信手相连——面向过程的程序设计 (C 语言)

5. 高级程序设计技术（动态规划）

崔冠宇

2018202147

https://github.com/GuanyuCui/RUC-infohands-2020

信息学院
中国人民大学

2020-2021 学年秋季学期
(12.19, 18:00-20:30, 教二 2109)

目录

进度

本章的主题是高级程序设计技术（主要是动态规划与贪心算法）。

- ✓ 基础知识（计算机结构、代码风格与规范……）；
- ✓ 数据类型（长短整、单双精度浮点、字符……）；
- ✓ 控制结构（顺序、选择、循环）；
- ✓ 数组；
- ✓ 函数；
- ✓ 指针；
- ✓ 自定义类型（枚举、联合体、结构体）；
- ✓ 文件；
- → 提高知识（递归、算法复杂度、动态规划、贪心算法、数据结构）。

算法的（时间）复杂度

衡量算法效率高低的一个重要因素是它的运行时间。由于算法工作时系统有一定的不确定因素，显然是不能直接通过运行时间来衡量它的效率的，而且有时候我们在刚设计出算法时也需要对算法的效率进行评估，那么如何衡量呢？一般我们是通过算法的复杂度分析来衡量的。

对于一个规模为 n 的问题（需要说明 n 的含义。比如乘法中 n 是乘数的值？还是乘数的位数？），首先选定一些基本操作（需要说明具体是什么操作，有时候也会省略掉），认为它们能在有界的常数时间内完成；然后分析该算法内基本操作的次数，将它表示为关于 n 的函数表达式 $f(n)$。

大 O 记号 / 大 Θ 记号

一般而言，我们不关心基本操作的具体次数 $f(n)$ 为何，我们只关心它的量级（增长速度）。为了描述两个函数之间的量级关系，引入两个记号——大 O 记号和大 Θ 记号。

大 O 记号 / 大 Θ 记号

设 $f(n)$、$g(n)$ 渐进非负（即 $\lim\limits_{n\to+\infty} f(n), g(n) \geq 0$）。

$$f(n) = O(g(n)) \Leftrightarrow \exists M > 0, \lim_{n\to+\infty} \left|\frac{f(n)}{g(n)}\right| \leq M.$$

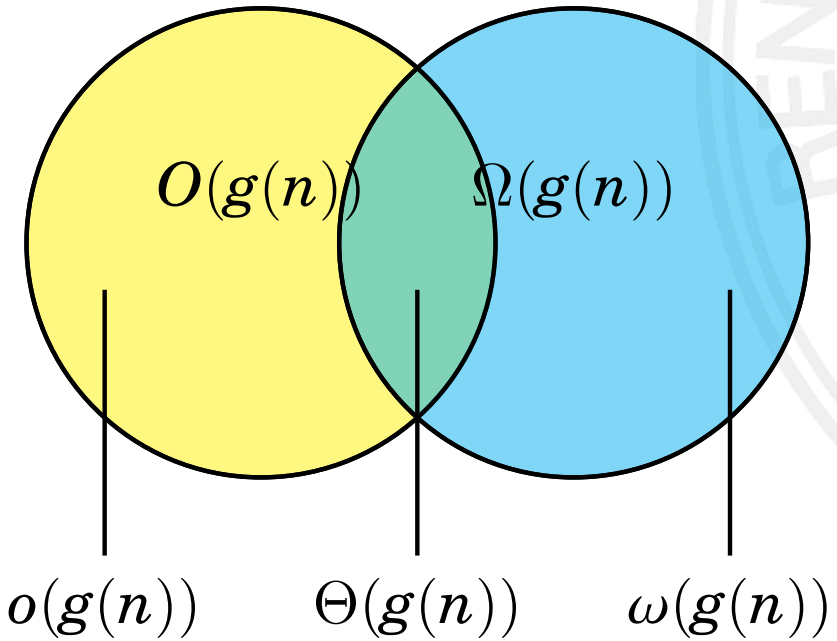
$$f(n) = \Theta(g(n)) \Leftrightarrow \exists C > 0, \lim_{n\to+\infty} \left|\frac{f(n)}{g(n)}\right| = C.$$

有时也把 $O(g(n))/\Theta(g(n))$ 看作集合。

大 O 记号 / 大 Θ 记号

为了方便，还有以下记号：

- $f(n) = \Omega(g(n)) \Leftrightarrow g(n) = O(f(n))$
- $f(n) = o(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) \neq \Theta(g(n))$
- $f(n) = \omega(g(n)) \Leftrightarrow f(n) = \Omega(g(n)) \wedge f(n) \neq \Theta(g(n))$



决定性问题与功能性问题

决定性问题是指回答为是 $YES(1)$ 或 $NO(0)$ 的问题。例如：

- $(PRIMES)$ 给定数 n，它是否是一个质数？
- (SAT) 给定合取范式 $\phi(x_1, x_2, \cdots, x_n; \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_n)$，是否有一组赋值 $< x_i > (i = 1, 2, \cdots, n, x_i \in \{0, 1\})$ 使公式 ϕ 为真？
- $(HALT)$ 给定程序 p 和输入 s，判定程序是否会在有限时间内停止？

功能性问题是指做一些操作，单一输出（不一定是真假）的问题。例如：

- $(SORT)$ 给定一组整数，输出他们的升序排序。
- (TSP) 给定图 $G = (V, E, W)$（其中 W 是边权），输出其中经过各顶点并回到起点的最短路径。
- $(KNAPSACK)$ 给定一组商品的重量 w_i 和价格 p_i，以及背包的最大承重 W，找到一组策略，最大化携带物品的价值。

*HALT*存在判定程序 → 存在一个证明利器

假设 *HALT* 存在一个判定程序 `is_halt(program, inputstr)`，对于程序 `program` 以及输入 `inputstr`，若能停机返回 `true`，否则返回 `false`。

写一段程序：

```
1 goldbach_verifier():
2     i = 6
3     while True:
4         if not can_decompose(i):
5             break
6         i += 2
```

于是 `is_halt` 成为了一个"神谕"一般的黑箱，若 `is_halt(goldbach_verifier, "") == False`，则 Goldbach 猜想为真。但是，这样的"神谕"能存在吗？

HALT 不可判定性的简单说明

很可惜，这种“神谕”程序是不能存在的。用反证法，假定存在 is_halt 程序，写一段“破坏”程序 hack，当 is_halt 成立时，陷入死循环，否则停机：

```
1 hack(program):
2     if is_halt(program, program):
3         while True:
4             pass
5     else:
6         return
```

现在问 is_halt(hack, hack) 会如何？——因此这种程序是不存在的。

归约

归约，是用来反映问题之间的转化关系和相对难度的。如果有一个问题 A，通过一个算法能在多项式时间内可以转化为另一个问题 B，就称 A 可以（多项式）归约到 B，记作 $A \leq B$。（这实际上定义了问题集合上的一个偏序关系。）

直觉上看，如果 A 能归约到 B，应该说明 B 至少不比 A 简单。

P 问题

P 是一个复杂度类，即问题的集合。

一个决定性问题 $p \in \mathbf{P}$，当且仅当该问题能被确定性图灵机以多项式时间解决（回答 YES/NO），也即 $\exists k \in \mathbb{N}$，算法的时间复杂度 $T(n) \in O(n^k)$。

P 类问题随着规模增大对资源的消耗是我们可以接受的，因此是真正有效的算法。

注意，虽然我们讨论的是决定性问题，但是可以证明对所有能在多项式时间内解决的功能性问题都能归约到决定性问题上。因此，大家也会不严谨地称某些功能性问题是 **P** 的。

常见的 **P** 问题：

1. $PRIMES$ 质数判定；
2. 求解最大公因数；

NP 问题

NP 也是一个复杂度类，即问题的集合。

一个决定性问题 $p \in$ **NP**，当且仅当该问题能被非确定性图灵机以多项式时间解决（回答 YES/NO），等价的说，就是给定一个问题的回答，存在算法能在多项式时间内验证结果的正确性。

很显然 $p \in$ **P** $\Rightarrow p \in$ **NP**，所以 **P** $\subseteq$ **NP**，但是我们仍不知道是真包含还是相等。

常见的 **NP** 问题：

1. SAT 布尔函数合取范式可满足性；
2. $HAMILTON-CYCLE$ 哈密顿圈存在性问题；

NP-Hard 问题 / NP-Complete 问题

NPH 也是一个复杂度类，表示那些比所有 **NP** 问题都要难的问题。即 $p \in \textbf{NPH} \Leftrightarrow \forall q \in \textbf{NP}, q \leq p$。

注意，一个 **NPH** 的问题不一定是 **NP** 的。若 $p \in \textbf{NP} \cap \textbf{NPH}$，则称该问题是 **NP** 完全的，记作 **NPC**，即 $\textbf{NPC} = \textbf{NPH} \cap \textbf{NP}$。

根据定义，**NPC** 问题是 **NP** 问题中最难的那些，一旦证明其中任何一个问题是 **P** 的，立刻就能得到 $\textbf{P} = \textbf{NP}$。

1971 年，Cook [1] 证明了 $SAT \in \textbf{NPC}$，随后的 1972 年，Karp[2] 利用归约一口气给出了 21 个 **NPC** 问题。

---

[1] Cook, S. A. (1971, May). The complexity of theorem-proving procedures. In Proceedings of the third annual ACM symposium on Theory of computing (pp. 151-158).

[2] Karp, R. M. (1972). Reducibility among combinatorial problems. In Complexity of computer computations (pp. 85-103). Springer, Boston, MA.

NP-Hard 问题 / NP-Complete 问题

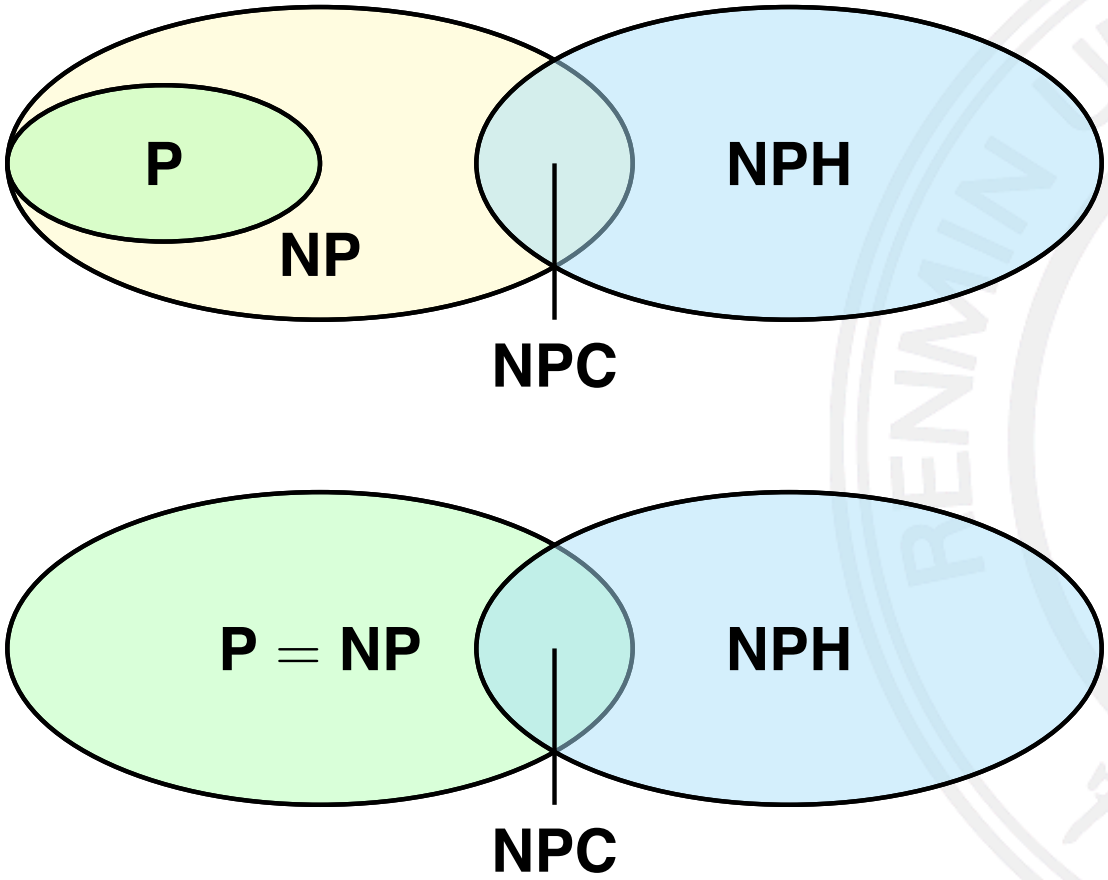


Figure: 假定 **P** ≠ **NP**（上图）/ **P** = **NP**（下图），各问题集合的关系

动态规划（dynamic programming）

动态规划是用来解决多阶段决策问题的一种方法，它也是一种通过解决子问题来解决问题的方法。与递归自顶向下解决问题的方法不同，动态规划是自底向上的解决问题的。它先计算子问题的解，然后组装更大问题的解。

因为有时递归求解会产生很多重复计算（例：斐波那契数列递归树和递推法），导致递归求解的复杂度较高，这时动态规划自底向上的优势就得以体现了。

因此，动态规划适用于子问题重叠较多的情况，同时要求动态规划具有最优子结构的性质，即原问题的最优解一定是子问题的最优解（剪贴法反证）。

一维动态规划经典问题——切割钢条

例：切割钢条

给定一段长为 n 的钢条，给定长度为 $i(i = 0, 1, 2, \cdots, n)$ 的钢条的价格表 $p_i(p_0 = 0)$，求一种切割方案（每段长为整数），使得收益最大。不考虑切割所需要的额外费用。

思路：假设第一刀下在距离左端为 i 的地方能得到最优解，只需要递归的对右半部分长为 $n - i$ 的部分求解即可。记长为 k 的钢条切割后的最大收益为 r_k，于是 $r_k = \max\limits_{0 \le i \le k}\{p_i + r_{k-i}\}$，因此长为 n 的钢条的最大收益为 $r_n = \max\limits_{0 \le i \le n}\{p_i + r_{n-i}\}$。

注意：如果用递归解决问题，重复计算太多（指数级别），所以应当采取动态规划的方法，预先保存子问题的解，自底向上填表。$O(n^2)$。（思考：如何输出解呢？记录第一段长度，递归）

（代码：第六讲代码/cutrod.c）

二维动态规划经典问题——最长公共子序列

序列 $s =< s_1, s_2, \cdots, s_n >$ 的子序列是它的一个子列 $s' =< s_{i_1}, s_{i_2}, \cdots, s_{i_k} >$（注意与子串必须连续不同，子序列可以不连续）。

例：最长公共子序列

给定两个序列 A、B，求它们的最长公共子序列。

分析：设 $c[i][j]$ 表示 $A[1...i]$ 与 $B[1..j]$ 的最长公共子序列的长度，则 $c[i][j]$ 的取值要考虑三种情况：

- 若 i 或 j 有一个为 0 时，显然 $c[i][j]$ 应该为 0；
- 若 $A[i] \neq B[j]$，$c[i][j]$ 应该为 $c[i-1]c[j]$ 和 $c[i][j-1]$ 中较大的；
- 若 $A[i] = B[j]$，则 $c[i][j]$ 应该为 $c[i-1]c[j-1]+1$。

二维动态规划经典问题——最长公共子序列

因此

$$c[i][j] = \begin{cases} 0, & i = 0 \vee j = 0 \\ \max\{c[i-1][j], c[i][j-1]\}, & i > 0 \wedge j > 0 \wedge A[i] \neq B[j] \\ c[i-1][j-1], & i > 0 \wedge j > 0 \wedge A[i] = B[j] \end{cases}$$

因为填表时 $c[i][j]$ 只依赖于 $c[i-1][j]$、$c[i-1][j-1]$ 和 $c[i][j-1]$，因此先填上 $i = 0$ 和 $j = 0$ 的部分，再按行填写。最长公共子序列长度即为 $c[len(A)][len(B)]$。时间复杂度 $O(n^2)$。

另外如何构建解呢？只有上面的第三种情况时，才是公共字符，才需要输出。保留获得 $c[i][j]$ 时是从哪个方向来的，就能恢复解。

（代码：第六讲代码/lcs.c）

二维动态规划经典问题——最长回文子序列

例：最长回文子序列

求一个序列 A 的最长回文子序列。

设 $c[i][j]$ 为 $A[i..j]$ 中最长回文子序列的长度，于是：

$$c[i][j] = \begin{cases} 1, & i = j \\ \max\{c[i+1][j], c[i][j-1]\}, & i < j \wedge A[i] \neq A[j] \\ c[i+1][j-1] + 2, & i < j \wedge A[i] = A[j] \end{cases}$$

注意填表顺序：$j - i + 1 = 1, 2, \cdots, n$。

代码留作练习。

类似题目：最大值路径条数（在一 $n \times n$ 的方格中有数字，要求从左下角走到右上角，求最大值路径条数）。

二维动态规划经典问题——矩阵链乘法

例：矩阵链乘法

给定 n 个矩阵的规模（以它们的行数、列数交错的形式给出）$p =< p_0, p_1, \cdots, p_n >$，求一种最优加括号方案，使得所做乘法数最少。

记 $m[i][j]$ 为矩阵链上第 i 个矩阵（$p_{i-1} \times p_i$）到第 j 个矩阵形成的矩阵链中最少所需乘法次数。于是有以下情况：

- 若链上仅有一个元素（$j-i+1=1$），不需要做乘法（$m[i][j]=0$）；
- 若链上多于一个元素（$j-i+1>1$），考虑在某下标 k 处断开链，于是所做乘法数为 $m[i][k]+m[k+1][j]+p_{i-1}p_kp_j$。

  另外，为了重构解，还需要设置 $c[i][j]$ 记录链切断点 k。

二维动态规划经典问题——矩阵链乘法

得到状态转移方程：

$$m[i][j] = \begin{cases} 0, & i = j \\ \min\limits_{i \le k \le j} \{m[i][k] + m[k+1][j] + p_{i-1}p_k p_j\}, & i < j \end{cases}$$

打印解的时候，若 $i \sim j$ 链上只有一个矩阵（$j - i + 1 = 1$），则可以直接打印；否则递归打印左括号、矩阵链 $i \sim s[i][j]$ 与矩阵链 $s[i][j] + 1 \sim j$ 和右括号。

（代码：第六讲代码/matrix-chain.c）

类似问题：数的最大乘积分解、数字串切分后的最大乘积。

二维动态规划经典问题——0-1 背包

例：0-1 背包问题

给定 n 个物品的价值 v_i、它们的重量 w_i 以及背包重量上限 W，求一组物品装载方案使背包内物品价值最大化。（物品只能拿或不拿，不能分割。）

设 $x_i \in \{0,1\}$ 表示该物品不取或取，于是原问题是一个优化问题：

$$\max \sum_{i=1}^{n} v_i x_i$$

$$s.t. \quad \begin{cases} \sum_{i=1}^{n} w_i x_i \le W \\ x_i \in \{0,1\} \end{cases}$$

先选择价值高的物品不一定得到最优解，反例很好构造。

二维动态规划经典问题——0-1 背包

根据 Karp 的论文[3]，背包问题的决定性版本是 **NPC** 的（原文如下）。

> 18. KNAPSACK
> INPUT: $(a_1, a_2, \cdots, a_r, b) \in \mathbb{Z}^{n+1}$
> PROPERTY: $\sum a_j x_j = b$ has a 0-1 solution.

但是我们仍然可以写出一个动态规划的伪多项式时间算法。记 $k[i][j]$ 为背包剩余容量为 j 时，仅考虑物品 $1 \sim i$ 时的最优价值。

[3]Karp, R. M. (1972). Reducibility among combinatorial problems. In Complexity of computer computations (pp. 85-103). Springer, Boston, MA.

二维动态规划经典问题——0-1 背包

最优子结构：当物品有 $1 \sim i$，背包剩余容量 j 时的最优解，一定包含了物品有 $1 \sim i-1$，背包剩余容量为 $j - w[i]$ 的最优解。

考虑 $k[i][j]$ 时有下列几种情况需要权衡：

- 当 $i = 0$ 或 $j = 0$ 时，$k[i][j] = 0$；
- 若现在的剩余空间 j 不够装物品 i（即 $j < w[i]$）时，最优解与 $k[i-1][j]$ 相同（因为没有放新物品）；
- 当空间足够时，不放物品 i 得到的价值为 $k[i-1][j]$；放物品 i 得到的价值为 $k[i-1][j-w[i]] + v[i]$；于是要选择二者较大的作为最优解。

因此，状态转移方程：

$$k[i][j] = \begin{cases} 0, & i = 0 \vee j = 0 \\ k[i-1][j], & i > 0 \wedge j > 0 \wedge j < w[i] \\ \max\{k[i-1][j], k[i-1][j-w[i]] + v[i]\}, & \text{else} \end{cases}$$

二维动态规划经典问题——0-1 背包

要得到解也是要与记录下何时进行了上面第三种选择，然后递归地打印解。

算法复杂度：$O(nW)$。这是一个伪多项式算法，因为 W 占用的二进制位随 W 线性增加而指数级别增加。

（代码：第六讲代码/knapsack.c）

贪心算法

动态规划和贪心算法的目标都是最优化多阶段决策问题。但二者不同之处在于，动态规划使用辅助空间并且考虑的比较深，而贪心算法的含义就如同它的名字一样，每一步都会选择当前看起来最优的选择。

对于有最优子结构的问题，使用动态规划可以找到最优解，但使用贪心算法则不一定。这是因为贪心算法往往过早的做出了判断。

使用贪心算法前，必须确认问题具有贪心选择性质，即贪心选择得到的解一定是最优解之一。

贪心算法能解决部分图论（如单源最短路径）问题，以及部分编码（如 Huffman 编码）问题。下面介绍几个现阶段对大家有用的题目。

贪心算法经典问题——找零钱

例子：找零钱

现在有面值为 25 美分、10 美分、5 美分和 1 美分面值的硬币若干。请你设计一种找 n 美分零钱的方案，使得找零硬币数目最少。

贪心策略：每次尽可能找币值大的硬币。

可以证明在这种币值设置下，按照贪心策略会得到最优解。但是请注意，这是一种特殊情况，并非所有币值安排都是如此。(你能举出反例吗 ?) 其实这也是硬币币值设计需要考虑的一点。

代码可以很简单地写出，此处略。

贪心算法经典问题——小数背包

例：小数背包

给定 n 个物品的价值 v_i、它们的重量 w_i 以及背包重量上限 W，求一组物品装载方案使背包内物品价值最大化。（允许拿小数比例的物品。）

小数背包也属于背包问题，但是与 0-1 背包不同，允许拿 $[0, 1]$ 间任何实数比例的物品。注意到与 0-1 背包问题不同，小数背包问题一定是可以装满背包的，而且可以使用贪心算法。

贪心策略：将各物品按价值率 v_i/w_i 降序排序，然后只要背包不满，就尽可能向背包内放价值率最高的物品。

代码可以很简单地写出，此处略。

往年期末题目讲解

2018 年期末试题：(/final/2018 期末.pdf)

- 提货单（基本 I/O、控制结构与运算）
- 获取密码（字符串操作）
- 求区间（循环搜索 + 分析缩小范围）
- 名次查询（排序，子函数）
- 单词统计（strtok 分词、字符串操作）
- 连连看游戏（图搜索，已讲解）

信院宣传——数据科学导论案例

选择信院，可以做数据大佬——
案例分享：

- 垃圾短信分类与搜索引擎构建；
- 天气之子影评分析。

抽奖环节

公开运行抽奖程序，随机生成 $1 \sim n$ 的一个排列，比照签到名单按生成的排列从上到下的顺序获奖。

奖项设置：

1. 一等奖（共一名）：*C++ Primer Plus*（中文第六版）+ 习题解 + 费列罗臻品礼盒（价值约 210 元）；
2. 二等奖（共两名）：费列罗礼盒（价值约 90 元）；
3. 三等奖（共三名）：闪迪 64GB USB3.0 U 盘（价值约 50 元）；
4. 参与奖（其他同学）：费列罗 3 颗装。

（抽奖代码：`roll.cpp`）

致谢

感谢大家一学期以来的支持。

祝大家期末考试顺利！下学期面向对象的程序设计（C++ 语言）的课程再见！