一组时，我们有：

$$T(n) \leq T(\tfrac{n}{3}) + T(\tfrac{2n}{3}) + O(n) = O(n \log n).$$

这意味着算法时间复杂度不是线性的（其原因在于两个子问题的规模之和 $\frac{n}{3} + \frac{2n}{3} = n$，而当每 5 个元素分成一组时，两个子问题的规模之和 $\frac{n}{5} + \frac{7n}{10}$ 严格小于 n）。此外，在上述分析中，我们既难以做出更紧的估计，也难以找到反例。

曾刚等 [?] 在此方向上取得了很好的进展；这个进展是基于对 BFPRT 算法运行过程的观察：如果 BFPRT 算法上一轮递归调用的目标是选择实例“中间区域”内的某个元素，则下一轮递归调用的目标就会是选择子实例中靠近两端的某个元素。以表2.2为例，假如第一轮是在 55 个数中找第 28 大的数（$28/55 = 0.51$，非常接近 0.50），则第二轮就变成在 31 个数中找第 28 大的数，是非常靠近端点的（$28/31 = 0.90$，非常接近 1）。我们用一句话形象地概括上述现象：‘*上一次目标居中，则下一次目标会靠边*。

对于中心元来说，理想的选取方案是：当目标靠近两端时，应该相应地选择靠近两端的元素作中心元，这样有希望减少迭代次数。从这个角度来重新检视 BFPRT 算法，我们会发现其不足：无论要找的目标数是在数组的中间还是两边，BFPRT 算法始终是选择“组中位数的中位数”—一个中间区域的数—作为中心元。

根据上述观察和分析，曾刚等改进了 BFPRT 算法，不是每次都采用“组中位数的中位数”作为中心元，而是依据待选择的目标数在数组中的位置，分别采用“组中位数”的 $\frac{1}{4}$ 分位数、中位数，或者 $\frac{3}{4}$ 分位数作为中心元，并证明了改进后的算法依然具有线性时间复杂度。

2.4.2 依据随机样本的统计量确定中心元

要构造数组 A 的近似，一个直观的想法是对 A 进行随机采样，以随机采样近似 A；进而*利用样本的统计量来确定中心元*。比如我们可以采用样本中位数作为中心元，由于样本中位数是总体中位数的无偏估计，因此能够高概率地保证选择出的中心元位于“中间区域”。此外，我们还可以做一个扩展：不是只使用样本的中位数，而是使用 $\frac{1}{4}$ 分位数和 $\frac{3}{4}$ 分位数，从而将*点估计*扩展成*区间估计*，使得待寻找的目标高概率地落在某个区间里。

算法设计与描述

1973 年，R. Floyd 和 R. Rivest 提出了首个基于随机样本确定中心元的算法，称做 Floyd-Rivest 选择算法 [?]；这里我们介绍一个更易于分析的版本（伪代码描述见算法 18）。

图??展示了 Lazy-Select-Median 算法运行过程的一个例子：对于包含 16 个元素的数组 A，我们设置 $r = 8, \delta = \frac{1}{2}$。我们首先随机采样出 8 个元素，构成样本 S；然后计算出 S 的分位数 $u = 4$ 和 $v = 13$；接着对每个元素都和 u 和 v 进行比较，依

Algorithm 18 FLOYD-RIVEST 算法的一个易于分析的版本

function LAZY-SELECT-MEDIAN(A, δ, r)

1: Randomly sample r elements (with replacement) from A, and denote the sample as S.

2: Sort S. Let u be the $\frac{1-\delta}{2}r$-th smallest element and v be the $\frac{1+\delta}{2}r$-th smallest element of S.

3: Divide A into three dis-joint subsets: $L = \{A_i : A_i < u\}$, $M = \{A_i : u \leq A_i \leq v\}$, $H = \{A_i : A_i > v\}$.

4: Check the following constraints of M:

- M covers the median: $|L| \leq \frac{n}{2}$ and $|H| \leq \frac{n}{2}$
- M should not be too large: $|M| \leq c\delta n$

If one of the constraints was violated, got to STEP 1.

5: Sort M and return the $(\frac{n}{2} - |L|)$-th smallest of M as the median of A.

据比较结果分别放入集合 L, M 和 H；最后将 M 排序，返回其第 5 个数字 8，即为数组 A 的中位数。

时间复杂度分析

下面我们来证明，当设置 $r = n^{\frac{3}{4}}$, $\delta = n^{-\frac{1}{4}}$ 时，算法的期望时间复杂度是 $O(n)$ 的。

直观地说，为了使得第 5 行的排序时间不会太长，我们需要控制 $\|M\|$ 不能太大：注意到当 $\|M\| \leq n^{\frac{3}{4}}$ 时，第 5 行排序只需 $O(n^{\frac{3}{4}} \log n) = O(n)$ 的时间（$\log n$ 比 $n^{\frac{1}{4}}$ 量级要小）。在此，我们设置 $\delta = n^{-\frac{1}{4}}$，从而使得位于 $[u, v]$ 区间的 S 中元素共有 $n^{-\frac{1}{4}} \times r$ 个，因此可以期望位于 $[u, v]$ 区间的 A 中元素共有 $n^{-\frac{1}{4}} \times n = n^{\frac{3}{4}}$ 个。

另一方面，我们可以证明：当设置 $r = n^{\frac{3}{4}}$ 时，$\|M\|$ 也不会太小，从而 M 包含中位数的概率比较大。

定理 2.4.1. *对于一个包含 n 个元素的数组 A，当设置 $r = n^{\frac{3}{4}}$, $\delta = n^{-\frac{1}{4}}$ 时，*LAZY-SELECT-MEDIAN *算法以 $1 - O(n^{-\frac{1}{4}})$ 的概率在第一轮结束，且第一轮的运行时间为 $2n + o(n)$。*

证明. 算法第一轮有两种失败情形（M 不包含中位数，以及 $\|M\|$ 太大），导致进行下一轮迭代。我们可以证明这两种情形发生的概率都足够小，是 $O(n^{-\frac{1}{4}})$ 的。下面给出第一种情形的证明，第二种情形的证明从略。

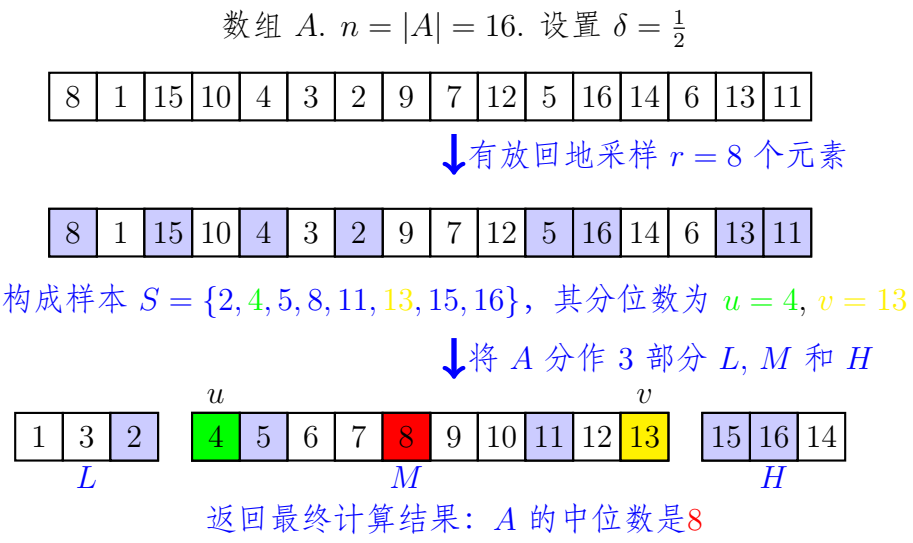


图 2.16: Lazy-Select-Median 算法运行过程的一个例子

具体地，我们来证明 $\Pr[\|L\| > \frac{n}{2}] = O(n^{-\frac{1}{4}})$。注意到 $\|L\| > \frac{n}{2}$ 意味着 u 比 A 的中位数大，从而 S 中至少有 $\frac{1+\delta}{2}r$ 个元素比中位数大。

令 $X = x_1 + x_2 + ...x_r$ 表示 S 中比 A 的中位数大的元素的个数，其中 x_i 是指示变量，定义为 $x_i = \begin{cases} 1 & \text{如果第}i\text{个样本比中位数大} \\ 0 & \text{否则} \end{cases}$

我们可以推出：$\mathbb{E}[x_i] = \frac{1}{2}$, $\sigma^2(x_i) = \frac{1}{4}$, $\mathbb{E}[X] = \frac{1}{2}r$, $\sigma^2(X) = \frac{1}{4}r$，故有：

$$\begin{aligned} \Pr[|L| > \tfrac{n}{2}] &\leq \Pr[X \geq \tfrac{1+\delta}{2}r] \\ &= \tfrac{1}{2}\Pr[|X - E(X)| \geq \tfrac{\delta}{2}r] \\ &\leq \tfrac{1}{2}\frac{\sigma^2(X)}{\left(\frac{\delta}{2}r\right)^2} \\ &= \tfrac{1}{2}\tfrac{1}{\delta^2 r} \\ &= \tfrac{1}{2}n^{-\frac{1}{4}} \end{aligned}$$

□

2.4.3 采用随机选择的一个元素作中心元

进一步地，即使我们随机采样 A 中的一个元素，也会以足够高的概率得到“足够好”的中心元。如图 2.8所示，我们能够以 $\frac{1}{2}$ 的概率挑选到“中间区域”的元素，使得无论是 S_- 还是 S_+，规模都不超过 $\frac{3}{4}n$，从而达到子实例规模呈指数性减少的目的。自然，我们在某一轮迭代时，也有可能采样出不好的中心元，使得子问题的规模不是呈

指数级降低；不过这并不用特别担心：连续迭代 2 轮，即可以很高的概率使得子问题的规模呈指数级降低。

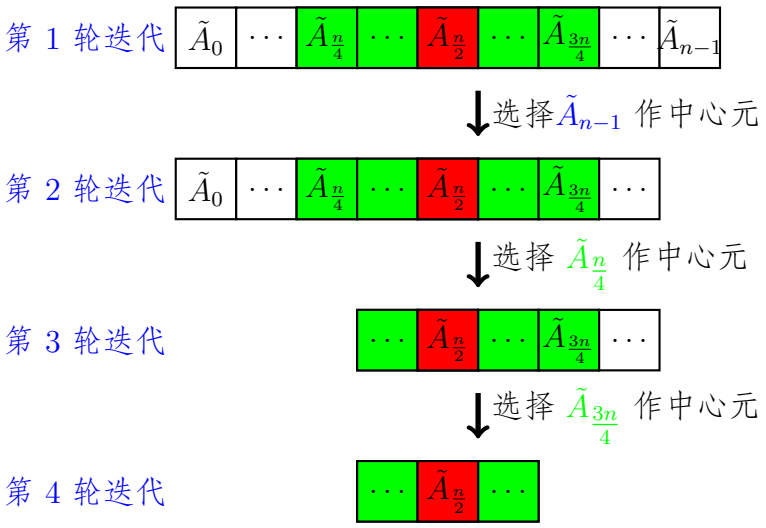


图 2.17: QuickSelect 算法运行过程示例

算法设计与描述

采用这种中心元挑选方案的算法称作 QuickSelect [?]，伪代码描述如下：

Algorithm 19 求解 SELECTION 问题的 QUICKSELECT 算法

function QUICKSELECT(A, k)

Choose an element A_i from A **uniformly at random**;
$S_+ = S_- = \emptyset$;
for all element A_j in A except A_i **do**
  if $A_j > A_i$ **then**
    $S_+ = S_+ \cup \{A_j\}$;
  else
    $S_- = S_- \cup \{A_j\}$;
  end if
end for
if $|S_-| = k - 1$ **then**
  return A_i;
else if $|S_-| > k - 1$ **then**
  return QUICKSELECT(S_-, k);
else
  return QUICKSELECT($S_+, k - |S_-| - 1$);
end if

图 2.17显示 QUICKSELECT 算法运行过程的一个示例。从图中可以看出，在第一轮中选择 $\tilde{A}_{n-1}$ 作为中心元，使得子实例的规模只减小了 1；而第二轮中选择 $\tilde{A}_{\frac{n}{4}}$ 作为中心元，使得子实例的规模减少了 $\frac{1}{4}n$。

时间复杂度分析

QUICKSELECT 算法是一个随机算法，每次运行过程以及运行时间可能都有所变化，但能够证明其期望运行时间是线性的。

定理 2.4.2. *对于一个包含 n 个元素的数组 A，QUICKSELECT 算法的期望运行时间是 $O(n)$ 的。*

QUICKSELECT 算法是一个递归算法，其运行过程包括多次递归调用，每个递归调用都是在一个子实例上运行；我们把所有步骤的比较次数求和，得到的总比较次数可以刻画算法的运行时间。

以图 2.17所示运行过程为例，总的比较次数为：

$$T(n) = n + (n-1) + (\tfrac{3n}{4} - 1) + (\tfrac{n}{2} - 2) + \cdots .$$

那如何估计这个总和的上界呢？我们会对等差数列求和，也会对等比数列求和，但是现在这个数列既不是等比数列，也不是等差数列，那如何进行求和呢？

仔细观察这个数列，可以发现有两个特点：

(1) 总体上看是等比下降，比如隔几个数取的子数列 $n, \frac{3n}{4}-1, \cdots$，下降剧烈；有些局部又是变化很缓慢的，比如 $n, n-1$。

(2) 缓慢变化的局部片段的长度是有限的；更确切地说，长度的期望值是 2。因此，我们可以使用局部片段中的最大数乘以 2 做为局部片段总和的上界。

不严格地说，我们把这样的序列叫做准等比数列。为了更清楚地表示这一点，我们在缓慢变化的局部片段两端加上括号，重写成下式：

$$T(n) = [n + (n-1)] + [(\tfrac{3n}{4} - 1) + (\tfrac{n}{2} - 2) + \cdots] + \cdots$$

基于上述两点观察，我们可以如下分析算法的时间复杂度。

证明. 我们依据子实例的大小将递归调用进行分组：如果子实例大小处于 $[n(\frac{3}{4})^{j+1} + 1, n(\frac{3}{4})^j]$ 区间，我们称算法运行处于第 j 期。直观上看，每一期就是变化缓慢的局部片段。

令 X 为 QuickSelect 算法运行过程中的元素比较次数，X_j 为运行过程处于第 j 期时的比较次数，则有：

$$X = X_0 + X_1 + X_2 + ...$$

考察第 j 期的运行过程：对于每个子实例来说，选择“中间区域”元素的概率是 $\frac{1}{2}$，而一旦选择一个中间区域的元素作为中心元，就会导致下一个子实例的规模减少至少 $\frac{1}{4}$，导致算法运行进入第 $j+1$ 期。因此第 j 期中递归调用的期望次数是 2，算法的期望比较次数不超过 $2n(\frac{3}{4})^j$（子实例最多有 $n(\frac{3}{4})^j$ 个元素）。

故有 $\mathbb{E}[X] = \mathbb{E}[X_0 + X_1 + ...] \leq \sum_j 2cn(\frac{3}{4})^j \leq 8cn$。 □

2.5 整数乘法：对一对数组的归约

迄今为止，我们讨论的问题的输入部分都是只涉及一个数组；下面我们做一些拓展，考察当输入包含两个数组时如何进行归约。我们从两个整数的乘法谈起。

整数乘法（Multiplication problem）

输入：两个 n 比特的整数 $p = (p_0 p_1 ... p_{n-1})_2$ 和 $q = (q_0 q_1 ... q_{n-1})_2$；

输出：p 与 q 的乘积 $p \times q$。

我们在小学阶段即已学习过计算乘积的“列竖式法”。以两个十进制整数 12 和 34 为例，我们采用如下步骤计算两者的乘积：

$$\begin{array}{r} 12 \\ \times 34 \\ \hline 48 \\ 36 \\ \hline 408 \end{array}$$

若以一位数字的加法和乘法作为基本操作的话，很显然“列竖式法”的时间复杂度是 $O(n^2)$。那么是否存在更快的算法呢？1962 年，A. Kolmogorov 在一个讨论班上提出猜测，认为不存在更快的算法，但是很快 A. A. Karatsuba 就利用*分而治之*技术，设计出了一个时间复杂度是 $O(n^{1.585})$ 的快速算法 [?, ?]。

Karatsuba 算法：把整数拆分成两部分

我们先确定如何将大的实例分解成小的子实例。注意到整数 p 的二进制表示 $(p_0p_1\cdots p_{n-1})_2$ 实质上是一个数组；因此我们可以采用数组排序中的方案，将大的数组 $p_0p_1\cdots p_{n-1}$ 分解成两个小的数组：“高位部分”$p_0p_1\cdots p_{\frac{n}{2}-1}$ 和“低位部分”$p_{\frac{n}{2}}p_{\frac{n}{2}+1}\cdots p_{n-1}$（为描述简单起见，我们假设 $n = 2^k$）。我们定义整数 $p_h = (p_0p_1\cdots p_{\frac{n}{2}-1})_2$，$p_l = (p_{\frac{n}{2}}p_{\frac{n}{2}+1}\cdots p_{n-1})_2$，则有：

$$p = p_h \times 2^{\frac{n}{2}} + p_l.$$

类似地，我们将 q 拆分成 q_h 和 q_l，则有：

$$q = q_h \times 2^{\frac{n}{2}} + q_l.$$

注意到原始实例的输入是一对数组 p 和 q，因此子实例也应当具有同样形式的输入，即包含两个数组作为输入，比如计算 p_h 和 q_h 的乘积。

在采用递归调用求解了子实例之后，我们需要考虑如何将子实例的解“组合”成原始实例的解。我们先尝试第一种“组合”方案：

$$\begin{aligned} pq &= (p_h \times 2^{\frac{n}{2}} + p_l)(q_h \times 2^{\frac{n}{2}} + q_l), && (2.5.1) \\ &= p_hq_h2^n + (p_hq_l + p_lq_h)2^{\frac{n}{2}} + p_lq_l. && (2.5.2) \end{aligned}$$

以十进制整数乘法 12×34 为例，可能更清楚地表述这种“组合”方案：

$$12 \times 34 = (1 \times 3) \times 10^2 + \big((1 \times 4) + (2 \times 3)\big) \times 10 + 2 \times 4.$$

采用这种“组合”方案，我们将 n 比特整数的乘法 pq，归约成 4 次 $\frac{n}{2}$ 比特整数的乘法，即 p_hq_h, p_hq_l, p_lq_h 和 p_lq_l，外加 3 次 n 比特整数的加法、2 次移位操作（乘以 2^n、乘

以 $2^{\frac{n}{2}}$)。因此整体算法的时间复杂度是:

$$T(n) = 4T(\tfrac{n}{2}) + O(n) = O(n^2).$$

要降低时间复杂度，减少子问题的数目是努力方向之一。Karatsuba 注意到计算交叉项之和 $(p_h q_l + p_l q_h)$ 时，除了分别计算两个交叉项 $p_h q_l$ 和 $p_l q_h$ 再相加之外，还可以如下计算:

$$p_h q_l + p_l q_h = (p_h + p_l)(q_h + q_l) - p_h q_h - p_l q_l.$$

因此 $p \times q$ 可以如下计算:

$$pq = p_h q_h 2^n + \big((p_h + p_l) \times (q_h + q_l) - p_h q_h + p_l q_l\big) 2^{\frac{n}{2}} + p_l q_l.$$

以十进制整数乘法为例，12×34 计算如下:

$$12 \times 34 = 1 \times 3 \times 10^2 + \big((1+2) \times (3+4) - 1 \times 3 - 2 \times 4\big) \times 10 + 3 \times 4.$$

时间复杂度分析

这种新的“组合”方案的好处是只需要求解 3 个子问题：我们将 n 比特整数的乘法 pq，归约成 3 次 $\frac{n}{2}$ 比特整数的乘法，即 $p_h q_h$, $(p_h + p_l) \times (q_h + q_l)$ 和 $p_l q_l$，以及 6 次 n 比特整数的加减法、2 次移位操作。

如果只进行 1 次递归调用，我们将子问题的数目从 4 降低至 3，的确算不上大的改进；但是当迭代进行多次递归调用、每一次都将子问题的数目从 4 降低至 3 时，其累计效应是非常显著的，能够将算法的时间复杂度降低至:

$$T(n) = 3T(\tfrac{n}{2}) + O(n) = O(n^{1.585}).$$

Karatsuba 算法是第一个比传统的“列竖式”法渐进更快的乘积计算方法。所谓渐进最快，是指当 n 比较大时，Karatsuba 算法比传统的“列竖式”法更快；而当 n 比较小时，Karatsuba 算法需要较多的加法操作，反而比较慢。在实际应用中这两种算法哪个更快，是和计算机的体系结构等多种因素密切相关的。

Toom-Cook 算法：把整数拆分成 3 部分或者更多部分

在 Karatsuba 算法中，整数 p 和 q 都被拆分成两部分；一个很自然的推广是将整数拆分成 3 部分甚至更多部分。1963 年，A. Toom 提出了这种拆分方案；1966 年，S. Cook 又做了进一步的改进 [?, ?]。采用这种拆分方案的算法被称作 Toom-Cook 算法，下面我们以拆分成 3 部分的方案 Toom-3 为例进行说明 [?]。

为描述简单起见，我们假设 $n = 3k$，并将 $p = (p_0 p_1 \cdots p_{n-1})_2$ 拆分成 3 部分，每部分 k 比特，即：“高位部分”$p_h = (p_0 p_1 \cdots p_{k-1})_2$、“中间部分”$p_m = (p_k p_{k+1} \cdots p_{2k-1})_2$ 和“低位部分”$p_l = (p_{2k} p_{2k+1} \cdots p_{3k-1})_2$。因此有：

$$p = p_h 2^{2k} + p_m 2^k + p_l.$$

类似地，我们将 q 也拆分成三部分 q_h, q_m 和 q_l，且有：

$$q = q_h 2^{2k} + q_m 2^k + q_l.$$

我们可以将乘积 pq 表示成：

$$pq = p_h q_h 2^{4k} + (p_h q_m + p_m q_h) 2^{3k} + (p_h q_l + p_m q_m + p_l q_h) 2^{2k} + (p_m q_l + p_l q_m) 2^k + p_l q_l.$$

这样，使用 9 次 k 比特整数的乘法，可以算出乘积 pq。采用这种归约方式的**分而治之**算法的时间复杂度是：

$$T(n) = 9T(\tfrac{n}{3}) + O(n) = O(n^2).$$

依据我们的经验，降低时间复杂度的途径之一是尽量减少 k 比特整数乘法的次数。Toom-3 算法能够只使用 5 次 k 比特整数的乘法，即可求出乘积 pq；其基本思想是将 pq 视为一个多项式的值，利用“**多项式值与系数之间的变换**”来快速求出系数，最终获得 pq 的值。详细地说，我们考虑如下的多项式：

$$\begin{aligned} P(x) &= p_h x^2 + p_m x + p_l, \\ Q(x) &= q_h x^2 + q_m x + q_l, \\ R(x) &= P(x)Q(x) \\ &= p_h q_h x^4 + (p_h q_m + p_m q_h) x^3 + (p_h q_l + p_m q_m + p_l q_h) x^2 + (p_m q_l + p_l q_m) x + p_l q_l. \end{aligned}$$

我们定义如下符号：

$$\begin{aligned} r_4 &= p_h q_h, \\ r_3 &= p_h q_m + p_m q_h, \\ r_2 &= p_h q_l + p_m q_m + p_l q_h, \\ r_1 &= p_m q_l + p_l q_m, \\ r_0 &= p_l q_l. \end{aligned}$$

从而可以将 $R(x)$ 写成 $R(x) = r_4 x^4 + r_3 x^3 + r_2 x^2 + r_1 x + r_0$，并可观察到如下事实：

(1) 待求的乘积是多项式 $R(x)$ 在点 2^k 处的值，即：$pq = R(2^k)$；因此我们只需要计算出多项式 $R(x)$ 的系数、然后移位相加即可求出 pq 来。

(2) 注意到多项式 $R(x)$ 是 4 阶多项式，因此我们只需要知道 $R(x)$ 在任意 5 个相异点处的值即可反推出 $R(x)$ 的系数来。为了提高算法效率，通常采用如

下精心设计的 5 个点处的值：$R(0)$, $R(-1)$, $R(+1)$, $R(-2)$ 和 $R(\infty)$，此处的 $R(\infty)$ 定义为 $R(x)$ 的最高阶的系数。

(3) 由于 $R(x) = P(x)Q(x)$，因此 $R(x)$ 在这 5 个点处的值可以很方便地利用 $P(x)$ 和 $Q(x)$ 的值相乘即可计算出来，即 $R(0) = P(0)Q(0)$, $R(1) = P(1)Q(1)$, $R(-1) = P(-1)Q(-1)$, $R(-2) = P(-2)Q(-2)$, $R(\infty) = P(\infty)Q(\infty)$。

值得指出的是：在这精心选择的 5 个点处，$P(x)$ 和 $Q(x)$ 的值都很小，都只有大约 k 个比特（严格地说，只有 $k + t$ 个比特，其中 t 是依赖于 k 的一个固定值）。这样，我们就将计算 $3k$ 比特整数 p 和 q 的乘积，归约成计算 5 次 k 比特整数的乘积。

算法设计与描述

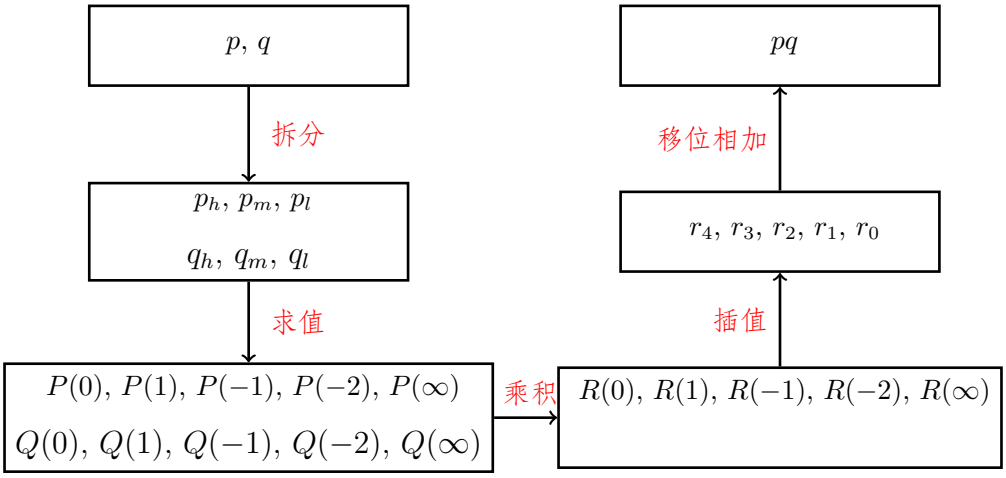


图 2.18: Toom-3 算法计算两个整数乘积 pq 的基本过程

如图 2.18所示，Toom-3 算法计算 pq 包括 5 步：整数 p 和 q 的拆分、多项式 $P(x)$ 和 $Q(x)$ 求值、计算 5 次 k 比特整数的乘积、插值计算多项式 $R(x)$ 的系数，以及 $R(x)$ 系数移位相加得到最终结果 pq。Toom-3 算法的伪代码描述如下：

Algorithm 20 TOOM-3 algorithm for multiplication

TOOM3($n = 3k, p = (p_0 p_1 ... p_{3k-1})_2, q = (q_0 q_1 ... q_{3k-1})_2$)

1: Decompose p into $p_h = (p_0 p_1 \cdots p_{k-1})_2$, $p_m = (p_k p_{k+1} \cdots p_{2k-1})_2$, $p_l = (p_{2k} p_{2k+1} \cdots p_{3k-1})_2$;
2: Decompose q into $q_h = (q_0 q_1 \cdots q_{k-1})_2$, $q_m = (q_k q_{k+1} \cdots q_{2k-1})_2$, $q_l = (q_{2k} q_{2k+1} \cdots q_{3k-1})_2$;
3: Evaluate $P(x) = p_h x^2 + p_m x + p_l$ at 0, 1, -1, -2 and ∞;
4: Evaluate $Q(x) = q_h x^2 + q_m x + q_l$ at 0, 1, -1, -2 and ∞;
5: **for** i in $\{0, 1, -1, -2, \infty\}$ **do**
6: $R(i) =$TOOM3$(k, P(i), Q(i))$;
7: **end for**
8: Derive the coefficients of $R(x)$, i.e., r_4, r_3, r_2, r_1, r_0, based on $R(0)$, $R(1)$, $R(-1)$, $R(-2)$, $R(\infty)$;
9: **return** $R(2^k)$;

我们以计算十进制整数 $p = 123$ 和 $q = 456$ 的乘积为例来说明 TOOM-3 算法的基本过程：

(1) 拆分：首先将 p 拆分成 $p_h = 1$, $p_m = 2$, $p_l = 3$，并构建多项式 $P(x) = 1x^2 + 2x + 3$；同样地，我们将 q 拆分成 $q_h = 4$, $q_m = 5$, $q_l = 6$，并构建多项式 $Q(x) = 4x^2 + 5x + 6$；

(2) 求值：我们如下计算 $P(x)$ 和 $Q(x)$ 在 $0, 1, -1, -2, \infty$ 处的值：

$$\begin{aligned}
P(0) &= p_h(0)^2 + p_m(0) + p_l &&= p_l &&= 3,\\
P(1) &= p_h(1)^2 + p_m(1) + p_l &&= p_h + p_m + p_l &&= 6,\\
P(-1) &= p_h(-1)^2 + p_m(-1) + p_l &&= p_h - p_m + p_l &&= 2,\\
P(-2) &= p_h(-2)^2 + p_m(-2) + p_l &&= 4p_h - 2p_m + p_l &&= 3,\\
P(\infty) &= p_h &&&&= 1.\\
Q(0) &= q_h(0)^2 + q_m(0) + q_l &&= q_l &&= 6\\
Q(1) &= q_h(1)^2 + q_m(1) + q_l &&= q_h + q_m + q_l &&= 15\\
Q(-1) &= q_h(-1)^2 + q_m(-1) + q_l &&= p_h - q_m + q_l &&= 5\\
Q(-2) &= q_h(-2)^2 + q_m(-2) + q_l &&= 4q_h - 2q_m + q_l &&= 12\\
Q(\infty) &= q_h &&&&= 4
\end{aligned}$$

(3) 相乘：通过递归调用，计算 $P(x)$ 和 $Q(x)$ 在这 5 个点处的值的乘积，得到

$R(x)$ 的值如下：

$$\begin{aligned} R(0) &= P(0)Q(0) &&= 18 \\ R(1) &= P(1)Q(1) &&= 90 \\ R(-1) &= P(-1)Q(-1) &&= 10 \\ R(-2) &= P(-2)Q(-2) &&= 36 \\ R(\infty) &= P(\infty)Q(\infty) &&= 4 \end{aligned}$$

(4) 插值：在描述插值过程之前，我们先看如何由 $R(x)$ 的系数计算 $R(x)$ 的值：

$$\begin{bmatrix} R(0) \\ R(1) \\ R(-1) \\ R(-2) \\ R(\infty) \end{bmatrix} = \begin{bmatrix} 0^0 & 0^1 & 0^2 & 0^3 & 0^4 \\ 1^0 & 1^1 & 1^2 & 1^3 & 1^4 \\ (-1)^0 & (-1)^1 & (-1)^2 & (-1)^3 & (-1)^4 \\ (-2)^0 & (-2)^1 & (-2)^2 & (-2)^3 & (-2)^4 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix}$$

因此我们对上述矩阵求逆即可由 $R(x)$ 的值反推出 $R(x)$ 的系数来，即：

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{3} & -1 & \frac{1}{6} & -2 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ -\frac{1}{2} & \frac{1}{6} & \frac{1}{2} & -\frac{1}{6} & 2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R(0) \\ R(1) \\ R(-1) \\ R(-2) \\ R(\infty) \end{bmatrix}$$

需要指出的是：虽然上述逆矩阵中包含有分数，但是最终计算出的 $R(x)$ 的系数都是整数。为加速矩阵和向量乘积，M. Bodrato 采用最短路径算法，计算出了求解 $R(x)$ 系数的最短计算序列 [**?**]，示例如下：

$$\begin{aligned} r_0 &= R(0) &&= 18 \\ r_4 &= R(\infty) &&= 4 \\ r_3 &= \big(R(-2) - R(1)\big)/3 &&= -18 \\ r_1 &= \big(R(1) - R(-1)\big)/2 &&= 40 \\ r_2 &= R(-1) - R(0) &&= -8 \\ r_3 &= (r_2 - r_3)/2 + 2R(\infty) &&= 13 \\ r_2 &= r_2 + r_1 - r_4 &&= 28 \\ r_1 &= r_1 - r_3 &&= 27. \end{aligned}$$

至此我们得到 $R(x)$ 的完整表达式为：

$$R(x) = 4x^4 + 13x^3 + 28x^2 + 27x + 18.$$

(5) 移位相加: 这个实例采用的是十进制，因此我们需要计算 $R(10)$，即为 123×456 的结果；而这只需移位相加即可完成：

$$123 \times 456 = R(10) = 40000 + 13000 + 2800 + 270 + 18 = 56088.$$

时间复杂度分析

Toom-3 算法将 $n = 3k$ 比特的整数乘积，归约成 5 次 k 比特整数的乘积，因此算法的时间复杂度是：

$$T(n) = 5T(\tfrac{n}{3}) + O(n) = O(n^{\log_3 5}) = O(n^{1.465}).$$

Toom-Cook 算法的一般情况是：将 n 比特的整数划分成 r 个部分，并将 n 比特整数的乘积归约成 $(2r+1)$ 次 $\frac{n}{r}$ 比特整数的乘积，因此时间复杂度是：

$$T(n) = (2r+1)T(\tfrac{n}{3}) + O(n) \leq C(r)n^{\log_{r+1}(2r+1)},$$

其中 $C(r)$ 是不依赖于 n 的一个常数。从这个角度讲，Karatsuba 算法也被称为 Toom-2 算法，是 Toom-Cook 算法的一个特例。

D. E. Knuth 证明了 Toom-Cook 算法时间复杂度是 $T(n) = O(n2^{\sqrt{2\log n}}\log n)$ [**?**]。在下一节中，我们将会看到采用快速傅里叶变换，能够使用 $O(n\log n\log\log n)$ 次比特操作完成整数相乘 [**?**]。

2.6 快速傅里叶变换：对数组的归约

快速傅里叶变换（Fast Fourier Transform, FFT）是计算离散傅里叶变换的快速算法，其目的是多项式求值；更确切地说，是给定多项式系数之后，求该多项式在 n 个特定点处的值，形式化描述如下：

离散傅里叶变换问题（Discrete Fourier transform problem）

输入：正整数 n，以及包含 n 个复数的数组 $a_0, a_1, \cdots a_{n-1}$；

输出：以 $a_0, a_1, \cdots a_{n-1}$ 为系数的多项式 $A(x) = a_0 + a_1x + \cdots a_{n-1}x^{n-1}$ 在 n 个特定点 $1, \omega, \omega^2, ..., \omega^{n-1}$ 的值 $A(1), A(\omega), \cdots A(\omega^{n-1})$。此处，$\omega = \mathrm{e}^{\frac{2\pi}{n}\mathrm{i}}$ 表示 n 次单位复根。

离散傅里叶变换就是将输入的 n 个复数 $a_0, a_1, \cdots a_{n-1}$ 变换成 n 个复数 $y_0, y_1, \cdots y_{n-1}$，即完成从多项式“系数”到多项式“值”的变换：

$$\begin{array}{llllll} y_0 = a_0 & + a_1 & + a_2 & \ldots & + a_{n-1} \\ y_1 = a_0 & + a_1\omega^1 & + a_2\omega^2 & \ldots & + a_{n-1}\omega^{n-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ y_{n-1} = a_0 & + a_1\omega^{n-1} & + a_2\omega^{2(n-1)} & \ldots & + a_{n-1}\omega^{(n-1)^2} \end{array}$$

为描述简单起见，我们假设 $n = 2^k$。我们先从最简单的情形入手：当 $n = 2$ 时，我们很容易计算出 $A(x)$ 在两个特定点 $1, -1$ 处的值，即：

$$A(1) = a_0 + a_1,$$
$$A(-1) = a_0 - a_1.$$

那么对于更复杂的实例，比如 $n = 4$，又该如何求解呢?

我们可以采用 Horner 规则来快速计算多项式的值 $A(x)$，即：

$$A(x) = a_0 + x(a_1 + x(a_2 + \ldots + xa_{n-1})).$$

即使采用了 Horner 规则，仍然需要 $n - 1$ 次的复数乘法和加法才能完成在 1 个点处的多项式求值，因此完成 n 个点处的多项式求值共需 $O(n^2)$ 次复数乘法和加法，时间复杂度是 $O(n^2)$。1965 年，J. Cooley 和 J. Tukey 采用**分而治之**策略，设计出快速傅里叶变换算法 [**?**]，将时间复杂度降低至 $O(n \log n)$。

应用**分而治之**策略的首要步骤是将原始实例拆分成子实例：我们注意到问题的输入是一个数组 $a_0, a_1, \cdots, a_{n-1}$；依据在数组排序中的经验，我们能够很容易地依据下标将一个大的数组拆分成小的数组，比如拆分成左一半 $a_0, a_1, \cdots, a_{\frac{n}{2}-1}$ 和右一半 $a_{\frac{n}{2}}, a_{\frac{n}{2}+1}, \cdots, a_n$。然而在 FFT 算法中，并不是简单地将数组拆分成左一半和右一半，而是按照下标的奇偶性拆分成 $a_0, a_2, \cdots, a_{n-2}$ 和 $a_1, a_3, \cdots, a_{n-1}$。之所以按照下标的奇偶性进行拆分，是和我们要在 n 个**特定点**（而不是任意点）处对多项式求值这一目标密切相关（另一个按照奇偶性拆分的例子是计算矩阵每行最小值的 SMAWK 算法 [**?**]，我们将在第 4 章讲述）。下面我们以 $n = 4$ 的情形为例进行说明。

当 $n = 4$ 时，我们的目标是计算多项式 $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3$ 在 4 个特定点 $1, \mathrm{i}, -1, -\mathrm{i}$ 处的值，即：

$$A(1) = a_0 + a_1 + a_2 + a_3,$$
$$A(\mathrm{i}) = a_0 + a_1\mathrm{i} - a_2 - a_3\mathrm{i},$$
$$A(-1) = a_0 - a_1 + a_2 - a_3,$$
$$A(-\mathrm{i}) = a_0 - a_1\mathrm{i} - a_2 + a_3\mathrm{i}.$$

我们把 3 阶多项式 $A(x)$ 拆分成奇数项和偶数项，定义如下两个 1 阶多项式：

$$A_{even}(x) = a_0 + a_2x$$
$$A_{odd}(x) = a_1 + a_3x$$

很容易能够推导出：

$$A(x) = (a_0 + a_2x^2) + x(a_1 + a_3x^2) = A_{even}(x^2) + xA_{odd}(x^2)$$
$$A(-x) = (a_0 + a_2x^2) - x(a_1 + a_3x^2) = A_{even}(x^2) - xA_{odd}(x^2)$$

因此，在 4 个特定点 $1, \mathrm{i}, -1, -\mathrm{i}$ 计算 $A(x)$ 的值这个问题，可以归约为在 $1, -1$ 两个点处计算 $A_{even}(x)$ 和 $A_{odd}(x)$ 的值的问题，即：

$$A(1) = A_{even}(1) + A_{odd}(1)$$
$$A(\mathrm{i}) = A_{even}(-1) + \mathrm{i}A_{odd}(-1)$$
$$A(-1) = A_{even}(1) - A_{odd}(1)$$
$$A(-\mathrm{i}) = A_{even}(-1) - \mathrm{i}A_{odd}(-1)$$

对于更复杂的实例，比如 $n = 8, 16$ 等，我们可以类似地归约成简单的子实例。

算法设计与描述

采用上述实例归约方案的*分而治之*算法被称为 Cooley-Tukey 快速傅里叶变换算法，其伪代码描述如下：

Algorithm 21 Cooley–Tukey Fast Fourier transform (FFT) algorithm

FFT$(n, a_0, a_1, ..., a_{n-1})$

1: **if** $n == 1$ **then**
2: **return** a_0;
3: **end if**
4: $(E_0, E_1, ..., E_{\frac{n}{2}-1})$ =FFT$(\frac{n}{2}, a_0, a_2, ..., a_{n-2})$;
5: $(O_0, O_1, ..., O_{\frac{n}{2}-1})$ = FFT$(\frac{n}{2}, a_1, a_3, ..., a_{n-1})$;
6: **for** $k = 0$ to $\frac{n}{2} - 1$ **do**
7: $\omega^k = \mathrm{e}^{\frac{2\pi}{n}k\mathrm{i}}$;
8: $y_k = E_k + \omega^k O_k$;
9: $y_{\frac{n}{2}+k} = E_k - \omega^k O_k$;
10: **end for**
11: **return** $(y_0, y_1, ..., y_{n-1})$;

其中 FFT$(\frac{n}{2}, a_0, a_2, ..., a_n)$ 是计算多项式 $A_{even}(x) = a_0 + a_2x + ... + a_nx^{\frac{n}{2}}$ 在 $\frac{n}{2}$ 个特定点 $1, \omega^2, \omega^4, ..., \omega^{n-2}$ 处的值，FFT$(\frac{n}{2}, a_1, a_3, ..., a_{n-1})$ 计算多项式 $A_{odd}(x) = a_1 + a_3x + ... + a_{n-1}x^{\frac{n}{2}}$ 在这些特定点处的值。

时间复杂度分析

我们以 $T(n)$ 表示 FFT 算法计算多项式 $A(x)=a_0+a_1x+\cdots a_{n-1}x^{n-1}$ 在 n 个特定点$1,\omega,\omega^2,...,\omega^{n-1}$ 的值的基本操作（复数乘法和加减法）次数，因此有：

$$T(n)=2T(\tfrac{n}{2})+O(n)=O(n\log n)$$

之所以能够将时间复杂度从 $O(n^2)$ 降低至 $O(n\log n)$，完全是由于发现并去除了多项式求值计算中的“冗余”计算。下面我们以 $n=4$ 的情形为例进行说明：我们把多项式中的项挪动一下位置，把偶数项集中到左边，把奇数项集中到右边，则可以重写成如下的表达式：

$$\begin{aligned}
A(1) &= a_0+a_2 \quad +a_1+a_3\\
A(\mathrm{i}) &= a_0-a_2 \quad +a_1\mathrm{i}-a_3\mathrm{i}\\
A(-1) &= a_0+a_2 \quad -a_1-a_3\\
A(-\mathrm{i}) &= a_0-a_2 \quad -a_1\mathrm{i}+a_3\mathrm{i}
\end{aligned}$$

注意到上述表达式中存在着冗余计算：红色框内的部分完全相同，因此只需计算一次即可；而蓝色框内的部分只差一个负号，也只需计算一次即可。当去除冗余计算之后，我们只需计算 $2+4+2=4\log_2 4$ 次计算即可完成多项式的求值。

对于 $n=8$ 的情形，我们采用类似的移项操作，也能够发现并去除冗余计算：

$$\begin{aligned}
y_0 &= a_0+a_4+a_2+a_6 \quad +a_1+a_5+a_3+a_7\\
y_1 &= a_0+a_4\omega^4+a_2\omega^2+a_6\omega^6 \quad +a_1\omega^1+a_5\omega^5+a_3\omega^3+a_7\omega^7\\
y_2 &= a_0+a_4+a_2\omega^4+a_6\omega^4 \quad +a_1\omega^2+a_5\omega^2+a_3\omega^6+a_7\omega^6\\
y_3 &= a_0+a_4\omega^4+a_2\omega^6+a_6\omega^2 \quad +a_1\omega^3+a_5\omega^7+a_3\omega^1+a_7\omega^5\\
y_4 &= a_0+a_4+a_2+a_6 \quad +a_1\omega^4+a_5\omega^4+a_3\omega^4+a_7\omega^4\\
y_5 &= a_0+a_4\omega^4+a_2\omega^2+a_6\omega^6 \quad +a_1\omega^5+a_5\omega^1+a_3\omega^7+a_7\omega^3\\
y_6 &= a_0+a_4+a_2\omega^4+a_6\omega^4 \quad +a_1\omega^6+a_5\omega^6+a_3\omega^2+a_7\omega^2\\
y_7 &= a_0+a_4\omega^4+a_2\omega^6+a_6\omega^2 \quad +a_1\omega^7+a_5\omega^3+a_3\omega^5+a_7\omega^1
\end{aligned}$$

此处 $\omega=\mathrm{e}^{\frac{2\pi}{8}\mathrm{i}}$。注意到红框部分完全相同，因此只需计算一次即可；篮框部分差一个负号，也只需计算一次即可。由于红框部分和篮框部分都是一个 3 阶多项式的求值，我们采用递归方式求解。如下图所示，我们用阴影部分表示冗余计算，当去除冗

余计算之后，我们只需 $2+4+2+8+2+4+2=8\log_2 8$ 次计算即可完成多项式的求值。

$$
\begin{aligned}
y_0 &= a_0 + a_4 + a_2 + a_6 + a_1 + a_5 + a_3 + a_7 \\
y_1 &= a_0 + a_4\omega^4 + a_2\omega^2 + a_6\omega^6 + a_1\omega^1 + a_5\omega^5 + a_3\omega^3 + a_7\omega^7 \\
y_2 &= a_0 + a_4 + a_2\omega^4 + a_6\omega^4 + a_1\omega^2 + a_5\omega^2 + a_3\omega^6 + a_7\omega^6 \\
y_3 &= a_0 + a_4\omega^4 + a_2\omega^6 + a_6\omega^2 + a_1\omega^3 + a_5\omega^7 + a_3\omega^1 + a_7\omega^5 \\
y_4 &= a_0 + a_4 + a_2 + a_6 + a_1\omega^4 + a_5\omega^4 + a_3\omega^4 + a_7\omega^4 \\
y_5 &= a_0 + a_4\omega^4 + a_2\omega^2 + a_6\omega^6 + a_1\omega^5 + a_5\omega^1 + a_3\omega^7 + a_7\omega^3 \\
y_6 &= a_0 + a_4 + a_2\omega^4 + a_6\omega^4 + a_1\omega^6 + a_5\omega^6 + a_3\omega^2 + a_7\omega^2 \\
y_7 &= a_0 + a_4\omega^4 + a_2\omega^6 + a_6\omega^2 + a_1\omega^7 + a_5\omega^3 + a_3\omega^5 + a_7\omega^1
\end{aligned}
$$

值得指出的是上述计算中的项的排列顺序：我们先将 $a_0, a_1, \cdots, a_7$ 拆分成偶数项 a_0, a_2, a_4, a_6 和奇数项 a_1, a_3, a_5, a_7，继而在进一步递归调用时，把 a_0, a_2, a_4, a_6 拆分成 a_0, a_4 和 a_2, a_6，把 a_1, a_3, a_5, a_7 拆分成 a_1, a_5 和 a_3, a_7，从而形成最终的下标排列顺序 $a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7$。

FFT 完成的是从“多项式系数到多项式值”的变换；与之方向完全相反的逆问题是从“多项式值到多项式系数”的变换。当已知 $n-1$ 阶多项式 $A(x)$ 在 n 个相异点处的值时，其系数可以唯一确定。这个问题被称作“逆离散傅里叶变换”（Inverse Discrete Fourier Transform）问题，描述如下：

逆离散傅里叶变换问题（Inverse discrete Fourier transform problem）

输入：正整数 n，$n-1$ 阶多项式 $A(x)=a_0+a_1x+\cdots a_{n-1}x^{n-1}$ 在 n 个特定点$1, \omega, \omega^2, ..., \omega^{n-1}$ 的值 $A(1), A(\omega), \cdots A(\omega^{n-1})$。此处，$\omega = \mathrm{e}^{\frac{2\pi}{n}\mathrm{i}}$ 表示 n 次单位复根；

输出：多项式系数 $a_0, a_1, \cdots a_{n-1}$。

IDFT 是和 DFT 完全反向的计算：如图**??**所示，DFT 解决的是“求值”问题，是已知多项式系数时，对多项式求值；而 IDFT 解决的是“插值”问题，是已知多项式的值，要反推出多项式的系数来。我们知道，如果已知 $n-1$ 阶多项式在 n 个相异点处的值，多项式的系数是唯一确定的。

图 2.19: DFT 和 IDFT 之间的关系

TBA: Pham theorem C(I outerproduct Q) = IFFT (FFT(C(I)) pointwisemutiply FFT(C(Q)))

C: Count Sketch

如果采用矩阵形式的话，我们可以把正向的离散傅里叶变换问题表示如下：

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^1 & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)^2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

上式中的矩阵被称为傅里叶矩阵 F_n。之所以 FFT 算法能够快速完成矩阵和向量乘积，完全是充分利用了矩阵 F_n 的一个特性：F_n 可以分解成稀疏矩阵的乘积 [?]。

现在我们要进行逆离散傅里叶变换，这只需对傅里叶矩阵求逆即可从多项式的值 $y_0, y_1, \cdots y_{n-1}$ 计算出多项式系数 $a_0, a_1, \cdots, a_{n-1}$。值得指出的是，对于一般的矩阵来说，求逆运算的时间复杂度是 $O(n^3)$；然而傅里叶矩阵是一个特殊的范德蒙矩阵（Vandermonde matrix），其逆矩阵具有几乎完全相同的形式，只是多了一个取共轭运算和倍数 $\frac{1}{n}$，从而将逆离散傅里叶变换表示如下：

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \bar{\omega}^1 & \bar{\omega}^2 & \dots & \bar{\omega}^{n-1} \\ 1 & \bar{\omega}^2 & \bar{\omega}^4 & \dots & \bar{\omega}^{2(n-1)} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & \bar{\omega}^{n-1} & \bar{\omega}^{2(n-1)} & \dots & \bar{\omega}^{(n-1)^2} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

由于逆傅里叶变换具有与正向离散傅里叶变换完全相同的形式，因此我们可以只需对 FFT 算法做简单的修改，即可完成逆傅里叶变换（伪代码描述见算法 22）。顺便说一下，和 FFT 算法相比，IFFT 算法的最终结果需要乘一个因子 $\frac{1}{n}$，看起来和 FFT 的计算不对称。其实这只是常用的描述方式之一；在另外的描述方式中，为了对称起见，在 FFT 算法和 IFFT 算法的最终结果中都乘以 $\frac{1}{\sqrt{n}}$ [?]。

Algorithm 22 Inverse FFT algorithm

IFFT$(n, y_0, y_1, ..., y_{n-1})$

1: **if** $n == 1$ **then**
2: **return** y_0 ;
3: **end if**
4: $(E_0, E_1, ..., E_{\frac{n}{2}-1}) = \text{IFFT}(\frac{n}{2}, y_0, y_2, ..., y_n)$;
5: $(O_0, O_1, ..., O_{\frac{n}{2}-1}) = \text{IFFT}(\frac{n}{2}, y_1, y_3, ..., y_{n-1})$;
6: **for** $k = 0$ to $\frac{n}{2} - 1$ **do**
7: $\omega^k = \mathrm{e}^{-\frac{2\pi}{n}k\mathrm{i}}$;
8: $a_k = E_k + \omega^k O_k$;
9: $a_{\frac{n}{2}+k} = E_k - \omega^k O_k$;
10: **end for**
11: **return** $\frac{1}{n}(a_0, a_1, ..., a_{n-1})$;

FFT 算法的应用之一：多项式乘法和整数乘法

FFT 算法有着广泛的应用，其中之一是快速计算两个多项式的乘积，即已知两个 $n-1$ 阶多项式 $A(x) = a_0 + a_1x + a_2x^2 + ... + a_{n-1}x^{n-1}$ 和 $B(x) = b_0 + b_1x + b_2x^2 + ... + b_{n-1}x^{n-1}$ 的系数，计算这两个多项式乘积 $C(x) = A(x)B(x) = c_0 + c_1x + c_2x^2 + ... + c_{2n-2}x^{2n-2}$ 的系数。

直接将 $A(x)B(x)$ 展开，我们能够推导出多项式 $C(x)$ 系数的卷积形式，即：$c_k = \sum_{i=0}^{k} a_i b_{k-i}$。采用这种“由系数推算系数”的方式，我们能够在 $O(n^2)$ 时间内计算出 $C(x)$ 的所有系数来。

然而换一种思路，如果采用“由多项式值推算系数”方式的话，能够更高效地计算出 $C(x)$ 的系数来。概况地说，两个函数卷积的 FFT 变换等于这两个函数的变换的乘积。详细地说，假如知道 $C(x)$ 在 $2n-1$ 个相异点 $1, \omega, \cdots, \omega^{2n-2}$ 处的值，我们就能够使用逆 FFT 算法在 $O(n \log n)$ 时间内计算出 $C(x)$ 的系数来；而 $C(x)$ 在这 $2n-1$ 个特异点处的值，可以先用 FFT 算法求出 $A(x)$ 和 $B(x)$ 的值，然后相乘得到。整体计算过程表示如图 2.20。

举例来说，给定多项式 $A(x) = 1 + 2x$, $B(x) = 3 + 4x$，我们计算乘积：

$$C(x) = A(x)B(x) = c_0 + c_1x + c_2x^2 + c_3x^3.$$

如表 2.3所示，我们先使用 FFT 计算出 $A(x)$ 和 $B(x)$ 在 4 个相异点 $1, -\mathrm{i}, -1, \mathrm{i}$ 处的值；然后相乘得到 $C(x)$ 的值，分别为 $21, -5-10\mathrm{i}, 1, -5+10\mathrm{i}$；最后使用 IFFT 算法，

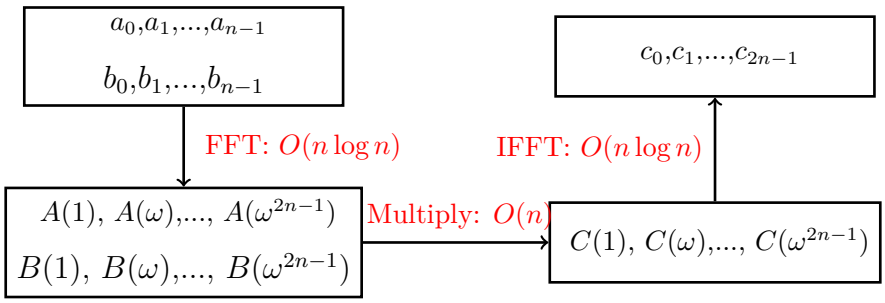


图 2.20: 使用 FFT 和 IFFT 计算多项式乘积 $C(x) = A(x)B(x)$

即 $\text{IFFT}(4, (21, -5-10\text{i}, 1, -5+10\text{i}))$，计算出 $C(x)$ 的系数为 $c_0 = 3, c_1 = 10, c_2 = 8, c_3 = 0$。整个过程调用 2 次 FFT 和 1 次 IFFT，时间复杂度是 $O(n\log n)$。

值得指出的是：这种先做正变换、运算之后再做逆变换的思想是非常常见的。例如矩阵的相似性变换 $B = P^{-1}AP$ 就是先将矩阵 A 乘以 P 做一次线性变换，再乘以 P^{-1} 变换回来。

表 2.3: 使用 FFT/IFFT 计算多项式乘积示例

x	1	$-\text{i}$	-1	i
$A(x)$	3	$1-2\text{i}$	-1	$1+2\text{i}$
$B(x)$	7	$3-4\text{i}$	-1	$3+4\text{i}$
$C(x)$	21	$-5-10\text{i}$	1	$-5+10\text{i}$

多项式乘积算法可以很自然地扩展到整数乘法：1971 年，A. Schönhage 和 V. Strassen 提出了 Schönhage-Strassen 算法，能够使用 $O(n\log n\log\log n)$ 次比特乘法计算出 n 比特整数的乘积 [?]。以表 2.3所示实例为例，在计算出 $c_0 = 3, c_1 = 10, c_2 = 8, c_3 = 0$ 之后，我们可以如下计算 21 和 43 的乘积：

$$21 \times 43 = 3 + 10 \times 10 + 8 \times 100 + 0 \times 1000 = 903.$$

广泛使用的 GMP 库（GNU Multi-Precision library）依据整数位数选择最合适的算法：当整数位数小于 250 时，使用传统的移位相加方法计算乘积；当整数位数大于 250 时，使用 Karatsuba 算法和 Toom-3 算法计算乘积；当整数位数大于 35000 时，则使用 FFT 来计算乘积。

FFT 算法的应用之二：信号的时域-频域转换

我们有两种方式表示一个信号：i) 将信号表示成时间的函数（**时域表示**），以及 ii) 将信号表示成频率的函数（**频域表示**）。FFT 能够将信号的时域表示变换成频域表示，而 IFFT 能够将信号的频域表示变换成时域表示。

图 2.6展示一个例子；这个例子是由下述 Matlab 代码生成的：

```
N = 8;
t = 0:1/N:1-1/N;
a = cos(2*pi*1*t) + 2*sin(2*pi*3*t);
Freq = 0:N-1;
bar( Freq, abs(fft(a)), "b", 0.2);
```

这段代码的意思是：考虑时域信号 $\cos(2\pi t) + 2\sin(6\pi t)$；我们以采样率 $N = 8$ 进行采样，共得到 8 个时刻的信号幅度值，称为**信号的时域表示**，用向量形式记为 $A = [a_0, a_1, \cdots, a_7] = [1, 2.707, 0, -2.707, -1, 1.292, 0, 1.292]$。对这个例子而言，信号中存在着频率为 1（8 个采样点内 1 个周期）的余弦分量，以及频率为 3（8 个采样点内 3 个周期）的正弦分量。然后，我们使用 FFT 将信号的时域表示 $a_0, a_1, \cdots, a_7$ 转换成频域表示，计算出两个分量的幅度值分别为 4 和 8，其比值和原始信号是一致的。

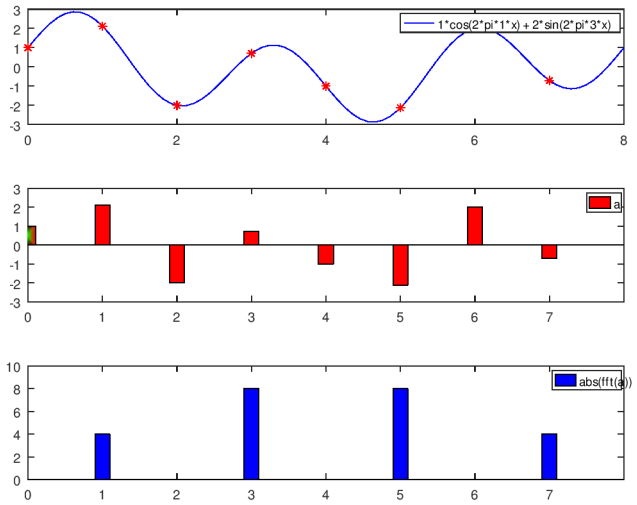


图 2.21: 时域信号及其采样，以及 FFT 频谱分析结果示例

按照上一小节的说法，FFT 计算的是多项式 $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$

在 n 个特定点处的值。那么以信号的时域采样 $a_0, a_1, \cdots, a_{n-1}$ 作为多项式系数时，为何多项式的值能够刻画信号中某一特定周期的正弦、余弦分量的幅度和相位呢?

这其中的原因可以直观解释如下:

(1) **多项式的值也可视为向量的点积**: $A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$ 是多项式的值，而从另一个角度来看，还是两个向量的点积，其中一个向量是多项式的系数 $(a_0, a_1, a_2, \cdots, a_{n-1})$，另一个向量是 $(1, x, x^2, \cdots, x^{n-1})$。

(2) **“周期型向量”的正交性**: 注意到 FFT 只计算多项式在特定点 $x = \mathrm{e}^{\frac{2k\pi}{n}\mathrm{i}}$ $(k = 0, 1, \cdots, n-1)$ 处的值; 此时，向量 $(1, x, x^2, ..., x^{n-1})$ 来自于一个周期函数（我们这里非规范地称之为 “周期型向量”），而来自不同周期函数的向量是正交的。

以 $n = 8$ 为例，我们考察如下 8 个周期型向量:

$$
\begin{aligned}
V_0 =&[1, && 1,1, && 1,1, && 1,1, && 1]\\
V_1 =&[1, && \omega,\omega^2, && \omega^3,\omega^4, && \omega^5,\omega^6, && \omega^7],\\
V_2 =&[1, && \omega^2,\omega^4, && \omega^6,\omega^8, && \omega^{10},\omega^{12}, && \omega^{14}],\\
& && && \cdots\\
V_7 =&[1, && \omega^7,\omega^{14}, && \omega^{21},\omega^{28}, && \omega^{35},\omega^{42}, && \omega^{49}].
\end{aligned}
$$

此处 $\omega = \mathrm{e}^{\frac{2\pi}{8}\mathrm{i}}$。很容易验证这些周期型向量的正交性，即: $V_0^T V_1 = 0$, $V_0^T V_2 = 0$, $\cdots$, $V_6^T V_7 = 0$，意味着这些向量构成一组基向量。

我们把多项式系数 $A = [a_0, a_1, \cdots, a_7] = [1, 2.707, 0, -2.707, -1, 1.292, 0, 1.292]$ 表示成基向量的加权和，得到下式:

$$A = 0 \times V_0 + \tfrac{1}{2} \times V_1 + 0 \times V_2 + V_3 + 0 \times V_4 + V_5 + 0 \times V_6 + \tfrac{1}{2}V_7.$$

进一步地，我们可以求出多项式的值：

$$\begin{aligned}
y_0 = &\ A(1) = A^T V_0 = 0 \times 8 = 0,\\
y_1 = &\ A(\omega) = A^T V_1 = \tfrac{1}{2} \times 8 = 4,\\
y_2 = &\ A(\omega^2) = A^T V_2 = 0 \times 8 = 0,\\
y_3 = &\ A(\omega^3) = A^T V_3 = 1 \times 8 = 8,\\
y_4 = &\ A(\omega^4) = A^T V_4 = 0 \times 8 = 0,\\
y_5 = &\ A(\omega^5) = A^T V_5 = 1 \times 8 = 8,\\
y_6 = &\ A(\omega^6) = A^T V_6 = 0 \times 8 = 0,\\
y_7 = &\ A(\omega^7) = A^T V_7 = \tfrac{1}{2} \times 8 = 4.
\end{aligned}$$

因此，FFT 算法计算出的多项式的值，的确是和正弦、余弦分量的强度是成正比的。以 $y_1 = A(\omega) = a_0 + a_1\omega + a_2\omega^2 + \ldots + a_7\omega^7$ 为例，y_1 表示的是两个向量的点积：一个向量是时域采样 $(a_0, a_1, ..., a_7)$，另一个向量是正弦函数的时域采样 $(1, \omega, \omega^2, ..., \omega^7)$，这个正弦信号的一个周期覆盖 8 个样本点。

因此，y_1 是可以看做是对时域采样 $a_0, a_1, ..., a_7$ 是否包含周期性分量的一次测试：当时域采样中不包含周期为 8 个样本点的正弦分量时，$y_1 = 0$；当时域采样包含该正弦分量时，$y_1 \neq 0$，其幅度表示正弦分量的强度，其幅角表示正弦分量的相位。类似地，y_2 刻画的时域采样中周期为 4 个样本点的正弦分量的强度和相位；$y_3, \cdots, y_8$ 依次类推。

上面的论证中，其核心是**周期型向量的正交性**，这很容易理解：周期型向量的正交性是三角函数正交性的离散化版本。具体地说，三角函数 $1, \cos x, \sin x, \cos 2x, \sin 2x, \cdots$ 构成正交系，即对于任意的整数 m, n 来说（注意：整数要求必不可少），有：

$$\begin{aligned}
\int_0^{2\pi} \cos mx \cdot \sin nx \mathrm{d}x = 0,\\
\int_0^{2\pi} \sin mx \cdot \sin nx \mathrm{d}x = 0,\\
\int_0^{2\pi} \cos mx \cdot \cos nx \mathrm{d}x = 0 \ (m \neq n).
\end{aligned}$$

我们对函数进行时域采样，从而得到一个向量；或者反过来说，函数可以视为“无穷维的向量” [?]。因此，我们可以有如下直观认识：

(1) **两个向量的点积是两个函数乘积的积分**的离散化版本；

(2) **函数正交性**（函数乘积的积分为 0）可视为**向量正交性**（点积为 0）的推广。

进一步地，依据三角函数的正交性，我们可以证明如果一个向量采样自正弦函数，则只有当另一个向量包含相同频率的正弦分量时，两个向量的点积才不为 0。对于余弦函数有相同的结论。证明过程在此不赘述。

一个函数 $f(x)$ 的傅里叶变换，是用一组正交基（由三角函数构成）表示 $f(x)$；与之形成对比的是：$f(x)$ 的泰勒展开是以 $1, x, x^2, x^3, \cdots$ 为基表示 $f(x)$，而这组基不是正交的。

2.7 矩阵乘法：对二维数组的归约

矩阵乘积（Matrix multiplication problem）

输入：两个 $n \times n$ 的矩阵

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

输出：矩阵乘积 $C = AB$。

依据矩阵乘积的定义，C 的每一个元素 c_{ij} 是 A 的第 i 行与 B 的第 j 列的点积，可以在 $O(n)$ 的时间内计算出来，因此矩阵乘积可以在 $O(n^3)$ 时间内完成。下面我们讲述采用分而治之技术的 Strassen 算法，能够在 $O(n^{2.807})$ 时间内完成矩阵乘积。

首先我们来看如何进行实例的分解。对于数组来说，我们可以依据下标将数组拆分成左一半和右一半；矩阵可以看做是二维数组，因此我们可以同时按照行和列进行拆分，将矩阵分块，表示如下：

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

其中

$$\begin{aligned} C_{11} &= (A_{11}B_{11}) + (A_{12}B_{21}) \\ C_{12} &= (A_{11}B_{12}) + (A_{12}B_{22}) \\ C_{21} &= (A_{21}B_{11}) + (A_{22}B_{21}) \\ C_{22} &= (A_{21}B_{12}) + (A_{22}B_{22}) \end{aligned}$$

因此，我们可以把大的 $n \times n$ 矩阵的乘积，归约成小的 $\frac{n}{2} \times \frac{n}{2}$ 矩阵的乘积。

我们很容易设计出使用这种归约方案的分而治之算法，在此不赘述。这个算法共计算 8 次小的 $\frac{n}{2} \times \frac{n}{2}$ 矩阵的乘积，外加 4 次矩阵加法，因此时间复杂度是：

$$T(n) = 8T(\tfrac{n}{2}) + cn^2 = O(n^3).$$

即使采用了分而治之技术，上述算法的时间复杂度依然是 $O(n^3)$，和传统计算方法的复杂度相同。那么能否设计出更快的算法呢？1969 年，V. Strassen 提出了一种更优的归约方案，能够降低子问题的数目 [**?**]，即首先求解如下 7 个子问题：

$$\begin{aligned}
P_1 &= A_{11}(B_{12} - B_{22}) \\
P_2 &= (A_{11} + A_{12})B_{22} \\
P_3 &= (A_{21} + A_{22})B_{11} \\
P_4 &= A_{22}(B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
P_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
P_7 &= (A_{11} - A_{21})(B_{11} + B_{12})
\end{aligned}$$

然后借助这 7 个小矩阵的乘积，计算出 C：

$$\begin{aligned}
C_{11} &= P_4 + P_5 + P_6 - P_2 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_1 + P_5 - P_3 - P_7
\end{aligned}$$

采用这种归约方案，Strassen 算法共需要计算 7 个子问题，外加 18 次矩阵加减法，因此时间复杂度是：

$$T(n) = 7T(\tfrac{n}{2}) + cn^2 = O(n^{\log_2 7}) = O(n^{2.807}).$$

Strassen 算法的优势是：当 n 比较大时，速度比直接依据定义计算矩阵乘积的方法要快。此外，我们还可以应用 Strassen 算法来快速求解一些相关问题，比如矩阵求逆、行列式计算、图中的三角形计数等；Strassen 的原初目的是“证明求解线性方程组的高斯消元法不是最优的” [**?**]。

与此同时，Strassen 算法的缺点也很明显：当 n 比较小时，Strassen 算法需要较多的矩阵加减法，因此速度反而比较慢；此外，由于需要多次递归调用，Strassen 算法的内存开销较大，数值稳定性较低。

1971 年，J. Hopcroft 和 L. Kerr 证明当采用 2×2 分块时，归约成 6 个子问题是不可能的 [**?**]。既然减少子问题数目是不可行的，就只能采用降低子问题的规模的路

线：当使用 20×20 分块方案时，可以归约成 4460 个子问题，相应的算法时间复杂度是 $O(n^{\log_{20} 4460}) = O(n^{2.805})$；当使用 48×48 分块方案时，可以归约成 47217 个子问题，相应的算法时间复杂度是 $O(n^{\log_{48} 47217}) = O(n^{2.780})$。

2.8 平面上最近点对寻找问题：对点集的归约

1975 年，M. Shamos 和 D. Hoey 总结归纳了计算几何学中几个关键问题，其中之一就是计算平面上的最近点对问题，描述如下：

最近点对寻找问题（Closest pair problem）

输入：平面上的 n 个点 $p_1(x_1, y_1), p_2(x_2, y_2), \cdots, p_n(x_n, y_n)$；

输出：欧式距离最近的两个点。

寻找最近点对的最简单的策略就是计算每两个点之间的距离，这样需要 $O(n^2)$ 的时间；而采用*分而治之*技术，只需 $O(n \log n)$ 的时间即可找到最近点对。

我们首先来看如何对实例进行拆分。由于问题的输入是平面上的 n 个点，因此一个自然的想法是如同矩阵分块一样“横向、纵向各切分一次”，从而把 n 个点分成 4 份。但是这种分法存在很大的缺陷：难以控制 4 份的大小，很容易造成严重的不均匀（图 2.22左），而且后续的“组合”解步骤也比较复杂。因此，我们抛弃分 4 份的策略，依然采用“分 2 份”的策略，即首先依据 x_i 对所有点排序，然后依据中位点即可将点集分成等大的 2 份（在此我们假定 n 个点的横坐标各不相同；即使有相同的情况发生，我们可以施加很小的随机扰动，转换成各不相同的情况）。如图 2.22右所示，这种策略的好处在于分得很均匀。

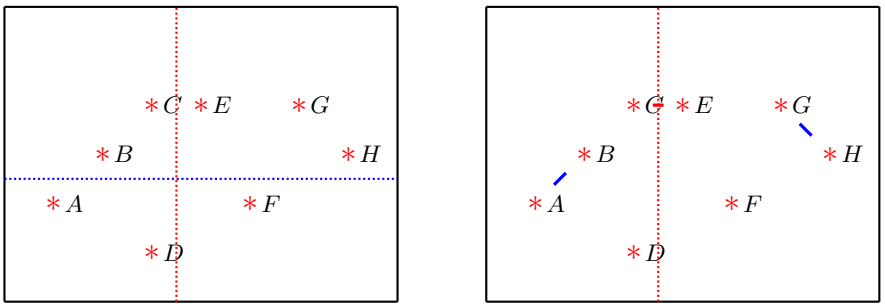


图 2.22: 平面上 8 个点的两种拆分方法：（左）拆分成 4 份，其中右下角内只有一个点 F；（右）均匀拆分成 2 份

在完成点集拆分之后，我们可以把最近点对分成三种类型：*i*) 两个点都在左一半；*ii*) 两个点都在右一半；*iii*) 一个点在左一半、另一个点在右一半。比如在图 2.22所示

点集中，左一半的最近点对是 (A, B)，右一半的最近点对是 (G, H)，而所有点中的最近点对是 (C, E)，距离为 1，是第三种类型。

第一种类型的最近点对可以通过对左一半应用递归调用来完成；类似地，第二种类型的最近点对可以通过对右一半应用递归调用来完成。因此，在将子实例的解“组合”成原始实例的解时，难点在于如何快速计算出第三种类型的最近点对。一种简单方法是计算所有的第三种类型点对之间的距离，然后找出最近点对；由于共有 $\frac{n^2}{4}$ 个点对，因此整个算法的时间复杂度是：

$$T(n) = 2T(\tfrac{n}{2}) + \tfrac{n^2}{4} = O(n^2).$$

这个算法和直接枚举所有点对的策略具有相同的时间复杂度，依然比较慢。

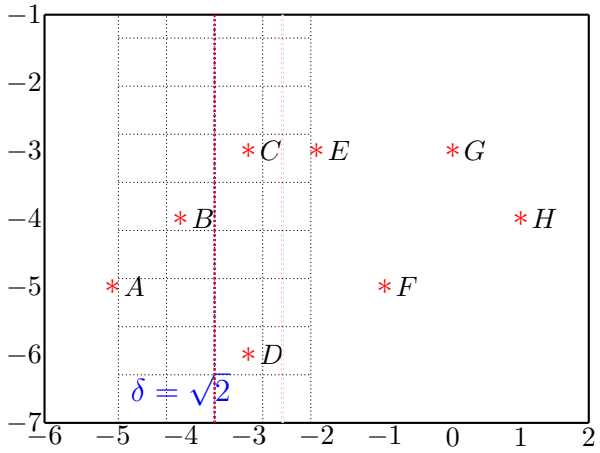


图 2.23: 在 $A, B, C, D4$ 个点中寻找最近点对的“冗余”计算。在红色所示中线两侧各 $\sqrt{2}$ 画竖线，形成 2δ-条带；进而以 $\frac{1}{2}\delta$ 为边长打格子。由于 A 不在条带内，故无需计算 $(A, D), (A, C)$ 之间的距离；点 B 和点 D 的行号相差大于 2，因此也无需计算 (B, D) 之间的距离

要想快速计算出第三种类型的最近点对，其关键在于发现并去除简单方法中的“冗余”计算。如图 2.23所示，我们可以去除下述冗余：

(1) **只需考虑条带内的点对**：以图 2.23所示的在 $A, B, C, D4$ 个点中找最近点对为例，我们在中线两侧距离 $\delta = \min\{\sqrt{2}, 3\}$ 处画两条竖线，形成一个宽 2δ 的条带。我们只需关注两个点都在条带内的点对；换句话说，如果两个点中有一个在条带之外，比如 (A, D)，那么这两个点的横坐标之差都会大于 δ，从而欧式距离大于 δ，不会构成最近点对。

(2) **对于两个点都在条带内的点对，也不是都要考虑，而只需考虑少数的点对**：

为了看清这一点，我们先将条带划分成正方形的“格子”，格子边长为 $\frac{1}{2}\delta$；我们容易证明每个格子内最多只有一个点（假如格子内存在两个点，则两个点之间距离最大是 $\frac{\sqrt{2}}{2}\delta$，比 δ 还要小，与 δ 是第一类、第二类中的最近点对距离矛盾)。

划分格子的好处在于“离散化”：点的坐标值 (x_i, y_i) 可能有小数部分，甚至可能是无理数；但是由于每个格子内只有一个点，从而我们可以用格子的行号和列号来表示一个点，而无需关心这个点的具体坐标值。

进一步地，考虑一个点，设其所在格子的行号为 i、列号为 j，与此点构成最近点对的另一个点只会出现在第 $i, i+1, i+2$ 行的 6 个格子之中。如图2.23所示，点 D 在第 2 行，点 B 在第 5 行，因此 (B, D) 不会构成最近点对。由于 3 行最多有 12 个点，因此我们在对点按照纵坐标排序之后，计算每个点与其后的 11 个点的距离，必定不会遗漏可能构成最近点对的那 6 个点。

算法设计与描述

采用上述的实例拆分方案，并去除冗余计算之后，我们可以设计出如下的算法：

Algorithm 23 Finding closest pair of points

function CLOSEST-PAIR(l, r)

Require: $p_l, ..., p_r$ have already been sorted according to x-coordinate;

1: **if** $r - l == 1$ **then**
2: **return** $d(p_l, p_r)$;
3: **end if**
4: Use the x-coordinate of $p_{\lfloor \frac{l+r}{2} \rfloor}$ to divide $p_l, ..., p_r$ into two halves;
5: δ_1 = CLOSEST-PAIR$(l, \lfloor \frac{l+r}{2} \rfloor)$; //$T(\frac{n}{2})$
6: δ_2 = CLOSEST-PAIR$(\lfloor \frac{l+r}{2} \rfloor + 1, r)$; //$T(\frac{n}{2})$
7: $\delta = \min(\delta_1, \delta_2)$;
8: Sort points within the 2δ-wide strip by y-coordinate; //$O(n \log n)$
9: Scan points in y-order and calculate distance between each point with its next 11 neighbors. Update δ if finding a distance less than δ; //$O(n)$
10: **return** δ;

时间复杂度分析

由于第 8 行对 2δ-条带内的点进行排序，需要 $O(n\log n)$ 的时间，因此整个算法的时间复杂度是：

$$T(n) = 2T(\tfrac{n}{2}) + O(n\log n) = O(n\log^2 n).$$

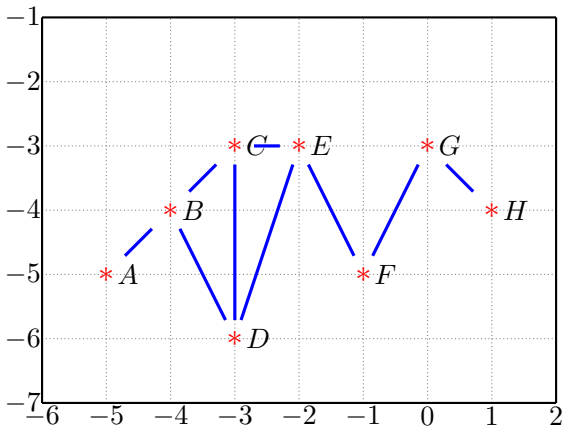


图 2.24: Closest-Pair 算法运行示例：8 个点共有 28 个点对；Closest-Pair 算法只计算了 9 个点对之间的距离（以蓝色表示）

值得指出的是，如果 2δ-条带内的点没有任何有助于排序的结构的话，我们不得不从头开始进行排序，需要 $O(n\log n)$ 的时间开销。因此，要想进一步提高速度，我们必须千方百计引入有助于排序的结构。一种可行的思路是：假如左、右两侧 δ-条带内的点都已事先排好序，则对整个 2δ-条带内的排序时就不必从头开始，只需执行归并排序中的归并操作即可完成排序，时间开销降低为 $O(n)$。这种思路也很容易实现：对输入的 n 个点，我们维护两个排序列表，一个是按照横坐标排序，以便于进行实例的拆分；一个是按照纵坐标排序，以便于进行归并操作。

在改进排序步骤之后，算法的时间复杂度降低为：

$$T(n) = 2T(\tfrac{n}{2}) + O(n) = O(n\log n).$$

图 2.24 展示了 Closest-Pair 算法运行过程的一个示例：对于给定的 8 个点，总共构成 28 个点对；Closest-Pair 算法只计算了其中 9 个点对，即求出最近点对是 (C, E)，距离是 1。

2.9 小结

延伸阅读

彭桓武先生的“分而治之”科研方法

“分而治之”策略不仅可以用于设计算法，还是一种重要的科研方法。我国著名的理论物理学家、“两弹元勋”彭桓武先生曾总结他的研究方法，其中一条就是“将大问题分成小问题研究解决”。

彭桓武先生说：“我曾请教过薛定谔如何做研究，他回答：‘Divide and Command’，即‘分而治之’，这样难点就化开了。其实这句话是古罗马的凯撒大帝说的。实践证明这个方法相当有效，不过要做到这一点，要能将大问题分解开来也是需要相当深厚的功力的。”

子问题大小呈线性降低与指数级降低

逆序数计算与 Kendall“秩”相关系数 τ

当衡量两个变量的相关程度时，通常可以使用 Pearson 相关系数衡量两个变量 x 和 y 的线性相关程度。1904 年，C. Spearman 提出 Spearman 系数 ρ [?] 来衡量变量“秩”的相关程度（Rank correlation），可以视为秩的 Pearson 相关系数。

1938 年，M. Kendall 提出了另一种“秩”相关程度度量—Kendall 系数 τ [?]。简要地说，考察随机变量 x 和 y 的 n 次采样 $x_1, x_2, ..., x_n$ 和 $y_1, y_2, ..., y_n$，Kendall 相关系数定义为：

$$\tau = \frac{2}{n(n-1)} \sum_{i<j} sgn(x_i - x_j) sgn(y_i - y_j).$$

实质上，Kendall 系数 τ 刻画的就是逆序数。

2014 年，Y. Wang 等对 Kendall 系数做了进一步的发展，提出了局部秩相关性系数，能够有效地刻画时序数据（比如蛋白质随时间变化的表达水平）的局部相关性 [?]。

QuickSort 算法的缺陷与改进

如果采用随机选择元素作为中心元的话，QuickSort 算法的期望比较次数是 $O(n\log n)$；在实际应用中，通常采用 R. C. Singleton 的**三元取中法**（Median of three）来选择中心元 [?]，具有优异的性能。

然而 QuickSort 的最坏情形时间复杂度是 $O(n^2)$ 的，因此存在着**复杂度攻击**的可能。事实上，D. R. Musser 构造出一种特定排列的数组（称为"Median of 3 killer"），能够使得 QuickSort 达到其最差性能；更重要的是，D. R. Musser 还证明了这种序列并不少见 [?]。

为克服 QuickSort 的这种缺陷，D. R. Musser 提出了一种**复合**排序算法 IntroSort，其基本思想是**限制迭代深度**：当深度小于 $2\log n$ 时，调用 QuickSort 算法进行排序；当深度达到 $2\log n$ 时，使用堆排序。由于堆排序的最坏情形时间复杂度是 $O(n\log n)$，因此 IntroSort 的平均情况时间复杂度和最坏情形时间复杂度都是 $O(n\log n)$。在包含 100,000 个元素的"Median of 3 killer"数组表明：IntroSort 算法比 QuickSort 算法快 200 倍。

2009 年，E. Upfal 等推广了数组排序，提出了**动态变化数组的排序问题**，即：考虑一个数组，数组中元素随时间**缓慢**地动态变化；查询者在每个时刻任意选择两个元素，询问它们的相对大小，其目标是通过多次查询，确定当前时刻的元素顺序 [?]。Upfal 等设计了能够精确找出最大元素的算法，并证明不能精确获得元素的完整顺序。2017 年，Q. Huang 等考虑"找出前 k 个元素、然后对前 k 个元素进行排序"这一问题 [?]，并设计了一个算法，其基本思想是：循环执行如下两步：i) **粗筛**：用快速排序对所有元素进行排序，然后截取前 $2k$ 名，并证明一定会包含待寻找的前 k 名且顺序差异不大；ii) **细调**：对前 $2k$ 名元素再次排序，截取前 k 名作为最终输出。当 $k \le \sqrt{n}$ 时，此算法可以准确找出前 k 个元素并排序。

Karatsuba 算法和 FFT 算法的思想溯源

值得指出的是，本章里有两处的关键思想都可以追溯到高斯（J. C. F. Gauss）：

(1) Karatsuba 算法将整数乘法归约成 3 个子问题；高斯采用类似的思想简化复数的乘法 [?]：当计算两个复数 $a+b\mathrm{i}$ 和 $c+d\mathrm{i}$ 的乘积时，传统的方法是归约成 4 次实数的乘法，即：

$$(a+b\mathrm{i})\cdot(c+d\mathrm{i}) = (ac-bd)+(bc+ad)\mathrm{i}$$

高斯只使用了 3 次实数乘法，分别计算：

$$k_1 = c \cdot (a + b)$$
$$k_2 = a \cdot (d - c)$$
$$k_3 = b \cdot (c + d)$$

然后可将复数乘积的实部表示成 $k_1 - k_3$，虚部表示成 $k_1 + k_2$。1963 年，P. Ungar 基于 Karatsuba 算法，提出了另一种只需 3 次实数乘法的计算方法 [?]：

$$(a + b\mathrm{i}) \cdot (c + d\mathrm{i}) = (ac - bd) + \big((a + b) \cdot (c + d) - ac - bd\big)\mathrm{i}$$

(2) 1805 年，高斯在计算行星轨道时，需要进行插值运算；高斯提出了类似 Cooley-Tukey 快速傅里叶变换算法的思想，只是缺乏对时间复杂度的分析 [?]。

快速数论变换

如果我们不考虑复数域中的乘法，而是考虑环 Z_P 内的 $\mod P$ 乘法，则可以将 FFT 推广成**快速数论变换**（Number theoretic transform, NTT）。简要地说，对于一个素数 P，令 n 是 $P-1$ 的一个因子（一般取 $n = 2^k$），我们采用模 P 的 n 次原根 g 替代 FFT 中的 n 次单位复根 ω，采用 $g^{\frac{P-1}{n}} \mod P$ 替代 $\mathrm{e}^{-\frac{2\pi}{\mathrm{n}}\mathrm{i}}$。由于 $g^n = 1 \mod P$，因此我们可以采用类似 FFT 的方式快速计算多项式 $A(x)$ 在 $1, g, g^2, \cdots, g^{n-1}$ 处的值。NTT 的优势是避免了复数运算和浮点运算，从而不仅能够避免浮点运算的误差累积，还可以显著提高速度 [?]。

FFT and 整数分解的量子算法 and Prism (TODO)

最近点对问题的扩展

1975 年，M. I. Shamos 和 D. Hoey 提出了计算平面上 n 个点中最近点对的 $O(n \log n)$ 算法 [?]。与之相反的问题是计算最远点对（All-farthest-neighbors）；1977 年，Toussaint 等 [?, ?] 提出了 $O(n \log n)$ 的算法。

如果限制 n 个点是凸多面体的顶点，则任一顶点最多是 6 个顶点的最近邻居；利用这个性质，Lee 等提出了计算最近点对的 $O(n)$ 时间算法 [?]。最远点对可以转化成计算全单调矩阵的行最大值问题，可以使用 SMAWK 算法在 $O(n)$ 时间内计算出来 [?]。

习题

1. 寻找 Monge 矩阵的行最小值。

2. 给定模式串 A 与文本串 B，两个串中只有 26 个大写字母与通配符'?'（即可以任意匹配一个字符），请设计时间复杂度为 $O(n\log n)$ 的算法，计算出 A 在 B 中的匹配数。

3. 你正有兴趣分析某些来自两个分离的数据库中难得的数据。每个数据库中包含 n 个数值，因此一共有 $2n$ 个数值，并且你可以假定没有两个数值是相同的。你想要确定包含这 $2n$ 个数值的集合的中位数，这里我们把它定义为第 n 个最小的数。

   但是你可以访问这些数的唯一方式是通过对数据库的查询。在一次查询中，你可以对这两个数据库中的其中一个指定 k 值，这个被选定的数据库将返回它包含的第 k 个最小的数。由于查询操作是昂贵的，你想要用尽可能少的查询来计算这个中位数。

   请给出一个最多使用 $O(\log n)$ 次查询来找出中位数的算法。

4. 给定一棵二叉树，假定任何两个相连的节点之间的距离为 1 。请给出一个算法来找到一棵二叉树中具有最远距离的两个节点之间的距离。

5. 考虑一棵具有 n 个节点的完全二叉树 T ，其中存在某个 d 使得 $n = 2^d - 1$ 。T 的每个节点 v 用一个实数 x_v 来标记。你可以假定标记这些节点的实数都是不相同的。如果对于所有与节点 v 相连的节点 w ，均有标记 x_v 小于标记 x_w ，则节点 v 是二叉树 T 的一个局部极小点。

   给定这样的一棵完全二叉树 T ，但是标记是以下面这种隐含的方式唯一指定的：对于每个节点 v ，你可以通过探查节点 v 来确定 x_v 的值。给出一个算法，要求只对 T 的节点进行 $O(\log n)$ 次探查来找到 T 的一个局部极小点。

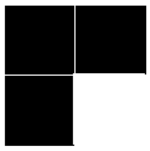
6. 假设现在给定一个 $n \times n$ 的网格图 G（一个 $n \times n$ 的网格图就是一个 $n \times n$ 的棋盘的邻接图。更精确地说，它是一个图，并且它的节点集合是所有自然数的有序对 (i, j) 的集合，其中 $1 \le i \le n$ 且 $1 \le j \le n$ ；节点 (i, j) 与节点 (k, l) 被一条边相连当且仅当 $|i - k| + |j - l| = 1$）。

我们使用题目 3 中的某些术语。同样地，每个节点 v 被一个实数 x_v 标记，你可以假定所有这些标记都是不同的。给出一个算法，要求只对图 G 中的节点进行 $O(n)$ 次探查来找到 G 的一个局部极小点。（注意 G 中有 n^2 个节点。）

7. 回顾求解逆序对数的问题。正如在课程中所讲，给定一个具有 n 个数的序列 $a_1, \cdots, a_n$ ，假设这 n 个数都是不相同的。我们定义一个逆序对是一对 $i < j$ 使得 $a_i > a_j$ 。

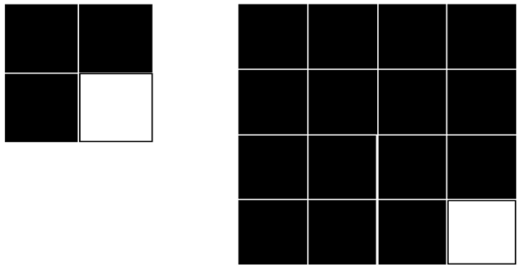
   我们引入逆序对数计算问题来作为衡量两个序列有多大差异的一个好的度量指标。但是，有些人可能认为这个度量指标太过于敏感。如果 $i < j$ 且 $a_i > 3a_j$ ，则我们将该对数值称为明显的逆序。给出一个复杂度为 $O(n \log n)$ 的算法来计算给定序列中明显的逆序对数。

8. 给定一个含有 $2^n \times 2^n$ 个方格的表格，我们想要用 L 型的模块（包含三个方格）去填充它。 L 型的模块如下图所示。



   请给出一个填充方法，使得表格的最后一个元素 $M_{2^n,2^n}$ 为空。

   例如：



第三章　动态规划算法

3.1　引言

在理论研究和实际应用中，我们常常会碰到*最优化问题*：问题的约束条件规定哪样的解是可行解；问题的描述部分包括一个目标函数，其自变量是可行解，其值表征可行解的质量；问题的要求是找出最优的可行解，使得其目标函数值在所有的可行解中是最小的（或者最大的）。

面对一个最优化问题，我们该从何处入手设计求解算法呢?

可行思路之一依然是分而治之策略，即：*先从最简单的子问题入手*，然后探索能*否将规模较大的问题分解成规模较小的子问题*，以及能*否将子问题的解组合成原始问题的解*。不过由于待求解的问题是最优化问题，导致在"解的组合"这一步，我们不再是简单地将子问题的可行解组合成原始问题的可行解，而是要将*子问题的最优解*组合成*原始问题的最优解*。此外，由于最优分解方案无法事先确定，所以这里的"组合"还包括对所有可能的分解方案进行*比较*，以获得原始问题的最优解。

假如原始问题的最优解能够由子问题的最优解组合而成的话（这称作*最优子结构*性质，或*最优性原理*），我们就可以设计如下的求解算法：从最简单的子问题开始，将"小"的子问题的最优解组合成"大"的子问题的最优解；如此不断进行，直至最后组合出原始问题的最优解。这种应用分而治之策略求解最优化问题的方法就是*动态规划*（Dynamic programming）。

那么，如何将一个最优化问题分解成子问题，并进而判定子问题和原问题之间存在着最优子结构性质呢?

一种行之有效的策略是尝试将问题的求解过程描述成*多步决策过程*（Multistage decision process）。事实上，正是在观察了大量多步决策问题的基础上，R. Bellman 首次提出了动态规划算法的一般性框架 [**?**, **?**, **?**]。

所谓多步决策过程，是指完整解能够一步一步地构建出来；在每一步，我们都从

若干决策选项中选择其一，从而确定出完整解的一个分量；把所有步骤的决策合在一起，就能够确定出最终的完整解。在此过程的每一步，之前步骤中已做出的决策构成了*部分解*，而后续步骤中有待做出的决策则构成了待求解的*子问题*。

本质上，多步决策过程表示的是求解思路的转变：对于含多个变量的目标函数来说，我们不是采用"*一步的、直接考虑多个变量*"的思路计算最优解，而是转化成"*多步的、每步考虑单个变量*"的求解思路 [?]。由于动态规划求解的是优化问题，因此不仅对解的性质有要求，同时目标函数也需要满足一定的性质，即：待优化的目标函数也是可分的，可以分做一次决策的收益和剩余子问题的目标函数两个部分。

值得指出的是：动态规划算法的核心是最优子结构性质，要求定义的子问题必须具有*递归性*，即：大的子问题能够分解成小的子问题。通常我们首先尝试直接依照原始问题的形式定义子问题；如果这样定义的子问题具有递归性的话，则可以继续进行算法设计，否则我们就得把原始问题变形，构造出一个有助于求解原始问题、同时又具有递归性的新问题。我们在隐马尔可夫模型解码问题、单源最短路径问题中详细阐述这个设计思路。

在本章里，我们将介绍动态规划算法的设计，并把重点放在如何将一个最优化问题描述成多步决策过程。和上一章类似，我们依据问题形式化描述中"输入部分"的关键数据结构来组织本章内容，分作在数组（序列）、集合、树、有向无环图、一般的图等数据结构上的归约。在下一章里，我们将介绍动态规划的高级技术，包括降低空间和时间开销的策略，以及使用神经网络改进动态规划算法的最新进展。

3.2 矩阵序列的链式相乘问题：对序列的归约

所谓矩阵序列的链式相乘，是指计算多个矩阵的乘积。以下面的 3 个矩阵 A_1, A_2 和 A_3 为例，

$$A_1 = \begin{bmatrix} 1 & 2 \end{bmatrix} \; A_2 = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \; A_3 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

我们可以采用两种顺序来计算矩阵乘积 $A_1A_2A_3$；这两种运算次序用括号序列表示如下:

(1) $A_1A_2A_3 = ((A_1)(A_2))(A_3)$：共需执行 $1 \times 2 \times 3 + 1 \times 3 \times 4 = 18$ 次矩阵元素

乘法；

(2) $A_1A_2A_3 = (A_1)((A_2)(A_3))$: 共需执行 $2 \times 3 \times 4 + 1 \times 2 \times 4 = 32$ 次矩阵元素乘法。

由于矩阵乘法满足结合律，不同的运算顺序并不影响最终算出的乘积，但是所需的矩阵元素乘法（Scalar multiplication）的次数却大相径庭：第一种运算顺序只需 18 次矩阵元素乘法，比第二种运算顺序快得多。因此对于给定的矩阵序列，如何找出矩阵元素乘法次数最少的运算顺序，是值得讨论的问题 [**?**]。我们将这个问题形式化描述如下：

矩阵序列的链式相乘问题（Matrix chain multiplication problem）

输入：一个包含 n 个矩阵 $A_1, A_2, ..., A_n$ 序列，其中矩阵 A_i 的规模是 p_{i-1} 行、p_i 列；

输出：计算矩阵乘积 $A_1A_2...A_n$ 的最优运算顺序，使得需要的矩阵元素乘积运算的次数最小。此处，我们使用*括号序列*表示矩阵乘积的计算次序。

在设计求解算法之前，我们先来看可行解的总数有多少。如图 3.1所示，我们可以把任意一个计算次序表示成包含 $2n-2$ 对括号的括号序列，或者等价地表示成一棵**完全二叉树**（Complete binary tree），其中叶子顶点表示 n 个矩阵，内部顶点表示增加一对括号。我们知道具有 n 个叶子顶点的完全二叉树的总数是 Catalan 数 [**?**]：

$$C_{n-1} = \frac{1}{n}\binom{2n-2}{n-1} = \binom{2n-2}{n-1} - \binom{2n-2}{n-2}$$

这意味着运算顺序的总数是指数量级的，直接枚举耗时太多、难以实用；因此我们面临的挑战是：设计出一种高效的方法，在指数多的运算顺序中，找出所需矩阵元素乘法最少的运算顺序来。

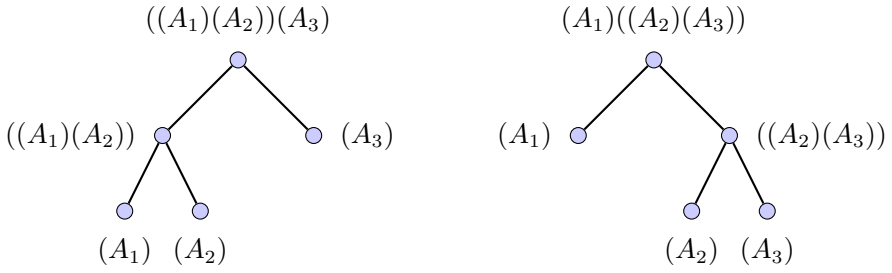


图 3.1: 计算 3 个矩阵 A_1, A_2, A_3 乘积的两种运算顺序，其中每一种运算顺序都对应着一棵完全二叉树

3.2.1 算法设计：一个比较慢的初始版本

既然难以直接求解规模较大的问题，我们就转换一下思路，尝试将规模较大的问题归约成规模较小的子问题。那么怎样对问题进行分解、又怎样将子问题的最优解组合成原问题的最优解呢？

我们尝试从分析解的结构入手，先把求解过程描述成一个多步决策过程，然后依据多步决策过程来定义子问题。

子问题定义

顺着这个思路，我们先考察 4 个矩阵乘积 $A_1A_2A_3A_4$ 的计算：问题的完整解是由 6 对括号表示的运算顺序；求解过程可以用这些括号的“写出”过程来表示：这 6 对括号是分成 3 步写出来的，每一步都添加 2 对括号；由于每一步可以添加括号的位置有多个选择项，因此我们需要做出决策，到底在哪里添加括号。

图 3.2展示添加括号的多步决策过程。以完整解 $(A_1)((A_2)(A_3)(A_4))$ 为例，其写出过程可描述为下述的 2 步决策过程：第一步可以在 3 个位置中选择其一添加括号，我们的决策是在 A_1 和 $A_2A_3A_4$ 之外分别添加 1 对括号，形成部分解 $(A_1)(A_2A_3A_4)$；第二步可以在 2 个位置添加括号，我们的决策是在 A_2A_3 和 A_4 之外分别添加 1 对括号，形成部分解 $(A_1)((A_2A_3)(A_4))$；最后一步无需决策，只需在 A_2 和 A_3 之外分别添加 1 对括号，从而形成完整解。

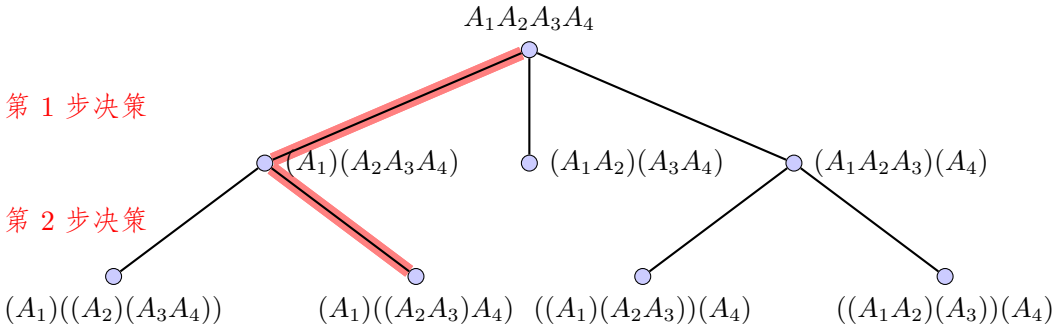


图 3.2: 计算矩阵乘积 $A_1A_2A_3A_4$ 最优计算顺序的 2 步决策过程；其中第一步决策有 3 个可能的决策选项，即：3 个加括号的位置；第二步有两个决策选项，即：两个加括号的位置。粗线展示写出 $(A_1)((A_2A_3)A_4)$ 所做的具体决策

上述观察可以推广到 n 个矩阵乘积 $A_1A_2\cdots A_n$ 的计算：完整解可表示为 $2n-2$ 对括号；这 $2n-2$ 对括号可以分作 $n-1$ 步写出，在其中每一步，我们都是在当前部

分解的基础上新选择一个位置并添加 2 对括号（见图 3.2）。在求解过程中，部分解不断扩充；我们以部分解作为动态系统，将其变化过程描述成如下的 $n-1$ 步决策过程：

(1) **部分解的状态**：和完全解相同，部分解也可表示成括号序列；不同之处仅在于部分解中的括号数目不足 $2(n-1)$ 对。比如只包含 2 对括号的 $(A_1)(A_2A_3\cdots A_n)$ 是一个部分解。作为特例，$A_1A_2A_3\cdots A_n$ 是一个包含 0 对括号的部分解，是多步决策过程的起始状态。

(2) **扩展部分解的决策**：在决策过程的每一步，我们在部分解上选择一个位置，新插入 2 对括号。以第一步为例，我们以 $A_1A_2A_3\cdots A_n$ 为起始状态，在 $n-1$ 个位置中选择一个位置添加 2 对括号，因此部分解的状态转移共有如下 $n-1$ 种可能：$(A_1)(A_2A_3\cdots A_n)$, $(A_1A_2)(A_3\cdots A_n)$, $\cdots$, $(A_1A_2A_3\cdots)(A_n)$。

(3) **决策项的收益**：考虑选择位置 k 添加 2 对括号，形成的部分解是 $(A_1\cdots A_k)(A_{k+1}\cdots A_n)$。我们定义这个选择项的收益是 $p_0p_kp_n$，之所以如此进行定义，是因为 $A_1\cdots A_k$ 是一个 p_0 行、p_k 列的矩阵，而 $A_{k+1}\cdots A_n$ 是一个 p_k 行、p_n 列的矩阵，因此计算 $A_1\cdots A_k$ 和 $A_{k+1}\cdots A_n$ 的乘积需要 $p_0p_kp_n$ 次矩阵元素乘法。如此定义了单步决策收益之后，原始问题的优化目标函数可表示成所有 $n-1$ 步决策的收益总和。

我们通常把多步决策过程形象地表示成一棵分枝树：结点表示**部分解**的状态（部分解与尚待求解的子问题是互补的，因此结点也表示了**子问题**），边表示扩展部分解的**决策项**（见图 3.2）。

在将求解过程描述成多步决策过程之后，我们将**子问题定义**为：在第 t 步（$1\le t<n-1$），对于一个包含 $2(t-1)$ 对括号的部分解来说，如何选择剩余的 $2(n-t)$ 对括号的插入位置，使得这些剩余决策的收益总和最小。

上述子问题定义可进行如下简化：以部分解 $(A_1\cdots A_k)(A_{k+1}\cdots A_n)$ 为例，在此部分解基础上选择 $2n-2$ 对括号的最优添加位置，等价于求解如下两个子问题：计算乘积 $A_1\cdots A_k$ 的最优运算顺序，以及计算乘积 $A_{k+1}\cdots A_n$ 的最优运算顺序。

接下来，我们对这两个具体的子问题进行归纳，总结出**子问题的一般形式**。我们注意到第一个子问题是计算乘积 $A_1\cdots A_k$，其左端 A_1 的下标固定、右端 A_k 下标可变；第二个子问题是计算乘积 $A_{k+1}\cdots A_n$，其左端 A_{k+1} 的下标可变，右端 A_n 下标固定。子问题的一般形式必须能够覆盖这两种情形，因此其两端下标应该都是可变的。

为此，我们将子问题的一般形式设计成：计算乘积 $A_iA_{i+1}\cdots A_j$ 的最优运算次序。我们使用 $OPT(i,j)$ 表示这个子问题的最优解的值，即最少的矩阵元素乘法次数。

对子问题定义合理性的一个检验标准是：能否用子问题表示出原始给定问题的解。对这个问题而言这个检验是成立的：$OPT(1,n)$ 表示计算乘积 $A_1A_2\cdots A_n$ 所需的最少矩阵元素乘法次数，是原始给定问题的求解目标。

子问题最优解之间的递归关系

上述分析已表明 $OPT(1,n)$ 就是原始给定问题的求解目标。那么，如何计算 $OPT(1,n)$ 呢?

R. Bellman 于 1952 年提出的*最优化原理*蕴含了计算 $OPT(1,n)$ 的方法。最优化原理可以表述为：*一个最优多步决策具有如下性质：不管初始状态和第一步决策如何，对于第一次决策导致的变换之后的新状态来说，剩余的决策必须构成最优多步决策* [**?**]。

具体到矩阵乘积问题而言，假如我们知道在计算 $A_1A_2\cdots A_n$ 时，最优解的第一步决策是选择位置 k，并添加 2 对括号，即：$(A_1\cdots A_k)(A_{k+1}\cdots A_n)$，我们就能够得到如下递归表达式：

$$OPT(1,n) = OPT(1,k) + OPT(k+1,n) + p_0p_kp_n,$$

其中 $p_0p_kp_n$ 来源于第一步决策的收益，而 $OPT(1,k)$ 和 $OPT(k+1,n)$ 这两项则有必要解释一下：

(1) $OPT(1,k)$ 和 $OPT(k+1,n)$ 分别来自于计算乘积 $A_1\cdots A_k$ 和 $A_{k+1}\cdots A_n$ 所需的最少矩阵元素乘法次数；换句话说，用计算乘积 $A_1\cdots A_k$ 和 $A_{k+1}\cdots A_n$ 的非最优解，不能够组合出计算乘积 $A_1A_2\cdots A_n$ 的最优解（Bellman 原文中的论断只有一句话：“用归谬法即可得证” [**?**]）。

(2) 这两个子问题是相互独立的（Independent subproblems），即对一个子问题选择运算顺序，不会干扰另一个子问题运算顺序的选择。

然而我们无法预先知道最优解的第一步决策到底是选择哪个决策项，因此我们不得不枚举 k 的所有可能取值，并且依据子问题的最优解的值，才能确定出运算次数最少的那个决策项，即：

$$OPT(1,n) = \min_{1\le k<n}\{OPT(1,k) + OPT(k+1,n) + p_0p_kp_n\}.$$

推广到一般的子问题，我们有如下递归关系及基始情形：

$$OPT(i,j)=\begin{cases}0 & \text{如果} i=j\\ \min\limits_{i\le k<j}\{OPT(i,k)+OPT(k+1,j)+p_{i-1}p_kp_j\} & \text{否则}\end{cases}$$

子问题最优解之间的递归关系，也被称为 Bellman 方程（Bellman equation [**?**]），是动态规划算法的核心。

要想依据上述递归表达式计算出 $OPT(i,j)$，最简单、直观的方式是使用递归调用技术，将上述递归表达式直译成如下的递归程序：

Algorithm 24 计算矩阵链式乘积的递归算法

function RECURSIVE-MATRIX-CHAIN($P=p_0p_1\cdots p_n$, i, j)

```
1: if i == j then
2:     return 0;
3: end if
4: OPT(i, j) = +∞;
5: for k = i to j − 1 do
6:     v = RECURSIVE-MATRIX-CHAIN(P, i, k) + RECURSIVE-MATRIX-CHAIN(P, k + 1, j) +
       p_{i−1} p_k p_j;
7:     if v < OPT(i, j) then
8:         OPT(i, j) = v;
9:     end if
10: end for
11: return OPT(i, j);
```

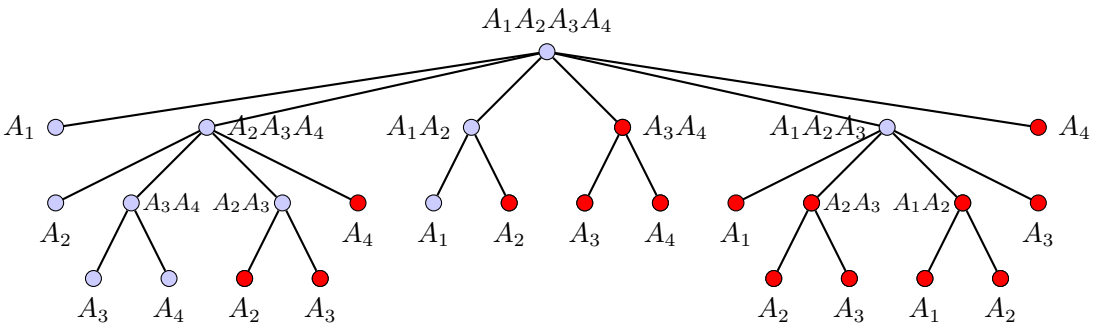


图 3.3: 算法 RECURSIVE-MATRIX-CHAIN 计算乘积 $A_1A_2A_3A_4$ 时的递归调用树，其中每个顶点表示一个子问题，红色顶点表示被重复求解的子问题

图 3.3展示 RECURSIVE-MATRIX-CHAIN 算法计算 4 个矩阵乘积 $A_1A_2A_3A_4$ 时的递归调用树；树中的每一个顶点表示一个子问题，边则表示子问题之间的归约关系。

Recursive-Matrix-Chain 算法是对递归表达式的忠实翻译，其正确性是毋庸置疑的；但是时间复杂度却很高，是指数级的。我们将此结论严格表述成下述定理：

定理 3.2.1. *令 $T(n)$ 表示计算 n 个矩阵乘积 $A_1A_2\cdots A_n$ 时，算法 Recursive-Matrix-Chain 的时间复杂度，则有 $T(n) \geq 2^{n-1}$。*

证明： 我们使用数学归纳法进行证明。

首先，当 $n = 1$ 时，我们有 $T(n) = 1 = 2^{1-1}$，结论成立。

其次，当 $n > 1$ 时，我们有如下递归关系：

$$T(n) \geq 1 + \sum_{k=1}^{n-1} \big(T(k) + T(n-k) + 1\big).$$

假设对任意的整数 $1 \leq m < n$，$T(m) \geq 2^{m-1}$ 成立，则可推出：

$$\begin{aligned} T(n) &\geq 1 + \textstyle\sum_{k=1}^{n-1} \big(T(k) + T(n-k) + 1\big) \\ &= n + 2\textstyle\sum_{k=1}^{n-1} T(k) \\ &\geq n + 2\textstyle\sum_{k=1}^{n-1} 2^{k-1} \\ &\geq n + 2(2^{n-1} - 1) \\ &\geq 2^{n-1} \end{aligned}$$

□

3.2.2 避免子问题的重复计算：“以存代算”

我们知道当计算乘积 $A_1A_2\cdots A_n$ 时，形如 $A_i\cdots A_j$ 的子问题只有 $\frac{1}{2}n(n+1)$ 个；那么为何算法 Recursive-Matrix-Chain 会花费指数多的时间呢？仔细检查图3.3所示的递归调用树，即可发现原因在于有大量的子问题被重复计算（图中红色顶点所示）。

如果两个子问题能够分解出相同的更小的子问题（Subsubproblem）的话，我们就称这两个子问题是*重叠子问题*（Overlapping subproblems）。比如子问题 $A_2A_3A_4$ 和 $A_1A_2A_3$ 是重叠子问题，因为它们都能分解成更小的子问题 A_2A_3。

子问题的重复计算是造成 Recursive-Matrix-Chain 算法低效的根本原因。去除这些重复计算的一个简单而有效的方法是：*以存代算*（Memorize），即：*建一个表格，存放所有已求解过的子问题的解*$OPT(i,j)$。当要求解一个子问题时，先查表看是否已求解过；如果已求解过，则直接返回表格中存放的解，否则才继续求解（值得指

出的是："以存代算"策略有时会需要很大的存储空间；为降低空间需求，我们在高级动态规划部分还会看到"以算代存"策略）。

这种"以存代算"策略很容易实现：只需在 RECURSIVE-MATRIX-CHAIN 算法的前面加一个判断语句即可完成，伪代码实现见算法 25。

Algorithm 25 计算矩阵链式乘积的快速递归算法

function MEMORIZE-MATRIX-CHAIN(i, j)

```
1: if OPT[i, j] has been calculated then
2:     return OPT(i, j);
3: end if
4: if i == j then
5:     OPT[i, j] = 0;
6: else
7:     for k = i to j − 1 do
8:         v = MEMORIZE-MATRIX-CHAIN(i, k) + MEMORIZE-MATRIX-CHAIN(k + 1, j) +
           p_{i−1} p_k p_j;
9:         if v < OPT[i, j] then
10:            OPT[i, j] = v;
11:        end if
12:    end for
13: end if
14: return OPT[i, j];
```

然而 MEMORIZE-MATRIX-CHAIN 算法仍然存在着一些不足之处：MEMORIZE-MATRIX-CHAIN 算法采用的是递归调用技术，时间复杂度是 $O(n^3)$，是多项式级的。虽然这个算法理论上很快，但是从程序的实际运行时行为来看，每次递归调用时都有较大的时间开销（包括保存和恢复递归调用的返回地址、调用的参数传递，以及局部变量占用空间的申请与释放等），从而导致大 O 符号中的常数因子较大，程序的效率较低。

为进一步提高速度，我们舍弃递归调用，改用迭代技术来实现递归表达式。要把一个递归算法改成迭代算法，只需要按照下面的步骤操作即可：分析子问题之间的依赖关系；先求解被依赖的子问题，再求解依赖条件已满足的子问题，如此逐步迭代进行，直至求解出给定的原始问题。

从图 3.3可以看出：叶子结点对应的子问题不依赖于其他子问题，可以直接求解；

在求解了所有叶子结点对应的子问题之后，倒数第二层结点对应的子问题的依赖条件满足，可以进行求解；类似地，在求解了所有叶子结点和倒数第二层结点对应的子问题之后，倒数第三层结点对应的子问题的依赖条件满足，可以进行求解；其他结点以此类推。采用迭代技术的算法伪代码描述见算法 26。

Algorithm 26 求解矩阵链式相乘最优顺序的迭代算法

function ITERATIVE-MATRIX-CHAIN$(p_0, p_1, ..., p_n)$

```
1: Set OPT(i,i) = 0 for all i (1 ≤ i ≤ n);
2: for l = 2 to n do
3:    for i = 1 to n − l + 1 do
4:       j = i + l − 1;
5:       OPT(i,j) = +∞;
6:       for k = i to j − 1 do
7:          v = OPT(i,k) + OPT(k+1,j) + p_{i−1} p_k p_j;
8:          if v < OPT(i,j) then
9:             OPT(i,j) = v;
10:            SPLITTER(i,j) = k;
11:         end if
12:      end for
13:   end for
14: end for
15: return OPT(1,n);
```

直观地看，MEMORIZE-MATRIX-CHAIN 算法采用递归策略，是一种“自顶向下”（Top-down）求解方式；与之相反，ITERATIVE-MATRIX-CHAIN 算法采用迭代策略，是一种“自底向上”（Bottom-up）求解方式。

3.2.3 算法运行过程示例

图 3.4展示算法 ITERATIVE-MATRIX-CHAIN 计算 4 个矩阵乘积 $A_1A_2A_3A_4$ 的最优运算顺序的过程，其中 A_1 有 1 行 2 列，A_2 有 2 行 3 列，A_3 有 3 行 4 列，A_4 有 4 行 5 列，即：$p_0 = 1, p_1 = 2, p_2 = 3, p_3 = 4, p_4 = 5$。

从图中可以看出，OPT 表格中主对角线上的单元（即 $OPT(i,i)$）初始值为 0；在第一轮迭代时，计算下一条对角线上的单元（即 $OPT(i,i+1)$）：

$$OPT[1,2] = p_0 \times p_1 \times p_2 = 1 \times 2 \times 3 = 6$$

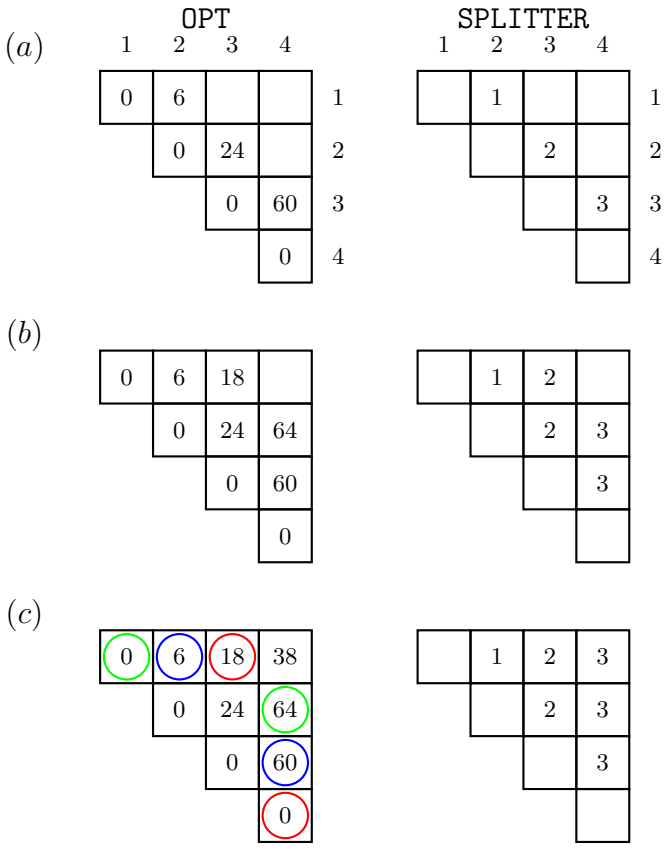


图 3.4: 算法 ITERATIVE-MATRIX-CHAIN 计算 $A_1A_2A_3A_4$ 乘积的运算过程。(*a*) 第一次迭代；(*b*) 第二次迭代；(*c*) 第三次迭代

$$OPT[2,3] = p_1 \times p_2 \times p_3 = 2 \times 3 \times 4 = 24$$

$$OPT[3,4] = p_2 \times p_3 \times p_4 = 3 \times 4 \times 5 = 60$$

在第二轮迭代时，接着计算下一条对角线上的单元（即 $OPT(i, i+2)$）：

$$OPT[1,3] = \min \begin{cases} OPT[1,2] + OPT[3,3] + p_0 \times p_2 \times p_3 \ (=18) \\ OPT[1,1] + OPT[2,3] + p_0 \times p_1 \times p_3 \ (=32) \end{cases} = 18$$

$$OPT[2,4] = \min \begin{cases} OPT[2,2] + OPT[3,4] + p_1 \times p_2 \times p_4 \ (=90) \\ OPT[2,3] + OPT[4,4] + p_1 \times p_3 \times p_4 \ (=64) \end{cases} = 64$$

在第三轮迭代时，我们最终计算出 $OPT(1,4)$ 的值：

$$OPT[1,4] = \min \begin{cases} OPT[1,1] + OPT[2,4] + p_0 \times p_1 \times p_4 \ (=74) \\ OPT[1,2] + OPT[3,4] + p_0 \times p_2 \times p_4 \ (=81) \\ OPT[1,3] + OPT[4,4] + p_0 \times p_3 \times p_4 \ (=38) \end{cases} = 38$$

至此，我们部分解答了原问题：计算 4 个矩阵乘积 $A_1A_2A_3A_4$，最少需要 38 次矩阵元素乘法。之所以说只是部分解答，是因为我们只知道最少需要 38 次矩阵元素乘法，但是却不知道到底哪样的计算顺序是最优的。这是由于我们采用“自底向上”的迭代策略造成的。

为求出最优的计算顺序，我们增添两个附加步骤：

(1) **记录步骤**：在求解 $OPT(1,n)$ 的过程中，**记录每一步决策时的最优决策选项**（算法第 10 行）。详细地说，我们新增加一个表格 $SPLITTER$，其中 $SPLITTER[i,j]$ 记录 $OPT(i,j)$ 是从哪个决策项来的。以 $SPLITTER[1,4]$ 为例：$OPT[1,4]=38$ 是对 3 个决策项（$k=1,2,3$）进行比较，然后选最优决策项（$k=3$）而获得的；我们做赋值 $SPLITTER[1,4]=3$，以记录最优决策项。

(2) **回溯步骤**：在计算出 $OPT(1,n)$ 之后，我们从 $SPLITTER[1,n]$ 得知 $OPT(1,n)$ 来源于哪个决策项，并在相应的位置添加括号；然后依据这个决策项确定子问题，进而追踪子问题最优解的来源，如此逐步进行即可获得完整的括号序列。这个过程和 $OPT(1,n)$ 的迭代求解过程方向完全相反，因此称为**回溯**（Backtracking）。回溯最优解是和迭代计算配套使用的；如果采用递归计算的话，每一步的最优决策都能直接得到，就无需回溯步骤了。

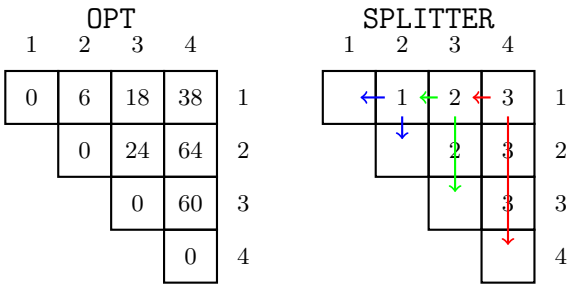


图 3.5: 计算乘积 $A_1A_2A_3A_4$ 最优运算顺序的回溯过程

如图 3.5所示，我们从 $SPLITTER[1,4]$ 开始回溯：$SPLITTER[1,4]=3$ 表示计算乘积 $A_1A_2A_3A_4$ 这一步的最优决策项是 $k=3$，即分别计算 $A_1A_2A_3$ 和 A_4，用括号表示如下：

$$(A_1A_2A_3)(A_4)$$

接下来，$SPLITTER[1,3]=2$ 表示计算乘积 $A_1A_2A_3$ 这一步的最优决策项

是 $k=2$，即分别计算 A_1A_2 和 A_3，用括号表示如下：

$$((A_1A_2)(A_3))(A_4)$$

最后，$SPLITTER[1,2]=1$ 表示计算乘积 A_1A_2 这一步的最优决策项是 $k=1$，即分别计算 A_1 和 A_2，用括号表示如下：

$$(((A_1)(A_2))(A_3))(A_4)$$

至此我们获得了表征最优运算顺序的完整括号序列。从这个例子，我们可以清晰看出最优解可拆分成“第一步决策的选择、由其余步骤决策构成的子问题最优解”两部分（见图 3.2）。

3.2.4 时间复杂度分析

动态规划算法的时间复杂度非常容易分析：用子问题的数目乘以每个子问题的求解时间即可获得。这是因为动态规划算法执行过程的主要操作是“填写”OPT 表格；因此其运行时间由OPT 表格中单元数目和计算每个单元所需的基本操作数二者的乘积确定。

以 Iterative-Matrix-Chain 算法为例：OPT 表格共有 $O(n^2)$ 个单元；计算每个单元时，最多需要在 n 个决策项中进行比较，因此总的时间复杂度是 $O(n^3)$。

类似地，Memorize-Matrix-Chain 算法避免了子问题的重复计算，对每个子问题只需求解一次；在求解每个子问题时，`for` 循环至多执行 n 次，因此求解 $O(n^2)$ 个子问题的总时间是 $O(n^3)$。

值得指出的是，这种分析方法只是时间复杂度的粗略估计，我们在下一章将会看到更细致的时间复杂度分析方法。

3.2.5 一些讨论

递归调用实现 v.s. 迭代实现

从实际运行速度来看，由于避免了递归调用的额外开销，Iterative-Matrix-Chain 算法比 Memorize-Matrix-Chain 算法更快。通常来说，在实现动态规划算法时，采用迭代技术的实现方式比采用递归调用技术的实现方式要更加高效。

然而值得特别指出的是：这并不表明采用递归调用技术的实现方式就一无是处；在某些情形下，采用递归调用技术的实现方式反而更快。其原因在于：采用迭代技术时，所有的子问题都要被计算；与之对比，采用递归调用技术时，我们可以只计算那

些实际会用到的子问题，因此反而可以更快。我们在背包问题的动态规划算法设计中将详细讨论这一点。

指数多的可行解 v.s. 多项式时间求解算法

在本节的一开始，我们就认识到所有可行解的总数是指数多的，而 Iterative-Matrix-Chain 算法却只需 $O(n^3)$ 的时间即可从指数多的可行解中找出最优解来。

这个看似不可能完成的目标，是如何做到的呢?

我们仔细分析 Iterative-Matrix-Chain 算法的运行过程，就会发现高效率的根本原因在于*避免了冗余计算：以子问题的非最优解为组成部分的可行解，根本没有希望成为原问题的最优解，因此完全可以不加考虑*。Iterative-Matrix-Chain 算法在每求出一个子问题的最优解之后，就不再考虑非最优解；更重要的是，也不再考虑以子问题的非最优解为组成部分的所有可行解。从这个角度来说，对可行解的逐个检查会包含大量的冗余操作；而动态规划算法往往能够避免这样的冗余操作，从而显著提高了找出最优解的速度。这种避免冗余操作生动体现了最优化原理的作用。

以 3.1.3 小节中的 4 个矩阵为例，Iterative-Matrix-Chain 算法没有检查 $((A_1)((A_2)(A_3)))(A_4)$ 这个可行解，原因很简单：当计算出 $OPT(1,3)$ 之后，我们就知道对于 3 个矩阵乘积 $A_1A_2A_3$ 来说，$(A_1)((A_2)(A_3))$ 不如 $((A_1)(A_2))(A_3)$，从而可以推论出 $((A_1)((A_2)(A_3)))(A_4)$ 根本不可能是计算 4 个矩阵乘积 $A_1A_2A_3A_4$ 的最优解，因此也就没有检查的必要。

总结一下，动态规划只枚举所有子问题的解（有时甚至都没有枚举所有子问题的解），并没有枚举所有的可行解。当子问题只有多项式个时，动态规划的优势就突显出来了。

动态规划中的回溯法 vs. 博弈论中的倒推法

在动态规划中，最优解不是直接求出来的，而是在求得最优解的值之后，采用回溯法逐步逆向确定的。这一点和博弈论中确定最优策略的*倒推法*很类似：博弈论中寻找的“*与其说是一种从开始来看最优的策略，不如说是一种从博弈的结局来看最优的策略*”[?]。

从研究历史上来看，在 1948 年左右，RAND 公司的研究课题之一就是博弈论的应用，而 von Neumann 也经常访问 RAND 公司；将大量实际问题抽象建模成多步决策问题，也是从博弈论研究开始的 [?]。因此，有研究者认为 Bellman 提出的动态规

划中的回溯法，很有可能是受 von Neumann 博弈论中倒推法的启发 [?]。

3.3 动态规划与多步决策过程

在上一节中，我们结合矩阵链式乘法问题，简要介绍了多步决策过程；在本节里，我们详细介绍多步决策过程。

能够将求解过程描述成多步决策过程的典型情况是：问题的**完整解**能够表示成多个分量 $x_1, x_2, \cdots, x_n$，其中每个分量 x_i 都有多个可能取值。在这种情况下，有两种求解策略可资采用：i) 一次性确定所有 n 个分量取值的策略；ii) 一个分量一个分量地逐步确定的策略。多步决策过程描述的是第二种求解策略；在每一步决策时，比如在第 i 步，我们只知道部分分量 $x_1, \cdots, x_{i-1}$ 的取值，称为**部分解**。

宽泛地说，多步决策过程的研究对象是一个状态可以变化的**动态系统**；而当具体落实到最优化问题的求解过程时，不断发生变化的是部分解，因此是以部分解作为动态系统。我们**将最优化问题的求解过程描述成如下的多步决策过程**：

(1) **部分解的状态**：之所以定义部分解 $x_1, \cdots, x_{i-1}$ 的**状态**，是为了表示在下一步决策时对 x_i 取值的限制。如果动态系统的研究对象是真实存在的**物理系统**（Physical system），状态定义通常比较直观，一般直接使用部分解 $x_1, \cdots, x_{i-1}$ 的具体取值作为状态。比如下一节要讲的背包问题中，背包中已装入的物品就是状态。然而对于抽象的**概念性系统**（Conceptual system）来说，状态的定义会比较抽象。比如在 3.12节中的整数规划问题中，我们是以部分解取值的一个函数作为状态。

(2) **扩展部分解的决策**：当已知部分解 $x_1, \cdots, x_{i-1}$ 的取值时，接下来需要确定分量 x_i 的取值，即需要在 x_i 的多种可能取值中选择其一，称为做一次**决策**（Decision）。确定 x_i 的取值会扩展部分解，导致部分解的状态发生**转移**（State transformation）。这里，我们假定状态转移具有马尔可夫性质，即：一次决策引发的状态转移，仅取决于当前的状态以及当前是第几步决策，而与历史状态无关 [?]。

(3) **决策的收益**：我们为 x_i 的每一个取值选项赋予一个**收益**（Reward），以表征当选择相应的取值时能够得到的“好处”。我们需要合理地设计收益，使得最优化问题的目标函数值就是所有步骤的决策收益总和。

当把最优化问题的求解过程表示成多步决策过程之后，定义子问题就变成很自然的一件事：从多步决策过程的视角来看，部分解 $x_1, \cdots, x_{i-1}$ 的取值表示已经做出的决策，$x_i, \cdots, x_n$ 则表示待做的决策；在获得部分解之后，剩下的问题是如何将其扩展成完整解。因此，我们可以如下**定义子问题**：当已知部分解 $x_1, \cdots, x_{i-1}$ 的取值时，在后续决策时如何选择 $x_i, \cdots, x_n$ 的取值，使得后续步骤的决策收益总和最大。直观上看，子问题和部分解是“互补”的。

对于有些问题而言，我们能够使用子问题的最优解组合成原始问题的最优解。这种递归关系有多种名称：在最优控制领域中，被称为**最优性原理**或 Bellman **方程**（Bellman equations）[?]；而在计算机科学领域，则被称作**最优子结构性质**（Optimal substructure）。

值得指出的是：通常我们仅依赖**部分解**即可定义子问题，但有时这种子问题表示方案比较“粗糙”，导致难以刻画出子问题最优解之间的递归关系。在这种情况下，我们可以把子问题“拆分”得更“细”一些：用“**部分解、剩余决策的性质**”二者的组合来表示子问题，这里解的性质包括剩余决策的步数、做下一步决策时 x_i 的选择项等。采用这种方式，我们通常可以**定义多个递归变量**，并相应地**建立多个递归表达式**，从而使得刻画子问题之间的递归关系更加容易。关于这一点，我们会在 HMM 解码问题、树上的顶点覆盖问题、单源最短路径部分详细阐述。

动态规划算法依赖回溯技术确定出最优解，或者更确切地说，最优的多步决策。以第 1 步决策为例，我们需要从 x_1 的多种可能取值中选择其一；然而，哪种取值才是最优决策，是无法仅凭 x_1 各取值选项的收益即可直接确定下来的，而需要综合考虑收益、子问题最优解的值才可确定。这是需要采用回溯过程的根本原因。

3.4 0-1 背包问题：对集合的归约

0-1 背包问题是在日常生活中经常会遇到的一个最优化问题：如图 3.6所示，我们有一个最多装 15kg 的背包；现有 5 个物品，其重量分别是 12kg, 1kg, 4kg, 2kg 和 2kg，其价值分别是 4 元、2 元、10 元、1 元和 2 元。请问：在不超过背包承重量的前提下，选择哪些物品装进背包里，装的物品价值总和最大?

我们把这个问题形式化描述如下：

> **0-1 背包问题（$0-1$ Knapsack problem）**
> **输入**：背包的承重量 W，n 个物品的集合 $I=\{I_1, I_2, \cdots, I_n\}$，其中第 i 个物品 I_i 的重量是 w_i，价值是 v_i；
> **输出**：I 的一个子集 I'，使得 I' 中的物品在总重量不超过 W 的前提下，总价值最大。

此处，“0-1”的意思是每个物品要么选（用 1 表示），要么不选（用 0 表示）；但是物品不能分割，不能只选一部分装入背包（一个直观的理解是“金币”和“金粉”的差异）。

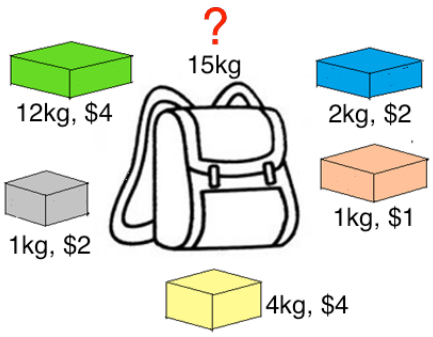


图 3.6: 0-1 背包问题示例，其中物品从左上角至右上角、按逆时针序分别是 1, 2, 3, 4, 5 号

3.4.1 算法设计：一个比较慢的初始版本

首先我们来考虑可行解的数目：当 n 比较小时，可能的物品组合比较少，背包问题还比较容易求解；然而当 n 和 W 都较大时，满足总重量不超过 W 的物品组合非常多，导致简单枚举所有的物品组合这一策略会非常慢。因此，我们来考虑能否将其分解成规模较小的子问题。

子问题定义

我们从分析解的结构入手，尝试将求解过程描述成一个多步决策过程：注意到问题的完整解是“物品集合 I 的一个子集 I'”；我们可以把完整解描述成从空集开始、每次选择并添加一个物品逐渐扩展而成的。这个多步决策过程详细描述如下：

(1) 部分解的状态：我们用“已装入背包的物品构成的子集”来表示部分解；与部分解相对应的背包剩余承重量可以很容易地计算出来。

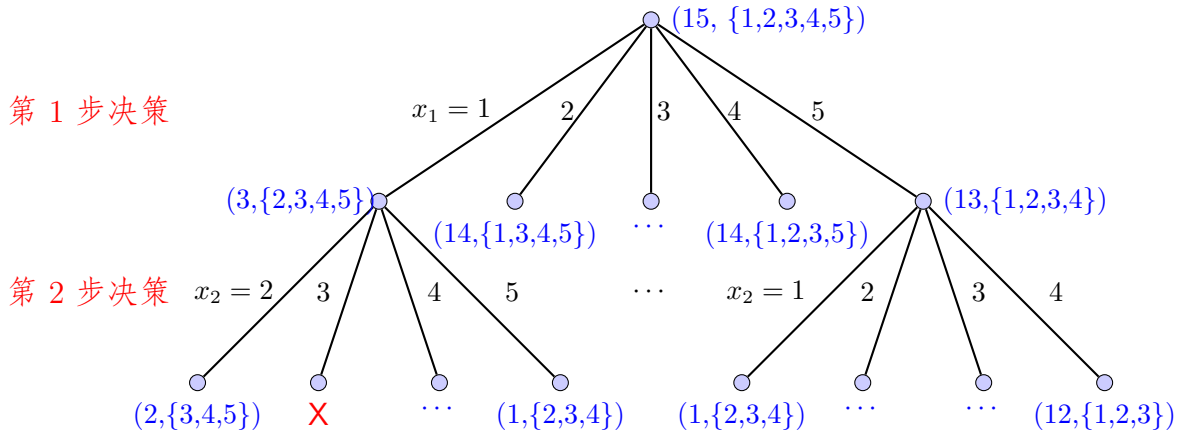


图 3.7: 求解 0-1 背包问题的第一种多步决策过程（此处仅列出前 2 步），其中的每一步决策都是“从尚未装入的物品中选择一个装入背包”。例如：$x_1 = 5$ 表示“第 1 步决策时，选择物品 5 装入背包”。括号内表示“系统的当前状态”。例如：$(14, \{1, 3, 4, 5\})$ 表示“背包还有 14 公斤剩余承重量时、还有物品 1, 3, 4, 5 号物品尚未装入”

(2) **扩展部分解的决策**：在每一步，尚未装入背包的物品都是决策的一个选择项；我们从中选择一个不超过背包剩余承重量的物品装入背包；如此重复，直至即将超过背包承重量或者所有物品都已经装入。

(3) **决策项的收益**：选择物品 I_i，我们将得到收益 v_i。因此，待优化的目标函数等价于所有步骤决策的收益总和。

基于上述多步决策过程，我们将子问题定义为：在获得一个部分解之后，我们能够推算出背包的剩余承重量，剩下的子问题是如何确定后续步骤的决策，使得后续决策的收益总和最大。

以第一步决策为例，最优解是在如下 n 种选择项中做出最优选择：选物品 I_1，选物品 I_2，...，选物品 I_n。假设最优解是选物品 I_i 装入背包，则我们已经获得的价值是 v_i，而背包的承重量只剩余 $W - w_i$，因此剩下的子问题就是：如何在背包还有 $W - w_i$ 公斤剩余承重量的情况下，在剩余物品集合 $I - \{I_i\}$ 中选择一些物品装入背包，使得总价值最大。

注意到在上述子问题中，背包的剩余承重量不是固定的，剩余物品集合也不是固定的，因此我们将子问题的一般形式定义为：在背包还有 w 公斤剩余承重量的情况下，在剩余物品集合 S 中选择一些物品装入背包，使得总价值最大。我们用 $OPT(S, w)$ 表示能够装入背包的物品价值的最大值；原始给定问题的最优解的值可以用 $OPT(I, W)$ 来表示。

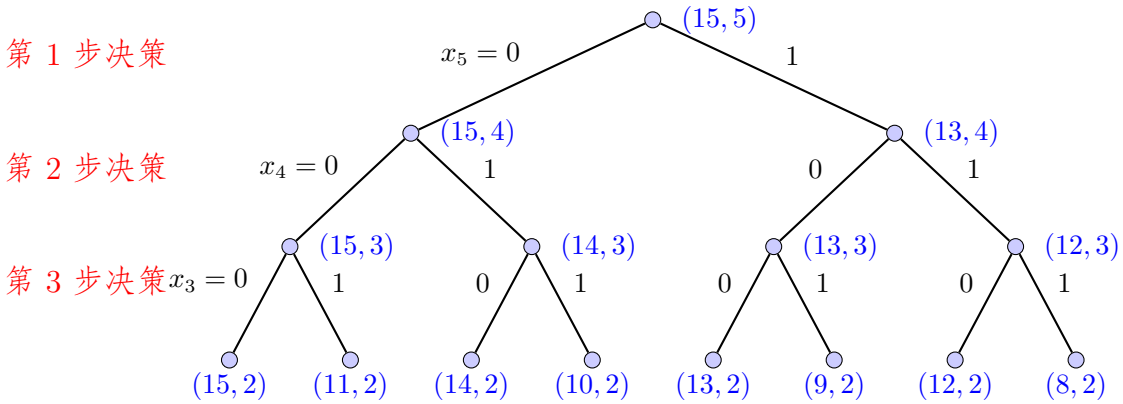


图 3.8: 求解 0-1 背包问题的第二种多步决策过程（此处仅列出前 3 步），其中每一步的决策都是“是否”型的，即：决定“是否装入某个物品”。以第 1 步决策为例，$x_5 = 0$ 表示决策项“不装 5 号物品”，$x_5 = 1$ 表示决策项“装 5 号物品”。括号内表示“当前的状态”。比如 $(15, 5)$ 表示“背包还有 15 公斤剩余承重量、还要考虑前 5 个物品”

子问题最优解之间的递归关系

由于无法预知最优解的第一步决策到底是选择了哪个物品 I_i，我们只好枚举所有可能的 I_i，从而得到如下递归关系及基础条件：

$$OPT(S, w) = \begin{cases} \max\limits_{I_i \in S, w_i \le w} \{v_i + OPT(S - \{I_i\}, w - w_i)\} & \text{如果} \exists I_i \in S, w_i \le w \\ 0 & \text{否则} \end{cases}$$

上述递归表达式的正确性是毋容置疑的；依据这个表达式设计动态规划算法，是能够正确计算出最优解来的。然而这样的动态规划算法却存在着严重的缺陷：S 是 I 的一个子集，而所有可能的子集 S 共有指数多个，从而导致子问题的数目很大，存放最优解 $OPT(w, S)$ 的表格很大，算法的时间复杂度很高。

3.4.2 算法设计：改进后的版本

那么怎样才能设计出更高效的算法呢？

我们注意到上述缺陷的根源出自于子问题的定义上：只要对任意的子集S 都定义一个子问题，就不可避免地导致子问题的数目有指数多。因此我们的思考方向是：如何重新定义子问题，使得仅对一部分子集 S、而不是任意子集 S，有对应的子问题。

为此，我们换一种方式描述求最优解的多步决策过程：按照 $I_n, I_{n-1}, \cdots, I_1$ 的顺序从后向前逐个考虑每个物品（也可采用从前向后的顺序）；在第 i 步，我们考虑物品 I_i，需要在两个选择项里做出决策：i) 装入 I_i；ii) 不装入 I_i。

以第一步决策为例，最优解中有如下两个选择项：

(1) 装入物品 I_n：则剩下的子问题是“如何在背包剩余承重量是 $W-w_n$ 的前提下，在前 $n-1$ 个物品 $\{I_1, I_2, \cdots, I_{n-1}\}$ 中找出价值总和最大的物品子集”；

(2) 不装入物品 I_n：则剩下的子问题是“如何在背包剩余承重量是 W 的前提下，在前 $n-1$ 个物品 $\{I_1, I_2, \cdots, I_{n-1}\}$ 中找出价值总和最大的物品子集”。

归纳这两种情形，我们将子问题的一般形式定义为：在背包剩余承重量是 $w \geq 0$ 的前提下，在前 i 个物品 $\{I_1, I_2, \cdots, I_i\}$ 中找出价值总和最大的物品子集，并将这个最大的价值记为 $OPT(i, w)$。

由于无法预知最优解中每一步决策到底选择了哪个决策项，我们只好枚举两种决策项，从而得到如下的递归关系及基始步骤：

$$OPT(i,w) = \begin{cases} 0 & \text{如果} i=0 \text{或} w=0 \\ OPT(i-1,w) & \text{如果} w \leq w_i \\ \max\{OPT(i-1,w), v_i + OPT(i-1,w-w_i)\} & \text{否则} \end{cases}$$

采用这种子问题定义，原始给定问题的最优解的值可以用 $OPT(n, W)$ 来表示。我们采用迭代技术计算 $OPT(n, W)$，求解算法描述如下：

Algorithm 27 求解 0-1 背包问题的迭代算法

function ITERATIVE-KNAPSACK(n, W)

```
1: for w = 1 to W do
2:     OPT[0, w] = 0;
3: end for
4: for i = 1 to n do
5:     for w = 1 to W do
6:         if w ≥ w_i then
7:             OPT[i, w] = max{OPT[i − 1, w], v_i + OPT[i − 1, w − w_i]};
8:         else
9:             OPT[i, w] = OPT[i − 1, w];
10:        end if
11:    end for
12: end for
13: return OPT[n, W];
```

3.4.3 算法运行过程示例

我们考虑如下的 0-1 背包问题实例：背包的承重量为 6；共有 3 个物品，重量分别是 $w_1 = 2, w_2 = 2, w_3 = 3$，价值分别是 $v_1 = 2, v_2 = 2, v_3 = 3$。

对于这个实例，Iterative-Knapsack 算法的运行过程如表 3.1所示：OPT 表中 $i = 0$ 这一行是为了方便计算而引入的；算法首先将此行的所有单元初始化为 0；然后从 $i = 1$ 行到 $i = 3$ 行逐行计算 OPT 表中单元。比如对于单元 $OPT[2, 3]$，我们有：

$$OPT[2, 3] = \max\{OPT[1, 3], OPT[1, 1] + v_2\} = 2.$$

再如单元 $OPT[3, 6]$，我们有：

$$OPT[3, 6] = \max\{OPT[2, 6], OPT[2, 3] + v_3\} = 5.$$

$OPT[3, 6]$ 就是我们要求的值，即能够装的物品总价值最高为 5；所装的物品可以通过回溯得到：5 是由 $OPT[2, 3] + v_3$ 得出的，而 $OPT[2, 3]$ 是由 $OPT[1, 1] + v_2$ 得出的，因此最优的物品组合是物品 I_3 和 I_2。

表 3.1: Iterative-Knapsack 算法运行时逐行填充 OPT 表格

$w =$	0	1	2	3	4	5	6
$i = 3$							
$i = 2$							
$i = 1$	**0**	**0**	**2**	**2**	**2**	**2**	**2**
$i = 0$	0	0	0	0	0	0	0

计算 $i = 1$ 行

$w =$	0	1	2	3	4	5	6
$i = 3$							
$i = 2$	**0**	**0**	**2**	**2**	**4**	**4**	**4**
$i = 1$	0	0	2	2	2	2	2
$i = 0$	0	0	0	0	0	0	0

计算 $i = 2$ 行

$w =$	0	1	2	3	4	5	6
$i = 3$	**0**	**0**	**2**	**3**	**4**	**5**	**5**
$i = 2$	0	0	2	2	4	4	4
$i = 1$	0	0	2	2	2	2	2
$i = 0$	0	0	0	0	0	0	0

计算 $i = 3$ 行

3.4.4 时间复杂度分析

Iterative-Knapsack 算法的时间复杂度非常容易分析：OPT 表格共有 $(n + 1)(W + 1)$ 个单元；计算每个单元时只需要对 2 个值进行比较，因此时间复杂度是 $O(nW)$。

那么 Iterative-Knapsack 算法是多项式时间算法吗?

从形式上看，$O(nW)$ 是多项式的；不过这是 W 的值的多项式，却是 W 的表示长度的指数函数。我们在第 1 章已说明时间复杂度是表示算法运行时间和问题规模之间的关系：问题规模的精确定义涉及到实例的编码方式，是指在特定的编码方式下实例的编码长度，称为输入长度。当我们采用 2 进制编码方案时，表示 W 需要用

$\log_2 W$ 个比特，而 $O(nW) = O(n2^{\log_2 W})$，因此 Iterative-Knapsack 算法是指数时间复杂度算法，不是多项式时间复杂度算法。事实上，0-1 背包问题是 NP-完全问题（见第 9 章），迄今尚未发现有多项式时间算法。

我们把这种是值的多项式、却是输入长度的指数的复杂度称作*伪多项式时间复杂度* (Pseudo-polynomial time complexity)。在第 11 章，我们还会讲到这个算法，讨论如何将这个具有伪多项式时间复杂度的算法改进成一个近似算法。

3.4.5 一些讨论

递归调用实现 vs. 迭代实现

在上一小节中，我们谈到过对动态规划算法来说，迭代实现是最常用的方式，通常会比递归调用实现更高效；但是这并不是绝对的。对 0-1 背包问题来说，Iterative-Knapsack 算法需要填充 OPT 表格中的所有单元。如果 W 和所有的 w_i 都是 100 的倍数，则计算 $OPT(i,1), OPT(i,2), \cdots, OPT(i,99)$ 都是冗余的；推广到一般情况，对于不是 100 倍数的 w 来说，计算 $OPT(i,w)$ 是冗余的。

对这种实例，我们在递归调用实现中使用 Hash 技术，记录下已经求解过的子问题的解，避免存储整个 OPT 表格，并且只求解那些需要用到的子问题，从而避免了冗余计算。采用递归调用技术的算法描述见*算法* 28。

Algorithm 28 *求解背包问题的递归算法*

function Memorize-Knapsack(i, w)

1: **if** Hash table H has an element with key (i, w) **then**
2: **return** $H(i, w)$;
3: **end if**
4: **if** $i == 0$ **or** $w == 0$ **then**
5: **return** 0;
6: **end if**
7: **if** $w \geq w_i$ **then**
8: $v = \max\{$Memorize-Knapsack$(i-1, w), v_i +$Memorize-Knapsack$(i-1, w-w_i)\}$;
9: **else**
10: $v =$Memorize-Knapsack$(i-1, w)$;
11: **end if**
12: Insert v into Hash table H with key (i, w);
13: **return** $H(i, w)$;

子集上的归约 vs. 序列上的归约

当对集合进行归约时，定义子问题时需要小心：如果子问题定义在任意的子集上，则不可避免地会导致子问题的数目有指数多。

在本例中，我们重新定义了一个多步决策过程：从在多个物品中选择一个，变为*按照预先指定的顺序逐个考虑*。这样一来，子问题就变成在前 i 个物品中寻找最优组合；考虑的不再是任意的子集，而是形如 $\{I_1, I_2, \cdots, I_i\}$ 的子集；这实质上是把子集上的归约转换成序列上的归约，从而将子问题的数目从指数变成多项式。

值得说明的是，我们这里采用的是从后向前，按照 $I_n, I_{n-1}, \cdots, I_1$ 的顺序逐个考虑每个物品；这并不关键，采用从前往后的方式也是可以的。

3.5 RNA 二级结构预测：对字符串的归约

RNA（核糖核酸）是以核糖核苷酸为单元，通过磷酸二酯键链接而成的一条链状大分子；核糖核苷酸上的碱基有 4 种常见类型，分别是 `A`（Adenine，腺嘌呤）、`U`（Uracil，尿嘧啶）、`G`（Guanine，鸟嘌呤）、`C`（Cytosine，胞嘧啶）。

在天然环境中，RNA 不是呈现松散的、无规则状态，而是会自发地折叠（Folding）成规则的三维结构，以使得自由能最小。驱动 RNA 折叠的是碱基之间的非共价相互作用，包括氢键、静电力、范德华力以及相邻碱基对之间的堆叠效应（Stacking）等；比如碱基 `A` 和 `U` 之间会形成 2 个氢键、碱基 `C` 和 `G` 之间会形成 3 个氢键，这两类常见的碱基配对称为 Watson-Crick 配对 [?]。RNA 碱基之间形成的配对称作 RNA 二级结构（见图 3.9）。

为简化问题，我们对 RNA 结构的自由能以及碱基配对做一些假设：

(1) 配对的碱基越多，自由能越小 [?]。

(2) 两个碱基配对之间不能形成*交叉*。在生物学上这种交叉被称为*假结*（Pseudo-knot），示例见图 3.10。换句话说，任意两个碱基配对之间要么是分离的，要么是嵌套的（Nested）。生物学观察表明，形成交叉的碱基配对一般来说数量比较少 [?]，忽略它们会影响 RNA 的三级结构，但是对于二级结构预测来说影响不大。

(3) 一个碱基至多只能和一个碱基配对，且我们只考虑 Watson-Crick 配对，即 `A-U` 配对和 `C-G` 配对，而忽略其他较少出现的碱基配对。

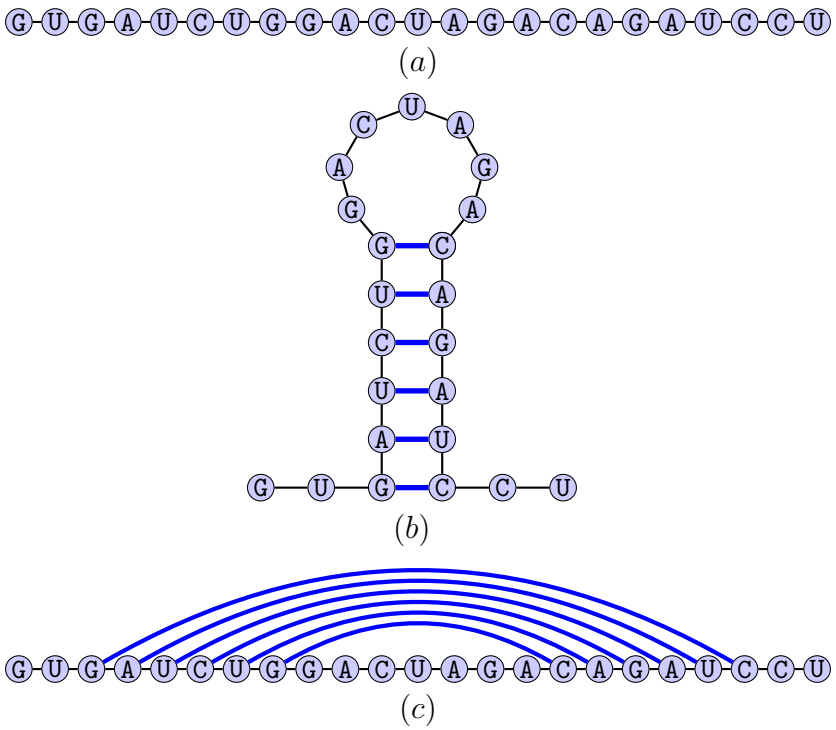


图 3.9: RNA 序列及其折叠成的二级结构示例。(*a*) RNA 序列，其中细线表示相邻碱基间的磷酸二酯键；(*b*) RNA 折叠而成的二级结构，其中粗线表示碱基之间的互补配对；(*c*) RNA 二级结构的另一种表示方式，其中弧线表示碱基配对

(4) 由于核糖核苷酸体积的制约，RNA 无法做剧烈的弯折，导致近邻的碱基之间无法形成配对。详细地说，如果第 i 个碱基和第 j 个碱基形成配对，则 $|i-j|>4$。

从计算机的观点来看，RNA 就是一个由 4 个字母 `A`，`C`，`T`，`G` 组成的字符串。所谓 RNA 二级结构预测，是指对于给定的碱基字符串，计算出最多的无交叉碱基配对。我们把这个问题形式化描述如下：

RNA 二级结构预测问题（RNA secondary structure prediction）
输入：一个字符序列 $R=R_1R_2\cdots R_n$，其中 $R_i\in\{$`A`，`C`，`U`，`G`$\}$；
输出：最多的无交叉碱基配对。此处配对碱基包括 `A-U` 和 `C-G`，其配对碱基的下标之差须大于 4。

3.5.1 算法设计与描述

当 RNA 的碱基少于 5 个时，无法形成碱基配对，因此算法可以直接返回结果："0 个碱基配对"；然而当 RNA 较长时，可能的二级结构很多，难以直接求解，因此我们

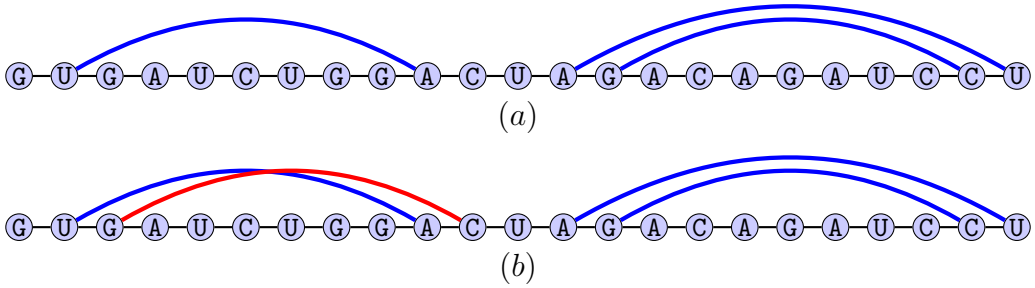


图 3.10: RNA 二级结构中的碱基配对。(*a*) 两个碱基配对之间要么是分离的，要么是嵌套的。(*b*) 左侧的两个碱基配对相互交叉，称作假结（pseudoknot）

来尝试能否将问题分解成小的子问题。

子问题定义

注意到完整解是一系列无交叉的碱基配对；我们将完整解的求解过程描述成从空集开始、逐个添加碱基配对的多步决策过程：

(1) **部分解的状态**：我们用碱基配对的集合来表示部分解。

(2) **扩展部分解的决策**：我们沿着 RNA 序列从后往前，逐个考虑碱基；在每一步，设当前步考虑的碱基是 R_i，我们都在如下两个决策项里做最优选择：*i*) R_i 不和任意碱基匹配；*ii*) R_i 和某一个碱基 $R_j (j < i-4)$ 形成 `A-U` 或者 `C-G` 配对，但不得与已知碱基配对形成交叉。

(3) **决策项的收益**：在每一步决策中，如果形成碱基配对，则收益为 1；否则收益为 0。因此，待优化的目标函数等价于所有步骤决策的收益总和。

因此我们可以如下定义子问题：当已知部分碱基配对时，如何在剩余碱基中进行决策，使得剩余步骤的决策收益总和最大。

以第一步对碱基 R_n 的决策为例（见图 3.11），最优解中有如下两种选择项：

(1) R_n **不与任何碱基形成配对**：则剩下的子问题是如何计算子序列 $R_1R_2\cdots R_{n-1}$ 形成的最多无交叉碱基配对。

(2) R_n **与某个碱基** R_k **形成碱基配对，其中** $k < n-4$：由于任何碱基配对都不能和 $R_n - R_k$ 碱基配对交叉，因此剩下的子问题是：*i*) 如何计算子序列 $R_{k+1}R_{k+2}\cdots R_{n-1}$ 形成的最多无交叉碱基配对；*ii*) 如何计算子序列

$R_1R_2\cdots R_{k-1}$ 形成的最多无交叉碱基配对。值得指出的是：这两个子问题是相互独立的；对一个子问题选择哪些碱基配对，不会影响另一个子问题的选择。

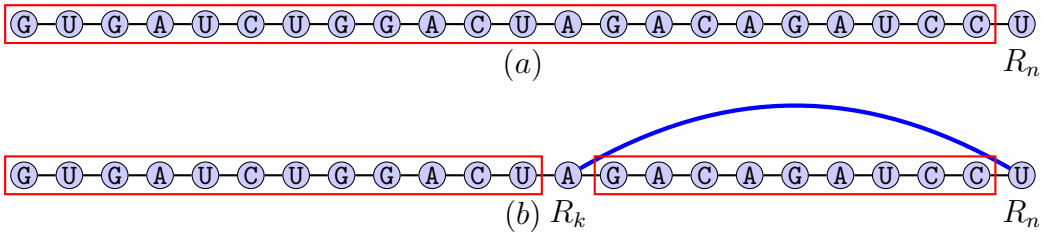


图 3.11: 多步决策过程中第一步决策的两个选择项：(a) 碱基 R_n 不和任何碱基配对；(b) 碱基 R_n 与碱基 R_i 配对

注意到在有的子问题中，子序列的起始下标是可变的，而在另外的子问题中，子序列的终止下标是可变的。归纳这些子问题的不同情形，我们设计子问题的一般形式为：计算子序列 $R_iR_j\cdots R_j$ 中的最多无交叉碱基配对，其总数记为 $OPT(i,j)$。

子问题最优解之间的递归关系

依据上述分析，我们可采用如下的基始步骤和递归关系计算 $OPT(i,j)$：

(1) 当 $|i-j|\leq 4$ 时，$OPT(i,j)=0$;

(2) 当 $|i-j|>4$ 时，我们有如下递归关系：

$$OPT(i,j)=\max\begin{cases} OPT(i,j-1) \\ \max\limits_{\substack{i\leq k<j-4 \\ R_j \text{ pairs with } R_k}} \{1+OPT(i,k-1)+OPT(k+1,j-1)\} \end{cases}$$

采用这种子问题定义，原始给定问题可以表示为计算 $OPT(1,n)$；我们可以采用迭代技术计算 $OPT(1,n)$，算法描述如下：

Algorithm 29 预测 RNA 二级结构的递归算法

function RNA2D(n)

1: Initialize all $OPT[i,j]$ with 0;
2: **for** $i = 1$ to n **do**
3: **for** $j = i+5$ to n **do**
4: $OPT[i,j] = \max\{OPT[i,j-1], \max_{\substack{i \le k < j-4 \\ R_j \text{ pairs with } R_k}} \{1 + OPT[i,k-1] + OPT[k+1,j-1]\}\}$;
5: **end for**
6: **end for**
7: **return** $OPT[1,n]$

3.5.2 算法运行过程示例

表 3.2展示算法 RNA2D 对 RNA 序列"ACCGGUAGU" 的运行过程：算法每一步都计算 OPT 表格中一条对角线上的单元，如此逐步进行，直至最后计算出 $OPT[1,9]$。

表 3.2: RNA2D 算法运行过程示例

	$j=$ 6	7	8	9	6	7	8	9	6	7	8	9	6	7	8	9
$i=4$	0	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0	0
$i=3$	0	0	**1**		0	0	1	**1**	0	0	1	1	0	0	1	1
$i=2$	0	**0**			0	0	**1**		0	0	1	**1**	0	0	1	1
$i=1$	**1**				1	**1**			1	1	**1**		1	1	1	**2**
	第 1 步				第 2 步				第 3 步				第 4 步			

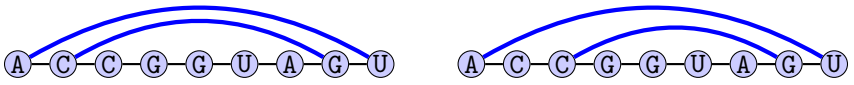


图 3.12: 算法 RNA2D 对 RNA 序列"ACCGGUAGU" 预测出的两种二级结构

以 $OPT[2,8]$ 为例，其计算过程为：

$$OPT[2,8] = \max\{OPT[2,7], 1 + OPT[3,7], 1 + OPT[4,7]\} = 1.$$

再如 $OPT[1,9]$，其计算过程为：

$$OPT[1,9] = \max\{OPT[1,8], 1 + OPT[2,8]\} = 2.$$

在计算出 $OPT[1,9]$ 之后，通过回溯即可求出 RNA 的二级结构。如图 3.12所示，有 2 种二级结构可以达到最多的碱基配对。

3.5.3 时间复杂度分析

算法 RNA2D 的时间复杂度也比较容易分析：OPT 表格中共有 $O(n^2)$ 个单元；计算每个单元最多需要进行 n 次比较，因此时间复杂度是 $O(n^3)$。

值得指出的是，RNA2D 算法虽然简单，但是略加变换即可解决更复杂的问题：

(1) 目标函数不是定义在**单个碱基对**上，而是定义在**相邻碱基对**上：在本小节中，我们采用的假设是“RNA 结构的自由能仅与碱基配对的数目相关”；这种假设非常简单，并未考虑相邻碱基配对之间的堆叠效应等相互作用。然而只需对递归关系适当改动，即可适用于更精确的自由能定义 [?]。

(2) RNA 二级结构的**概率模型及参数训练方法**：当不允许碱基配对之间的交叉时，RNA 二级结构可以用随机上下文无关文法（Stochastic Context-Free Grammar）进行概率建模。这是因为上下文无关文法很容易刻画嵌套结构。采用概率模型，能够更好地刻画碱基对的置信度，并基于训练集训练出合适的参数 [?]。

3.6 隐马尔可夫模型的解码问题：对序列的归约

语音识别、基因预测等很多应用问题都可以建模成隐马尔可夫模型的解码问题（Hidden Markov model decoding）。在此我们以**偶尔作弊的赌场**（Occasionally dishonest casino）为例介绍解码问题及算法，描述如下：一个赌场里有两个骰子，一个是公平的骰子（Fair dice，记为 `F`），其掷出 6 种点数的概率都是 $\frac{1}{6}$；另一个骰子是灌铅的（Loaded dice，记为 `L`），其掷出 5 点和 6 点的概率是 $\frac{3}{10}$，掷出其他点数的概率是 $\frac{1}{10}$（见图 3.13）。赌场有时使用公平的骰子，有时则会偷偷换成灌铅的骰子。我们假设赌场按照如下的方式使用两个骰子：在第一次投掷时，赌场选择公平骰子的概率是 $\frac{3}{5}$，选择灌铅骰子的概率是 $\frac{2}{5}$；在后续投掷时，如果上一次使用的是公平骰子，则赌场下一轮使用灌铅骰子的概率是 0.2；如果上一轮使用的是灌铅骰子，则下一轮使用公平骰子的概率是 0.1。我们考虑如下问题：当观察到 6 次投掷的点数 $1, 3, 4, 5, 5, 6$ 时，能否推测出每一次投掷最可能使用的是哪个骰子呢？

为刻画点数的**产生过程**，我们定义如下两类随机变量：

(1) **隐含状态变量**$X = (x_1, x_2, \cdots, x_n)$：$x_i \in S$ 表示在第 i 个时刻**待观察对象**的状态；这个状态对于观察者是不可见的、**隐含**的。我们用 $S = \{s_1, s_2, \cdots, s_N\}$

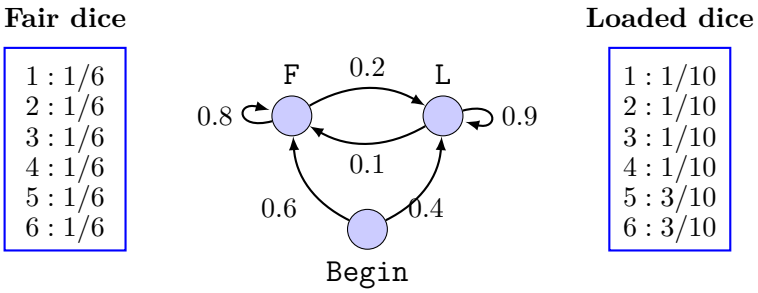


图 3.13: 刻画“偶尔作弊的赌场”中点数产生过程的隐马尔可夫模型

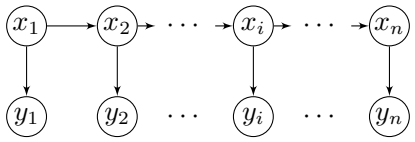


图 3.14: 隐马尔可夫模型中隐含状态变量与观察变量之间的依赖关系

表示状态的所有可能取值。在本例中，观察对象是骰子；$x_i \in \{\texttt{F}, \texttt{L}\}$ 表示第 i 次投掷使用的是哪个骰子；

(2) **观察变量** $Y = (y_1, y_2, \cdots, y_n)$：$y_i \in O$ 表示在第 i 个时刻观察者对研究对象的观察结果。我们用 $O = \{o_1, o_2, \cdots, o_M\}$ 表示一次观察所能获得的所有可能取值。在本例中，$y_i \in \{1, 2, 3, 4, 5, 6\}$ 表示在第 i 次投掷时，观察者看到投掷出的点数。

上述随机变量之间存在着**依赖关系**。我们假设在任一时刻 i，待观察对象的状态 x_i 仅依赖于前一个时刻的状态 x_{i-1}，而与前 $i-1$ 个时刻的隐含变量无关（即 Markov 性质）；观察变量 y_i 的取值仅依赖于隐含状态变量 x_i，而与其它变量的取值无关（见图 3.14）。我们使用如下三组参数刻画上述依赖关系：

(1) **状态转移概率**（Transition probability）：待观察对象可在 N 种状态之间互相转移；我们用条件概率表示相继两个时刻隐含状态变量之间的依赖关系，即：

$$a_{kl} = P(x_i = l | x_{i-1} = k), \quad k, l \in S$$

此处我们假设这些条件概率不随时刻 i 改变；其全体构成一个矩阵，记为 $\mathbf{A} = [a_{kl}]_{N \times N}$。

(2) **发射概率**（Emission probability）：在状态 k 下，观察变量以一定的概率取观察值；或者换句话说，待观察对象以一定的概率“发射”（Emit）出观测值，此

概率记为：

$$b_k(j) = P(y_i = j | x_i = k), \quad k \in S,\ j \in O$$

同样地，我们假设这些条件概率也不随时刻 i 而改变；其全体构成一个矩阵，记为 $\mathbf{B} = [b_k(j)]_{N \times M}$。

(3) *初始状态概率*：待观察对象在第 1 个时刻可处于多种状态，处于状态 k 的概率记为：

$$\pi_k = P(x_1 = k), \quad k \in S$$

以“偶尔作弊的赌场”为例：对于赌场来说，每次投掷使用哪个骰子是清楚的，因此多次投掷时骰子的使用情形构成一个 Markov 链；而对于观察者来说，只能观察到投掷出的点数，每一次具体使用的是哪个骰子是不知道的、是“隐含”的，因此观察到的点数变换过程是一个隐 Markov 过程。

在定义了随机变量以及变量之间的依赖关系之后，我们可以将观察变量的产生过程描述如下：

(1) 在第 1 个时刻，依据初始状态概率分布 π 确定状态 x_1 的取值，并依据发射概率确定观察变量 y_1 的取值；

(2) 在后续时刻 $i = 2, \cdots, n$，依据状态转移概率确定状态 x_i 的取值，并依据发射概率确定观察变量 y_i 的取值。

隐含状态变量和观察变量的联合分布为：

$$P(x_1, y_1, \cdots, x_n, y_n) = \pi_{x_1} \prod_{i=1}^{n-1} \big(b_{x_i}(y_i) a_{x_i x_{i+1}}\big) b_{x_n}(y_n)$$

和观察变量的产生过程方向完全相反，解码问题是从观察变量出发，推测隐含的状态变量概率最大的取值，描述如下：

HMM 解码问题（HMM decoding）

输入： HMM 参数 $\mathbf{A}, \mathbf{B}, \pi$，观察变量序列 $y_1, y_2, \cdots, y_n$；

输出：隐含状态变量 $x_1, x_2, \cdots, x_n$，使得联合概率 $P(x_1, y_1, \cdots, x_n, y_n)$ 最大。

为直观起见，我们把每个时刻的所有状态表示成顶点，并画在一列上；相邻时刻的状态转移用有向边表示。这样构成的有向无环图称为*格栅图*（Trellis [**?**]）。在格栅图中，n 个时刻的状态表示成从 `Beigin` 到 `End` 的一条状态路径，而与一条状态路径

对应的概率 $P(x_1, y_1, \cdots, x_n, y_n)$ 可以表示成路径上所有边和点的权重乘积（示例见图 3.15）。因此解码问题可以视为在图中寻找一条权重乘积最大的路径。我们记联合概率的最大值为：

$$\begin{aligned} OPT(y_1, \cdots, y_n) &= \max_{x_1, \cdots, x_n} P(x_1, y_1, \cdots, x_n, y_n) \\ &= \max_{x_1, \cdots, x_n} \pi_{x_1} \prod_{i=1}^{n-1} (b_{x_i}(y_i) a_{x_i x_{i+1}}) b_{x_n}(y_n) \end{aligned} \tag{3.6.1}$$

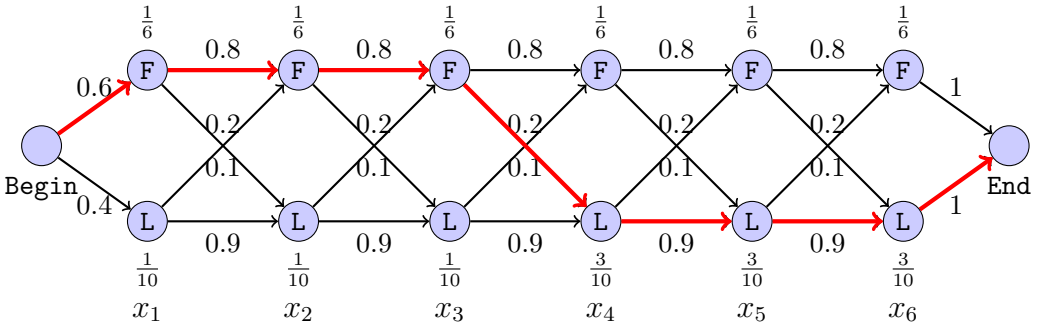


图 3.15: “偶尔作弊的赌场”中的所有状态路径及最优状态路径示例。在此例中，观察变量序列 $\mathrm{y} = (1, 3, 4, 5, 6, 6)$；边上的权重表示状态转移概率；顶点的权重表示“发射”概率。从 Begin 到 End 的任意一条路径都对应着一种可能的隐含状态变量取值；采用 Viterbi 算法计算出的最大可能状态路径以红色标示

3.6.1 算法设计与描述

对于最简单的实例 $n = 1$ 来说，我们只需要比较 $\pi_{\mathrm{F}} b_{\mathrm{F}}(y_1)$ 和 $\pi_{\mathrm{L}} b_{\mathrm{L}}(y_1)$ 的大小即可确定最可能的隐含状态；然而当 n 比较大时，n 个时刻的状态总数 2^n 很大，使得简单枚举策略不可行。因此，我们来尝试能否将复杂问题分解成简单问题，经过逐级分解，最终分解成能够求解的问题。对这个思路而言，关键是定义合适的子问题。

依照原始问题的形式直接定义子问题：一个不成功的尝试

动态规划算法的核心是最优子结构性质，描述的是*子问题之间的递归关系*。在迄今为止的所有例子中，我们都是依照原始问题的形式直接定义子问题，即：从原始问题的优化目标函数中“截取”出一部分作为子问题的定义，这样定义出的子问题依然具有递归性。

对于“偶尔作弊的赌场”这个问题，我们也先尝试一下这种定义方法：原始问题的

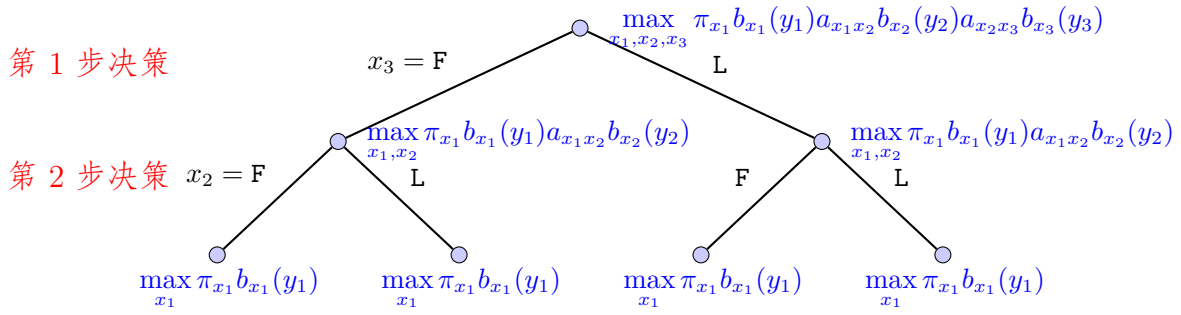


图 3.16: 一次不成功的尝试：依照“偶尔作弊的赌场”的原始问题形式直接定义子问题。此例中包含 3 次投掷结果 y_1, y_2, y_3 和 3 个隐含状态 x_1, x_2, x_3。多步决策的第一步确定 x_3 的取值，第二步决策时确定 x_2 的取值。然而这样直接定义的子问题并不具有递归性

目标是 $\max_{x_1,\cdots,x_n} \pi_{x_1} \prod_{i=1}^{n-1} (b_{x_i}(y_i) a_{x_i x_{i+1}}) b_{x_n}(y_n)$，具有连乘的形式；我们从中截取一部分 $\max_{x_1,\cdots,x_l} \pi_{x_1} \prod_{i=1}^{n-1} (b_{x_i}(y_i) a_{x_i x_{i+1}}) b_{x_l}(y_l)$ 作为子问题的一般定义。图 3.16展示了一个含有 3 个观察变量、3 个隐含状态的例子，说明采用这种方法定义的子问题以及相应的多步决策过程。

然而仔细的分析表明，按照原始问题的形式直接定义的**子问题不具有递归性**，即：

$$\max_{x_1,x_2,x_3} \pi_{x_1} b_{x_1}(y_1) a_{x_1x_2} b_{x_2}(y_2) a_{x_2x_3} b_{x_3}(y_3) \neq \max_{x_3} \Big\{ \big(\max_{x_1,x_2} \pi_{x_1} b_{x_1}(y_1) a_{x_1x_2} b_{x_2}(y_2) \big) a_{x_2x_3} b_{x_3}(y_3) \Big\}.$$

究其原因，在于两个子问题之间的“夹缝”里有一个项 $a_{x_2x_3}$，它同时关联了两个变量 x_2, x_3，然而最外层的 $\max_{x_3}$ 只对一个变量 x_3 取最大。因此，即便求解了子问题 $\max_{x_1,x_2} \pi_{x_1} b_{x_1}(y_1) a_{x_1x_2} b_{x_2}(y_2)$，求出的最优解 x_1, x_2 不一定会使得 $\pi_{x_1} b_{x_1}(y_1) a_{x_1x_2} b_{x_2}(y_2) a_{x_2x_3} b_{x_3}(y_3)$ 最大。

表 3.3展示了一个很有说服力的例子：当考虑 3 次投掷的点数 $(y_1, y_2, y_3) = (1, 5, 6)$ 时，用枚举法可以得知最可能状态序列是 (L, L, L)；而当只考虑前 2 次投掷的点数 $(y_1, y_2) = (1, 5)$ 时，最可能状态序列是 (F, F)；然而 (F, F) 并不是 (L, L, L) 的前缀，这表明我们无法基于子问题的最优解构造出原问题的最优解。

构造具有递归性的子问题：添加对决策变量的限制

依照原始问题的形式直接定义的子问题之所以不具有递归性，根源在于目标函数中有 $a_{x_i x_{i+1}}$ 这样的关联项，关联了两个变量 x_i 和 x_{i+1}。搞清了原因，解决方案也就很清楚了：**在子问题中添加一个限制条件，限制决策变量 x_{i+1} 的取值是事先给定的常数，从而切断两个变量之间的关联性。**我们可以证明：添加了限制条件之后的子问

表 3.3: 对 3 次投掷的点数 $(y_1, y_2, y_3) = (1, 5, 6)$ 及其前 2 次投掷的点数 $(y_1, y_2) = (1, 5)$ 计算出的最可能状态序列，分别为“LLL”和“FF”；“FF”不是“LLL”的前缀，意味着不存在递归关系

x_1	x_2	x_3	$P(x_1, x_2, x_3, y_1, y_2, y_3)$
F	F	F	0.00177
F	F	L	0.00079
F	L	F	0.00009
F	L	L	0.00162
L	F	F	0.00008
L	F	L	0.00004
L	L	F	0.00018
L	L	L	0.00292

x_1	x_2	$P(x_1, x_2, y_1, y_2)$
F	F	0.01333
F	L	0.00599
L	F	0.00066
L	L	0.01080

题具有递归性。

如图 3.17所示，问题的完整解是所有隐含状态变量 $x_1, x_2, \cdots, x_n$ 的取值，我们将求解过程描述成“按照 $x_n, x_{n-1}, \cdots, x_1$ 的顺序、从后往前逐个确定隐含状态变量取值”的多步决策过程：

(1) 部分解的状态：在第 i 步，我们已知部分解 $x_n, x_{n-1}, \cdots, x_{i+1}$ 的取值。

(2) 扩展部分解的决策：在第 i 步，我们确定隐含状态变量 x_i 的取值，为此需要在 2 个决策项 x_i =F 和 x_i =L 中做出选择。

(3) 决策项的收益：在一般情况下，对于隐含状态变量 x_i 的 2 种可能取值，我们使用相同的方式 $b_{x_i}(y_i)a_{x_i x_{i+1}}$ 计算其收益；对“一头一尾”两个特殊情形需要单

独处理：i) 当 $i=n$ 时，收益为 $b_{x_i}(y_i)$；ii) 当 $i=1$ 时，收益为 $\pi_{x_1}b_{x_1}(y_1)a_{x_1x_2}$。

采用上述收益函数，原始问题的优化目标函数就是所有决策步骤的收益乘积。

以对 x_n 的决策为例，我们有如下决策项：

(1) x_n 取值 F：剩下的子问题是“确定隐含状态变量 $x_1,\cdots x_{n-1}$ 的取值，使得当 $x_n=\mathtt{F}$ 时，$P(x_1,y_1,\cdots,x_n,y_n)$ 最大”；

(2) x_n 取值 L：剩下的子问题是“确定隐含状态变量 $x_1,\cdots x_{n-1}$ 的取值，使得当 $x_n=\mathtt{L}$ 时，$P(x_1,y_1,\cdots,x_n,y_n)$ 最大”。

归纳这两个子问题，我们将子问题的一般形式定义为：**确定隐含状态** $x_1,\cdots,x_{i-1}$，**使得当** $x_i=k$ **时，** $P(x_1,y_1,\cdots,x_i,y_i)$ **最大**，此处 $k\in\{\mathtt{F},\mathtt{L}\}$ 表示一个状态取值。我们将此最大值记为 $V_i(k)$，即：

$$V_i(k)=\max_{x_1,\cdots,x_{i-1}} P(x_1,y_1,\cdots,k,y_i).$$

和 $\max\limits_{x_1,\cdots,x_{i-1},x_i} P(x_1,y_1,\cdots,x_i,y_i)$ 相比，$V_i(k)$ 加了一个限制条件 $x_i=k$，因而是一个更“细”的子问题。

采用这种子问题定义，原始问题的最优解可以如下计算：

$$OPT(y_1,\cdots,y_n)=\max\{V_n(\mathtt{F}),V_n(\mathtt{L})\}.$$

子问题最优解之间的递归关系

直观地说，$V_i(k)$ 对应于结束于状态 $x_i=k$ 的概率最大状态路径（图 3.15）；假如这条路径经过 $x_{i-1}=l$ 顶点，则其前 $i-1$ 个顶点必定构成结束于 $x_{i-1}=l$ 的概率最大状态路径。换句话说，结束于 $x_i=k$ 状态的概率最大状态路径存在着最优子结构。因此，我们有如下递归关系式及基始情形赋值：

$$V_i(k)=\begin{cases}\pi_k b_k(y_1) & \text{如果}i=1\\ \max\limits_{l\in\{\mathtt{F},\ \mathtt{L}\}} V_{i-1}(l)a_{lk}b_k(y_i) & \text{否则}\end{cases}$$

和上一小节对照来看，上一小节定义的子问题是计算 $OPT(y_1,\cdots,y_i)$，$OPT(y_1,\cdots,y_i)$ 是在 $V_i(\mathtt{F})$ 和 $V_i(\mathtt{L})$ 中取最大值；虽然 $V_i(k)$ 有递归表达式，但是“取最大”这个操作使得我们无法递归地计算 $OPT(y_1,\cdots,y_i)$。

我们可以采用迭代技术来计算 $V_i(k)$；相应的算法称为 Viterbi 算法 [**?**]，伪代码描述见*算法* 30。

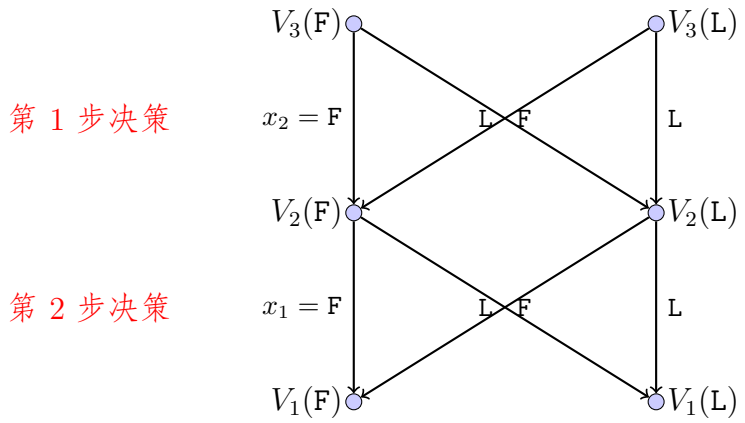


图 3.17: 在 HMM 解码问题中采用更“细”的子问题定义 $V_i(k) = \max_{x_1,\cdots,x_{i-1}} P(x_1, y_1, \cdots, k, y_i)$。子问题具有递归性：我们可以基于 $V_1(\mathtt{F}), V_1(\mathtt{L})$ 计算出 $V_1(\mathtt{F}), V_1(\mathtt{L})$，进而基于 $V_2(\mathtt{F}), V_2(\mathtt{L})$ 计算出 $V_3(\mathtt{F}), V_3(\mathtt{L})$

3.6.2 算法运行过程示例

表 3.4: Viterbi 算法的运行过程示例

i	y_i	$V_i(\mathtt{F})$	$ptr_i(\mathtt{F})$	$V_i(\mathtt{L})$	$ptr_i(\mathtt{L})$
1	1	1.000×10^{-1}	-	4.000×10^{-2}	-
2	3	1.333×10^{-2}	F	3.600×10^{-3}	L
3	4	1.778×10^{-3}	F	3.240×10^{-4}	L
4	5	3.370×10^{-4}	F	1.067×10^{-4}	F
5	5	3.161×10^{-4}	F	2.880×10^{-5}	L
6	6	4.214×10^{-6}	F	7.776×10^{-6}	L

表 3.4展示 Viterbi 算法对观察变量序列 $\mathrm{y} = (1, 3, 4, 5, 5, 6)$ 的运算过程：计算出的联合概率最大值是 $\max\{V_6(\mathtt{F}), V_6(\mathtt{L})\} = 7.776 \times 10^{-6}$，且因为 $V_6(\mathtt{L}) > V_6(\mathtt{F})$，所以状态变量 x_6 最大可能取值是 L。我们从 $x_6\mathtt{L}$ 开始回溯（见图 3.15），最终可求得最大可能状态路径为 $\mathrm{x} = (\mathtt{F}, \mathtt{F}, \mathtt{F}, \mathtt{L}, \mathtt{L}, \mathtt{L})$。

Algorithm 30 求解 HMM 解码问题的迭代算法

function VITERBI(y)

Set $n = |y|$;
Initialize $V_1(k) = \pi_k b_k(y_1)$ for each state $k \in S$;
for $i = 2$ to n **do**
  for each state k **do**
    $V_i(k) = b_k(y_i) \max_{l \in S} \left(V_{i-1}(l) a_{lk}\right)$;
    $ptr_i(k) = \operatorname{argmax}_{l \in S} \left(V_{i-1}(l) a_{lk}\right)$;
  end for
end for
return $\max_{k \in S} V_n(k)$;

function VITERBI-BACKTRACK(V, ptr, n)

$x_n = \operatorname{argmax}_{k \in S} V_n(k)$;
for $i = n - 1$ to 1 **do**
  $x_i = ptr_{i-1}(x_{i+1})$;
end for
return x;

3.6.3 时间复杂度分析

令 $N = |S|$ 表示状态可能取值的数目（对于“偶尔作弊的赌场”这个问题来说，$S = \{\texttt{F}, \texttt{L}\}$，故 $N = 2$）。注意到 VITERBI 算法伪代码的第 5 行共执行 nN 次，而每次执行的 $\max_{l \in S}$ 需考虑 N 个子问题的值，因此 VITERBI 算法的时间复杂度是 $O(nN^2)$。

VITERBI 算法只需多项式的时间即可在指数多条状态路径中找出最优路径，其原因在于并未一一检查所有的路径。例如状态路径 (F, F, F, F, L, L) 的联合概率就没有计算；这是因为即使只考察前 5 个状态时，(F, F, F, F, L) 的联合概率就比 (F, F, F, L, L) 的联合概率低，因此当计算完整状态路径的联合概率时，无需考虑状态路径 (F, F, F, F, L, L)。

3.7 字符串之间的编辑距离：对一对字符串的归约

使用过 TextEdit 或 Microsoft Office 等工具的用户，应该对“单词拼写改错”功能不会陌生。比如我们输入字符串"OCURRANCE" 时，单词拼写工具会报告“发现拼写错误”；更重要的是，拼写工具往往还会猜测出我们的真实意图是想输入单词"OCCURRENCE"，

从而进行自动改正。

单纯的发现拼写错误并不困难：我们只需将输入字符串和词典中的每一个单词进行比较；如果在词典中找不到输入字符串，则报告“发现拼写错误”。相比之下，改正拼写错误就有些困难了：我们需要从字典里找出和输入字符串最相似的单词，并推测这个单词就是输入者的真实意图。为此目的，我们需要定量地衡量两个字符串之间的相似度，或者等价地，字符串之间的差异。编辑距离是字符串之间差异的定量度量之一。

衡量两个字符串之间的差异：编辑距离

度量两个字符串之间差异大小的方式之一是共有 k-mer 的数目，或者更进一步，将字符串转换成由 k-mer 出现次数构成的向量，然后计算两个向量之间的夹角。上述方式在衡量两个微生物基因组之间差异时取得了很大的成功 [?]，但是却无法表示出如何从一个字符串变成另一个字符串。

下面我们来介绍能达到这个目标的另一种字符串差异度量方式：*编辑距离*（Edit distance），也称 Levenshtein 距离 [?]。

编辑距离的*直观思想*是：我们把拼写过程描述为从意图输入的单词（记为 x）到实际拼写出的字符串（记为 y）的变换过程，其中包括多步的*编辑操作*（插入、删除，和替换一个字符）；而能够将 x 变换成 y 的*最少编辑操作数目*称为 x 和 y 之间的编辑距离。以 x = `"OCCURRENCE"` 和 y = `"OCURRANCE"` 为例，只需执行 2 步编辑操作（删除 x 中的第 2 个字符`'C'`，并将第 7 个字符`'E'` 替换成`'A'`）即可将 x 变换成 y，因此它们之间的编辑距离是 2。

详细地说，从 x 写出 y 的过程中，可能的编辑操作有如下 4 种 [?]：

(1) 意图输入 x 中一个字符时“写对了”（称作相对于这个字符的“匹配”操作，Match），例如 y 中第 1 个字符`'O'`，是意图写出 x 中第 1 个字符`'O'` 时“写对了”；

(2) 意图输入 x 中一个字符时“写错了”（称作相对于这个字符的“替换”操作，Substitution），例如 y 中第 7 个字符`'A'`，是意图输入 x 中字符`'E'` 时“写错了”；

(3) 意图输入 x 中一个字符时却“遗漏了”（称作相对于 x 的“删除”操作，Deletion），例如 x 中有两个字符`"CC"`，而 y 中却只有一个字符`'C'`，遗漏了一个字符`'C'`；

(4) “额外多输入了”一个 x 中并不存在的字符（称作相对于 x 的“插入”操作，Insertion）。这是与遗漏字符对称的情况，具体示例在后面展示。

字符串编辑距离的计算：全局联配

为便于计算 x 和 y 之间的编辑距离，我们采用*联配*（Alignment，也称为“对齐”）的方式直观地表示它们之间的异同，即：把 x 和 y 各写在一行、让 y 中的每个字符与其在 x 中的来源字符放在同一列上。

如果拼写 y 时仅有“写对了”和“写错了”这两种操作，则直接写出 x 和 y 就能保证 y 中字符与其来源字符在同一列。然而如果在拼写时发生了“插入”或者“删除”，直接写出 x 和 y 会发生“错位”现象；在这种情况下，为保证 y 中的每个字符都与其来源字符在同一列，我们引入一个辅助的*占位字符*，记作`'-'`，其添加规则如下：

(1) 如果在拼写 y 时遗漏了一个字符，我们就在 y 中相应位置上添加一个占位字符`'-'`，并把扩展之后的 y 称作 y′；

(2) 与之完全对称地，我们也可以在 x 中添加占位字符`'-'`，以表示在拼写 y 时额外多输入了一个字符，并把扩展之后的 x 称作 x′。

在添加占位字符之后，y′ 中的每个字符都与 x′ 中的一个字符“对齐”了，因此二者具有相同的长度。例如：

x′: `OCCURRENCE`
y′: `O-CURRANCE`

此处 y′ 中的字符`'-'` 表示 x′ 这个位置上的字符`'C'` 被遗漏了，而“写对了”的那些字符，比如字符`'U'`，处于 x,y 的同一列。

我们把这种向字符串 x 和 y 中插入占位符、扩展成等长字符串 x′ 和 y′ 的操作称为*字符串全局联配*，记为 (x′,y′)。在扩展时对占位符的约束是：

(1) *占位符的数目*：在向 x′ 与 y′ 添加占位符之后，得到的 x′ 与 y′ 必须等长。

(2) *占位符的位置*：对于任意一个位置 k ($1 \le k \le \|\mathrm{x}'\|$)，对齐的两个字符 x'_k 和 y'_k 不能同时是占位符（占位符用来表示“漏抄的”或“额外多输入的”的字符，因此 x′ 中的占位符和 y′ 中的占位符“对齐”是无意义的）。

采用联配这种表示方式，我们可以很方便地计算出*编辑操作数目*：一个联配 (x′,y′) 表示“从 x 到 y 的一种可能的变换过程”，而此变换过程中的编辑操作数目

可以表示成如下的线性加和函数

$$S(\mathrm{x}', \mathrm{y}') = \sum_{k=1}^{\|\mathrm{x}'\|} s(x'_k, y'_k),$$

其中函数 $s(\cdot, \cdot)$ 表示两个字符之间的编辑操作数目：

$$s(x, y) = \begin{cases} 1 & \text{如果} x = \texttt{'-'} \text{ 或} y = \texttt{'-'} \\ 0 & \text{如果} x = y \\ 1 & \text{如果} x \neq y \end{cases}$$

以联配 x′ ="OCCURRENCE", y′ ="O-CURRANCE" 为例，其编辑操作数目为：

$$S(\mathrm{x}', \mathrm{y}') = 0 + 1 + 0 + 0 + 0 + 0 + 1 + 0 + 0 + 0 = 2.$$

再以第二种联配 x′ ="OCCURRE-NCE", y′ ="O-CURR-ANCE" 为例，其编辑操作数目为：

$$S(\mathrm{x}', \mathrm{y}') = 0 + 1 + 0 + 0 + 0 + 0 + 1 + 1 + 0 + 0 + 0 = 3.$$

由于第一个联配比第二个联配的编辑数目更少，我们有理由说第一个联配更可能是从 x 到 y 的真实变换过程。

采用联配这个概念，我们可以将字符串编辑距离计算问题形式化描述如下：

字符串编辑距离计算问题（Edit distance of sequences）

输入：两个字符串 $\mathrm{x} = x_1 x_2 \cdots x_m$ 和 $\mathrm{y} = y_1 y_2 \cdots y_n$；

输出：计算联配 $(\mathrm{x}', \mathrm{y}')$，使得编辑操作数目 $S(\mathrm{x}', \mathrm{y}')$ 最小。其中 x′ 和 y′ 是由 x 和 y 各自添加若干占位符'-' 扩展而成的，且 $\|\mathrm{x}'\| = \|\mathrm{y}'\|$。

我们容易证明最小的编辑操作数目满足三角不等式，甚至在进一步扩展之后，比如对每类编辑操作赋予一个权重 [?]，以及对连续占位符赋予更复杂的权重 [?]，依然满足三角不等式；因此，我们称之为 x 和 y 之间的**编辑距离**，记为 $d(\mathrm{x}, \mathrm{y})$。

3.7.1 算法设计与描述

我们从最简单的实例入手：如果 y 是空字符串（即 $n = 0$），则意味着在拼写 x 时，所有字符都被“遗漏了”；因此 x 和 y 之间只存在一种联配方案：x′ = "$x_1 x_2 ... x_m$", y′ ="--...-"，编辑距离为 m。类似地，当 x 是空字符串时，也是只存在一种联配方案，编辑距离为 n。

对于 m 和 n 都比较大的实例来说，可能的联配总数 $\sum_{k=0}^{n} \binom{m+n}{k}$ [?] 很大；虽然枚举策略可以找到最优联配，但是效率非常低，是不可行的。

下面我们尝试设计动态规划算法，用子问题的最优解组合出原始问题的最优解，从而避免了对所有联配的枚举。

子问题定义

我们先回顾一下完整解的形式和含义：完整解是联配 $(\mathrm{x}', \mathrm{y}')$，由字符 $x_1, x_2, ..., x_m$ 和 $y_1, y_2, ..., y_n$ 以及一些插入的占位符构成的；联配的目的是表示 y 中每个字符是怎样从 x 中一个字符（或占位符）产生的。因此，我们可以把问题的求解过程描述下面的*多步决策过程*：初始时 x', y' 都是空字符串，然后逐步扩展；在每一步扩展时，我们向 x', y' 中添加 x, y 中的字符或者占位符，以表示 y 中一个字符的产生过程。

然而 y 中共有 n 个字符，按照哪个顺序考虑这些字符呢?

我们有三种顺序可供选择：

(1) *从后往前*：第一步决策时考虑 y 最后一个字符 y_n 是如何产生的；

(2) *从前往后*：第一步决策时考虑 x, y 第一个字符 y_1 是如何产生的；

(3) *从中间开始*：第一步决策时考虑 y 最中间的字符 $y_{\lceil \frac{n}{2} \rceil}$ 是如何产生的。

事实上，*这三种顺序都是可行的；每一种顺序对应一种多步决策过程，进而对应一种动态规划算法*。我们在下一章里讲与后两种决策过程对应的动态规划算法，这里只讨论基于第一种多步决策过程的动态规划算法，描述如下：

(1) *部分解的状态*：我们在初始时设置 x' 和 y' 为空字符串；从后往前、逐个考虑 y 中的字符，依据这个字符的产生过程扩展 x' 和 y'。因此，部分解是由 x 和 y 的后缀，比如 $x_{i+1}x_{i+2}\cdots x_m$ 和 $y_{j+1}y_{j+2}\cdots y_n$，构成的联配。

(2) *扩展部分解的决策*：在每一步，我们考虑字符 y_j 的产生过程，共有 3 种可能：*i*) y_j 由 x_i 产生，因此向 x' 中添加字符 x_i，向 y' 中添加字符 y_j；*ii*) y_j 是“额外多写出来的”，因此向 x' 前部添加占位符`'-'`，向 y' 中添加字符 y_j；*iii*) y_j 是由 $x_1...x_{i-1}$ 产生的，也就是说 x_i 被“遗漏了”，因此向 x' 中添加字符 x_i，向 y' 前部添加占位符`'-'`。

(3) *决策项的收益*：设在扩展联配时新添加的字符分别为 x' 和 y'，则定义收益为 $s(x', y')$。采用这种收益定义，原始问题的优化目标函数等价于所有步骤决策的收益总和。

以第一步考虑 y 的最后一个字符 y_n 的产生过程为例，共有 3 种可能的决策项（见图 3.18）：

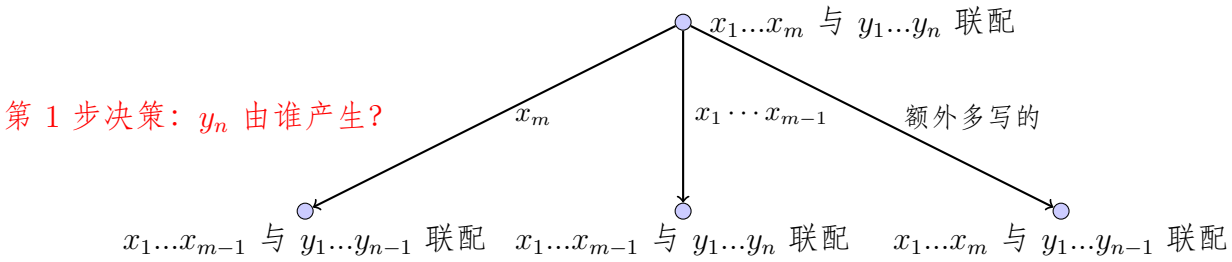


图 3.18: 计算两个字符串 x,y 最优联配的多步决策过程（仅示出第一步决策）。在第一步决策时，我们考虑 y 的最后字符 y_n 是怎样产生的，共有 3 种可能：y_n 是源自 x_m、y_n 由 $x_1x_2\cdots x_{m-1}$ 产生（即：x_m 被遗漏了），以及 y_n 是“额外多写出来的”，

(1) y_n 源自 x_m：因此我们向 x′ 中添加字符 x_m，向 y′ 中添加字符 y_n；如果字符 $x_m \neq y_n$，则表示一个“替换”操作，对最终形成的联配贡献 1 分，否则贡献 0 分；剩下的子问题是计算 x 的前缀 $x_1x_2\cdots x_{m-1}$ 与 y 的前缀 $y_1y_2\cdots y_{n-1}$ 的最优联配（图 3.19）；

(2) y_n 源自 $x_1x_2\cdots x_{m-1}$（即：x_m 在输入过程中被“遗漏了”）：因此我们向 x′ 中添加字符 x_m，向 y′ 前部添加占位符'-'，对最终形成的联配贡献 $s(x_m,\text{'-'})=1$ 分；剩下的子问题是计算 x 的前缀 $x_1x_2\cdots x_{m-1}$ 与 y 的全体 $y_1y_2\cdots y_n$ 的最优联配；

(3) y_n 是“额外多输入的”：因此我们向 x′ 前部添加占位符'-'，向 y′ 中添加字符 y_n，对最终形成的联配贡献 $s(\text{'-'},y_n)=1$ 分；剩下的子问题是计算 x 的全体 $x_1x_2\cdots x_m$ 与 y 的前缀 $y_1y_2\cdots y_{n-1}$ 的最优联配。

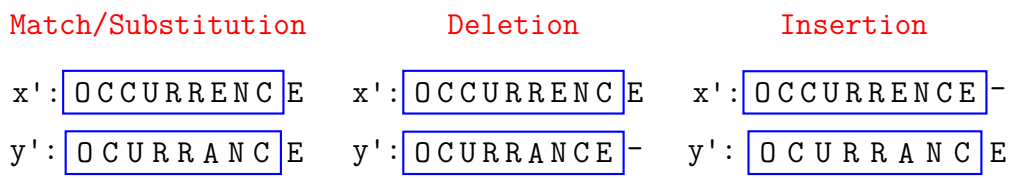


图 3.19: 字符串 x 和 y 联配的 3 种情形

通过归纳这 3 个具体的子问题，我们可以总结出子问题的一般形式：计算 x 的前缀 $x_1x_2\cdots x_i$ 与 y 的前缀 $y_1y_2\cdots x_j$ 的最优联配，其打分记为 $OPT(i,j)$。

子问题最优解之间的递归关系

上述分析表明原始问题的最优解可被拆分成单步决策的决策项、子问题的最优解两个部分；因此，$OPT(i,j)$ 可采用如下递归关系式计算：

$$OPT(i,j) = \min \begin{cases} OPT(i-1,j-1) + s(x_i, y_j) \\ OPT(i-1,j) + s(x_i, \texttt{'-'}) \\ OPT(i,j-1) + s(\texttt{'-'}, y_j) \end{cases}$$

如此递归，直至基始情形，即两个字符串中有一个是空字符串的情形。依据上述的分析，基始情形下 $OPT(i,j)$ 应如下计算：

(1) 如果 $i = 0$，则 $OPT(i,j) = j$；

(2) 如果 $j = 0$，则 $OPT(i,j) = i$。

Algorithm 31 求解两序列编辑距离的迭代算法

function EDIT-DISTANCE(x, y)

1: Set $m = |\text{x}|$ and $n = |\text{y}|$;
2: Set $OPT[i,0] = i$ for all $i = 0$ to m, and set $OPT[0,j] = j$ for all $j = 0$ to n;
3: **for** $j = 1$ to n **do**
4: **for** $i = 1$ to m **do**
5: $OPT[i,j] = \min\{OPT[i-1,j-1] + s(x_i, y_j), OPT[i-1,j] + s(x_i, \texttt{'-'}), OPT[i,j-1] + s(\texttt{'-'}, y_j)\}$;
6: $OPTIndex[i,j] = \text{argmin}\{OPT[i-1,j-1] + s(x_i, y_j), OPT[i-1,j] + s(x_i, \texttt{'-'}), OPT[i,j-1] + s(\texttt{'-'}, y_j)\}$;
7: **end for**
8: **end for**
9: **return** $OPT[m,n]$;

采用这种子问题定义，$OPT(m,n)$ 即为原始给定问题要求解的 x 和 y 的编辑距离 $d(\text{x},\text{y})$。我们可以采用迭代技术计算 $OPT(m,n)$ [**?**]：在计算单元 $OPT(i,j)$ 时，需要已知左上方单元 $OPT(i-1,j-1)$、左侧单元 $OPT(i,j-1)$ 以及上方单元 $OPT(i-1,j)$ 的值；按照“被依赖单元先计算”原则，我们“从左到右逐列计算、每一列从上到下逐个单元计算”，方能满足单元之间的依赖关系（在下一章我们会看到反过来的“从右到左逐列计算、每一列自下而上逐个单元计算”，这是因为依赖关系反过来了）。算法的伪代码描述见算法 31。

3.7.2 算法运行过程示例

图 3.20(a) 展示 EDIT-DISTANCE 算法计算字符串 x = "OCCURRENCE" 和字符串 y = "OCURRANCE" 编辑距离的运算过程：首先，算法初始化 OPT 表格的 $i = 0$ 行和 $j = 0$ 列，计算规则是 $OPT(0, j) = j$, $OPT(i, 0) = i$；然后从 $j = 1$ 列开始逐列计算；在每一列，都是从上往下逐个计算 OPT 表格中的单元。

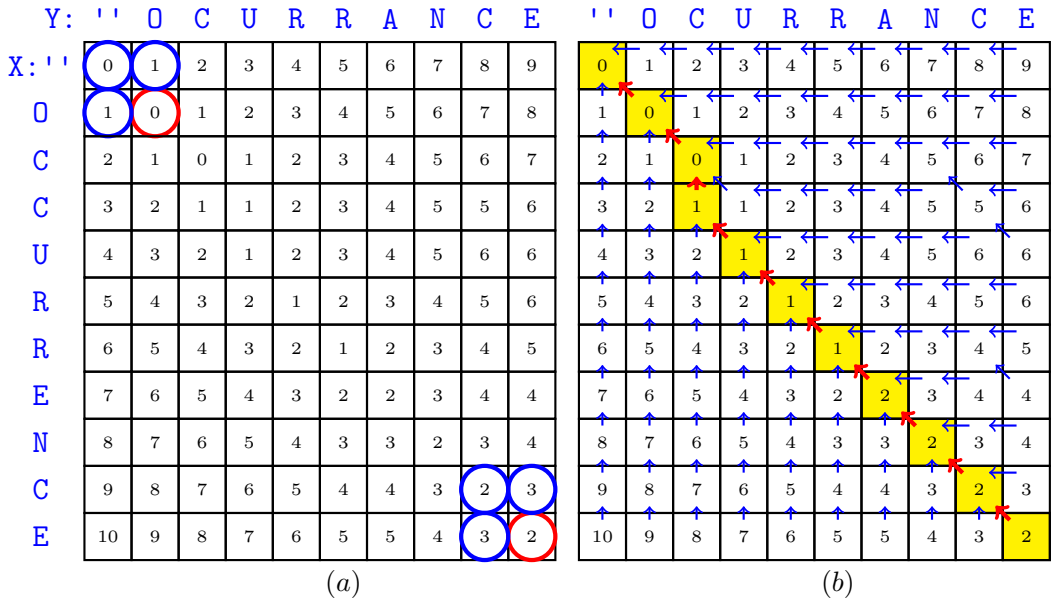


图 3.20: EDIT-DISTANCE 算法计算字符串 x = "OCCURRENCE" 和 y = "OCURRANCE" 编辑距离的运行过程。(a) OPT 表格填充过程；(b) 依据 $OPTIndex$ 表格从 (m, n) 单元开始回溯，回溯产生的路径表示最优联配。这里，我们用箭头直观表示 $OPTIndex$ 表格中的元素

以 $OPT(1, 1)$ 为例，此单元的值依赖于其左上角、上方、左侧 3 个单元的值，而这三个单元在初始化阶段已完成赋值，从而可以得出：

$$OPT(1,1) = \min \begin{cases} OPT(0,0) & (=0) \\ OPT(0,1)+1 & (=2) \\ OPT(1,0)+1 & (=2) \end{cases} = 0$$

如此逐列计算，直至 $j = n$ 列，从而获得原始问题的最优联配的打分：

$$OPT(10,9) = \min \begin{cases} OPT(9,8) & (=2) \\ OPT(9,9)+1 & (=4) \\ OPT(10,8)+1 & (=4) \end{cases} = 2$$

如果想继续获得最优联配的话，我们需新增一个表格（称作 $OPTIndex$），记

录 OPT 表格的每个单元来源于其左上、上方、左侧 3 个单元中的哪一个，比如 $OPT(10,9)$ 的值来源于左上方单元 $OPT(9,9)$。基于 $OPTIndex$ 表格中记录的来源信息，我们从 $OPT(10,9)$ 回溯到 $OPT(9,9)$，并依次类推，直至回溯到 $OPT(0,0)$ 单元（图 3.20）。在回溯的同时，我们从后向前逐步扩增出 x′ 和 y′，规则如下：

(1) 如果 $OPT(i,j)$ 来源于 $OPT(i-1,j-1)$，则向 x′ 扩增 x_i，向 y′ 扩增 y_j；

(2) 如果 $OPT(i,j)$ 来源于 $OPT(i-1,j)$，则向 x′ 扩增'-'，向 y′ 扩增 y_j；

(3) 如果 $OPT(i,j)$ 来源于 $OPT(i,j-1)$，则向 x′ 扩增 x_i，向 y′ 扩增'-'。

按照这个规则，图 3.20(b) 中红色箭头所示路径对应的最优联配为：

x′: OCCURRENCE
y′: O-CURRANCE

3.7.3 时间复杂度分析

Edit-Distance 算法的时间复杂度分析如下：OPT 表格中共有 mn 个单元；计算每个单元时只需要做 3 次简单加法和比较，因此时间复杂度和空间复杂度都是 $O(mn)$。

3.7.4 一些讨论

编辑距离与相似度

在编辑距离算法中，我们是在 3 种情况中取最小值；如果换成取最大值的话，相应的算法称为 Needleman-Wunsch 算法 [?]，最终计算出的是两个字符串之间的相似度。相似度计算问题也被称为字符串全局联配问题（Global sequence alignment）。

自然地，相似度的具体含义依赖于函数 $s(\cdot,\cdot)$ 的定义。这个函数表示“意图写出字符 x，结果写成了字符 y 的可能性”，因此常被称为字符替换打分函数。比如我们采用如下定义：

$$s(x,y)=\begin{cases}0 & \text{如果}x=\texttt{'-'}\ \text{或}y=\texttt{'-'}\\ 1 & \text{如果}x=y\\ 0 & \text{如果}x\neq y\end{cases}$$

则字符串 x 和 y 之间的相似度就是这两个字符串**最长公共子序列**（Longest common subsequence）的长度 [?]。

在生物信息学中，Needleman-Wunsch 算法被广泛用于计算两个 DNA 序列或两个蛋白质序列之间的相似度，其中字符替换打分函数 $s(\cdot,\cdot)$ 的定义至关重要。目前行之有效的方案是从大量已知联配数据中进行参数估计，进而计算出**对数似然比**作为替换打分，比如广泛使用的 BLOSUM [?] 系列替换打分矩阵。我们将在“延伸阅读”中会简要解释采用对数似然比的原因。

3.8 字符串局部联配：对一对字符串的归约

对于单词拼写改错而言，我们关注的是单词 x 与 y 的**整体**，计算的是两个单词之间的**全局联配**；然而在很多应用中，两个字符串之间只有中间一部分相似，导致全局联配并不适用。

一个典型的例子是文章抄袭的检测：我们将待比较的两篇文章分别表示成字符串 x 和 y；相对于通篇抄袭而言，部分抄袭更为常见，即 y 中只有一部分来自于 x（经过了一系列替换、插入和删除操作），其余部分则是和 x 无关的原创内容。在这种情形下，如果依然对 x 和 y 做全局联配的话，就会发现虽然抄袭部分的联配会得高分，但是无关部分的罚分却很大；因此整体相似度分值很低，会诱导我们得出“x 和 y 不相似”的错误结论。再如基因序列，从同一个祖先基因演化而成的两个基因，往往只在一部分区域上表现出显著相似性，但整体来看很不相似 [?]。

与计算两个字符串的整体相似度相反，上述例子关注的是**局部片段对**之间的相似性；两个字符串中局部片段对的相似度计算被称为**局部联配**（Local alignment）。

为便于描述局部联配算法，我们引入一个新的概念，叫做**修正相似度** [?]。在上一节中，我们描述了如何衡量和计算两个字符串 x, y 之间的相似度 $Sim(\mathrm{x},\mathrm{y})$。通过合理地设置替换打分和罚分的大小，我们总可以使得对于**不相似**的字符串 x 和 y 来说，有 $Sim(\mathrm{x},\mathrm{y}) \le 0$。在此基础上，我们定义修正相似度 $\overline{Sim}(\mathrm{x},\mathrm{y})$：

$$\overline{Sim}(\mathrm{x},\mathrm{y}) = \max\{Sim(\mathrm{x},\mathrm{y}), 0\}.$$

直观地看，$\overline{Sim}(\mathrm{x},\mathrm{y})$ 只是将不相似字符串 x 和 y 的相似度从负数修正为 0，而对于相似字符串的相似度则不做改变。以空字符串和字符串 "$y_1y_2\cdots y_j$" 为例，其相似度为：

$$Sim(\text{""}, \text{"}y_1y_2\cdots y_j\text{"}) = -3\times j,$$

因此其修正相似度为：

$$\overline{Sim}(\text{""}, \text{"}y_1y_2\cdots y_j\text{"}) = 0.$$

我们可以合理地假设最相似片段对 $\bar{\mathrm{x}}$ 和 $\bar{\mathrm{y}}$ 至少有一个字符相同，因此相似度 $Sim(\bar{\mathrm{x}}, \bar{\mathrm{y}}) > 0$，其修正相似度与相似度相同。采用修正相似度定义，我们可以将字符串局部联配问题描述如下：

字符串局部联配问题（Sequence local alignment problem）

输入：两个字符串 $\mathrm{x} = x_1x_2\cdots x_m$ 和 $\mathrm{y} = y_1y_2\cdots y_n$；

输出：x 和 y 中的*最相似片段对*，即 x 中的连续片段 $\bar{\mathrm{x}}$ 和 y 中的连续片段 $\bar{\mathrm{y}}$，其修正相似度 $\overline{Sim}(\bar{\mathrm{x}}, \bar{\mathrm{y}})$ 最高。

为便于说明问题，我们在本节中采用如下字符替换打分函数：

$$s(x,y) = \begin{cases} -3 & \text{如果} x = \text{'-'} \text{ 或} y = \text{'-'} \\ 1 & \text{如果} x = y \\ -1 & \text{如果} x \neq y \end{cases}$$

3.8.1 算法设计与描述

局部联配问题的*困难*在于最相似片段对的*起始点和终止点是未知的*。一种直观但是笨拙的方法是归结为多个全局联配，即：枚举 x 和 y 中所有的片段，并对这些片段两两做全局联配，最后找出相似度最高的片段对来。虽然这个方法是正确的，但是片段对的数目很多，共有 $O(m^2n^2)$ 个，导致这个方法效率非常低 [**?**]。

那如何改进以提升效率呢?

一个重要的观察是，上述枚举方法中存在着大量的冗余计算，表现为即使片段对不同，全局联配 OPT 表中也会有很多单元是相同的（示例见图 3.21）；因此，只要避免这些冗余计算，将会显著提升算法效率。

1981 年，T. F. Smith 和 M. S. Waterman 提出了一种避免上述冗余计算的高效算法（称作 Smith-Waterman 算法 [**?**]），能够只使用一个 OPT 表即可同时计算出所有片段对的联配；直观上看，片段对联配时使用的小 OPT 表都被*迭放*在字符串联配的大 OPT 表上，从而避免了冗余计算（见图**??**）。Smith-Waterman 算法的*基本思想*描述如下：

由于问题的难点在于最相似片段对的起始点和终止点是未知的，因此我们先来解决这个问题的简化版本：*当已知最相似片段对的终止点时，识别出最相似片段对*。具

	Y: ''	G	T	C	C	C
X:''	0	−3	−6	−9	−12	−15
G	−3	1	−2	−5	−8	−11
T	−6	−2	2	−1	−4	−7
C	−9	−5	−1	3	0	−3
G	−12	−8	−4	0	2	−1
C	−15	−11	−7	−3	1	3
T	−18	−14	−10	−6	−2	0

(a)

	Y: ''	G	T	C	C	C	T
X:''	0	−3	−6	−9	−12	−15	−18
G	−3	1	−2	−5	−8	−11	−14
T	−6	−2	2	−1	−4	−7	−10
C	−9	−5	−1	3	0	−3	−6
G	−12	−8	−4	0	2	−1	−4
C	−15	−11	−7	−3	1	3	0

(b)

图 3.21: 不同字符串对的全局联配 OPT 表中的相同单元。(a) 字符串 x = "GTCGCT" 和 y = "GTCCC" 的全局联配 OPT 表；(b) 字符串 x = "GTCGC" 和 y = "GTCCCT" 的全局联配 OPT 表

体说来，我们考虑 x 中所有以 x_i 结尾的片段与 y 中所有以 y_j 结尾的片段，识别出最相似的片段对，其修正相似度记为 $OPT(i,j)$。

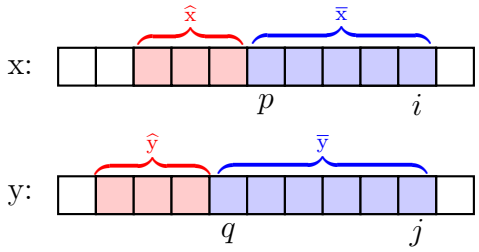


图 3.22: 序列 x 和 y 中的结束于 x_i 和 y_j 的最相似片段对 $\bar{x}$ 和 $\bar{y}$。我们记该片段对的起始点为 x_p 和 y_q，并记结束于 x_{p-1} 和 y_{q-1} 的最相似片段对为 $\hat{x}$ 和 $\hat{y}$

如图 3.22所示，我们记终止于 x_i 和 y_j 的最相似片段对为 $\bar{x}$ 和 $\bar{y}$，其最优联配记为 $(\bar{x}', \bar{y}')$。我们把联配 $(\bar{x}', \bar{y}')$ 的构造过程描述成一个**多步决策过程**：在每一步，我们考虑 $\bar{x}$ 和 $\bar{y}$ 中的一对字符，决定如何对其进行扩展以生成 $\bar{x}'$ 和 $\bar{y}'$。以第一步对字符 x_i 和 y_j 的决策为例，有如下 3 种方式生成联配：

(1) 向 $\bar{x}'$ 中添加字符 x_i，向 $\bar{y}'$ 中添加字符 y_j：如果字符 $x_i = y_j$，则对最终形成的联配贡献 1 分，否则贡献-1 分；剩下的子问题是识别出终止于 x_{i-1} 和 y_{j-1} 的最相似片段对，并计算最优联配；

(2) 向 $\bar{x}'$ 中添加占位符'-'，向 $\bar{y}'$ 中添加字符 y_j：'-' 和 y_j 对齐对最终形成的联配贡献 $s(\text{'-'}, y_j) = -3$ 分；剩下的子问题是识别出终止于 x_i 和 y_{j-1} 的最

相似片段对，并计算最优联配；

(3) 向 $\bar{x}'$ 中添加字符 x_i，向 $\bar{y}'$ 中添加占位符'-'：x_i 和'-' 对齐对最终形成的联配贡献 $s(x_i, \text{'-'}) = -3$ 分；剩下的子问题是识别出终止于 x_{i-1} 和 y_j 的最相似片段对，并计算最优联配。

值得特别强调的是这里的子问题与全局联配中子问题的差别：在这 3 种情形中，子问题求解出的最相似片段对的起始点通常并不相同（见图 3.23）；而在全局联配的 3 种情形中，子问题要计算的是两个字符串前缀的最优联配，其起始点都是 0（见图**??**）。

依据上述分析，$OPT(i,j)$ 可采用如下递归关系式计算：

$$OPT(i,j) = \max \begin{cases} OPT(i-1, j-1) + s(x_i, y_j) \\ OPT(i-1, j) + s(x_i, \text{'-'}) \\ OPT(i, j-1) + s(\text{'-'}, y_j) \\ 0 \end{cases}$$

Smith-Waterman 算法采用迭代技术计算出 $OPT(m,n)$（伪代码见*算法* 32），然后采用下述方式识别出字符串 x 和 y 的最相似片段：

Algorithm 32 计算局部序列联配的 Smith-Waterman 算法

function Smith-Waterman(x, y, m, n)

1: Set $m = |\text{x}|$ and $n = |\text{y}|$;
2: Set $OPT[i,0] = 0$ for each $i = 0$ to m and set $OPT[0,j] = 0$ for each $j = 0$ to n;
3: **for** $j = 1$ to n **do**
4: **for** $i = 1$ to m **do**
5: $OPT[i,j] = \max\{OPT[i-1,j-1] + s(x_i, y_j), OPT[i-1,j] - 3, OPT[i,j-1] - 3, 0\}$;
6: **end for**
7: **end for**
8: **return** the largest element in OPT table;

(1) *先确定最相似片段对的终止点*：由于 $OPT(i,j)$ 表示的是结束于 x_i 和 y_j 的最相似片段对的修正相似度，因此整个字符串 x 和 y 中最相似片段对的修正相似度可由 $\max_i \max_j OPT(i,j)$ 来求出。因此，我们在 OPT 表中寻找最大单元，即是终止点。

(2) **再确定最相似片段对的起始点**：设最相似片段对的终止点为 x_i 和 y_j，其起始点为 p 和 q，而终止于 x_{p-1} 和 y_{q-1} 的最相似片段记为 $\hat{\mathrm{x}}$ 和 $\hat{\mathrm{y}}$（见图 3.22），我们可以证明 p 和 q 具有如下性质：

$$\overline{Sim}(\hat{\mathrm{x}},\hat{\mathrm{y}})=0.$$

证明是用反证法完成的：假设 $\overline{Sim}(\hat{\mathrm{x}},\hat{\mathrm{y}})>0$，则：

$$\begin{aligned}\overline{Sim}(\hat{\mathrm{x}}\overline{\mathrm{x}},\hat{\mathrm{y}}\overline{\mathrm{y}}) &\geq \overline{Sim}(\hat{\mathrm{x}},\hat{\mathrm{y}})+\overline{Sim}(\overline{\mathrm{x}},\overline{\mathrm{y}})\\ &> \overline{Sim}(\overline{\mathrm{x}},\overline{\mathrm{y}}).\end{aligned}$$

而这与 $(\overline{\mathrm{x}},\overline{\mathrm{y}})$ 是最相似片段对矛盾。

因此我们可以按照如下方法确定起始点：从终止点开始，按照 OPT 表逐步回溯，直至到达一个单元 $OPT(p-1,q-1)=0$，则 x_p 和 y_q 即为起始点。

3.8.2 算法运行过程实例

图 3.24展示 Smith-Waterman 算法计算字符串 x = "AGTCGCTT" 和字符串 y = "TGTCCCT" 最相似片段对的识别过程：首先，算法初始化 OPT 表格的 $i=0$ 行和 $j=0$ 列，其中单元全部设置为 0（注意这一点和 Needleman-Wunsch 算法的差异）；然后从 $j=1$ 列开始逐列计算；在每一列，都是从上往下逐个计算 OPT 表格中的单元。

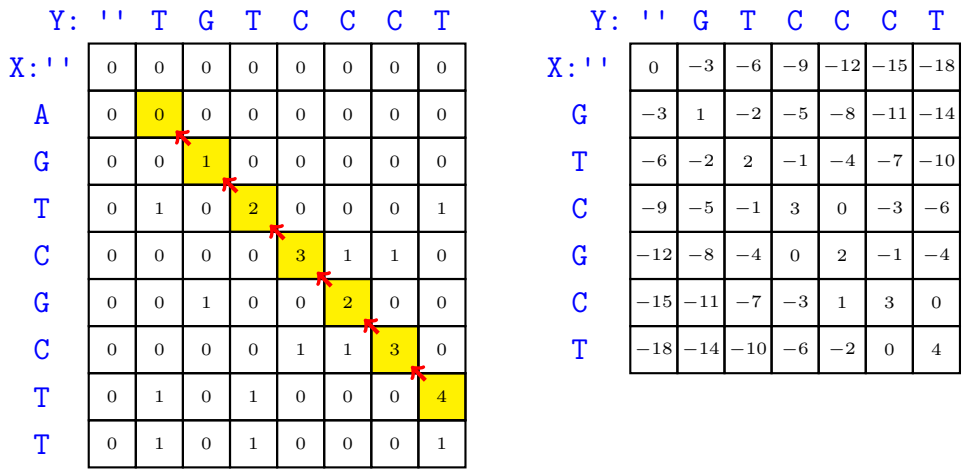
在计算完 OPT 表之后，我们找出最大值 $OPT(8,9)=4$，然后从此单元开始回溯，直至值为 0 的单元。回溯路径即对应着 x 和 y 中的最相似片段对：$\overline{\mathrm{x}}$ ="GTCGCT"，$\overline{\mathrm{y}}$ = "GTCCCT"。

Substitution	Insertion	Deletion
$\overline{\mathrm{x}}'$: T	$\overline{\mathrm{x}}'$: T -	$\overline{\mathrm{x}}'$: G T
$\overline{\mathrm{y}}'$: G	$\overline{\mathrm{y}}'$: T G	$\overline{\mathrm{y}}'$: G -

图 3.23: 计算结束于 x_3 = "T" 与 y_2 = "G" 的最相似片段对的 3 种选择项

值得仔细观察的是 $OPT(i,j)$ 递归表达式中第 4 个选择项“0”的作用。以 $OPT(3,2)$ 的计算为例，我们有：

$$OPT(3,2)=\max\begin{cases}OPT(2,1)-1 & (=\text{-}1)\\ OPT(3,1)-3 & (=\text{-}2)\\ OPT(2,2)-3 & (=\text{-}2)\\ 0\end{cases}=0$$



(a) (b)

图 3.24: Smith-Waterman 算法计算字符串 x = "AGTCGCTT" 和 y = "TGTCCCT" 的最相似片段对的运行过程示例。(a) Smith-Waterman 算法计算出的局部联配 OPT 表；(b) 最相似片段对 $\bar{x}$ = "GTCGCT" 和 $\bar{y}$ = "GTCCCT" 的全局联配 OPT 表

$OPT(3,2)$ 表示的是结束于 x_3 = "T" 与 y_2 = "G" 的最相似片段对的修正相似度。前三个选择项对应的联配如图 3.23所示：这三种选择项得到的联配打分都小于 0，意味着如果以这三种选择项为基础计算后续 x_4 = "C" 与 y_3 = "T" 的联配，还不如从头开始，即以空字符串 $\bar{x}$ = "" 和 $\bar{y}$ = "" 重新开始计算后续的联配—这就是第 4 个选择项 "0" 的作用。

此外，我们还可以考察局部联配和全局联配之间的关系。以最相似片段对 $\bar{x}$ = "GTCCGC" 和 $\bar{y}$ = "GTCCCT" 为例：其全局联配 OPT 表明 $\bar{x}$ 和 $\bar{y}$ 的相似度是 4（见图 3.24(b)），这与 Smith-Waterman 算法计算出的修正相似度是相同的。更重要的是，由最优全局联配得出的回溯路径上单元的值与局部联配是相同的。这也验证了我们在本节一开头的说法：直观上看，Smith-Waterman 算法得到的 OPT 表相当于所有片段对的全局联配 OPT 表的“叠合”，从而使得 Smith-Waterman 算法只使用一个 OPT 表格即可计算出所有片段对的修正相似度。

3.8.3 时间复杂度分析

Smith-Waterman 算法中 OPT 有 $O(mn)$ 个单元；计算每个单元只需 3 次加法和 4 次比较运算，因此时间复杂度和空间复杂度都是 $O(mn)$。

3.8.4 一些讨论

识别最相似片段对和次相似片段对

对于两个字符串 x 和 y 来说，除最相似片段对 $\bar{x}$ 和 $\bar{y}$ 之外，通常还存在一些次相似片段对，其相似度仅比最相似片段对略低。因此，次相似片段对的识别也是很有价值的。

识别次相似片段对的一个简单策略是：采用 Smith-Waterman 算法识别出最相似片段对 $\bar{x}$ 和 $\bar{y}$ 之后，从 x 和 y 中去除 $\bar{x}$ 和 $\bar{y}$，然后重新运行 Smith-Waterman 算法，在剩下的字符串中识别最相似片段对。如此不断重复，直至识别出所有的次相似片段对。

这种策略虽然可行，但是计算复杂。M. Waterman 等提出了一种高效 declump 算法 [**?**]：在计算出最相似片段对之后，沿着最优联配修改 OPT 表，以去除所有和最优联配有交集的联配的影响；然后在修改后的 OPT 表中继续寻找最优联配。

字符串联配算法的并行化

Smith-Waterman 算法中存在着高度的并行性：OPT 表中的所有列都是可以同时进行计算的；由于 $OPT(i,j)$ 依赖于 $OPT(i-1,j-1)$ 和 $OPT(i-1,j)$，因此计算 $OPT(i,j)$ 时需要等待 $OPT(i-1,j)$ 计算完成方可进行。

徐琳等设计了执行 Smith-Waterman 并行算法的 FPGA 卡；卡上包含多个处理单元（PE, Processing Element），每个 PE 负责 OPT 表中一列的计算；通过控制 PE 的时序，保证第 j 列的 PE 比第 $j-1$ 列的 PE“慢一拍”即可实现多个 PE 的并行运算。在实际数据上的测试表明，采用 FPGA 卡能够达到上千倍的加速比 [**?**, **?**]。

3.9 树上的顶点覆盖问题：对树的归约

顶点覆盖问题（Vertex cover）来源于下述的实际需求：如图 3.25所示的 7 个地点，地点之间有道路相连接；在一个地点上部署摄像头，将能够监控与此顶点邻接的所有道路。问：最少部署几个摄像头，即可监控所有的道路?

对于这个实例来说，至少需要部署 3 个摄像头；其中一种部署方案见图 3.25中的蓝色顶点。

我们在第 9 章将会看到：对于一般的图来说，顶点覆盖问题是 NP-完全问题，迄今还未设计出多项式时间的求解算法。在此，我们考虑这个问题的受限版本：顶点和

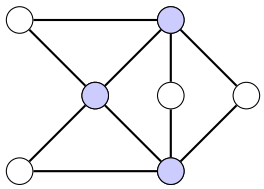


图 3.25: 顶点覆盖问题实例。选择蓝色点所示 3 个顶点，可覆盖所有的边

边构成的不是一般的图，而是一棵树。树上的顶点覆盖问题形式化描述如下：

树上的顶点覆盖问题（Vertex cover over a tree）

输入：一颗包含 n 个顶点的树，其根顶点记为 r；

输出：选择最少的顶点，能够覆盖所有的边。所谓覆盖一条边，是指其两个端点至少有一个被选择。

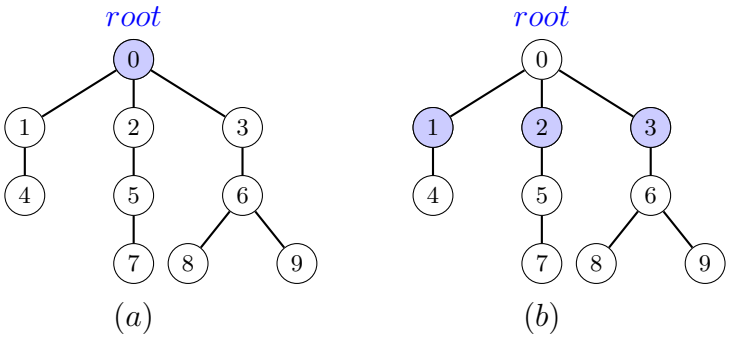


图 3.26: 树上的最小顶点覆盖问题实例及其求解。(a) 选择根顶点；(b) 不选择根顶点

3.9.1 算法设计与描述

对规模较小的实例，我们很容易直接求出最小顶点覆盖：比如只有 1 个顶点的树，其最小顶点覆盖的顶点数是 0；只有 2 个顶点的树，其最小顶点覆盖的顶点数是 1。然而当树比较大时，难以直接求解，因此我们转而尝试将大的实例分解成小的实例。

注意到问题的完整解是“被选出的一些顶点构成的子集”，因此我们将求解过程描述成从空集开始、逐步添加顶点的**多步决策过程**：从根顶点开始，我们每次考虑一个顶点，所做的决策是选择这个顶点还是不选这个顶点。

以求解过程的第一个决策为例（图 3.26），对于根顶点 r 来说，最优解中有如下两个决策选择项：

(1) 选择根顶点 r：则边 $(0,1),(0,2),(0,3)$ 已经被覆盖。因此顶点 $1,2,3$ 可以选择，也可以不选；故剩余的子问题的是求解以顶点 1 为根顶点的子树、以顶点 2 为根顶点的子树、以顶点 3 为根顶点的子树中的最小顶点覆盖。

(2) 不选择根顶点 r：则为了覆盖边 $(0,1),(0,2),(0,3)$，我们必须选择顶点 $1,2,3$；故剩余的子问题是求解以顶点 4 为根顶点的子树、以顶点 5 为根顶点的子树、以顶点 5 为根顶点的子树中的最小顶点覆盖。

我们对上述子问题进行归纳，定义子问题的一般形式为：求以顶点 v 为子树的最小顶点覆盖，并记最小顶点覆盖的顶点数目为 $OPT(v)$。

由上述分析可知：原始问题的最优解可被拆分成单步决策的决策项，以及子问题的最优解两个部分；因此，我们可以得到如下的递归表达式：

$$OPT(v)=\begin{cases}0 & \text{如果}C(v)=\emptyset\\ \min\{1+\sum\limits_{c\in C(v)}OPT(c),\#children+\sum\limits_{g\in G(v)}OPT(g)\} & \text{否则}\end{cases}$$

其中 $C(v)$ 表示顶点 v 的儿子顶点，$G(v)$ 表示 v 的孙子顶点。

特别地，原始给定问题的最优解的值可以用 $OPT(r)$ 来表示。我们可采用递归调用技术或迭代技术来计算 $OPT(r)$；算法的详细描述从略。

3.9.2 时间复杂度分析

我们依然按照“子问题的数目”和“计算每个子问题所需的时间”来分析算法的时间复杂度。由于对每个顶点 v 对应一个子问题，因此共有 n 个子问题；而计算 $OPT(v)$ 时，需要考虑其所有儿子顶点 c 的最优解值 $OPT(c)$ 和孙子顶点 g 的最优解值 $OPT(g)$。

儿子顶点 c 的数目还好估计，但是孙子顶点 g 的数目就不太好估计。鉴于这个困难，我们反过来思考：看一个顶点“被”考虑过几次。对于任意一个顶点 v 来说，它“被”考虑过最多 2 次：一次是作为儿子顶点，另一次是作为孙子顶点。因此，算法的时间复杂度是 $O(n)$，是多项式时间算法。

3.9.3 算法设计：更“细”的子问题表述方式

上文中采用“以 v 为根顶点的子树”来表示子问题。下面我们尝试更“细”的表示方式：用“以 v 为根顶点的子树、第一步决策的选择项”二者的组合来表示子问题。直观上看，这是在子问题描述中增加一个表示决策的变量，从而将子问题拆分得更“细”一些。采用这个策略，我们定义如下 2 个子问题：

(1) 求以顶点 v 为根顶点的子树的最小顶点覆盖，其中根顶点 v 必选；最小顶点覆盖的大小记为 $OPT(v, \texttt{YES})$。

(2) 求以顶点 v 为根顶点的子树的最小顶点覆盖，其中根顶点 v 不选；最小顶点覆盖的大小记为 $OPT(v, \texttt{NO})$。

和上一小节所示方案不同，这里我们定义了两个递归变量 $OPT(v, \texttt{NO})$ 和 $OPT(v, \texttt{YES})$，并可建立如下递归表达式：

$$OPT(v, \texttt{NO}) = \begin{cases} 0 & \text{如果} C(v) = \emptyset \\ \sum\limits_{c \in C(v)} OPT(c, \texttt{YES}) & \text{否则} \end{cases}$$

以及

$$OPT(v, \texttt{YES}) = \begin{cases} 1 & \text{如果} C(v) = \emptyset \\ 1 + \sum\limits_{c \in C(v)} \min\{OPT(c, \texttt{YES}), OPT(c, \texttt{NO})\} & \text{否则} \end{cases}$$

采用这种子问题定义，原始给定问题可以通过 $\min\{OPT(r, \texttt{YES}), OPT(r, \texttt{NO})\}$ 来求解。我们可以设计递归算法或迭代算法来求解，在此也不赘述。

这种更“细”的子问题定义带来的好处是算法时间复杂度易于分析：

(1) **子问题数目**：对于每个顶点 v，我们有 2 个子问题，因此共有 $2n$ 个子问题；

(2) **求解每个子问题所花费的时间**：与顶点 v 的子顶点数目成正比。

因此，算法的时间复杂度为：$2\sum_v 2\#children(v) = O(n)$。

3.10 有向无环图上的单源最短路径：对图的归约

在 GPS 导航系统中，一个常见的任务是规划从出发地到目的地的最短路径。这个任务可以形式化建模成在一个赋权有向图中寻找两个顶点间的路径，其经过的所有边的权重加和最小。边的权重不限于表示距离；为了方便起见，我们依然沿用“最短路径”这个称呼。最短路径问题可以分为如下 3 类：

(1) **一对顶点间的最短路径问题**（Single-pair shortest paths）：求从特定的源顶点（Source）到特定的目的顶点（Destination）的最短路径；

(2) **单源最短路径问题**（Single-source shortest paths）：计算从特定的源顶点到每一个顶点的最短路径，或者对称地，从每一个顶点到特定的目的顶点的最短路径；

(3) 所有顶点对之间的最短路径问题（All-pairs shortest paths）：求所有顶点对之间的最短路径。

其中单源最短路径问题和一对顶点间最短路径问题没有本质性的差异：即使知道特定的目的结点，但是无法预知从哪些中间结点走最近，因此在实际求解时，往往是求出了到达所有结点的最短路径。与这两个问题不同，计算所有顶点对之间的最短路径问题需另作讨论。

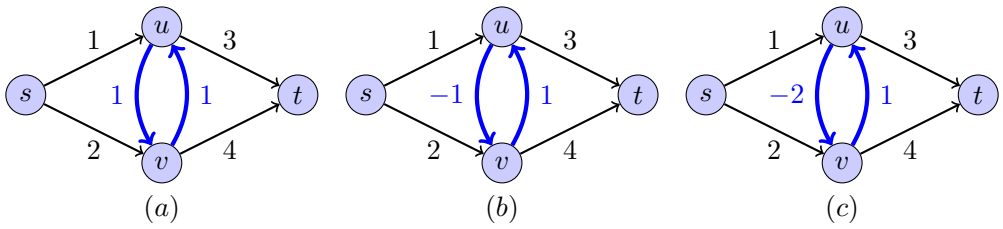


图 3.27: 有向图中的正权重环、0 权重环和负权重环

有向图中的最短路径和环有着密切关系。我们定义环的权重为环中边的权重加和，并依据权重将环分作如下 3 类：

(1) 正权重环：对正权重环的一个重要认识是其不会出现在最短路径上。这一点很容易证明：假如路径 $p = \langle v_0, v_1, \cdots, v_k \rangle$ 中存在一个权重为正的环 $c = \langle v_i, v_{i+1}, \cdots, v_j \rangle$（即 $v_j = v_i$），那么我们从路径 p 中去除 c，生成新的路径 $p = \langle v_0, \cdots, v_i, v_{j+1}, \cdots v_k \rangle$；由于环 c 的权重为正，因此 p' 是比 p 更短的路径。这个结论意味着在求最短路径时，正权重的环会自动被排除掉，因此无需特殊考虑。

(2) 0 权重环：如图 3.27(b) 所示，考虑 s-t 最短路径 $p = \langle s, u, v, u, t \rangle$，我们从中去除一个权重为 0 的环 $c = \langle u, v, u \rangle$ 之后，就会得到另一条路径 $p' = \langle s, u, t \rangle$；由于环的权重为 0，因此 p' 也是最短路径，且顶点数目比 p 少；如此不断重复，最终会得到一个不包含权重为 0 的环的最短路径。这个结论，以及上面对正权重环的结论，综合起来意味着对有向图 $G = \langle V, E \rangle$ 来说，我们只需考虑不超过 $n - 1$ 条边的、无重复顶点的简单路径（Simple path）即可。

(3) 负权重环：与上两类环不同，负权重环会影响最短路径的计算：如图 3.27(c) 所示，环 $c = \langle u, v, u \rangle$ 的权重为-1；s-t 路径 $p = \langle s, u, t \rangle$ 的长度是 4；但经过环 c 一次，形成新的路径 $p' = \langle s, u, v, u, t \rangle$，其权重为 3；如此重复，会生成

权重为 $-\infty$ 的 s-t 路径。这意味着当存在着 s 可达的负权重环时，最短路径不是“良定义”的（Well-defined）。

鉴于环对于最短路径的重要性，我们依据是否存在环对图分类：i) 不存在环的有向图，称为**有向无环图**（Directed acyclic graph，简记为 DAG）；ii) 一般的有向图。对于一般有向图来说，我们不仅需要在没有负权重环时能够计算最短路径，同时还需要判断是否存在负权重环。我们首先来研究有向无环图中的最短路径问题，描述如下：

有向无环图上的单源最短路径问题（Single-source shortest-paths in DAG）

输入：有向无环图 $G = \langle V, E\rangle$，边的权重 $w : E \to \mathbf{R}$，源顶点 s，其中 $|V| = n$，$|E| = m$；

输出：s 到每一个顶点 $v \in V$ 的最短路径。

3.10.1 算法设计与描述

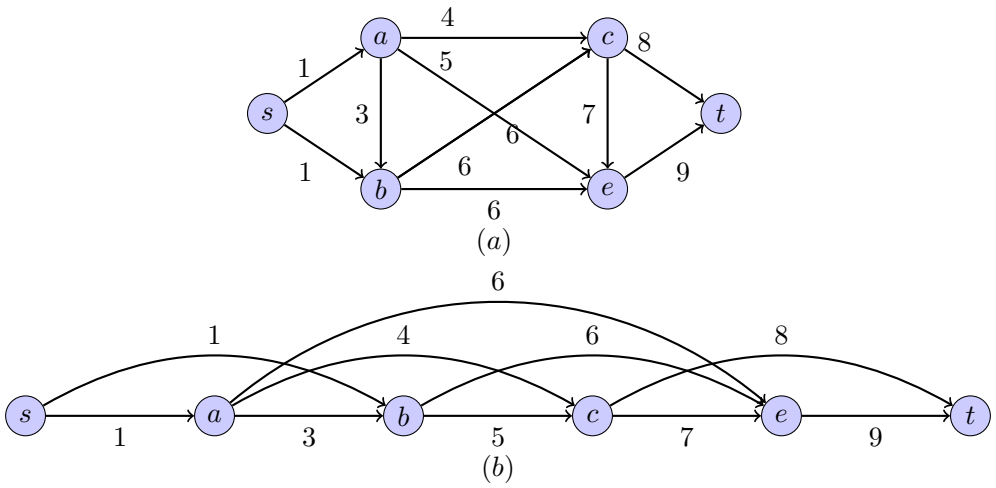


图 3.28: 有向无环图及顶点的拓扑排序示例

如图 3.28(a) 所示，$G = \langle V, E\rangle$ 中从源顶点 s 到目的顶点 t 的最短路径求解过程可以描述成如下的**多步决策过程**：从 t 开始，每步确定最短路径中一条边。以第一步对 t 的决策为例，由于 t 有两个前驱顶点 b 和 d，因此我们需要在如下两个决策项中做最优选择：

(1) **从 b 到达 t**：剩下的子问题是计算从 s 到 b 的最短路径；

(2) 从 d 到达 t：剩下的子问题是计算从 s 到 d 的最短路径。

之所以剩下的子问题还是一个寻找最短路径的问题，是因为最短路径具有如下性质：最短路径中的子路径是其端点间的最短路径，严格表述如下：

定理 3.10.1. 给定赋权有向图 $G = \langle V, E\rangle$，边上的权重函数 $w : E \to \mathbf{R}$，设 $p = \langle v_1, v_2, \cdots, v_k\rangle$ 是从顶点 v_1 到 v_k 的最短路径，$p_{ij} = \langle v_i, v_{i+1}, \cdots, v_j\rangle$ 是 p 的子路径，则 p_{ij} 是从顶点 v_i 到 v_j 的最短路径。

证明：

我们采用反证法进行证明：

首先将路径 p 拆分成三部分：$p = v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$，则有 $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$。

然后假设存在另一条从 v_i 到 v_j 的路径 p'_{ij}，比 p_{ij} 更短，即：$w(p'_{ij}) < w(p_{ij})$，则我们可以构造出一条新的从 v_1 到 v_k 的路径：

$$p' = v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

且有 $w(p') = w(p_{1i}) + w(p'_{ij}) + w(p_{jk}) < w(p)$，而这与 p 是最短路径矛盾，因此原结论成立。 □

我们用 $d(v)$ 表示从 s 到顶点 v 的最短距离，则可得到如下递归关系和基始赋值：

$$d(v) = \begin{cases} 0 & \text{如果} v = s \\ \min\limits_{(u,v)\in E}\{d(u) + w(u,v)\} & \text{否则} \end{cases}$$

基于上述递归关系计算 $d(v)$ 时，我们既可以采用递归调用的方式，也可以采用迭代方式。当采用迭代方式计算时，我们需要在计算 $d(v)$ 之前，先得计算出其每个前驱顶点 u 的值 $d(u)$ 来。那么怎样规定计算顺序，才能使得每个顶点都排在其前驱顶点之后呢?

对于一般有向图来说，这不一定能够做到；但是对于有向无环图来说，却总是可以做到的：将所有顶点排成一个顶点序列，使得每个顶点都在其前驱顶点之后，即边的方向都是从左向右（见图 3.28)。这个操作称为线性化（Linearization)，也称为拓扑排序（Topological sorting)，可以通过对深度优先搜索算法略作修改即可完成 [**?**, **?**]。

在完成拓扑排序之后，我们即可从 s 开始，按照拓扑顺序，逐个计算每个顶点的 $d(v)$。算法的伪代码描述见算法 33，其中 $\pi[v]$ 记录在最短路径上 v 的前驱顶点，这样逐级回溯直至源顶点 s，即可得到完整的 s-v 最短路径。对于图 3.28所示实例，算法计算出的最短距离见表 3.5。

Algorithm 33 计算有向无环图中单源最短路径的动态规划算法

function DAG-SHORTEST-PATHS($G = \langle V, E\rangle, s$)

1: L=TOPOLOGICAL-SORT(G);
2: Set $d(v) = \infty$ for each node $v \in L$, and $d(s) = 0$;
3: **for** $i = 1$ to $|L|$ **do**
4: $v = L[i]$;
5: $d(v) = \min\limits_{(u,v)\in E}\{d(u) + w(u, v)\}$;
6: $\pi(v) = \operatorname{argmin}\limits_{(u,v)\in E}\{d(u) + w(u, v)\}$;
7: **end for**

function TOPOLOGICAL-SORT($G = \langle V, E\rangle$)

1: Set L as an empty list, and set $color(u) =$ `WHITE` for each node $u \in V(G)$;
2: **for** each node $u \in V(G)$ **do**
3: Run DFS-VISIT(u) if $color(u) \neq$ `BLACK`;
4: **end for**
5: **return** L;

function DFS-VISIT(u)

1: Report "Found a cycle" and exit if $color(u) ==$ `GRAY`;
2: Set $color(u) =$ `GRAY`;
3: **for** each edge $(u, v) \in E$ **do**
4: Run DFS-VISIT(v) if $color(v) \neq$ `BLACK`;
5: **end for**
6: Set $color(u) =$ `BLACK`;
7: Add u to the head of L;

3.10.2 时间复杂度分析

在 DAG-SHORTEST-PATHS 算法中，拓扑排序的运行时间是 $O(m+n)$ 的，而计算 $d(v)$ 步骤的运行时间是 $O(m)$ 的，因此时间复杂度是 $O(m+n)$ 的；此处 $m = |E|$ 表示图的边数，$n = |V|$ 表示图的顶点数。

值得特别指出的是，对于有向无环图来说，我们既可以快速计算出最短路径，也可以快速求出最长路径：我们只需要将所有边的权重取反，然后运行 DAG-SHORTEST-PATHS 算法即可求出最长路径。自然，对于一般图来说，"对权重取反"会导致负权重环变成正权重环、而正权重环会变成负权重环；因此这种策略不适用于一般图。事实上，我们在第九章中将会看到求最长简单路径是 NP-完全问题。

表 3.5: DAG-Shortest-Paths 算法运行过程示例（实例见图 3.28）

目的顶点v	最短距离$d(v)$	前驱顶点$\pi(v)$
s	0	-
a	1	s
b	1	s
c	5	a
e	7	b
t	13	c

3.11 一般有向图上的单源最短路径：对图的归约

我们考虑一般有向图中的单源最短路径问题，描述如下：

有向图上的单源最短路径问题（Single-source shortest-paths in directed graph, SSSP）
输入：有向图 $G = \langle V, E\rangle$，边上的权重 $w : E \to \mathbf{R}$，源顶点 s，其中顶点数 $|V| = n$，边数 $|E| = m$；
输出：当 G 不包含负权重环时，计算 s 到所有顶点 $v \in V$ 的最短路径；否则报告“存在负权重环”。

对于一般有向图中的最短路径问题而言，其难点在于可能存在负权重环。如上节所述，当存在 s 可达的负权重环时，从 s 到环上顶点 v 的最短距离 $d(v)$ 不是良定义的。这里，我们首先在不存在负权重环的假设之下研究如何计算最短路径，然后再来讨论如何判定图中是否存在负权重环。

3.11.1 算法设计与描述

依据原始问题的形式直接定义子问题：一个不成功的尝试

我们先来尝试依据原始问题的形式直接定义子问题，即：给定不存在 s 可达的负权重环的图，计算从 s 到顶点 v 的最短路径和最短距离 $d(v)$。这种策略如果成功的话，可直接求出原始问题的解，非常方便。

然而不幸的是，我们无法建立 $d(v)$ 之间的直接递归关系。如图 3.29(a) 所示，G_1 包含一个正权重环 $a \to b \to a$，即：a 是 b 的前驱顶点，b 也是 a 的前驱顶点；如果

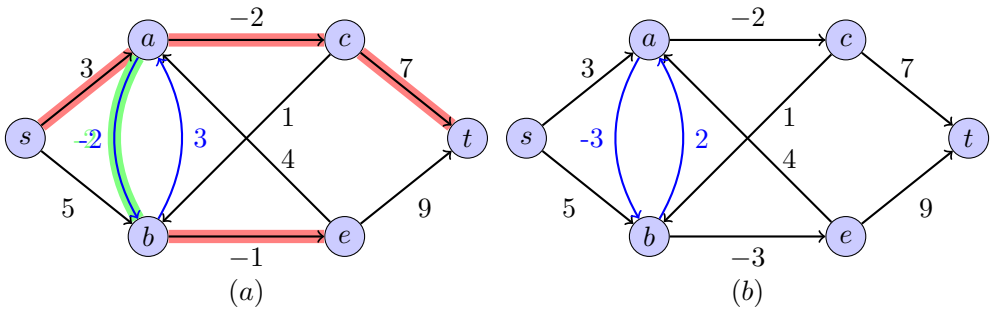


图 3.29: 带有环的有向图示例。(a) 有向图 G_1 包含 s 可达的正权重环 $a \to b \to a$，阴影边表示到达各个顶点的最短路径；(b) 有向图 G_2 包含 s 可达的负权重环 $a \to b \to a$

我们还是按照有向无环图中的推理的话，会得到下述表达式：

$$d(a) = \min\{3, 3 + d(b), 4 + d(e)\}$$

$$d(b) = \min\{5, -2 + d(a), 1 + d(c)\}$$

也就是说，要计算 $d(a)$，需要事先知道 $d(b)$ 的值；而要计算 $d(b)$，又得事先知道 $d(a)$ 的值。$d(a)$ 和 $d(b)$ 的这种*循环依赖性*使得迭代计算无法起始，而递归调用计算方式则会陷入无限循环。

那如何避免这种循环依赖性，定义出具有递归关系的子问题呢?

子问题间无法建立递归关系的一个常见原因是子问题太“粗”了（我们在陈述完算法设计之后，再来讨论这一点，会看得更清楚）。因此，我们可以把子问题再细分成一些*更“细”的子问题*，“粗”的子问题的解由更“细”的子问题的解组合而成；有时更“细”的子问题之间存在着递归关系，但是这种组合操作却阻碍了“粗”的子问题之间建立递归关系。

构建具有递归性的子问题：添加限制条件

那如何定义更“细”的子问题呢?

一种有效的方法是“拆分”，即：*在子问题定义中添加限制条件*。这里我们尝试在 $d(v)$ 中引入一个描述*决策步数*的变量：注意到当图中不存在 s 可达的负权重环时，从 s 到 t 的最短路径是一条边数不超过 $n-1$ 的简单路径；此处 $n = |V|$ 表示顶点数目。

以图 G_1 中的 s-t 最短路径为例，其求解过程可以描述成如下的*多步决策过程：从目的顶点 t 开始，每步确定最短路径中的一条边*。比如在第一步对 t 做决策时，由于 t 有两个前驱顶点 c 和 d，因此我们需要在如下两个决策项中做最优选择：

(1) *从 c 到达 t*：剩下的子问题是计算从 s 到 c 的、最多包含 $n-2$ 条边的最短

路径；

(2) 从 d 到达 t：剩下的子问题是计算从 s 到 d 的、最多包含 $n-2$ 条边的最短路径。

这个递归关系实质上是说：**最短路径的一部分必定还是最短路径**。通过归纳这两个具体的子问题，我们将**子问题的一般形式**定义为：**计算从 s 到 v 的、最多包含 k 条边的最短路径**，其长度记为 $d(v,k)$。需要注意的是：$d(v,k)$ 比 $d(v)$ 多了一个限制条件，表示的是更“细”的子问题的解。

子问题最优解之间的递归关系

那么能否建立 $d(v,k)$ 之间的递归关系呢？我们分作两种情况讨论：

(1) 当 $k=1$ 时，从 s 出发经过至多 1 条边到达 v，只有可能在存在 (s,v) 边时发生，因此 $d(v,1)=w(s,v)$；

(2) 当 $k>1$ 时，假设最后一步是从前驱顶点 u 到达 v，则剩下的子问题是“从 s 经过至多 $k-1$ 条边到达 u”，因此有：

$$d(v,k)=\min_{(u,v)\in E}\{d(u,k-1)+w(u,v)\}$$

值得注意的是，由于 $d(v,k)$ 定义中表达的是“至多 k 条边的最短 s-v 路径”，换句话说，路径中可能有 $k-1,k-2,\cdots,1$ 条边，因此我们还得加上 $d(v,k-1),d(v,k-2),\cdots,d(v,1)$ 等选择项，得到如下递归表达式：

$$d(v,k)=\min\{d(v,1),\cdots,d(v,k-1),\min_{(u,v)\in E}\{d(u,k-1)+w(u,v)\}\}$$

进一步地，在递归计算时，$d(v,k-1)$ 实质上已经表达了 $d(v,k-2),\cdots,d(v,1)$，因此上式可简化为：

$$d(v,k)=\min\{d(v,k-1),\min_{(u,v)\in E}\{d(u,k-1)+w(u,v)\}\}$$

采用这个更“细”的子问题定义，原始问题的最优解可以使用子问题的最优解表示：$d(v,n-1)$ 即为 s-v 的最短距离。

我们采用迭代方式计算 $d(v,k),1\le k\le n$；相应的算法称为 Bellman-Ford 算法 [**?**]，其伪代码见**算法** 34。为简化起见，这个算法没有采用矩阵方式详尽地记录每一个 $d(v,k)$ 的值，而是采用一个数组 d 记录最短路径的距离，另一个数组 π 记录每个结点在最短路径上的前驱结点。在第 k 轮循环时，$d[v]$ 记录了 $d(v,k)$ 的值，同时相应地更新 $\pi[v]$。

Algorithm 34 计算一般有向图上单源最短路径的 Bellman-Ford 算法

function Bellman-Ford($G = \langle V, E\rangle, s$)

Set $d[s] = 0$, set $d[v] = \infty$ and $\pi[v] =$ `NIL` for each node $v \in V$;
for $k = 1$ to $|V|$ **do**
  for each edge $(u, v) \in E$ (in an arbitrary order) **do**
    if $d[v] > (d[u] + w(u, v))$ **then**
      $d[v] = d[u] + w(u, v)$;
      $\pi[v] = u$;
    end if
  end for
end for
if $d[v] > (d[u] + w(u, v))$ for a certain edge (u, v) **then**
  return "Found negative cycle.";
end if
return array d and π;

3.11.2 算法运行过程示例

对于图 3.29(a) 所示的图 G_1，Bellman-Ford 算法运行过程如下表所示：

表 3.6: 对含正权重环的图 G_1 计算出的 $d(v, k)$，其中第 6 列和第 5 列完全相同

	路径边数上限 k					
目的顶点v	1	2	3	4	5	6
s	0	0	0	0	0	0
a	3	3	3	3	3	3
b	5	1	1	1	1	1
c	∞	1	1	1	1	1
e	∞	4	0	0	0	0
t	∞	∞	8	8	8	8

最短路径采用回溯方式获得：对于每个顶点 v，$\pi[v]$ 记录了最短路径上 v 的前驱顶点，比如：$\pi[t] = c$, $\pi[c] = a$, $\pi[a] = s$，$\pi[e] = b$, $\pi[b] = a$。这样逐级回溯直至源顶点 s，即可得到完整的 s-v 最短路径（见图 3.29中阴影边）。

由于图 G_1 中不存在 s 可达的负权重环，Bellman-Ford 算法运行至第 $n - 1$

轮即可得到 s 到每个顶点 v 的最短距离，即：$d(v) = d(v, n-1)$。假如我们再多运行一轮，计算出 $d(v, n)$，我们会发现 $d(v, n)$ 和 $d(v, n-1)$ 完全一致。

表 3.7: 对含负权重环的图 G_2 计算出的 $d(v, k)$，其中第 6 列和第 5 列有差异

	路径边数上限 k					
目的顶点v	1	2	3	4	5	6
s	0	0	0	0	0	0
a	3	3	3	2	2	2
b	5	1	1	1	0	0
c	∞	1	1	1	0	0
e	∞	2	-2	-2	-2	-3
t	∞	∞	8	7	7	7

而对存在 s 可达的负权重环的图 G_2 来说，必定存在一些顶点 v（比如环中顶点 a），使得 $d(v, n)$ 比 $d(v, n-1)$ 小。因此，我们可以依据$d(v, n)$ 与 $d(v, n-1)$ 是否有差异来判别是否存在 s 可达的负权重环。

这个判别规则的证明也很简单：假如 $d(v, n)$ 比 $d(v, n-1)$ 更小，则必定是因为发现了一条 s-v 的“恰好”包含 n 条边的更短路径 p；由于路径 p 中有 n 条边，则必定存在环；从 p 中去除环，得到新的路径 p'，而 p' 的长度至少是 $d(v, n-1)$，比 $d(v, n)$ 要大，因此此环必定是负权重环。

3.11.3 时间复杂度分析

Bellman-Ford 算法共需执行 n 轮；在每一轮，都需要依据每一条边更新 $d(v, k)$；因此时间复杂度是 $O(mn)$。

3.11.4 一些讨论

“粗”的子问题表示 vs 更“细”的子问题表示

在上一小节中，我们以图 3.29(a) 所示的环 $a \to b \to a$ 为例，说明了 $d(v)$ 之间可能存在循环依赖性；如果写成程序，在运行时就会表现为“死循环”。一个有意思的问题是：采用更“细”的子问题定义 $d(v, k)$，是如何绕过这个困难的呢？

原来采用更细的子问题之后，我们能够得到如下递归关系：

$$d(a,2) = \min\{d(a,1), 3 + d(b,1), 4 + d(e,1)\},$$

$$d(b,2) = \min\{d(b,1), -2 + d(a,1), 1 + d(c,1)\}.$$

换句话说，虽然我们无法建立型如“$d(u)$ 和 $d(v)$ 之间的递归关系”，但是通过引入“路径中边数 k”，使得我们对于每一个顶点 v，都定义了 $n-1$ 个变量 $d(v,k)$，$k = 1, 2, \cdots, n-1$，并能够建立型如“$d(v,k)$ 与 $d(u,k-1)$ 之间的递归关系”。

最后我们再来看看“粗”的子问题的解与更“细”的子问题的解之间的关系：

$$d(a) = \min\{d(a,1), d(a,2), \cdots, d(a,n-1)\},$$

$$d(b) = \min\{d(b,1), d(b,2), \cdots, d(b,n-1)\}.$$

虽然 $d(a,k)$，$d(b,k)$ 之间存在递归关系，但是“取最小”这个操作导致了 $d(a)$ 和 $d(b)$ 是更“粗”的问题，出现了循环依赖性。

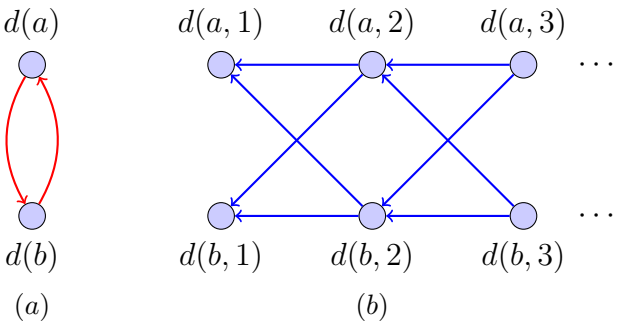


图 3.30: 单源最短路问题中两种子问题定义。(a) “粗”的子问题之间存在循环依赖性，比如计算 $d(a)$ 需要事先知道 $d(b)$ 的值，而计算 $d(b)$ 又得事先已知 $d(a)$ 的值；(b) 与之相反，当采用“更细”的子问题定义时（增加了限制条件），这种循环依赖性变成了“交叉依赖”，从而被打破了

在 Bellman-Ford 算法中，运算步骤 $d[v] = \min\{d[v], d[u] + w(u,v)\}$ 被称作*松弛*（Relaxation），其目的是：我们用 $d[v]$ 来表示 s-v 最短距离的上界，并依据边 (u,v)，逐步降低上界。之所以称为“松弛”，是来源于描述最短路径的线性规划模型中的约束 [?]；我们在第 8 章中将会详细阐述。

从 s 到 t 的最短路径问题 vs 从 s 到每一个结点的最短路径问题

从表 3.6可以看出，即使给定了目的结点 t，我们还是不限于只计算 s 到 t 的最短路径，而是计算出从 s 到所有结点的最短路径。这是因为我们无法事先得知从哪个结点才能够最近到达 t，因此只好计算出到达所有结点的最短路径。事实上，迄今为

止求解这两个问题的算法在渐进意义下都一样快。

通过回溯，我们能够找到把从 s 到每一个结点的最短路径；如果把这些结点搜集在一起的话，会形成一颗不会包含圈的**最短路径树**（示例见图 3.29）。

其他顺序的多步决策过程

在 Bellman-Ford 算法设计中，多步决策过程的第一步决策考虑的是“从哪个结点到达的目的结点 t”。除此之外，还有其他顺序的多步决策过程：

(1) **从源节点 s 开始考虑**：第一步决策考虑“从源结点 s 往哪个结点走”，以逐步构建出以 s 为根结点的最短路径树；

(2) **从最短路径的“中间结点”开始考虑**：最短路径是否经过某个结点 v。

用这两种顺序构建多步决策过程也是可行的，会相应地得出不同的动态规划算法。

3.12 应用动态规划技术求解整数规划问题

很多优化问题可以表示成整数规划，即：寻找满足约束条件的一些整数，使得特定的目标函数的值最大。在本小节，我们考虑如下的整数规划问题：

$$\begin{array}{rl} \max & \sum_{i=1}^{n} f_i(x_i) \\ s.t. & \sum_{i=1}^{n} \phi_i(x_i) \ \leq \ C \\ & x_i \ \in \ \mathbf{Z} \quad (1 \leq i \leq n) \end{array}$$

其中 max 行表示待优化的目标函数；$s.t.$ 是 *“subject to”* 的缩写，表示对变量 x_i 的约束条件；$f_i(\cdot)$ 是单变量函数；$\phi_i(\cdot)$ 是单变量增函数，C 表示一个常数。

例如下述的非线性优化问题：

$$\begin{array}{rlllllll} \max & f_1(x_1) & + & f_2(x_2) & + & f_3(x_3) & & \\ s.t. & x_1^2 & + & 3x_2^2 & + & 2x_3^2 & \leq & 20 \\ & x_1, & & x_2, & & x_3 & \in & \mathbf{N} \end{array}$$

其中函数 $f_1(\cdot), f_2(\cdot), f_3(\cdot)$ 的解析表达式未知，故以列表形式表示如下：

x	0	1	2	3	4	5	6	7	8	9	10
$f_1(x)$	2	4	7	11	13	15	18	22	18	15	11
$f_2(x)$	5	10	15	20	24	18	12	9	5	3	1
$f_3(x)$	8	12	17	22	19	16	14	11	9	7	4

上述整数规划的难点在于：i) 目标函数和约束条件都是非线性的；ii) 即使我们采用拉格朗日乘子法将这个约束优化问题转变成无约束优化问题，但是目标函数解析表达式未知，使得我们无法对其求导；iii) 更困难的是，这些变量有整数的限制，使得经典优化方法难以应用。

但是上述整数规划也有一个显著的特点：目标函数和约束条件都是**可分的** [?]。详细地说，虽然目标函数是一个 3 变量的函数，但是可以写成 3 个单变量函数的加和；此外，约束条件是 3 个增函数之和。这个特点使得我们可以把上述优化问题改写成如下的**递归形式**：

$$\max_{2x_3^2\le 20}\Big\{\max_{3x_2^2\le 20-2x_3^2}\Big[\max_{x_1^2\le 20-2x_3^2-3x_2^2}\big(f_1(x_1)\big)+f_2(x_2)\Big]+f_3(x_3)\Big\}.$$

采用这种递归形式定义的问题可以很自然地定义子问题，从而便于设计动态规划算法进行求解。

3.12.1 算法设计与描述

为定义子问题，我们先来研究两种特殊情况：

(1) 假如我们已知最优解中部分解 (x_2, x_3) 的取值，记为 $(\widehat{x_2}, \widehat{x_3})$，则原始优化问题等价为如下的**单变量优化问题**：

$$\max_{x_1^2\le 20-2\widehat{x_3}^2-3\widehat{x_2}^2}\big(f_1(x_1)\big).$$

然而最优解 $(\widehat{x_2}, \widehat{x_3})$ 是未知的，换句话说，$20-2{x_3}^2-3{x_2}^2$ 的取值不是固定的；我们记 $\lambda_1 = 20-2{x_3}^2-3{x_2}^2$，并将原始优化问题进一步改写为：

$$g_1(\lambda_1) \triangleq \max_{x_1^2\le\lambda_1}\big(f_1(x_1)\big). \tag{3.12.1}$$

此处的 λ_1 称为与部分解 (x_2, x_3) 对应的**状态变量**，用于刻画下一步对变量 x_1 做决策时的约束条件 $x_1^2 \le \lambda_1$。由 $x_2, x_3 \ge 0$ 可知 λ_1 的取值范围是：$0 \le \lambda_1 \le 20, \lambda_1 \in \mathbf{N}$。

上式意味着当 λ_1 取任一合理的值时，原始优化问题变成一个单变量优化问题（变量是 x_1），其最优值记为 $g_1(\lambda_1)$。我们可以采用枚举技术求解这个单变量优化问题。

(2) 接下来，假如我们已知最优解中部分解 (x_3) 的取值，记为 $(\widehat{x_3})$，则原始优化问题等价为如下的 2 **变量优化问题**：

$$\max_{3x_2^2\le 20-2\widehat{x_3}^2}\Big[\max_{x_1^2\le 20-2\widehat{x_3}^2-3x_2^2}\big(f_1(x_1)\big)+f_2(x_2)\Big],$$

然而最优解 $(\widehat{x_3})$ 事实上是未知的，换句话说，$20-2{x_3}^2$ 的取值不是固定的；我们记 $\lambda_2 = 20-2{x_3}^2$，并将原始优化问题进一步改写为：

$$g_2(\lambda_2) \triangleq \max_{3x_2^2 \le \lambda_2} \Big[\max_{x_1^2 \le \lambda_2 - 3x_2^2} \big(f_1(x_1)\big) + f_2(x_2) \Big]. \tag{3.12.2}$$

此处的 λ_2 称为与部分解 (x_3) 对应的**状态变量**，用来刻画下一步对变量 x_2 做决策时的约束条件 $3x_2^2 \le \lambda_2$。由 $x_3 \ge 0$ 可知 λ_2 的取值范围是：$0 \le \lambda_2 \le 20, \lambda_2 \in \mathbf{N}$。

注意到我们已经定义了函数 $g_1(\lambda_1) = \max_{x_1^2 \le \lambda_1} \big(f_1(x_1)\big)$，因此我们可以进一步将原始优化问题改写成下式：

$$g_2(\lambda_2) \triangleq \max_{3x_2^2 \le \lambda_2} \big[g_1(\lambda_2 - 3x_2^2) + f_2(x_2) \big]. \tag{3.12.3}$$

上式意味着当 λ_2 取任一合理的值时，原始优化问题变成一个**单变量优化问题**：变量是 x_2，目标函数是由 $f_2(\cdot)$ 和新定义的函数 $g_2(\cdot)$ 构成的复合函数。我们可以采用枚举技术求解这个单变量优化问题。

子问题定义及递归关系

上述分析表明我们可以把求解过程描述成一个**多步决策过程**：从 x_3 到 x_2 再到 x_1，每一步决策确定一个变量的取值；各决策的收益分别定义为 $f_1(x_1), f_2(x_2), f_3(x_3)$，这样原始优化问题的目标函数就是所有步骤决策的收益总和。

基于这个多步决策过程，我们可以定义如下两个子问题：

(1) 当已知部分解 (x_2, x_3) 时，计算 x_1 的最优决策，即问题 3.12.1。

(2) 当已知部分解 (x_3) 时，计算 x_2 的最优决策，即问题 3.12.2；

上述子问题之间存在着递归关系，即公式 3.12.3；原始优化问题可以用上述子问题表示如下：

$$\begin{aligned} & \max_{2x_3^2 \le 20} \Big\{ \max_{3x_2^2 \le 20 - 2x_3^2} \Big[\max_{x_1^2 \le 20 - 2x_3^2 - 3x_2^2} \big(f_1(x_1)\big) + f_2(x_2) \Big] + f_3(x_3) \Big\} \\ = & \max_{2x_3^2 \le 20} \big\{ g_2(20 - 2{x_3}^2) + f_3(x_3) \big\}. \end{aligned}$$

依据上述分析，我们可以设计如下的动态规划算法：

3.12.2 算法运行过程示例

以优化问题 3.12为例，算法首先计算出 $g_1(\lambda_1)$ 和 $\widehat{x_1}(\lambda_1)$：

Algorithm 35 求解整数规划的动态规划算法

function DP-FOR-IP(n)

1: **for** $k = 1$ to $n-1$ **do**
2: **for** $\lambda_k = 0$ to C **do**
3: Calculate $g_k(\lambda_k) = \max\limits_{\phi_k(x_k)\le\lambda_k}\{g_{k-1}\big(\lambda_k - \phi_k(x_k)\big) + f_k(x_k)\}$;
4: Set $\widehat{x_k}(\lambda_k) = \operatorname{argmax}\limits_{\phi_k(x_k)\le\lambda_k}\{g_{k-1}\big(\lambda_k - \phi_k(x_k)\big) + f_k(x_k)\}$;
5: **end for**
6: **end for**
7: Set $\widehat{x_n} = \operatorname{argmax}\limits_{\phi_n(x_n)\le C}\{g_{n-1}\big(C - \phi_n(x_n)\big) + f_n(x_n)\}$;
8: **return** $\max\limits_{\phi_n(x_n)\le C}\{g_{n-1}\big(C - \phi_n(x_n)\big) + f_n(x_n)\}$;

λ_1	0	1	2	3	4	5	6	7	8	9	10
g_1	2	4	4	4	7	7	7	7	7	11	11
$\widehat{x_1}$	0	1	1	1	2	2	2	2	2	3	3
λ_1	11	12	13	14	15	16	17	18	19	20	
g_1	11	11	11	11	11	13	13	13	13	13	
$\widehat{x_1}$	3	3	3	3	3	4	4	4	4	4	

然后计算 $g_2(\lambda_2)$ 和 $\widehat{x_2}(\lambda_2)$：

λ_2	0	1	2	3	4	5	6	7	8	9	10
g_2	7	9	9	12	14	14	14	17	17	17	17
$\widehat{x_2}$	0	0	0	1	1	1	1	1	1	1	1
λ_2	11	12	13	14	15	16	17	18	19	20	
g_2	17	21	21	21	21	22	22	22	23	23	
$\widehat{x_2}$	1	1	1	1	1	2	2	2	1	1	

最后我们计算出原始问题的最优值为：$\max\limits_{2x_3^2\le 20}\big\{g_2(20-2{x_3}^2)+f_3(x_3)\big\} = 38$，且 $\widehat{x_3} = 2$。x_2 和 x_1 的最优解可以通过回溯计算出来：由 $\widehat{x_3} = 2$ 可知 $\lambda_2 = 20 - 3{\widehat{x_3}}^2 = 12$，经查表格 $\widehat{x_2}(\lambda_2)$ 可知，$\widehat{x_2} = 1$；进一步地，我们可以计算出 $\lambda_1 = 20 - 3{\widehat{x_3}}^2 - 2{\widehat{x_2}}^2 = 10$，经查表格 $\widehat{x_1}(\lambda_1)$ 可知 $\widehat{x_1} = 3$。因此原始问题的最优解是 $(\widehat{x_1}, \widehat{x_2}, \widehat{x_3}) = (3, 1, 2)$。

3.12.3 时间复杂度分析

由于每个表格 $g_k(\lambda_k)$ 共有 C 个单元，且计算每个单元时，需要枚举满足 $\phi_k(x_k) \le \lambda_k$ 的所有可能的 x_k。令 $m = \max_k |\{x_k|\phi_k(x_k) \le \lambda_k\}|$，则算法的时间复杂度为

$O(nCm)$。

3.12.4 一些讨论

从上述例子中，我们能够看到使用动态规划求解整数规划问题的特点：

(1) 应用动态规划，我们可以把一个 n 个变量的优化问题，转变成 n 个单变量优化问题；而且这 n 个单变量优化问题可以递归进行求解。

(2) 动态规划技术可以确保得到全局最优解，从而避免了其他优化技术局部极值的困扰。

(3) 对于优化问题来说，约束条件往往会带来很大的麻烦。在经典优化技术中，我们可以采用拉格朗日乘子法去掉等式约束，采用 KKT 条件去掉不等式约束；然后这些技术却难以处理要求变量取整数值的约束。与之恰恰相反，这些整数约束条件反而能够简化动态规划的计算。换句话说，一些类型的约束条件对于经典优化技术会造成困难，但是却能够大大帮助动态规划算法。

本节所示的优化问题只有一个约束条件，且目标函数是单变量函数之和；事实上，动态规划技术可以用于解决更复杂的问题：

(1) 多个约束条件：当优化问题有多个约束条件时，我们可以相应地定义多维状态变量，即可设计出动态规划算法。由于子问题数目和维数成指数关系，因此当维数过多时，会造成时间复杂度急剧上升；这被称为“维数灾难”（Dimension curse）[?, ?]。为应对维数灾难，研究者提出了一系列方法，比如强化学习，不依赖于“倒推”，直接“时间向前地”求解控制问题，从而适用于状态方程和性能指标未知的控制问题中 [?]；再如近似动态规划，采用神经网络等技术近似求解 Bellman 方程，也能够达到“时间向前地”获得近似最优控制 [?]。

(2) 含相关项的目标函数：有些优化问题的目标函数具有如下形式：

$$\max\Big\{\sum\nolimits_{i=1}^{n} f_i(x_i) + \sum\nolimits_{i=2}^{n} s_i(x_{i-1}, x_i)\Big\},$$

其中 $s(x_{i-1}, x_i)$ 称为相关项。当设置 $s(x_{i-1}, x_i) = (x_i - x_{i-1})^2$ 时，可以用于避免相邻变量之间差异过大，从而起到“平滑”的作用；在 HMM 解码问题中，我们设置 $s(x_{i-1}, x_i) = b_{x_{i-1}x_i}$，以表示相邻时刻隐含状态之间的依赖关系。

对于含相关项的目标函数，我们可以设置状态变量 $\lambda_k = x_{k-1}$，然后即可仿照本小节的思路设计动态规划算法。

(3) **极大极小型目标函数**：有些优化问题的目标函数不是单变量函数之和，而是单变量函数取最大：

$$\min_{x_1,x_2,\cdots,x_n} \Big\{ \max\Big(f_1(x_1), f_2(x_2), \cdots, f_n(x_n)\Big)\Big\}.$$

对于这种目标函数，我们只需将求和变换成取最大，其他部分扔沿用本小节所述思路即可设计求解算法。类似地，当目标函数是多个单变量函数的乘积时（假设各个单变量函数值都大于 0），也可以设计动态规划算法进行求解。

3.13 动态规划与马尔可夫决策过程

马尔可夫决策过程（Markov decision process）是对马尔可夫过程的扩展（增加了**状态转移的收益**和能够**影响状态转移的动作**）。在介绍其形式化定义之前，我们先来看一个实际问题：

考虑一位出租车司机，其所在的城市有两个区，分别是工业区 A 和风景区 B；因此，每次载客共有 4 种可能的行程：$A \to A$, $A \to B$, $B \to A$ 以及 $B \to B$；出租车司机依据经验，总结出两种典型的候客模式：

(1) **巡游揽客优先模式** m_1：优先采用巡游兜揽乘客方式；当兜揽不到乘客时，才去工厂或风景区门口候客；

(2) **定点候客优先模式** m_2：优先采用在工厂或风景区门口候客方式；当等不到乘客时，才去巡游兜揽乘客。

我们将出租车当前所在的区称为**状态**。出租车司机的历史数据表明：工业区和风景区人群属性存在着差异，导致两种候客模式下的状态转移概率、每次行程的期望收入也迥然不同；详细统计数据见下表：

表 3.8: 出租车司机不同候客模式下状态转移概率及每个行程的期望收入

状态 i	模式 k	转移概率 $P_k(i,j)$		净收入 $R_k(i,j)$		每行程期望收入
		$j=A$	B	$j=A$	B	$\sum_j P_k(i,j)R_k(i,j)$
A	1	0.6	0.4	10	50	26
	2	0.8	0.2	25	60	32
B	1	0.4	0.6	55	25	37
	2	0.2	0.8	60	30	36

出租车司机面临的问题是：假如一天按平均运送 5 位乘客计，每次候客采用哪种模式，才能使得总收入最大?

上述问题是一个典型的马尔可夫决策过程最优策略问题；马尔可夫决策过程可形式化定义如下：

定义 3.13.1 (马尔可夫决策过程). 马尔可夫决策过程由四元组 (S, A, P_a, R_a) 构成，其中 S 表示状态集合；A 表示动作集合；$P_a(s, s') = \Pr(s_{t+1} = s' | s_t = s, a_t = a)$ 表示在时刻 t、动态系统的状态是 s 时，如果执行动作 a，则下一个时刻状态变换成 s' 的概率；$R_a(s, s')$ 表示上述状态变换的即时收益。

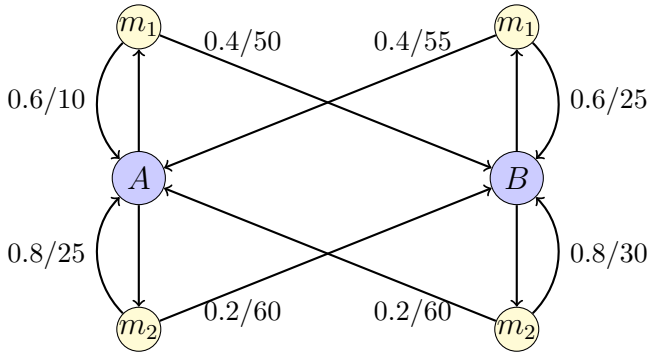


图 3.31: 与出租车司机最优候客模式问题对应的马尔可夫决策过程。图中大圆圈表示状态，小圆圈表示动作，边上的数字分别表示状态转移概率与平均净收入

在出租车司机最优候客模式问题中，对候客模式的选择对应于马尔可夫决策过程中的动作（见图 3.31）；所有的动作决策合在一起，称为策略（Policy）。详细地说，制定一个策略，就是制定一系列的 `if...then...` 型的规则，每条规则说明在一种情形下、该采取何种动作；这些规则要覆盖所有的可能情形。就出租车候客模式问题为例，策略共有 n 条规则，每条规则都具有如下形式：

规则 i：考虑第 i 个行程，如果当前处于工业区 A，则采取候客模式 $a_i(A)$；如果当前位于风景区 B，则采取候客模式 $a_i(B)$。

马尔可夫决策过程最优策略问题描述如下：

马尔可夫决策过程最优策略问题

输入：马尔可夫决策过程 (S, A, P_a, R_a)，初始状态 s_0，正整数 n；

输出：最优策略，即：n 次动作决策 $a_0(s_0), a_1(s_1), \cdots, a_{n-1}(s_{n-1})$，使得期望累计收益 $\mathbb{E}[\sum_{i=0}^{n-1} R_{a_i}(s_i, s_{i+1})]$ 最大，其中 $P_{a_i}(s_i, s_{i+1})$ 表示状态转移概率 $\Pr(s_{i+1}|s_i, a_i)$，$a_i(s_i)$ 表示在第 i 个时刻、依赖于状态 s_i 所采取的动作 a_i。

值得指出的是，当指定每个时刻的动作之后，马尔可夫决策过程就退化成一个带收益的马尔可夫过程。为简单起见，我们假设这个马尔可夫过程是**完全各态历经的**（Ergodic），即：在多次状态转移之后，极限状态概率分布与初始状态分布无关 [?]。

3.13.1 算法设计与描述

马尔可夫决策问题的最优策略求解是一个**多步决策过程**：在每一步决策时，需要确定最优动作。以第一步决策为例，如果我们选择了动作 $a_0(s_0)$，且按照概率 $P_{a_0}(s_0, s_1)$ 进行状态转移，得到新状态 s_1，则剩下的子问题变为：如何选择动作序列 $a_1(s_1), \cdots, a_{n-1}(s_{n-1})$，使得后续 k 步决策的期望累计收益 $\mathbb{E}[\sum_{i=1}^{n-1} R_{a_i}(s_i, s_{i+1})]$ 最大。

因此，我们可以将子问题的一般形式定义最后 k 步的最优决策问题，即：当状态 s_{n-k} 已知时，如何选择动作序列 $a_{n-k}(s_{n-k}), \cdots, a_{n-1}(s_{n-1})$，使得后续步骤的期望累计收益 $\mathbb{E}[\sum_{i=n-k}^{n-1} R_{a_i}(s_i, s_{i+1})]$ 最大。这个后续步骤期望累计收益的最大值依赖于决策步数 k 和起始状态 s，称作**值函数**，记为 $V_k(s) = \max_{a_{n-k}, \cdots, a_{n-1}} \mathbb{E}[\sum_{i=n-k}^{n-1} R_{a_i}(s_i, s_{i+1})]$。值函数有如下递归关系及基始情况赋值：

$$V_k(s) = \begin{cases} 0 & \text{如果} k = 0 \\ \max\limits_{a_k \in A}\{\sum\limits_{s' \in S} P_{a_k}(s, s')[R_{a_k}(s, s') + V_{k-1}(s')]\} & \text{否则} \end{cases}$$

原始问题的最优解可以表示为 $V_n(s_0)$。基于上述递归关系，我们可以设计出求解 $V_n(s_0)$ 的动态规划算法（称为**值迭代算法** [?, ?]），其伪代码从略。

3.13.2 运行过程示例

采用值迭代算法求解出租车司机最优候客方式问题的过程见表 3.9，其中决策步数 $n = 5$。从表中我们可以获得如下观察：

(1) 如果出租车司机从 A 区开始一天的营运的话，全天收入 165.82 元；从 B 区开始一天的营运的话，则全天收入 175.18 元。

(2) 如果只营运一个行程的话，最优候客模式是在工业区 A 采用“定点候客优先模式”，在旅游区 B 则采用“巡游揽客优先模式”；但是随着行程数目的增加，最优候客模式变为不论在 A 区还是 B 区，都采用“定点候客优先模式”。

(3) 如图 3.32所示，$V_k(A)$ 和 $V_k(B)$ 的渐近线都是直线且平行，其斜率为 34.22，表示当行程数目足够大时，每个行程的期望收入。

表 3.9: 求解出租车司机最优候客方式问题的值迭代算法运行过程示例

	决策步数 k					
值函数及最优动作	0	1	2	3	4	5
$V_k(A)$	0	32	65	98.40	132.04	165.82
$V_k(B)$	0	37	72	106.60	140.96	175.18
$a_k(A)$	-	2	2	2	2	2
$a_k(B)$	-	1	1	2	2	2

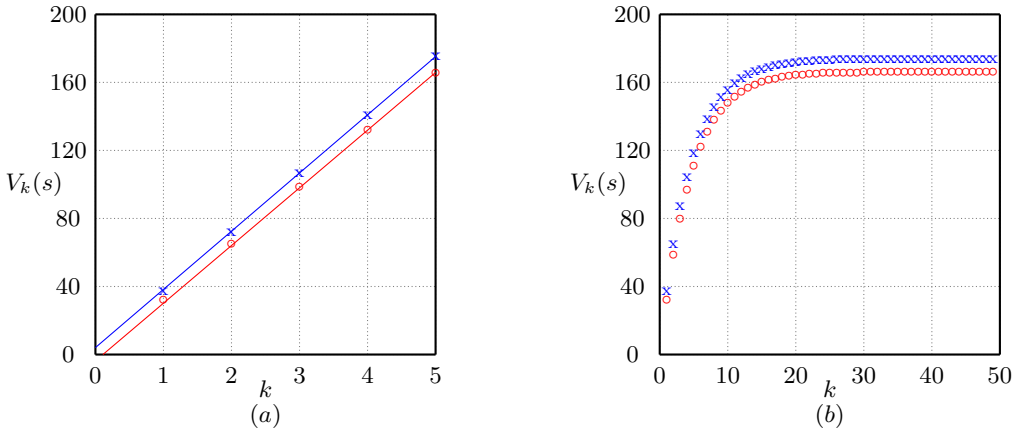


图 3.32: 值函数 $V_k(s)$ 的渐近行为。图中红色“○”号表示 $V_k(A)$，蓝色“x”号表示 $V_k(B)$。(a) 无折扣因子的情形：两条渐近线有相同的斜率约为 $g = 33.78$，其截距分别为 $v_A = -3.18, v_B = 4.08$。(b) 有折扣因子的情形：当折扣因子 $\gamma = 0.80$ 时，$V_k(A)$ 趋近于 166.15，$V_k(B)$ 趋近于 173.85

3.13.3 一些讨论

策略迭代算法

值迭代法要求事先已知决策步数，从而导致应用时存在一些严重的限制：*i*) 决策步数通常是未知的：以出租车司机为例，每天的行程总数事先无法准确预知，从而导致值迭代法无法适用。*ii*) 对于无限期或者决策步数足够多的过程（比如出租车司机

一年的营运），值迭代法也是难以直接应用的；即便我们可以考虑值函数 $V_k(s)$ 的渐进行为，但是往往难以判断值函数何时收敛。

为克服值迭代法的缺陷，R. A. Howard 于 1960 年提出了**策略迭代法**（Policy iteration algorithm）[?]。在介绍策略迭代法之前，我们首先来介绍一下最优策略和值函数的两个性质：

(1) **最优策略**：R. Bellman 证明值迭代法可以收敛至最优策略 [?]。对于无限期或者决策步数足够多的过程来说，最优策略与决策步数 k 无关，因此可以表示成对于每个状态 s 指定一个动作 $a(s)$，而不再使用带决策步数的记号 $a_k(s)$。换句话说，策略中的 `if...then...` 型规则不用再考虑当前是第几个行程，因此可以把 n 条规则简化成一条规则，即：**如果当前位于工业区 A，则采用候客模式 $a(A)$；如果当前位于风景区 B，则采用候客模式 $a(B)$**。

(2) **值函数**：R. A. Howard 证明了对任一策略来说，当决策步数 k 足够大时，总期望收益 $V_k(s)$ 与 k 呈线性关系（见图 3.32），即：

$$V_k(s) = g \cdot k + v_s.$$

这意味着只需计算出斜率 g 和偏移量 v_s，即可完整刻画期望累计收益。

策略迭代法从一个初始策略开始，迭代执行**策略评估**和**策略改进**两个步骤，不断提高期望累计收益，直至收敛；这两个基本步骤可描述如下：

(1) **策略评估**：对于一个给定策略来说，我们可以评估它的优劣，即：按此策略决策 k 步，可以获得的期望累计收益的最大值。

为此目的，我们将值函数的线性表达式代入 $V_k(s) = \sum_{s' \in S} P_a(s, s')[R_a(s, s') + V_{k-1}(s')]$，可得：

$$g + v_s = \sum_{s' \in S} P_a(s, s') R_a(s, s') + \sum_{s' \in S} P_a(s, s') v_{s'}.$$

我们将所有状态 s 的线性方程联立起来，构成一个线性方程组；求解此线性方程组，即可解出斜率 g 和偏移量 v_s，进而算出按此策略做 k 步决策的总收益 $V_k(s) = g \cdot k + v_s$。

(2) **策略改进**：假如对于状态 s 而言，当前策略不是最优的，则选择新的动作 $a(s)$，以增加收益，即：

$$a(s) = \underset{a \in A}{\operatorname{argmax}} \{ \sum_{s' \in S} P_a(s, s') R_a(s, s') + \sum_{s' \in S} P_a(s, s') v_{s'} \}.$$

策略迭代法被广泛用于强化学习 [?] 和近似动态规划中 [?]。

具有折扣因子的马尔可夫决策过程

对描述经济活动的马尔可夫决策过程来说，折扣因子的引入是非常自然的：例如出租车司机将每个行程的收入存入银行，假设利率是 β，则在当前时刻看来，上一个行程的收入需乘以 $(1+\beta)$，以表示收入与利息的总和；这等价于本次行程的收入乘以一个折扣因子 $\gamma = \frac{1}{1+\beta}$。

值得强调的是，即使在不涉及经济活动的马尔可夫决策过程中，引入折扣因子也有着重要作用：i) 可以方便地表示顺序信息，比如下棋时棋局的先后顺序；ii) 可以表示长期过程发生的概率相对较小；iii) 有时则仅仅是为了使得无限期过程的总收益能够收敛 [?]。

引入折扣因子 γ，只需对值函数的递归表达式进行小幅修改，即：

$$V_k(s) = \max_{a \in A} \sum_{s' \in S} P_a(s, s')[R_a(s, s') + \gamma \cdot V_{k-1}(s')].$$

表 3.10: 带折扣因子时值函数渐进行为示例

	决策步数 k					
值函数	0	10	20	30	40	50
$V_k(A)$	0	147.99	164.20	165.94	166.13	166.15
$V_k(B)$	0	155.67	171.90	173.64	173.82	173.84

折扣因子最显著的影响是改变了值函数的渐进行为：当 $0 \leq \gamma < 1$ 时，随着决策步数 k 的增大，期望累计收益的最大值 $V_k(s)$ 不再收敛至一条斜率为 g 的直线，而是收敛至一个固定值。以出租车司机候客策略问题为例，当设置折扣因子 $\gamma = 0.80$ 时，计算出的值函数 $V_k(s)$ 见表 3.10。从表中可以看出，随着决策步数 k 的增大，无论是 $V_k(A)$ 还是 $V_k(B)$，都趋近于一个固定值（见图 3.32）。

延伸阅读

动态规划算法的提出过程

动态规划算法是 Richard Bellman 于 1952 年提出的 [?]。1949 年左右，R. Bellman 进入兰德公司（RAND Corporation）工作；当时兰德公司主要研究和空军相关的最优控制问题。J. von Neumann 经常访问兰德公司，从事博弈论的应用研究；他在两个方面深刻影响了兰德公司以及 Bellman 的研究：*i*) 应用博弈论解决实际问题时，常将问题形式化成多步决策过程；这也是 Bellman 研究多步决策过程的动机之一 [?]；*ii*) 动态规划中的求最优解的“回溯法”，或多或少能够看出博弈论中“倒推法”的影子 [?]。

当时兰德公司关注的一个多步决策问题是**导弹分配问题**：设有 n 驾敌机来袭，每驾敌机的价值各有不同；我方有 m 枚导弹（$m > n$），已知使用 k 枚导弹攻击第 i 驾敌机的毁伤概率是 $p_i(k)$。问：每驾敌机各使用几枚导弹攻击，能够使得毁伤价值的期望值最大?

这是一个优化问题。那在 1949 年之前已有的数学工具能否解决这个优化问题呢?

在那个时期，流行的优化方法是经典变分法：Euler 于 1744 年、Lagrange 于 1755 年提出的 Euler-Lagrange 方程是求解最优控制问题的经典方法 [?]。然而经典变分法难以直接应用于导弹分配问题：导弹的数目有整数限制；毁伤概率 $p_i(k)$ 不可微。

R. Bellman 发现，对于导弹分配问题以及与之类似的一大类多步决策问题来说，采用“最优性原理”能够很方便地建立起泛函方程，从而有利于求出最优解 [?]；进而，R. Bellman 又提出了一个新的理论—**马尔可夫决策过程** [?]。

顺带说一下 R. Bellman 选用 “Dynamic programming” 一词的考虑：“Dynamic”指的是研究对象（物理系统或抽象的概念性系统）的状态在不断变化，而“Programming”指的是“规划”—在发明电子计算机之后，“Programming” 常用于指代“编程序”（Coding），但其原意是“列表”（Tabular）和“规划”，从而很适合表达用表格存储中间结果的求解过程；这和 G. Dantzig 采用 “Linear programming” 一词的初衷很相似。R. Bellman 没有选用数学味儿更浓厚的其他名词，另一个动机是避免可能引发的争议。

动态规划算法与最优控制之间的联系

求解最优控制的方法有三大类：L. Euler 和 J. L. Lagrange 提出的变分法、R. Bellman 提出的动态规划算法，以及 Pontryagin 极小值原理 [?]。

在一些问题中，经典变分法暴露出一些缺陷：

(1) 经典变分法要求目标泛函的定义域是一个开集，而当变分问题中存在约束条件时，定义域往往不再是开集，即：对函数 $x(t)$ 的小的扰动 $x(t)+\alpha\delta x(t)$ 可能落在定义域之外。

(2) 经典变分法给出了最优函数需要满足的必要条件，即 Euler-Lagrange 方程；由于 Euler-Lagrange 方程是二阶（或更高阶）微分方程，因此需要两个（或多个）边界条件方可确定唯一解；然而微分方程的两点边值问题往往不易求解 [**?**]。

(3) 经典变分法要求待求的函数是连续可微的，然而在实际问题中，这个要求往往难以满足。在 1900 年，D. Hilbert 提出了“23 个数学问题”，其中的第 23 题就是“变分法的长远发展”。

动态规划和另外两类方法有着显著的差别：

(1) 无论是变分法还是极小值原理，都是寻找最优函数需要满足的必要条件；而在求解满足条件的最优函数时，最优函数是作为一个整体一次性求出来的（或者用一系列函数逐渐逼近最优函数）；或者反过来说，最优函数不是逐点计算出来的。

(2) 采用动态规划时，不是一次性求出整个函数，而是**逐点计算**的：每次都是从上一个点，计算下一个点。在离散时间问题中，这种逐点计算就是多步决策过程；而在连续时间问题中，则引入**值函数**这个概念进行逐点计算。

尽管有这些差别，但是这些方法是有着密切联系的：*i*) L. Euler 最初解决最简变分问题的“几何法”，其基本思想**将区间离散化、用折线代替曲线**，本质上是**逐点计算** [**?**]；*ii*) 采用动态规划技术，可以推导出 Euler-Lagrange 方程 [**?**]。

1958 年，R. Bellman 建立起动态规划和最优控制之间的联系 [**?**]：在最优控制中，待求的函数不仅是时间的函数，还依赖于决策以及待研究系统当前的**状态**；这类函数描述“**在哪样的状态下，应该采用何种动作才能使得目标泛函最大化**”，因此被称为策略。Bellman 发现采用“状态”、“决策”以及“策略”这些概念，可以很容易将最优控制问题描述成多步决策过程，进而设计出动态规划求解算法。为此，Bellman 感慨道：（**他或许）应该更早一些意识到能够应用动态规划解决控制论中的问题**；他觉得现在再讲起控制论的话，最好先开门见山地说“**控制过程可以被视为一个多步决策过程**” [**?**, **?**]。

子问题的常见形式

恰当地定义子问题是设计动态规划算法的核心步骤，需要创造性以及反复的试验。S. Dasgupta 等总结了几种常见的子问题形式 [?]；我们在其基础上做了一些扩展，陈述如下：

(1) **原始问题的输入部分是一个字符串（或序列）** $x_1x_2\cdots x_n$：我们可以定义子问题为字符串的前缀 $x_1\cdots x_i$ 或者 $x_i\cdots x_n$，这样子问题的数量是 $O(n)$；在 RNA 二级结构预测中，我们定义型如 $x_i\cdots x_j$ 的子问题，这样子问题的数量是 $O(n^2)$。

(2) **原始问题的输入部分是两个字符串** $x_1x_2\cdots x_n$ **和** $y_1y_2\cdots y_m$：我们可以定义子问题为两个字符串的前缀 $x_1\cdots x_i$ 和 $y_1\cdots y_j$（或者后缀 $x_i\cdots x_n$ 和 $y_j\cdots y_m$），这样子问题的数量是 $O(n^2)$。

(3) **原始问题的输入部分是有向无环图**：我们可以将有向无环图中顶点进行拓扑排序，从而将顶点转换成序列，然后参照序列的方式定义子问题。

(4) **原始问题的输入部分是一棵树**：我们可以定义子问题为其子树；在树上的最大独立集问题中，我们定义以孙子顶点为根的子树为子问题。

(5) **原始问题的输入部分是一个集合**：我们可以定义子问题为子集，不过这会造成指数多个子问题；在 0-1 背包问题中，我们事先为集合中的元素规定一个顺序，从而把集合转换成序列，然后参照序列的方式定义子问题。

矩阵链式乘法与多边形三角剖分

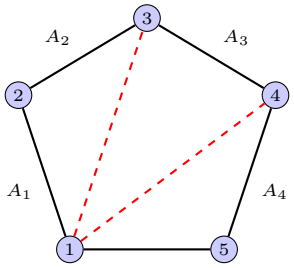


图 3.33: 凸多边形的三角剖分示例

1981 年，Hu 和 Shing 证明了对矩阵的链式乘法来说，可以构造一个凸多边形，其三角剖分方案与矩阵乘积顺序一一对应。以 3.1.3 节所示矩阵 A_1, A_2, A_3, A_4 为例：我们构造一个 5 个顶点的凸多边形（图 3.33），其中前 4 条边分别对应一个矩阵；图中展示的三角剖分方案对应着矩阵乘积顺序 $(((A_1A_2)A_3)A_4)$。

接下来，我们为前 4 条边的端点都赋予权重，分别表示与边相应矩阵的行数与列数；并定义三角形的权重为三个顶点的权重乘积。因此，计算最优的矩阵乘积顺序等价于计算最优的三角剖分，使得剖分之后形成的三角形权重总和最小。Hu 和 Shing 设计了求解最优三角剖分的 $O(n \log n)$ 的算法；由于最优的矩阵乘积顺序等价于计算最优的三角剖分，因此该算法也可以用于求解最优矩阵乘积顺序 [**?**]。

Hu 和 Shing 的原始算法比较复杂；19?? 年，F. Yao 对 Hu 和 Shing 的原始算法做了简化和改进，并提出了一个简单的 $O(n^2)$ 的算法 [**?**]。

字符串联配问题的发展历程

1970 年，Needleman 和 Wunsch 提出了字符串最优联配的动态规划算法 [**?**]，其时间复杂度是 $O(n^2)$；1974 年，Sellers 证明了字符串最优联配打分是一种距离函数。

此后，字符串联配问题在如下几个方面得到了长足的发展：

(1) 在空位罚分函数方面，如何设计联配中的 k 个连续占位符的打分是一个关键问题；罚分函数不仅决定了联配的合理性，也显著影响联配算法的速度。利用似然比技术，S. Eddy 等说明一个合理的罚分函数可以不考虑与占位符对应的字符，而只考虑占位符的个数即可 [**?**]。

经典的 Needleman-Wunsch 算法对联配中的 k 个连续占位符罚 δk 分，因此可视为采用线性罚分函数。1976 年，M. Waterman 等考虑了一般情况，证明了在任意的罚分函数之下，字符串最优联配打分依然是一种距离函数；由于必须要枚举各种可能长度的连续占位符，导致联配算法的时间复杂度变为 $O(n^3)$ [**?**]。1982 年，Gotoh 考虑了空位罚分函数的特殊情况：当采用仿射函数时，可以对第一个占位符以及其后的占位符给予不同的罚分，从而更具有实际应用价值；此时只需在每个递归步增加一种判断情况即可，因此联配算法的时间复杂度依然是 $O(n^2)$ 的 [**?**]。

1988 年，Webb Miller 和 Gene Myers 证明了当采用凹性空位罚分函数时，只需在每个递归步增加二分搜索步骤即可，因此联配算法的时间复杂度是

$O(n^2 \log n)$ [?]。1989 年，D. Eppstein 等利用稀疏性，提出一系列的加速策略 [?, ?]。1990 年，M. M. Klawe 等提出了 $O(n\alpha(n))$ 的算法，其中 $\alpha(n)$ 是增长非常缓慢的逆 Ackerman 函数 [?]。

上述空位罚分函数以及相应的联配算法的时间复杂度总结如下：

空位罚分函数	联配算法复杂度
线性函数	$O(n^2)$
仿射函数	$O(n^2)$
凹函数	$O(n^2 \log n)$
一般函数	$O(n^3)$

(2) 1981 年，T. Smith 和 M. Waterman 将全局联配算法推广到局部联配，可以发现两序列之间的高相似局部区域 [?]。

(3) 为进一步提高序列联配的速度，W. J. Wilbur 和 D. J. Lipman 于 1983 年提出了基于“k-mer 链接”的联配算法 [?]，即首先计算出两序列共有的 k-mer（无占位符），然后基于这些共有 k-mer 的最优链接构建联配。这种算法的时间复杂度为 $O(M^2)$，其中 M 表示共有 k-mer 数目；由于 M 远小于序列长度，因此这种算法的效率很高。

(4) 1975 年，D. Hirschberg 采用“以算代存”策略，不用回溯表格记录每一步决策的最优决策项，而是用“两个子问题之间的最优组合”算出最优决策项，继而采用分而治之策略求解子问题。分而治之算法的好处是：*递归调用可以重用空间*；采用这种策略，空间复杂度从 $O(mn)$ 降至 $O(m+n)$ [?]。1982 年，Jan Munro 发现了最优决策项之间的递归关系，并据此做了进一步改进，扩大了“以算代存”策略的适用范围，能够适用于大多数动态规划算法。

(5) 1984 年，Fredman 等将两序列联配算法推广到多序列联配 [?]。

Smith-Waterman 算法中阈值“0”的设置

在 Smith-Waterman 中，第 4 个选择项“0”是关键之一，也是和 Needleman-Wunsch 全局联配算法最重要的区别。选择项“0”意味着我们做了如下假设：如果

$Sim(\mathrm{x},\mathrm{y}) \leq 0$，则我们相信 x 和 y 是**不相似**的字符串。值得说明的是，我们还可以采用其他的阈值，详细说明如下。

现在广泛采用的相似度定义以及字符替换打分函数都是采用对数似然比计算出来的 [**?**]；而 Meng 等采用**假设检验**为打分函数设计建立了一个统一框架 [**?**]。以下述的无"插入"和"删除"情形的序列联配为例，

$$\mathrm{x}: \quad x_1 x_2 \cdots x_n$$

$$\mathrm{y}: \quad y_1 y_2 \cdots y_n$$

我们对这个联配做如下的假设检验：

H_0： $2n$个字符是独立同分布的，即：$P(x_i = a) = p_a, P(y_j = b) = p_b$;

H_1：x_i和y_i是相关的，即：$P(x_i = a, y_i = b) = q_{ab}$。

此处 $a, b \in A$ 表示两个字符，而 A 表示字符集。直观地看，原假设 H_0 是说两个字符串无关，而备择假设 H_1 是说两个字符串相关。依据 Neyman-Pearson 引理 [**?**]，似然比是**最大功效**检验统计量，即：

$$\prod_{i=1}^{n} \frac{q_{x_i y_i}}{p_{x_i} p_{y_i}}$$

基于对数似然比，我们构造如下的检验函数：

$$\Phi(\mathrm{x},\mathrm{y}) = I\Big(\sum_{i=1}^{n} \Big(\log \frac{q_{x_i y_i}}{p_{x_i} p_{y_i}} \Big) \geq t \Big)$$

令 $s(a,b) = \log\left(\frac{q_{x_i y_i}}{p_{x_i} p_{y_i}}\right)$，则可将上述检验函数表示为：

$$\Phi(\mathrm{x},\mathrm{y}) = I\Big(\sum_{i=1}^{n} s(x_i, y_i) \geq t \Big)$$

因此我们直接从大量单词中对字符出现进行计数，归一化后即可获得 $p_a, a \in A$；然后从已知联配数据中估计出 $q_{ab}, a, b \in A$；最终计算出字符替换打分函数 $s(a,b)$。

Smith-Waterman 算法中通常设置阈值 $t = 0$，意味着 x 和 y 是独立的和相关的概率相同；我们还可以设置其他的阈值 t，表示不同的**检验水平**。

习题

1. 在 1949 年左右，引发 R. Bellman 研究多步决策过程的是**导弹分配问题** [**?**]，描述如下：假设有 n 架敌机来袭，第 i 架敌机的价值是 v_i 元；我方有 m 颗导弹（$m > n$）。如果对飞机 i 发射 k 颗导弹的话，击落概率是 $p_i(k)$。请问：将 m 颗导弹分配给 n 架飞机的最优分配方案，使得击落飞机的期望价值总和最大。试求解如下实例：$m = 10$，$n = 3$，$v_1 = 1, v_2 = 2, v_3 = 5$，击落概率 $p_i(k)$ 见下表：

	对飞机 i 发射导弹的数目 k										
飞机编号 i	0	1	2	3	4	5	6	7	8	9	10
1	0	0.02	0.05	0.12	0.27	0.50	0.73	0.88	0.95	0.98	0.99
2	0	0.03	0.05	0.08	0.12	0.27	0.38	0.50	0.62	0.73	0.82
3	0	0.04	0.06	0.09	0.13	0.18	0.25	0.33	0.43	0.52	0.62

2. 考虑“再生资源问题” [?]：假设一位农夫租一块地 10 年种小麦；每一年都需要留出一部分做种子，其余全部出售。如果出售的话，价格是 2000 元/吨；如果用做种子的话，一粒种子第二年可期望收获 80 粒小麦（即：繁殖系数为 80）。设农夫初始时有 1 吨小麦，在第 10 年把累积的所有小麦全部卖出。请问：农夫每年需要留多少小麦做种子、卖出多少小麦，才能使得 10 年的总获利最大?

3. 考虑如下的整数“分割”问题 [?]：给定正整数 n，问如何分成 b 个正整数之和，使得这 b 个正整数乘积最大。请设计算法，求出 $n = 100$，$b = 7$ 时的最优分割。

4. 请修改 Edit-Distance 算法，使得能够返回前 k 个最优联配。

5. 考虑如下的“多重背包问题”：有一个承重量为 20 公斤的背包，现有 4 类物品，其重量和价值分别为：

物品编号	重量	价值
1	2	13
2	3	16
3	5	20
4	4	21

和 0-1 背包问题不同的是：每类物品的数量足够多；我们可以选择多个同类物品装入背包。因此，使得所装物品总价值最大的策略可以形式化如下的整数规划问题：

$$\begin{aligned} \max \quad & 13x_1 + 16x_2 + 20x_3 + 21x_4 \\ s.t. \quad & 2x_1 + 3x_2 + 5x_3 + 4x_4 \leq 20 \\ & x_1, \quad x_2, \quad x_3, \quad x_4 \in \mathbf{N} \end{aligned}$$

请设计动态规划算法求解上述问题。

6. 考虑如下的“设备更新问题” [?]：一台机床的买入价格是 22 万元，最多可以使用 5 年；在扣除维护费用之后，使用机床获得的净利润随机床年龄 t 递减，具体描述为：

$$I(t)=\begin{cases}25-2t-0.5t^2 & 0\le t\le 4\\ 0 & t\ge 5\end{cases}$$

请设计动态规划算法，求出在 5 年之内，何时购买新机床可获得的总利润（减去购买新机床的费用）最大。

7. 设计动态规划算法求解下述只包含一个约束条件的整数规划问题：

$$\begin{array}{lrcrcrcl}\min & x_1^2 & + & x_2^2 & + & x_3^2 & & \\ s.t. & 2x_1 & + & 3x_2 & + & 5x_3 & \ge & 20\\ & x_1, & & x_2, & & x_3 & \in & \mathbf{N}\end{array}$$

8. 设计动态规划算法求解下述包含两个约束条件的整数规划问题：

$$\begin{array}{lrcrcrcl}\min & x_1^2 & + & x_2^2 & + & x_3^2 & & \\ s.t. & 2x_1 & + & 3x_2 & + & 5x_3 & \ge & 20\\ & 3x_1 & + & 2x_2 & + & x_3 & \ge & 10\\ & x_1, & & x_2, & & x_3 & \in & \mathbf{N}\end{array}$$

提示：对每个约束条件设计一组状态变量。

9. 设计动态规划算法求解下述带有乘积型目标函数的整数规划问题：

$$\begin{array}{lrcrcrcl}\min & x_1 & \times & x_2^2 & \times & x_3^3 & & \\ s.t. & 2x_1 & + & 3x_2 & + & 5x_3 & \ge & 20\\ & x_1, & & x_2, & & x_3 & \in & \mathbf{N}\end{array}$$

10. 设计动态规划算法求解下述带有 MiniMax 型目标函数的整数规划问题：

$$\begin{array}{lrcrcrcl}\min\max & \{f_1(x_1), & & f_2(x_2), & & f_3(x_3)\} & & \\ s.t. & x_1 & + & 2x_2 & + & x_3 & = & 20\\ & x_1, & & x_2, & & x_3 & \le & 10\\ & x_1, & & x_2, & & x_3 & \in & \mathbf{N}\end{array}$$

其中函数 $f_1(\cdot)$,$f_2(\cdot)$,$f_3(\cdot)$ 的定义见 3.12。

11. 动态规划算法的适用范围不限于求解最优化问题；一般来说，只要能将求解过程描述成多步决策过程，即可应用。比如下述的计数问题：

一层楼梯共有 15 级台阶，上楼时每一步能够走 1 级或者 2 级台阶。请问共有多少种走法?

12. 考虑旅行商问题的一个特殊版本：寻找无向赋权图中的最短“金字塔”型环游（Pyramidal tour）。具体地说，给定图 $G = \langle V, E\rangle$，其中 $|V| = n$；顶点 i 和 j 之间的距离记为 d_{ij}。一个环游 t 是 n 的顶点的一个排列，而金字塔型环游是如下一种顶点序号“先升后降”的特殊排列，表示为：$1, i_1, \cdots, i_r, n, j_1, \cdots, j_{n-r-2}$，其中 $i_k < i_{k+1}$ $(1 \le k \le r-1)$，但是 $j_k > j_{k+1}$ $(1 \le k \le n-r-3)$。

    请设计求解最短金字塔型路径的 $O(n^2)$ 算法，并证明当距离矩阵具有 Monge 性质时（对任意的 4 个顶点 $i < i'$, $j < j'$，总有 $d_{ij} + d_{i'j'} \le d_{i'j} + d_{ij'}$），总存在一条最短环游是金字塔型的。

13. The matrix chain multiplication problem generalizes to solving a more abstract problem: given a linear sequence of objects, an associative binary operation on those objects, and a way to compute the cost of performing that operation on any two given objects (as well as all partial results), compute the minimum cost way to group the objects to apply the operation over the sequence.

    https://www.csc.lsu.edu/ gb/TCE/Publications/OptFramework-HIPS02.pdf

14. 计算 Pr[*seed*|*randombackgound*]. Two cases: uniformly random and Markov.

第四章　高级动态规划

4.1　引言

在上一章里，我们介绍了动态规划算法设计的基本技术，并把重点放在如何将实际问题建模成多步决策过程上；然而，这只是动态规划技术的一小部分内容：无论是空间复杂度还是时间复杂度，上一章讲述的算法还有很大的改进空间。在本章里，我们将从降低空间复杂度和降低时间复杂度两个方面介绍动态规划设计的高级技术。

通常来说，动态规划算法需要建立两个表格，其中 OPT 表记录子问题最优解的值，而 $OPTIndex$ 表则记录与最优解相应的最优决策项，以用于回溯。以 3.7节的字符串联配算法为例，对于长度分别为 m, n 的两个字符串来说，OPT 和 $OPTIndex$ 表都需要 $\Theta(mn)$ 的空间；当 m, n 较大时，表格的规模也会比较大，有时甚至存储不下。

那么，怎样才能降低动态规划算法的空间复杂度呢?

我们有两条思路可资使用：

(1) *以算代存*：回忆上一章中，我们是为了避免子问题的重复计算，采用“以存代算”策略，引入了子问题最优解值表和最优决策项表。现在既然存储不下，无法采用表格方式直接保存最优决策项，我们只好还是回归到“算”的策略，“算”出最优决策项；一旦算出了某一步的最优决策项之后（一般是最中间的决策步），其他步骤的最优决策项可以采用“分而治之”策略计算出来。

(2) *以“存函数”代替“存值”*：在 FFT 和 Tim-Cook 算法中，我们已经看到多项式的系数与多项式的值是表示多项式的两种方式。与之类似，最优解也有两种表示方式：一种是直接保存最优解的“值”，另一种是保存一个能够计算出这些值的“函数”；采用函数表示方式能够显著降低空间复杂度。当然，这种表示方式也要付出一些代价：采用函数表示方式时，通常只能表示出最优解的近似值，因此被称为*近似动态规划*。

进一步地，我们怎样才能降低动态规划算法的时间复杂度呢?

降低时间复杂度的基本思想是**避免冗余计算**。我们可以从如下几个方面来发现并进而避免冗余计算:

(1) 利用最优解值的稀疏性：在动态规划算法中，通常是以表格的方式存储所有子问题的最优解值。对于大多数问题来说，表格的每一项都需要填充。对于一些具有特殊性质的问题而言，表格会呈现出明显的稀疏性：大多数表格项是相同的，导致整个表格只有少数几种不同的值。稀疏性往往意味着加速的可能性：假如我们对相同项只计算一次的话，无疑将能避免很多的冗余计算，从而显著提高算法速度。

(2) 利用最优解值出现位置的集中性：如果最优解值出现位置比较集中的话，即使只计算最优解附近的那些表格项，也能够较为准确地计算出最优解来。直观上看，就是与最优解不太相关的表格项是冗余的，可以省略。

(3) 利用最优值表格的完全单调性 (Totally monotone):

(4) 利用最优决策项的:

4.2 降低空间复杂度：以算代存

我们以字符串联配算法为例描述降低空间复杂度的*以算代存*策略。回顾 3.7 节讲述的字符串联配算法 Edit-Distance：当计算两个字符串 $\mathrm{x} = x_1x_2\cdots x_m$ 和 $\mathrm{y} = y_1y_2\cdots y_n$ 的最优联配时，我们使用一个 $m \times n$ 的矩阵 OPT 记录子问题最优解的值，并且使用另一个 $m \times n$ 的矩阵 $OPTIndex$ 记录最优决策项以便于回溯（见图 3.20）；因此 Edit-Distance 算法的空间复杂度是 $O(mn)$ 的。

采用*以算代存*策略设计算法，我们能够将空间复杂度从 $O(mn)$ 降低至 $O(m+n)$。

4.2.1 线性空间复杂度的字符串联配算法：一个不太成功的尝试

即使只使用 $O(m+n)$ 的空间，我们也能够完成字符串联配。一种最基本的实现方案描述如下:

(1) *计算最优联配打分*: 如果只需要计算最优联配打分的话，我们无需使用 $m \times n$ 的矩阵 OPT，只需使用两个长度为 m 的数组即可完成。简要地说，矩阵 OPT

的右下角单元 $OPT[m,n]$ 记录了最优解的值；我们采用数组 $score$ 存放矩阵的第 $j-1$ 列、采用数组 new_score 存放矩阵的第 j 列；在第 j 步时，依据数组 $score$ 计算 new_score 中的单元，然后将数组 new_score 赋值给 $score$，接着执行第 $j+1$ 步；如此重复，直至第 n 步。当算法结束时，$new\_score[m]$ 即为最优解的值（图 4.1，算法的伪代码见**算法 36**）。

(2) **回溯构建最优联配**：这个问题的难点在于回溯。当不允许使用 $m \times n$ 的矩阵 $OPTIndex$ 记录最优决策项时，我们该依据哪些信息进行回溯，从而计算出最优联配呢？

以图 4.1所示联配为例，$new\_score[10]$ 是从 $score[9]$ 计算得来的，因此我们可以从 $new\_score[10]$ 回溯至 $score[9]$，但是由于没有保存 OPT 矩阵中其他列（白色背景），导致无法继续回溯。

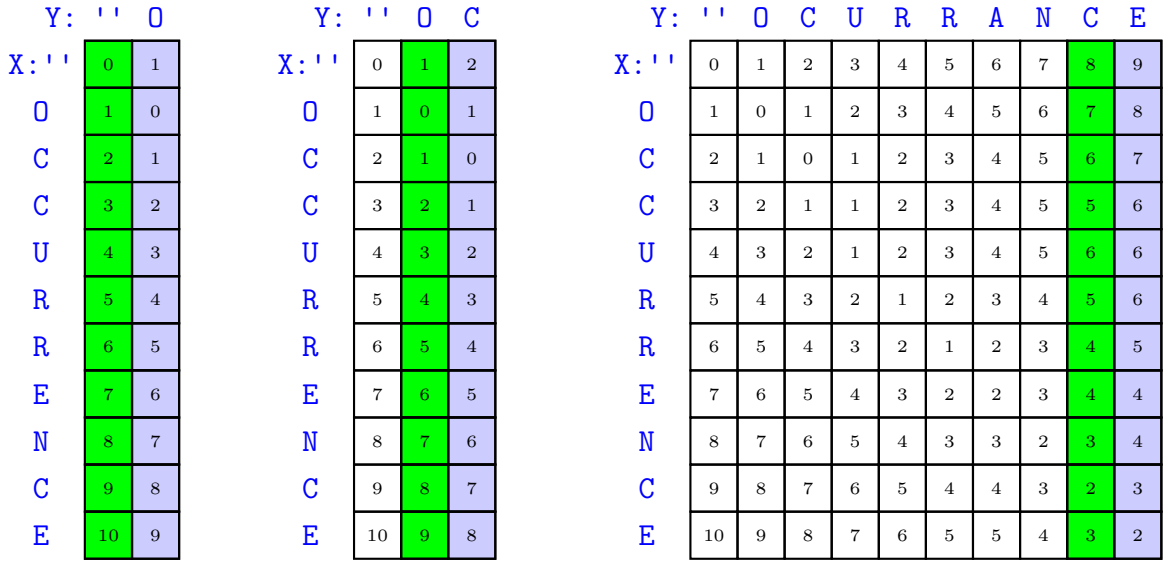


图 4.1: 只使用两个数组计算字符串 x = "OCCURRENCE" 和 y = "OCURRANCE" 编辑距离的运行过程：绿色数组 $score$ 存储已计算好的值，依据绿色数组可以计算出蓝色数组 new_score 中单元的值；当蓝色数组全部计算完毕之后，将其复制给绿色数组，如此逐步迭代，直至最后一列。在计算蓝色数组单元时，无需考虑绿色数组左侧单元，因此也无需保存（用白色背景表示）

一种可行的方案是**采用递归方式计算最优联配**，即：$new\_score[10]$ 是从 $score[9]$ 计算得来的，这意味着最后字符的最优决策是“字符 x_{10} 与字符 y_9 联配”；接下来我们对前缀 $x_1 \cdots x_9$ 和 $y_1 \cdots y_8$ 进行递归计算，就能得到其他字符的最优决策项。

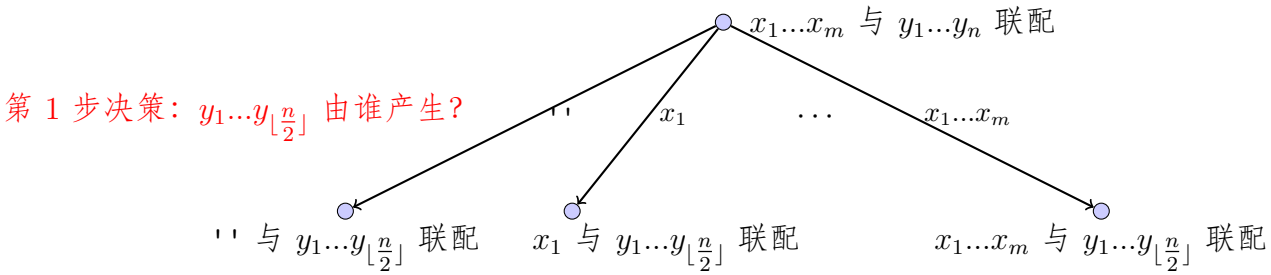


图 4.2: 计算两个字符串 x,y 最优联配的另一种多步决策过程（此处仅示出第一步决策）：在第一步决策时，我们考虑 y 的左一半 $y_1...y_{\lfloor\frac{n}{2}\rfloor}$ 由字符串 x 的哪些部分产生，共有 $m+1$ 种可能

这种方案是可行的，不过每次只把子问题的规模减少 1，导致较高的时间复杂度：共 $O(m+n)$ 次递归计算，每次用时 $O(mn)$，总计用时 $O((m+n)mn)$。

4.2.2 更高效的动态规划算法：从中间字符开始的多步决策过程

那是否有更高效的算法呢?

回顾我们在排序算法中学习到的经验：插入排序每次只把子问题规模减少 1，时间复杂度高；而归并排序每次把子问题规模减少一半，时间复杂度低。借鉴归并排序的思想，如果我们先求出“y 的中间字符 $y_{\lfloor\frac{n}{2}\rfloor}$ 的来源字符”的话，然后分别对 y 的左一半和右一半进行联配，这样分而治之，子问题的规模会每次减半，有可能降低时间复杂度。

从另一个观点来看，这实际上是换了一种多步决策过程：*第一步决策时考虑的是“y 的中间字符 $y_{\lfloor\frac{n}{2}\rfloor}$ 由谁产生”，而不是最后字符 y_n 或第一个字符 y_1 由谁产生。*

从中间字符开始的多步决策过程

有一个细节需要指出：$y_{\lfloor\frac{n}{2}\rfloor}$ 可能是“额外多写出来的”，可进一步细分成：写 x_1 之前额外多写出来的、写完 x_1 之后额外多写出来的、写完 x_2 之后额外多写出来的、.....、写完 x_m 之后额外多写出来的。罗列这么多种可能很啰嗦；为了简化描述，我们不是仅考虑中间字符 $y_{\lfloor\frac{n}{2}\rfloor}$ 由谁产生，而是考虑 y 的左一半 $y_1...y_{\lfloor\frac{n}{2}\rfloor}$ 由谁产生。

图 4.2展示了基于上述策略的多步决策过程的第一步决策：y 的左一半 $y_1...y_{\lfloor\frac{n}{2}\rfloor}$ 由 x 的哪个前缀产生的？这共有 $m+1$ 种可能，即：$y_1...y_{\lfloor\frac{n}{2}\rfloor}$ 由空字符串'' 产生、由 x_1 产生、由 x_1x_2 产生、.....、由 $x_1x_2...x_m$ 产生。

那这 $m+1$ 种决策项里，哪一个是最优决策项、能够产生最优联配呢?

回顾一下联配的定义：联配表示的从字符串 x 到字符串 y 的变换过程。既然整个字符串 y 是从 x 变换而来的，则其前缀 $y_1 \cdots y_{\lfloor \frac{n}{2} \rfloor}$ 必定是从 x 的一个前缀（设为 $x_1 \cdots x_r$）变换而来的，而其后缀 $y_{\lfloor \frac{n}{2} \rfloor + 1} \cdots y_n$ 也必定是从 x 的一个后缀 $x_{r+1} \cdots x_m$ 变换而来的（见图 4.3(a)）。

上述分析表明，x 和 y 的最优联配 (x′, y′) 可以拆分成前缀的联配和后缀的联配两个部分（见图 3.20(b)）；由于联配打分函数是线性加和形式，故有：

$$OPT(m,n) = OPT(r, \lfloor \tfrac{n}{2} \rfloor) + \overleftarrow{OPT}(r+1, \lfloor \tfrac{n}{2} \rfloor + 1).$$

此处，$\overleftarrow{OPT}(r+1, \lfloor \frac{n}{2} \rfloor + 1)$ 表示后缀 $y_{\lfloor \frac{n}{2} \rfloor + 1} \cdots y_n$ 与后缀 $x_{r+1} \cdots x_m$ 最优联配打分。由于 $OPT(m,n)$ 是最优联配打分，因此最优决策项 r 应该使其最大，即：

$$r = \underset{1 \le i \le m}{\operatorname{argmin}} \left(OPT(i, \lfloor \tfrac{n}{2} \rfloor) + \overleftarrow{OPT}(i+1, \lfloor \tfrac{n}{2} \rfloor + 1) \right).$$

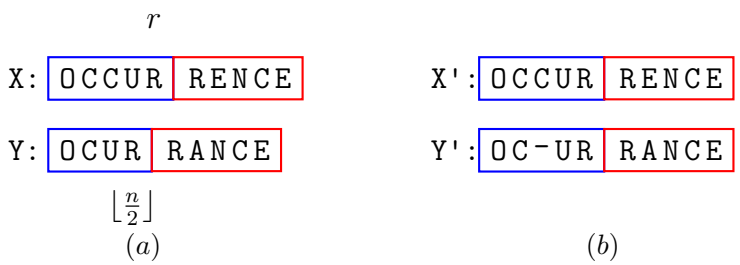


图 4.3: 计算两个字符串 x, y 之间最优联配的“分而治之”策略。(a) 字符串 y 从 x 的产生过程可以拆分成两部分：y 的前缀 y1$\cdots y_{\lfloor \frac{n}{2} \rfloor}$ 由 x 的前缀 $x_1 \cdots x_r$ 产生的过程，以及 y 的后缀 y$\lfloor \frac{n}{2} \rfloor + 1 \cdots y_n$ 由 x 的后缀 $x_{r+1} \cdots x_m$ 产生的过程；(b) 相应地，最优联配 (x′ y′) 也可以拆分成两个部分

算法设计与描述

1975 年，D. S. Hirschberg 提出了计算序列编辑距离的线性空间复杂度联配算法[?]，伪代码见算法 36，其基本思想就是上面提到的 3 点：

(1) 多步决策过程的第一步决策是问“y 的中间字符 $y_{\lfloor \frac{n}{2} \rfloor}$ 由谁产生”；

(2) 对于中间字符来说，有多个决策项，每个决策项都形成两个子问题，分别对应前缀 y1$\cdots y_{\lfloor \frac{n}{2} \rfloor}$ 和后缀 y$\lfloor \frac{n}{2} \rfloor + 1 \cdots y_n$ 的产生过程；由于联配打分是子问题打分的简单加和，因此，最优决策项是使得“子问题打分加和”最大的决策项；

(3) 采用“分而治之”策略计算 x, y 前缀之间、后缀之间的最优联配。*

值得指出的是：算法中的 Reverse-Space-Efficient-Edit-Distance 也是计算两个字符串之间的编辑距离，不过在执行过程中是从后向前进行计算，因此能够将所有后缀对之间的编辑距离计算出来。这只需修改 Space-Efficient-Edit-Distance 中的两处即可完成：*i*) 第 2 行的执行顺序反转，变为 j 从 |y| 到 1；*ii*) 第 4 行的执行顺序反转，变为 i 从 |x| 到 1。具体代码从略。

Algorithm 36 求解两字符串间编辑距离的线性空间复杂度算法

function Linear-Space-Edit-Distance(x, y, A, `offset`)

1: Set $m = |\mathrm{x}|$ and $n = |\mathrm{y}|$;
2: Allocate two arrays f and b; each array has a size of $m + 1$.
3: Space-Efficient-Edit-Distance(x, y$[1..\frac{n}{2}]$, f);
4: Reverse-Space-Efficient-Edit-Distance(x, y$[\frac{n}{2} + 1..n]$, b);
5: Let $r = \underset{0 \le i \le m|}{\operatorname{argmin}}(f[i] + b[i])$;
6: Append aligned-position $\langle r + \texttt{offset}, \frac{n}{2} \rangle$ to array A;
7: Free arrays f and b;
8: Linear-Space-Edit-Distance(x$[1..r]$, y$[1..\frac{n}{2}]$, A, `offset`);
9: Linear-Space-Edit-Distance(x$[r + 1..m]$, y$[\frac{n}{2} + 1..n]$, A, `offset` $+ \frac{n}{2}$);
10: **return** $\min\limits_{0 \le i \le m|}(f[i] + b[i])$;

function Space-Efficient-Edit-Distance(x, y, *score*)

1: Set $m = |\mathrm{x}|$ and $n = |\mathrm{y}|$;
2: Initialize $score[i] = s(x_i, '-') * i$ for each i $(0 \le i \le m)$;
3: **for** $j = 1$ to n **do**
4: $\quad new\_score[0] = s('-', y_j) * j$;
5: $\quad$ **for** $i = 1$ to m **do**
6: $\quad\quad new\_score[i] = \min\{score[i-1] + s(x_i, y_i), score[i] + s(x_i, '-'), newscore[i-1] + s('-', y_j)\}$;
7: $\quad$ **end for**
8: $\quad$ Assign $score[i] = newscore[i]$ for each i $(0 \le i \le m)$;
9: **end for**
10: **return** $score[m]$;

*在计算图中所有结点对之间最短路径时，Floyd-Warshall 算法也是采用类似的思想，即：使用“分而治之”策略计算最短路径，并记录最短路径经过的结点以便于回溯 [**?**, **?**]

运行过程示例

图??展示 Linear-Space-Edit-Distance 算法计算字符串 x = "OCCURRENCE" 和 y = "OCURRANCE" 之间编辑距离的运行过程。图的左侧显示交替使用两个数组（用绿色和蓝色表示）计算 x 和 y 的前缀 "OCUR" 之间的编辑距离，右侧显示计算 x 和 y 的后缀 "RANCE" 之间的编辑距离。在计算前缀之间的编辑距离时，是从左上角开始自上而下逐个单元进行计算；与之相反，在计算后缀之间的编辑距离时，是从右下角开始，自下而上逐个单元进行计算。

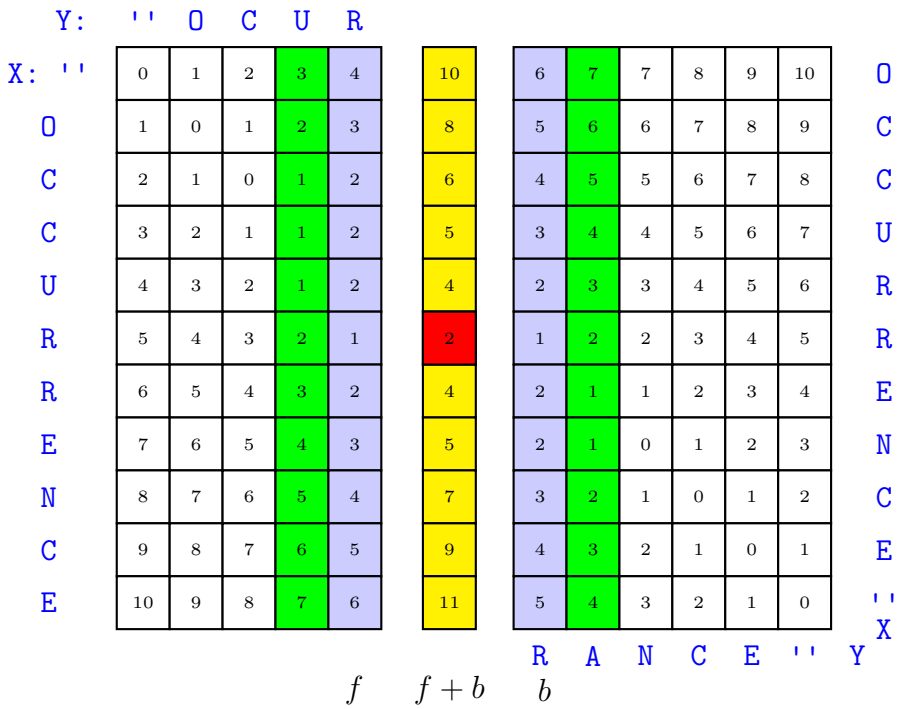


图 4.4: Linear-Space-Edit-Distance 算法计算字符串 x = "OCCURRENCE" 和 y = "OCURRANCE" 之间编辑距离的运行过程

接下来，Linear-Space-Edit-Distance 算法计算数组 $f + b$ 中的最小元。最小元有两方面的含义：

(1) *值就是编辑距离*：最小元的值表示 x 和 y 之间的编辑距离。在这个例子中，最小元的值是 2，验算一下，与 Edit-Distance 算法的计算结果是一致的（见图 3.20中 OPT 矩阵的右下角单元）。

(2) *位置就是最优决策项*：最小元的位置 $r = 5$，意味着在 x 和 y 的最优联配中，y 的第 $1 - 4$ 个字母 "OCUR" 是和 x 的第 $1 - 5$ 个字母 "OCCUR" 进行联配，y

的第 $5-9$ 个字母 "RANCE" 是和 x 的第 $6-10$ 个字母 "RENCE" 进行联配。为记录最优联配中的这个拆分位置，我们在数组 A 中增加一项 <5, 4>。

然后，Linear-Space-Edit-Distance 算法递归计算 "OCUR" 和 "OCCUR" 之间的最优联配（拆分位置 <3, 2>）、"RANCE" 和 "RENCE" 之间的最优联配（拆分位置 <7, 6>），详细过程不再赘述。

最终，我们将得到如下的数组 A：

$A = [$<5, 4>, <3, 2>, <1, 1>, <4,3>, <7,6>, <6,5>, <8,7>, <9,8>$]$.

数组 A 记录了最优联配的“分而治之”求解过程中全部拆分位置（见图 4.5）；我们把图中的叶子结点集合在一起，即可重构出如下的最优联配：

x′: OCCURRENCE

y′: OC-URRANCE

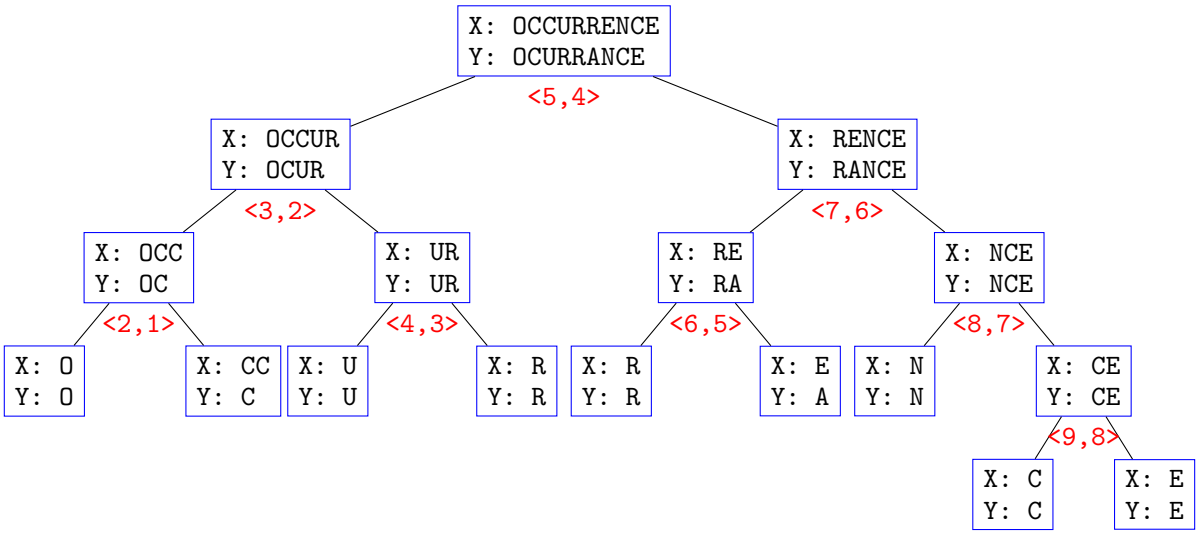


图 4.5: Linear-Space-Edit-Distance 算法计算字符串 x = "OCCURRENCE" 和 y = "OCURRANCE" 之间最优联配的“分而治之”运算过程

时间复杂度与空间复杂度分析

在 Linear-Space-Edit-Distance 算法中，数组 f 和 b 各包含 m 个单元；记录联配位置的数组 A 占用 $2n$ 个单元；因此，算法的空间复杂度是 $O(m+n)$ 的，相比于传统的 Edit-Distance 算法有显著的降低。

一个合理的担心是：空间复杂度是降低了，那时间复杂度会不会上升了呢?

下述的定理表明这个担心是不必要的：Linear-Space-Edit-Distance 算法的时间复杂度和传统的 Edit-Distance 算法相同，依然是 $O(mn)$ 的。换句话说，空间复杂度的降低，并未以时间复杂度的上升为代价。

定理 4.2.1. Linear-Space-Edit-Distance *算法的时间复杂度是* $O(mn)$ *的。*

Linear-Space-Edit-Distance 算法包含两个递归调用，因此运行时间是：

$$T(m,n) = cmn + T(r,\tfrac{n}{2}) + T(m-r,\tfrac{n}{2}),$$

其中 cmn 表示 Space-Efficient-Edit-Distance 算法的运行时间。

分析这个递归表达式的难处在于：与通常的“分而治之”算法不同，这里的 r 无法事先确定。只有当执行完 Space-Efficient-Edit-Distance 和 Reverse-Space-Efficient-Edit-Distance 之后，我们才能确定 r 的值。

这导致我们无法直接应用第 2 章中 Master 定理分析时间复杂度。我们只好采用“先猜测、再验证”策略，或者“展开递归表达式、观察规律”策略。这里我们使用第一种策略进行证明；采用第二种策略也可完成证明，在此不赘述。

证明：

假设对于任意的 $0 < m' < m$ 和 $0 < n' < n$，都有 $T(m',n') \le km'n'$ 成立，其中 k 是待定参数；则可证得：

$$\begin{aligned} T(m,n) &= cmn + T(r,\tfrac{n}{2}) + T(m-r,\tfrac{n}{2}) \\ &\le cmn + kr\tfrac{n}{2} + k(m-r)\tfrac{n}{2} \\ &\le (c+\tfrac{k}{2})mn \\ &= kmn \qquad\qquad (\text{令} k = 2c) \end{aligned}$$

□

4.2.3 进一步的改进：采用递归策略计算最优决策项

上一小节讲述的 Hirschberg 算法虽然有效，但是由于下面两个原因，导致其难以推广到其他问题。

(1) *需要新设计一个算法*：Hirschberg 算法在中心决策步处把原问题分解成两个子问题，一个子问题计算 x 前缀与 y 前缀之间的编辑距离，另一个子问题计算后缀之间的编辑距离；在计算后缀之间编辑距离时，还需要新设计 Reverse-Space-Efficient-Edit-Distance 算法。只有在计算出前缀之间

以及后缀之间编辑距离之后，我们才能够确定中心决策步的最优决策项 r 的值。

(2) 依赖于打分函数的线性加和性质：在计算中间字符的最优决策项时，Hirschberg 算法是把前缀打分和后缀打分相加，得到与一个决策项相对应的打分；进而枚举所有的决策项，确定出最优决策项。

1982 年，J. Ian Munro 等提出了一个更简洁和通用的算法：不采用枚举策略计算中心决策步的最优决策项 r，而是采用递归方式进行计算。这样带来的好处是：一方面可以将对最优决策项 r 的计算融入计算 OPT 值的过程之中，另一方面，由于避免了新设计 Reverse-Space-Efficient-Edit-Distance 算法，从而具有普适性，可以推广到解决其他问题的动态规划算法 [?]。

那如何采用递归方式进行计算字符串 $\mathrm{x} = x_1x_2\cdots x_m$ 和 $\mathrm{y} = \mathrm{y_1y_2}\cdots\mathrm{y_n}$ 的最优联配中，y 的前一半 $y_1\cdots y_{\lfloor\frac{n}{2}\rfloor}$ 与 x 的联配位置呢?

我们可以这样思考：

(1) 最简单的情形：保持 x 不变，把 y 缩短为原来的一半，即：计算字符串 $\mathrm{x} = x_1x_2\cdots x_m$ 和 $\mathrm{y} = \mathrm{y_1y_2}\cdots\mathrm{y}_{\lfloor\frac{\mathrm{n}}{2}\rfloor}$ 的最优联配中，$y_1\cdots y_{\lfloor\frac{n}{2}\rfloor}$ 与 x 的联配位置。这种情形很容易求解，就是 m。

(2) 复杂一些的情形：计算字符串 $\mathrm{x} = x_1x_2\cdots x_m$ 和 $\mathrm{y} = \mathrm{y_1y_2}\cdots\mathrm{y}_{\lfloor\frac{\mathrm{n}}{2}\rfloor+1}$ 的最优联配中，$y_1\cdots y_{\lfloor\frac{n}{2}\rfloor}$ 与 x 的联配位置。这时我们就能够发现递归关系。

例如，对于字符串 x = "OCCURRENCE" 和 y = "OCURRANCE" 来说，计算得到中心决策步的最优决策项 $r = 5$；而对于 x 的前缀 "OCCURRENC" 和 y 的前缀 "OCURRANCE"，计算得到中心决策步的最优决策项依然是 $r = 5$。这意味着有可能根据前缀联配的中心决策步的最优决策项，计算出整个字符串联配的中心决策步的最优决策项。

算法设计与描述

为了描述的方便，我们首先把一个整数 r 扩展成一个二维矩阵 R_{ij}，以表示前缀联配中心决策步的最优决策项：

定义 4.2.1 (前缀联配中心决策步的最优决策项). 考虑字符串 $\mathrm{x} = x_1x_2\cdots x_m$ 和 $\mathrm{y} = \mathrm{y_1y_2}\cdots\mathrm{y_n}$，定义 R_{ij} 为 y 的前缀 $y_1\cdots y_j, \lfloor\frac{n}{2}\rfloor \le j \le n$ 与 x 的前缀 $x_1\cdots x_j$ 最优联配中，y 的前一半 $y_1\cdots y_{\lfloor\frac{n}{2}\rfloor}$ 与 x 的联配位置。

直观上看，$R[i,j]$ 表示在 x 和 y 的最优联配打分矩阵 OPT 中，从 (i,j) 单元到 $(0,0)$ 单元的回溯路径经过第 $\lfloor \frac{n}{2} \rfloor$ 列时的行号。以图 3.20(b) 所示联配及回溯过程为例，我们有 $R_{10,9} = 5$ 。至于 Hirschberg 算法中的 r，不过是 $R_{i,j}$ 的特例，即：$r = R_{m,n}$。

从图 3.20(b) 我们还可以看出：$R_{10,9} = R_{9,8} = 5$；其中的道理很简单：$OPT(10,9)$ 是从 $OPT(9,8)$ 计算得出的，因此第一步回溯时，我们从 $OPT(10,9)$ 回溯至 $OPT(9,8)$；进一步地，从 $OPT(9,8)$ 回溯时在第 5 行经过第 4 列，从 $OPT(10,9)$ 回溯时必定也会在第 5 行经过第 4 列。

我们把这个观察总结成 $R(i,j)$ 之间的递归关系：

$$R_{i,j} = \begin{cases} i & \text{如果} j = \lfloor \frac{n}{2} \rfloor \\ R_{i-1,j} & \text{如果} j > \lfloor \frac{n}{2} \rfloor \text{且} OPT[i,j] = OPT[i-1,j] + s(x_i, \texttt{'-'}) \\ R_{i-1,j-1} & \text{如果} j > \lfloor \frac{n}{2} \rfloor \text{且} OPT[i,j] = OPT[i-1,j-1] + s(x_i, y_j) \\ R_{i,j-1} & \text{如果} j > \lfloor \frac{n}{2} \rfloor \text{且} OPT[i,j] = OPT[i,j-1] + s(\texttt{'-'}, y_j) \end{cases}$$

	Y: ''	O	C	U	R	R	A	N	C	E
X:''	–	–	–	–	0	0	0	0	0	0
O	–	–	–	–	1	1	1	1	1	1
C	–	–	–	–	2	2	2	2	2	2
C	–	–	–	–	3	3	3	3	2	2
U	–	–	–	–	4	4	4	4	4	2
R	–	–	–	–	5	5	5	5	5	5
R	–	–	–	–	6	5	5	5	5	5
E	–	–	–	–	7	5	5	5	5	5
N	–	–	–	–	8	5	5	5	5	5
C	–	–	–	–	9	5	5	5	5	5
E	–	–	–	–	10	5	5	5	5	5

(a)

	Y: ''	O	C	U	R
X:''	–	–	0	0	0
O	–	–	1	1	1
C	–	–	2	2	2
C	–	–	3	2	2
U	–	–	4	3	2, 3
R	–	–	5	3	3

(b)

	Y: ''	O	C
X:''	–	0	0
O	–	1	1
C	–	2	1
C	–	3	1, 2

(c)

图 4.6: 采用递归表达式计算前缀联配位置 $R_{i,j}$ 的运算过程示例。(a) 对字符串 x = "OCCURRENCE"，y = "OCURRANCE"，最终计算得出 $r = R_{10,9} = 5$；(b) 对字符串 x = "OCCUR"，y = "OCUR"，最终计算得出 $r = R_{5,4} = 3$；(c) 对字符串 x = "OCC"，y = "OC"，最终计算得出 $r = R_{3,2} = 1$ 或 2，表示最优联配存在两种情形

值得指出的是，上述分析意味着对 $R_{i,j}$ 的计算完全可以融入 $OPT[i,j]$ 的计算过程之中：只需对 Space-Efficient-Edit-Distance 代码做小幅修改，依据 $OPT[i,j]$

的三种情形对 $R_{i,j}$ 进行赋值即可完成，且不会影响算法的时间复杂度和空间复杂度。具体的伪代码从略。

运行过程示例

图 4.6展示依据上述递归表达式计算 r 的过程：以 $R_{10,9}$ 为例，由于 $OPT(10,9)$ 来自于 $OPT(9,8)$，因此我们进行赋值 $R_{10,9} = R_{9,8}$；我们最终计算得出 $r = R_{10,9} = 5$，这与图**??**展示的结果相同。最终，我们将得到和上一小节相同的数组 A，并进而重构出相同的最优联配。

和上一小节讲述的 Hirschberg 原始算法相比而言，Ian Munro 提出的算法更简洁、通用性更强。联合使用 Hirschberg 的“分而治之”策略以及 Ian Munro 的最优决策项递归计算技术，我们可以使用与计算最优值相同的时间与空间，回溯计算出最优解来。这具有很重要的指导性意义：对动态规划算法来说，我们只需考虑如何高效地计算出最优解的值，而不必要再担心如何回溯的问题了。

当然了，如果仅计算最优解的值就需要很大空间的话，上述策略也不奏效。在这种情况下如何降低空间复杂度，我们将在下一小节介绍。

4.3 降低空间复杂度：以“存函数”代替“存值”

降低动态规划算法空间复杂度的第二种策略是以“存函数”代替“存值”，即：不采用表格方式显式地保存所有子问题最优解的值，而是保存一个能够算出这些值的“函数”；当需要知道子问题最优解的值时，直接调用这个函数进行求解。

在 FFT 变换中，我们已经见到过这种“值”与“函数”之间的相互转换；这里我们希望利用这种转换的下述优点：和保存函数的值相比，保存函数的系数通常需要更少的空间。

那么采用何种函数，才能够对于形式多变的子问题，都能够计算出（或者近似计算出）最优解的值来呢?

1977 年，Paul J. Werbos 首次提出采用神经网络作为计算子问题最优解值的函数，并利用*反向传播算法*[†]求解 Bellman 方程。这个策略的好处是显而易见的：可以“时间向前”(forward in time）地求解子问题，从而一定程度上避免了“维数灾难”。当然，这样处理也要付出相应的代价：通常无法计算出最优解的值，只能得到近似解；

[†]Paul J. Werbos 是神经网络训练中反向传播算法的提出者 [**?**]

因此，这种策略被称为近似动态规划（Adative/approximate dynamic programming）[?, ?, ?]。

沿着这条路线，杨飞雕等提出了求解 TSP 问题的 NNDP 算法 [?]；和经典的 Bellman-Held-Karp 算法相比，NNDP 算法的空间复杂度显著降低。本小节以 NNDP 为例讲述这种采用神经网络技术降低空间复杂度的策略。

算法设计与描述

回顾第一章中讲述的 Bellman-Held-Karp 动态规划算法，其核心步骤是计算从起始结点 s 出发、经过结点集合 S 中的结点一次且仅一次、最终到达目的结点 x 的最短里程 $M(s,S,x)$。我们采用下面的递归表达式计算 $M(s,S,x)$：

$$M(s,S,x)=\min_{v\in S,\ v\neq x} M(s,S\backslash\{v\},v)+d_{v,x}. \tag{4.3.1}$$

由于要枚举所有的结点子集 S，因此 Bellman-Held-Karp 算法的时间复杂度和空间复杂度分别是 $O(2^n n^2)$ 和 $O(2^n n)$ 的，对规模稍大的实例即无能无力。

为降低算法的空间复杂度，NNDP 算法不显式地使用表格保存 $M(s,S,x)$ 的值，而是试图训练出一个能够直接计算出 $M(s,S,x)$ 的神经网络。如图 4.7所示，这个神经网络有两类输入：结点集合 S（用 n 个二进制位表示，每个二进制位表示一个结点是否在 S 中），以及目的结点 x（用 one-hot 方式表示）。为描述简洁起见，我们固定起始结点 s 是 V 中第一个结点 v_1，因此不显式地将其作为神经网络的输入。神经网络的输出记为 $f(S,x;\theta)$，其中 θ 表示神经网络的参数。

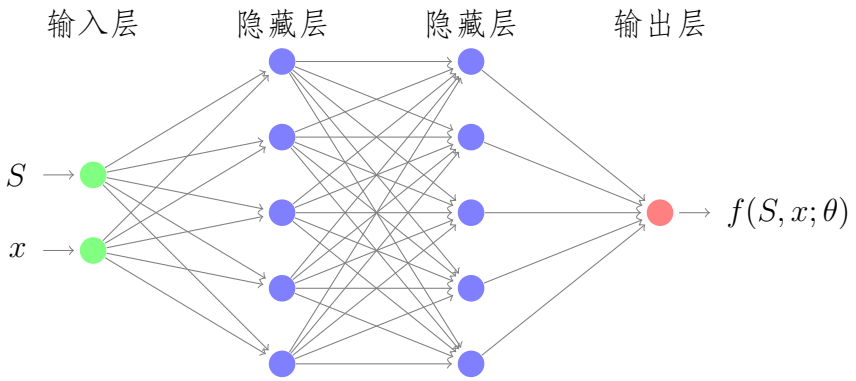


图 4.7: NNDP 算法中使用的全连接神经网络模型。输入为目的结点 x、中间结点集合 S；输出为 $f(S,x;\theta)$，用于逼近 $M(s,S,x)$；中间两个隐藏层各包含 $4n$ 个神经元

在理想情况下，神经网络的输出 $f(S,x;\theta)$ 就是 $M(a,S,x)$；因此最直观的神经网络训练方法就是直接最小化输出 $f(S,x;\theta)$ 与理想输出 $M(a,S,x)$ 之间的差异。然而，

这个想法有一个非常大的困难：计算 $M(a,S,x)$ 本身就需要指数多的时间。按照机器学习的术语来讲，我们是以 $M(s,S,x)$ 作为“标注”（Label）；但是由于标注数据难以获得，使得我们无法使用通常的优化方法训练神经网络。

为克服这个困难，杨飞雕等从动态规划算法的特点出发，设计了一个新的优化目标：不是直接去优化输出 $f(S,x;\theta)$ 与理想输出 $M(a,S,x)$ 之间的差异，而是使得 $f(S,x;\theta)$ 满足递归关系。其基本思想是：既然 $f(S,x;\theta)$ 的理想值是 $M(a,S,x)$，则在理想情况下，$f(S,x;\theta)$ 也应该如同 $M(a,S,x)$ 一样，满足递归关系 4.3.1，即：

$$f(S,x;\theta)=\min_{v\in S,\ v\neq x}\{f(S\backslash\{v\},v;\theta)+d_{v,x}\}.$$

因此，我们可以将优化目标设置为最小化不满足上述递归关系的程度，即：

$$\min_{\theta}\sum_{(S,x)\in\mathcal{T}}\left(f(S,x;\theta)-\min_{v\in S,\ v\neq x}\{f(S\backslash\{v\},v;\theta)+d_{v,x}\}\right)^2.$$

此处的 $\mathcal{T}$ 表示训练集，其构建过程值得解释：对一个给定的 TSP 实例而言，所有可能的 (S,x) 共有指数多个；为降低训练的难度，NNDP 算法中使用带随机扰动的采样策略，采样出有代表性的 (S,x) 以构建训练集。

在训练神经网络得到最优的参数 θ^* 之后，NNDP 算法采用“时间向前”的方式计算出最短环游：

Algorithm 37 Algorithm to solve TSP using neural network

function NNDP(V,D,θ^*)

Set $n=|V|$ and set Π^* as an arbitrary tour;
for $x=v_2$ to v_n **do**
  Initialize tour Π as (v_1,x);
  for $i=1$ to $|V|-2$ **do**
    $S=V\backslash\Pi$;
    $v^*=\text{argmin}_{v\in S,\ v\neq x}\{f(S\backslash\{v\},v;\theta^*)+d_{v,x}\}$;
    Append v^* to Π;
    $x=v^*$;
  end for
  Calculate the length of tour Π, and update $\Pi^*=\Pi$ if Π is shorter than Π^*;
end for
return Π^*;

实验结果分析

在经典测试集 TSPLIB 上，杨飞雕等测试了 NNDP 算法，并与近似算法 Christofides、采用最近邻规则的贪心算法进行了比较。如表 4.1所示，尽管 Held-Karp 算法能够得到最优解，但是对规模稍大的实例即无法运行；NNDP 算法表现出较好的性能，其相对误差一般都在 10% 以内，显著优于其他算法。

表 4.1: NNDP 算法性能分析与比较

		NNDP		Held-Karp		Christofides		Greedy	
TSP 实例	最短环游里程	里程	比值	里程	比值	里程	比值	里程	比值
Gr17	2085	**2085**	**1**	**2085**	**1**	2287	1.16	22	1.04
Bayg29	1610	**1610**	**1**	NA	NA	1737	1.08	1935	1.20
Dantzig42	699	**709**	**1.01**	NA	NA	966	1.38	863	1.23
HK48	11461	**11539**	**1.01**	NA	NA	13182	1.15	12137	1.06
Att48	10628	**10868**	**1.02**	NA	NA	15321	1.44	12012	1.13
Eil76	538	**585**	**1.09**	NA	NA	651	1.11	598	1.11
Rat99	1211	**1409**	**1.16**	NA	NA	1665	1.37	1443	1.19
Br17	39	**39**	**1**	**39**	**1**	NA	NA	56	1.44
Ftv33	1286	**1324**	**1.03**	NA	NA	NA	NA	1589	1.20
Ft53	6905	**7343**	**1.06**	NA	NA	NA	NA	8584	1.17

时间复杂度与空间复杂度分析

NNDP 算法采用含两个隐层的神经网络结构，每个隐层各包含 $4n$ 个神经元；因此，神经网络共有 $O(n^2)$ 个参数。这意味着 NNDP 算法只需保存 $O(n^2)$ 个参数，而无需像 Bellman-Held-Karp 算法那样直接保存 $O(2^n n)$ 个子问题最优解的值。

神经网络的训练过程采用梯度下降法；实验结果表明只需少数几轮迭代即可收敛。在训练完神经网络之后，NNDP 算法求解时只需执行 $O(n^2)$ 次神经网络的前向传播，因此时间复杂度是 $O(n^4)$。

一些讨论

值得特别指出的是，NNDP 算法采用**时间向前**的方式构建环游，这与动态规划算法依赖回溯技术计算最优解迥然不同，也是 NNDP 算法能够部分克服“维数灾难”的原因之一。

还有一点需要注意的是：NNDP 算法依赖于神经网络优异的拟合函数的能力；然而，无论拟合能力有多强，神经网络计算出来的只是最优解值的一个近似。那么，为何采用这种“近似”的策略，却依然能够计算出足够好的环游呢？其原因大致有如下三点：

(1) *有时动态规划中子问题的解并不需要绝对精确地计算*：在 NNDP 算法中，即使是采用训练之后得到的最优参数 θ^*，神经网络的输出 $f(S,x;\theta^*)$ 依然可能与其理想值 $M(v_1,S,x)$ 之间存在差异。但是这不构成大的妨碍：我们利用 $f(S,x;\theta^*)$ 的目的是试图寻找最短环游；在找到一个环游之后，其里程是可以精确计算的。

(2) *在实际应用中，次优解往往足够*：即使神经网络表示的动态规划函数不甚精确，但只要能够高效地求得次优解，也是十分有价值的。这一点对于 NP-完全问题求解尤为重要。

(3) *并非所有的子问题都是同等重要的*：一个典型例子如图**??**所示，偏离对角线特别远的单元，对于计算最优解作用很小，可以忽略。因此，对于神经网络训练过程来说，我们期望以较高的精度逼近重要子问题的最优解，而对于那些不太重要的子问题，其逼近程度可以放松。在 NNDP 算法中，重要的子问题是利用带扰动的采样技术获得的。

4.4 降低时间复杂度：利用子问题最优解值的稀疏性

稀疏是很常见的一种现象。比如对包含 n 个结点的图来说，所有可能的边数共有 $\frac{1}{2}n(n-1)$ 条；然而通常来说，真正出现的边数远小于这个数目。这样的图称作*稀疏图*，反之则称为*稠密图*。再以压缩感知为例，其基本假设是信号的稀疏性，即：在表示信号的向量中，非 0 元的个数很少 [**?**]。在本节中，我们对稀疏性的定义略有不同：不是以非 0 元的个数，而是以不同取值的总数。详细地说，假如在一个动态规划最优值表中，*只有少数几种不同的取值*，则称这个表格是*稀疏*的。

我们以最长公共子串问题为例，来说明如何利用最优值表格的稀疏性改进动态规划算法。最长公共子串问题描述如下：给定两个字符串 $\mathrm{x} = x_1\cdots x_m$, $\mathrm{y} = y_1\cdots y_n$，计算 x 和 y 的最长公共子序列（为描述简洁起见，我们假设 $m \le n$）。这里值得指出的是子序列（Subsequence）和子串（Substring）的不同之处：一个字符串的子序

列是指从字符串中抽取出一些字符、并依照它们在字符串中的出现顺序合并而成的短字符串；这些字符在原字符串中可以是不连续出现的。例如："Course" 是字符串 "Computer science" 的一个子序列，虽然字符'o' 和字符'u' 并不相邻。

如同我们在第 3 章中讲述的，最长公共子串问题实质上是最小编辑距离问题的特例，因此可以对 EditDistance 算法略作改动即可求解。简要地说，我们定义子问题为：计算字符串 x 的前缀 $x_1 \cdots x_i$ 与 y 的前缀 $y_1 \cdots y_j$ 之间的最长公共子序列，其长度记为 $OPT(i,j)$。$OPT(i,j)$ 可以按下述递归关系进行计算：

$$OPT(i,j) = \begin{cases} 0 & \text{如果} i=0 \text{或} j=0 \\ OPT(i-1,j-1)+1 & \text{如果} x_i = y_j \\ \max\{OPT(i-1,j), OPT(i,j-1)\} & \text{否则} \end{cases} \tag{4.4.1}$$

其中第二项来自于如下事实：如果 $x_i = y_j$，则对于前缀 $x_1 \cdots x_i$ 与 $y_1 \cdots y_j$ 来说，必定存在一个以字符 x_i 结尾的最长公共子序列。

依据上述递归关系，我们很容易设计出计算 $OPT(m,n)$ 的动态规划算法（称作 LCS 算法，伪代码从略），其时间复杂度为 $O(mn)$。我们在此仅展示对字符串 x = "abcdbb" 和 y = "cbacbaaba" 的运行过程（见图 4.8）。

	Y: ''	c	b	a	c	b	a	a	b	a
X:''	0	0	0	0	0	0	0	0	0	0
a	0	0	0	①	1	1	①	①	1	①
b	0	0	①	1	1	②	2	2	②	2
c	0	①	1	1	②	2	2	2	2	2
d	0	1	1	1	2	2	2	2	2	2
b	0	1	②	2	2	③	3	3	③	3
b	0	1	②	2	2	③	3	3	④	4

图 4.8: LCS 算法的运行过程示例。红色圆圈表示等值区域的边界点，蓝色圆圈表示除边界点之外的字符匹配点

4.4.1 利用稀疏性加速动态规划算法

从图 4.8所示实例中，我们可以观察到虽然最优解值表有 70 项，却仅有 $0,1,2,3,4$ 共 5 种不同的取值。值得指出的是，这种现象不是个例，而是一种普遍现象：最优值表共有 mn 项，然而不同取值的个数取决于最长公共子序列的长度，因此至多只

有 $\min\{m,n\}+1$ 种可能的取值。此外，从计算最优解的回溯过程来看，回溯只需要 $O(n)$ 的时间，这意味着起作用的关键项只有 $O(n)$ 个。我们将这种现象描述如下：

观察 1：当采用 LCS 算法计算最长公共子序列时，子问题最优解值呈现出显著的稀疏性。

算法设计与描述

子问题最优值表的稀疏性带来一些启发：既然可能的取值数目很少，或许我们只需计算最优值表中的部分单元，即：启发我们存在大量的冗余计算，也意味着加速的可能性。比如图 4.8中取值为 2 的共有 23 项；然而这 23 项不必每项都计算：我们只需计算圆圈标示的 5 项即可，其余 18 项都可以由这 5 项确定。图中的圆圈标示字符匹配点：当发现两个相等字符 $x_i=y_j$ 时，我们称 (i,j) 为**字符匹配点**。

之所以可以只关注字符匹配点，这是因为对任意一个不匹配点 (i,j) 来说，我们可连续应用递归关系式 4.4.1中的第 3 项，直至将其表示成字符匹配点，即：

$$\begin{aligned} OPT(i,j) &= \max\{OPT(i-1,j), OPT(i,j-1)\} \\ &= \cdots\cdots \\ &= \max\{OPT(i',j') | x_{i'}=y_{j'} \text{且} i'\le i, j'\le j\} \end{aligned}$$

依据上述分析，我们将递归表达式 4.4.1改进成只使用匹配点的形式：

$$OPT(i,j)=\begin{cases} 0 & \text{如果} i=0 \text{或} j=0 \\ \max\{OPT(i',j') | x_{i'}=y_{j'} \text{且} i'<i, j'<j\}+1 & \text{如果} x_i=y_j \\ \max\{OPT(i',j') | x_{i'}=y_{j'} \text{且} i'\le i, j'\le j\} & \text{否则} \end{cases}.$$

基于这个改进后的递归表达式，J. Erickson 设计了稀疏动态规划算法 Sparse-LCS，其伪代码见算法 38[**?**]。

Algorithm 38 求解最长公共子序列问题的稀疏动态规划算法

function SPARSE-LCS(x, y)

```
1: Matches =FIND-MATCHES(x, y);
2: Set M = |Matches| and return 0 if M == 0;
3: Sort Matches lexicographically (i.e., in the row-major order);
4: for k = 1 to M do
5:     (i, j) = Matches[k];
6:     LCS[k] = 1;
7:     for l = 1 to k − 1 do
8:         (i′, j′) = Matches[l];
9:         if i′ < i and j′ < j then
10:            LCS[k] = max{LCS[k], 1 + LCS[l]};
11:        end if
12:    end for
13: end for
14: return LCS[M];
```

如图 4.8所示，SPARSE-LCS 算法首先计算出下列的字符匹配点：

$$Matches = [(1,3),(1,6),(1,7),(1,9),(2,2),(2,5),(2,8),\cdots].$$

然后只对这些字符匹配点按行从小到大、按列从左到右进行计算。比如先计算第一行的字符匹配点，得到：

$$LCS(1,3) = LCS(1,6) = LCS(1,7) = LCS(1,9) = 1.$$

然后计算第二行，得到：

$$LCS(2,2) = 1, LCS(2,5) = LCS(1,3) + 1 = 2.$$

时间复杂度分析

在这个算法中，FIND-MATCHES 用来计算字符串 x, y 中所有的匹配字符对；这可以采用如下步骤完成：先将字符串表示成字符、出现位置对（例如：(x_i, i)）；然后按字符进行排序；最后找出所有的 $x_i = y_j$，将 (i, j) 加入 $Matches$ 数组中。由于字符集大小 s 通常远远小于 m 和 n，因此采用基数排序（Radix sort）更为高效；故 FIND-MATCHES 的时间复杂度是 $O(m \log s + n \log s + M)$，其中 M 是匹配点的数量。

SPARSE-LCS 算法中包括两重循环，因此时间复杂度是 $O(m \log s + n \log s + M^2)$。当 $M = o(\sqrt{mn})$ 时，SPARSE-LCS 的时间复杂度优于 LCS 算法。

4.4.2 只与边界点比较以进一步加速

虽然 Sparse-LCS 算法只对字符匹配点进行操作，但是其中依然还有一些计算是冗余的。以图 4.8中的字符匹配点 $(3,4)$ 为例，Sparse-LCS 算法将其与前 8 个字符匹配点进行下标比较运算。事实上，我们只需将其与 $(2,1)$ 和 $(2,5)$ 这两个“边界点”（红色圆圈所示）进行列号的比较，即可得知字符匹配点 $(3,4)$ 的列号 4 位于这两个边界点之间，从而得出 $LCS(3,4) = LCS(2,1) + 1 = 2$。

之所以只需跟数目很少的边界点进行比较，是因为图 4.8所示的最优值表不仅具有稀疏性，还具有规整的形式，即：

观察 2：*最优值表中的每一行都可以划分成一系列等值区间；考虑字符匹配点 $(i+1,j)$，如果 (i,j) 位于第 i 行中值为 k 的等值区间的内部，则有 $LCS(i+1,j) = k+1$；如果位于该等值区间的边界，则有 $LCS(i+1,j) = k$。*

最优值表中的每一行、每一列都是单调递增的；正是这种单调递增性，使得我们可以把表中的每一行分成一系列的等值区间。为便于判断一个字符匹配点 (i,j) 是否在值为 k 的等值区间内部，我们定义等值区间的边界点（主要关注其列号）。详细地说，我们用 $T(i,k)$ 表示第 i 行、值为 k 的最左单元的列号 [?]，即：

$$T(i,k) = \begin{cases} \min\{j | OPT(i,j) = k\} & \text{如果} \exists j, OPT(i,j) = k \\ \infty & \text{否则.} \end{cases}$$

以图 4.8为例，我们有：$T(2,1) = 2, T(2,2) = 5, T(3,1) = 1, T(3,2) = 4$。

单从最优值表的一行来看，任何等值区间的最左侧边界点必然是字符匹配点。如果考查相邻两行的边界点的话，就会发现无论是边界点的位置还是边界点的值，都存在着紧密的联系。

我们首先来看边界点的位置：第 $i+1$ 行中值为 k 的等值区间的边界点，必定位于第 i 行的、值为 $k-1$ 的等值区间之中（仅看列号）。比如在图 4.8的第 2 行中，值为 1 的等值区间是 $[2,4]$；而第 3 行边界点 $(3,4)$ 的列号位于此区间之中。我们将此性质严格表述成下述引理 [?]：

引理 4.4.1. *$T(i+1,k)$ 必定位于第 i 行的两个边界点 $T(i,k-1)$ 与 $T(i,k)$ 之间，即：$T(i,k-1) < T(i+1,k) \le T(i,k)$.*

证明： 一方面，由 $T(i,k)$ 的定义可知：x 的前缀 $x_1 \cdots x_i$ 与 y 的前缀 $y_1 \cdots y_{T(i,k)}$ 之间存在一个长度为 k 的公共子序列；因此，更长的前缀 $x_1 \cdots x_{i+1}$ 与 $y_1 \cdots y_{T(i,k)}$

之间的最长公共子序列长度至少为 k，从而得出：

$$T(i+1,k) \le T(i,k).$$

另一方面，由 $T(i+1,k)$ 的定义可知：x 的前缀 $x_1 \cdots x_{i+1}$ 与 y 的前缀 $y_1 \cdots y_{T(i+1,k)}$ 之间存在一个长度为 k 的公共子序列（记为 $S_{i+1,k}$）。假如将这两个前缀都去除最后一个字符的话，至多导致公共子序列 $S_{i+1,k}$ 减少一个字符；这意味着前缀 $x_1 \cdots x_i$ 与 $y_1 \cdots y_{T(i+1,k)-1}$ 之间存在着一个长度至少为 $k-1$ 的公共子序列，从而得出：

$$T(i,k-1) \le T(i+1,k) - 1,$$

以及下述的严格不等式：

$$T(i,k-1) < T(i+1,k).$$

□

接下来，我们考查相邻两行边界点的值之间的关系。以图 4.8为例，第 3 行的边界点 $(3,4)$ 的列号是 4，位于第 2 行的值为 1 的等值区间内部，其值 $OPT(3,4)=2$。与之不同的是，第 6 行的边界点 $(6,2)$ 的列号是 2，位于第 5 行的值为 2 的等值区间的边界上，其值 $OPT(6,2)=2$。我们把这两种情形描述成下述定理：

定理 4.4.1. 令下标集合 $J_{i,k} = \{j | x_{i+1} = y_j \text{且} T(i,k-1) < j \le T(i,k)\}$，则有：

$$T(i+1,k) = \begin{cases} T(i,k) & \text{如果} J_{i,k} = \emptyset \\ \min\{j | j \in J_{i,k}\} & \text{否则}. \end{cases}$$

证明： 首先来看第一种情形：$T(i+1,k)$ 的最小性意味着前缀 $x_1 \cdots x_{i+1}$ 与 $y_1 \cdots y_{T(i+1,k)}$ 之间的最长公共子序列（记为 $S_{i+1,k}$）必定以 $y_{T(i+1,k)}$ 为最终字符。

当 $J_{i,k} = \emptyset$ 时，字符 x_{i+1} 与 $y_{T(i+1,k)}$ 不匹配；因而，假如我们把字符 x_{i+1} 去除之后，更短的前缀 $x_1 \cdots x_i$ 与 $y_1 \cdots y_{T(i+1,k)}$ 之间的最长公共子序列就是 S_k，故有：$T(i,k) = T(i+1,k)$。

接下来考虑第二种情形：当 $J_{i,k} \ne \emptyset$ 时，我们令 $j^* = \min\{j | j \in J_{i,k}\}$；由 $T(i,k-1)$ 的定义可知，前缀 $x_1 \cdots x_i$ 与 $y_1 \cdots y_{T(i,k-1)}$ 之间存在着长度为 $k-1$ 的公共子序列（记为 $S_{i,k-1}$）。因为 $j^* > T(i,k-1)$ 且 $x_{i+1} = y_{j^*}$，所以更长的前缀 $x_1 \cdots x_{i+1}$ 与 $y_1 \cdots y_{j^*}$ 之间必然存在一条长度为 k 的公共子序列（在 $S_{i,k-1}$ 后附加字符 x_{i+1}）。因此有：

$$T(i+1,k) \le j^*.$$

下面我们使用反证法证明 $T(i+1,k) = j^*$。假设 $T(i+1,k) < j^*$，我们考虑前缀 $x_1 \cdots x_{i+1}$ 与 $y_1 \cdots y_{T(i+1,k)}$ 之间的长度为 k 的最长公共子序列 $S_{i+1,k}$。由于 j^* 是与

x_{i+1} 匹配的字符的最小下标，依据我们的假设 $T(i+1,k) < j^*$，可知 $y_{T(i+1,k)}$ 必然与 x_{i+1} 不匹配。因此，$S_{i+1,k}$ 也是更短的前缀 $x_1 \cdots x_i$ 与 $y_1 \cdots y_{T(i+1,k)}$ 之间的公共子序列，故：

$$T(i,k) \leq T(i+1,k).$$

再由引理 4.4.1可知 $T(i+1,k) \leq T(i,k)$，故有：

$$T(i,k) = T(i+1,k).$$

再由 j^* 的定义可知 $j^* \leq T(i,k)$，进而推出 $j^* \leq T(i+1,k)$，而这与我们的假设 $T(i+1,k) < j^*$ 矛盾。

因此，当 $J_{i,k} \neq \emptyset$ 时，必然有：

$$T(i+1,k) = j^* = \min\{j | j \in I_{i+1,k}\}.$$

□

算法设计与描述

上述定理意味着对于字符匹配点 $(i+1,j)$ 来说，我们只需知道 j 位于第 i 行的某个等值区间（分内部和边界两种情形），即可立即求出 $OPT(i+1,j)$ 的值。基于上述分析，James W. Hunt 等设计了如下的高效稀疏动态规划算法 [**?**]：

Algorithm 39 求解最长公共子序列问题的 HUNT-SZYMANSKI 算法

function HUNT-SZYMANSKI(x, y)

```
1: MatchList =FIND-MATCH-LIST(x, y);
2: Set n = |y|, set T[0] = 0 and T[k] = ∞ for each 1 ≤ k ≤ n;
3: for i = 1 to m do
4:    for j in MatchList[i] (in decreasing order) do
5:       Find k such that T[k − 1] < j ≤ T[k];
6:       Set T[k] = j;
7:    end for
8: end for
9: return the largest k such that T[k] ≠ ∞;
```

这里的 FIND-MATCH-LIST 和 SPARSE-LCS 算法中的 FIND-MATHES 有相同的功能，也是寻找字符串 x, y 之间的所有字符匹配点；不同之处仅在于对结果的组织形式：FIND-MATCH-LIST 返回的是一个二维表，其中第 i 行记录所有与 x_i 匹配的字符

y_j 的列号（按逆序排列），例如：

$$\begin{aligned} MatchList[1] &= (9,7,6,3), \\ MatchList[2] &= (8,5,2), \\ MatchList[4] &= (). \end{aligned}$$

以字符串 x = "abcdbb" 和 y = "cbacbaaba" 为例，Hunt-Szymanski 算法的运行过程（前 10 步）见表 4.2。值得指出的是，为保证找到最左边界点，Hunt-Szymanski 算法是按照行从小到大、列从大到小的方式逐个检查字符匹配点。

表 4.2: Hunt-Szymanski 算法运行过程示例（前 10 步）

		边界点列号										
步骤	匹配点	$T[0]$	$T[1]$	$T[2]$	$T[3]$	$T[4]$	$T[5]$	$T[6]$	$T[7]$	$T[8]$	$T[9]$	值
初始	-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	-
1	$(1,9)$	0	9	∞	∞	∞	∞	∞	∞	∞	∞	1
2	$(1,7)$	0	7	∞	∞	∞	∞	∞	∞	∞	∞	1
3	$(1,6)$	0	6	∞	∞	∞	∞	∞	∞	∞	∞	1
4	$(1,3)$	0	3	∞	∞	∞	∞	∞	∞	∞	∞	1
5	$(2,8)$	0	3	8	∞	∞	∞	∞	∞	∞	∞	2
6	$(2,5)$	0	3	5	∞	∞	∞	∞	∞	∞	∞	2
7	$(2,2)$	0	2	5	∞	∞	∞	∞	∞	∞	∞	1
8	$(3,4)$	0	2	4	∞	∞	∞	∞	∞	∞	∞	2
9	$(3,1)$	0	1	4	∞	∞	∞	∞	∞	∞	∞	1

时间复杂度分析

在 Hunt-Szymanski 算法中，对于每一个字符匹配点，需要查找其所在的等值区间（第 6 行）。由于等值区域的边界点列号是递增的，故此步可以采用二分搜索技术完成，其时间复杂度是 $O(M \log n)$，此处 M 表示字符匹配点的数量。因此，整个算法的时间复杂度是 $O(m \log s + n \log s + M \log n)$。

值得强调指出的是：Hunt-Szymanski 之所以优于 Sparse-LCS 算法，其根源在于对于每一个字符匹配点来说，进行二分搜索时只会和很少的边界点进行比较。相比较而言，Sparse-LCS 算法对每一个字符匹配点，都与其之前的所有字符匹配点进行比较，导致较高的时间复杂度。

4.4.3 一些讨论

Eppstein

range 与解决 range 冲突

Linear / concave

数据结构：Boas 树

4.5 降低时间复杂度：基于最优解估计值的去冗余技术

对一些问题实例来说，在运行动态规划算法求解之前，我们往往可以对最优解的值做出比较准确的估计（或者说猜测）。以字符串 x = "OCCURRENCE" 和 y = "OCURRANCE" 为例，这两个字符串有 7 个相同的字母、6 个相同的 2-mer，因此可以合理地猜测它们的编辑距离比较小。

知道最优解的估计值，会给动态规划算法的执行带来非常大的益处：在运行动态规划算法时，只需计算 OPT 值表中那些优于估计值的单元；而对于那些确信劣于估计值的单元，则无需计算。这种去冗余技术将显著加快动态规划算法的执行过程。

还是以上述的字符串 x 和 y 为例，我们可以如下快速计算其编辑距离：首先猜测编辑距离 $d \leq 1$，然后检验此猜测是否成立。这里的检验可以通过运行快速的动态规划算法来完成。当然，我们对最优解值的猜测有可能是错误的；在这种情况下，后续检验步骤会报告“失败”，我们就增大编辑距离的估计值，并再次进行检验。如此重复，直至检验成功。

4.5.1 计算两序列编辑距离的 Ficket 算法

基于上述思想，J. W. Ficket 提出了一个高效的动态规划算法 [?]，其伪代码见算法 40。如图 4.9所示：(i) 当估计编辑距离不超过 $D = 1$ 时，算法计算出 OPT 表格中值不超过 1 的所有单元；然而在后续检验时，发现仅使用这些单元无法从 $OPT[10, 9]$ 回溯至 $OPT[0, 0]$，意味着估计值偏小；(ii) 因此，我们将估计值增大至 $D = 2$，重新进行计算；共计算出 33 个单元，经检验发现可以从 $OPT[10, 9]$ 回溯至 $OPT[0, 0]$，故算法结束。

这个算法的正确性在于：虽然仅计算了 OPT 表格中的部分单元，但是小于等于 D 的那些单元都可以准确计算；或者换句话说，那些大于 D 的单元的缺失，不会影

Algorithm 40 计算两序列编辑距离的 FICKET 算法

function CALCULATE-EDIT-DISTANCE-WITH-BOUND(x, y, D)

Set $m = |\mathrm{x}|$ and $n = |\mathrm{y}|$;
Set $OPT[i, 0] = i$ for $i = 0$ to m, set $OPT[0, j] = j$ for $j = 0$ to n, set the other entries as ∞;
for $i = 1$ to m **do**
    Set L_{i-1} as the minimal j such that $OPT[i-1, j] < D$;
    Set R_{i-1} as the minimal j such that $OPT[i-1, j] \geq D$ and $j > L_i$;
    $j = L_{i-1}$;
    repeat
        $OPT[i, j] = \min\{OPT[i-1, j-1] + s(x_i, y_j), OPT[i-1, j] + 1, OPT[i, j-1] + 1\}$;
        $j = j + 1$;
    until $j > R_{i-1}$ and $OPT[i, j] \geq D$;
end for

function FAST-EDIT-DISTANCE(x, y)

Set $D = 1$;
repeat
    CALCULATE-EDIT-DISTANCE-WITH-BOUND(x, y, D);
    $D = D * 2$;
until find a backtrack path from $OPT[m, n]$ to $OPT[0, 0]$;
return $OPT[m, n]$;

响小于等于 D 的那些单元的计算。以图 4.9 b 中的 $OPT[1, 4]$ 为例，依据递归表达式可得：

$$OPT[2, 4] = \min\{OPT[1, 3] + s(\texttt{'C'}, \texttt{'R'}), OPT[2, 3] + 1, OPT[1, 4] + 1\}.$$

虽然 $OPT[1, 4]$ 的确切值未计算出来，但是我们知道 $OPT[1, 4]$ 必定大于 D，不可能是三项之中最小者，因此忽略之后并不会影响 $OPT[2, 4]$ 的计算。

如果距离估计值比较准确而且比较小的话，FAST-EDIT-DISTANCE 算法只需计算少数单元，因此具有较好的实际性能。当然，在最坏的情况下，算法需要重复多次，不断增大距离估计值 D，甚至最后使得 $D = \min\{m, n\}$。在这种情况下，算法会计算出所有单元，因此其时间复杂度是 $O(mn)$。

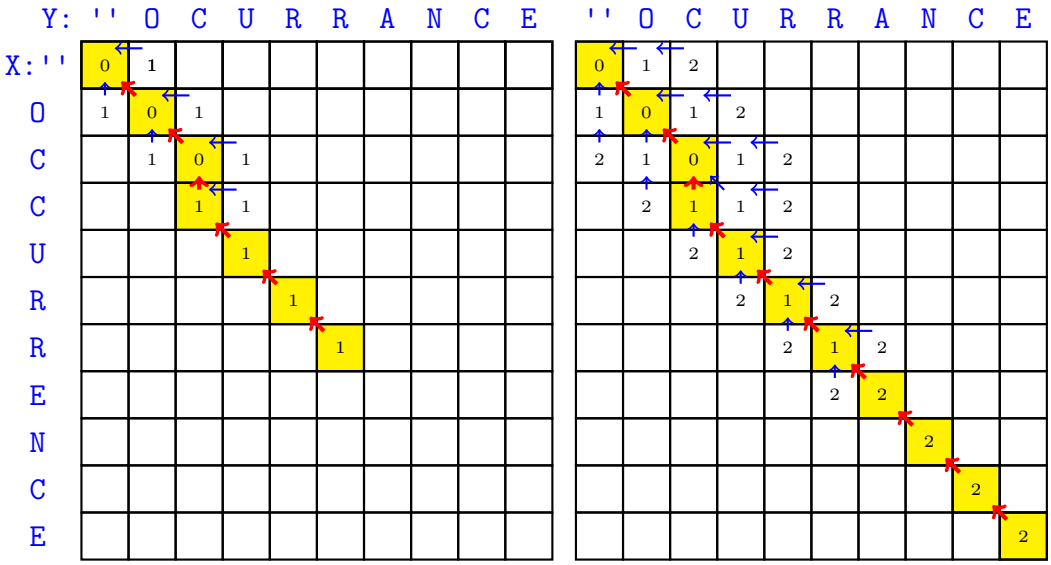


图 4.9: Fast-Edit-Distance 算法计算字符串 x = "OCCURRENCE" 和 y = "OCURRANCE" 编辑距离的运行过程。(*a*) 当估计编辑距离 $D = 1$ 时，无法从 $OPT[10, 9]$ 回溯至 $OPT[0, 0]$；(*b*) 当估计编辑距离 $D = 2$ 时，可以从 $OPT[10, 9]$ 回溯至 $OPT[0, 0]$

4.5.2 改进版本：条带型动态规划算法

如上一小节所述，在最坏情况下，Ficket 算法需要计算 OPT 值表中所有 mn 个单元。一种改进方式是控制需要计算的单元总数：只计算以主对角线为中心的、宽度为 α 的条带之内的单元；条带之外的单元，直接赋值为 ∞。

Algorithm 41 计算两序列编辑距离的条带型动态算法

function Banded-Edit-Distance(x, y, α)

1: Set $m = |\mathrm{x}|$ and $n = |\mathrm{y}|$;
2: Set $OPT[i, 0] = i$ for $i = 0$ to m, set $OPT[0, j] = j$ for $j = 0$ to n;
3: **for** each entry (i, j) within the α-wide band centered at the main diagnoal **do**
4: Set $L = OPT[i - 1, j] + 1$ if $(i - 1, j)$ within the α-wide band and $L = \infty$ otherwise;
5: Set $U = OPT[i, j - 1] + 1$ if $(i, j - 1)$ within the α-wide band and $U = \infty$ otherwise;
6: $OPT[i, j] = \min\{OPT[i - 1, j - 1] + s(x_i, y_j), L, U\}$;
7: **end for**
8: **return** $OPT[m, n]$;

改进后的算法见算法 41，其时间复杂度是 $O(\alpha \min\{m, n\})$。运行示例见图 4.10，其中条带宽度 $\alpha = 5$。值得指出的是，当 α 设置过小时，算出的 $OPT(i, j)$ 和原始的

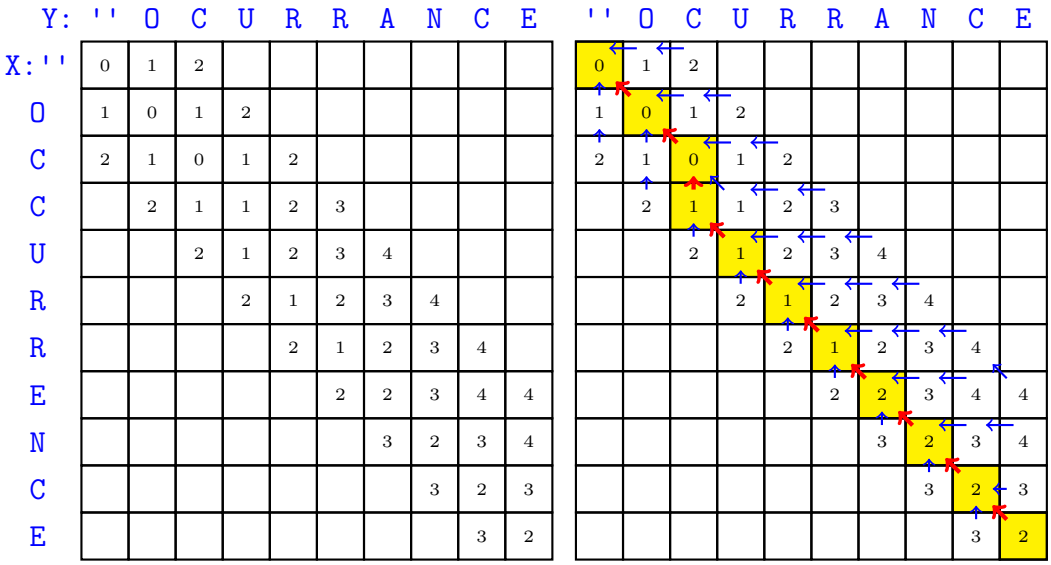


图 4.10: Banded-Edit-Distance 算法计算字符串 x = "OCCURRENCE" 和 y = "OCURRANCE" 编辑距离的运行过程。(*a*) *OPT* 表格填充过程；(*b*) 计算最优联配的回溯过程

Edit-Distance 算法会有差异；因此，在实践中，需要事先估算 $OPT(i,j)$ 的值，并据此设置合理的条带宽度 α。

4.6 降低时间复杂度：利用单调性的动态规划

WebProxy2000 (K-center), SMAWK

LARSCH? (Klawe1990, Galil1989)

4.7 降低时间复杂度：QI

Yao1980 BST, Knuth1971 BST, Hirschberg1985 LWS QUEUE Curves and Paragraph partition problem

KY and TM:

Golin2005 QI vs Monge：KY: 结果矩阵，上三角；SMAWK: nxm 矩阵，整个

延伸阅读

利用“拐角点”快速计算最长公共子序列

1977 年，D. Hirschberg 等提出了另一个利用稀疏性的动态规划算法，其基本思想非常简单：如图 4.11所示，字符串 x, y 之间的最长公共子序列长度是 4；与之相应地，我们将最优值表格划分成 4 个等值区域 [?]。因此，问题转化成计算等值区域的个数。

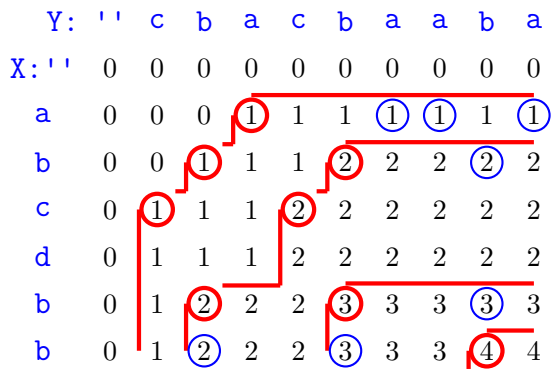


图 4.11: Hirschberg 提出的 LCS 算法的运行过程示例。图中红色线条表示最优解值的等值区域的边界线，红色圆圈表示边界线的“拐角点”（*dominant matches*[?]），蓝色圆圈表示除“拐角点”之外的字符匹配点

由于最优值表格中从左到右、从上到下都是递增的，因此等值区域完全由其边界线上的“拐角点”(Dominant matches) 刻画。拐角点的直观含义是：在其下方的字符匹配点必然在其左侧，而在其上方的字符匹配点必然在其右侧，其严格定义描述如下：

定义 4.7.1 (k-拐角点). 一个字符匹配点 (i, j) 称作 k-拐角点，如果 $OPT(i, j) = k$，且对于其他满足 $OPT(i', j') = k$ 的字符匹配点 (i', j') 来说，当 $i' > i$ 时，有 $j' \leq j$；当 $i' \leq i$ 时，有 $j' > j$。

在图 4.11中，拐角点以红色圆圈示出。注意拐角点都是边界点，但反之不必然成立。比如 $(6, 2)$ 是边界点，但并不是拐角点。

在 Hunt-Szymanski 算法中，按行号对字符匹配点建立索引；与之不同，Hischberg 的算法按字符对字符串 y 建立索引。当已知值为 $k-1$ 的等值区域时，Hischberg 的算法采用如下步骤计算值为 k 的等值区域：从上到下逐行扫描；在每一行，从右到左逐个检查字符匹配点；当找到一个字符匹配点，其列号位于上方拐角点的左侧时，即意味着找到一个拐角点；如此逐行进行，直至最后一行。

在构建值为 k 的等值区域的边界线时，虽然 Hischberg 算法是逐行扫描的，但是由于采用了按字符的索引方式，因此找到所有的 k-拐角点只需 $O(n)$ 的时间。Hischberg 算法的总时间是 $O(ln + n \log s)$，其中 l 表示字符串 x, y 的最长公共子序列的长度，$n \log s$ 来自于按字符对 y 建立索引的时间。

习题

1. Goad1982 作为习题

2. Web Proxy

3. LWS

4. Paragraph formatting

5. DTW for 10M-element vectors

第五章　贪心算法

5.1　引言

贪心算法（Greedy algorithm）和动态规划算法一样，也是主要用于求解优化问题，其典型应用场景之一是优化一个集合函数$f(S)$，即：给定全集 U，选择一个满足给定约束的子集 $S \subseteq U$，使得目标函数值 $f(S)$ 最小（或最大）。

如果是实数域上的优化问题，比如 $\min f(x), x \in \mathcal{R}$，我们很熟悉，也有很有效的梯度下降法，即：从一个初始解开始，计算梯度，沿着梯度方向走一定的步长，得到一个新的解；如此重复，直至梯度接近 0，从而最终得到一个局部最优解。

对于集合函数来说，首要的问题就是如何构造可行解。一个行之有效的思路是把这类问题的求解过程表示成多步决策过程，这可以从如下两个角度入手：

(1) “多选一”型决策过程：从 $S = \{\}$ 开始，每一步从满足约束条件的多个元素中选择一个元素添加入 S；

(2) “是/否”型决策过程：从 $S = \{\}$ 开始，按事先指定的顺序逐个考虑每个元素；每一步做“是/否”型决策，决定是否将此元素添加入 S。

对于一个优化问题来说，如果其求解过程能够描述成多步决策过程，并且满足最优子结构性质的话，通常既可以设计动态规划算法，又可以设计贪心算法进行求解。更深刻一些地说，在每一个贪心算法的背后，一般都有一个动态规划算法（尽管这个动态规划算法可能比较笨拙）[?]。

自然，贪心算法和动态规划也有着显著的不同：在多步决策过程的每一步，动态规划算法需要枚举所有可能的选择项，并递归求解与每个选择项对应的子问题；至于哪个选择项是最优的，只有等到所有子问题都求解完之后，通过回溯过程才能确定。与动态规划算法不同，贪心算法在面对多个决策选择项时，无需考虑子问题的解，而是使用贪心选择规则（Greedy-selection rule）直接确定出哪个决策选择项是最优选择

项（或“足够好”的选择项）。

直观地说，贪心算法无需考虑子问题的解即可做出最优或足够好的决策，因而是**短视的**（Near-sighted）；另一方面，在做决策时，贪心算法仅依赖于由已做出的决策构成的部分解，因而是**局部的**（Locally-optimal）。

贪心算法的核心是贪心选择规则。那么，贪心选择规则该如何设计呢？一般来说我们可以尝试下面三种设计策略：

(1) **直接针对优化目标设计贪心规则**：我们首先将求解过程描述成一个多步决策过程，然后对一个决策步，评估每个决策项对于达成整体优化目标的“贡献度”，进而设计贪心规则选择出贡献度最大的决策项。当然，这是一个“试错”过程（Trial-and-error），贡献度设计得是否合理，即：贡献度最大的决策项是否就是最优决策项，甚至多步决策过程设计得是否合适，都需要尝试和验证。

(2) **先设计动态规划算法，再简化成贪心算法**：和上一个策略不同，我们可以先设计动态规划算法，然后在一些具体实例上运行算法，以精确找出每一步的最优决策项；接下来我们可以采用两种方法：观察最优决策项的规律，进而设计能够选择出最优决策项的贪心规则；或者反过来，观察非最优决策项的规律，进而修改动态规划算法中的枚举决策项部分，避免对非最优决策项的枚举。这样我们就能将动态规划算法简化成贪心算法。

(3) **采用机器学习技术“学出”贪心规则**：上述两种策略都是单纯凭借人工经验设计贪心规则，是一个充满“灵感”的过程；找到有效的贪心规则具有一定“运气”的成分。最近的一些工作表明，我们可以应用机器学习技术，从大量的实例及其最优解中，学习出贪心决策规则，甚至直接使用训练后的神经网络做贪心决策。

虽然贪心算法是一个“短视”的算法，但是对一些问题依然能够求出最优解或足够好的解。那么，当问题具有哪些特性时，贪心算法能够求出最优解或足够好的解呢？这既依赖于问题的**目标函数**，又依赖于**约束条件**的性质；迄今已得到的结论是：

(1) 当目标函数 $f(S)$ 是**线性函数**，且满足约束的可行解 S 构成一个**拟阵**（Matroid）时（拟阵保证了最优决策的步数是固定值），可以证明贪心算法能够计算出最优解；

(2) 当目标函数是次模函数（Submodular function）时（次模函数可以直观理解为“以集合为定义域的凸函数”），可以证明贪心算法能够计算出足够好的解。

在本章里，我们首先介绍如何依据优化目标直接设计贪心选择规则，然后介绍如何将动态规划算法简化成贪心算法，并重点强调动态规划算法和贪心算法之间的密切联系，接下来介绍用机器学习和人工智能技术设计贪心算法的最新进展。在本章的最后，我们将介绍贪心算法背后的理论基础，包括拟阵和次模函数。

5.2 依据优化目标直接设计贪心规则：作业规划问题

我们考虑如下的作业规划问题：一位同学需要完成 7 份作业，其中作业 a 的截止时间是星期三晚上 24 点，完成得 3 分；作业 b 的截止时间是星期三晚上 24 点，完成得 1 分；其他作业的截止时间和分值见图 5.1。假设完成每份作业都需要 1 天的时间，因此至多只能选做 3 份作业。现在这位同学面临的问题是：选择哪些作业，并规划在哪天做，能够得到的总分最高?

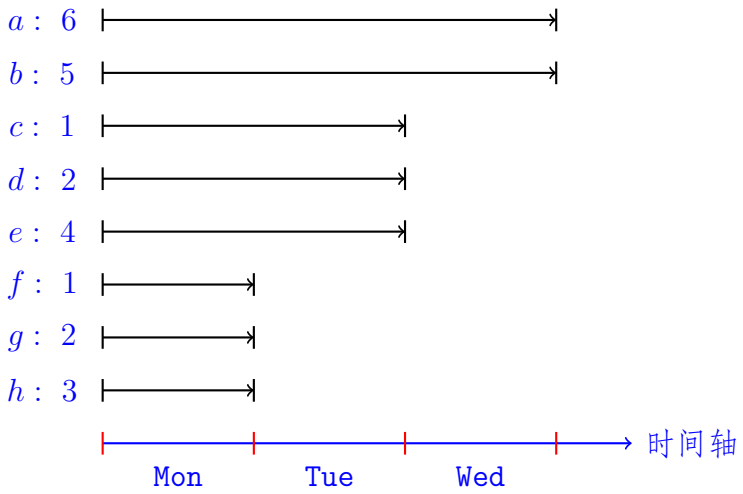


图 5.1: 作业规划问题的一个实例。图中左侧标明作业名称及分值，右侧箭头处表示截止时间。完成每项作业皆需用时 1 天

作业规划问题可形式化描述如下：

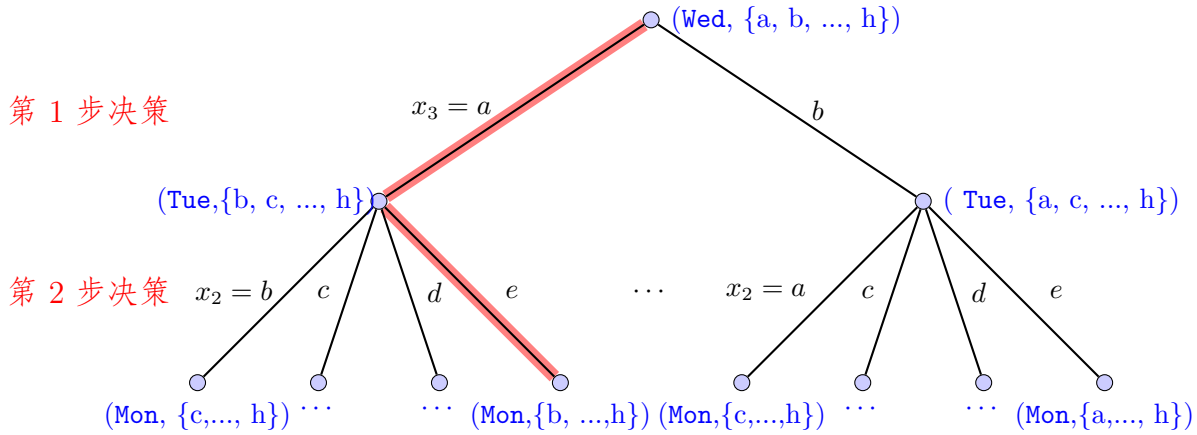


图 5.2: 求解作业规划问题的一种多步决策过程（此处仅列出前两步）。我们按照时间倒序考虑，即：第一步决策考虑星期三选择哪项作业，第二步决策考虑星期二选择哪项作业。我们采用“(当前时间，待完成作业)”的形式表示子问题；图中带阴影边表示按照贪心选择规则做出的最优决策

作业规划问题（Assignment-scheduling problem）

输入： n 项作业 $A = \{A_1, A_2, ..., A_n\}$；作业 A_i 的截止时间为第 D_i 天，分值记为 S_i；完成每项作业皆需 1 天的时间；

输出：选择 A 的一个子集 S，并确定 S 中每项作业的执行时间，使得 i) 各项作业执行时间互不冲突；ii) 每项作业在其截止时间之前完成；iii) 所选作业的总分值最大。

5.2.1 算法设计与描述

由于问题的解是所有作业 A 的一个子集，因此我们可以把求解过程描述成如下的“多选一”型多步决策过程：从最晚的截止时间开始，自后向前逐天进行，每天从多个可选的作业中选择一项作业（图 5.2）。以最后一天为例，我们有两个决策选项：

(1) 选做作业 a：获得分数 3，剩下的子问题是在前两天内、从 $\{b, c, ..., h\}$ 中选择作业，以最大化得分；

(2) 选做作业 b：获得分数 1，剩下的子问题是在前两天内、从 $\{a, c, ..., h\}$ 中选择作业，以最大化得分。

那这两个决策选项中，哪一个是最优决策项呢?

考虑到整体优化目标是“完成作业的总分最大”，因此在每一步决策时选择分值最大的作业（比如图 5.2所示第一步决策时选择作业 a），似乎有利于达到整体目标。我们把这个想法概括成这样的**贪心选择规则**（Greedy-selection rule）：*在当前可选的所有作业中选择分值最大的作业。*

事实上，使用这个贪心规则的确能够产生最优解，其严谨证明表述如下。

定理 5.2.1. *考虑作业集合 $A = \{A_1, A_2, \cdots, A_n\}$。令 $t = \max_{A_i \in A} D_i$ 表示最晚的截止时间，令 A_1 是截止时间最晚的作业中分值最大的，则必定存在一个包含 A_1 的最优解。*

证明. 我们采用**交换论证法**（Exchange argument）进行证明，即：假设有一个最优解 $S = \{A_{i_1}, A_{i_2}, \cdots, A_{i_k}\}$，但 S 不包括 A_1，我们试图证明可以构造出一个新的解 S'，S' 包含 A_1，且总分值大于等于 S。

在基于 S 构造 S' 时，我们区分两种情况：

(1) *S 中不存在截止时间为 t 的作业*：我们直接将 A_1 加入 S 以构造 S'，即：$S' = S \cup \{A_1\}$。

(2) *S 包含一个截止时间为 t 的作业*：令 A_t 表示 S 中截止时间为 t 的作业，我们先从 S 中去掉 A_t，再添加 A_1，即：$S' = S \backslash \{A_t\} \cup \{A_1\}$。

在这两种情况下，构造出的 S' 中的作业执行时间互不冲突，且其总分不会低于 S。 □

依据这个贪心规则设计的贪心算法如下：

Algorithm 42 求解作业规划问题的贪心算法

function GREEDY-FOR-ASSIGNMENT-SCHEDULING(A)

```
Initialize the selected assignments as S = {};
for t = max_{A_i ∈ A} D_i down to 1 do
    Set the candidate assignments C = {A_i | D_i ≥ t};
    Select from C the assignment with the largest score and add it to S;
end for
return S;
```

5.2.2 运行过程示例

以图 5.1所示作业为例，Greedy-for-Assignment-Scheduling 算法从最后截止时间（星期三）开始逆序执行：

(1) 在星期三，有 2 个作业 a, b 可以安排，其中分值 a 最高，因此选择 a 并安排在星期三执行；

(2) 在星期二，有 4 个作业 b, c, d, e 可以安排，其中 b 分值最高，因此选择 b 并安排在星期二执行；

(3) 在星期一，有 6 个作业 c, d, e, f, g, h 可以安排，其中 e 分值最高，因此选择 e 并安排在星期一执行。

最终算法选择了 3 项作业 a, b, e，总分值为 15。

5.2.3 时间复杂度分析

算法中的循环共执行 n 轮，每轮需要在当前可选作业中找分值最大的作业。我们可以使用数组保存当前可选的作业，则时间复杂度为 $O(n^2)$；采用堆保存当前可选的作业，则可将时间复杂度降低至 $O(n \log n)$。

5.2.4 一些讨论

作业规划问题有多种多步决策过程，我们来尝试一下其他方案，看是否可行：

(1) “是/否”型多步决策：如果我们按照分值从大到小的顺序逐个考虑各项作业，并优先执行分值最大的作业，这能否得到最优解呢？以图 5.1所示作业为例，采用这种策略，星期一安排了作业 a 之后，无法再安排 h，导致最终选择作业 a, b，总分值是 11，不是最优解。

(2) 从最早截止时间开始、从前往后逐天考虑各项作业：按照这种策略，有可能最终会选择作业 h, e, a，总分值是 13，也不是最优解。

总结一下，在想到正确的多步决策过程和贪心选择规则之前，我们往往要经历“尝试—评价—修正”过程；换句话说，贪心算法的设计通常是一个典型的“试错”（Trial and error）过程。

5.3 依据优化目标直接设计贪心规则：最短路径问题

在第三章里，我们描述了一般有向图上的单源最短路径问题，以及适用于无负圈情形的 Bellman-Ford 动态规划算法；我们还了解到如下两个事实：

(1) 即便指定了目的结点，由于无法预知经过哪些中间结点才能最快地到达目的结点，因此在实际操作中，通常需要计算出从源结点到所有结点的最短路径；

(2) 如果把从源结点 s 到所有结点的最短路径都找出来，会构成一棵以 s 为根的树，称作**最短路径树**。

因此，我们可以把单源最短路径问题的求解目标描述成**求一棵以 s 为根节点的树，使得从 s 出发、沿着这棵树到达每个结点的距离都最短**，并把求解过程描述成“从 s 起始”的多步决策过程，即：初始时，最短路径树仅包含 s 一个结点（作为根节点）；在每个决策步，选择与已有部分树的一条邻接边加入树中，从而不断扩展树直至覆盖所有结点（示例见图 5.3）。

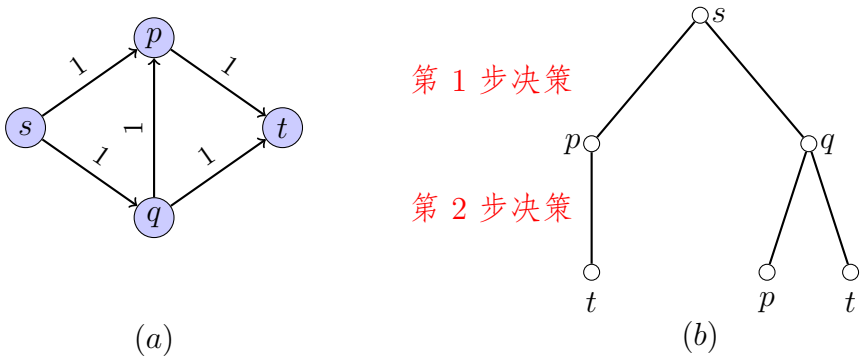


图 5.3: 计算单源最短路径的“从 s 起始”的多步决策过程（此处仅示出前两步决策）。(a) 无负边有向图示例；(b) “从 s 起始”的多步决策过程，第一步从 s 的邻接边中选择一条边加入最短路径树，第二步从当前部分树的邻接边中选择一条边加入树中

那为了达到生成最短路径树这个整体目标，在每一个决策步该如何决策呢?

最理想的决策是这样的：每一步新加一条边，使得沿着这条边是到达此边另一端点的最短路径。对于一般的有向图来说，每一步决策时不易确定应该增加哪条边；但是对于无负边的图来说，这是可以立即确定出来的，详细过程描述如下。

5.3.1 算法设计与描述

我们从无负边图的最简单情形—所有边的权重都是 1 的有向图—详细讲起。

对这种特殊情形来说，从 s 到任意结点的路径长度就是边数，距离都是整数，比如：1，2，3，...。因此，我们可以采用“逐步尝试距离”的策略计算最短路径，即：设一个距离阈值 r，将阈值 r 从 1 开始逐步增大，每一步枚举到 s 的距离小于 r 的那些结点。

图 5.4展示一个使用这种方法计算最短路径的实例。从 s 出发，一步可达 p, q 两个结点；从 p, q 再多走一步，就是从 s 两步可达的结点，即：p, t。注意 p 已经访问过了，所以从 s 到达 p 的最短距离不是 2，而是 1。

直观上看，就像是以 s 为中心的一个水波，不断向外扩展，逐步画出半径为 1, 2, 3, ... 的圆，看水波先到达哪个结点。当然了，对于某个结点 v 来说，有可能多次被枚举到；而第一次枚举到 v 时的阈值 T，就是从 s 到 v 的最短距离（值得强调的是，对于有负边的图，距离有可能是负值，因此这种从小到大尝试距离的方法不适用）。

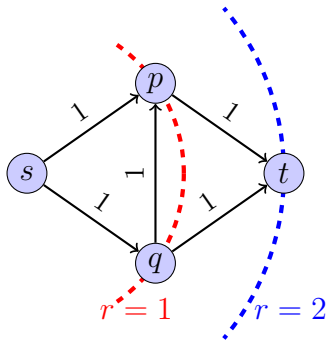


图 5.4: 对边长全为 1 的有向图，应用 BFS 算法计算单源最短路径的运行过程示例。直观地看，就像从 s 发出一个水波，当半径等于 1 时，到达 p, q；当半径等于 2 时，到达 p, t。因此，从 s 到 p, q 的最短距离是 1，到 q 的最短距离是 2

采用上述思想的算法就是宽度优先搜索算法（Breadth-first search, BFS），伪代码见算法 43，其中每个结点 v 都设置一个布尔型的标记 $v.visited$，用来表示该结点是否已被访问过，而“先入先出”队列 Q 则保存那些有待扩展的结点。

对于边长全是 1 的最简单情形我们会做了，接下来我们扩展一下：假如图中的边长不全是 1，又该如何计算最短路径呢？

一种自然的想法是“显式地添加新结点，把边长不全是 1 的图转化成边长全是 1 的图”。以图 5.6为例，边 $s \to q$ 的长度为 2，我们就在 s 和 q 的正中间添加一个新的结点 m，使得新的边 $s \to m$ 和 $m \to q$ 长度都是 1；然后对这个新的图运行 BFS 算法就可以了。

这种方法很容易理解，不过有两个缺陷：i) 如果边长很大的话，比如 1000，需要

Algorithm 43 对边的权重全是 1 的图计算最短路径的宽度优先搜索算法

function BFS($G = \langle V, E\rangle, s$)

```
1: Set v.visited =FALSE for each node v ∈ V, and set d(s) = 0, s.visited =TRUE;
2: Create an empty queue Q and execute Q.ENQUEUE(s);
3: while Q.ISEMPTY() ≠ TRUE do
4:     u = Q.DEQUEUE();
5:     for all u's neighbor v do
6:         if v.visited ≠ TRUE then
7:             Set v.visited = TRUE;
8:             Set d(v) = d(u) + 1;
9:             Q.ENQUEUE(v);
10:        end if
11:    end for
12: end while
```

添加 999 个结点，这样算法会很复杂；*ii*) 如果边长不是整数，比如是个小数或无理数 3.1415926...，无法插入结点。

为了克服上述两个缺陷，我们抛弃“显式插入新结点”方法，改用“隐式插入新结点”的思路，即：假设插入了新的结点，但只关心原有结点何时被搜索到。我们能够证明：从已被搜索到的结点再多走一步，可以到达的邻接点中的最近者的最短距离就确定了。注意这和 BFS 算法的区别：在 BFS 算法中，对那些与已被搜索到的结点邻接的顶点来说，其最短距离皆可直接确定；而现在不是所有邻接点，只有最近的邻接点的最短距离才能被确定下来。

为了严格地表述“邻接点中的最近者”，我们引入下面的定义：

定义 5.3.1 (最短距离的上界、已探索结点集、邻接点中的最近者)**.** 考虑无负边有向图 G。令 $d(u)$ 表示从起始点 s 到顶点 u 的最短距离，S 表示距 s 的最短距离已知的顶点集合，称为“已探索结点集”(*Explored nodes*)。对于不在 S 中的、但与 S 中顶点相邻的顶点 v，定义 $\overline{d(v)} = \min_{u\in S}\{d(u) + w(u, v)\}$，是最短距离 $d(v)$ 的上界；而最短距离的上界最小的邻接点，称为邻接点中的最近者。

粗略地讲，最短距离 $d(v)$ 的上界就是最短距离的估计值。如图 5.6所示，当第一步决策选择边 $\langle s, p\rangle$ 之后，我们更新“已探索结点”集合 $S = \{s, p\}$，最短距离 $d(s) =$

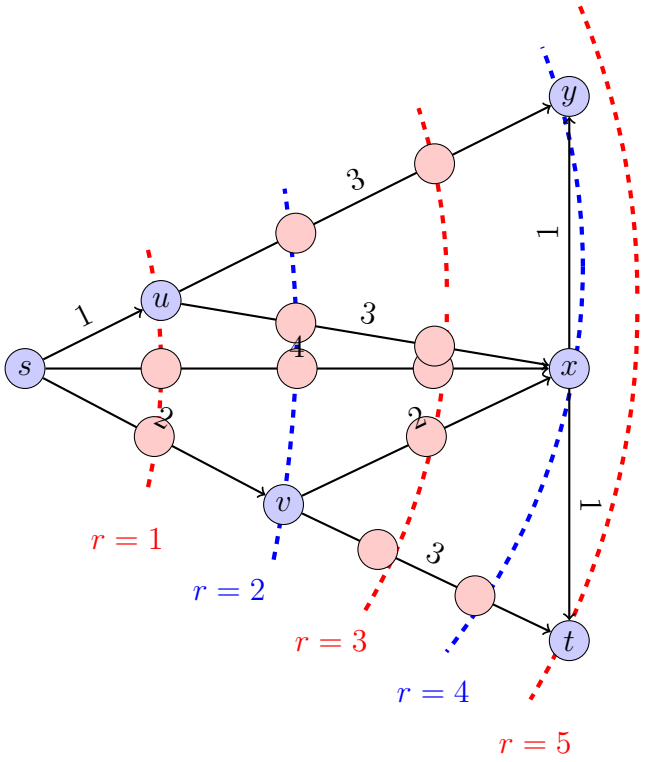


图 5.5:

0, $d(p) = 1$；对于与 S 相邻的顶点 x, y, q，最短距离的上界计算如下：

$$\overline{d(y)} = d(p) + 3 = 4,$$
$$\overline{d(x)} = \min\{d(p) + 1, d(s) + 4\} = 2,$$
$$\overline{d(q)} = d(s) + 2 = 2.$$

其中结点 x, q 的上界最小，是 S 所有邻接点中的最近者。

采用这个定义，我们描述并证明下面的贪心选择性质：

定理 5.3.1. *考虑无负边有向图 G。令 S 表示距 s 的最短距离已知的顶点集合。设 v^* 是与 S 相邻的顶点中最短距离上界最小的顶点，即：$v^* = \operatorname{argmin}_{v \notin S}\{\overline{d(v)}\}$，则从 s 到 v^* 的最短距离 $d(v^*)$ 必定是 $\overline{d(v^*)}$。*

证明： 我们仅需证明对于任意的路径 $P = s \rightsquigarrow v^*$，都有 $|P| \geq \overline{d(v^*)}$。

如图 5.7所示，设路径 P 上第一个不在 S 中的顶点为 q，即：P 可表示为 $P =$

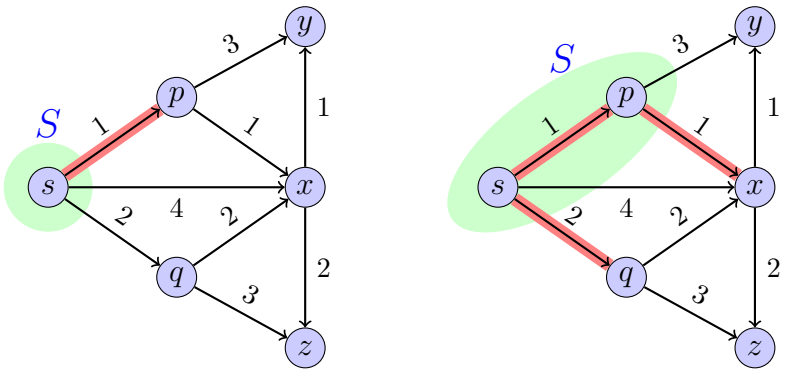


图 5.6: 一个边长不全为 1 的有向图实例，以及最短路径计算过程。直观地看，就像从 s 发出一个水波，当半径等于 1 时，水波到达结点 p。此时，“已探索结点”集合 $S = \{s, p\}$，且最短距离 $d(s) = 0$, $d(p) = 1$；对于与 S 相邻的顶点 x, y, q，最短距离的上界计算如下：$\overline{d(y)} = d(p) + 3 = 4$，$\overline{d(x)} = \min\{d(s) + 4, d(p) + 1\} = 2$，$\overline{d(q)} = d(s) + 2 = 2$，其中最近者 x, q 的最短距离也可确定。直观上看，当水波的半径等于 2 时，水波到达 x, q

$s \rightsquigarrow q \rightsquigarrow v^*$，则有：

$$
\begin{aligned}
|P| &= |s \rightsquigarrow q| + |q \rightsquigarrow v^*| && (5.3.1)\\
&\geq |s \rightsquigarrow q| && (5.3.2)\\
&\geq \overline{d(v^*)} && (5.3.3)
\end{aligned}
$$

其中不等式 (5.3.2) 成立是由于边的非负性，而不等式 (5.3.3) 成立是由于 v^* 是与 S 相邻的顶点中最短距离上界最小的顶点。 □

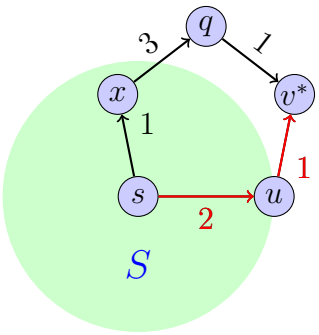


图 5.7: Dijkstra 算法使用的贪心规则正确性证明示意图。当扩展最短路径树时，在与 S 相邻的结点中，选择最短距离上界 $\overline{d(v)}$ 最小的结点 v^*，到达 v^* 的另一条路径 $s \to x \to q \to v^*$ 的长度是 5，比 $s \to u \to v^*$ 更长

Algorithm 44 计算无负边有向图上单源最短路径的 Dijkstra 算法

function Dijkstra($G = \langle V, E\rangle, s$)

1: Set $d(s) = 0$, and set $d(v) = +\infty$ for each node $v \neq s$;
2: Set $S = \{\}$; //Let S be the set of explored nodes;
3: **while** $S \neq V$ **do**
4: Select the unexplored node v^* ($v^* \notin S$) that minimizes $d(v)$;
5: $S = S \cup \{v^*\}$;
6: **for** each unexplored node v adjacent to an explored node **do**
7: $d(v) = \min\{d(v), \min_{u \in S}\{d(u) + d(u, v)\}\}$;
8: **end for**
9: **end while**

采用上述贪心规则确定下一步到达哪个结点的算法就是 Dijkstra 算法，也被称为*加权的宽度优先搜索*算法（Weighted BFS）。概而言之，Dijkstra 算法不断重复“*找 S 的最近邻居并加入 S、依据新加入结点更新其他结点的最短距离估计值*”两个步骤，直至计算出所有结点的最短路径（伪代码见*算法* 44）。

5.3.2 运行过程示例

对于图 5.6所示实例，Dijkstra 算法的运行过程见表 5.1。

表 5.1: Dijkstra 算法运行过程示例

步骤	已探索结点集 S	顶点的最短距离的上界 p	q	x	y	z	S 的最近邻居
1	$\{s\}$	1	2	4	∞	∞	p
2	$\{s, p\}$	1	2	2	4	∞	q, x
3	$\{s, p, q, x\}$	1	2	2	3	4	y
4	$\{s, p, q, x, y\}$	1	2	2	3	4	z
5	$\{s, p, q, x, y, z\}$	1	2	2	3	4	-

5.3.3 时间复杂度分析

与我们过去见到的算法不同，Dijkstra 算法的时间复杂度依赖于其关键步骤的具体实现方式。详细地说，Dijkstra 算法中关键步骤是找 S 的最近邻顶点，这需要

在所有尚待探索结点中找估计值最小的结点。优先队列（Priority quene）是一种支持“在一些动态变化的数中找最小数”的数据结构，包括如下基本操作：

(1) INSERT：向优先队列中插入一个数；

(2) EXTRACTMIN：从优先队列存储的数中找出最小数；

(3) DECREASEKEY：更改优先队列中已保存的数。

采用优先队列描述的 DIJKSTRA 算法见算法 45，其执行过程见课程网站上的演示。

优先队列有多种实现方式，比如用数组实现、用二叉树实现、用二项式堆实现；在 1984 年，Robert Tarjan 等为了加速 DIJKSTRA，专门设计了一个新的数据结构—Fibonacci 堆（详见附录一）。用不同实现方式的优先队列，DIJKSTRA 算法的时间复杂度也各不相同，总结于表 5.2。

Algorithm 45 用优先队列实现的 DIJKSTRA 算法

function DIJKSTRA($G = \langle V, E\rangle, s$)

Set $key(s) = 0$ and set $key(v) = +\infty$ for each node $v \neq s$; //$key(v)$ stores $\overline{d(v)}$, an upper bound of the shortest distance from s to v;
for all node $v \in V$ **do**
  PQ. INSERT (v) //n times in total
end for
Set $S = \{\}$;
while $S \neq V$ **do**
  $v^* = PQ$. EXTRACTMIN(); //n times in total
  $S = S \cup \{v^*\}$;
  for all $v \notin S$ and $\langle v^*, v\rangle \in E$ **do**
    if $key(v^*) + d(v^*, v) < key(v)$ **then**
      PQ.DECREASEKEY($v, key(v^*) + d(v^*, v)$); //m times in total
    end if
  end for
end while

	优先队列的实现方式			
基本操作	数组/链表	二叉堆	二项式堆	Fibonacci 堆
MakeHeap	1	1	1	1
Insert	1	$\log n$	$\log n$	1
ExtractMin	n	$\log n$	$\log n$	$\log n$
DecreaseKey	1	$\log n$	$\log n$	1
Delete	n	$\log n$	$\log n$	$\log n$
Union	1	n	$\log n$	1
FindMin	n	1	$\log n$	1
Dijkstra 算法	$O(n^2)$	$O(m \log n)$	$O(m \log n)$	$O(m + n \log n)$

表 5.2: Dijkstra 单源最短路径算法的时间复杂度。Dijkstra 算法共执行 n 次 Insert、n 次 ExtractMin、m 次 DecreaseKey；当采用不同的优先队列实现方式时，算法的时间复杂度也不相同

5.3.4 一些讨论

Bellman-Ford 算法与 Dijkstra 算法的对比

对于无负边的有向图来说，Dijkstra 算法比 Bellman-Ford 算法更高效，但这并不意味着 Bellman-Ford 算法毫无优势。

事实上，从算法所需信息的角度来看，Dijkstra 算法需要维护“已探索结点集合 S”；而在 Bellman-Ford 算法中，每个结点只需依据其邻居对最短距离的估计值即可修正自己的估计值。因此，Bellman-Ford 算法可以改造成由各个结点异步并发执行的算法，尤其适用于结点和边频繁动态变化的情形。比如，在 Internet 上的路由协议 RIP 中，每个路由器上运行的就是 Bellman-Ford 算法，当其发现一条到达目的地的更优路径时，就发送消息给其邻居，以提醒邻居更新信息，这样即可获得从源 IP 地址到目的 IP 地址的最快路径 [?]。

队列 vs. 优先队列

在 BFS 算法中，我们只需要按照结点的加入顺序逐个考虑，逐个扩展其邻居结点，因此使用队列保存有待扩展的结点即可；但是在 Dijkstra 算法（或者说“加权

的 BFS 算法”）中，我们不能按照结点的加入顺序，而需要按照最短距离的估计值进行排序，因此需要使用优先队列，其中的优先级是指最短距离的估计值。

5.4 将动态规划简化成贪心算法：区间调度问题

区间调度问题的一个典型例子是学校常常会遇到的课程安排问题：学校有一间教室，有 n 门课程 $A_1, A_2, \cdots, A_n$ 申请使用这间教室；课程 A_i 向教务处申明课程起止时间和修课人数 W_i（实例见图 5.8）。教务处面临的问题是：选择哪些课程使用这间教室，使得上课的学生总数最多?

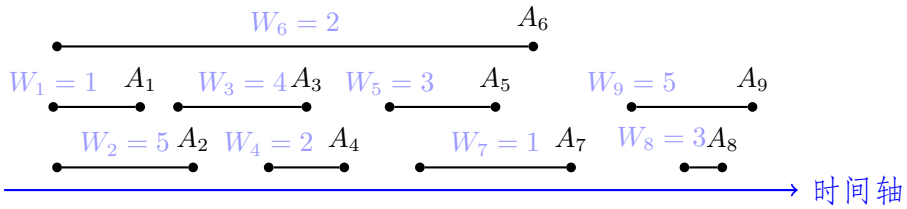


图 5.8: 课程安排问题实例

由于只有一间教室，因此所选择课程的时间区间相互不能冲突。在图 5.8所示实例中，最优解是 $S = \{A_2, A_4, A_5, A_9\}$，共有 15 名学生上课。

区间调度问题可形式化描述如下：

> **区间调度问题（Interval-scheduling problem）**
> **输入**： n 个活动 $A = \{A_1, A_2, ..., A_n\}$；活动 A_i 的开始时间记为 S_i，结束时间记为 F_i；如果选择活动 A_i，则获得收益 W_i；
> **输出**：选择一些时间区间互不冲突的活动，使得总收益最大。

此处我们假设所有的活动已经按照结束时间升序排列，即：对于活动 A_i 和 A_j 来说，如果 $i < j$，则有 $F_i \leq F_j$。之所以进行如此排序，完全是为了子问题表示的方便；我们稍后做详细介绍。

5.4.1 求解区间调度问题的动态规划算法

算法设计及描述

由于问题的解是 A 的一个子集，因此我们可以把求解过程描述成如下的“是/否”型多步决策过程：从最后一个活动 A_n 开始，逐个考虑每个活动 A_i，每次都在两个选

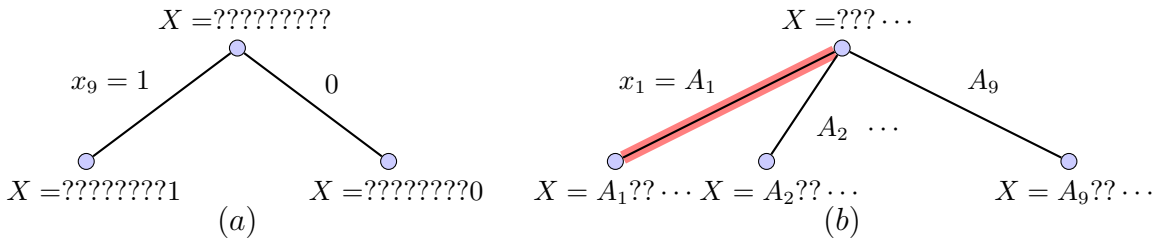


图 5.9: 课程安排问题的解的两种表达形式以及相应的多步决策过程（此处仅示出第一步决策）。(a) “是/否”型多步决策：按一定顺序逐个考虑各个活动；每一步决策决定是否选择一个活动；(b) “多选一”型多步决策：每一步都从剩余的可选活动中选择一个活动。动态规划算法枚举所有决策项，而贪心算法直接选择课程 A_1

择项中进行决策（图 5.9a）。以活动 A_n 为例，两个选择项描述如下：

(1) 选择活动 A_n：如果选择了活动 A_n，则所有与 A_n 时间冲突的活动都不能再考虑。由于事先已将所有的活动按照结束时间升序排列，因此剩余的可选活动可以表示成 $A_1, \cdots, A_{pre(n)}$，其中 $pre(n)$ 表示与 A_n 不冲突的最后一个活动的下标。有待求解的子问题是：从 $A_1, \cdots, A_{pre(n)}$ 中选择一些不冲突的活动，使得总收益最大。

(2) 不选择活动 A_n：如果不选择活动 A_n，则剩余的可选活动是 $A_1, \cdots, A_{n-1}$；有待求解的子问题是从 $A_1, \cdots, A_{n-1}$ 中选择一些不冲突的活动，使得总收益最大。

综合上述两种情形，我们可以把子问题的一般形式表示成：从 $A_1, \cdots, A_i$ 中选择不冲突的活动，使得总收益最大。最大收益（记为 $OPT(i)$）满足下面的递归关系和基始条件：

$$OPT(i) = \begin{cases} 0 & \text{如果} i = 0 \\ \max\{OPT(pre(i)) + W_i, OPT(i-1)\} & \text{否则} \end{cases}$$

我们通过调用 DP-FOR-IS(n) 即可求解原始问题；由于只有 n 个子问题，且每个子问题只有两个决策项可供选择，因此上述算法的时间复杂度是 $O(n)$ 的。

除了上述“是/否”型多步决策之外，我们还可以将求解过程描述成“多选一”型多步决策：首先将问题的解表示成 $X = [x_1, x_2, \cdots]$，其中 $x_i \in \{A_1, A_2, \cdots, A_9\}$；求解过程可以描述成如下的多步决策过程：*每次决策是从当前可选活动中挑选一个*。每选择一个活动 A_i 之后，就从当前可选活动中删除 A_i 以及与 A_i 冲突的所有活动。这两种思路的对比见图 5.9。

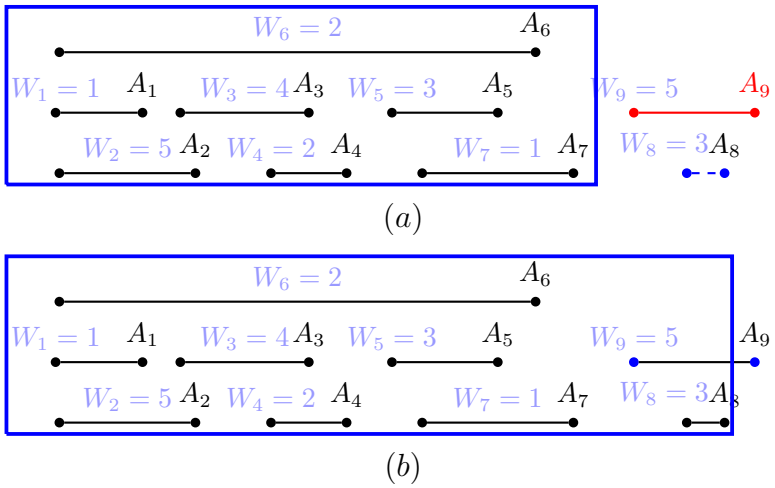


图 5.10: 课程安排问题中活动 A_n 的两种决策选项。(a) 选择 A_9，则需排除与 A_9 冲突的活动 A_8，剩余的子问题是从 $A_1, \cdots, A_7$ 中做出最优选择；(b) 不选 A_9，则剩余的子问题是从 $A_1, \cdots, A_8$ 中做出最优选择

Algorithm 46 求解区间调度问题的动态规划算法

function DP-FOR-IS(i)

Require: All activities have been sorted in the increasing order of finishing time;

1: **if** $i \leq 0$ **then**
2: **return** 0;
3: **end if**
4: Let $pre(i)$ denote the final activity that ends before S_i;
5: **return** $\max\{\text{DP-FOR-IS}(pre(i)) + W_i, \text{DP-FOR-IS}(i-1)\}$;

采用第二种思路，我们可以将子问题描述成从活动集合 S 中选择收益最大的互不冲突子集，将其收益记为 $OPT(S)$，则可得到如下的递归表达式：

$$OPT(S) = \max_{A_i \in S}\{OPT(Remove(S, A_i)) + W_i\},$$

此处的 $Remove(S, A_i)$ 表示从 S 中去除 A_i 以及与其冲突的活动之后的剩余活动。相应的动态规划算法伪代码见算法 47，其时间复杂度是 $O(2^n)$。

相对于第一个算法而言，第二个算法比较“笨拙”；我们之所以还描述这个算法，是因为它与将要谈到的贪心算法有着紧密的联系。

5.4.2 区间调度问题的特殊情形及贪心求解算法

下面我们来看课程安排问题的特殊情形：每门课程仅有一位学生；这等价于最大化不冲突的课程数目。我们将这种特殊情形形式化描述如下：

Algorithm 47 求解区间调度问题的“笨拙”动态规划算法

function AWKWARD-DP-FOR-IS(S)

```
if S is empty, return 0;
m = +∞;
for each activity A_i ∈ S do
    if DP2-FOR-IS(Remove(S, A_i)) + W_i < m then
        m = DP2-FOR-IS(Remove(S, A_i)) + W_i;
    end if
end for
return m;
```

区间调度问题的特殊情形

输入：n 个活动 $A_1, A_2, ..., A_n$；活动 A_i 的开始时间记为 S_i，结束时间记为 F_i；

输出：选择一些时间区间互不冲突的活动，使得活动总数最大。

算法设计及描述

既然是区间调度问题的特殊情形，那么直接应用上一小节的两个动态规划算法都可以进行求解；然而这种特殊情形具有一些特殊性质，使得“多选一”型多步决策过程可以大大简化，从而动态规划算法 AWKWARD-DP-FOR-IS 可以简化成一个更高效的贪心算法（经尝试，“是/否”型多步决策过程不易于简化）。

我们首先在一些简单实例上运行动态规划算法 AWKWARD-DP-FOR-IS，观察最优决策有何规律。对图 5.11(a) 所示实例来说，每个活动的权重都是 1，因此动态规划算法所依据的递归表达式退化成：

$$OPT(\{A_1, A_2, A_3\}) = \max \begin{cases} OPT(\{A_3\}) + 1 \\ OPT(\{\}) + 1 \\ OPT(\{A_1\}) + 1 \end{cases} = 2.$$

因为 $OPT(\{A_3\}) > OPT(\{\})$ 且 $OPT(\{A_1\}) > OPT(\{\})$，所以第一步的最优决策是选择 A_1 或 A_3。再以图 5.11(b) 所示实例为例，第一步的最优决策是选择 A_1 或 A_2。

这两个实例给我们带来如下启示：对于第一个决策来说，“最早下课”的课程（如图 5.11(a) 中的 A_1）和“最晚上课”的课程（如图 5.11(a) 中的 A_3）都是最优决策。事实上，这个规律对所有实例都适用，其严谨证明陈述如下。

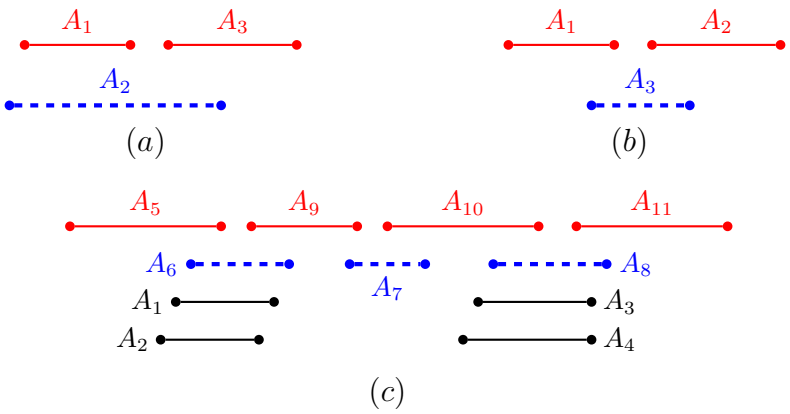


图 5.11: 课程安排问题的 3 个实例。(a) 采用“选择最早上课的课程”规则，只能安排一门课程，但最优解值为 2；(b) 采用“选择时长最短课程”规则，只能安排一门课程，但最优解值为 2；(c) 采用“选择冲突最少课程”规则，只能安排 3 门课程，但最优解值为 4。图中虚线表示按照贪心规则选择出的课程

定理 5.4.1. *考虑活动集合 $A = \{A_1, A_2, \cdots, A_n\}$。设 A_1 是所有活动中结束时间最早的，则 A_1 是第一步决策的最优决策，即：必定存在一个最优解，其中包含活动 A_1。*

证明： 我们采用*交换论证法*（Exchange argument）进行证明。

假如存在一个互不冲突的最大活动子集 O，但是 O 不包含 A_1。我们执行如下操作获得一个新的子集 O'：去除 O 中最早结束的活动（记为 A_{O1}），然后添加 A_1，即：$O' = O - \{A_{O1}\} \cup \{A_1\}$（见图 5.12）。可以证明 O' 具有如下性质：

(1) O' 中的所有活动互不冲突：因为 A_{O1} 与 O 中其他活动不冲突，而 A_1 比 A_{O1} 结束更早，因此 A_1 也不与其他活动冲突；

(2) $|O'| = |O|$。

因此 O' 是一个包含 A_1 的最优解。 □

依据上述定理，我们可以设计如下的*贪心选择规则*（Greedy-selection rule）：*每次从可选的活动中选择结束时间最早的活动*，并进而设计出如下的贪心算法（见算法 48）。

运行过程示例

图 5.13示出应用 GREEDY-FOR-IS 算法求解课程安排问题的运行过程。从图中可以看出，贪心算法只需一趟扫描即可求解，其时间复杂度是 $O(n)$。

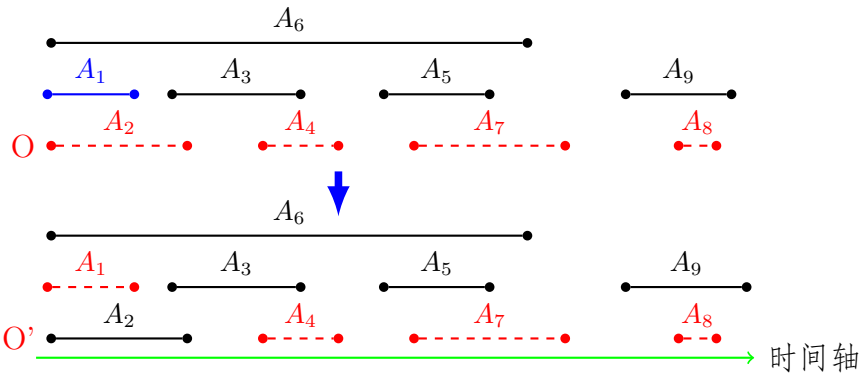


图 5.12: 课程安排问题的贪心选择性质。从最优解 O 中去除最早结束的活动 A_2，然后添加 A_1，则可获得一个包含 A_1 的最优解 O'

Algorithm 48 求解区间调度问题的贪心算法

GREEDY-FOR-IS(A)

Require: All activities have been sorted in the increasing order of finishing time;

```
previous_finish_time = −∞;
O = {};
for i = 1 to |A| do
    if S_i > previous_finish_time then
        O = O ∪ {A_i};
        previous_finish_time = F_i;
    end if
end for
return O;
```

值得指出的是：GREEDY-FOR-IS 贪心算法实质上是动态规划算法 AWKWARD-DP-FOR-IS 的简化。具体地说，图 5.9(b) 所示的第一步决策共有 9 种可能的决策选项；在一般情形下，我们无法事先确定哪个选项是最优的，因此不得不枚举所有的决策项；然而对于上述特殊情形来说，无需枚举所有决策项，即可直接断言 A_1 必定是最优决策项。

从这个例子，我们可以得到如下几点观察：

(1) 由于避免了动态规划算法中的回溯过程，贪心算法更加简洁高效；

(2) 课程安排问题中小的变化（每门课题的人数从取任意值变为都相同），导致求解算法的显著变化；

(3) 贪心算法 Greedy-for-IS 的背后，蕴含着更为“笨拙”的动态规划算法 Awkward-DP-for-IS，因而可以视为 Awkward-DP-for-IS 的一个简化 [?]。

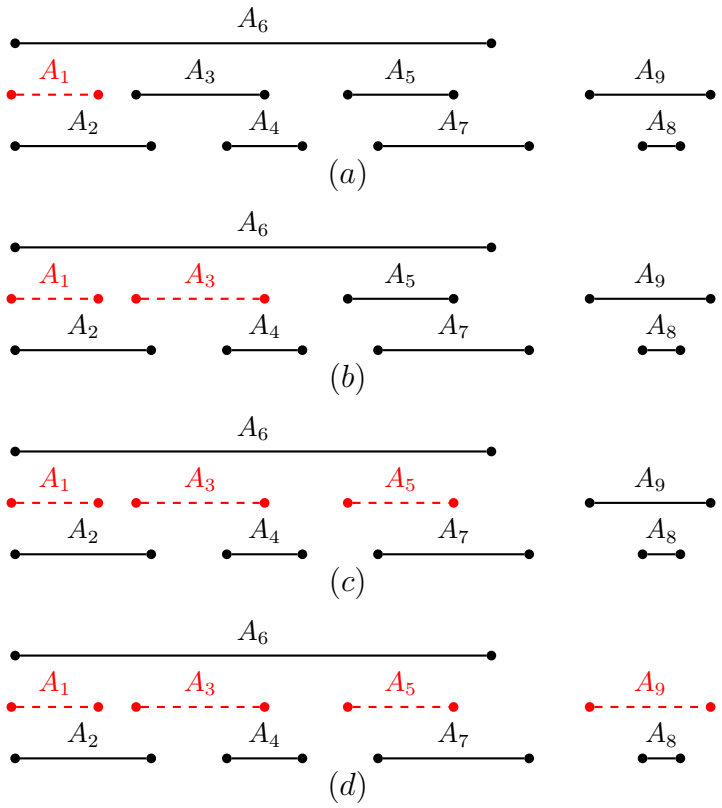


图 5.13: 应用贪心算法 Greedy-for-IS 求解课程安排问题的运行过程示例。图中虚线表示依据贪心规则逐步选择的课程

5.4.3 关于贪心规则设计过程的一些讨论

在展示了如何从动态规划算法简化成贪心算法之后，一个有意思的话题是与“直接依据优化目标设计贪心规则”策略做个对比。

既然整体优化目标是找出最多不冲突的课程，那在每步决策时都尽量减少冲突，似乎有助于达到整体目标；我们依靠“灵感”，猜测如下几种规则或许可行 [?]：

(1) 选择最早上课的课程：上课越早，似乎越不容易与其他课程冲突；

(2) 选择时长最短的课程：时长越短，似乎越不容易与其他课题冲突；

(3) 选择冲突数最少的课程：每次都选择与别的课冲突最少的课程，似乎有助于最终选择出最多的不冲突课程；

(4) 选择最早下课的课程：下课越早，给剩余子问题留下的“时间范围”就越大，似乎有可能选择出越多的不冲突课程。

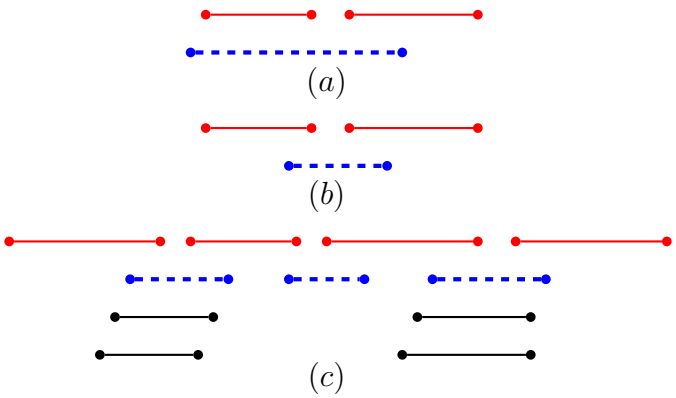


图 5.14: 课程安排问题的 3 条贪心规则及其反例。(*a*) 采用“选择最早上课的课程”规则，只能安排一门课程，但最优解值为 2；(*b*) 采用“选择时长最短课程”规则，只能安排一门课程，但最优解值为 2；(*c*) 采用“选择冲突最少课程”规则，只能安排 3 门课程，但最优解值为 4。图中虚线表示按照贪心规则选择的课程

然而逐一验证表明，应用前三条规则无法保证得到最优解，反例见图 5.14；而应用第 4 条规则能够获得最优解。

上述的贪心规则设计过程，尤其图 5.14所示的反例，表明了如下事实：*i*) 想出一条有效的贪心规则是不容易的；*ii*) “拆分整体目标”策略是一个“试错”过程，有较大的盲目性，其成功可能性不太高。其根本性的原因在于：上述设计策略是基于对少数实例的观察，更大程度上是依赖“人的灵感”。那么，有没有更好的贪心规则设计策略呢?

最近隋京言等 [?] 尝试了一个新的设计策略：**将贪心规则的设计问题转化成一个机器学习问题，用神经网络学习出贪心规则**。其关键步骤包括：*i*) 首先随机生成大量的实例集合；*ii*) 然后对每个实例运行“笨拙”的动态规划算法，计算出最优决策项；*iii*) 最后训练神经网络学习最优决策。这种策略的好处是：在大量的实例集合上，采用机器学习技术“学习”出贪心规则，有希望避免试错法依赖于“人的灵感”、仅观察少量实例的缺陷。实验结果表明：采用神经网络技术学习出的贪心算法，其性能接近于采用“选择最早下课”规则的贪心算法。

5.5 将动态规划简化成贪心算法：再论最短路径问题

在前面的小节里，我们已经描述了如何应用*直接依据优化目标设计贪心规则*策略，设计出贪心算法；现在我们再从简化动态规划算法的角度，描述贪心算法的设计过程。

那怎样简化动态规划算法呢?

我们首先要观察*多步决策中最优决策项的规律*，如果观察到最优决策项的规律，就能够从两个方面对动态规划算法进行简化：

(1) *简化决策过程*：在决策时（也就是逆向回溯时），不再是枚举所有的决策项，而是依据发现的规律直接选择最优决策项；

(2) *简化计算过程*：在正向计算 OPT 值表时，依据发现的规律识别出 OPT 表中不必要计算的单元，进而简化动态规划算法，避免这些冗余计算。

以单源最短路径问题为例，对无负圈的有向图来说，Bellman-Ford 动态规划算法可以求出单源最短路径；如果再进一步加强条件，要求图中无负边的话，Bellman-Ford 动态规划算法自然也适用。我们在一些实例上运行 Bellman-Ford 动态规划算法，观察运行过程中的规律，进而利用这些规律简化动态规划算法，最终将得到 Dijkstra 贪心算法。

5.5.1 Bellman-Ford 算法中的第一类冗余计算

在 Bellman-Ford 算法中，多步决策中的最优决策项是通过回溯确定下来的。对于图 5.15(a) 所示实例来说，Bellman-Ford 算法的计算及回溯过程见图 5.15(b)。以结点 x 为例，Bellman-Ford 算法计算出最短距离是 $d(x,5)=2$，最优决策项的回溯过程如下：

由 $d(x,5)$ 的计算过程

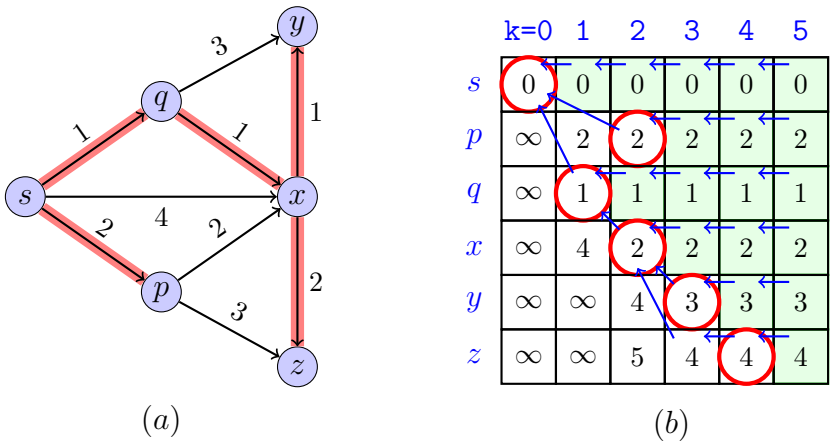
$$d(x,5)=\min\big\{d(x,4),d(p,4)+2,d(s,4)+4,d(q,4)+1\big\}=d(x,4),$$

得知需从 $d(x,5)$ 回溯至 $d(x,4)$。由

$$d(x,4)=\min\big\{d(x,3),d(p,3)+2,d(s,3)+4,d(q,3)+1\big\}=d(x,3),$$

得知需从 $d(x,4)$ 回溯至 $d(x,3)$。由

$$d(x,3)=\min\big\{d(x,2),d(p,2)+2,d(s,2)+4,d(q,2)+1\big\}=d(x,2),$$



(a) (b)

图 5.15: 无负边有向图上单源最短路径的计算过程示例。(a) 一个无负边有向图上从 s 到各顶点的最短路径，构成一棵以 s 为根结点的最短路径树（带阴影边）；(b) Bellman-Ford 算法计算出的最短距离 $d(v,k)$ 表。蓝色箭头表示计算最优解的回溯过程；圆圈内单元表示每一列的最小值（扣除此前各列最小值之后）；阴影区单元与左侧圆圈内单元具有相同的值，因此无需重复进行计算

得知需从 $d(x,3)$ 回溯至 $d(x,2)$。由

$$d(x,2) = \min\{d(x,1), d(p,1)+2, d(s,1)+4, d(q,1)+1\} = d(q,1)+1,$$

得知需从 $d(x,2)$ 回溯至 $d(q,1)$。

总结上述的最优项回溯过程，可知有 4 个单元的值是完全相同的，即：$d(x,5) = d(x,4) = d(x,3) = d(x,2)$。换句话说，在 Bellman-Ford 算法的正向计算过程中，对 $d(x,3)$, $d(x,4)$, $d(x,5)$ 这些单元的计算是冗余的。

那么，这些冗余单元的出现位置有何规律呢?

这些冗余单元都出现*在一列的最小单元之后*：在图 5.15(b) 所示表格中，$d(x,2)$ 是第 2 列中的最小单元（扣除第 0 列中的最小单元 $d(s,0)$、第 1 列中的最小单元 $d(q,1)$ 之后）；在后续的计算过程中，这些最小单元的值保持不变，即：$d(x,2) = d(x,3) = d(x,4) = d(x,5)$。类似的，$d(p,2)$ 也是最小单元，因此在后续过程中也不再变化，即：$d(p,2) = d(p,3) = d(p,4) = d(p,5)$。

所谓每一列的最小单元，是不太严格的说法，因为它们是*扣除一些单元之后的最小单元*。为严格表述这一点，我们先引入下述定义：

定义 5.5.1 (距 s 不超过 k 条边的最近顶点). *考虑包含 n 个顶点的无负边有向图，以及起始顶点 s。对于 $k = 1, \cdots, n-1$，定义 v_k^* 为除 $v_0^*, \cdots, v_{k-1}^*$ 之外、距 s 不超过 k 条边的最近顶点，其中 $v_0^* = s$。*

以图 5.15(b) 为例，我们有 $v_0^* = s$, $v_1^* = p$, $v_2^* = \{q, x\}$, $v_3^* = y$, $v_4^* = t$。利用这个定义，我们可以严格地表述每一列的最小单元：第 k 的最小单元，就是距 s 不超过 k 条边的最近顶点。接下来，我们可以证明最小单元与其右侧单元具有相同的值，即：

定理 5.5.1. *考虑包含 n 个顶点的无负边有向图，以及起始顶点 s。对任意的 k，$0 \le k \le n-1$，令 $d(v,k)$ 表示从 s 出发、经过不超过 k 条边到达顶点 v 的最短距离，v_k^* 表示距 s 不超过 k 条边的 UI 金顶点，则有：*

$$d(v_k^*, k) = d(v_k^*, k+1) = \cdots = d(v_k^*, n-1).$$

证明： 我们在此只证明 $d(v_k^*, k) = d(v_k^*, k+1)$，$d(v_k^*, k+1) = d(v_k^*, k+2) = \cdots = d(v_k^*, n-1)$ 的证明类似，在此不赘述。

对 $d(v_k^*, k+1)$ 应用第三章里的递归表达式，可得：

$$d(v_k^*, k+1) = \min\big\{d(v_k^*, k), \min_{(u, v_k^*) \in E}\{d(u,k) + w(u, v_k^*)\}\big\}.$$

因此，只需证明对于任意的边 $(u, v_k^*) \in E$，不等式 $d(v_k^*, k) \le d(u,k) + w(u, v_k^*)$ 成立即可。下面我们区分顶点 u 的两种情形进行证明：

(1) $u \notin \{v_0^*, \cdots, v_{k-1}^*\}$：由于 v_k^* 是不超过 k 条边的最近顶点，故 $d(v_k^*, k) \le d(u,k)$；再考虑到边的权重都是正数，即：$w(u, v_k^*) > 0$，因此有 $d(v_k^*, k) \le d(u,k) + w(u, v_k^*)$ 成立。

以图 5.15为例，当 $k = 2$ 时，我们有 $v_2^* = x$。由于 $d(x,2) \le d(v,2)$ 且 $w(v,x) = 2 > 0$，故可得：$d(x,2) < d(v,2) + w(v,x)$.

(2) $u \in \{v_0^*, \cdots, v_{k-1}^*\}$：注意到在此情形下，依照递归假设，有 $d(u, k-1) = d(u,k)$。对 $d(v_k^*, k)$ 再次应用递归表达式，可得：

$$d(v_k^*, k) = \min\big\{d(v_k^*, k-1), \min_{(v, v_k^*) \in E}\{d(v, k-1) + w(v, v_k^*)\}\big\}.$$

特殊地，令 $v = u$，可得：

$$d(v_k^*, k) \le d(u, k-1) + w(u, v_k^*) = d(u,k) + w(u, v_k^*).$$

以图 5.15为例，我们有 $d(u,1) = d(u,2)$, $d(s,1) = d(s,2)$，因此可得：

$$d(x,2) \le d(u,1) + w(u,x) = d(u,2) + w(u,x),$$
$$d(x,2) \le d(s,1) + w(s,x) = d(s,2) + w(s,x).$$

□

需要强调指出的是，在上述证明过程中，关键是边的权重都是正数这个条件；对于有负边的图来说，这个定理是不能保证成立的。

综上所述，我们可以这样描述 Bellman-Ford 算法中的第一类冗余计算：令集合 S 表示最短距离已确定的顶点，也就是第 k 列的最小单元 v_k^*；在后续的计算中，无需考虑 S 中的顶点，只需考虑不在 S 中的那些顶点即可。

5.5.2 对 Bellman-Ford 算法的第一次简化

我们去除 S 内顶点的重复计算，将显著改进 Bellman-Ford 算法；改进后的算法见算法 49。

Algorithm 49 计算无负边有向图上单源最短路径的快速 Bellman-Ford 算法

function Fast-Bellman-Ford($G = \langle V, E\rangle, s$)

Set $S = \{s\}$; //S denotes the set of explored nodes;
Set $d[s] = 0$, $d[v] = \infty$ and $\pi[v]$ =NIL for each node $v \in V$;
for $k = 1$ to $|V| - 1$ **do**
  for each node $v \notin S$ **do**
    for each edge $(u, v) \in E$ **do**
      if $d[v] > (d[u] + w(u, v))$ **then**
        $d[v] = d[u] + w(u, v)$;
        $\pi[v] = u$;
      end if
    end for
  end for
  Add node v^* to S, where $d[v^*]$ is the minimum in $d[v]$;
end for
return array d and π;

和第三章中的 Bellman-Ford 算法一样，我们用数组 $\pi[v]$ 表示最短路径上 v 的前驱顶点，用数组 $d[v]$ 而不是矩阵来表示最短距离 $d(v, k)$，即：在第 k 轮循环中，$d[v]$ 的值表示 $d(v, k)$。如图 5.16(a) 所示，Fast-Bellman-Ford 算法避免了对 $d(v, k)$ 表中重复单元的计算。

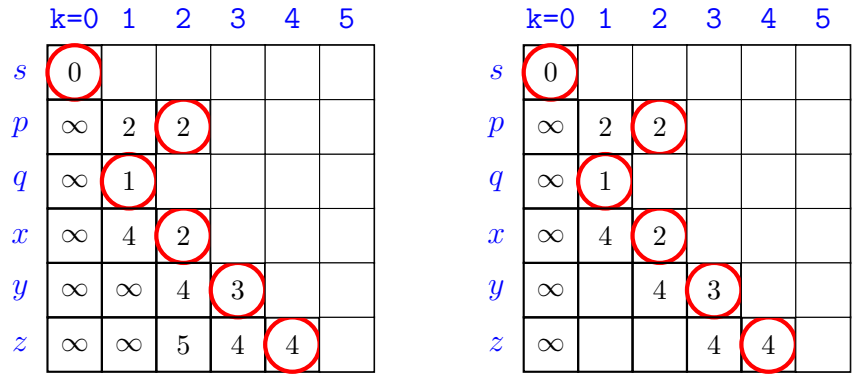


图 5.16: Bellman-Ford 算法的两次简化。(*a*) 每一列最小单元（用红色圆圈标示）右侧单元是冗余计算，Fast-Bellman-Ford 算法避免了对这些重复单元的计算；(*b*) 每一列最小单元（用红色圆圈标示）左侧部分单元也是冗余计算；Faster-Bellman-Ford 算法避免了对这些重复单元的计算，只考虑前一列最小单元的邻居结点

5.5.3 Bellman-Ford 算法中的第二类冗余计算

刚才我们看到了 Bellman-Ford 算法中的第一类冗余计算：每列最小单元右侧的单元都是冗余的。事实上，最小单元的左侧也有部分单元是冗余的。一个典型例子是图 5.15中的 $d(z,2)$：Bellman-Ford 算法计算出 $d(z,2)=5$，然而这完全是冗余计算，因为在计算最优解的回溯过程中，根本不会用到这个单元。

那为何回溯时不会用到这个单元呢?

这背后的道理很简单：第 2 列的最小单元只能是前两列最小单元的邻居之一；对这个例子来说，第 1 列的最小单元是 $d(q,1)$，第 0 列的最小单元是 s；然而 z 既不是 q 的邻居，也不是 s 的邻居，因此根本不可能成为第 2 列的最小单元。类似地，$d(y,1)$ 和 $d(z,1)$ 也是冗余单元。

5.5.4 对 Bellman-Ford 算法的再次简化：Dijkstra 贪心算法

基于上述观察，我们可以对 Fast-Bellman-Ford 算法做进一步的改进：在计算每一列的最小单元时，考虑不在 S 中的所有顶点是不必要的，只考虑与 S 中顶点相邻的那些顶点就足够了。改进后的伪代码见算法 50，执行过程见表 5.16。

值得指出的是，在去除这两类冗余计算之后，简化后的 Bellman-Ford 动态规划算法实质上就是 Dijkstra 贪心算法。

总结一下，我们既可以应用“拆分整体目标”策略，依据多步决策过程直接设计贪心算法，也可以先设计动态规划算法，然后观察算法运行过程中最优决策项的规律，进而将动态规划算法简化成贪心算法。

Algorithm 50 去除两类冗余计算之后的快速 Bellman-Ford 算法

function Faster-Bellman-Ford($G = \langle V, E \rangle, s$)

Set $S = \{s\}$; //S denotes the set of explored nodes;
Set $d[s] = 0$, $d[v] = \infty$ and $\pi[v] =$NIL for each node $v \in V$;
for $k = 1$ to $|V| - 1$ **do**
  for each edge $(u, v) \in E$ such that $u \in S$, $v \notin S$ **do**
    if $d[v] > (d[u] + w(u, v))$ **then**
      $d[v] = d[u] + w(u, v)$;
      $\pi[v] = u$;
    end if
  end for
  Add node v^* to S, where $d[v^*]$ is the minimum in $d[v]$;
end for
return array d and π;

5.6 最小加权集合覆盖问题：基于神经网络的贪心算法

5.7 贪心算法的理论基础：拟阵

5.8 贪心算法的理论基础：次模函数

延伸阅读

Dijkstra 20 分钟想出的算法

习题

1. Goad1982 作为习题

2. Web Proxy?

3. LWS?

4. Paragraph formatting?

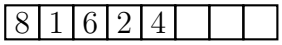
附录

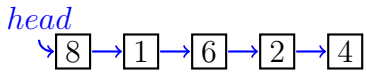
附录 A　优先队列及均摊分析

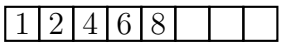
A.1　附录一：优先队列

A.1.1　引言

A.1.2　实现方式 1：数组和链表







A.1.3　实现方式 2：二叉堆

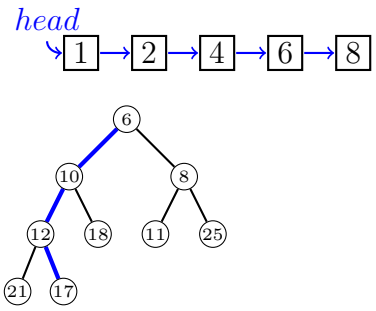
定理 A.1.1. *In an* **implicit** *binary heap, any sequence of* m INSERT, . DECREASEKEY, *and* EXTRACTMIN *operations with* n INSERT *operations takes* $O(m \log n)$ *time.*

定理 A.1.2. *In an* **explicit** *binary heap with* n *nodes, the* INSERT, . DECREASEKEY, *and* EXTRACTMIN *operations take* $O(m \log n)$ *time in the worst case.*

定理 A.1.3. *Given* n *elements, a binary heap can be constructed using* $O(n)$ *time.*

证明.

- There are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h;
- It takes $O(h)$ time to sink a node of height h;



- The total time is:

$$\begin{aligned}\sum_{h=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil h &\leq \sum_{h=0}^{\lfloor \log_2 n \rfloor} n \frac{h}{2^h} \\ &\leq 2n\end{aligned}$$

□

A.1.4 实现方式 3：二项式堆

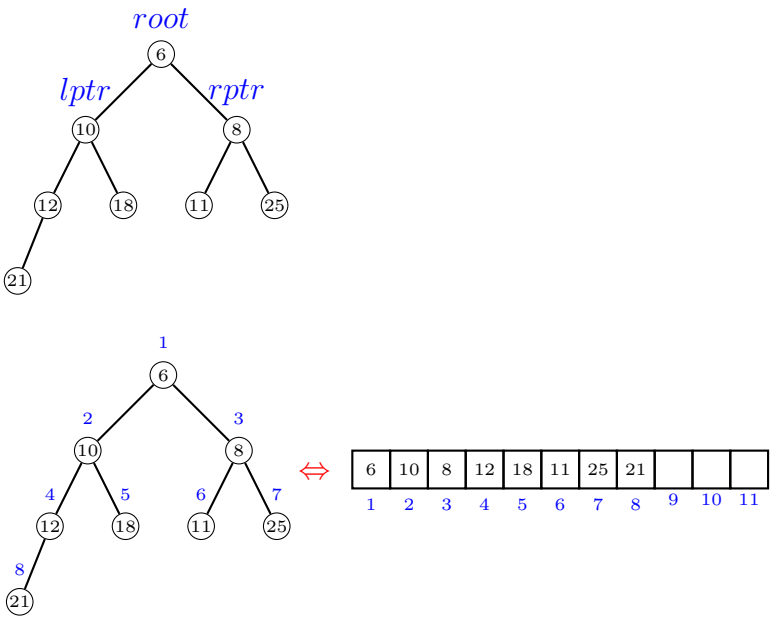
定义 A.1.1 (Binomial tree). *The binomial tree is defined recursively: a single node is itself a B_0 tree, and two B_k trees are linked into a B_{k+1} tree.*

Properties:

1. $|B_k| = 2^k$.
2. $height(B_k) = k$.
3. $degree(B_k) = k$.
4. The i-th child of a node has a degree of $i - 1$.
5. The deletion of the root yields trees B_0, B_1, ..., B_{k-1}.
6. Binomial tree is named after the fact that the node number of all levels are binomial coefficients.

定义 A.1.2 (Binomial forest). *A binomial heap is a collection of several binomial trees:*

- *Each tree is heap ordered;*
- *There is either 0 or 1 B_k for any k.*



1. A binomial heap with n nodes contains the binomial tree B_i iff $b_i = 1$, where $b_k b_{k-1} ... b_1 b_0$ is binary representation of n.

2. It has at most $\lfloor \log_2 n \rfloor + 1$ trees.

3. Its height is at most $\lfloor \log_2 n \rfloor$.

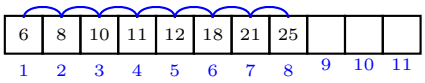
Thus, it takes $O(\log n)$ time to find the minimum element via checking the roots.

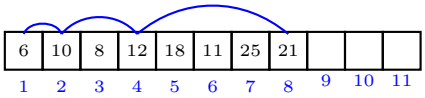
INSERT(x)

1: Create a B_0 tree for x;
2: Change the pointer to the minimum root node if necessary;
3: **while** there are two B_k trees for some k **do**
4: Link them together into one B_{k+1} tree;
5: Change the pointer to the minimum root node if necessary;
6: **end while**

EXTRACTMIN()

1: Remove the min node, and insert its children into the root list;
2: Change the pointer to the minimum root node if necessary;
3: **while** there are two B_k trees for some k **do**
4: Link them together into one B_{k+1} tree;
5: Change the pointer to the minimum root node if necessary;





6: **end while**

- A single INSERT operation takes time $1 + w$, where $w = \#\texttt{WHILE}$.
- For the sake of calculating the total running time of a sequence of operations, we represent the running time of a single operation as decrease of a potential function.
- Consider a quantity $\Phi = \#trees$ (called potential function). The changes of Φ during an operation are:
  - Φ increase: 1.
  - Φ decrease: w.
- Thus the running time of INSERT can be rewritten in terms of Φ as $1 + w = 1+$ decrease in Φ. Note that this representation makes it convenient to sum running time of a sequence of INSERT operations.
- A single EXTRACTMIN operation takes $d + w$ time, where d denotes degree of the removed root node, and $w = \#\texttt{WHILE}$.
- For the sake of calculating the total running time of a sequence of operations, we represent the running time of a single operation as decrease of a potential function.
- Consider a potential function $\Phi = \#trees$. The changes during an operation are:
  - Φ increase: d.
  - Φ decrease: w.
- Similarly, the running time is rewritten in terms of Φ as $d + w = d+$ decrease in $\#trees$. Note that $d \leq \log n$.
- Let's consider any sequence of n INSERT and m EXTRACTMIN operations.
- The total running time is at most $n + m \log n +$ total decrease in $\#trees$.

图 A.1: Heap order is violated: $15 > 13$. Exchange them to resolve the conflict.

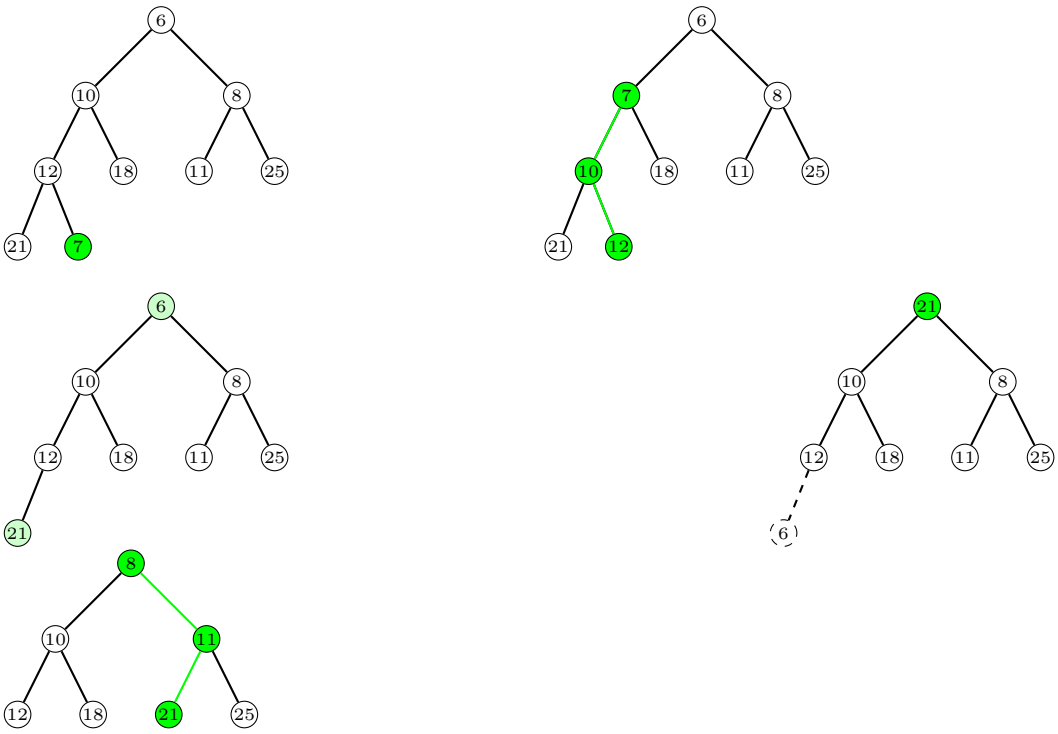
图 A.2: Heap order is violated: $20 > 12$, and $20 > 18$. Exchange 20 with its smaller child (12) to resolve the conflicts.

- Note: total decrease in $\#trees \leq$ total increase in $\#trees$ (why?), which is at most $n + m \log n$.
- Thus the total time is at most $2n + 2m \log n$.
- We say INSERT takes $O(1)$ amortized time, and EXTRACTMIN takes $O(\log n)$ amortized time.

定义 A.1.3 (Amortized time)**.** *For any sequence of n_1 operation 1, n_2 operation 2..., if the total time is $O(n_1T_1 + n_2T_2...)$, we say that operation 1 takes T_1 amortized time, operation 2 takes T_2 amortized time ....*

- The actual running time of an INSERT operation is $1 + w$. A large w means that the INSERT operation takes a long time. Note that the w time was spent on "decreasing trees"; thus, if the w time was amortized over the operations "creating trees", the "amortized time" of INSERT operation will be only $O(1)$.

- The actual running time of an EXTRACTMIN operation is at most $\log n + w$. Note that at most $\log n$ new trees are created during an EXTRACTMIN operation; thus,

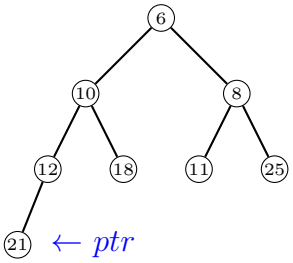


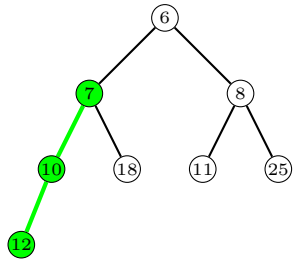
the amortized time is still $O(\log n)$ even if some costs have been amortized to it from other operations due to "tree creating".

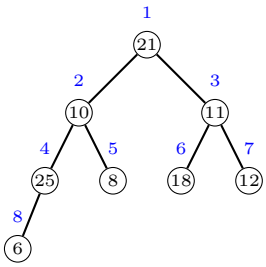
A.1.5 实现方式 4：Fibonacci 堆

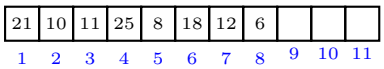
DecreaseKey(v, x)

1: $key(v) = x$;
2: **if** heap order is violated **then**
3: $u = v's$ parent;
4: Cut subtree rooted at node v, and insert it into the root list;
5: Change the pointer to the minimum root node if necessary;
6: **while** u is marked **do**
7: Cut subtree rooted at node u, and insert it into the root list;
8: Change the pointer to the minimum root node if necessary;
9: Unmark u;
10: $u = u's$ parent;
11: **end while**
12: Mark u;









13: **end if**

INSERT(x)

1: Create a tree for x, and insert it into the root list;

2: Change the pointer to the minimum root node if necessary;

Note: **Being lazy!** Consolidating trees when extracting minimum.

EXTRACTMIN()

1: Remove the min node, and insert its children into the root list;

2: Change the pointer to the minimum root node if necessary;

3: **while** there are two roots u and v of the same degree **do**

4: Consolidate the two trees together;

5: Change the pointer to the minimum root node if necessary;

6: **end while**

DECREASEKEY(v, x)

1: $key(v) = x$;

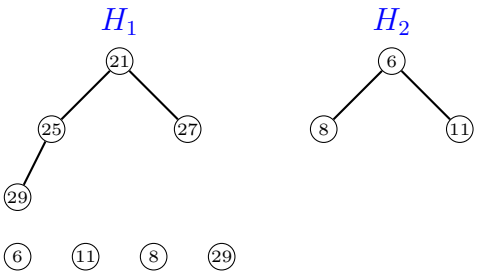
2: **if** heap order is violated **then**

3: $u = v's$ parent;

4: Cut subtree rooted at node v, and insert it into the root list;

5: Change the pointer to the minimum root node if necessary;

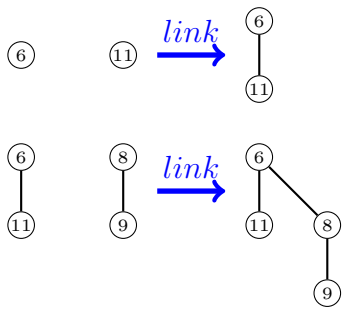
6: **while** u is marked **do**

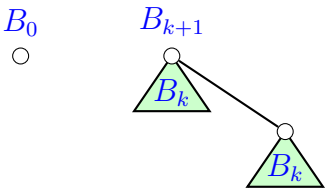


7: Cut subtree rooted at node u, and insert it into the root list;
8: Change the pointer to the minimum root node if necessary;
9: Unmark u;
10: $u = u's$ parent;
11: **end while**
12: Mark u;
13: **end if**

- The actual running time of a single operation is $1 + w$, where $w = \#\texttt{WHILE}$.
- To calculate the **total running time** of **a sequence of operations**, we represent the running time of **a single operation** as **decrease of a potential function**.
- Consider a **potential function** $\Phi = \#trees + 2\#marks$. The changes of Φ during an operation are:
  - Φ increase: $1 + 2 = 3$.
  - Φ decrease: $(-1 + 2 * 1) * w = w$.
- Thus we can rewrite the running time in terms of Φ as $1 + w = 1 + \Phi$ decrease.

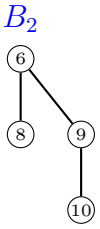
ExtractMin()

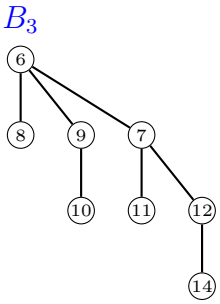


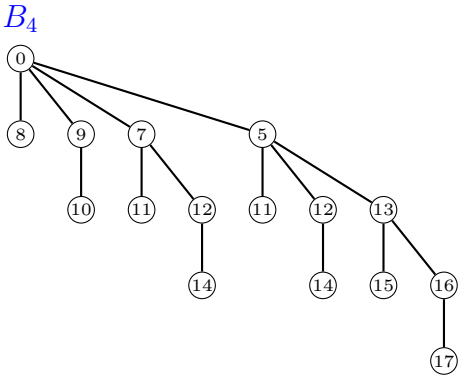

B_0
B_{k+1}
B_k
B_k

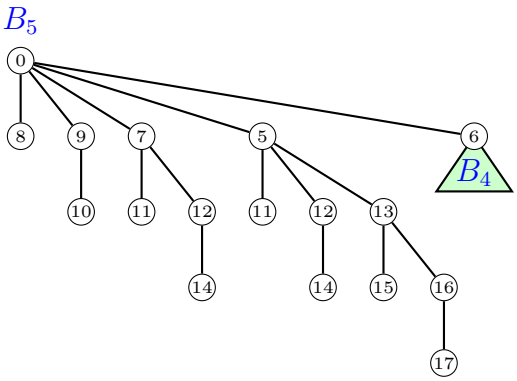

B_0
6

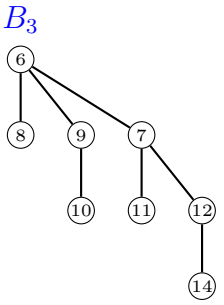

B_1
6
8

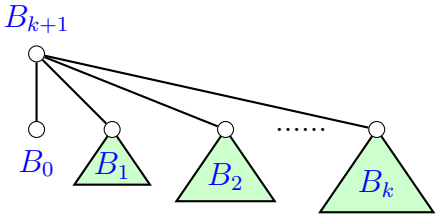

B_2
6
8
9
10

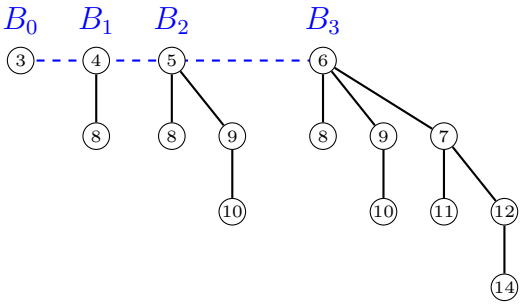

B_3
6
8
9
7
10
11
12
14

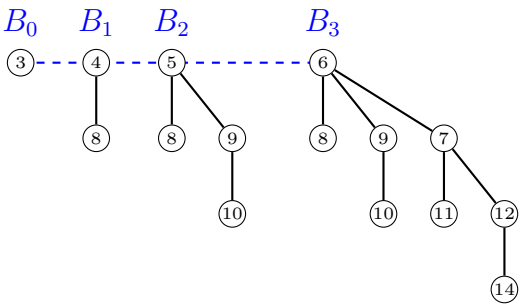

B_4
0
8
9
7
5
10
11
12
11
12
13
14
14
15
16
17


B_5
0
8
9
7
5
6
B_4
10
11
12
11
12
13
14
14
15
16
17


B_3
6
8
9
7
10
11
12
14


B_{k+1}
B_0
B_1
B_2
.....
B_k


B_0
B_1
B_2
B_3
3
4
5
6
8
8
9
8
9
7
10
10
11
12
14


B_0
B_1
B_2
B_3
3
4
5
6
8
8
9
8
9
7
10
10
11
12
14

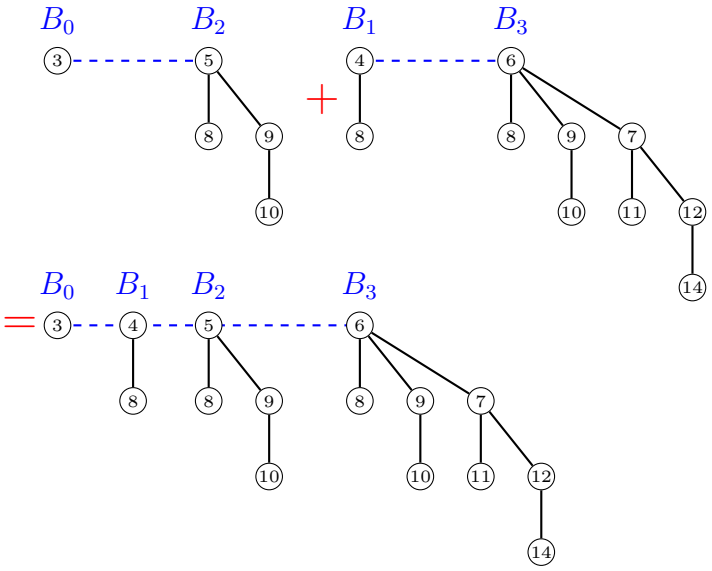


图 A.3: An easy case: no consolidating is needed

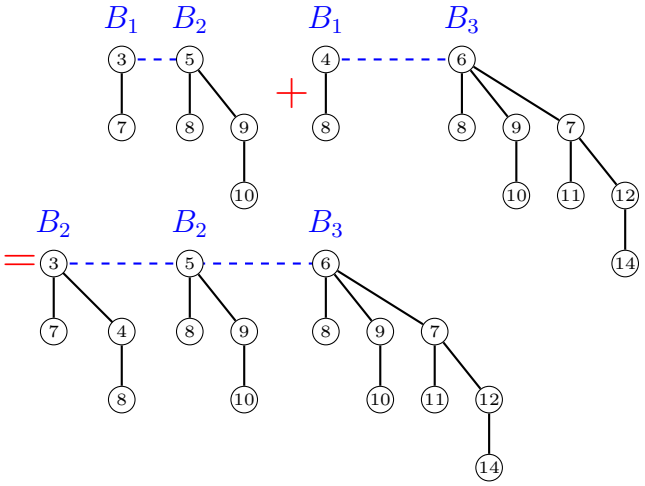


图 A.4: Consolidating two B_1 trees into a B_2 tree

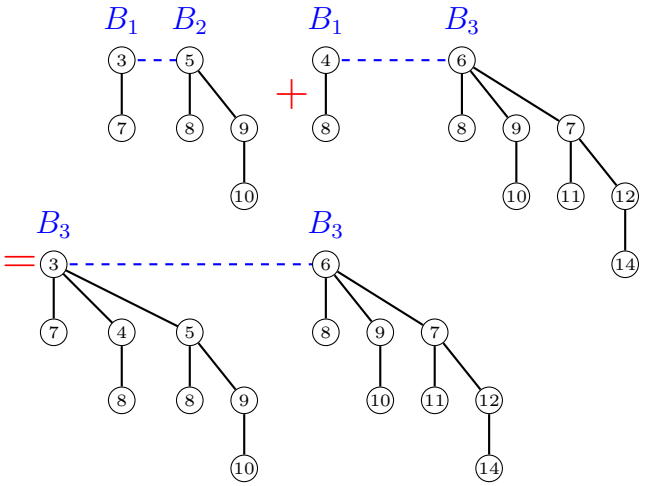


图 A.5: Consolidating two B_2 trees into a B_3 tree

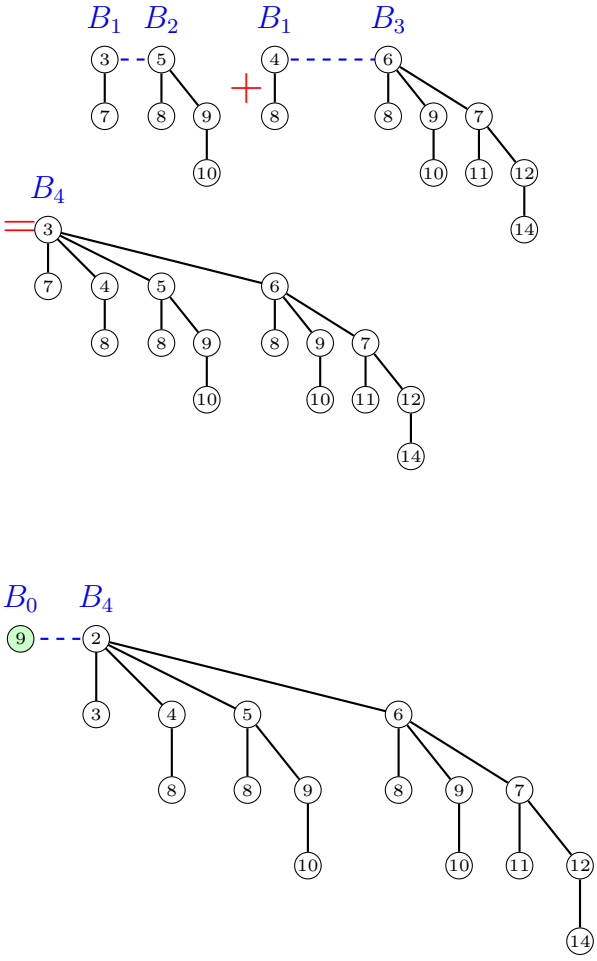


图 A.6: An easy case: no consolidating is needed

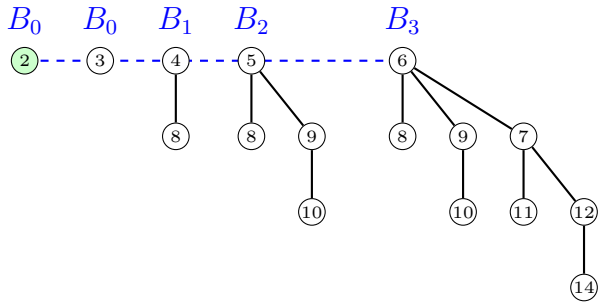


图 A.7: Consolidating two B_0

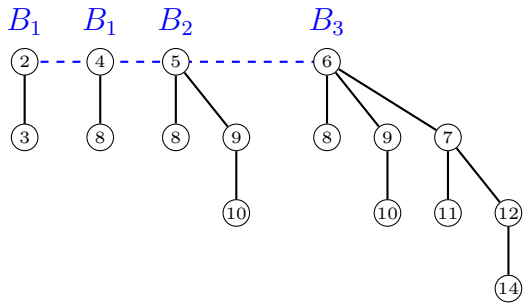


图 A.8: Consolidating two B_1

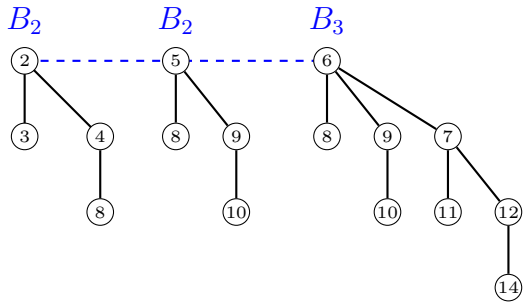


图 A.9: Consolidating two B_2

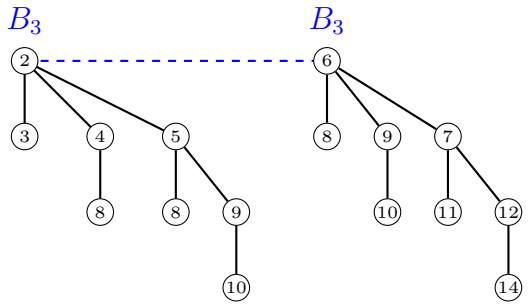


图 A.10: Consolidating two B_3

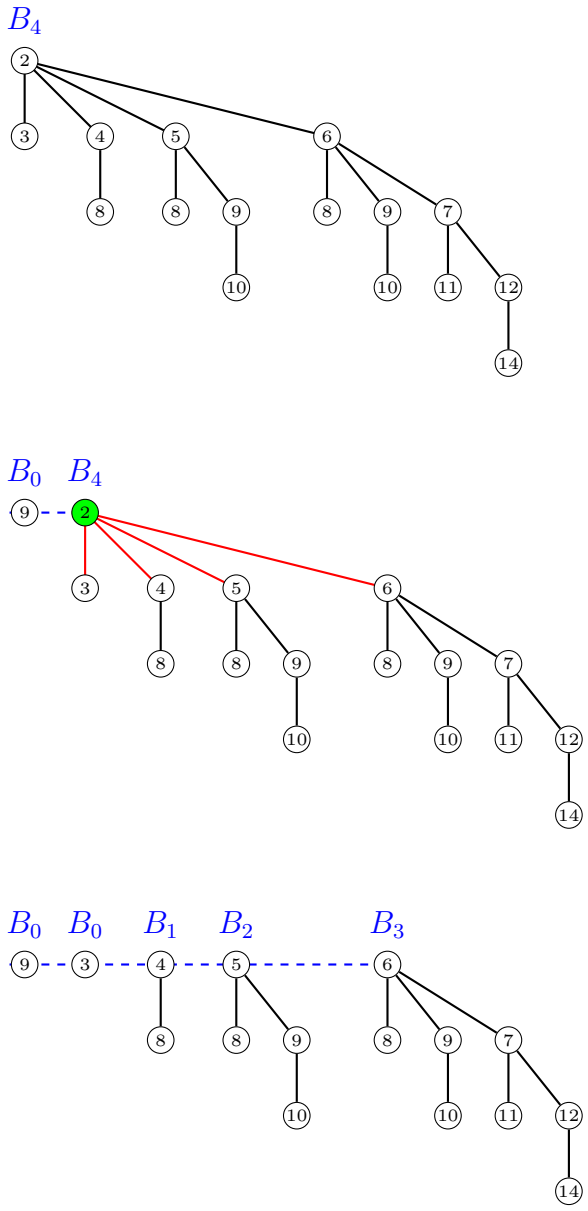


图 A.11: The four children become trees

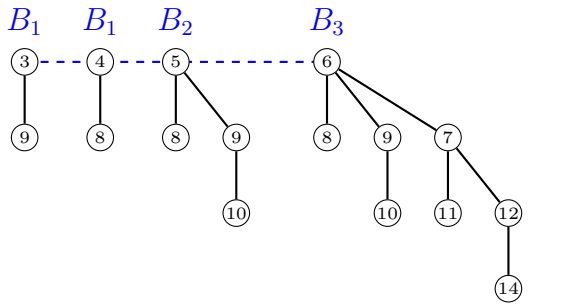


图 A.12: Consolidating two B_1 trees

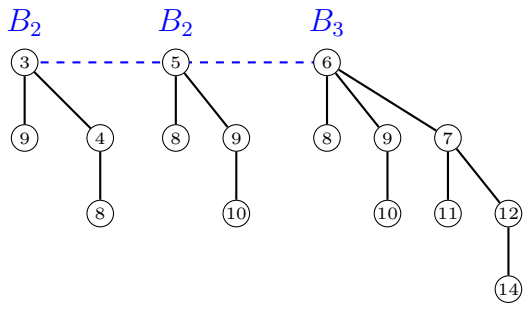


图 A.13: Consolidating two B_2 trees

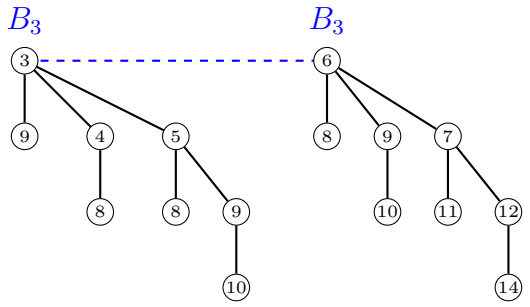


图 A.14: Consolidating two B_2 trees

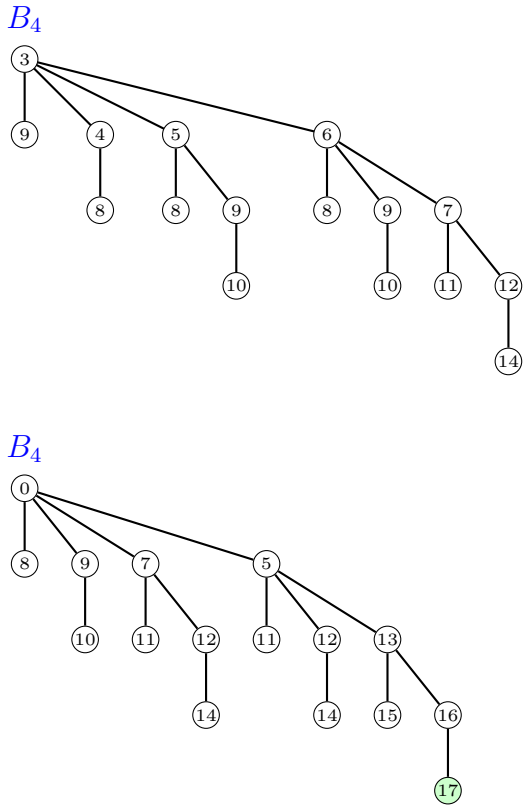


图 A.15: DecreaseKey: 17 to 1

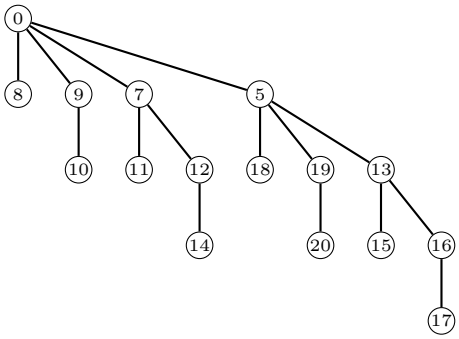


图 A.16: A Fibonacci heap. To DecreaseKey: 19 to 3.

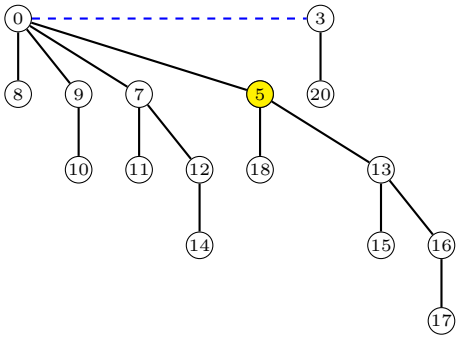


图 A.17: After DecreaseKey: 19 to 3. To DecreaseKey: 15 to 2.

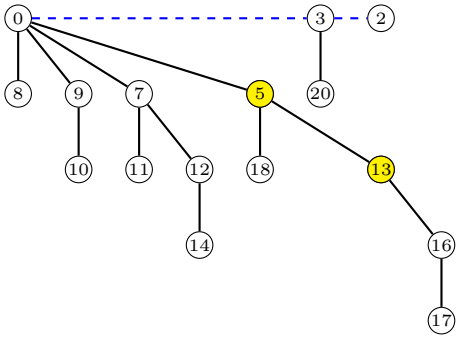


图 A.18: After DecreaseKey: 15 to 2. To DecreaseKey: 12 to 8.

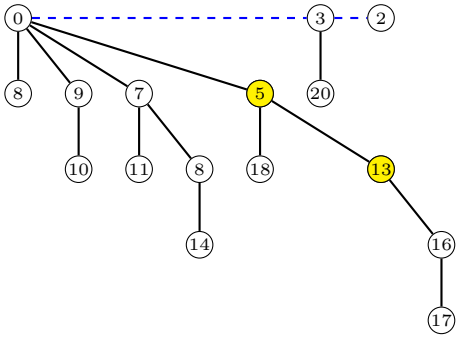


图 A.19: After DecreaseKey: 12 to 8. To DecreaseKey: 14 to 1.

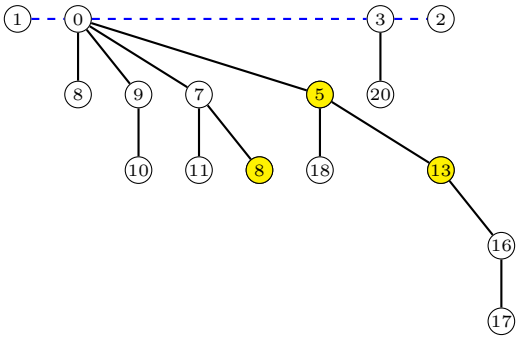


图 A.20: After DecreaseKey: 14 to 1. To DecreaseKey: 16 to 9.

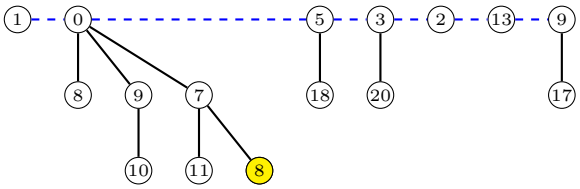


图 A.21: After DecreaseKey: 16 to 9

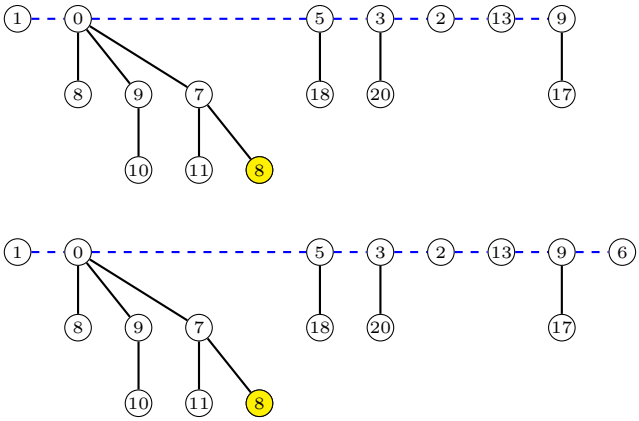


图 A.22: Insert(6): creating a new tree, and insert it into the root list

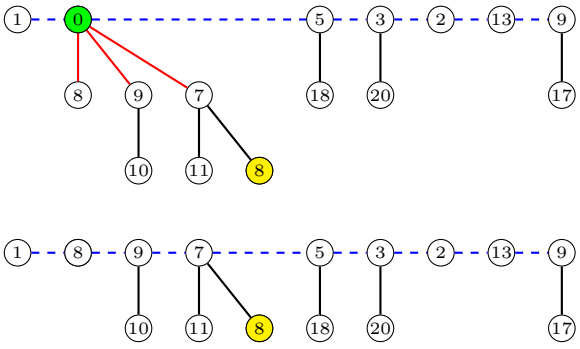


图 A.23: ExtractMin: removing the min node, and adding 3 trees

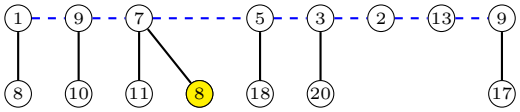


图 A.24: ExtractMin: after consolidating two trees rooted at node 1 and 8

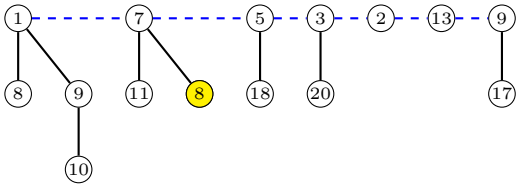


图 A.25: ExtractMin: after consolidating two trees rooted at node 1 and 9

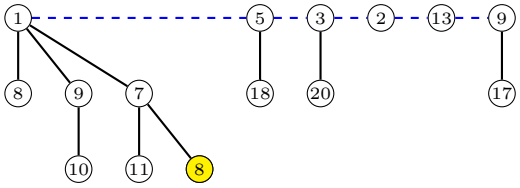


图 A.26: ExtractMin: after consolidating two trees rooted at node 1 and 7

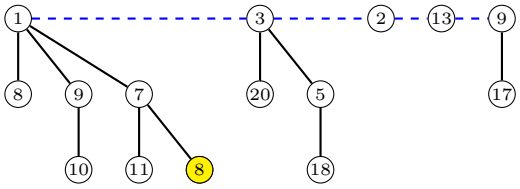


图 A.27: ExtractMin: after consolidating two trees rooted at node 3 and 5

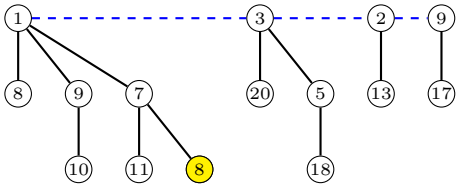


图 A.28: ExtractMin: after consolidating two trees rooted at node 2 and 13

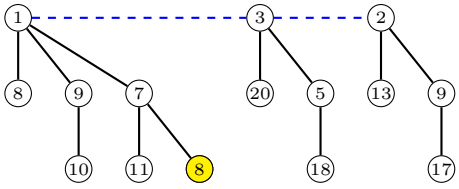


图 A.29: ExtractMin: after consolidating two trees rooted at node 2 and 9

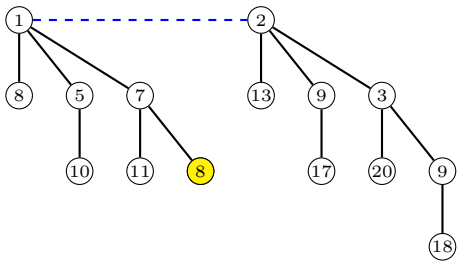


图 A.30: ExtractMin: after consolidating two trees rooted at node 2 and 3

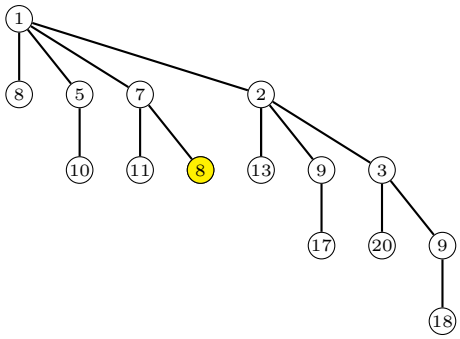
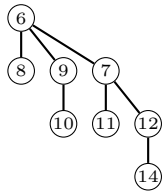


图 A.31: ExtractMin: after consolidating two trees rooted at node 1 and 2

1: Remove the min node, and insert its children into the root list;
2: Change the pointer to the minimum root node if necessary;
3: **while** there are two roots u and v of the same degree **do**
4: Consolidate the two trees together;
5: Change the pointer to the minimum root node if necessary;
6: **end while**

- The actual running time of a single operation is $d + w$, where d denotes degree of the removed node, and $w = \#\texttt{WHILE}$.
- To calculate **the total running time** of **a sequence of operations**, we represent the running time of **a single operation** as **decrease of a potential function**.
- Consider a **potential function** $\Phi = \#trees + 2\#marks$. The changes of Φ during an operation are:
  - Φ increase: d.
  - Φ decrease: w.



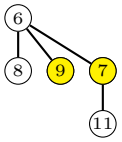
- Thus the running time can be rewritten in terms of Φ as $d + w = d+$ decrease in Φ.

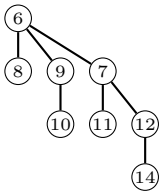
INSERT(x)

1: Create a tree for x, and insert it into the root list;
2: Change the pointer to the minimum root node if necessary;

Analysis:

- The actual running time is 1, and the changes of Φ during this operation are:
  - Φ increase: 1.
  - Φ decrease: 0.
- Consider any sequence of n INSERT, m EXTRACTMIN, and r DECREASEKEY operations.
- The total running time is at most: $n + md_{max} + r+$ total decrease in Φ.
- Note: total decrease in $\Phi \leq$ total increase in $\Phi = n + md_{max} + 3r$.
- Thus the total running time is at most: $n + md_{max} + r + n + md_{max} + 3r = 2n + 2md_{max} + 4r$.
- Thus INSERT takes $O(1)$ amortized time, DECREASEKEY takes $O(1)$ amortized time, and EXTRACTMIN takes $O(d_{max})$ amortized time.
- In fact, EXTRACTMIN takes $O(\log n)$ amortized time since d_{max} can be upper-bounded by $\log n$ (why?).
- Recall that for a binomial tree having n nodes, the root degree d is **exactly** $\log_2 n$, i.e. $d = \log_2 n$.

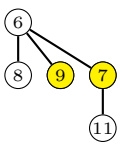




- In contrast, a tree in a Fibonacci heap might have several subtrees cutting off, leading to $d \geq \log_2 n$.
- However, the "marking technique" guarantees that any node can lose at most one child, thus limiting the deviation from the original binomial tree, i.e. $\log_\phi n \geq d \geq \log_2 n$, where $\phi = \frac{1+\sqrt{5}}{2} = 1.618...$.
- Recall that for a binomial tree, the i-th child of each node has a degree of exactly $i - 1$.
- For a tree in a Fibonacci heap , we will show that the i-th child of each node has degree $\geq i - 2$.

引理 A.1.1. *For any node in a Fibonacci heap, the i-th child has a degree $\geq i - 2$.*

证明.
- Suppose u is the **current** i-th child of w;
- If w is not a root node, it has at most 1 child lost; otherwise, it might have multiple children lost;
- Consider the time when u is linked to w. At that time, $degree(w) \geq i - 1$, so $degree(u) = degree(w) \geq i - 1$;
- Subsequently, $degree(u)$ decreases by at most 1 (Otherwise, u will be cut off and no longer a child of w).



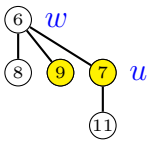


图 A.32: $|B_1| = 2^1$ and $|F_0| = 1 \geq \phi^0$

- Thus, $degree(u) \geq i - 2$.

□

- Let F_k be **the smallest tree** with root degree of k, and for any node of F_k, the i-th child has degree $\geq i - 2$;

- Let F_k be the smallest tree with root degree of k, and for any node of F_k, the i-th child has degree $\geq i - 2$;

- Let F_k be the smallest tree with root degree of k, and for any node of F_k, the i-th child has degree $\geq i - 2$;

- Recall that a binomial tree B_{k+1} is a combination of two B_k trees.
- In contrast, F_{k+1} is the combination of an F_k tree and an F_{k-1} tree.
- We will show that though F_k is smaller than B_k, the difference is not too much. In fact, $|F_k| \geq 1.618^k$.

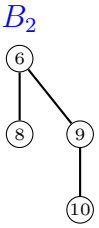


图 A.33: $|B_2| = 2^2$ and $|F_1| = 2 \geq \phi^1$

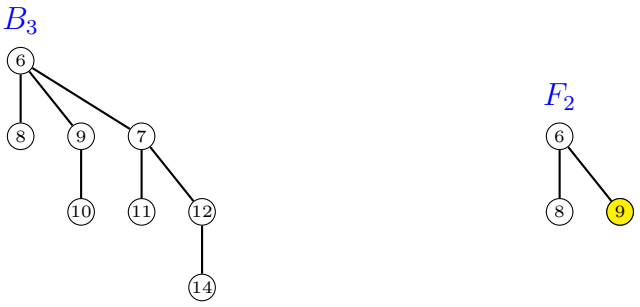


图 A.34: $|B_3| = 2^3$ and $|F_2| = 3 \geq \phi^2$

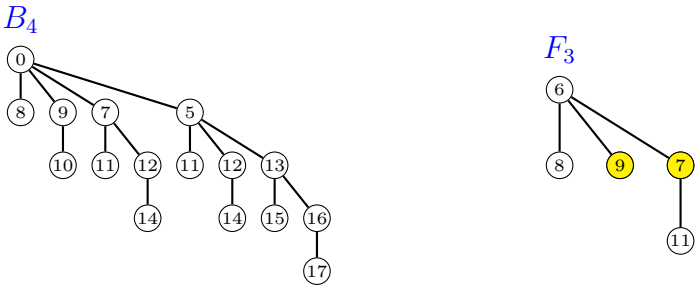


图 A.35: $|B_4| = 2^4$ and $|F_3| = 5 \geq \phi^3$

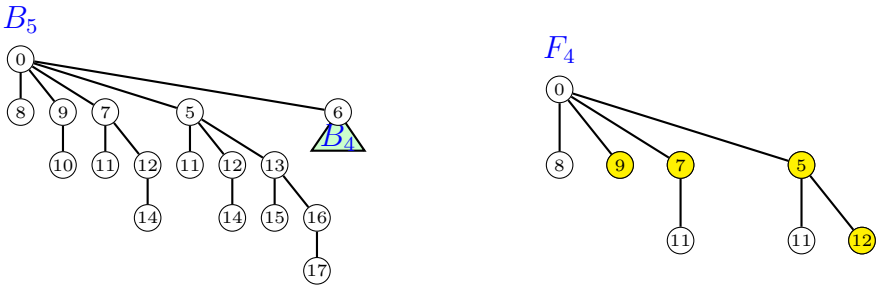
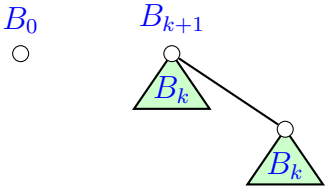
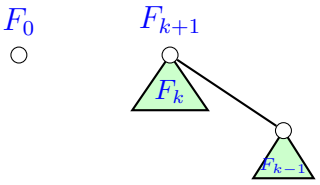


图 A.36: $|B_5| = 2^5$ and $|F_4| = 8 \geq \phi^4$





定义 A.1.4 (Fibonacci numbers)**.** *The Fibonacci sequence is* $0, 1, 1, 2, 3, 5, 8, 13, 21, 34....$

It can be defined by the recursion relation: $f_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ f_{k-1} + f_{k-2} & \text{if } k \geq 2 \end{cases}$

- Recall that $f_{k+2} \geq \phi^k$, where $\phi = \frac{1+\sqrt{5}}{2} = 1.618....$
- Note that $|F_k| = f_{k+2}$, say $|F_0| = f_2 = 1$, $|F_1| = f_3 = 2$, $|F_2| = f_4 = 3$.
- Consider a Fibonacci heap H having n nodes. Let T denote a tree in H with root degree d.
- We have $n \geq |T| \geq |F_d| = f_{d+2} \geq \phi^d$.
- Thus $d = O(\log_\phi n) = O(\log n)$. So, $d_{max} = O(\log n)$.

Therefore, ExtractMin operation takes $O(\log n)$ amortized time.

附录 B　集合的 Union-Find

Outline

- Introduction to UNION-FIND data structure
- Various implementations of UNION-FIND data structure:
  - Array: store "set name" for each element separately. Easy to FIND set of any element, but hard to UNION two sets.
  - Tree: each set is organized as a tree with root as "set name". It is easy to UNION two sets, but hard to FIND set for an element.
  - Link-by-rank: maintain a balanced-tree to limit tree depth to $O(\log n)$, making FIND operations efficient.
  - Link-by-rank and path compression: compress path when performing FIND, making subsequent FIND operations much quicker.

UNION-FIND: motivation

- Motivation: Suppose we have a collection of **disjoint sets**. The objective of UNION-FIND is to keep track of elements by using the following operations:
  - MAKESET(x): to create a new set $\{x\}$.
  - FIND(x): to find the set that contains the element x;
  - UNION(x, y): to union the two sets that contain elements x and y, respectively.
- Analysis: total running time of a sequence of m FIND and n UNION.

UNION-FIND is very useful

图 B.1: Joseph Kruskal

- UNION-FIND has extensive applications, such as:
  - Network connectivity
  - Kruskal's MST algorithm
  - Least common ancestor
  - Games (Go)
  - .....

  An example: Kruskal's MST algorithm
  Kruskal's algorithm [1956]

- Basic idea: during the execution, F is always an **acyclic forest**, and the **safe edge** added to F is always a least-weight edge connecting two distinct components.

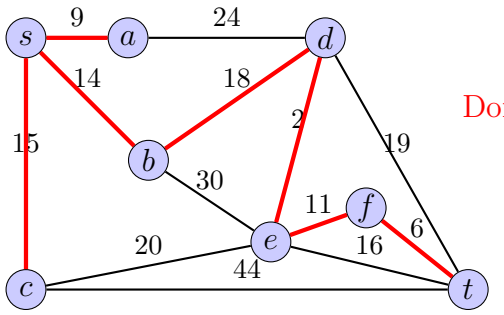
  Kruskal's algorithm [1956]
  MST-KRUSKAL(G, W)

1: $F = \{\}$;
2: **for all** vertex $v \in V$ **do**
3: MAKESET(v);
4: **end for**
5: sort the edges of E into nondecreasing order by weight W;
6: **for** each edge $(u, v) \in E$ in the order **do**
7: **if** FINDSET$(u) \neq$ FINDSET(v) **then**

Step 8

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s, b, c, d, e, f, t\}$



Done!

8: $F = F \cup \{(u, v)\}$;

9: UNION (u, v);

10: **end if**

11: **end for**

- Here, UNION-FIND structure is used to detect whether a set of edges form a cycle.
- Specifically, each set represents a connected component; thus, an edge connecting two nodes in the same set is "unsafe", as adding this edge will form a cycle.

Kruskal's MST algorithm: an example

Time complexity of KRUSKAL'S MST algorithm

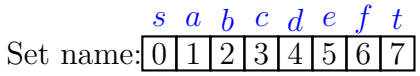
KRUSKAL'S MST algorithm: n MAKESET, $n-1$ UNION, and m FIND operations.

Implementing UNION-FIND: array or linked list

Implementing UNION-FIND: array

- Basic idea: for each element, we record its "set name" individually.

Operation	Array	Tree	Link-by-rank	Link-by-rank + path compression
MAKESET	1	1	1	1
FIND	1	n	$\log n$	$\log^* n$
UNION	n	n	$\log n$	$\log^* n$
MST-KRUSKAL	$O(n^2)$	$O(mn)$	$O(m \log n)$	$O(m \log^* n)$



- Operation:

  FIND(x)

  1: **return** SetName[x];

- Complexity: $O(1)$

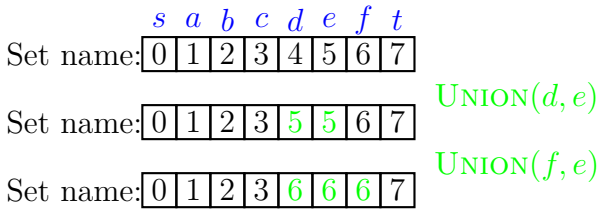
Implementing UNION-FIND: array

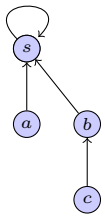
- Operation:

  UNION(x, y)

  1: s_x =FIND(x);
  2: s_y =FIND(y);
  3: **for all** element i **do**
  4: **if** SetName[i]==s_y **then**
  5: SetName[i]=s_x
  6: **end if**
  7: **end for**

- Complexity: $O(n)$



Set: $\{s, a, b, c\}$ 

Tree implementation: organizing a set into a tree with its root as representative of the set

Tree implementation: FIND

- Basic idea: We use a tree to store elements of a set, and use root as "set name". Thus, only one representative should be maintained.
- Operation:

  FIND(x)

  1: $r = x$;
  2: **while** $r! = parent(r)$ **do**
  3: $\quad r = parent(r)$;
  4: **end while**
  5: **return** r;

Tree implementation: UNION

- Operation:

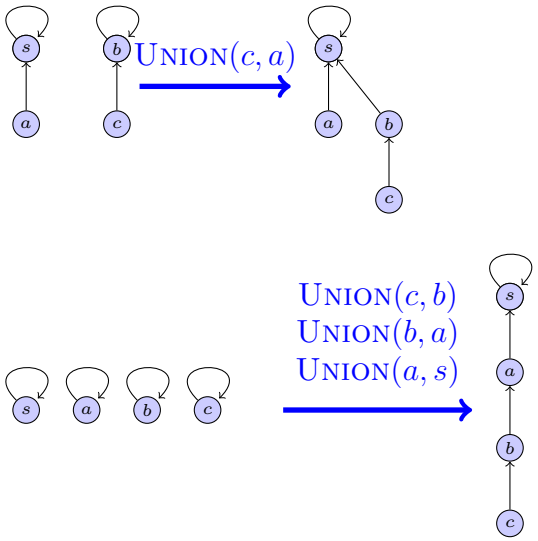
  UNION(x, y)

  1: r_x =FIND(x);
  2: r_y =FIND(y);
  3: $parent(r_x) = r_y$;

- Example: UNION(c, a)

Tree implementation: worst case

- Worst case: the tree degenerates into a linked list. For example, UNION(c, b), UNION(b, a), UNION(a, s).



- Complexity: FIND takes $O(n)$ time, and UNION takes $O(n)$ time.
- Question: how to keep a "good" tree shape to limit path length?

Link-by-rank: shorten the path by maintaining a balanced tree

Tree implementation with link-by-size

- Basic idea: We shorten the path by maintaining a balanced-tree. In fact, this will limit path length to $O(\log n)$.
- How to maintain a balanced tree? Each node is associated with a *rank*, denoting its height. The tree has a balanced shape via linking smaller tree to larger tree; if tie, increase the rank of new root by 1.

Tree implementation with link-by-size: UNION operation

UNION(x, y)

1: r_x =FIND(x);
2: r_y =FIND(y);
3: **if** $rank(r_x) > rank(r_y)$ **then**

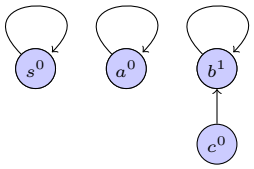
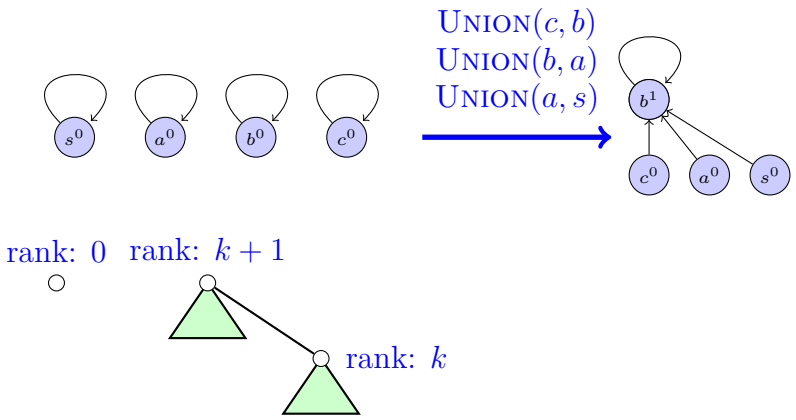


图 B.2: Three sets: $\{s\}$, $\{a\}$, $\{b, c\}$



4: $parent(r_y) = r_x$;
5: **else**
6: $parent(r_x) = r_y$;
7: **if** $rank(r_x) == rank(r_y)$ **then**
8: $rank(r_y) = rank(r_y) + 1$;
9: **end if**
10: **end if**

Note: a node's rank will not change after it becomes an internal node.

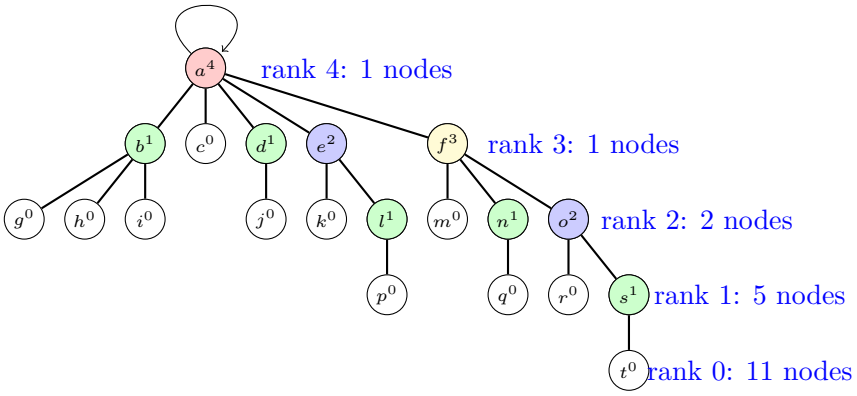
[allowframebreaks] Properties of rank

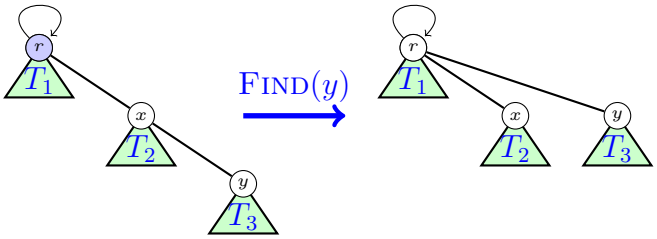
1. For any node x, $rank(x) < rank(parent(x))$.
2. Any tree with root rank of k contains at least 2^k nodes. (Hint: by induction on k.)
3. Once a root node was changed into internal node during a UNION operation, its rank will not change afterwards.
4. Suppose we have n elements. The number of rank k nodes is at most $\frac{n}{2^k}$. (Hint: Different nodes of rank k share no common descendants.)

- Thus, all of the trees have height less than $\log n$, which means both FIND and UNION take $O(\log n)$ time.

Path compression: compress paths to make further FIND efficient

Path compression





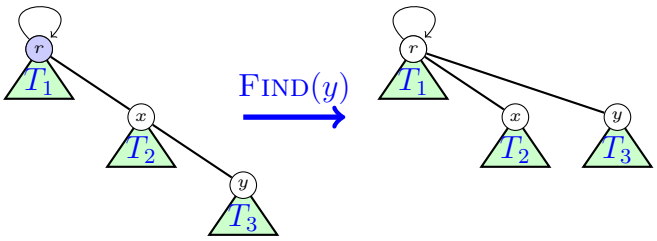
- Basic idea: After finding the root r of the tree containing x, we change the parent of the nodes along the path to point directly to r. Thus, the subsequent FIND(x) operations will be efficient.

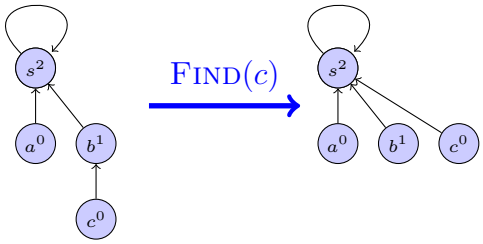
- Note: Path compression changes height of nodes but does not change rank of nodes. We always have $height(x) \leq rank(x)$; thus, the three properties still hold.

Path compression: FIND operation

FIND(x)

1: **if** $x! = parent(x)$ **then**
2: $parent(x) =$ FIND($parent(x)$);
3: **else**
4: **return** x;
5: **end if**





Some properties of FIND and UNION

- FIND operations change internal nodes only while UNION operations change root node only.
- Path compression changes parent node of certain internal nodes. However, it will not change the root nodes, rank of any node, and thus will not affect UNION operations.

Path compression: complexity

- Example: FIND(c)
- A FIND(c) operation might takes long time; however, the path compression makes subsequent FIND(c) (and other middle nodes in the path) efficient.

定理 B.0.1. *Starting from each item forming an individual set, any sequence of m operations (including* FIND *and* UNION*) over n elements takes $O(m \log^* n)$ time.*

Analysis of path compression: a brief history

- In 1972, Fischer proved a bound of $O(m \log \log n)$.
- In 1973, Hopcroft and Ullman proved a bound of $O(m \log^* n)$.
- In 1975, R. Tarjan et al. proved a bound using "inverse Ackerman function".
- Later, R. Tarjan, et. al. and Harfst and Reingold proved the bound using the potential function technique.

Here, we present the proof in *Algorithms* by S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani.

$\log^* n$: Iterated logarithm function

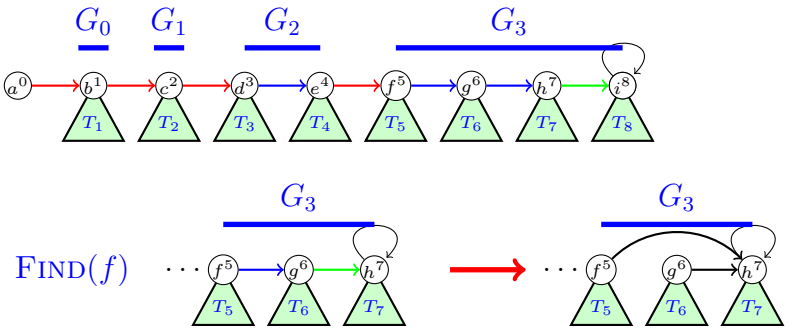
n	$\log^* n$
1	0
2	1
$[3, 2^2]$	2
$[5, 2^4]$	3
$[17, 2^{16}]$	4
$[65537, 2^{65536}]$	5

Group	Rank	Upper bound of #elements
0	1	$\frac{n}{2}$
1	2	$\frac{n}{2^2}$
2	$[3, 2^2]$	$\frac{n}{2^2}$
3	$[5, 2^4]$	$\frac{n}{2^4}$
4	$[17, 2^{16}]$	$\frac{n}{2^{16}}$
5	$[65537, 2^{65536}]$	$\frac{n}{2^{65536}}$

- Intuition: the number of logarithm operations to make n to be 1.
- $\log^* n = \begin{cases} 0 & \text{if } n = 1 \\ 1 + \log^*(\log n) & \text{otherwise} \end{cases}$
- Note: $\log^* n$ increases very slowly, and we have $\log^* n < 5$ unless n exceeds the number of atoms in the universe.

Analysis of rank

- Let's divide the nonzero ranks into groups as below.
- Note:
  - Group number is $\log^* rank$ and the number of groups is at most $\log^* n$.
  - The number of elements in the rank group G_k ($k \geq 2$) is at most $\frac{n}{\underbrace{2^{2^{\cdots 2}}}_{\text{k}}}$ as the number of nodes with rank r is at most $\frac{n}{2^r}$. We will see why the group was set to take the form $[\underbrace{2^{2^{\cdots 2}}}_{\text{k-1}}+1, \underbrace{2^{2^{\cdots 2}}}_{\text{k}}]$ soon.

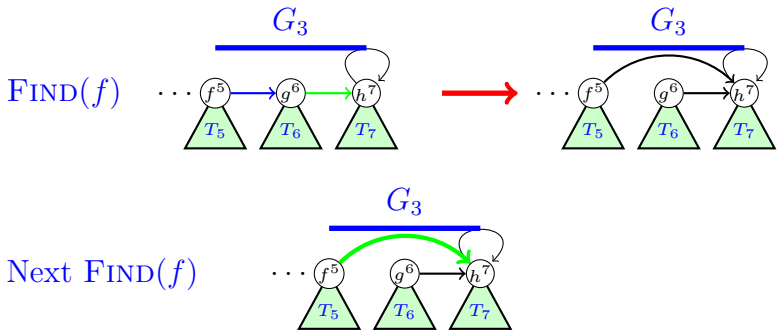


Amortized analysis: total time of m FIND operations

- Basic idea: a FIND operation might take long time; however, path compression makes subsequent FIND operations efficient.
- Let's consider a sequence of m FIND operations, and divide the traversed links into the following three types:
  - **Type 1:** links to **root**
  - **Type 2:** links traversed **between** different rank groups
  - **Type 3:** links traversed **within** the same rank groups
- For example, the links that FIND(a) travels:
- The total time is $T = T_1 + T_2 + T_3$, where T_i denotes the number of links of type i. We have:
  - $T_1 = O(m)$.
  - $T_2 = O(m \log^* n)$. (Hint: there are at most $\log^* n$ groups.)
  - $T_3 = O(n \log^* n)$. (To be shown later.)
- Thus, $T = O(m \log^* n)$.

Amortized analysis: why $T_3 = O(n \log^* n)$?

- Note that **the link $f \to parent(f)$ of type 3** in FIND(f) will change $parent(f)$: the rank of $parent(f)$ increases by at least 1. In the example shown below, $parent(f)$ changes from g^6 to h^7.



- Let's consider the next FIND(f) operation. There are two cases:

  1. If no UNION was executed before the next FIND(f) operation, $parent(f)$ is itself a root, and **the link $f \to parent(f)$** will be accounted into T_1.
  2. If a UNION operation linked h^7 to another root node, say i^8, before the next FIND(f) operation, then the next FIND(f) operation will again lead to the increase of the rank of $parent(f)$.

Case 1 of the next FIND(f): no UNION was executed before

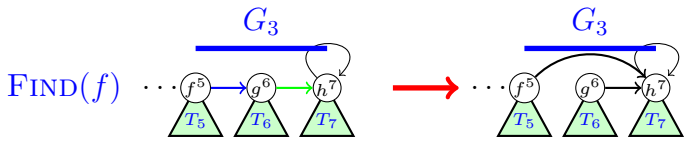
- If no UNION was executed before the next FIND(f) operation, $parent(f)$ is itself a root, and **the link from f to $parent(f)$** will be accounted into T_1.

Case 2 of next FIND(f): an UNION was executed before

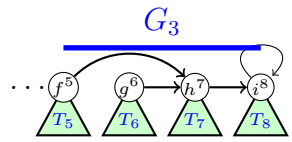
- If an UNION was executed before, the next FIND(f) will again lead to the increase of the rank of $parent(f)$, in which the **link $f \to parent(f)$ might still be of type 3**; however, we claim that the link cannot be of **type 3** over 2^4 times.

The link $f \to parent(f)$ cannot be of type 3 over 2^4 times

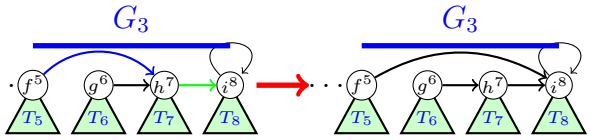
- The link $f \to parent(f)$ cannot be of **type 3** over 2^4 times since after performing at most 2^4 FIND(f),
  - $parent(f)$ is itself a root; thus, the link $f \to parent(f)$ in subsequent FIND(f) are of **type 1** and will be accounted into T_1.



UNION(f, i)



Next FIND(f)



– or the rank of $parent(f)$ increase to make it lie in another group different from f; thus, the link $f \to parent(f)$ in subsequent FIND(f) operations are of **type 2** and will be accounted into T_2.

Why $T_3 = O(n \log^* n)$? continued

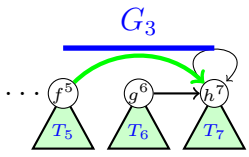
- Formally we have

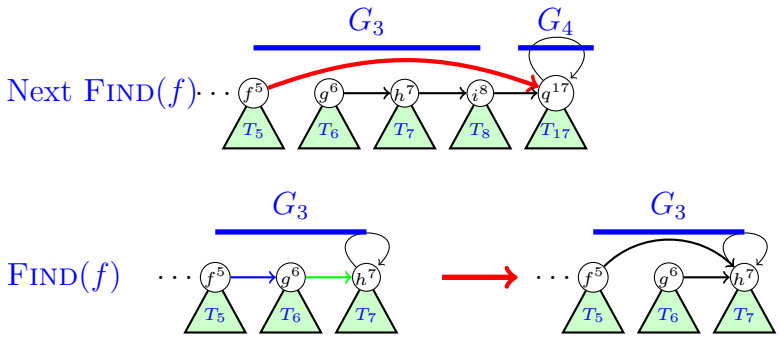
$$\begin{aligned} \mathbf{T_3} &\le \sum_{k=2}^{\log^* n} \sum_{f \in G_k} \underbrace{2^{2^{\cdots 2}}}_{\text{k}} && \text{(the largest rank in group } G_k \text{ is } \underbrace{2^{2^{\cdots 2}}}_{\text{k}}\text{)} \\ &\le \sum_{k=2}^{\log^* n} \frac{n}{\underbrace{2^{2^{\cdots 2}}}_{\text{k}}} \underbrace{2^{2^{\cdots 2}}}_{\text{k}} && \text{(\#nodes in group } G_k \le \frac{n}{\underbrace{2^{2^{\cdots 2}}}_{\text{k}}}\text{)} \\ &= O(n \log^* n) \end{aligned}$$

$T_3 = O(n \log^* n)$: another explanation using "credit"

- Let's give each node credits as soon as it ceases to be a root. If its rank is in the group $[k+1, 2^k]$, we give it 2^k credits.
- The total credits given to all nodes is $n \log^* n$. (Hint: each group of nodes receive n credits.)
- If $rank(f)$ and $rank(parent(f))$ are in the same group, we will charge f 1 credit.
- In this case, $rank(parent(f))$ increases by at least 1.

Next FIND(f)





- Thus, after at most 2^k FIND operations, $rank(parent(f))$ will be in a higher group.
- Thus, f has enough credits until $rank(f)$ and $rank(parent(f))$ are in different group, which will be accounted into T_2.