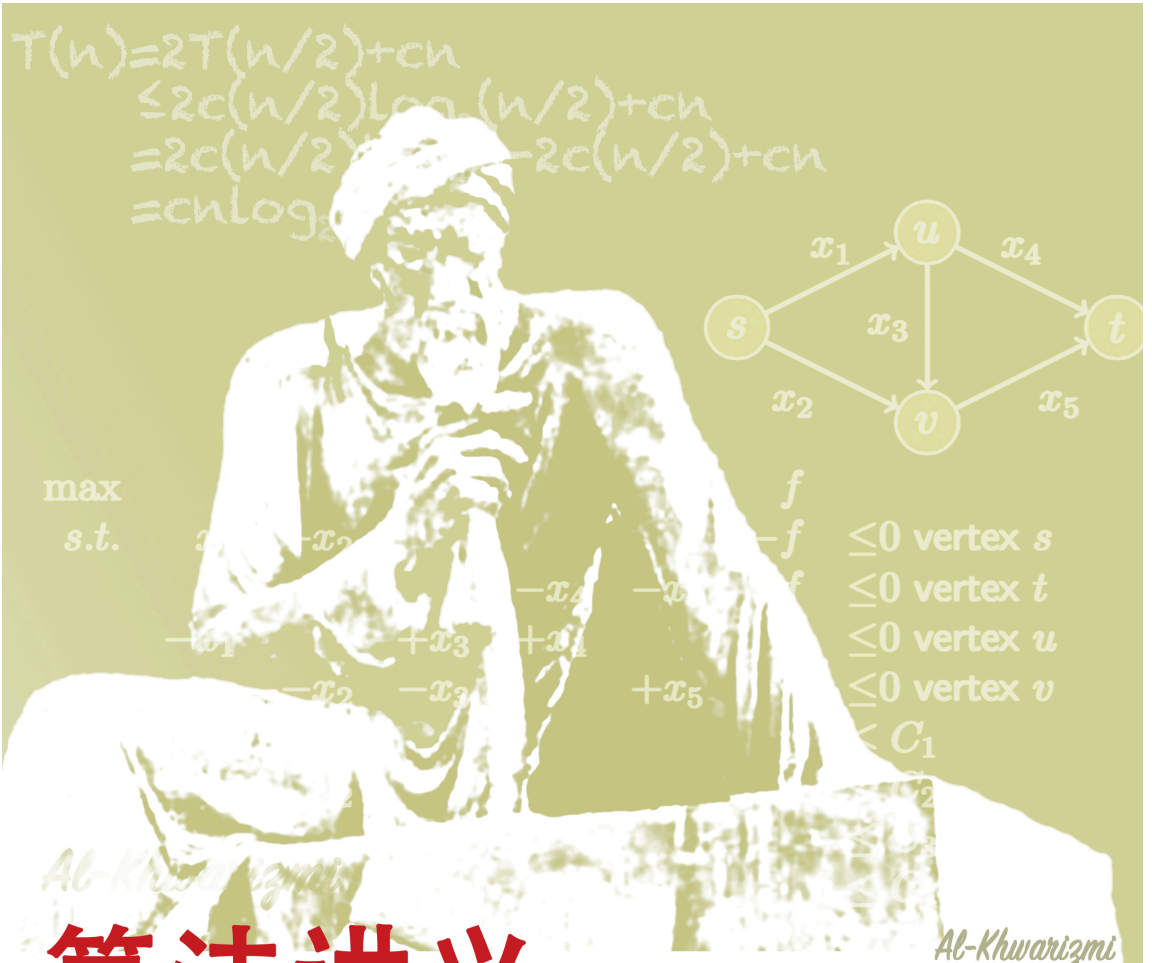


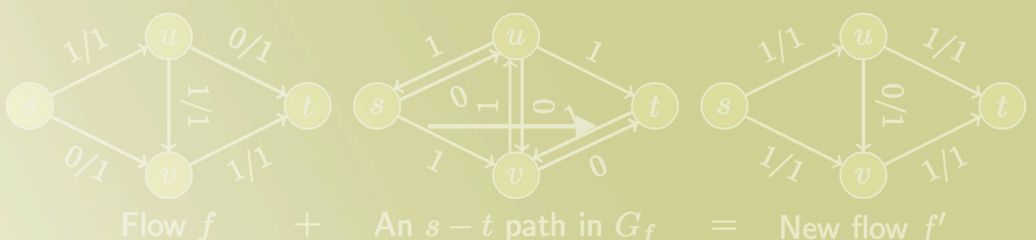
算法讲义

关于问题求解方法的十八讲

卜东波 张家琳 编著

LECTURES ON ALGORITHMS

Al-Khwarizmi



待定出版社

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。

这本书记载了一些贸易、测量、遗产分配等方面的实际问题，以及从这些实际问题出发抽象建模而成的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方法—这些方法具有清晰描述的步骤；任何人按步骤机械式地操作即可求解方程。这样的问题求解方法被称为 “Algorithm”，中文译作“算法”。Algorithm 一词来源于 Al-Khawarizmi 名字的拉丁文译法，以表达人们对他的敬意。

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专注于证明定理。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。

我们所写的这本讲义，是沿着“实际问题 ⇒ 抽象出的数学问题 ⇒ 算法设计”这条脉络总结我们的一些心得体会。如果说有一些特色的话，可能在于这本讲义不是简单地罗列现有的算法技术，而是试图强调“如何观察问题的结构”、强调“如何基于问题的结构进行算法设计”—求解问题的过程不应当只是逐个尝试各个算法技术或者纯粹依赖于灵感，而是应该依赖于我们对问题结构的认识；我们对问题结构认识得越深入，越有助于求解算法的设计。

这本讲义是沿着“首先观察问题结构、继而依据对问题结构的观察进行算法设计”这个问题求解思路来组织的。讲义分作三个大的部分，分别对应于不同的问题结构：(*i*) 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 章）和贪心算法（第 4 章）。(*ii*) 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（第 5 章）及对偶理论（第 6 章）、网络流算法及其应用（第 7, 8 章）。(*iii*) 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（第 9 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三部分，包括只要求能够得到近似解的近似算法（第 10 章）、允许算法中存在随机化行为的随机算法（第 11 章），以及实用的启发式算法（第 12 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。

我在滑铁卢大学听过 Timothy Chan 讲授的算法课。在把复杂的算法讲得清楚明白这一点上，我从 Timothy Chan 那里领会到了很多。在写作风格方面，我偏爱于《费恩曼物理学讲义》的写法：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理解本质性的东西。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、江涛等同学致以谢忱。

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。我们在此对林宇、袁雄鹰、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—或许将他们列为本书的共同作者，更能彰显他们的贡献。

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 OJ 编程题目)；这些课件可以从 `http://bioinfo.ict.ac.cn/~dbu/AlgorithmCourses/CS091M4041H/`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，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又不丢失严谨性，是讲课和写作中最难把握的，也是最让我们困惑的。在这一点上，我们始终惴惴不安，只能静候读者的建议和指正。

卜东波、张家琳

2018 年于中科院计算所、中国科学院大学

目录

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，如何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。在评估算法性能的基础之上，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 Fibonacci 数的计算为例介绍算法复杂度分析的基本思想。

1.1　问题的形式化定义

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*[1] 这本书中记载的一个真实问题：

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，我们将它们剥离出去，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, $|V| = n$, 以及结点间距离矩阵 $D = (d_{ij}) \in \mathbf{R}^{n\times n}$, 其中 d_{ij} 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，其中“输入”部分是指算法的“输入”，表示我们已知的信息，描述问题的所有参量及其格式；“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。

算法问题的一个特定的输入被称作**实例** (Problem instance)。比如图 1.1示出旅行商问题的 3 个实例，其中结点间距离标注于边上。对只包含 3 个结点的实例 (a) 来说，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；而对包含 4 个结点的实例 (b) 来说，总共有如下 6 种环游路线：

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

表 1.1: TSP 问题实例 (b) 的所有环游路线及其总里程

从表中可以看出，最短环游的总里程是 18；$a \to b \to c \to d \to a$ 是里程最短的环游之一。我们的目的是对于任意给定的 TSP 实例，能够计算出最短环游。

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。

当一个算法能够应用于问题的任何实例，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。

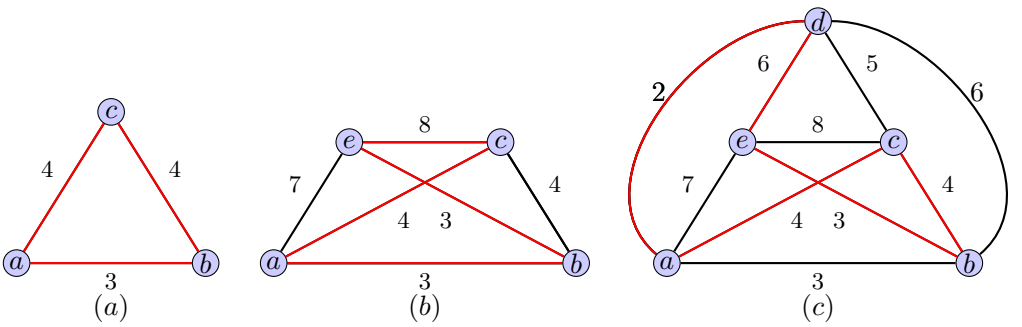


图 1.1: 旅行商问题的 3 个实例：$n = 3, 4, 5$，其中结点对之间的距离标注于相应的边上。最短环游路线使用红色示出。

1.2 算法设计的基本过程

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 [2]:

无论是算法设计还是算法的讲解，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。

或许反过来想，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？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？

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”[3]。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，目前最高效的算法都是采用逐步改进的策略 [4]。

“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”策略 (Enumeration) 设计求解算法，即产生出所有的可行解，进而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”策略 (Intelligent enumeration)，即对枚举树进行剪枝，忽略一些低质量的可行解 [5]。

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，最终形成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：有些问题迄今为止尚未找到快速的求解算法，从而被称作“难”的问题。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 [6]。

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

上述困难导致我们难以设计出直接求解 TSP 问题的分而治之算法。因此我们转换一下目标，先求解一个辅助问题：计算从结点 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 e 的里程最短路径，记为 $M(s, S, e)$。

研究这个辅助问题 $M(s, S, e)$ 有两点好处：(i) 可以基于辅助问题的解构造出 TSP 问题的解；(ii) 容易设计一个分而治之算法计算 $M(s, S, e)$。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，分别是从 b 返回 a、从 c 返回 a，以及从 d 返回 a；因此我们可以将最短

环游路线的里程表示如下：

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

上述观察可以推广到任意的 TSP 实例：如果对任意的 $s \in V$, $e \in V$, $S \subseteq V$，都能够计算出 $M(s, S, e)$, 则可以求解原始的 TSP 问题。此算法称为 Bellman-Held-Karp 算法 [7, 8]，伪代码如下：

Algorithm 1 Bellman-Held-Karp algorithm for TSP

function Bellman-Held-Karp(V, D)

Require: $|V| \geq 3$

1: **if** $|V| = 3$ **then**
2: Let's represent the nodes in V as $V = \{a, b, c\}$;
3: **return** $d_{ab} + d_{bc} + d_{ca}$;
4: **else**
5: **return** $\min_{e \in V, e \neq s}\{M(s, V - \{e\}, e) + d_{es}\}$;
6: **end if**

其中 $s \in V$ 是任意一个结点。

接下来我们看第二点，如何计算 $M(s, S, e)$。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：

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

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 $M(a, \{c, e\}, b)$：

$$M(a, \{c, e\}, d) = \min\{d_{cb} + M(a, \{e\}, c), d_{eb} + M(a, \{c\}, e)\}$$

因此我们可以设计如下的分而治之算法计算 $M(s, S, e)$：

Algorithm 2 Algorithm to calculate $\mathrm{M}(s, S, e)$

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

1: **if** $|S| = 1$ **then**
2: Let's represent S as $S = \{v\}$;
3: $\mathrm{M}(s, S, e) = d_{sv} + d_{ve}$;
4: **return** $\mathrm{M}(s, S, e)$;
5: **end if**
6: **return** $\min_{i \in S,\ i \neq e} \mathrm{M}(s, S - \{i\}, i) + d_{ei}$;

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，$M(s, S, e)$ 的计算避开了这个困难：引入不同于出发点 s 的目的结点 e 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 $M(s, S, e)$ 的原因之所在。

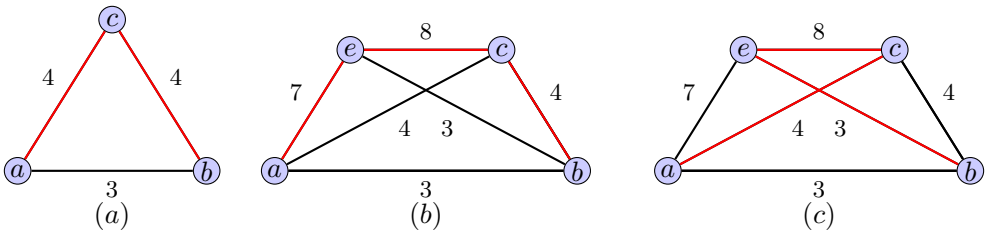


图 1.2: $M(s, S, e)$ 的计算过程。(a). 当 S 只包含一个结点 c 时，$M(a, \{c\}, b) = d_{ac} + d_{cb}$。$(b)$, (c). 当 S 包含 2 个结点 c, e 时，$M(a, \{c, e\}, b) = \min\{d_{cd} + M(a, \{e\}, c), d_{eb} + M(a, \{c\}, e)\}$。

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 Improvement algorithm for TSP

function GENERICIMPROVEMENT(V, D)

```
1: Let S be an initial tour;
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 [9]。

(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：若对当前可行解无法做进一步改进，则算法终止；或者迭代次数超过预先定义的阈值时算法结束。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 $S = a \to b \to c \to d \to a$(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 $S = a \to b \to e \to c \to d \to a$，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 $S = a \to b \to e \to c \to d \to a$ 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。

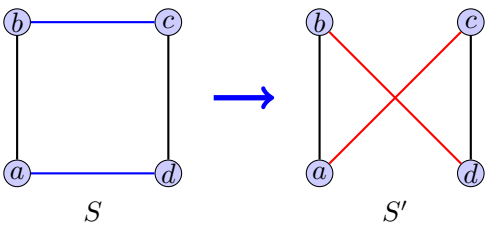


图 1.3: TSP 问题中解变换的 `2-Opt` 规则。采用此规则，将环游 $S = a \to b \to c \to d \to a$ 变换成新的环游 $S' = a \to b \to d \to c \to a$。

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，蓝色标识的环游只包括其中 5 条边，可以表

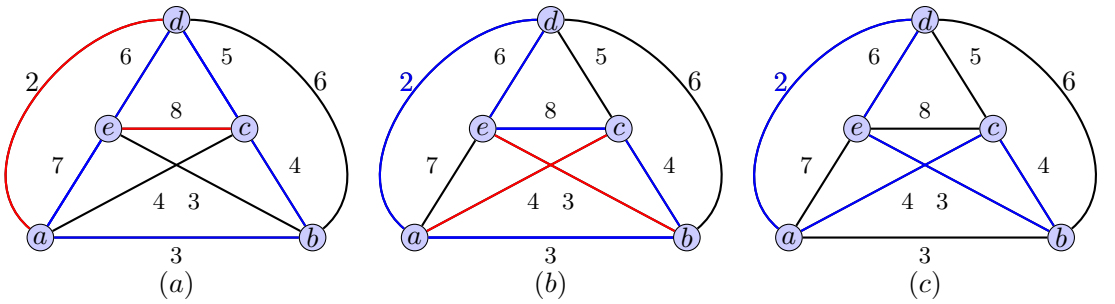


图 1.4: “逐步改进”算法求解旅行商问题的运行示例。(a) 和 (b) 显示算法执行的两次迭代改进过程，(c) 显示求得的最终解。在算法运行过程中，完整可行解用蓝色边示出，`2-Opt` 操作产生的新边用红色示出。

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 表示结点数，m 表示所有边的总数。

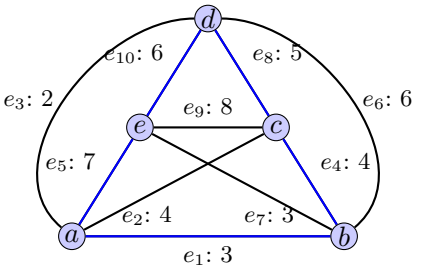


图 1.5: 旅行商问题可行解的向量表示。首先对所有边进行标号，蓝色标识的环游包含 5 条边，可以表示为 $X = 1001100101$，其中 $x_i = 1$ 表示环游经过边 e_i，$x_i = 0$ 表示环游不经过边 e_i。

用于枚举可行解的“部分解树”

对于具有向量形式的可行解，我们可以将所有的可行解组织成一棵树，称为“部分解树”(Partial Solution Tree [5, 10])。如图 1.6所示，部分解树中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。比如根结点 X =?????????? 表示部分解，其中所有的边都尚未确定是否被环游使用；结点 $X = 1$????????? 表示另一个部分解，表示仅已知环游使用了边 e_1，其他边是否使用尚未确定；叶子结点 $X = 1011000011$ 表示一个完整解，对应于环游 $a \to b \to c \to e \to d \to a$。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。

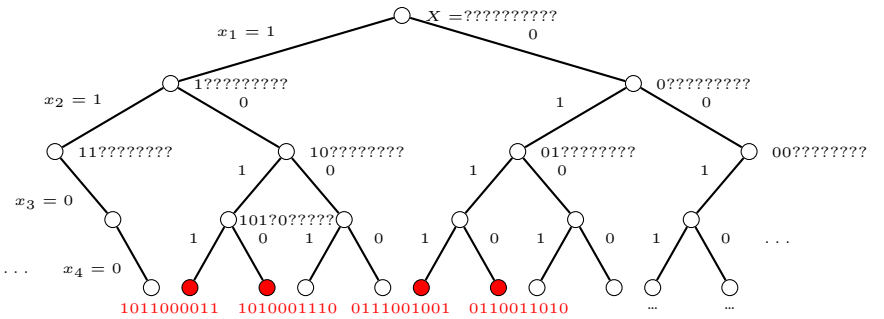


图 1.6: 枚举 TSP 可行解的部分解树。每个内部结点表示一个部分解，叶子结点表示一个完整解（即环游）；边表示设定某一个分量 x_i 的取值。

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 Generic enumeration algorithm for TSP

function GENERICENUMERATIONALGORITHMFORTSP(V, D)

Let $A = \{X_0\}$. //Start with the root node X_0 =??...?, where $|X_0| = |V|$, A denotes the active set of unexplored nodes
$best\_so\_far = \infty$;
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”，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被称为“智能枚举”(Intelligent enumeration [5])。

“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：
$$LB(X) = \tfrac{1}{2}(5+6+8+7+9) = 17.5$$
- 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：
$$LB(X) = \tfrac{1}{2}(5+6+9+7+9) = 18$$

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 Intelligent enumeration algorithm for TSP

function IntelligentEnumerationForTSP(V, D)

1: Let $A = \{X_0\}$; //Start with the root node X_0 =??...?, where $|X_0| = |V|$, and A denotes the active set of unexplored nodes.
2: $best\_so\_far = \infty$; //Store the best tour till now;
3: **while** $A \neq \{\}$ **do**
4: **Choose** a node $X \in A$ such that $LB(X) \leq best\_so\_far$, 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** $LB(X') \leq best\_so\_far$ **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 $X = abcde$。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 1.9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。

1.3 算法复杂度分析

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

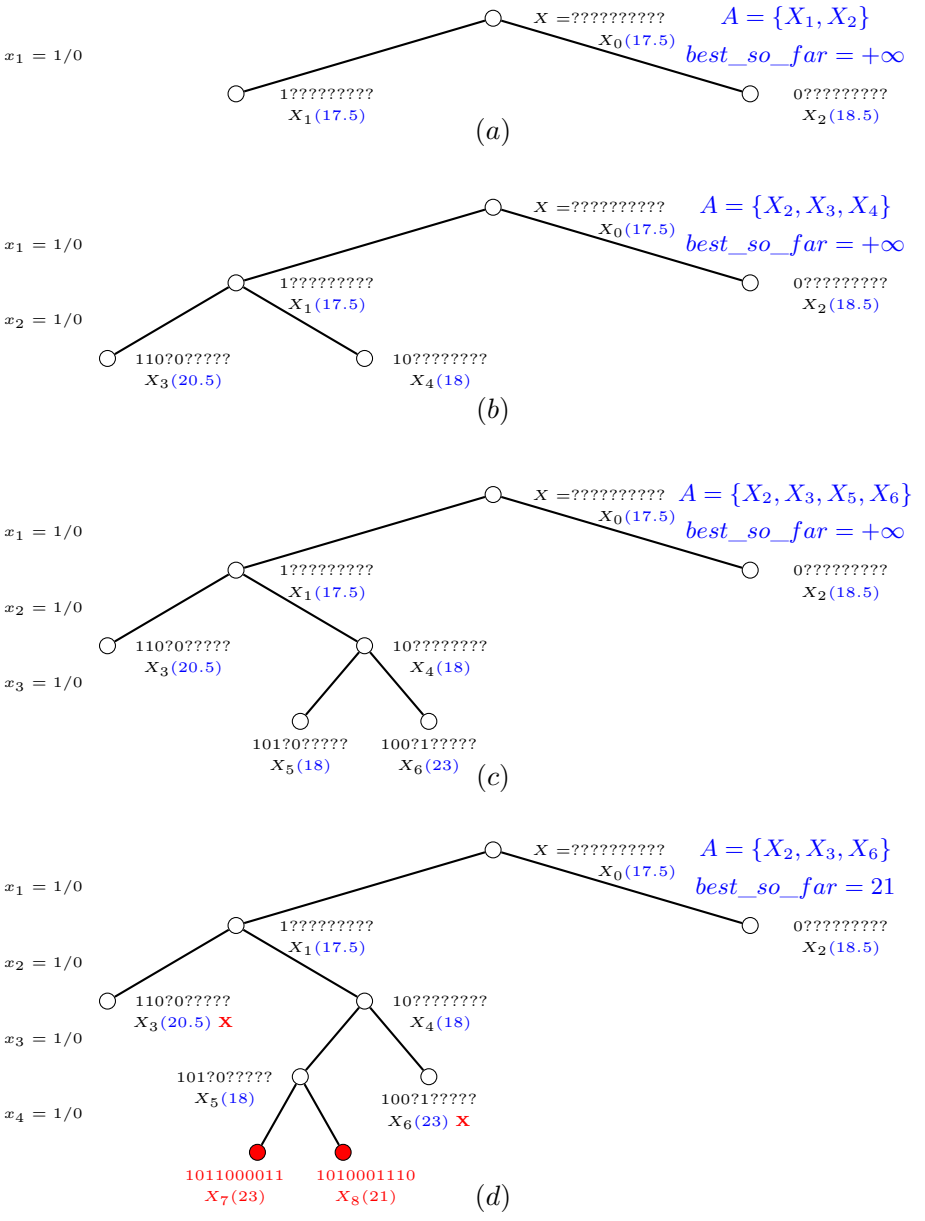


图 1.7: 算法 IntelligentEnumerationForTSP 迭代执行 1-4 轮产生的部分解树。

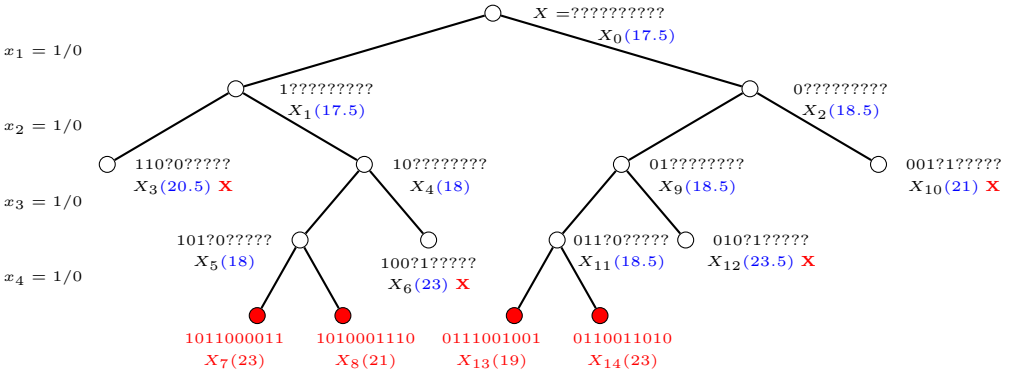


图 1.8: 算法 IntelligentEnumerationForTSP 迭代执行 7 轮后产生的部分解树。$A = \{\}$，$best\_so\_far = 19$。

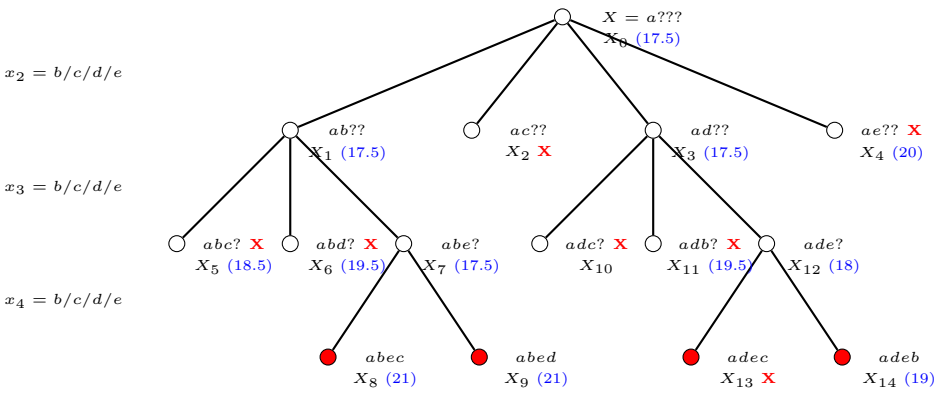


图 1.9: 算法 IntelligentEnumerationAlgoForTSP 迭代执行 6 轮后产生的部分解树。$A = \{\}$，$best\_so\_far = 19$。这里我们在 $x_2 = ac??$ 处进行剪枝的原因是仅考虑 b 出现在 c 之前的环游，以避免重复。

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：

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

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 Calculation of Fibonacci number

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 4 行）、两个数相加（第 7 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 $n > 1$ 是，上述算法需要执行两次递归调用，再加上 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。

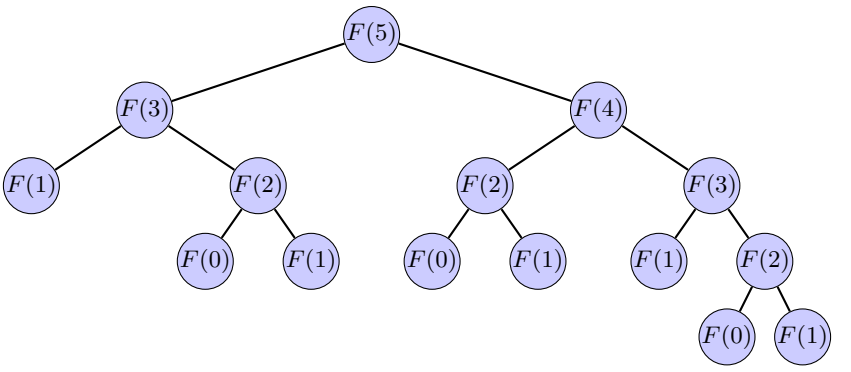


图 1.10: $F(5)$ 的递归计算过程。

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 Calculation of Fibonacci number

function FIBONACCI(n)

Require: $n \geq 0$

1: **if** $n == 0$ **then**
2: **return** 0;
3: **else if** $n == 1$ **then**
4: **return** 1;
5: **end if**
6: Allocate an array $f[0..n]$;
7: $f[0] = 0$;
8: $f[1] = 1$;
9: **for** $i = 2$ to n **do**
10: $f[i] = f[i-1] + f[i-2]$;
11: **end for**
12: **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，算法所使用的存储器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。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”。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**??**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。

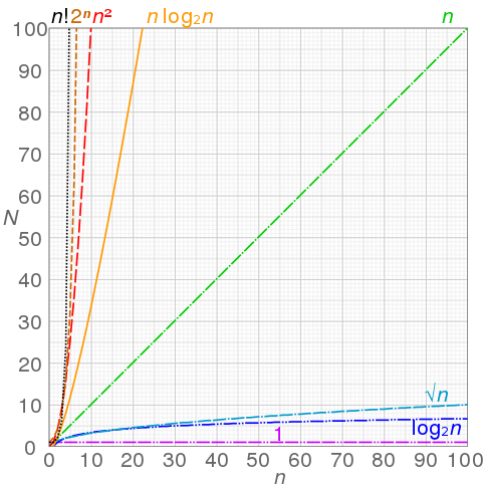


图 1.11: 常见的复杂度函数及其之间的关系。

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 (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”[11]。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 [12]：

定义 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 $100n^2 + n + 1 = O(n^2)$。

在对复杂度函数进行简化之后，我们常常能够获得形如 $O(n^c)$ 的上界，其中 c 是大于 0 的常数，这种界称作多项式界；形如 $O(2^{(n^c)}$ 的上界称作指数界。

延伸阅读

1999 年，Steven S. Skiena 做了一项关于算法需求的“市场调研”[13]。他首先建立了 Stony Brook 算法库 (http://www.cs.sunysb.edu/∼algorith)，涵盖 7 大类、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。

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 的辗转相除法，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

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。假如我们已经找到了求解最简单实例的办法，那接下来求解大的实例时的思考方向是：能否将大的实例分解（Divide）成规模较小的实例？能否将小实例的解“组合加工”（Combine）成大实例的解？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”；我们连续执行归约操作，就能将原给定实例逐步归约成最简单的实例，然后反方向操作，再将最简单实例的解逐步“组合加工”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。

进一步地，如何判断能否将小实例的解“组合加工”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。

基于归约思想算法的典型代表是“分而治之”算法（Divide and conquer）。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）、矩阵（二维序列）、集合、树、图等数据结构上的归约。

2.1　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，实例的规模可以数组大小 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 $A[1]$ 进行一次比较、必要时执行一次交换操作，即可完成排序。

下面考虑大的实例 $A[0..n-1]$（$n \ge 3$）。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分解成小的数组：一种是基于元素下标，另一种是基于元素值。我们首先来看第一种方式。

2.1.1 依据元素下标将大数组分解成小数组：插入排序与归并排序算法

即使是基于元素下标，我们也有两种方案将大数组分解成小数组，不同的分解方案导致不同的算法，分别描述如下：

第一种分解方案及相应的插入排序算法

基本思想：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 $A[0..n-2]$，以及最后一个元素 $A[n-1]$（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。

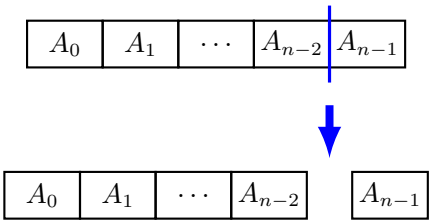


图 2.1: 依据元素下标将数组 $A[0..n-1]$ 分解成小的数组 $A[0..n-2]$ 和最后一个元素 $A[n-1]$。

在将原给定实例分解成子实例之后，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 $A[n-1]$ 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。

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），用伪代码描述如下：

Algorithm 9 InsertionSort algorithm

function InsertionSort(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
```

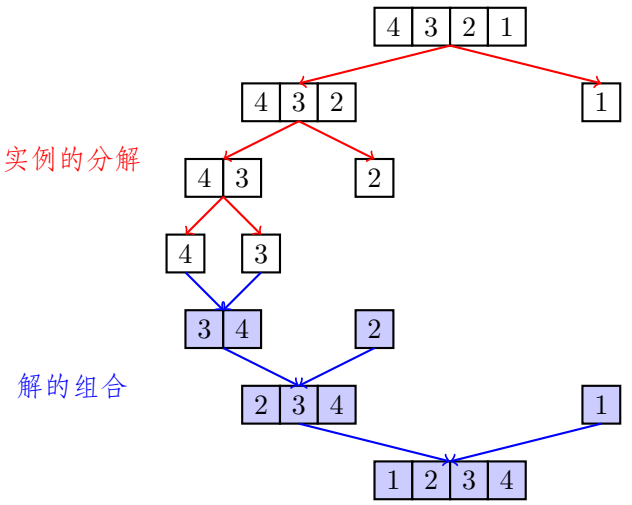


图 2.2: InsertionSort 算法对 $n = 4$ 的一个数组的排序过程。

插入排序算法的时间复杂度分析：图 2.2展示 InsertionSort 算法对 $n = 4$ 的一个数组的排序过程。由于初始数组是逆序排列的，因此在“组合”解时，共需要执行

6 次比较操作和赋值操作。我们用 $T(n)$ 表示 InsertionSort 算法的时间复杂度，得到如下的递归关系：

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

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：

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

此处的 c 是为了分析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，故而不实用。算法低效的原因是：在运行过程中，问题规模是线性下降的，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。

第二种分解方案及相应的 MergeSort 算法

基本思想：下面我们来看另外一种问题分解方法，能够使得问题规模成指数下降。如图 2.3所示，我们可以将大的数组 $A[0..n-1]$ 按下标切成规模相同的两半，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。迭代执行分解操作，即可使得问题规模成指数形式下降。

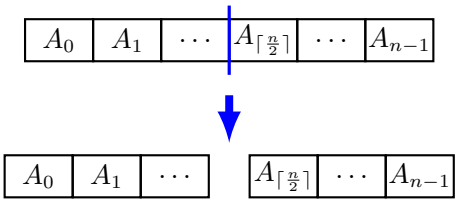


图 2.3: 依据元素下标将数组 $A[0..n-1]$ 分解成两个小的数组：左一半 $A[0..\lceil \frac{n}{2} \rceil - 1]$ 和右一半 $A[\lceil \frac{n}{2} \rceil..n-1]$。

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）[**?**]，伪代码描述如下：

Algorithm 10 MERGESORT algorithm: Sort elements in $A[l..r]$

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: **end if**

上述代码是对数组 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。算法中的 MERGE 函数完成归并操作，可以这样来直观理解：整体数组的最小元素要么是小数组 $A[l..m]$ 的最小元素（由于小数组 $A[l..m]$ 已排好序，最小元素存放于 $A[l]$ 中），要么是小数组 $A[m+1..r]$ 的最小元素（由于小数组 $A[m+1..r]$ 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做比较即可。如此循环 n 次，即可完成整个数组 $A[0..n-1]$ 的排序。归并过程见图**??**），伪代码表示如下：

Algorithm 11 Merge presorted $A[l..m]$ (denoted as L) and $A[m+1..r]$ (denoted as R)

function MERGE(A, l, m, r)

```
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:    else
7:       A[k] = L[i];
8:       i++;
9:    end if
10: end for
```

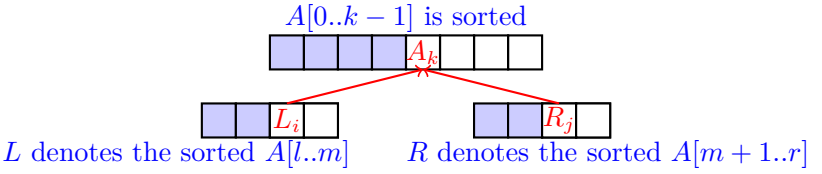


图 2.4: 归并过程：将已排好序的两个小数组 $A[l..m]$ 和 $A[m+1..r]$ 合并，并整理成有序数组 $A[0..n-1]$。

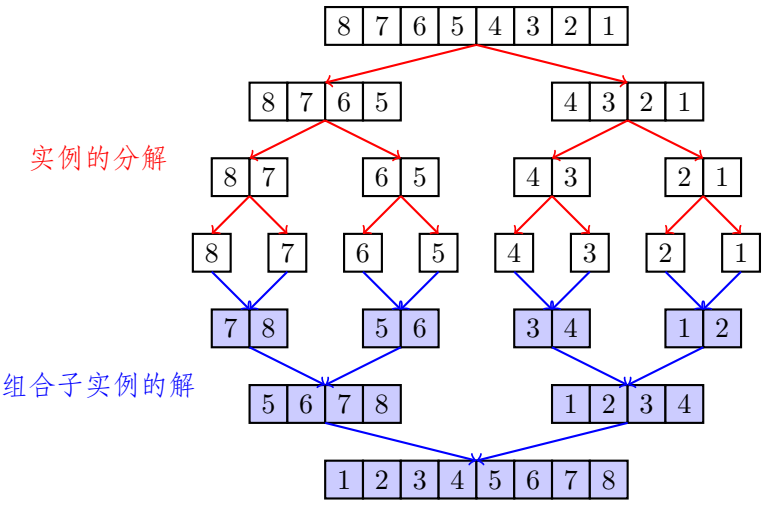


图 2.5: MergeSort 算法对 $n = 8$ 的一个数组的运行过程。

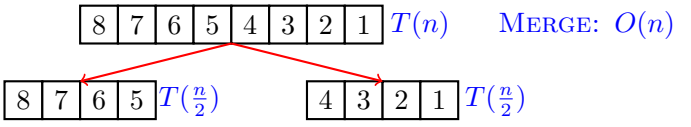


图 2.6: 归并排序算法的时间复杂度分析：以 $n = 8$ 的一个数组为例。

归并排序算法的时间复杂度分析：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 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 $2T(\frac{n}{2})$，归并时间为 $O(n)$（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，生成一颗递归树。如图**??**所示，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 MergeSort 算法相比，InsertionSort 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??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。反观 MergeSort 算法的最后一次迭代，在通过比较确定了 1 的正确位置之后，需要执行 n 次移动元素操作，才能把 1 放入正确位置。

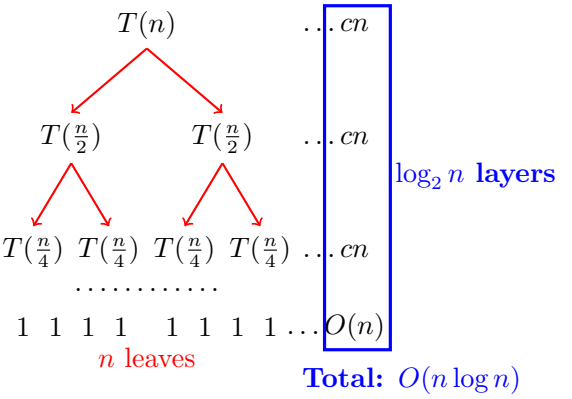


图 2.7: MergeSort 算法执行过程中的递归关系树。

2.1.2 “分而治之”算法的时间复杂度分析及 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 的实例归结 a 个子实例，每个子实例规模都相同（记为 $\frac{n}{b}$）。假如“组合”子实例解用时 $O(n^d)$，则我们可以将时间复杂度 $T(n)$ 递归表示如下：

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

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 [5]，因此只需直接套用定理即可。Master 定理陈述如下：

定理 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，可得出：

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

此处 c 表示一个大于 0 的常数。 □

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，对于子实例规模不相等等特殊情况则不适用。对这些特殊情况，我们要么使用“先尝试展开几级、观察总结规律”的方法，要么采用“先猜测上界的形式、再加以证明”的方式。我们在第 3 章将会看到这种情况的例子。

2.1.3 依据元素的值将大数组分解成小数组：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分解成小数组，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分解成小数组，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 [?]，其伪代码描述如下：

Algorithm 12 QuickSort algorithm

function QuickSort(A)

1: Create two empty arrays S_- and S_+;
2: Choose an element $A[j]$ from A **uniformly at random** and use it as pivot;
3: **for** $i = 0$ to $|A| - 1$ **do**
4: **if** $A[i] < A[j]$ **then**
5: Put $A[i]$ into S_-;
6: **else**
7: Put $A[i]$ into S_+;
8: **end if**
9: **end for**
10: QuickSort(S_+);
11: QuickSort(S_-);
12: 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，因此子实例的大小是可以控制、在运行前可以知道的。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，成线性降低。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，成指数下降。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。

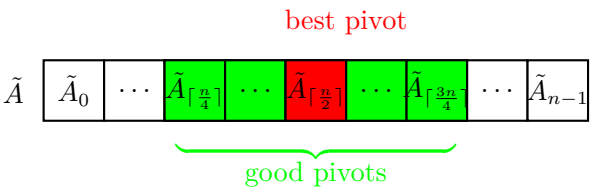


图 2.8: QuickSort 算法中的中心元的“最优选择”和“足够好”选择。

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。

ModifiedQuickSort 算法：一个复杂度易于分析的版本

Algorithm 13 ModifiedQuickSort algorithm

function ModifiedQuickSort(A)

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

和 QuickSort 算法相比，ModifiedQuickSort 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时间是 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，生成子实例的规模成指数下降：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：

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

QuickSort 算法时间复杂度分析

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，我们先指出关于运行时间的 3 点事实：

(1) 运行时间由比较次数界定：我们定义 X 为执行算法第 4 行中的比较操作的总次数。如果我们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 X 确定；我们的目标就是计算 $E(X)$。

(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）：

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

并可以将 X 表示成 $X = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}$。

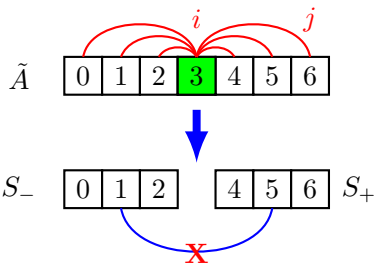


图 2.9: 在 QuickSort 算法执行过程中，$\tilde{A}[i]$ 和 $\tilde{A}[j]$ 最多只会比较一次。

(3) $\tilde{A}[i]$ 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. QuickSort 算法的期望运行时间是 $E(X) = O(n \log n)$。

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

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

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 $\tilde{A}[j]$ $(0 \le i < j \le n-1)$ 发生比较的概率是 $\frac{2}{j-i+1}$，而和数组的大小无关呢？我们以 $i = 0, j = 2$ 为例，证明 $\tilde{A}[0]$ 和 $\tilde{A}[2]$ 比较的概率是 $\frac{2}{3}$。我们先从两个简单的情况看起：

(1) 数组只包含 3 个元素的情况：

如图 2.10所示，当 $\tilde{A}[0]$ 或 $\tilde{A}[2]$ 作为中心元时，会进行 $\tilde{A}[0]$ 和 $\tilde{A}[2]$ 的比较，故有：

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

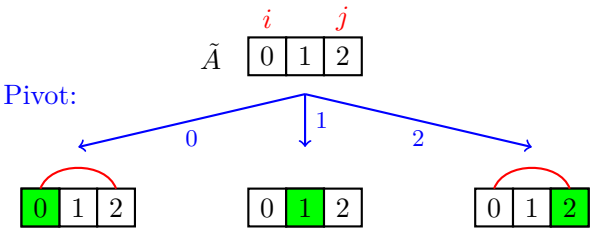


图 2.10: 对包含 3 个元素的数组 A 运行 QuickSort 算法排序时，执行过程中，当且仅当 $\tilde{A}[0]$ 或 $\tilde{A}[2]$ 被选做中心元时，会进行 $\tilde{A}[0]$ 和 $\tilde{A}[2]$ 的比较，因此比较发生的概率是 $\frac{2}{3}$。

(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：

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

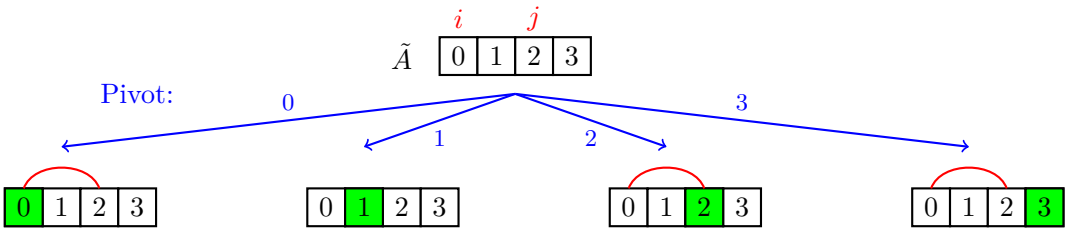


图 2.11: 对包含 4 个元素的数组 A 运行 QuickSort 算法排序时，$\tilde{A}[0]$ 和 $\tilde{A}[2]$ 发生比较有两种可能：(i) 当 $\tilde{A}[0]$ 或 $\tilde{A}[2]$ 被选做中心元时，在本轮迭代即进行比较；(ii) 当 $\tilde{A}[3]$ 被选做中心元时，$\tilde{A}[0], \tilde{A}[1], \tilde{A}[2]$ 被归入小数组 S_-，并对 S_- 执行下一轮迭代；在下一轮迭代时可能发生比较，发生概率是 $\frac{2}{3}$。

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：

$$\begin{aligned}\Pr(\tilde{A}[i] \text{ is compared with } \tilde{A}[j]) &= \frac{1}{n} + \frac{1}{n} + \frac{n-(j-i+1)}{n} \times \frac{2}{j-i+1} \\ &= (\frac{j-i+1}{n} + \frac{n-(j-i+1)}{n}) \times \frac{2}{j-i+1} \\ &= \frac{2}{j-i+1}\end{aligned}$$

从而证明了两个元素 $\tilde{A}[i]$ 和 $\tilde{A}[j]$ 发生比较的概率是 $\frac{2}{j-i+1}$，仅与元素的下标有关，和数组的大小 n 无关。

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。

xxx 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 [?]。

基本思想：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 2.12）。

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 LomutoQuickSort algorithm

function LomutoQuickSort(A, l, r)

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

Algorithm 15 Partition algorithm used by LomutoQuickSort

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;
```

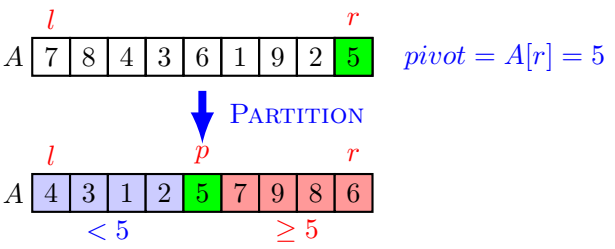


图 2.12: LOMUTO 快速排序算法中使用的 PARTITION 函数运行结果示例。

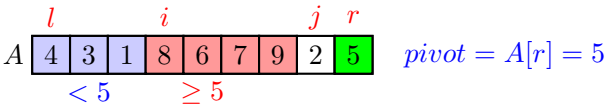


图 2.13: LOMUTO 快速排序算法中使用的 PARTITION 函数运行过程示例：当发现 $A[j]$ 比中心元小时，则交换 $A[j]$ 与 $A[i]$。注意：i 记录当前比中心元大的元素所在区域的最小下标。

算法性能比较

Hoare 做了一个有趣的实验，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。

	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

表 2.1: MERGESORT 算法和 QUICKSORT 算法性能比较 [?]。

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 QUICKSORT 算法性能不好。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。

2.2 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 [**?**]。

给定一个数组，我们可以依据逆序对的定义，用两重 `for` 循环即可计算出逆序对的数目。不过这种直接依据定义的算法的时间复杂度是 $O(n^2)$，那么能否设计出更高效的算法呢?

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。

下面考虑如何求解规模更大的实例。考虑包含 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 $A[0..\lceil \frac{n}{2} \rceil - 1]$ 和右一半 $A[\lceil \frac{n}{2} \rceil..n-1]$。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左一半中的逆序对数目、以及两个元素都在右一半中的逆序对数目。现在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；这需要对一个元素在左一半、另一个元素在右一半所构成的逆序对进行计数。

一种直接的“组合”方法如图**??**所示：检查所有的一个在左一半、一个在右一半的元素对，对其中出现的逆序对进行计数。可是这种“组合”方法的时间复杂度是 $O(n^2)$，

从而导致整个算法的时间复杂度是：

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

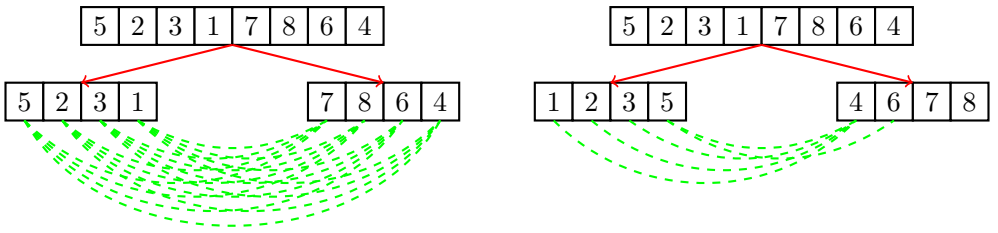


图 2.14: 逆序对计数算法中“组合”子问题的解的两种策略：

那么能否更高效地进行“组合”呢？这里的一个关键性的思考是：如果左一半和右一半是任意的、没有任何结构的，我们没有别的办法，只能采用上述简单的逐对检查策略。

SORT-AND-COUNT(A)

1: Divide A into two sub-sequences L and R;
2: (RC_L, L) = SORT-AND-COUNT(L);
3: (RC_R, R) = SORT-AND-COUNT(R);
4: (C, A) = MERGE-AND-COUNT(L, R);
5: **return** $(RC = RC_L + RC_R + C, A)$;

MERGE-AND-COUNT (L, R)

1: $RC = 0$; $i = 0$; $j = 0$;
2: **for** $k = 0$ to $\|L\| + \|R\| - 1$ **do**
3: **if** $L[i] > R[j]$ **then**
4: $A[k] = R[j]$;
5: $j++$;
6: $RC += (\|L\| - i)$;
7: **else**
8: $A[k] = L[i]$;
9: $i++$;

10: **end if**
11: **end for**
12: **return** (RC, A);

Time complexity: $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

延伸阅读

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，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，或者使用 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 [**?**]。详细地说，考察变量 X 和 Y 的 n 次采样 $x_1, x_2, ..., x_n$ 和 $y_1, y_2, ..., y_n$，Kendall 相关系数定义如下：

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

H. Huang 等对 Kendal 系数做了进一步的发展，提出了一种新的统计量，可以刻画变量的“局部秩相关性”[**?**]。

- When the data changes gradually, the goal of a sorting algorithm is to sort the data at each time step, under the constraint that it only has limited access to the data each time.
- As the data is constantly changing and the algorithm might be unaware of these changes, it cannot be expected to always output the exact right solution; we are interested in algorithms that guarantee to output an approximate solution.
- In 2011, Eli Upfal et al. proposed an algorithm to sort dynamic data.
- In 2017, Liu and Huang proposed an efficient algorithm to determine top k elements of dynamic data.

动态数据排序
曾刚

习题

1. 比较如下的复杂度函数

参考文献

[1] Timothy B. Spears. *100 Years on the Road: The Traveling Salesman in American Culture*. Yale University Press, New Haven, 1997.

[2] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin Heidelberg, 2001.

[3] Udi Manber. *Introduction to the Theory of Computation*. Addison-Wesley, Boston, 1989.

[4] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall Inc., New Jersey, 1982.

[5] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw Hill, Boston, 2007.

[6] Juraj Hromkovič. *Algorithmics for Hard Problems*. Springer, Berlin, 2002.

[7] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. In *Proceedings of the 1961 16th ACM National Meeting*, ACM '61, pages 71.201–71.204, New York, NY, USA, 1961. ACM.

[8] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *J. ACM*, 9(1):61–63, January 1962.

[9] Shen Lin and Brian W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21:498–516, 1973.

[10] 朱大铭，马绍汉. 算法设计与分析. 高等教育出版社, 北京, 2009.

[11] Michael Sipser. *Introduction to Algorithms: A Creative Approach*. PWS Publishing, Boston, 1997.

[12] Donald E. Knuth. Big Omicron and big Omega and big Theta. *ACM SIGACT News*, 8:18–24, 04 1976.

[13] Steven S. Skiena. Who is interested in algorithms and why? *ACM SIGACT News*, 30:65–74, 1999.