# 浅谈复杂度及其在解决问题方面的应用

南京外国语学校 杨敏行

2024 年 1 月 14 日

# 前言

- 因为笔者能力有限，所以本文的难度和创新程度可能有所不足，具有较高水平的读者可能不会有很多收获，还请多多海涵。
- 本文旨在介绍利用复杂度来解决问题的方法，提供一种可行的寻找正解的思路。其将包含从数据范围分析可能的复杂度、从几个特殊的条件推测复杂度、以及精细复杂度理论等数个方面的内容。

# 为什么要引入复杂度?

- 算法设计希望可以更快速地得到正确的答案。因此，我们需要寻找效率更优异的算法。
- 在现代计算机领域，通常认为四则运算、取模、左右移、与或非、异或、输入输出、赋值等操作，如果所操作的变量不过大，不超过计算机的字长（常见为 $2^{32}, 2^{64}, 2^{128}$ 等)，则可以认为只需一次操作即可完成。本文中理想地认为所有上述运算的效率相等，所以一份算法的效率就可以被看作是其执行的操作数目。
- 计算理论方面往往认为，输入规模更大时的算法效率是更重要的，因此，设计一个通过算法在输入规模 $n$ 充分大时的表现来衡量其效率高低的方式是有必要的。
- 因此，我们引入复杂度的概念。

# 一元复杂度的定义

- 对于两个函数 $f(n), g(n)$，如果存在实数 $M > 0$ 以及一个 $N > 0$，使得 $\forall n \geq N$，都有 $|f(n)| \leq M|g(n)|$，此时称 $g$ 是 $f$ 的渐进上界，使用 big-O-notation 记号可记作 $f = O(g)$。此时，认为函数 $f$ 有着 $g$ 或者 $O(g)$ 的复杂度。
- 注意到 $O(g)$ 其实指代了所有满足上述条件的 $f$ 构成的集合，因此正确的写法其实应该是 $f \in O(g)$，但因为历史习惯原因，通常采取 $f = O(g)$ 的写法，但是这里的 $=$ 符号仍然表示 $\in$，因此没有反向的 $O(g) = f$ 成立。

# 一元复杂度相关定理

- 易验证一元复杂度有着如下定理：
  1. 传递性：若 $f = O(g), g = O(h)$，则 $f = O(h)$。
  2. 线性性：若 $f = O(h), g = O(h)$，那么 $\lambda f + \mu g = O(h)$，其中 $\lambda, \mu \in \mathbb{R}$。
  3. 乘法结合性：若 $f = O(F), g = O(G)$，那么 $f \times g = O(F \times G)$。
  4. 常底数对数的互换：由换底公式，底数为常数的 $O(\log_a n)$ 可以互换底数，也即 $\log_b n = \dfrac{\log_a n}{\log_a b}$，而 $\log_a b$ 是在渐进复杂度中无影响的常数。因此，在研究底数为常数的对数时，我们常常忽略底数，统一记作 $O(\log n)$。

# 一元复杂度扩展定义

- 粗略定义一些常见的复杂度级别：
  - 对数多项式复杂度（$O(\text{polylog}(f))$）：复杂度是关于 $\log f$ 的多项式。
  - 多项式复杂度（$O(\text{poly(f)})$）：复杂度是关于 $f$ 的多项式。
  - 指数复杂度：复杂度是 $a^f$，其中 $a$ 是一个大于 $1$ 的实数。
- 同时，定义软复杂度概念：如果 $f = O(g(n)\text{polylog}(g(n)))$，则可记 $f = \tilde{O}(g(n))$。软复杂度的另一种定义是 $O(g(n)\text{polylog}(n))$，但是为了有 $O(n^{\alpha}\beta^{n})$ 可以被写作 $\tilde{O}(\beta^{n})$，所以采取第一种写法。

# 一元复杂度的问题

- 但是，一元复杂度也有缺憾。例如，$n$ 个数相加，一共做了 $O(n)$ 次加法，但是还需考虑加法自身的复杂度：其与值域和字长有关，因此要想精确刻画执行的操作数，仅使用一个变量可能是不太合适的。
- 另一种常见思路是，认为 $n$ 是输入规模：输入一个大小为 $A$ 的数需要 $\log_2 A$ 个 bit；这样描述虽然消减了变量数目，但是当问题的输入更加繁杂时，这样分析出的结果会比较混乱，因此我们还是有必要引入多元复杂度的概念。

# 邻域

- 在定义多元复杂度的概念前，首先需要完善邻域的概念。
- 对于 $a \in \mathbb{R} \cup \{\pm\infty\}$，$a$ 的一个邻域是指：
  - $(a-\delta, a+\delta)$，如果 $a \in \mathbb{R}$；其中 $\delta$ 是任何正实数。
  - $(N, +\infty)$，如果 $a = +\infty$；其中 $N$ 是任何实数。
  - $(-\infty, N)$，其中 $a = -\infty$；其中 $N$ 是任何实数。

# 多元复杂度

- 对于两个多元函数 $f(x_1, x_2, \ldots, x_n), g(x_1, x_2, \ldots, x_n)$，若对于确定的一组 $y_1, y_2, \ldots, y_n \in \mathbb{R} \cup \{\pm\infty\}$，有：
  - 存在一个实数 $M > 0$，使得存在 $y_1, \ldots, y_n$ 的邻域，对于该邻域中的任何 $z_1, \ldots, z_n$，都有 $|f(z_1, \ldots, z_n)| \leq M|g(z_1, \ldots, z_n)|$。

  此时，称 $g$ 是 $f$ 在 $y_1, y_2, \ldots, y_n$ 处的渐进上界，记作 $f = O(g)$ 或者 $f \in O(g)$。
- 同时，一元复杂度的对数多项式复杂度、多项式复杂度和软复杂度的概念也可以简单推广到多元复杂度。

# 多元复杂度相关定理

- 首先，易知传递性、线性性、乘法结合性、常底数对数的互换等性质可以推广至多元复杂度的场合。
- 其次，如果变量的值域是 $V$，计算机的字长是 $w = 32/64/128\dots$ 那么加减、与或非、异或、输入输出、赋值等操作的复杂度按位处理是 $O(\log_{2^w} V)$，而乘除、取模等操作，使用 FFT 算法可以在 $O(\log_{2^w} V \log_2 \log_{2^w} V)$；整理可得加减类运算的复杂度 $O(\frac{\log V}{w})$，乘除类运算的复杂度 $O(\frac{\log V}{w} \log \frac{\log V}{w})$。

# 为什么要简化复杂度?

- 复杂度可以直接代入变量的范围，得到一个对操作数目的估计。这是复杂度最常见的应用：实际应用时各个变量往往是有范围限制的，并不会真的趋向无穷。
- 比如说，在一个问题中，如果有 $n \le 10^5$，那么基本可以立刻放弃所有复杂度为 $O(n^2)$ 的做法，并且认为所有复杂度为 $O(n \log n)$ 的算法基本上都是可以通过的。复杂度就像一个标签，它帮助我们快速过滤掉一批算法，并判定另一些算法有成为正解的可能。
- 标签之间也有好坏之分：例如 $O(n)$ 就明显比 $O(\frac{n \log A}{w})$ 要更加简洁。而假如在给定的数据范围下，$\frac{\log A}{w}$ 一项的值不算很大，那么把它看成一个常数也未尝不可。比如说，在前述的 $n$ 个数求和问题中，一个常见的数据范围是 $n \le 10^5, A \le 10^9, w = 32$，那么就有 $O(\frac{n \log A}{w}) \in O(\frac{n \log 10^9}{32}) = O(n)$。

# 如何简化复杂度

- 从另一个角度说，也有 $O(\frac{n\log A}{w}) \in O(\frac{10^5\log 10^9}{32}) = O(1)$。但是，此时这个 $O(1)$ 的标签就不是一个合适的标签，因为向其中代入得到的结果并非对实际操作数的合格估计。
- 这说明，将复杂度式子中一些量看成常数有助于简化问题，但是如果不分青红皂白就将一切变量都看作常数，则并非一个好的近似。
- 同样的简化复杂度形式的操作还有，将一些变量用其它变量表示 (比如说，简单图的边数 $m$ 数目不超过 $n^2$，于是涉及到 $m$ 的复杂度 $f(m)$ 可以被写作 $f(n^2)$)；将数据范围相同的变量合并（比如说一道题元素个数 $n$ 和询问次数 $q$ 的范围都是 $10^5$，那么它们就可以统一以 $n$ 表示)。这些操作以减少复杂度的泛用性和严谨性为代价，大大降低了复杂度描述的繁琐度。

# 前提

- 信息学竞赛遇到的问题与实际生活中遇到的问题相比，区别之一是竞赛的问题总是有解的。这意味着存在一个算法可以在题目的数据范围下通过问题，于是数据范围也可以成为辅助我们寻找答案的工具之一。
- 从数据范围到算法，我们需要复杂度作为桥梁。所以我们才需要适当地对复杂度加以简化，否则枚举各种复杂得毫无意义的复杂度是舍本逐末的。
- 出于同样的道理，我们仅考虑最基础的场合，即认为输入仅有一维 $n$ 是重要的。至于更多变量的场合，如果不存在乘积量，可以尝试对于每个变量分开分析，因为复杂度具有线性性。否则，可以尝试把乘积整体当作变量分析。如果上述分析无法成立，则这种过于复杂的场合或许并不是根据数据范围反推复杂度的合适情景。另外，下文仅提供思路的参考，具体问题还是应该具体分析。

# 数据范围对应的复杂度

- $n \le 1 \sim 12$：可能对应着一些比较高的复杂度，例如 $4^n, 5^n$ 或者 Bell$(n)$、$n!$ 等。当然，这些复杂度都可以是软复杂度。
- $n \le 15 \sim 17$：可能对应着各种大到小 poly$(n)$ 系数的 $\tilde{O}(3^n)$，或者具有较大 poly$(n)$ 系数的 $\tilde{O}(2^n)$。
- $n \le 20$：$2^n$，带有一般来说不超过二次的 poly$(n)$ 系数。需要注意的是，也有一种偶见的 $\binom{n}{n/2}$ 的复杂度，例题如 UNR#6 D2T2 神隐、九省联考 2018 一双木棋，可自行阅读论文查看相关解法。
- $n \le 26 \sim 28$：这是 $O(2^n)$ 复杂度的上限。需要注意的是，有时复杂度可以是 $\sum\limits_{i=0}^{n} 2^i = O(2^n)$。

# 数据范围对应的复杂度

- $n \leq 30 \sim 60$：这些数据范围往往不会有多项式复杂度。例如 DP on DP、划分数 DP、剪枝后的搜索、meet in middle 等算法是此区间内较主流的算法。
  同时，对于所有的非多项式复杂度算法，永远都存在一种搜索所有本质不同的状态并通过压缩、分类等操作将状态数削减至可以承受的范围的方法。分析此种方法的复杂度往往是毫无意义且与实际大相径庭的，此时最佳策略莫过于亲自写一份暴力得到状态数：可能会发现真实的状态数其实出乎意料得小。大部分 DP on DP 题都能在这种策略下找到思路。
- $n \leq 50 \sim 80$：可能存在一些具有高次多项式复杂度（例如 $\tilde{O}(n^6)$ 或者 $\tilde{O}(n^7)$）的 DP 在此范围内。需要注意的是，高次多项式复杂度往往可能拥有一个较小的常数（因为循环嵌套时，内层循环的起讫值依赖于外层循环，进而会产生一个小于 $1$ 的常数），所以此时在理论的大 $O$ 复杂度内代入 $n$ 得到的结果和实际运行的效率可能会有较大的分野，因此有时不能因为理论复杂度大就贸然放弃一个算法的可能性。

# 数据范围对应的复杂度

- $n \le 100$：随着计算机性能的提高，近年来 $n \le 100$ 的数据范围往往已经对应着 $O(n^4)$ 的复杂度了。
- $n \le 200 \sim 500$：可能对应着带有不同级别 polylog$(n)$ 的 $\tilde{O}(n^3)$、$O(n^{3+\varepsilon})$（其中 $\varepsilon \in (0,1)$）、直到严格的 $O(n^3)$。需要注意的是，$n \le 150 \sim 200$ 可能有一种特殊的技巧，将在下文提到。
- $n \le 1000 \sim 10000$：带有不同级别的 polylog$(n)$ 的 $\tilde{O}(n^2)$、$O(n^{2+\varepsilon})$、直到严格 $O(n^2)$。除此之外，在 $n \le 1000$ 左右，有一些复杂度为 $\frac{n^3}{w}$ 的使用 bitset 的做法，或者复杂度为 $\frac{n^3}{\log n}$ 的做法。

# 数据范围对应的复杂度

- $n \leq 5 \times 10^4$：这是一个几乎所有 $\tilde{O}(n^{1+\varepsilon})$ 都可以通过的复杂度级别；虽然有一些例如 $O(n\sqrt{n}\log^2 n)$ 的复杂度或许不太可行。在 $5 \times 10^4 \sim 5 \times 10^5$ 的范围内，$O(\log n)$ 和 $O(n^{0.25})$ 的表现通常是相近的，虽然前者具有较优的理论复杂度，但是后者（通常在 $O(n\log^2 n)$ 和 $O(n\sqrt{n})$ 作比较时）往往有着较小的常数，综合而言，二者效果一般大致相同。同时，像 $O(n\sqrt{n}\log^2 n)$ 或 $O(n\log^4 n)$ 之类的复杂度，其对应的算法即使效率或许可以承受，代码的复杂程度也是不可接受的。

  另外，存在一些 $O(\frac{n^{t=1/2/3}}{\log n})$ 的算法，比如说 The Method of Four Russians。$t=2$ 时的该技巧有可能在这种数据范围下应用。
- $n \leq 10^5$：例如 $n\sqrt{n}\log n, n\log^3 n$ 这样的复杂度有可能通过这一级别，前提常数必须较小。另外，此数据范围下一个不可忽略的算法是 $\frac{n^2}{w}$ 的 bitset 算法。同时，这往往是 KDT 能处理的数据范围的极限：KDT 具有过大的常数。

# 数据范围对应的复杂度

- $n \le 2 \times 10^5$：$n\sqrt{n \log n}$ 有可能通过本数据范围。常数中等的 $n\sqrt{n}$ 亦可。某些常数很大的 $n \log^2 n$ 可能只能通过这种范围。
- $n \le 5 \times 10^5$：常数很小的 $n\sqrt{n}$ 有可能可以通过。常数适中的 $n \log^2 n$，以及 LCT 这种常数较大的 $n \log n$ 也会有这种数据范围。
- $n \le 10^6$：常数较小的 $n \log^2 n$（例如二分、树状数组、排序、分治的 $\log n$），常数不过大的 $n \log n$。
- $n \le 2 \times 10^6 \sim 5 \times 10^6$：常数中等至很小的 $n \log n$。也可能是某些亚 $n \log n$ 做法，例如 $n\sqrt{\log n}$，或者 $O(n\alpha(n))$ 的并查集。
- $n \le 10^7 \sim 5 \times 10^7$：往往意味着 $O(n)$ 的复杂度，其中有可能同时包含着对空间的限制。

# 数据范围对应的复杂度

- $n \le 10^8 \sim 10^9$：往往任何低于 $O(n)$ 的复杂度都可以通过，但一般 polylog$(n)$ 不会只出这么小的复杂度。某些分块打表的做法也可能通过这种数据范围。筛法是这种数据范围的常客：其中 $O(n^{2/3})$ 的杜教筛往往在这种范围下出现。
- $n \le 10^{10} \sim 10^{11}$：一般是 min25 筛的范围，其它算法出这个级别的比较少见。
- $n \le 10^{12} \sim 10^{14}$：常见筛法中能通过这种范围的或许只有 $O(\sqrt{n})$ 的 powerful number 筛。其它 $\tilde{O}(\sqrt{n})$ 的算法也可能出这个数据范围。
- 更大的 $n$：此时往往不会有 $n^\alpha$ 的复杂度（但或许可能会出现 $n^{1/3}$ 之类），更常见的还是 polylog$(n)$，例如数位 DP。这种数据范围下，一种可行的分析方式是取 $n' = \log n$ 然后对 $n'$ 展开如上文的分析。

# 总结

- 上述数据范围对应的复杂度仅供参考。同时，经常也能见到一些题目，如果复杂度并非题目考察的关键，$n \log n$ 的复杂度只出 $n \leq 10^5$ 的数据范围也是有可能的：也即，复杂度可以向下兼容数据范围，但不能向上。同时，一些很复杂的复杂度可能是光靠猜想不可能得到的，不能想当然地认为复杂度只有上述列的几种。笔者希望达到的效果是，看到 $10^6$ 不去考虑根号算法，看到 $10^5$ 记得试一试 bitset 优化，看到 $40$ 想想 meet in middle，而不是遇到问题就来查表。

# 前言

- 本节将介绍一些很有特征的条件，其常常对应着一些比较优秀的复杂度。

# 很小的 $K$

- 一些问题要求统计 $n$ 个元素中，所有大小不超过 $K$ 且额外满足某些特殊性质的元素集合，对应的某种函数的最值。如果这个 $K$ 很小，一种常见的方法是对元素随机染色，计算不同颜色间的上述集合的最值。答案集合期望在不过多次随机内被正确染色，进而被求出。
- 例：恰经过 $K$ 个不重复点的最优路径。其余例题可见论文。

# $n \times m \le$ 定值

- 这是一类常见限制，令 $n \times m \le C$，则 $\min(n, m) \le \sqrt{C}$。于是，算法可能包含一些与 $\sqrt{C}$ 有关的复杂度，例如 $\tilde{O}(2^{\sqrt{C}})$ 或者 $O(C\sqrt{C})$ 之类。
- 此时，解法往往与 $n, m$ 二维构成的网格图有关：设计算法分别从 $n, m$ 两维进行处理。有时 $n, m$ 两维是对称的因此只需设计一个算法，但更多的时候是不对称的。
- 例：网格图上状压，状压 $n, m$ 较小一侧的结果。其余例题可见论文。

# $\sum_{i=1}^{n} a_i = A$ 的启示

- $\sum\limits_{i=1}^{n} a_i = A$ 的条件能带来很多有趣的结论：
  - 不同的 $a_i$ 数目是 $O(\sqrt{A})$ 的。
  - 上一条可以扩展为，所有的 $a_i$ 可以分为两类：$\leq \sqrt{A}$ 的，和 $O(\sqrt{A})$ 个大于 $\sqrt{A}$ 的。
  - 如果额外有 $a_i \leq n$（例：$a_i$ 是简单图的边数），那么 $\min a_i$ 是 $O(\sqrt{A})$ 的。
- 这几条限制都有着很多用处。第一条可以用于构建压缩 trie（Radix Tree，Radix 树）：用长度集合为 $a_i$ 的串建立的压缩 Trie 的深度是 $\sqrt{A}$ 级别的，进而一些暴力算法可能有较优的复杂度；第二条是很经典的根号分治的思想，在包括众数问题、图论等情形都经常被应用；第三条体现在降低一些搜索问题的复杂度，其应用可见论文。

# 为什么要研究精细复杂性问题

- 除了从数据范围来推理复杂度，还可以从待解决的的问题来推理复杂度。在思考问题的时候，一个常见的思路是将原问题转化成其它的问题；而转化至的问题有可能是一些经典的、有着复杂度下界的问题。因此，牢记这些经典问题的下界，也可以助力迅速判定原问题转化的思路是否正确。
- 这方面的理论，被称作 Fine Grained Complexity 问题，即精细复杂性问题。本节将介绍数个精细复杂性理论中较为常用的结论。
- 本章专注于实用解的概念，基本不关注在信息学竞赛方面几乎没有实用性的做法。因此，一些问题给出的复杂度可能与现在提出的最优复杂度不同。相反，此处认为 bitset 优化是很有价值的，并会特别指出某些算法存在 bitset 优化的解法。

# 矩阵乘法的复杂度

- 两个 $n \times n$ 矩阵的乘法是精细复杂性问题中极其基础的一个环节。截至完稿日，现行复杂度最优的矩阵乘法已可以在 $O(n^{2.37188})$ 的复杂度内解决，将来也必然存在更优秀的算法。但是，在信息学竞赛方面，基本上任何 $\tilde{O}(n^{3-\varepsilon})$ 的算法都实现过于复杂或常数过大，基本不实用。
- 因此竞赛方面认为两个 $n \times n$ 的矩阵乘法的实用复杂度是 $O(n^3)$。特别地，0/1 矩阵的乘法（其中 0/1 间的乘法看作与，加法看作或）可以用 bitset 优化至 $O(\frac{n^3}{w})$，其中 $w$ 是计算机字长。

# 可规约至矩阵乘法的简单问题

- 有若干问题都是可以归约至矩阵乘法的，例如行列式、求逆、有向图上传递闭包、高斯消元。因此，这些问题同样只有 $O(n^3)$ 的实用解，其中传递闭包更常见的是 bitset 优化解法。
- 除此之外，还存在一种方法，其将问题弱化或转化，使得转化后的问题一旦被解决则矩阵乘法被解决。于是原问题不弱于矩阵乘法问题，进而其复杂度也就拥有了下界。这种手法被称作规约矩乘。

# 规约矩乘的例子

- 例：给定一棵 $n$ 个点、点上带颜色树，多次询问链上不同颜色数目。
- 令 $m = \sqrt{n/2}$。考虑自根挂下 $2m$ 条长度为 $m$ 的链，令 $A_{i,j}$ 表示第 $i$ 条链上有无第 $j$ 种颜色，$B_{j,i}$ 表示第 $i+m$ 条链上有无第 $j$ 种颜色 (其中 $1 \le i, j \le m$)，则一切 $A, B$ 矩阵都可以被构造。而对第 $i$ 条链底点和第 $j$ 条链底点的询问，就等价于询问 $A, B$ 两矩阵乘积中下标为 $i, j$ 处的值，询问 $m^2$ 次即可问出整个矩阵。因此，如果原问题中点数和询问数同阶，则原问题的最劣情况不弱于计算 $m \times m$ 矩阵乘法，那么其不存在低于 $O(m^3) = O(n\sqrt{n})$ 的实用解。
- 可以发现，规约矩乘是一个很强大的工具，运用得当的话可以排除相当一部分错误的做法：假如用规约矩乘证明了复杂度不低于 $n\sqrt{n}$，显然再去考虑 polylog 解法就是很荒谬的。
- 其余可以用规约矩乘证明的例子有区间逆序对计数、区间相等对计数等。

# LCS 问题

- 给定两个长度为 $n$ 的串，寻找其最长公共子序列。
- 结论：不存在低于 $O(n^2)$ 的解。特别地，存在 $O(\frac{n^2}{w})$ 的 bitset 做法，读者可自行查找 LOJ#6564 的题解。

# $(\min, +)$ 卷积

- 给定长度为 $n$ 的序列 $a, b$，计算 $c_k = \min\limits_{i+j=k}\{a_i + b_j\}$。
- 结论：不存在低于 $O(n^2)$ 的解。特别地，如果 $a, b$ 满足凸性，可以用类 Minkowski 和做法做到 $O(n)$。

# KSUM 问题

- 在长为 $n$ 的数列中，寻找大小恰为 $K$ 的零和子集。
- 3SUM、4SUM 都只有 $O(n^2)$ 做法。还有一些问题与之等价，例如长为三的等差数列计数、三点共线计数、三线共点计数、$a_i + a_j = a_u + a_v$ 计数等。
- 需要注意的是，存在与 $A = \max a$ 有关的高速做法，例如朴素的 FFT 卷积做法，当 $K = O(1)$ 时，结合容斥可以做到 $O(A \log A)$。

# APSP 问题

- 全源最短路问题，其复杂度下界是 $O(n^3)$ 的 Floyd 算法。特别地，使用 bfs，可以达到 $O(n^2w)$ 的时间复杂度，其中 $w$ 是最大边权。
- 一个与其类似的问题是矩阵的 $(\min, +)$ 乘法，复杂度下界也是 $O(n^3)$。

# 进一步学习精细复杂度理论

- 读者可自行在网上搜索相关内容，或者参阅 MIT 的线上课程资料 https://people.csail.mit.edu/virgi/6.s078/。

# 致谢

谢谢大家