arXiv:2303.07229v1 [cs.DS] 13 Mar 2023

一般字母表上的最优平方检测

Jonas Ellert*[1]、Pawe-l Gawrychowski†[2] 和 Garance Gourdel†[3]

[1]德国多特蒙德工业大学[2]波兰弗罗茨瓦夫大学计算机科学研究所[3]法国巴黎文理研究大学 DI/ENS、IRISA Inria 雷恩中心

摘要

平方（形如 xx 的片段，其中 x 是某个字符串）可以说是字符串中最自然的重复类型。关于平方的基本算法问题是检查给定的长度为 n 的字符串是否无平方子串，也就是说，是否不包含这种形式的片段。Main 和 Lorentz [在 J. Algorithms 1984] 中为这一问题设计了一个 $\mathcal{O}(n\log n)$ 时间算法，并在所谓的一般字母表假设下证明了与之匹配的下界；该假设意味着算法只能检查两个字符是否相等。不过，他们的下界还假设字符串中有 $\Omega(n)$ 个不同符号。他们提出了一个开放问题：如果限制字母表的大小，能否设计更快的算法？Crochemore [在 Theor. Comput. Sci. 1986] 中为常数大小字母表设计了一个线性时间算法；结合较新的结果，其方法实际上可以推出适用于可线性排序字母表的算法。最近，Ellert 和 Fischer [在国际自动机、语言与编程研讨会 2021] 上设计了一个适用于一般有序字母表的线性时间算法，大幅放宽了这一假设；也就是说，假定字符之间存在线性序，且可以在常数时间内进行序比较。然而，Main 和 Lorentz 于 1984年提出的开放问题，对于一般（无序）字母表仍未解决。本文表明，对于大小为 σ 的一般字母表上的长度为 n 的字符串，测试其无平方性需要 $\mathcal{O}(n\log\sigma)$ 次比较，且无法用少于 $o(n\log\sigma)$ 次比较完成。与此结果相辅相成，我们在 Word RAM 模型中给出一个 $\mathcal{O}(n\log\sigma)$ 时间算法。最后，我们将该算法扩展到以相同的复杂度报告所有 runs（极大重复）。

1 引言

重复是词组合学和字符串算法中的一个核心概念。在这一语境中，词或字符串只不过是由某个有限字母表Σ中的字符组成的序列。在最基本的情形下，重复由同一片段连续出现两次（或更多次）构成。重复不仅在纯理论层面上具有研究价值，在生物信息学 [51]中也十分重要。重复可以是平方（定义为同一片段连续出现两次）、高次幂串（例如立方串），也可以是 run，即长度尽可能大的周期子串。例如，在 bananananas 中，anan 和 nana 都是平方，且各有两处出现位置；它们同属于同一个 run：ananana。本文首先聚焦于平方，随后将我们的结果推广至 runs。

*部分经德国研究联合会（DFG）通过协同研究中心 SFB 876 的项目 A6 资助。

†部分经法国国家研究署（ANR）的 ANR-20-CE48-0001 项目资助。

对字符串中的平方的研究可追溯到 Thue 的工作，他在 1906 [73], 发表的论文中探讨了如何构造一个不含平方的无限词。显然，任何足够长的二元词都必定包含平方，而 Thue 证明了存在一个不含平方的无限三元词。他的结果后来多次被重新发现。1979年，Bean、Ehrenfeucht 和 McNulty [9]开始系统研究所谓的可避免重复，相关内容可参见 Currie 的综述 [27]。

词组合学。词组合学领域的基本工具是所谓的周期性引理。字符串 $T[1..n]$ 的周期是满足以下条件的整数 d：对每个 $i \in [1, n-d]$，都有 $T[i] = T[i+d]$；周期性引理指出，如果 p 和 q 都是这样的周期，且 $p+q \leq n+$ gcd(p,q)，那么 gcd(p,q) 也是 [32] 的一个周期。这一结论以无数种方式得到了推广，适用于字符串 [17,49,74],、部分词（含有无关字符的词） [11–13,47,50,69,70],、阿贝尔周期[14,20], 、参数化周期 [6], 、保序周期 [42,64], 、近似周期 [2–4]。现在，平方可以定义为长度是其周期两倍的片段。字符串 a^n 含有 $\Omega(n^2)$ 个这样的片段，因此从组合学的角度来看，只统计不同平方是自然的。Fraenkel 和 Simpson [35] 给出了不同平方最大数量的上界 $2n$ 和下界 $n - \Theta(\sqrt{n})$，其中字符串长度为 n。经过一系列改进 [28,48,72], ，上界最近被改进为 n [16]。这一最新结果已经推广到高次幂串 [59]。避免 a^n 这类平凡示例的另一种方法是只统计极大周期片段，即周期不超过其长度一半，且无法在不破坏周期的情况下向左或向右延伸的片段。这类片段通常称为 run。Kolpakov 和 Kucherov [52] 给出了其数量的上界 $\mathcal{O}(n)$，由此开启了确定精确常数 [24,25,40,41,67,68], 的一系列研究，最终 Bannai 等人 [8] 的论文给出了上界 n；随后，针对二元字符串又得到了更好的上界 [33,45]。与之相辅相成的是一系列下界 [36,62,63,71]。

字符串算法。 本文关注字符串中重复的检测算法。在这一方向上，最基本的问题是检查给定的长度为n的字符串是否至少包含一个平方，而最一般的问题则是计算所有 runs。Main 和 Lorentz [61], 最早研究了平方自由性测试，他们设计了一种基于分治方法的 $\mathcal{O}(n\log\ n)$时间算法，并提出了一个线性时间过程，用于找出拼接两个字符串时产生的所有新平方。事实上，他们的算法可以在相同的时间复杂度内，找出给定字符串中的所有平方（的紧凑表示）。他们还证明，在最坏情况下，任何基于字符比较的算法都需要进行 $\Omega(n\log\ n)$ 次此类操作，才能测试平方自由性。这里，字符比较是指检查输入字符串中两个位置上的字符是否相同。然而，为了得到这个下界，他们不得不考虑包含多达 n 个不同字符的实例，也就是说，字母表的大小为 n。这有些令人不满意，因此引出了 Main 和 Lorentz [61]明确提出的以下开放问题：

问题 1.1. 如果限制字母表的大小，是否有更快的算法来判断一个字符串是否为无平方子串？

Crochemore 还提出了另一种 $\mathcal{O}(n\log\ n)$ 时间算法，用于找出所有重复；他还证明，对于常数大小字母表，可以在 $\mathcal{O}(n)$ 时间内完成平方自由性测试 [23]。事实上，对于字符具有线性序、大小为 σ 的字母表，后一种算法的运行时间为 $\mathcal{O}(n\log\ \sigma)$。也就是说，它需要测试某个位置上的字符是否

小于另一位置上的字符。在本文余下部分，我们将这种模型称为一般有序字母表；而只能测试字符是否相等的模型则称为一般（无序）字母表。后来，Kosaraju [53] 证明，事实上，假设字母表大小为常数，只需 $\mathcal{O}(n)$ 时间，就足以找出输入字符串中从每个位置开始的最短平方。Apostolico 和 Preparata [7] 提出了另一种 $\mathcal{O}(n\log n)$ 时间算法，该算法假设使用一般有序字母表，其依据更多是数据结构方面的考量，而非词的组合性质。最后，在线 [46, 54, 56] 和并行 [5] 平方检测方面的研究成果，也可以得到若干替代性的 $\mathcal{O}(n \log n)$ 和 $\mathcal{O}(n\log \sigma)$ 时间算法（分别适用于一般无序字母表和一般有序字母表）；有趣的是，在相关的流模型 [65, 66]中，无法高效地做到这一点。

针对一般有序字母表上的字符串进行无平方性测试的更快算法，是在研究如何找出所有 runs 时作为更一般结果的副产品而得到的。Kolpakov 和 Kucherov [52]不仅证明了任意长度为n的字符串至多包含 $\mathcal{O}(n)$个 runs，还展示了在字母表可线性排序的前提下，如何在相同时间内找出这些 runs。每个平方都包含在某个 run 中，而每个 run 至少包含一个平方，因此这尤其意味着，对于这类字母表，存在一个线性时间的无平方性测试算法。对于一般有序字母表，Kosolobov [55] 证明了这个问题的决策树复杂度仅为 $\mathcal{O}(n)$，随后又给出了一个高效的$\mathcal{O}(n(\log n)^{2/3})$时间算法 [57] （仍只使用 $\mathcal{O}(n)$次比较）。之后，通过提供一种适用于一般有序字母表的最长公共扩展（LCE）查询通用机制 [39], ，时间复杂度被改进至 $\mathcal{O}(n\log\log n)$；再后来，通过观察到 LCE 查询具有额外结构 [26]，复杂度进一步降至 $\mathcal{O}(n\alpha(n))$。最后，Ellert 和 Fischer 给出了一个优雅的$\mathcal{O}(n)$时间算法，从而彻底解决了一般有序字母表上的平方检测复杂度问题。然而，对于一般（无序）字母表，Main 和 Lorentz 提出的问题仍未解决，目前最佳上界为 $\mathcal{O}(n\log n)$，且已知只有当字母表大小为 $\Theta(n)$时，该上界才是渐近紧确的。

一般字母表。 尽管在许多应用中，可以不失一般性地假设字母表中的字符具有某种次序，但定义什么是平方并不需要这样的次序。因此，从数学角度来看，自然会希望找到不依赖这种次序、又能高效检测无平方性的算法。类似的考量也带来了关于模式匹配问题的多个优美结果，例如常数空间算法 [15, 38], ，以及关于所需相等性比较确切次数的研究[18, 19]。较新的例子包括 Duval、Lecroq 和 Lefebvre 关于计算无边框共轭词／循环移位的研究 [29] ，以及 Kosolobov 关于寻找最左临界点的研究 [58]。

主要结果。 我们研究判定给定字符串 $T[1..n]$ （其中包含 σ个不同字符）是否无平方性的复杂度。只能通过进行比较$T[i] \stackrel{?}{=} T[j]$来访问输入字符串，且不假设已知 σ 的值。我们首先分析该问题的决策树复杂度。也就是说，我们只考虑所需且必要的比较次数，而不考虑如何高效实现。我们证明，即使假设已知 σ 的值，也至少需要 $\Omega(n\log \sigma)$ 次比较。

定理 1.1。 对于满足 $8 \le \sigma \le n$ 的任意整数 n 和 σ，不存在一种确定性算法，能够在最坏情况下执行至多 $n\ln\sigma - 3.6n = \mathcal{O}(n\ln\sigma)$ 次比较，并判断一个长度为 n 、由一般无序字母表中至多 σ 个不同符号构成的字符串是否为无平方子串。

接下来，我们证明只需进行 $\mathcal{O}(n\log \sigma)$ 次比较。需要强调的是，我们并不假设已知 σ 的值。事实上，作为上述定理的预备，我们先证明，寻找

该值的次线性乘法近似需要 $\Omega(n\sigma)$ 次比较。这与所声称的上界并不矛盾，因为我们只是在说，针对某个特定输入字符串，所用比较次数至多为 $\mathcal{O}(n\log\sigma)$，但实际次数可能更少。因此，无法根据已用的比较次数，对 σ 的值作出任何有意义的近似。

定理 1.2. 对于一个长度为n 、包含来自一般无序字母表的 σ 个不同符号的字符串，判断其是否无平方子串，只需进行 $\mathcal{O}(n\log\sigma)$ 次比较。

上述结果的证明并不高效，因为它只限制了比较的总次数，却没有限制实际确定应进行哪些比较所需的时间。直接实现会得到一个平方时间算法。我们首先展示如何将其改进为 $\mathcal{O}(n\log\sigma + n\log^* n)$时间（同时仍保持渐近最优的 $\mathcal{O}(n\log\sigma)$次比较），最后进一步改进为 $\mathcal{O}(n\log\sigma)$。在论文的这一部分，我们假设采用字长为$\Omega(\log n)$的 Word RAM 模型。需要强调的是，我们仍假设输入字符串由只能进行相等性测试的字符组成；也就是说，可以认为我们获得了一个函数的预言机访问，该函数接收 i 和 j，并检查 $T[i] = T[j]$是否成立。

定理 1.3. 对于一个长度为n 、包含来自一般无序字母表的 σ 个不同符号的字符串，其无平方性测试可通过 $\mathcal{O}(n\log\sigma)$ 次比较和时间实现。

最后，我们还将这一结果推广到了 runs 的计算。

定理 1.4. 对于一个长度为n 、包含来自一般无序字母表的 σ 个不同符号的字符串，计算所有 *runs* 可通过 $\mathcal{O}(n\log\sigma)$ 次比较和时间实现。

总而言之，我们的结果彻底解决了 Main 和 Lorentz 关于一般无序字母表和确定性算法情形的开放问题。我们将下界推广到随机化算法的情形留作一个开放问题。

方法概述。 如前所述，Main 和 Lorentz [61] 设计了一种 $\mathcal{O}(n\log n)$时间算法，用于测试一般字母表上的长度为n 的字符串的无平方性。他们算法的核心思路如下。首先，他们设计了一个过程，用于检查给定的两个字符串 x 和 y的拼接是否包含一个平方，且该平方并未完全包含在 x 中，也未完全包含在y 中，时间为 $\mathcal{O}(|x| + |y|)$。然后，可以使用分治方法，在总计 $\mathcal{O}(n\log n)$时间内检测整个输入字符串中是否存在平方。对于大小无界的一般字母表，这一复杂度无法改进；但 Crochemore [23] 证明，对于大小为 σ的一般有序字母表，存在一种更快的 $\mathcal{O}(n\log\sigma)$时间算法。他的方法要点是，首先求出输入字符串的所谓f-分解（与著名的 Lempel-Ziv 分解有关），从某种意义上说，这种分解能够“发现”重复片段。随后，可以利用该分解，在输入字符串中恰当选取的片段上应用 Main 和 Lorentz 的过程，从而检测到每个不同平方的最左出现位置；应用该过程的字符串总长度仅为 $\mathcal{O}(n)$。对于大小为 σ 的一般有序字母表，通过大致构造某种后缀结构（后缀数组、后缀树或后缀自动机），即可在 $\mathcal{O}(n\log\sigma)$时间内求得该分解。

对于一般（无序）字母表，计算 f 因子分解（或类似的分解）似乎很棘手。事实上，我们证明（这是我们关于近似字母表大小的下界的一个推论），对给定的长度为 n 、包含 σ 个不同字符的字符串，计算 f 因子分解或 Lempel-Ziv 因子分解（LZ 因子分解）需要进行 $\Omega(n\sigma)$ 次相等性测试。因此，我们需要另辟蹊径。此外，Ellert 和 Fischer 的 $\mathcal{O}(n)$ 时间算法 [30] 依赖于 Lyndon 词的概念，而该概念对一般字母表上的字符串并无定义。因此，乍一看

似乎 $\Theta(n\sigma)$ 是测试长度为 n、取自大小为 σ 的一般字母表的字符串是否无平方子串所需的时间复杂度。然而，鉴于 Main 和 Lorentz 给出的下界 $\Omega(n\log n)$（用于测试由最多 n 个不同字符组成的长度为 n 的字符串是否无平方子串），当字符串中只有 σ 个不同字符时，人们或许会期待存在一个 $\mathcal{O}(n\log \sigma)$ 时间算法。

我们首先证明这类字符串的下界为 $\Theta(n\log \sigma)$。直观地说，我们证明平方自由性测试具有直和性质：可以将针对长度为 σ 的字符串的 $\frac{n}{\sigma}$ 个实例合并为针对单个长度为 n 的字符串的实例。与 Main 和 Lorentz 的证明一样，我们采用对抗方法。尽管底层计算本质上相同，但我们需要以恰当的方式合并较小的实例，这通过无限无平方的 Prouhet-Thue-Morse 序列来实现；此外，我们还需要使用复杂得多的规则来处理后续的相等性测试。作为对抗方法的热身，我们证明，计算不同字符数量的任何有意义的近似值都需要进行 $\Omega(n\sigma)$ 次此类测试；这也意味着，计算 f 分解和 Lempel-Ziv 分解（如果事先不知道字母表的大小）同样存在这一数量级的下界。

接下来，我们着手设计一种方法，使用 $\mathcal{O}(n\log \sigma)$ 次相等性比较来测试无平方性。如前所述，检测平方的一种方法是使用字符串的 f 分解，这与其 LZ 分解类似。然而，正如推论1 和2所证明的，我们无法在一般无序字母表上通过 $o(n\sigma)$ 次比较计算这两种分解。因此，我们将改用一种新型分解，即 Δ 近似 LZ 分解，可以将其视为 LZ 分解的近似版本。直观而言，它旨在“捕获”所有足够长的平方，而原始的 LZ 分解（或 f 分解）则能捕获所有平方。每个 Δ 近似 LZ 分解中的短语都由一个长度至多为 Δ的头部，以及一个尾部（可以为空）组成；该尾部必须至少在此前出现过一次，并且整个短语的长度至少与从同一位置开始的经典 LZ 短语一样长。与经典 LZ 分解不同，这种分解并不唯一。采用这种改进方式的好处是，短语数量更少（而且选择短语的灵活性更高），因此有望更高效地计算这种分解。

为了设计一种高效的 Δ 近似 LZ 分解构造方法，我们首先说明如何在尽量减少符号比较次数的同时计算稀疏后缀树。随后，将此方法应用于一个由所谓差分覆盖选出的位置集合，该覆盖具有一些便于使用的同步性质。然后，利用 Δ 近似 LZ 分解检测长度为 $\geq 8\Delta$ 的平方。

第一个预热算法在（假设已知 σ 的情况下）根据 n 和 σ 确定 Δ，并利用近似 LZ 分解找出所有长度至少为 8Δ 的平方。然后，它将字符串划分为长度为 8Δ 的块，并在每个块对上应用 Main 和 Lorentz 的原始算法，以找出所有更短的平方。我们对 Δ 的选择带来 $\mathcal{O}(n(\lg \sigma + \lg\lg n))$ 次比较。

改进算法无需预先知道 σ，而是从一个较大的 $\Delta = \Omega(n)$ 开始，然后逐步减小 Δ，最多经过 $\mathcal{O}(\lg\lg n)$ 个阶段，后续阶段会检测更短的平方。一旦发现字母表中有许多不同字符，我们就可以通过仔细调整参数，采用 Main 和 Lorentz 的方法处理输入字符串中足够短的片段。由于每个阶段都无法承担 $\Omega(n)$ 次比较的开销，我们采用停用技术：每当某个阶段执行了大量比较，就会在之后的所有阶段舍弃字符串的一大部分。具体来说，在某个阶段中，我们不再查找完全包含在先前阶段尾部中的片段里的平方。这样便能以最优的 $\mathcal{O}(n\lg \sigma)$ 次比较完成计算。

上述方法在最坏情况下使用渐近最优数量的相等性测试，但并未得到高效的算法。主要瓶颈在于构造稀疏后缀树。不过，利用一般字母表上的字符串的最长公共扩展查询通用机制，实现一种高效算法并不困难 [39]。遗憾的是，目前已知的最佳

查询此类问题的方法会使时间复杂度额外增加 $\mathcal{O}(n\log^* n)$，即使字母表的大小是常数也是如此。我们通过谨慎地停用文本片段来计入已完成的工作，从而克服了这一技术难题。

我们的许多技术都可以轻松修改，用于计算所有 run，而不是检测平方。我们利用近似分解能够揭示具有较早出现位置的长子串这一特点。因此，我们只计算这类子串首次出现位置对应的 run，而对于之后的出现位置，则直接复制已计算出的 run。通过仔细安排计算顺序，我们确保复制所需的总时间不超过 run 的数量，而已知该数量为$\mathcal{O}(n)$。这样，我们便能在 $\mathcal{O}(n\lg\sigma)$时间内完成计算所有 run，并进行相应的比较。

2 预备知识

字符串。 长度为 n的字符串是由大小为 σ的有限字母表 Σ中的字符组成的序列 $T[1]\dots T[n]$。子串 $T[i..j]$是字符串 $T[i]\cdots T[j]$，而片段 $T[i..j]$指的是从 T中位置 i 开始的 $T[i..j]$ 的特定出现位置。若 $i>j$，则 $T[i..j]$ 为空字符串。 T 的后缀形式为 $T[i..n]$。如果 $i<i'\le j'<j$，我们就说片段 $T[i'..j']$ 被另一个片段 $T[i..j]$ 真包含。若一个子串等于某个被 $T[i..j]$真包含的片段，则该子串被 $T[i..j]$真包含。我们用 $T[i..j)$作为 $T[i..j-1]$的简写。同样，我们用 $[i,j]=[i,j+1)$作为整数区间 $\{i,\dots,j\}$的简写。给定两个位置$i\le j$，它们的最长公共扩展（LCE）是后缀 $T[i..n]$和 $T[j..n]$之间最长公共前缀的长度，形式化定义为 $\mathrm{LCE}(i,\ j)=\mathrm{LCE}(j,\ i)=\max\{\ell\in\{0,\dots,n-j+1\}\mid T[i..i+\ell)=T[j..j+\ell)\}$.

定义 2.1. 正整数 p是字符串 $T[1..n]$的一个周期，当且仅当对每个$i\in\{1,\dots n-p\}$，都有 $T[i]=T[i+p]$。最小的此类 p称为 $T[1..n]$的周期；若字符串的周期 p 至多为 $\frac{n}{2}$，则称该字符串是周期性的。

计算模型。 对于一般无序字母表 Σ，对字符唯一允许的操作是比较相等。特别地，字母表上不存在任何线性序。除非另有明确说明，我们只会使用此类比较。一般有序字母表具有全序，因此可以进行小于等于类型的比较。

在论文的算法部分，我们假设采用标准的单位代价 Word RAM 模型，字长为 $\Omega(\log\ n)$，但算法只能通过比较 $T[i]\stackrel{?}{=}T[j]$来访问输入字符串 $T[1..n]$，且假设这些比较需要常数时间。如果长度为n 的字符串可以在 $\mathcal{O}(n)$ 时间内对其中的 n 个符号进行排序，我们就称该字符串属于可线性排序字母表。请注意，字母表是否可线性排序，不仅取决于字母表本身，也取决于字符串。例如，对于长度为$n=\Omega(m)$的字符串，字母表 $\Sigma=\{1,\dots,m^{\mathcal{O}(1)}\}$ 是可线性排序的（例如，使用基数排序），但对于所有长度为 $n=o(m)$ [44]的字符串，它是否可线性排序尚不得而知。我们的算法将在内部使用由可线性排序字母表构成的字符串。需要强调的是，这类字符串中的字符并非输入字符串中的字符，而只是算法计算得到的整数。请注意，每个可线性排序字母表也是一般有序字母表。

平方与 runs。 平方是一个长度为2ℓ 、周期为 ℓ的片段。以下定理是 Main 和 Lorentz [61] 的经典结果。

定理 2.1。 在一般字母表上测试 $T[1..n]$ 无平方子串可以在$\mathcal{O}$（n对数 n） 时间内通过比较实现。

上述定理的证明基于运行一个分治过程，该过程使用以下引理。

引理 1. 给定一般字母表上的两个字符串 x 和 y，我们可以在 $\mathcal{O}(|x|+|y|)$ 时间内通过比较，判断 xy 中是否存在一个平方，且该平方既未完全包含在 x 中，也未完全包含在 y 中。

重复是一个长度为 ℓ、周期至多为 $\frac{\ell}{2}$的片段。run 是一个极大的重复。形式化地说，$T[1..n]$ 中的重复是一个三元组 $\langle s,e,p\rangle$，其中 $s,e\in[1,n]$和 $p\in[1,\frac{e-s+1}{2}]$，且 p是 $T[s..e]$的最小周期。run 是一个重复 $\langle s,e,p\rangle$，无法在保持相同周期的情况下向左或向右延伸；换言之，$s=1$ 或 $T[s-1]\neq T[s-1+p]$，且 $e=n$ 或$T[e+1]\neq T[e+1-p]$。著名的 runs猜想由 Bannai 等人 [8],证明，该猜想指出，任意长度为 n 的字符串中的 runs 数量小于 n。Ellert 和 Fischer [30] 证明，一般有序字母表上的字符串中的所有 runs 均可在 $\mathcal{O}(n)$ 时间内计算得出。如前所述，每个 run 都包含一个平方，而每个平方都包含在某个 run 中。因此，一个字符串包含平方，当且仅当它包含 run，由此可得：

定理 2.2. 在一般有序字母表上计算 $T[1..n]$ 的所有 *runs*（从而测试其无平方性），可以在 $\mathcal{O}(n)$ 时间内完成。

Lempel-Ziv 分解。 从 $T[1..n]$ 中的位置 s 开始的唯一 LZ 短语，是一个片段 $T[s..e]$，满足 $T[s..(e-1)]$ 在 $T[1..(e-1)]$ 中至少出现两次，并且 $e=n$ 或 $T[s..e]$ 在 $T[1..e]$ 中只出现一次。T 的 Lempel-Ziv 分解由 z 个短语 $f_1,\ldots,f_z$ 组成，满足拼接 $f_1\ldots f_z$ 等于 $T[1..n]$，且每个 f_i 都是从位置 $1+\sum_{j=1}^{i-1}|f_j|$ 开始的唯一 LZ 短语。

字典树。 给定由某个字母表 Σ 上的字符串组成的集合 $\mathcal{S}=\{T_1,\ldots,T_k\}$，其字典树是一棵有根树，边标签取自 Σ。对于任意节点 v，从根节点到该节点的边标签拼接起来表示 一个字符串。节点的字符串深度是其表示的字符串的长度。任意两个节点表示的字符串都不相同；也就是说，对于任意节点，其各个子节点所对应边的标签两两不同。每个叶节点表示其中一个 T_i，且每个 T_i 都由内部节点或叶节点表示。

$\mathcal{S}$ 的压缩字典树可以通过收缩其（未压缩）字典树中叶节点或分支节点与其最近的分支祖先之间的每条路径，得到一条单独的边（也就是说，通过收缩消除所有非分支内部节点）。新边的标签是从根到叶方向上，被收缩路径中各边标签的拼接。由于叶节点至多有 k 个，且所有内部节点都是分支节点，因此压缩字典树中有 $O(k)$ 个节点。每个边标签都是字符串集合中的某个子串 $T_i[s..e]$，我们可以不显式存储该标签，而改为存储引用（i，s，e）。因此，存储压缩字典树只需 $O(k)$ 个字。考虑一个由未压缩字典树中的某个节点拼写出的字符串 T'。当且仅当该字符串由压缩字典树中的某个节点拼写出时，我们称 T' 为显式的。否则，T' 为隐式的。

字符串 $T[1..n]$ 的后缀树是恰好包含其所有后缀的压缩字典树，即针对字符串集合 $\{T[i..n]\mid i\in\{1,\ldots,n\}\}$ 构建的字典树。它是字符串算法领域最基础的数据结构之一，广泛应用于压缩和索引等场景 [43]。后缀树可用 $\mathcal{O}(n)$ 个字存储，占用内存；对于可线性排序字母表，可在 $\mathcal{O}(n)$ 时间内构建 [31]。对于某个起始位置集合 $B\subseteq\{1,\ldots,n\}$，T 的稀疏后缀树是恰好包含后缀 $\{T[i..n]\mid i\in B\}$ 的压缩字典树。它可用$\mathcal{O}(|B|)$ 个字存储，占用内存。

我们假设 T 以某个特殊符号 $T[n] = \$$ 结尾，且该符号在T中其他位置均未出现。这样可以确保每个后缀都由一个叶节点表示，并用相应后缀的起始位置为叶节点标记。注意，对于任意两个叶节点 $i \neq j$，它们的最近公共祖先（即同时为 i 和 j祖先的最深节点）表示一个长度为 $\text{LCE}(i, j)$ 的字符串。

3 下界

本节中，我们给出计算字母表大小的有意义近似值所需符号比较次数的下界（第 3.1 节），以及测试无平方性所需符号比较次数的下界（第3.2 节）。对于这两个下界，我们都采用对抗方法，下面先简要介绍。

当前的计算模型可以这样理解：处理一般无序字母表上的字符串的算法无法访问字符串本身，只能向预言机询问两个位置上的符号是否相同。所提问的次数恰好就是执行比较的次数。为了证明解决某个问题所需比较次数的下界，我们描述一个接替预言机角色的对手，迫使算法尽可能多地进行符号比较。

我们使用冲突图 $G = (V, E)$，其具有 $V = \{1, \ldots, n\}$ 和 $E \subseteq V^2$，用于记录对手给出的回答。图中的节点与字符串的位置一一对应。初始时，我们有 $E = \emptyset$，且所有节点均无颜色，这在形式上意味着它们的颜色为 $\gamma(i) = \perp$。在算法执行过程中，对手可以从集合 $\Sigma = \{0, \ldots, n-1\}$ 中为节点指定颜色，这可以看作是永久确定相应位置上的字母表符号（即每个节点至多着色一次）。用于给节点着色的规则取决于我们要证明的下界（相关细节将在相应章节中介绍）。除着色规则外，对手的一般行为如下：每当算法询问 $T[i] = T[j]$ 是否成立时，当且仅当 $\gamma(i) = \gamma(j) \neq \perp$，对手才回答“是”。否则，它回答“否”，并在 E中插入一条边 (i, j)。每当对手为某个节点指定颜色时，必须选择一个与冲突图中所有相邻节点所用颜色都不同的颜色。这样可以确保着色不会与先前给出的回答矛盾。

定义一个与对手给出的答案一致的字符串集合 $\mathcal{T} \subseteq \Sigma^n$。如果满足以下条件，字符串 $T \in \Sigma^n$ 就是 $\mathcal{T}$ 的成员：

$$\forall i \in V : \gamma(i) \in \{\perp, T[i]\} \quad \wedge \quad \forall i, j \in V : (T[i] = T[j]) \implies (i, j) \notin E.$$

请注意，$\mathcal{T}$会随时间变化。最初（算法开始之前），我们有 $\mathcal{T} = \Sigma^n$。每提出一个问题，算法都可能从 $\mathcal{T}$ 中排除一些字符串。不过，$\mathcal{T}$ 中始终至少有一个字符串：只需为每个未着色节点指定一种此前完全未用过的颜色，就能得到这样的字符串。

3.1 近似字母表大小

给定一个字母表大小 $\sigma \geq 2$ 未知的字符串 $T[1..n]$，假设我们要计算 σ 的近似值。我们证明，如果某个算法在最坏情况下至多进行 $\frac{n\sigma}{8}$ 次比较，那么它就无法区分至多含有 σ 个不同符号的字符串与至少含有 $\frac{n}{2}$ 个不同符号的字符串。因此，对 σ 进行任何有意义的近似都需要 $\Omega(n\sigma)$ 次比较。

为证明这一点，考虑这样一种算法：对于长度为n 且至多包含 $\sigma \geq 2$ 种不同符号的字符串，该算法最多执行 $\frac{n\sigma}{8}$ 次比较。我们采用第3节开头所述的对手，并确保集合 $\mathcal{T}$中与对手的回答相符的字符串始终包含一个至多有 σ 种不同符号的字符串。因此，算法会终止

且最多执行 $\frac{n\sigma}{8}$ 次比较。与此同时，我们确保 $\mathcal{T}$ 也包含一个至少有 $\frac{n}{2}$ 种不同符号的字符串，由此得到所需结果。对手采用以下着色规则。所有颜色均取自 $\{1,\ldots,\sigma\}$。每当冲突图中某个节点的度变为 $\sigma-1$ 时，就为其指定颜色。我们避开冲突图中与之相邻节点的颜色。在任意时刻，我们都可以假设通过为每个未着色节点指定 $\{1,\ldots,\sigma\}$ 中的一种颜色，并避开相邻节点的颜色，来完成着色。如此一来，每个节点都会被指定 σ 种颜色之一，这意味着 $\mathcal{T}$ 包含一个至多有σ 种不同符号的字符串。因此，该算法最多经过 $\frac{n\sigma}{8}$ 次比较后终止。每次比较最多会使两个节点的度各增加一。因此，经过 $\frac{n\sigma}{8}$ 次比较后，度至少为 $\sigma-1$ 的节点至多有 $\frac{n\sigma}{8}\cdot\frac{2}{\sigma-1}\leq\frac{n}{2}$个。所以，至少有 $\frac{n}{2}$ 个节点未着色。我们可以假设用 $\frac{n}{2}$ 种不同颜色为它们着色，这意味着 $\mathcal{T}$ 包含一个至少有 $\frac{n}{2}$ 种不同符号的字符串。由此得到以下结果。

定理 3.1。 对于满足 $2\leq\sigma<\frac{n}{2}$ 的任意整数 n 和 σ，不存在一种确定性算法，能在最坏情况下至多执行 $\frac{n\sigma}{8}$ 次相等性比较，并区分长度为 n 且至多包含 σ 个不同符号的字符串，与长度为 n 且至少包含 $\frac{n}{2}$ 个不同符号的字符串。

该定理给出了计算 LZ 分解（定义见第 2 节）和 f-分解所需比较次数的下界。在唯一的 f-分解 $T=f_1f_2\ldots f_z$中，每个因子 f_i要么是一个在 $f_1\ldots f_{i-1}$中未出现过的单独符号，要么是满足以下条件的最长片段： f_i在 $f_1\ldots f_i$中出现两次。

推论 1。 对于满足 $2\leq\sigma<\frac{n}{4}$ 的任意整数 n 和 σ，不存在一种确定性算法，能在最坏情况下至多执行 $\frac{(n-1)}{16}\sigma$ 次相等性比较，并计算一个长度为 n 、至多包含 σ 个不同符号的字符串的 f-分解。

证明。 对于一个含有 σ 个不同符号的字符串 $T=T[1]T[2]\ldots T[\frac{n}{2}]$，考虑将 T 的每个字符重复一次后构造出的、含有 σ 个不同符号的长度为 n 的字符串 $T'=T[1]T[1]T[2]T[2]\ldots T[\frac{n}{2}]T[\frac{n}{2}]$。 T 的字母表大小恰好等于从 T' 的奇数位置开始的 T' 的 f- 分解中长度为 1 的短语数。因此，根据定理 3.1，我们需要进行 $\frac{n\sigma}{16}=\frac{|T|\sigma}{8}$ 次比较，才能找到 T' 的 f-分解。我们假设 n 为偶数；对于奇数的 n，则通过将界调整为 $\frac{(n-1)}{16}\sigma$ 来处理。 □

推论 2。 对于任意满足 $3\leq\sigma<\frac{n}{6}+1$ 的整数 n 和 σ，不存在这样的确定性算法：它在最坏情况下至多进行 $(n-2)(\sigma-1)\ 24$ 次相等性比较，并计算长度为 n 、含有至多 σ 个不同符号的字符串的 *Lempel-Ziv* 分解。

证明。 对于某个含有 $\sigma-1$ 个不同符号的字符串 $T=T[1]T[2]\ldots T[\frac{n}{3}]$，令 T' 为一个长度为 n、含有 σ 个不同符号的字符串：通过在 T 的每个字符之间插入分隔符，将每个字符加倍构造而成，即 $T'=T[1]T[1]\#T[2]T[2]\#\ldots\ldots\#T[\frac{n}{3}]T[\frac{n}{3}]\#$。字符 $\bar{x}$ 在 T 中的首次出现，对应于 $xx\#$ 在 T' 中的首次出现，因此前一个短语（长度可能为一）在 $xx\#$ 首次出现中的第一个 x 处结束，而后一个短语必定是 $x\#$。于是，对于 $xx\#$ 后续的出现位置，不可能有 $x\#$ 是一个短语。因此， T 的字母表大小恰好等于 T' 中从位置 $i\equiv2$（模 3）开始的 T' 的 Lempel-Ziv 分解中长度为二的短语数量。由定理 3.1，我们需要 $^{n}(\sigma-1\ =\ |T|)\,(\sigma-1)\,8$ 次比较，才能找到T'的 LZ 分解。我们假设 n 可被 3 整除，并通过将界调整为$\frac{24}{(n-2)(\sigma-1)\ 24}$ 来考虑这一点。 □

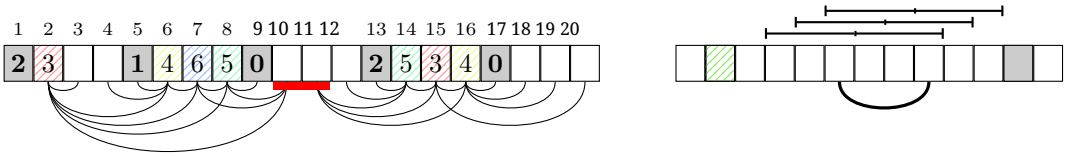


图1：第 3.2 节所述对手的冲突图示例。字母表的大小为 $\sigma = 16$。块的长度为 $\frac{\sigma}{4} = 4$。灰色节点恰好是块的起始位置，包含三元 Thue-Morse 序列 $v = 2$，1, 0, 2, 0, 1, 2, . . . 中的符号，该序列无平方子串。我们假设着色节点按以下顺序着色：2, 6, 8, 7, 15, 16, 14。在给节点 8 着色时，我们必须避开颜色 0、1、2（因为这些颜色保留给分隔符位置）、3（因为相邻节点 2 已经使用颜色 3）和 4（因为节点 6 与其位于同一块且已经使用颜色 4）。算法尚未消除所有平方。例如，节点 10 和 11 之间缺少边 (10, 11)，它们与颜色为 $\{3, 6, 5\} \cup \{5, 3, 4\}$ 的节点相邻。因此，可以为这两个节点都分配颜色 $\{0, 1, 2\} \cup \{7, \ldots, 15\}$ 中的任意一种，从而形成平方 $T[10..11]$。如右图所示，长度为 ℓ 的边最多消除 ℓ 个平方。

3.2 平方自由性测试

本节中，我们证明平方自由性测试至少需要 $n \ln \sigma - 3.6n$ 次比较（即使已知 σ）。该证明结合了 Main 和 Lorentz 最初提出的 Ω(*n*lg *n*) 下界背后的思路 [61] 以及第 3 节开头所述的对手策略。这一次，我们确保 $\mathcal{T}$ 始终包含一个无平方子串，且其中不同符号不超过 σ 个。同时，我们还力求确保 $\mathcal{T}$ 也包含至少有一个平方的字符串。我们将证明，这种状态至少可以维持到完成 $n \ln \sigma - 3n$ 次比较为止。

对手构造的字符串（或者更确切地说，字符串族）被组织成长度为 $\frac{\sigma}{4}$ 的 $\lceil \frac{4n}{\sigma} \rceil$ 个互不重叠的块（我们假设 $\frac{\sigma}{4} \in \mathbb{N}$ 和 $8 \le \sigma \le n$）。每个块都以一个特殊分隔符开头。更准确地说，第 k 个块的第一个符号是字母表 $\{0, 1, 2\}$ 上某个三元无平方词的第 k 个符号（例如，在普鲁厄 - 图埃-莫尔斯序列中，0 的第 k^{th} 次和第 $(k + 1)^{\text{th}}$ 次出现位置之间的距离；该序列也称为三元 Thue-莫尔斯序列，见 [1, 推论 1]）。最初，对手按照 $\{0, 1, 2\}$ 中各自的颜色，为对应于分隔符位置的节点着色。之后，所有其余节点将被赋予不同于 $\{0, 1, 2\}$ 的颜色。任何跨越块边界的片段都可以投影到颜色$\{0, 1, 2\}$上；根据构造，该字符串不可能包含平方。因此，分隔符确保没有平方跨越块边界，这意味着该字符串无平方子串，当且仅当它的每个块都无平方子串。

在算法执行过程中，我们使用以下着色规则。可用颜色为$\{3, \ldots, \sigma - 1\}$。每当节点的度变为$\frac{\sigma}{4}$时，我们就为其着色。我们不仅避开冲突图中已着色相邻节点所用的至多 $\frac{\sigma}{4}$ 种颜色，也避开同一块内节点所用的少于 $\frac{\sigma}{4}$ 种颜色（由于 $\sigma \ge 8$，因此至少有 $\sigma - 3 - \frac{\sigma}{2} \ge 1$ 种颜色可用）。图1展示了冲突图的一个示例。在任意时刻，我们都可以设想通过为每个未着色节点指定一种颜色 $\{3, \ldots, \sigma - 1\}$ 来完成着色，同时避开相邻节点的颜色以及同一块内节点的颜色。之后，每个节点都具有 σ 种颜色之一，但同一块内任意两个节点的颜色都不同。因此，每一块都是无平方子串的，所以 $\mathcal{T}$ 总是包含一个至多含有σ 种不同符号的无平方子串字符串。

现在我们考虑<排版 id='1'>算法终止后</排版>冲突图的状态。我们尤其关注无色节点的连续区间。以下引理指出，对于每个这样的区间，算法要么执行了大量比较，要么我们可以在该区间内强制得到一个平方

在范围内。

引理 2. 设 $R = \{i, \ldots, j\} \subset V$ 是冲突图中 $m = j - i + 1$ 个无色节点组成的连续区间。那么，要么 $\left|E \cap R^2\right| \geq \sum_{\ell=1}^{\lfloor m/2 \rfloor} m-2\ell+1 \choose \ell$，要么存在一个字符串 ~~$T \in \mathcal{T}$~~，其不同符号至多有 σ 个，且 $T[i..j]$ 含有一个平方。

证明。如果某个整数区间 $[x, x + 2\ell - 1]$ 满足 $i \leq x <$ （$x + 2\ell - 1$） $\leq j$，并且存在一个 y 满足 $x \leq y < x + \ell$，使得冲突图中有一条边 $(y, y + \ell)$，我们就称该区间已被消除。若存在这样的边，则根据 $\mathcal{T}$ 的定义，所有字符串 $T \in \mathcal{T}$ 都满足 $T[y] \neq T[y + \ell]$。因此，对其中任何字符串而言，$T[x..x + 2\ell - 1]$ 都不是平方。

现在我们证明，如果 $[x, x + 2\ell - 1]$ 尚未被消除，那么存在一个字符串 $T \in \mathcal{T}$，使得 $T[x..x + 2\ell - 1]$ 是一个平方。为此，考虑任意满足 $x \leq y < x + \ell$ 的位置 y，也就是潜在平方前半部分中的一个位置。由于 $[x, x + 2\ell - 1]$ 尚未被消除，$(y,\ y + \ell)$ 不是冲突图中的一条边。因此，我们可以给 y和 $y + \ell$赋予相同的颜色。我们只需避开冲突图中与 y 和$y + \ell$ 相邻的节点所用的颜色，至多有 $2 \cdot (\frac{\sigma}{4} - 1)$ 种。由此，存在 $\frac{\sigma}{2} + 2$ 种合适的颜色，可以赋给这两个节点。与算法执行期间不同，我们不必避开特殊分隔符颜色或同一块中的颜色；由于我们正试图构造一个平方，因此不必担心意外产生平方。对所有可能的 y 选择应用这一着色方案后，我们便能确保所有字符串 $T \in \mathcal{T}$都含有一个平方 $T[x..x + 2\ell - 1]$。注意，在算法终止后给额外节点着色，只会从 $\mathcal{T}$中移除元素。因此，含有平方 $T[x..x + 2\ell - 1]$ 的字符串在算法终止时就已经位于 $\mathcal{T}$中。由此可知，如果该算法确实保证无平方性，那么它必定消除了所有满足 $i \leq x <$ $(x + 2\ell - 1)$ $\leq j$ 的可能区间 $[x, x + 2\ell - 1]$。

虽然每个区间至少需要消除一条边，但一条边可以消除多个区间。不过，一条边所消除的所有区间长度都必须相同。现在，我们给出消除所有长度为 2ℓ 的区间所需边数的下界。任意一条边 $(y, y + \ell)$ 都会消除 ℓ 个区间，即满足 $x \leq y < x + \ell$ 的区间 $[x, x + 2\ell - 1]$。在 R 内，我们必须消除长度为 2ℓ 的 $m - 2\ell + 1$ 个区间，即满足 $i \leq x \leq j - 2\ell + 1$ 的区间 $[x, x + 2\ell - 1]$（见图 1 右侧）。因此，要消除所有长度为 2ℓ 的平方，至少需要 $m-2\ell+1 \choose \ell$ 条边。最后，对 ~~ℓ 的~~所有可能取值求和，我们至少需要 $\sum_{\ell=1}^{\lfloor m/2 \rfloor} m-2\ell+1 \choose \ell$ 条边来消除 R 中的所有区间。注意，用于消除的边的两个端点都在 R 中，因此这些边包含在 $E \cap R^2$ 中。因此，如果 $\left|E \cap R^2\right| < \sum_{\ell=1}^{\lfloor m/2 \rfloor} m-2\ell+1 \choose \ell$，那么并非所有区间都已被消除，而 $\mathcal{T}$ 中就存在一个包含平方的字符串。 □

最后，我们证明：该算法要么至少进行了 $\Omega(n \lg \sigma)$ 次比较，要么存在一个包含平方的字符串 $T \in \mathcal{T}$。设 $c_1, c_2, \ldots, c_k$ 恰好是这些着色节点。最初（即算法执行之前），对手给 $\lceil \frac{4n}{\sigma} \rceil$ 个节点着色。因此 $k \geq \lceil \frac{4n}{\sigma} \rceil$，并且有 $k - \lceil \frac{4n}{\sigma} \rceil$ 个节点是在度达到 $\frac{\sigma}{4}$ 后被着色的。于是，所有着色节点的度数之和至少为 $(k - \lceil \frac{4n}{\sigma} \rceil) \cdot \frac{\sigma}{4} \geq \frac{\sigma k - 4n - \sigma}{4} \geq \frac{\sigma k - 5n}{4}$。每次比较至多使两个节点的度各增加 1。因此，着色节点对应至少 $\frac{\sigma k - 5n}{8}$ 次比较。节点中有 k 个互不重叠的极大无色区间，即辅助值为 $c_{k+1} = n + 1$ 的 $\{c_i + 1, \ldots, c_{i+1} - 1\}$（其中 $1 \leq i \leq k$）。根据引理 2，每个相应区间对应 $e_i = \sum_{\ell=1}^{\lfloor m_i/2 \rfloor} m_i-2\ell+1 \choose \ell$ 条边，其中 $m_i = c_{i+1} - c_i - 1$。（由于这些区间互不重叠，且相应边的两个端点都在区间内，因此每条边不会被重复计算。）因此，为了验证无平方性，该算法必须至少进行了 $\sum_{i=1}^{k} e_i + \frac{\sigma k - 5n}{8}$ 次比较。证明的其余部分是简单的代数运算。首先，我们给出 e_i 的一个方便下界（下文将对此作出说明）：

$$
\begin{aligned}
e_i = \sum_{\ell=1}^{\lfloor m_i/2 \rfloor} \frac{m_i - 2\ell + 1}{\ell} = \sum_{\ell=1}^{\lceil m_i/2 \rceil} \frac{m_i - 2\ell + 1}{\ell} &\geq (m_i+1)\left(\left(\sum_{\ell=1}^{\lceil m_i/2 \rceil} \frac{1}{\ell}\right) - 1\right) \\
&> (m_i+1)\cdot(\ln\frac{m_i}{2} - \frac{1}{2}) \\
&= (m_i+1)\cdot\ln\frac{m_i}{2\sqrt{e}} \\
&\geq (m_i+1)\cdot\ln\frac{m_i+1}{2.5\sqrt{e}}
\end{aligned}
$$

我们可以将 $\lfloor m_i/2 \rfloor$ 替换为 $\lceil m_i/2 \rceil$，因为若 m_i 为奇数，额外的加项就等于零。第一个不等式使用了简单的算术运算。第二个不等式利用了调和数的经典下界 $(\ln x + \frac{1}{2}) < H_x$。最后一个不等式在 $m_i \geq 4$ 时成立。当 $m_i < 4$ 时，结果为负，因此仍然是比较次数的正确下界。我们得到：

$$
\begin{aligned}
\underbrace{\sum_{i=1}^{k}(m_i+1)\cdot\ln\frac{m_i+1}{2.5\sqrt{e}}}_{\text{comparisons within colorless ranges}} + \underbrace{\frac{\sigma k - 5n}{8}}_{\text{comparisons for colored nodes}} &\geq n\cdot\ln\frac{n}{2.5\sqrt{e}k} + \frac{\sigma k - 5n}{8} \\
&= n\cdot\ln\frac{\sigma}{2.5\sqrt{e}x} + \frac{xn - 5n}{8} \\
&= n\cdot\ln\sigma + n\cdot\left(\frac{x-5}{8} - \ln 2.5\sqrt{e}x\right) \\
&> n\cdot\ln\sigma - 3.12074n
\end{aligned}
$$

第一步由 $\sum_{i=1}^{k}(m_i+1) = n$ 以及对数和不等式得出（参见 [21, 定理 2.7.1]）。第二步通过使用 $x = \frac{\sigma k}{n}$ 替换 k。第三步使用简单的算术运算。最后一步是代入使方程最小化的 $x = 8$ 得出的。最后，我们假设 σ 能被 4 整除。我们通过将下界调整为 $n\ln(\sigma - 3) - 3.12074n$来处理这一点；对于 $\sigma \geq 8$，该值大于 $n\ln\sigma - 3.6n$。

定理 1.1. 对于满足 $8 \leq \sigma \leq n$的任意整数 n 和 σ，不存在一种确定性算法，能在最坏情况下执行至多 $n\ln\sigma - 3.6n = \mathcal{O}(n\ln\sigma)$ 次比较，并判断一个长度为 n、包含来自一般无序字母表的至多 σ 个不同符号的字符串是否为无平方子串。

本节中，我们考虑给定字符串的无平方性测试问题。我们提出一种算法，只需 $\mathcal{O}(n\lg\sigma)$ 次比较，即可判定一个字符串是否为无平方子串，这与第3.2节给出的下界相匹配。请注意，该算法目前尚未达到时间高效，因为除执行符号比较外，它还使用了 Word RAM 模型中代价高昂的其他操作。第4节将介绍该算法的时间高效实现：我们先达到 $\mathcal{O}(n\lg\sigma + n\log^* n)$ 时间，然后将其改进为 $\mathcal{O}(n\lg\sigma)$ 时间。第5节中，我们将这一结果推广到以相同的时间复杂度计算所有 runs。

3.3 稀疏后缀树与差分覆盖

引理 3. 包含任意 b 后缀 $T[i_1..n], \ldots, T[i_b..n]$ 的 $T[1..n]$ 稀疏后缀树，可通过 $\mathcal{O}(b\sigma\log b)$ 次比较构造，此外，所有引理调用共用 $\mathcal{O}(n)$ 次比较。

证明。 我们在 $T[1..n]$的位置上维护一个并查集。初始时，每个位置都属于一个单独的连通分量。在执行查询 $T[x] \stackrel{?}{=} T[y]$之前，我们检查 x和 y是否属于并查集中的同一个连通分量；如果是，则立即返回 $T[x] = T[y]$ ，而不进行任何比较。否则，我们执行该查询；如果查询结果表明 $T[x] = T[y]$ ，则合并 x 和 y所在的连通分量。因此，在所有引理调用中，答案为肯定的查询总数少于 n，接下来只需界定答案为否定的查询数。

我们将后缀 $T[i_j..n]$ 逐个插入一棵初始为空的稀疏后缀树中。要插入下一个后缀，我们从树的根节点开始下降过程，找出对应于 $T[i_j..n]$ 与任一已插入后缀之间最长公共前缀的节点 u 。接着，除非 u 已经是显式的，否则就将其显式化，并添加一条从 u 指向新叶节点的边，该叶节点对应整个 $T[i_j..n]$。我们称插入过程在 u 处终止。节点 u 只需通过 $\mathcal{O}(\sigma \log b)$次比较即可识别，且这些比较的结果均为否，方法如下。令 v 为当前节点（初始为树的根节点），令 v_1、……、v_d 为它的子节点，其中 $d \leq \sigma$。这里， v 可以是显式的，也可以是隐式的；若为隐式，则 $d = 1$。我们排列 v 的子节点，使得以 v_1 为根节点的子树中的叶节点数不少于以 v 的任何其他子节点为根节点的子树中的叶节点数。然后，将通向 v_1 的边上的字符与当前后缀中对应的字符进行比较。如果二者相等，就继续处理 v_1；否则，依次将通向 v_2、……、 v_d 的边上的字符与当前后缀中对应的字符进行比较。之后，我们要么继续处理某个 v_j、 $j \geq 2$，要么在 v 处终止。为了界定结果为否的比较次数，我们注意到，只有在 v 处终止或继续处理 v_j、 $j \geq 2$ 时，才会发生此类比较。每当我们继续处理 v_j、 $j \geq 2$ 时，以当前 v_j 为根节点的子树中的叶节点数，至少比以 v 为根节点的子树中的叶节点数少一半（因为以 v_1 为根节点的子树中的叶节点数最多）。因此，在一次插入过程中从根节点开始的整个下降过程里，这种情况最多发生 $1 + \log b$ 次。每当我们没有继续处理子树 v_1 时，可能会有多达 $d \leq \sigma$ 次结果为否的比较，因此此类比较的总次数如 [1] 所述。

□

现在我们描述稍后用于计算近似 LZ 分解的采样位置。如果集合 $S \subseteq \mathbb{N}$ 是 $\{1, \ldots, n\}$ 的 t-覆盖，则存在一个可在常数时间内计算的函数 h，使得对任意 $1 \leq i, j \leq n-t+1$，都有 $0 \leq h(i,j) < t$，且 $i+h(i,j)\ j+h(i,j) \in S$。下面的引理给出了一种 t-覆盖的构造方法，并在图 2 中进行了示意。

引理 4。 对于任意 n 和 $t \leq n$，存在一个 $\{1, \ldots, n\}$ 的 t-覆盖 $\mathbf{D}(t)$，其大小为 $\mathcal{O}(n/\sqrt{t})$。此外，其元素可以按与元素数量成正比的时间枚举。

证明。我们采用一种称为差分覆盖的著名组合构造方法，参见例如 [60]。令 $r = \lfloor \sqrt{t} \rfloor$，并定义 $\mathbf{D}(t) = \{i \in \{1, \ldots, n\} : i \bmod r = 0 \text{ 或 } i \bmod r^2 \in \{0, \ldots, r-1\}\}$。根据定义， $|\mathbf{D}(t)| \leq \lfloor n/r \rfloor + \lfloor n/r^2 \rfloor r = \mathcal{O}(n/r) = \mathcal{O}(n/\sqrt{t})$。函数 $h(i,j)$ 定义为$a + b \cdot r$，其中 $a = (r-i) \bmod r$，且 $b = (r - \lfloor (j+a)/r \rfloor) \bmod r$。注意， $i + h(i,j) \leq n$，且$j+h(i,j) \leq n$。于是， $i+(a+b \cdot r) = 0 \pmod r$，而 $\lfloor (j+(a+b \cdot r))/r \rfloor = \lfloor (j+a)/r+b \rfloor = 0 \bmod r$ 意味着 $j + h(i,j) \bmod r^2 \in \{0, \ldots, r-1\}\}$，因此 $i + h(i,j)\ j + h(i,j) \in \mathbf{D}(t)$，如所需。

□

3.4 使用 Δ-近似 LZ 分解检测平方

我们算法中的一个关键概念是对标准 Lempel-Ziv 分解的以下变体：

[1]在下降过程中，如果所有子节点都按子树大小排序，比较次数会降至 $\mathcal{O}(b(\sigma/\log \sigma)\log b)$，但这似乎与我们的最终算法无关。



图 2： Δ-差分覆盖中的位置。

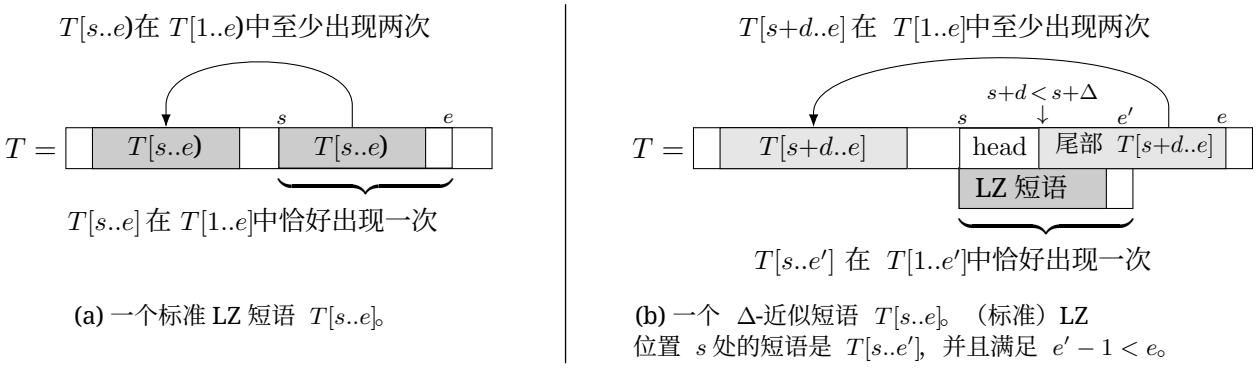


图 3：LZ 短语和 Δ-近似短语的定义示意图。

定义 3.1 (Δ-近似 LZ 分解)。 对于正整数参数 Δ，片段 $T[s..e]$ 是一个 Δ-近似 *LZ* 短语，如果它可以拆分为头部和尾部 $T[s..e]$ =头部($T[s..e]$)尾部($T[s..e]$)，使得 |头部($T[s..e]$)| < Δ，并且

- 尾部（$T[s..e]$）要么为空，要么在 $T[1..e]$ 中至少出现两次，并且
- 从位置 s 开始的唯一（标准）*LZ* 短语 $T[s..e']$ 满足 $e' - 1 \le e$。

在 Δ-近似 *LZ* 分解 $T = b_1 b_2 \dots b_z$ 中，每个因子 b_i 都是一个 Δ-近似短语$T[s..e]$，且满足 $s = 1 + \sum_{j=1}^{i-1} |b_j|$ 和 $e = \sum_{j=1}^{i} |b_j|$。

请注意，标准 LZ 短语并不是 Δ-近似短语。此外，尽管从每个位置开始的 LZ 短语（因此 LZ 分解也）都有唯一的定义，但从每个位置开始的 Δ-近似短语可能有多个不同的。这也意味着，同一个字符串可能有多个不同的 Δ-近似分解。标准 LZ 短语和 Δ-近似 LZ 短语的定义均在图 3 中示意。

上述定义背后的直觉是， Δ的值越大，构造 Δ-近似 LZ 分解就越容易。特别是，当 $\Delta = n$ 时，一个短语就足够了。我们将在下述引理中将其形式化；为得到本节的最终结果，我们将引理推广得更一般。

引理 5。 对于任意参数 $\Delta \in [1, m]$，任意长度为 m 的片段$T[x..y]$ 的 Δ-近似 *LZ* 分解，都可以通过 $\mathcal{O}(m\sigma \log m/\sqrt{\Delta})$ 次比较计算得到，此外，所有调用该引理的过程还共同分担 $\mathcal{O}(n)$ 次比较。

证明。根据引理 4，存在一个 Δ-覆盖 D(Δ)，覆盖 $\{1, \dots, n\}$，其大小为 $\mathcal{O}(n/\sqrt{\Delta})$。令 $S = \text{D}(\Delta) \cap \{x, x+1, \dots, y\}$。令 $S = \{i_1, i_2, \dots, i_b\}$。不难验证，该构造还保证 $b = \mathcal{O}(m/\sqrt{\Delta})$。我们对后缀 $T[i_1..n], \dots, T[i_b..n]$ 应用引理 3，得到它们的稀疏后缀树 T，需要进行 $\mathcal{O}(b\sigma \log b)$ 次比较，此外，所有对该引理的调用还共享 $\mathcal{O}(n)$ 次比较。 T 使我们无需额外比较即可求出任意两个片段 $T[i..y]$ 和 $T[j..y]$的最长公共前缀，其中 $i, j \in S$。根据 D(Δ) 的性质，对于任意i, $j \in \{x, x+1, \dots, y - \Delta + 1\}$，都有 $0 \le h(i,j) < \Delta$，且 $i + h(i,j)$ $j + h(i,j) \in S$。

我们逐短语计算 $T[x..y]$ 的 Δ-近似 LZ 分解。将整个 $T[x..y]$ 的剩余后缀记为 $T[x'..y]$，我们需要找到 $x' \le y' \le y$，使得 $T[x'..y']$ 是一个 Δ-近似短语。具体步骤如下。我们遍历每个满足 $x'' \in S$的 $x' \le x'' < x' + \Delta$。对于每个这样的 x''，我们考察每个满足 $a' \in S$ 的 $x \le a' < x'$，并计算 $T[x''..y]$ 和 $T[a'..y]$的最长公共前缀长度ℓ。在所有这样的 x'', a'中，我们选择使 $x''-x'+\ell-1$ 取值最大的那一对，并将下一个短语选为 $T[x'..(x''+\ell-1)]$，其头部为 $T[x'..(x''-1)]$，尾部为 $T[x''..(x''+\ell)-1]$。最后，如果不存在这样的配对，或者与找到的配对相对应的 $x''-x'+\ell-1$ 值小于 $\Delta-2$，我们就将下一个短语选为 $T[x'..\min\{x'+\Delta-1, y\}]$（尾部为空）。选择这样的配对无需额外比较，因为对于每个 $x', a' \in S$，我们都可以使用稀疏后缀树计算 ℓ。显然，生成的 Δ-近似短语具有所需的形式，但我们还需要证明它的长度足够长。

令 $T[x'..y'']$ 为 $T[x..y]$ 中以 $T[x'..y]$ 为前缀的（唯一）标准 LZ 短语。若 $y'' < x'+\Delta-1$，那么我们只需确保生成的 Δ-近似短语的长度至少为 $\min\{\Delta-1, y-x'+1\}$，而这确实成立。因此，接下来只需考虑 $y'' \ge x'+\Delta-1$ 的情况。令 $T[a..b]$ 为 $T[x..y]$ 中 $T[x'..(y''-1)]$ 的前一次出现位置（由于 $T[x'..y'']$ 是一个短语，因此该位置定义良好）。于是，$T[a..b] = T[x'..(y''-1)]$ 且 $a < x'$。由于 $y'' \ge x'+\Delta-1$ 且 $y'' \le y$，如上所述，$0 \le h(a,x') < \Delta$，并且 $a+h(a,x')\ x'+h(a,x') \in S$。我们将在上述过程中考虑 $x'' = x' + h(a,x')$ 和 $a' = a + h(a,x')$。接下来，$T[a'..b] = T[x''..(y''-1)]$，因此在考虑这一对时，我们将得到 $\ell \ge |T[x''..(y''-1)]|$。因此，对于找到的这一对，我们将有 $x''+\ell-1 \ge y''-1$，这符合 Δ-近似短语定义中的要求。 □

接下来，我们证明，尽管 Δ-近似 LZ 分解不像标准 LZ 分解那样能捕获所有不同平方，但它仍有助于检测所有足够长的平方。Δ-近似 LZ 分解的以下性质是其中的关键。

引理 6. 设 $b_1 b_2 \dots b_z$ 是字符串 T的一个 Δ-近似 *LZ* 分解。对于每个长度为 $2\ell \ge 8\Delta$的平方$T[s..s+2\ell-1]$，至少存在一个短语 b_i，其尾部$(b_i)| \ge \frac{\ell}{4} \ge \Delta$ 满足 $\mathit{tail}(b_i)$ 与该平方的右侧 $T[s+\ell..s+2\ell-1]$ 相交。

证明。假设所有与 $T[s+\ell..s+2\ell-1]$ 相交的尾部长度都小于 $\frac{\ell}{4}$，那么这些尾部对应的短语长度至多为 $\frac{\ell}{4}+\Delta-1$（因为每个头部的长度都小于 Δ）。这意味着 $T[s+\ell..s+2\ell-1]$ 至少与 $\lceil \ell/(\frac{\ell}{4}+\Delta-1) \rceil - 1 = 3$个短语相交。因此，存在某个短语 $b_i = T[x..y]$，它被严格包含在 $T[s+\ell..s+2\ell-1]$中，形式化地说，$s+\ell < x \le y < s+2\ell-1$。然而，这与 Δ-近似 LZ 分解的定义矛盾，因为 $T[x..s+2\ell]$ 是一个标准 LZ 短语的前缀（由 $T[x..s+2\ell-1] = T[x-\ell..s+\ell-1]$可知），而近似短语 $b_i = T[x..y]$ 必须满足 $y \ge s+2\ell-1$。这一矛盾说明 $T[s+\ell..s+2\ell-1]$ 与一个长度至少为 $\frac{\ell}{4} \ge \Delta$的尾部相交。 □

引理 7。 给定一个 Δ-近似 *LZ* 分解 $T = b_1 b_2 \dots b_z$，我们可以在 $\mathcal{O}\left(\sum_{|\text{尾部}(b_i)| \ge \Delta} |\text{尾部}(b_i)| + z\right)$ 时间内检测到大小为$\ge 8\Delta$ 的平方，并进行 $\mathcal{O}\left(\sum_{|\text{尾部}(b_i)| \ge \Delta} |\text{尾部}\ (b_i)|\right)$ 次比较。

证明。我们分别考虑每个短语 $b_i = T[a_1..a_3]$，其头部为 $b_i = T[a_1..a_2-1]$，尾部为 $b_i = T[a_2..a_3]$。令 $k = |\mathsf{tail}(b_i)|$。如果 $k \ge \Delta$，我们对 $x_1 = T[a_2-8k..a_2-1]$ 和$y_1 = T[a_2..a_3+4k-1]$以及修剪至 $T[1..n]$的 $x_2 = T[a_2-8k..a_3-1]$和 $y_2 = T[a_3..a_3+4k-1]$ 应用引理 1 。这需要 $\mathcal{O}(|\mathsf{tail}(b_i)|)$ 时间和比较；或者，对所有短语而言，需要 $\mathcal{O}\left(\sum_{|\text{尾部}(b_i)| \ge \Delta} |\mathsf{tail}\ (b_i)|\right)$ 时间和比较。我们还需要额外花费 $\mathcal{O}(z)$ 时间，检查每个短语是否满足 $k \ge \Delta$。

现在我们证明，所述策略能够检测到大小至少为 8Δ 的平方。令 $T[s..s+2\ell-1]$ 为任意一个这样的平方。根据引理 6，该平方的右侧 $T[s+\ell..s+2\ell-1]$ 与某个尾部 $\mathsf{tail}(b_i) = T[a_2..a_3]$ 相交，其长度为 $k = |\mathsf{tail}(b_i)| \geq \frac{\ell}{4} \geq \Delta$。由于二者相交，我们有$a_2 \leq s + 2\ell - 1$ 和 $a_3 \geq s + \ell$。因此，在处理 b_i 并应用引理 1 时， x_1 和 x_2 的起始位置满足 $a_2 - 8k \leq s + 2\ell - 1 - 8\frac{\ell}{4} = s - 1$，而 y_1 和y_2 的结束位置满足 $a_3 + 4k - 1 \geq s + \ell + 4\frac{\ell}{4} - 1 = s + 2\ell - 1$。因此，该平方完全包含在分别对应于 x_1y_1 和 x_2y_2 的片段中。如果 $s < a_2 \leq s + 2\ell - 1$，我们通过选择 x_1 和 y_1 找到该平方。如果 $s < a_3 \leq s + 2\ell - 1$，我们通过选择$x_2$ 和 y_2 找到该平方。否则， $T[s..s + 2\ell - 1]$ 完全包含在尾部 $T[a_2..a_3]$ 中，我们可以在更靠左的位置找到该平方的另一个出现位置。

□

3.5 检测平方串的简单算法

现在，我们已经具备所有工具，可以介绍一种简单方法，在假设 σ 已知的情况下，使用 $T[1..n]$ ($\mathcal{O}(\log\ \sigma + \log\log n)$) 次比较来测试 n 的无平方性。令 $\Delta = (\sigma \log\ n)^2$。我们将 $T[1..n]$ 划分为长度为 8Δ 的块，并将第 k^{th} 个块记为 B_k。调用定理 2.1，分别处理 B_1B_2、 B_2B_3 等，即可找到长度不超过 8Δ 的平方。对于每一对相邻块，这需要 $\mathcal{O}(\Delta\log\ \Delta) = \mathcal{O}(\Delta(\log\ \sigma + \log\log\ n))$ 次比较，总计需要 $\mathcal{O}(n(\log\ \sigma + \log\log n))$ 次比较。接下来还需检测长度超过 8Δ 的平方。为此，先调用引理 5，以 $\mathcal{O}(n\sigma \log\ n/\sqrt{\Delta}) = \mathcal{O}(n)$ 次比较计算 $T[1..n]$ 的 Δ-近似 LZ 分解；然后使用引理 7，另需 $\mathcal{O}(n)$ 次比较。比较次数的主导项是将定理 2.1 应用于块对所需的 $\mathcal{O}(n(\log\ \sigma + \log\log n))$ 次比较。

3.6 检测平方的改进算法

现在我们可以描述该算法：它无需知道 $\mathcal{O}(n\log\ \sigma)$ 次比较，即可运行。直观地说，我们将分阶段进行，尝试“猜测” σ的值。首先注意到，引理 5 可以扩展为以下结论。

引理 8。 存在一种算法，对于任意参数 $\Delta \in [1, m]$、估计值 $\tilde{\sigma}$ 以及长度为 m的片段 $T[x..y]$，使用 $\mathcal{O}(m\tilde{\sigma}\log m/\sqrt{\Delta})$ 次比较，另加由该引理的所有调用共同分摊的 $\mathcal{O}(n)$ 次比较；该算法要么计算出 $T[x..y]$ 的 Δ-近似 *LZ* 分解，要么判定 $\sigma > \tilde{\sigma}$。

证明。 我们运行引理 5 的证明中所述的过程，并跟踪答案为否的比较次数。一旦该次数超过 $\mathcal{O}(m\tilde{\sigma}\log m/\sqrt{\Delta})$（其中常数由复杂度分析确定），我们便知道必然有 $\sigma > \tilde{\sigma}$，因此可以终止。否则，算法会得到一个 Δ-近似 LZ 分解，使用 $\mathcal{O}(m\tilde{\sigma}\log m/\sqrt{\Delta})$ 次比较。答案为是的比较由全局共同分摊。

□

现在，我们说明如何通过 $\mathcal{O}(n\lg\ \sigma)$ 次比较找出任意平方。我们定义序列$\sigma_t = 2^{2^{\lceil \log\log n\rceil - t}}$，其中 $t = 0$、1、.....、$\lceil\log\log\ n\rceil$。我们观察到 $\sigma_{t-1} = (\sigma_t)^2$,，并按照 t 的取值分阶段进行。在第 t^{th} 阶段，我们可以保证，所有长度至少为 $(\sigma_t)^2$ 的平方都已被检测到；我们的目标是检测长度小于 $(\sigma_t)^2$，且至少为 σ_t 的平方。我们将整个 $T[1..n]$ 划分为长度为 $(\sigma_t)^2$，的块，并将第 k^{th} 个块记为 B_k。长度小于 $(\sigma_t)^2$ 的平方完全包含在某两个连续块 B_iB_{i+1} 中，因此我们考虑每一对这样的块 B_1B_2、 B_2B_3，依此类推。首先，我们应用引理 8，其中 $\Delta = \sigma_t/8$ 且 $\tilde{\sigma} = (\sigma_t)^{1/4}/\log(\sigma_t)$，以求得对应片段的一个 $(\sigma_t/8)$-近似 LZ 分解

的 $T[1..n]$，然后使用引理7 检测长度至少为 σ_t的平方。我们有时无法对所有块对都应用引理7 。因此，我们必须停用其中一些块；在分析算法所执行的比较次数时，我们会对此作出解释。如果在当前阶段调用引理8 时，有任何一次检测到 $\sigma > \tilde{\sigma}$，我们就对当前阶段的每一对块 B_iB_{i+1} 应用定理2.1 ，然后终止整个算法。

现在我们分析比较的总次数，不计所有引理 8 调用中共有的 $\mathcal{O}(n)$次比较。在 t^{th} 阶段中，我们使用 $\mathcal{O}(n \cdot \tilde{\sigma}\log\sigma_t/\sqrt{\Delta}) = \mathcal{O}(n \cdot (\sigma_t)^{1/4}/\log(\sigma_t) \cdot \log(\sigma_t)/\sqrt{\sigma_t}) = \mathcal{O}(n/(\sigma_t)^{1/4})$次比较来构造 Δ-近似分解（使用引理 8），直到处理完所有块对，或检测到 $\sigma > (\sigma_t)^{1/4}/\log(\sigma_t)$为止。后一种情况下，我们使用 $\mathcal{O}(n\log(\sigma_t))$次比较完成整个计算（使用定理 2.1）；根据对 σ 的假设，这就是所需的 $\mathcal{O}(n\log\ \sigma)$。在此情况发生之前（或直到我们到达 $t = \lceil\log\log n\rceil - 3$ 阶段，此时 $\sigma_t \le 256$），我们使用 $\mathcal{O}(\sum_{t=0}^{t'} n/(\sigma_t)^{1/4})$次比较来构造 Δ-近似分解，其中某个 $0 \le t' \le \lceil\log\log n\rceil$。为了分析总和，我们需要以下界（为下一节的目的而作了更一般化）。

引理 9. 对于任意 $0 \le x \le y$ 和 $c \ge 0$，有 $\sum_{i=x}^{y} 2^{ic}/2^{2^i} = \mathcal{O}(2^{xc}/2^{2^x})$。

证明。 我们注意到，指数序列 2^i 从 $i = 0$开始严格递增，因此

$$\sum_{i=x}^{y} \frac{2^{ic}}{2^{2^i}} \le \sum_{i=2^x}^{2^y} \frac{i^c}{2^i} \le \sum_{i=2^x}^{\infty} \frac{i^c}{2^i} = \sum_{i=0}^{\infty} \frac{(2^x+i)^c}{2^{(2^x+i)}} \le \sum_{i=0}^{\infty} \frac{2^{xc}\cdot(i+1)^c}{2^{(2^x+i)}} = \frac{2^{xc}}{2^{2^x}} \cdot \sum_{i=0}^{\infty} \frac{(i+1)^c}{2^i}.$$

$\sum_{i=0}^{\infty} \frac{(i+1)_{2^i}}{}^{c}$ 是一个正项级数，因此我们使用达朗贝尔比值判别法 $\frac{(i+2}{2^{i+1}}{}^{c} \cdot \frac{2^i)()^c}{i+1} = \frac{1}{2}\frac{}{(i+2)^c\ (i+1)^c}$，当 i 趋于无穷大时，它趋于$_1{}_2$，因此该级数收敛到一个常数。 □

推论 3. 对于任意 $0 \le t' \le \lceil$双对数 $n\rceil$，有 $\sum_{t=0}^{t'} n \cdot$多重对数$(\sigma_t)/(\sigma_t)^{1/4} = \mathcal{O}(n)$。

证明。 我们需要证明，对于任意常数 $\sum_{t=0}^{t'} n\log^c(\sigma_t)/(\sigma_t)^{1/4}) = \mathcal{O}(n)$ $c \ge 0$。我们通过拆分求和并应用引理 9来实现这一点。

$$\begin{aligned}\sum_{t=0}^{t'} \frac{n\log^c(\sigma_t)}{(\sigma_t)^{1/4}} &\le \sum_{t=0}^{\lceil\log\log n\rceil} \frac{n\cdot(2^{\lceil\log\log n\rceil - t})^c}{(2^{2^{\lceil\log\log n\rceil - t}})^{1/4}} = \sum_{t=0}^{\lceil\log\log n\rceil} \frac{n\cdot(2^t)^c}{(2^{2^t})^{1/4}} \\ &= n\cdot\sum_{t=0}^{\lceil\log\log n\rceil} \frac{2^{tc}}{2^{2^t-2}} = n\cdot\left(\frac{1}{2^{2-2}} + \frac{2^c}{2^{2-1}} + 4\cdot\sum_{t=0}^{\lceil\log\log n\rceil-2} \frac{2^{tc}}{2^{2^t}}\right) = \mathcal{O}(n)\end{aligned}$$

□

$_{\text{T}}$因此，对引理 8 的所有调用都会导致 $\mathcal{O}(\sum_{t=0}^{t'} n/(\sigma_t)^{1/4}) = \mathcal{O}(n)$ 次比较。

停用块对

接下来需要分析引理 7在所有阶段中使用的比较次数。正如前面提到的，我们实际上无法承担对所有块对应用引理 7 的代价。因此，我们引入一种机制来停用其中一些块对。

首先请注意，所有阶段中共有 $\mathcal{O}(\sum_{t=0}^{t'} n/(\sigma_t)^2) \subseteq \mathcal{O}(\sum_{t=0}^{t'} n/(\sigma_t)^{1/4}) = \mathcal{O}(n)$ 个块对。对于每个块对，我们都会记录它是否已停用；广义而言，停用意味着我们无需检查该块对，因为其中不包含最左侧不同平方。对于当前阶段 t 中的每个块对 B_iB_{i+1}，我们首先检查它是否已被标记为停用。如果没有，还要检查它是否已被隐式 停用，也就是说，是否有任意一个

包含 B_iB_{i+1} 的前一阶段中的两个块对已被标记为停用。如果 B_iB_{i+1}已被隐式停用，那么我们就将其标记为停用，并且不应用引理8和引理 7 （隐式停用的作用是将停用状态传播到所有后续阶段）。请注意，如果某个阶段中字符串的某个位置未包含在任何活动块对中，那么在所有后续阶段中，该位置也不会包含在任何活动块对中。这是因为始终有 $\sigma_{t-1} = (\sigma_t)^2$ （无需舍入），从而保证前面阶段的块边界不会与后续阶段的块相交。

我们仅在块对既未显式也未隐式停用时才应用引理 8 ，然后对 B_iB_{i+1} 应用引理 7。尾部 $T[a..a+\ell)$ 贡献 $\mathcal{O}(\ell)$ 次比较，如果 $\ell \geq \Delta = \sigma_t/8$ (否则不贡献任何比较)。由于整个 $T[a..a+\ell)$ 出现得更早，因此它不可能包含整个 T 中平方的最左出现位置。因此，包含在 $T[a..a+\ell)$ 中的任何块对（无论属于哪个阶段）也不可能包含这样的出现位置，因而可以停用这些块对。

停用所用的机制如下。设 $T[a..a+\ell)$ 是一个尾部，它在阶段 t 中、借助 $\ell \geq \Delta = \sigma_t/8$ 贡献了 $\mathcal{O}(\ell)$ 次比较。我们将完全包含在 $T[a..a+\ell)$ 中的阶段 $t+2$ 的所有块对标记为已停用。注意，阶段 $t+2$ 中的块长度为$\sqrt{\sigma_t}$，并考虑片段 $T[a+2\sqrt{\sigma_t}..a+\ell-2\sqrt{\sigma_t})$。在阶段 $t+2$ 中，以及由于隐式停用而在此后的所有阶段中，这个片段只与已停用的块对发生重叠（部分或完全重叠）。因此，在阶段 $t+1$ 之后，我们将不再检查 $T[a+2\sqrt{\sigma_t}..a+\ell-2\sqrt{\sigma_t})$ 中的任何符号。我们称尾部 $T[a..a+\ell)$ 停用了长度为 $\ell - 4\sqrt{\sigma_t} = \Omega(\ell)$ 的片段；在阶段 $t = \lceil \log\log n \rceil - 3$ 之前，该长度为正，因为 $\sigma_t > 256$。由于停用位置的数量与尾部为引理 7 贡献的比较次数呈线性关系，只需证明每个位置至多被停用常数次即可。在单个阶段中，任意位置至多被停用两次。这是因为按定义，同一分解的尾部互不重叠，但相邻块对 B_iB_{i+1} 和 $B_{i+1}B_{i+2}$ 的两个分解的尾部可能重叠。如果某个位置在阶段 t 首次被停用，那么（如前所述）我们不会在阶段 $t' \geq t+2$ 中考虑它。因此，该位置只可能在阶段 $t+1$ 再次被停用，而不会在任何更晚的阶段被停用。总之，每个位置至多被停用四次。因此，引理 7 总共贡献 $\mathcal{O}(n)$ 次比较。

我们已经证明：

定理 1.2。 对于一个包含来自一般无序字母表的 σ 个不同符号、长度为 n 的字符串，检测其是否无平方子串可通过 $\mathcal{O}(n\log\sigma)$ 次比较完成。

4 算法

本节将说明如何实现上一节所述的方法，使其能在 $\mathcal{O}(n\log\sigma)$ 时间内运行。主要难点在于高效实现稀疏后缀树构造算法，然后计算一个 Δ-近似因式分解。我们首先说明如何获得一个在 $\mathcal{O}(n\log\sigma + n\log^* n)$ 时间内运行、但仍只使用 $\mathcal{O}(n\log\sigma)$ 次比较的算法，然后进一步将其运行时间改进到 $\mathcal{O}(n\log\sigma)$。

4.1 构造后缀树和 Δ-近似因式分解

为了高效地构造引理 3中的稀疏后缀树，我们将使用最长公共扩展的一种受限版本，其中，ShortLCE 查询$_x(i,j)$（对任意正整数 x）返回 $\min(x, \mathrm{LCE}(i,j))$。Gawrychowski 等人给出了以下结果 [39]:

引理 10 ([39]中的引理 14）。对于一般无序字母表上的长度为 n 的字符串[2]，，可以在线回答 q 个 ShortLCE$_{4^{k_i}}$ 查询（针对 $i \in \{1, \ldots, q\}$），总时间为 $\mathcal{O}(n \log^* n + s)$，并且使用$\mathcal{O}(n+q)$ 次比较[3]，，其中 $s = \sum_{i=1}^{q}(k_i + 1)$。

在该引理中，除 $\mathcal{O}(n\log^* n)$ 的时间外，每次 ShortLCE$_{4^{k_i}}$ 查询还需 $\mathcal{O}(k_i + 1)$ 的时间。请注意，我们可以在线回答查询，无需预先知道查询的数量和长度。此外，在长度为 m 的片段 $T[x..y]$ 中计算最长公共扩展，可直接归约为在 T 上进行一次 ShortLCE$_{4^{\lceil \log_4 m \rceil}}$ 查询。因此，我们有：

推论 4. 在一般无序字母表上，对长度为 m 的片段 $T[x..y]$ 进行 q 次最长公共扩展查询，可以在 $\mathcal{O}(q \log\ m)$ 时间内回答，此外，所有引理调用共享 $\mathcal{O}(n\log^* n)$ 时间。比较次数为 $\mathcal{O}(q)$，此外，所有引理调用共享 $\mathcal{O}(n)$ 次比较。

构造稀疏后缀树时，我们将使用 Gabow [37] 提出的重建方案维护轻重分解。令 $L(u)$ 表示节点 u 的子树中的叶节点数。我们采用以下递归方式构造轻重分解：从一个节点 r （初始为树的根）出发，找到满足 $L(e) \geq \frac{5}{6}L(r)$（可能为 $e = r$）的最深后代节点 e 。从根 $r(p)$ 到 $e(p)$ 的路径 p 是一条重路径。该路径上的任意边 (u, v) 均满足 $L(v) \geq \frac{5}{6}L(u)$，我们称这些边为重边。因此，节点 u 至多有一个子节点 v 满足边 (u, v) 为重边。对于每条边 (u, v)，若 u 位于重路径上而 v 不在其上，则从 v 开始递归构造一条新的重路径。

向树中插入一个新后缀时，我们按以下方式跟踪插入操作：对于每条重路径的根节点，我们维护 $I(u)$ 自构建该子树的轻重分解以来，在 u的子树中进行的插入操作次数。当 $I(u) \geq \frac{1}{6}L(u)$ 时，我们会重新计算 $L(v)$ 中 v 的所有节点在 u 子树中的值，并为 u的子树重新构建轻重分解。

这确保了即使进行插入操作，对于从节点 r 出发的任意重路径，以及该重路径上的任意节点 u，$L(e) \geq \frac{2}{3}L(r)$。跨越轻边时，子树中的节点数至少缩小 5 6 倍，因此有以下性质：

观察 4.1. 从任意节点到根节点的路径至多经过 $\mathcal{O}(\log m)$ 条重路径。

重建一个大小为 s 的子树需要 $\mathcal{O}(s)$ 时间，而向树中添加一个后缀 $T[i_j..y]$ 会使从根节点 r 到新叶节点的每条路径 p 的 $I(r)$ 增加。这些节点至多有 $\mathcal{O}(\log m)$ 个，因此，维护重路径分解的每次插入操作摊还需要 $\mathcal{O}(\log n)$ 时间。

现在这些构件都已明确定义，我们可以开始描述稀疏后缀树的构造。

引理 11。 包含任意 b个后缀 $T[i_1..y], \ldots, T[i_b..y]$ （属于 $T[x..y]$，且满足$m = |T[x..y]|$）的稀疏后缀树，可以用 $\mathcal{O}(b\sigma \log\ b \log\ m)$ 时间构造，另需由该引理的所有调用共享 $\mathcal{O}(n \log^* n)$ 时间。

证明。 与引理 3的证明一样，我们考虑将后缀 $T[i_j..y]$ 插入后缀为 $T[i_1..y], T[i_2..y] \cdots T[i_{j-1}..y]$ 的稀疏后缀树中。在任何时刻，我们都维护重路径

[2]文献 [39] 中的引理 14 并未明确说明它适用于一般无序字母表。不过，该引理的证明仅依赖相等性测试。[3]文献 [39] 中的引理 14 并未明确说明它需要 $\mathcal{O}(n+q)$ 次比较。不过，该引理采用并查集方法，其中结果为“相等”的比较最多有 $\mathcal{O}(n)$ 次；每次最长公共扩展查询仅执行一次结果为“不相等”的比较，这与我们在引理 3 的证明中的描述类似。

分解。此外，我们为每条重路径维护一个前驱数据结构。给定长度 ℓ，我们可以快速确定重路径上最深的、表示长度至少为 ℓ 的字符串的显式节点。该数据结构例如可以是平衡二叉搜索树，支持插入和搜索操作，时间为 $\mathcal{O}(\log b)$（最终的稀疏后缀树以及其中的每条重路径都包含 $\mathcal{O}(b)$ 个节点）。重建重路径分解中的某个子树时，我们还必须为其中的每条重路径重建前驱数据结构。因此，重建一个大小为 q 的子树需要 $\mathcal{O}(q \log b)$ 时间（每个节点恰好位于一条重路径上，并且必须插入一个前驱数据结构），摊销插入时间也因此从 $\mathcal{O}(\log m)$ 增加到$\mathcal{O}(\log m \cdot \log b)$。每当插入一个后缀时，我们至多显式化一个节点，因此至多需要向一个前驱数据结构执行一次插入。该操作需要 $\mathcal{O}(\log b)$ 时间，低于前一项。

插入 $T[i_j..y]$ 时，我们查找节点 u，它对应于 $T[i_j..y]$ 与待插入后缀之间的最长公共前缀；如有必要，将 u 设为显式节点，并添加一个对应于 $T[i_j..y]$ 的新叶节点，将其连接到 u。令 v 为当前节点（初始化为根节点，且始终是显式节点），v_1、$\cdots$、v_d 为其（显式）子节点。如果对于 $1 \le a \le d$ 存在一条重边 (v, v_a)，令 p 为对应的重路径。对于每条重路径 p，我们存储一个叶节点的标签（即某个后缀的起始位置），该叶节点位于 $e(p)$ 的子树中。因此，我们可以利用推论 4 计算由 $e(p)$ 表示的字符串与 $T[i_j..y]$ 之间的最长公共扩展。现在，我们在这条重路径上使用前驱数据结构，找出路径上最深的（显式或隐式）节点 v'，使其表示 $T[i_j..y]$ 的一个前缀。如果 v' 是隐式节点，我们就将其设为显式节点并添加叶节点。如果 v' 是显式节点且 $v' \neq v$，我们就将 v' 作为新的当前节点并继续。否则，我们有 $v' = v$，也就是说，该后缀不属于以 v_a 为根的子树。在这种情况下，我们对 $T[i_j..y]$ 与节点 v_1、.....、v_d 所表示的每个字符串发起 d 次最长公共扩展查询。这要么表明我们可以将其中一个 v_a 作为新的当前节点继续处理，要么表明我们可以在某条 (v, v_a) 边上创建一个新的显式节点并将叶节点连接到该节点，要么表明我们可以直接将一个新叶节点连接到 v。

现在我们分析插入一个后缀所花费的时间。将 $\mathcal{O}(b)$ 个节点插入动态重路径分解和前驱数据结构，总共花费了 $\mathcal{O}(b \cdot \log m \cdot \log b)$ 时间。在插入过程的每一步中，我们要么 (i) 尽可能沿某条重路径移动，要么 (ii) 沿某条轻边移动。对于 (i)，我们执行一次最长公共扩展查询和一次前驱查询。对于 (ii)，我们执行 $\mathcal{O}(\sigma)$ 次最长公共扩展查询。根据观察 4.1，对于每个后缀，(i) 和 (ii) 各自最多发生 $\mathcal{O}(\log b)$ 次。因此，对于所有后缀，我们执行 $\mathcal{O}(b \log b)$ 次前驱查询和 $\mathcal{O}(b\sigma \log b)$ 次最长公共扩展查询。前驱查询的总时间为 $\mathcal{O}(b \log^2 b)$，最长公共扩展查询的总时间为 $\mathcal{O}(b\sigma \log b \log m)$（此外，推论 4 的所有调用共同花费 $n \log^* n$ 时间）。

□

引理 12. *对于任意参数 $\Delta \in [1, m]$，任意长度为 m 的片段$T[x..y]$ 的 Δ-近似 LZ 分解可在 $\mathcal{O}(m\sigma \log^2 m/\sqrt{\Delta})$ 时间内计算，另需所有引理调用共同分摊 $\mathcal{O}(n \log^* n)$ 时间。*

证明。设 $T' = T[x..y]$，并令 $\{i_1, i_2, \ldots, i_b\}$ 为 $\{1, \ldots, m\}$ 的一个 Δ-覆盖，这意味着 $b = \Theta(m/\sqrt{\Delta})$。根据引理 11，我们可以构造后缀 $T'[i_1..m]$、.....、$T'[i_b..m]$ 的稀疏后缀树，耗时 $\mathcal{O}(b\sigma \log b \log m) \subseteq \mathcal{O}(m\sigma \log^2 m/\sqrt{\Delta})$，另需所有引理调用共同分摊 $\mathcal{O}(n \log^* n)$ 时间。现在，我们在 $\mathcal{O}(b)$ 时间内，根据稀疏后缀树计算 T' 的 Δ- 近似 LZ 分解。

在以下证明中，我们交替使用 $i_1, i_2, \ldots, i_b$ 来表示差分覆盖位置及其在稀疏后缀树中对应的叶节点。假设差分覆盖位置的顺序为 $i_1 < i_2 < \cdots < i_b$。首先，我们为每个 $i_k > i_1$ 确定

位置 $\mathsf{src}(i_k) = i_h$ 和长度 $\mathsf{len}(i_k) = \mathrm{LCE}(i_h, i_k)$，其中 $i_h \in \{i_1, \ldots, i_{k-1}\}$ 是使 $\mathrm{LCE}(i_h, i_k)$ 最大的位置。这与 [34] 中针对 LZ77 分解所做的工作类似。我们首先为稀疏后缀树的节点赋予从 $\{1, \ldots, b\}$ 开始的标签。当且仅当 i_k 是该节点最小的后代叶节点时，该节点的标签为 k。我们按如下方式赋予标签。初始时，所有节点均未标记。我们为从 i_1 到根节点路径上的每个节点赋予标签 1。然后，按递增顺序处理其余叶节点 $i_2, \ldots, i_b$。对于每个 i_k，我们沿着从 i_k 到根节点的路径向上遍历。遇到尚未标记的节点时，我们为其赋予标签 k。一旦到达一个已经标记的节点（例如，其标签为 h，字符串深度为 ℓ），就完成了对叶节点 i_k 的处理。容易看出，i_h 恰好也是使 $\mathrm{LCE}(i_h, i_k)$ 最大的所需索引，并且有 $\mathrm{LCE}(i_h, i_k) = \ell$。因此，我们找到了 $\mathsf{src}(i_k) = i_h$ 和 $\mathsf{len}(i_k) = \ell$。所需的总时间与稀疏后缀树节点数成线性关系，即为 $\mathcal{O}(b)$。

最后，我们使用 src 和 len 得到一个 Δ-近似 LZ 分解。先前计算出的值可以解释如下：i_k 可以成为长度为 $\mathsf{len}(i_k)$ 的尾部的起始位置（其前一次出现的位置为 $\mathsf{src}(i_k)$）。对于 Δ-近似 LZ 分解，我们将从左到右贪心地创建各个因子。假设我们已经分解了 $T'[1..s-1]$，那么下一个短语从位置 s 开始，因此下一个尾部从 $T'[s..s+\Delta$ 中的某个位置开始（提醒一下，根据定义，头部的长度小于 Δ）。令 $S = \{i_1, i_2, \ldots, i_b\} \cap \{s, \ldots, s+\Delta-1\}$。如果不存在满足 $i_k + \mathsf{len}(i_k) > s + \Delta - 1$ 的 $i_k \in S$，那么下一个短语就是 $T'[s..\min(|T'|, s+\Delta-1))$，且尾部为空。否则，下一个短语包含（可能为空的）头部 $T'[s..i_k)$ 和尾部 $T'[i_k..i_k + \mathsf{len}(i_k))$（其前一次出现位置为 $\mathsf{src}(i_k)$），其中 i_k 从 S 中选取，使 $i_k + \mathsf{len}(i_k)$ 最大。以这种方式创建短语显然需要 $\mathcal{O}(|S|)$ 时间。由于下一个短语最早从位置 $s+\Delta-1$ 开始，因此 $S \setminus \{s+\Delta-1\}$ 中的所有位置都不会再被考虑为其他尾部的起始位置。于是，在创建短语的过程中，每个 i_k 至多被考虑两次，创建所有短语所需的总时间为 $\mathcal{O}(b)$。

还需要证明，计算得到的分解确实是 Δ-近似 LZ 分解；也就是说，如果我们输出短语 $T'[s..e]$，那么从位置 s 开始的唯一（非近似）LZ 短语 $T'[s..e']$满足 $e' - 1 \le e$。首先注意，对于构造出的近似短语（可能是 T 的最后一个短语除外），我们有 $s + \Delta - 2 \le e$。假设 $e' < s + \Delta$，那么显然 $e' - 1 \le e$。因此，我们只需考虑 $e' > s + \Delta - 1$。由于 $T'[s..e']$ 是一个 LZ 短语，存在某个 $s' < s$ 使得 $\mathrm{LCE}(s', s) = e' - s$。令 h 为定义 Δ-覆盖的可在常数时间内计算的函数，并令 $i_{k'} = s' + h(s', s)$ 和 $i_k = s + h(s', s)$。注意，$i_{k'} \in \{i_1, i_2, \ldots, i_{k-1}\}$ 且 $i_k \in \{i_1, i_2, \ldots, i_b\} \cap \{s, \ldots, s+\Delta-1\}$。因此，我们有 $\mathsf{len}(i_k) \ge \mathrm{LCE}(i_{k'}, i_k) = \mathrm{LCE}(s', s) - h(s', s) = (e' - s) - (i_k - s) = e' - i_k$。在计算 Δ-近似短语 $T'[s..e]$时，我们将 i_k 视为尾部的起始位置，这意味着$e \ge i_k + \mathsf{len}(i_k) - 1 \ge e' - 1$。 □

引理 13. *对于任意参数 $\Delta \in [1, m]$、估计值 $\tilde{\sigma}$ 和长度为 m 的片段 $T[x..y]$，存在一种算法，其运行时间为 $\mathcal{O}(m\tilde{\sigma}\log^2 m/\sqrt{\Delta})$，此外，所有引理调用共享 $\mathcal{O}(n \log^* n)$ 的运行时间；该算法要么计算出 $T[x..y]$ 的 Δ-近似 LZ 分解，要么确定 $\sigma > \tilde{\sigma}$。*

证明。 我们只需使用引理 12 来计算分解。第一步中，我们必须使用引理 11中的算法构造稀疏后缀树。尽管该算法需要$\mathcal{O}(m\sigma \log^2 m/\sqrt{\Delta})$ 时间，但不难看出，更精确的时间界为 $\mathcal{O}(md\log^2 m/\sqrt{\Delta})$，其中 d 是稀疏后缀树中任意节点的最大度。如果构造过程中某个节点的最大度达到 $\tilde{\sigma}+1$，我们就立即停止并返回该 $\sigma > \tilde{\sigma}$。否则，我们便能在所需时间内完成构造。

□

现在我们可以描述这个算法，它能在 $\mathcal{O}(n\lg\sigma + n\log^* n)$ 时间内检测平方，并且$\mathcal{O}(n\lg\sigma)$ 次比较。我们只需使用第 3.2 节中的算法，但用引理 13 替代

引理 8。接下来，我们分析除所有引理 13 调用共同花费的 $\mathcal{O}(n\log^* n)$ 时间之外所需的时间。在整个 t^{th} 阶段中，我们使用 $\mathcal{O}(n\cdot\tilde{\sigma}\cdot\log^2(\sigma_t)/\sqrt{\Delta})$ $=\mathcal{O}(n\cdot(\sigma_t)^{1/4}/\log(\sigma_t)\cdot\log^2(\sigma_t)/\sqrt{\sigma_t})$ $=\mathcal{O}(n\log(\sigma_t)/(\sigma_t)^{1/4})$ 次比较，来构造所有 Δ- 近似分解。和之前一样，如果我们在任何时刻发现 $\tilde{\sigma}>(\sigma_t)^{1/4}/\log(\sigma_t)$，就使用定理 2.1，在 $\mathcal{O}(n\lg\sigma_t)=\mathcal{O}(n\log\sigma)$ 时间内完成计算。在此之前（或直到完成全部 $\lceil\log\log n\rceil$ 阶段），我们使用 $\mathcal{O}(\sum_{t=0}^{t'}n\log(\sigma_t)/(\sigma_t)^{1/4})$ 时间；根据推论 3，这为 $\mathcal{O}(n)$。检测平方时，我们仍使用引理 7；如第 3.2 节所述，该引理总计需要 $\mathcal{O}(n)$ 时间和比较次数，另需 $\mathcal{O}(Z)$ 时间，其中 Z 是所有引理调用过程中考察的近似 LZ 因子的数量。我们对每个近似 LZ 分解恰好应用一次该引理，并且根据构造，第 t 阶段中的每个因子大小至少为 $\Delta=\Omega(\sigma_t)$。此外，每个文本位置在每个阶段至多被两个尾部覆盖。因此$Z=\mathcal{O}(\sum_{t=0}^{t'}n/\sigma_t)$，根据推论 3，这为 $\mathcal{O}(n)$。

最后需要说明的是如何实现块的记账，即在每个阶段，我们都必须按照第 3.2 节末尾所述，高效地停用块对。我们用 $\lceil\log\log n\rceil$ 个位向量来维护块对，总长度为 $\mathcal{O}(n)$，其中置位表示相应的块对已被停用（回想一下，块对总数为 $\mathcal{O}(n)$）。位向量 t 在位置 j 处包含对应于块对 $B_jB_{j+1}=T[i..i+2(\sigma_t)^2)$ 的位，其中 $i=(\sigma_t)^2\cdot(j-1)$。注意，在 i 和 j 之间转换只需常数时间。对于阶段 t 中每个足够长的尾部，我们只需遍历阶段 $t+2$ 中相关的块对并将其停用，即设置对应的位。所需时间与被停用块的数量成线性关系。由于块对共有 $\mathcal{O}(n)$ 个，且每个块对最多只会被停用常数次，因此这项记账的总成本为 $\mathcal{O}(n)$。

比较次数主要由以下部分决定： $\mathcal{O}(n\log\sigma)$ 次比较，这是利用定理 2.1完成计算时所用的。其他比较仅由引理7执行；我们已将其界定为 $\mathcal{O}(n)$；此外，推论4中的最长公共扩展查询也会执行比较。由于总共进行了 $\mathcal{O}(n)$ 次此类查询，因此比较次数也为 $\mathcal{O}(n)$。由此我们得到：

引理 14. 第 *3.2* 节中的平方检测算法可在 $\mathcal{O}(n\lg\sigma+n\log^* n)$ 时间内实现，并使用 $\mathcal{O}(n\log\sigma)$ 次比较。

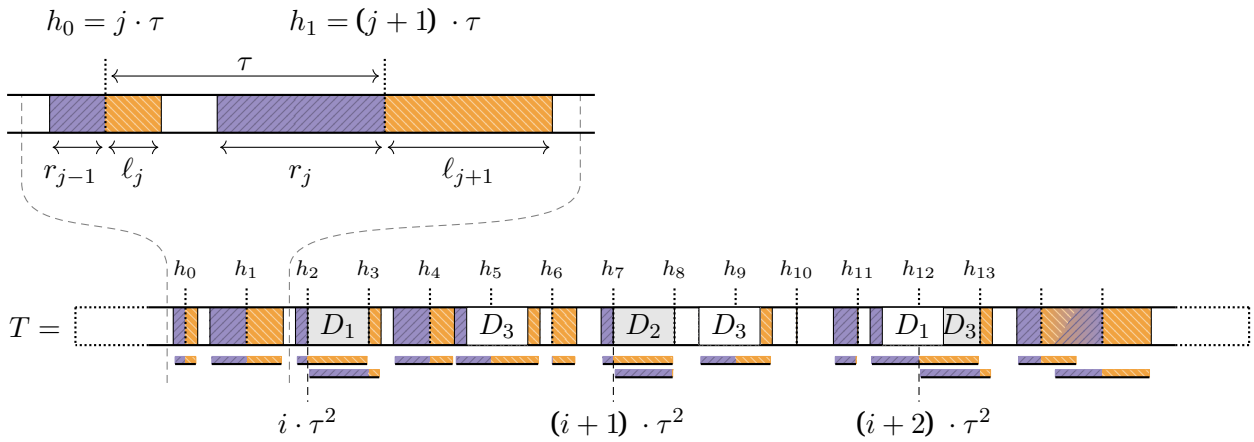
4.2 最终改进

为了实现最后的改进，我们需要用自己的机制替换由推论 4 实现的最长公共扩展查询。目标仍然相同：给定参数 Δ 和字母表大小的估计值 $\tilde{\sigma}$，找出任意片段 Δ的 $T[x..y]$-近似 LZ 分解，使其在 $\mathcal{O}(m\tilde{\sigma}\log m/\sqrt{\Delta})$ 时间内完成，其中 $m=|T[x..y]|$ （当 $m=\Theta(\Delta^2$ 时，我们无需检测任何内容）。

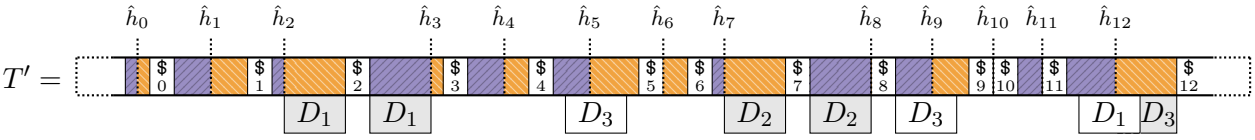
与上一节一样，算法可能会检测到字母表的大小大于 $\tilde{\sigma}$，此时我们便改用分治算法。令 $\tau=\lfloor\sqrt{\Delta}\rfloor$。

起初，我们只考虑 $T[x..y]$ 的一些片段。我们称 $T[i\cdot\tau^2..i\cdot\tau^2+\tau)$ 为稠密片段。首先，将所有与 $T[x..y]$ 相交的稠密片段中的字符重新映射到一个可线性排序字母表中。对于每个位置，通过维护一个已见不同字符的列表，可以在 $\mathcal{O}(\tilde{\sigma})$ 时间内完成此操作。对于稠密片段中的每个位置，我们遍历列表中的字符；如果某个字符尚未出现，则可能将其追加到列表中。一旦列表大小超过 $\tilde{\sigma}$，我们就终止该过程并改用分治算法。否则，我们将每个字符替换为它在列表中的位置。总而言之，$T[x..y]$ 的稠密片段中共有 $\mathcal{O}(m/\sqrt{\Delta})$ 个位置，重新映射需要 $\mathcal{O}(m\tilde{\sigma}/\sqrt{\Delta})$ 时间。

接下来，我们构造两棵广义后缀树 [43], 第一棵包含所有稠密片段，第二棵包含它们的反转。（一组字符串的广义后缀树是包含该集合中所有字符串的所有后缀的压缩



(a) 对稠密片段进行采样，并将文本切分为块。虚线表示块边界，$h_x = (j+x) \cdot \tau$对于某个整数 j 和 $x \in [0, 13]$ 是块边界的位置。稠密片段为 $D_1 = T[h_2..h_3)$、$D_2 = T[h_7..h_8)$ 和 $D_3 = T[h_{12}..h_{13})$。稠密片段的主要出现位置以灰色表示，而次要出现位置（我们要找出的那些）以白色表示。文本中的紫色方框以及文本下方对应的紫色线段表示某个子串 $T[j \cdot \tau - r_{j-1}..j \cdot \tau)$。同理，橙色方框和线段表示子串 $T[j \cdot \tau..j \cdot \tau + \ell_j)$。



(b) 用于查找稠密片段所有出现位置的字符串 T'。每个位置 $\hat{} h_x$ 映射到图中的位置 h_x，见图 4a。位于 $h_x = (j+x)\tau$ 之前的紫色方框所标示的子串，以及位于 h_x 之后的橙色方框所标示的子串，恰好是 $T[h_x - r_{j+x-1}..h_x + \ell_{j+x})$。每个 \$ 都是不同的分隔符符号，并且在 T'中是唯一的。$_x$

图4：第4.2节的补充示意图。

字典树。）同样，由于现在处理的是可线性排序的字母表，这只需 $\mathcal{O}(m/\sqrt{\Delta})$ 时间 [31]。我们考虑与 $T[x..y]$ 有非空交集的形如$T[i \cdot \tau..(i+1) \cdot \tau)$ 的片段。我们称这样的片段为块。注意，块的数量为 $\mathcal{O}(m/\sqrt{\Delta})$，其总长度为 $\mathcal{O}(m)$。对于每个块，我们找出其最长前缀 $T[i \cdot \tau..i \cdot \tau + \ell_i)$ 和最长后缀 $T[(i+1) \cdot \tau - r_i..(i+1) \cdot \tau)$，它们都出现在某个稠密片段中。图 4a 展示了稠密片段、块以及最长前缀和后缀。我们分别沿着所有稠密片段及其反转所构成的广义后缀树的重路径分解，即可高效完成此操作。在当前重路径上，我们只需尽可能直接匹配字符。若发生不匹配，则花费 $\mathcal{O}(\tilde{\sigma})$ 时间下降到相应的子树；由于重路径分解，这种情况最多发生 $\mathcal{O}(\log m)$ 次。找到 ℓ_i 和 r_i 后，我们检验 $T[i \cdot \tau..i \cdot \tau + \ell_i)$ 和$T[(i+1) \cdot \tau - r_i..(i+1) \cdot \tau)$ 是否无平方子串。由于它们都出现在稠密片段中，且我们已对所有稠密片段的字母表重新映射，因此可以利用定理 2.2 在 $\mathcal{O}(\ell_i + r_i)$ 时间内完成此操作。因此，每个块所需的总时间为 $\mathcal{O}(\tilde{\sigma} \log m)$ 加上 $\mathcal{O}(\ell_i + r_i)$。前者总计为 $\mathcal{O}(m\tilde{\sigma} \log m/\sqrt{\Delta})$；我们稍后会说明，后者可通过停用较低层上的块来摊还。

—

到目前为止，我们已将所有稠密片段的字母表重新映射为可线性

排序的字母表，并且对于每个块，我们都知道它在某个稠密片段中出现的最长前缀和后缀。我们将所有形如 $T[i\cdot\tau-r_{i-1}..i\cdot\tau+\ell_i)$（与 $T[x..y]$的交集）的片段串接起来，并在其间添加互不相同的分隔符，形成一个新的字符串 T'。需要强调的是，由于我们已重新映射所有稠密片段的字母表，且每个块找到的最长前缀和后缀也都出现在某个稠密片段中，因此 T' 使用的是可线性排序的字母表。因此，我们可以在 ST上用 T'的 $\mathcal{O}(|T'|$时间构建 [31]后缀树 T'。图4b展示了

设 $\mathcal{D}=\{D_1,D_2,\ldots\}$ 为不同的稠密片段集合。我们希望构造 $\mathcal{D}$ 中各字符串在 $T[x..y]$中的所有出现位置集合。利用 T' 的后缀树，我们可以检索 T'中每个 D_j 的所有出现位置。注意，由于我们对 $T[i\cdot\tau..i\cdot\tau+\ell_i)$ 和 $T[(i+1)\cdot\tau-r_i..(i+1)\cdot\tau)$ 的定义，这实际上会给出原始$T[x..y]$中每个 D_j的所有出现位置。为了高效实现这一过程，我们按如下步骤进行。首先，对于每个 i，我们从根节点开始遍历 ST，找到与稠密片段$T[i\cdot\tau^2..i\cdot\tau^2+\tau)$对应的（显式或隐式）节点。这只需 $\mathcal{O}(m\tilde{\sigma}/\sqrt{\Delta})$ 时间。然后，以这类节点为根的每棵子树中的所有叶节点都对应于某个 D_j的出现位置；遍历子树即可报告这些位置，所需时间与子树大小成正比，因此总计至多为 $\mathcal{O}(|T'|)$。最后，通过预先计算 T'中每个位置在 $T[x..y]$中的对应位置，即可在每个出现位置花费常数时间，将出现位置映射回 $T[x..y]$；构造T'时即可在 $\mathcal{O}(|T'|)$ 时间内完成这项预处理。因此，在 $\mathcal{O}(|T'|)$ 时间内，我们便可得到集合 S，其中包含 $\mathcal{D}$中各字符串所有出现位置的起始位置。下面总结 S 的性质。

命题 1. S 具有以下性质：

1. 对于每个满足 $i=0\,(\mathrm{mod}\ \tau^2)$ 的 $i\in[x,y]$，$i\in S$。*2.* 对于每个 $i\in[x,y-\tau]$，$i\in S$ 当且仅当 $T[i..i+\tau)\in\mathcal{D}$。*3.* $|S|\le|T'|$。

现在，我们基于 S 定义对 $T[x..y]$\$ 的解析。令 $i_1<i_2<\ldots i_k$ 为 S 中所有位置，即，对于每个 $j=1,2,..,k-1$，有 $(i_j,i_{j+1})\cap S=\emptyset$。对于每个 $j=1,2,..,k-1$，我们创建短语 $T[i_j..i_{j+1}+\tau)$。我们添加最后一个短语 $T[i_k..y]$\$。需要强调的是，相邻短语会重叠 τ 个字符，并且每个短语都以一个长度为 τ 的片段开头，该片段从 S 中的某个位置开始。结合 S 的性质 2，可得以下性质。

观察 4.2. 这些不同的短语构成前缀无关集合。

我们希望构造所有这类短语的压缩字典树 $\mathcal{T}_{\text{phrase}}$，从而（尤其是）识别出相同的短语。首先注意到，每个短语都以一个片段 $T[i_j..i_j+\tau)$ 开头，该片段在 T' 中有对应的出现位置。注意，给定 T 中的一组位置 P，我们可以通过在 $\mathcal{O}$ 中排序并扫描，在 $|P|+|T'|$ 时间内找到它们在 T' 中对应的位置（如果存在）。

因此，我们可以假设，对于每个 i_j，都知道它在 T' 中对应的位置 i'_j。接下来，我们在 $|T'|$ 时间内预先计算 ST 的每个节点在 $\mathcal{O}$ 中字符串深度为 τ 的唯一祖先。然后，对于每个片段 $T[i_j..i_j+\tau)$，我们都可以访问它在 ST 中对应的（隐式或显式）节点。这使我们能够根据所有短语长度为 τ 的前缀对其进行划分。事实上，这样便能在 $\mathcal{O}$ 时间内得到包含所有这类前缀的 $\mathcal{T}_{\text{phrase}}$ 顶部部分，并且对于每个短语，我们都可以假设已知 $\mathcal{T}_{\text{phrase}}$ 中对应其长度为 τ 的前缀的节点。

为构建 $\mathcal{T}_{\text{phrase}}$ 的剩余部分，我们将短语分为短和长两类。当 $i_{j+1}\le i_j+\tau$（即其长度至多为 2τ）时，$T[i_j..i_{j+1}+\tau)$ 属于短语；否则属于长语。

我们首先构造包含所有短语的压缩字典树 $\mathcal{T}'_{\text{phrase}}$。这可以采用与构造 $\mathcal{T}_{\text{phrase}}$顶部部分类似的方法完成，只不过现在这些片段的长度可能

各不相同。不过，每个短语 $T[i_j..i_{j+1}+\tau)$ 都会在 T' 中以 $T'[i'_j..i'_{j+1}+\tau)$ 的形式出现。我们声称，对于每个 $T'[i'_j..i'_{j+1}+\tau)$， ST 中对应的节点都可以在 $\mathcal{O}(|T'|)$ 时间内找到。具体做法是按深度优先顺序遍历 ST，同时维护一个栈，其中包含当前路径上所有字符串深度至少为 τ 的显式节点。随后，访问对应于从 T' 的 i'_j 位置开始的后缀的叶节点时，我们遍历当前栈以找到所需节点。此过程最多需要 $\mathcal{O}(|T'[i_j+\tau..i_{j+1}+\tau]|)$ 时间，总计为 $\mathcal{O}(|T'|)$。找到 ST 中对应于 $T[i_j..i_{j+1}+\tau)$ 的节点后，我们在 $\mathcal{O}(|T'|)$ 时间内从 ST 中提取 $\mathcal{T}'_{\text{短语}}$。

有了 $\mathcal{T}'$ 短语后，我们按如下方式构造整个 $\mathcal{T}_{\text{phrase}}$。首先，将 $\mathcal{T}'$ 短语与已获得的 $\mathcal{T}_{\text{phrase}}$ 顶部部分取并集，这可以在 $\mathcal{O}(|T'|)$ 时间内完成。对于每个长短语 $T[i_j..i_{j+1}+\tau)$，我们知道与 $T[i_j..i_j+\tau)$ 对应的节点，并希望将整个字符串 $T[i_j..i_{j+1}+\tau)$ 插入 $\mathcal{T}_{\text{phrase}}$。我们按 i_j 的递增顺序执行插入操作（这对之后摊还时间至关重要）。这通过类似第4.1节中的动态重路径分解来实现，但有一项重要变化。具体来说，我们固定 $\mathcal{T}_{\text{phrase}}$ 中与 $\mathcal{T}'_{\text{phrase}}$ 和 $\mathcal{T}_{\text{phrase}}$ 的顶部部分之并集对应部分的重路径分解，并维护从该部分伸出的每棵子树的动态重路径分解。得益于这一变化，维护动态字典树和所有重路径分解所需的时间为 $\mathcal{O}(m\log m/\sqrt{\Delta})$，因为长短语只有 $\mathcal{O}(m/\sqrt{\Delta})$ 个。接下来，对于每个长短语 $T[i_j..i_{j+1}+\tau)$,—我们从已知的、与 $T[i_j..i_j+\tau)$ 对应的节点开始插入，然后沿重路径继续插入：首先沿 $\mathcal{T}_{\text{phrase}}$ 中与 $\mathcal{T}'_{\text{phrase}}$ 对应部分的静态重路径分解进行，其次沿相应子树中的动态重路径分解进行。在每条重路径上，我们以朴素方式尽可能多地匹配字符。插入单个短语 $T[i_j..i_{j+1}+\tau)$ 所需的时间为 $\mathcal{O}(\log\ m)$（两次），再加上 $T[i_j+\tau..i_{j+1}+\tau)$ 中与某个 $T[i_{j'}+\tau..i_{j'+1}+\tau)$ 的前缀相同的最长前缀的长度，其中 $j'<j$。前一部分的总和为另一个 $\mathcal{O}(m\log m/\sqrt{\Delta})$，我们稍后将说明，后一部分可以通过停用较低层的块来摊还。

—

$\mathcal{T}_{\text{phrase}}$ 使我们能够构造与这些短语对应的元字符，并将 $T[x..y]$ 转换为由这些元字符组成、长度为 $\mathcal{O}(|T'|)$ 的字符串 T_{parse}。我们在这些可线性排序的元字符上构建字符串 $\mathcal{O}(|T'|)$ 的后缀树 $\mathcal{S}_{\text{parse}}$，耗时为线性时间。接下来，我们按如下方式将其转换为包含所有后缀 $T[i_j..y]$ 的稀疏后缀树 $\mathcal{S}'_{\text{parse}}$。考虑一个显式节点 $u\in\mathcal{S}_{\text{parse}}$，其子节点为 v_1、v_2、……、v_d、$d\geq 2$。我们首先计算由对应于边 (u,v_i) 上第一个元字符的叶节点所诱导的 $\mathcal{T}_{\text{phrase}}$ 的子树 $\mathcal{T}_u$，其中 $i=1$ 取 2、……、d，并将每个 v_i 连接到 $\mathcal{T}_u$ 中相应的叶节点。假设可以在常数时间内进行关于 $\mathcal{T}_{\text{phrase}}$ [10] 的最近公共祖先查询，并使用栈从左到右处理叶节点，这一过程可在 $\mathcal{O}(d)$ 时间内实现；方法与笛卡尔树构造算法 [75] 类似。注意，叶节点的顺序与元字符的顺序相同，因此无需额外排序。总计耗时为 $\mathcal{O}(|T'|)$。接下来我们注意到，除非 u 是 $\mathcal{S}_{\text{parse}}$ 的根节点，否则边 (u,v_i) 上的所有元字符都对应于以同一长度为 τ 的前缀开头的字符串。通过截去此前缀（如果 u 是根节点，则取 $\mathcal{T}_u$），得到子树 $\mathcal{T}'_u$。最后，将 $\mathcal{T}'_u$ 的根节点与 u 合并，并将每个子节点 v_i 与 $\mathcal{T}'_u$ 中对应的叶节点合并。由于我们截去了长度为 τ 的重叠前缀，在对 $\mathcal{S}_{\text{parse}}$ 的每个节点执行此过程后，就得到一棵树 $\mathcal{S}'_{\text{parse}}$，其中每个叶节点都对应一个后缀 $T[i_j..y]$。此外，根据观察 4.2，每个节点发出的边都以不同的字符开头，符合要求。

沿用引理 12 的证明思路，$\mathcal{S}'_{\text{parse}}$ 使我们能够确定，对于每个后缀 $T[i_j..y]$，其与某个 $T[i'..y]$ 的前缀相等的最长前缀，其中 $i'<i_j$，只要其长度至少为 τ。事实上，在这种情况下，由性质 2 可知必有 $i'\in S$，因此实际上 $i'=i_{j'}$，只需最大化它与 S 中所有更早位置的公共前缀长度，这可以通过 $\mathcal{S}'_{\text{parse}}$ 完成。因此，我们要么知道这个最长前缀的长度小于 τ，要么知道其确切值（以及对应的位置 $i'\in S$）。

引理 15。 对于任意参数 $\Delta \in [1, m]$ 和字母表大小的估计值 $\tilde{\sigma}$，任意片段 $T[x..y]$ 的一个 $(\Delta + \tau)$ 近似 *LZ* 分解可在 $\mathcal{O}(m/\sqrt{\Delta})$ 时间内计算得出，且$m = |T[x..y]|$（假设已完成本节前文所述的预处理）。

证明。令 $e \in [x, y]$，并假设我们已经构造出 $T[x..e-1]$ 的分解，现在正尝试构造下一个短语。令 e' 为 τ^2 的下一个倍数，则有 $e' - e < \tau^2 \le \Delta$，且 $T[e'..e' + \tau)$ 是一个稠密片段。因此，根据性质 1，我们有 $e' \in S$。

第一种情况是，$T[e'..y]$ 与从更早位置开始的任意后缀之间的最长公共前缀短于 τ。此时，我们只需将新短语的头部设为 $T[e..e' + \tau)$，并将尾部设为空。否则，根据上述预处理，我们知道这个最长前缀的长度 ℓ。我们将新短语的头部设为 $T[e..e')$，尾部设为 $T[e'..e'' + \ell)$。每个短语只需常数时间，而每个短语的长度至少为 τ，由此得到所述的总体时间复杂度。还需证明每一步的正确性。

令 $T[e..s]$ 为从位置 e 开始的最长 LZ 短语。要证明我们得到一个有效的 $(\Delta + \tau)$-近似短语，只需证明 $s \le e' + \max(\tau, \ell)$。设 $T[e..s)$ 的前一次出现位于位置 $p < e$。若 $s - e' < \tau$，则无需证明。否则，$T[e'..s)$ 是一个长度至少为 τ 的字符串，并且它也从更早的位置 $p + e' - e < e'$ 开始出现。因此，我们会正确地确定 $\ell \ge \tau$，并找到一个使 ℓ 的值最大的字符串的前一次出现。特别地，我们将得到所需的 $\ell \ge s - e'$。 □

为了达到定理 1.3 的界，我们现在按照第 3.6 节中的做法进行，不同之处在于，我们使用引理 13 的地方改用引理 15。对于每个 $T[x..y]$，当 $m = |T[x..y]|$ 时，这需要 $\mathcal{O}(m\tilde{\sigma} \log m/\sqrt{\Delta})$ 时间，此外还需要计算每个块的最长前缀和后缀所用的时间（后者也包括构建后缀树 ST 以及其他步骤所用的时间；在上述推理中，这些步骤的时间估计为$\mathcal{O}(|T'|)$），另外还需加上在 $\mathcal{T}_{\text{phrase}}$ 中插入 $T[i_j + \tau..i_{j+1} + \tau)$ 的时间，当 $i_{j+1} \ge i_j + \tau$。

我们注意到，可以停用完全包含在 $T[i \cdot \tau..i \cdot \tau + \ell_i)$ 和$T[(i+1) \cdot \tau - r_i..(i+1) \cdot \tau)$ 中的任意块对，因为我们已经检查过这些片段是无平方子串。此外，对于某个 $j' < j$，我们也可以停用完全包含在 $T[i_j + \tau..i_{j+1} + \tau)$ 中最长前缀里的任意块对，该前缀等于$T[i_{j'} + \tau..i_{j'+1} + \tau)$，因为这样的片段不可能包含平方的最左出现位置。

共有 $\mathcal{O}(m/\sqrt{\Delta})$ 个块和长短语。如果某个块或长短语使总时间增加了 $x = \Omega(\sqrt[4]{\Delta})$，那么我们就显式停用阶段 $t + 3$ 中完全包含在相应片段内的块对。阶段 $t + 3$ 中的块对长度为 $\mathcal{O}(\sqrt[4]{\Delta})$，因此我们停用了 $\Omega(x)$ 个位置。因此，所有阶段中此类块和长短语所花费的时间总和为 $\mathcal{O}(n)$。其余块和长短语使总时间增加了 $\mathcal{O}(\sqrt[4]{\Delta})$，共有 $\mathcal{O}(m/\sqrt{\Delta})$ 个，合计为 $\mathcal{O}(m/\sqrt[4]{\Delta})$。每个阶段中，这都是$\mathcal{O}(n/\sqrt[4]{\Delta})$，因此根据推论 3，总体为 $\mathcal{O}(n)$。

5 计算 runs

现在我们调整算法，使其能够计算所有 runs。我们从第 3.2 节和 第 4 节中的算法开始，但不采用第 4.2 节中的最终改进。首先要注意，Δ-近似 LZ 分解的关键性质，尤其是引理 6 和 7，对于 runs 的计算同样成立。下面的引理对此进行了说明。

引理 16. 设 $b_1 b_2 \dots b_z$ 是字符串 T 的 Δ-近似 *LZ* 分解。对于每个长度为 $e - s + 1 \ge 8\Delta$ 的 *run* $\langle s, e, p \rangle$，至少存在一个短语 b_i，其尾部为 $|\mathit{tail}(b_i)| \ge \frac{e-s+1}{8} \ge \Delta$，使得 $\mathit{tail}(b_i)$ 与该 *run* 的右侧 $T[s + \lceil \frac{e-s+1}{2} \rceil ..e]$ 相交。

证明. 设 $\ell = \frac{e-s+1}{2}$，并注意到 $\frac{\ell}{4} \geq \Delta$ 和 $e = s + 2\ell - 1$。假设所有与 $T[s + \lceil \ell \rceil .. e]$ 相交的尾部长度都小于 $\frac{\ell}{4}$，那么这些尾部对应的短语长度至多为 $\frac{\ell}{4} + \Delta - 1 \leq \frac{\ell}{2} - 1$（因为每个头部的长度都小于 Δ）。这意味着 $T[s + \lceil \ell \rceil .. e]$（长度为 $\lfloor \ell \rfloor$）至少与 $\left\lceil \lfloor \ell \rfloor / (\frac{\ell}{2} - 1) \right\rceil \geq 3$ 个短语相交（该不等式对 $\ell \geq 4$ 成立，而这由 $\Delta \geq 1$ 可知）。因此，存在某个短语 $b_i = T[x..y]$ 完全包含于 $T[s + \lceil \ell \rceil .. e]$ 中，形式化地说，$s + \lceil \ell \rceil < x \leq y < e$。然而，这与Δ-近似 LZ 分解的定义矛盾，因为 $T[x..e + 1]$ 是标准 LZ 短语的前缀（由 $T[x..e] = T[x - p..e - p]$ 可知）。这一矛盾意味着 $T[s + \lceil \ell \rceil .. e]$ 与一个长度至少为 $\frac{\ell}{4}$ 的尾部相交。

□

在说明如何通过算法应用引理 16之前，我们需要解释引理 1 如何扩展到计算 runs，以及这如何表明 Main 和 Lorentz 的方法 [61] 可以轻松扩展到计算所有 runs。我们并不声称这是新结果，但原论文只讨论了如何找到所有平方的一种表示，而我们需要找到 runs，因此为完整起见，补充说明如下。

引理 17。 给定一般字母表上的两个字符串 x 和 y，我们可以在 xy 中计算出所有包含 x 的最后一个字符或 y 的第一个字符的 *runs*，所需时间和比较次数为 $\mathcal{O}(|x| + |y|)$。

证明。 考虑 $t = xy$ 中的一个 run $\langle s, e, p \rangle$，它包含 x 的最后一个字符或 y 的第一个字符，也就是说，$s \leq |x| + 1$ 且 $e \geq |x|$。令 $\ell = \lfloor \frac{e-s+1}{2} \rfloor \geq p$。我们分别计算满足 $s + \ell \leq |x| + 1$ 和 $s + \ell > |x| + 1$ 的所有 runs。下面描述前一种情况，后一种情况与之对称。

由于 $s + \ell \leq |x| + 1$，长度为 p 的子串 $x[|x| - p + 1.. |x|]$ 完全位于 run 内。这提示我们可以采用以下策略来生成所有满足 $s + \ell \leq |x| + 1$ 的 runs。我们遍历 $p = 1$ 的所有可能取值，2，……，$|x|$。对于给定的 p，我们计算 $x[|x| - p + 1.. |x|]y$ 和 y 的最长公共前缀长度，记为 pref，以及 $x[1.. |x| - p]$ 和 x 的最长公共后缀长度，记为 suf。显然，$t[|x| - p + 1 - \text{suf}.. |x| + \text{pref}]$ 是一个长度最大的 p-周期子串，其长度为 $\ell' = p + \text{suf} + \text{pref}$。如果 pref + suf $\geq p$ 且 $s + \lfloor \ell'/2 \rfloor \leq |x| + 1$，那么我们就将该子串报告为一个 run。（后一个条件保证每个 run 恰好只会在两个对称情形之一中被报告。）

我们使用前缀表来计算最长公共前缀。对于给定的字符串，该表在位置 i 处记录从位置 i 开始、同时也是该字符串前缀的最长子串的长度。为了计算 pref 的值，我们使用 $y\$xy$ 的前缀表（其中 \$ 是一个新字符，与 x 和 y 中的任何字符都不匹配）。类似地，为了计算 suf 的值，我们使用新字符串 $x\$x$ 反转后的前缀表。该表可以在 $\mathcal{O}(|x| + |y|)$ 时间内通过若干次比较计算得出（例如，参见 [61] 中表 *lppattern* 的计算）。随后，可以在常数时间内检查 p 的每个值。

□

引理 18. 在一般无序字母表上计算长度为 n 的字符串的所有 *runs*，可以在 $\mathcal{O}(n \log n)$ 时间内通过若干次比较实现。

证明。 设输入字符串为 $T[1..n]$。我们采用分治。设 $x = T[1..\lfloor n/2 \rfloor]$ and $y = T[\lfloor n/2 \rfloor + 1..n]$。首先，我们递归地计算 x 和 y 中的所有 runs。在报告的 runs 中，我们筛除所有包含 x 的最后一个字符或 y 的第一个字符的 run，这需要 $\mathcal{O}(|x| + |y|)$ 时间。这样，如果某个报告的 run 是相对于 x（或 y）而言的 run，但不是相对于 xy 而言的 run，那么它就会被筛除。我们已经生成了除包含 x 的最后一个字符或 y 的第一个字符（或两者皆有）的 run 之外的所有 runs。因此，我们只需调用

引理 17 关于 xy，它将恰好输出 $\mathcal{O}(|x|+|y|$ 中缺失的 runs，所需时间和比较次数为 $\mathcal{O}(\log n)$ 层递归，每一层总共需要 $\mathcal{O}(n)$ 时间和比较次数。 □

引理 19. 设 $T = b_1 b_2 \dots b_z$ 是一个 Δ-近似 *LZ* 分解，且 $\chi = \sum_{|\text{尾部}(b_i)|\geq\Delta} |\text{尾部 } (b_i)|$。我们可以在 $\mathcal{O}(\chi + z)$ 时间内，通过 $\mathcal{O}(\chi)$ 次比较，计算出一个大小为 $\mathcal{O}(\chi)$ 的多重集 R，其中包含若干 *run*，并满足以下性质：只有当 $e - s + 1 < 8\Delta$ 或存在某个尾部 $\textit{tail}(b_i) = T[a_2..a_3]$，使得 $a_2 < s$ 且 $e < a_3$ 时，*run* $T[s..e]$ 才可能不属于 R。

证明。设 $n = |T|$。我们分别考虑每个短语 $b_i = T[a_1..a_3]$，其中 head$(b_i) = T[a_1..a_2 - 1]$ 且 tail$(b_i) = T[a_2..a_3]$。设 $k = |\text{tail}(b_i)|$。若 $k \geq \Delta$，则我们对 $x_1 = T[a_2 - 8k..a_2 - 1]$ 和 $y_1 = T[a_2..a_3 + 4k]$，以及修剪至 $T[1..n]$ 的 $x_2 = T[a_2 - 8k..a_3 - 1]$ 和 $y_2 = T[a_3..a_3 + 4k]$，应用引理 17。这需要 $\mathcal{O}(|\text{tail}(b_i)|)$ 时间和若干次比较，并报告相对于 $x_1 y_1 = x_2 y_2 = T[a_2 - 8k..a_3 + 4k]$（修剪至 $T[1..n]$）的 $\mathcal{O}(|\text{tail}(b_i)|)$ 个 run。在这些 run 中，我们筛除包含以下任一位置的 run：$a_2 - 8k$（仅当 $a_2 - 8k > 1$ 时）和 $a_3 + 4k$（仅当 $a_3 + 4k < n$ 时），这需要 $\mathcal{O}(|\text{tail}(b_i)|)$ 时间。这样，每个报告的 run 不仅是相对于 $x_1 y_1$ 的 run，也是相对于 T 的 run。总计，我们报告 $\mathcal{O}(\chi)$ 个 run（可能包含重复项），并在应用引理 17 时花费 $\mathcal{O}(\chi)$ 时间和若干次比较。对于每个短语，还需要额外花费 $\mathcal{O}(z)$ 时间来检查 $|\text{tail}(b_i)| \geq \Delta$。

现在我们证明，所述策略能够计算出所有长度至少为 8Δ 的 runs，但不包括那些被完全包含在尾部中的 runs。设 $\langle s, e, p\rangle$ 是一个长度为 2ℓ 的 run，其中 $\ell \geq 4\Delta$ 是 1 2 的倍数。根据引理 16，该 run 的右侧 $T[s + \lceil \ell \rceil ..e]$ 与某个长度为 $k = |$ 的尾部 tail$(b_i) = T[a_2..a_3]$ 相交，$\text{tail}(b_i)| \geq \frac{\ell}{4} \geq \Delta$。由于这处交集，我们有 $a_2 \leq e$ 和 $a_3 \geq s + \lceil \ell \rceil$。因此，在处理 b_i 并应用引理 17 时，x_1 和 x_2 的起始位置满足 $a_2 - 8k \leq e - 8\frac{\ell}{4} < s$，而 y_1 和 y_2 的结束位置满足 $a_3 + 4k \geq s + \lceil \ell \rceil + 4\frac{\ell}{4} > e$。所以，该 run 被包含在片段 $T[a_2 - 8k..a_3 + 4k]$（截短至 $T[1..n]$）中，该片段对应于 $x_1 y_1$ 和 $x_2 y_2$；并且该 run 不包含位置 $a_2 - 8k$ 和 $a_3 + 4k$。如果 $s \leq a_2 \leq e$，那么在对 x_1 和 y_1 应用引理 17 时，我们会找到该 run。如果 $s \leq a_3 \leq e$，那么在对 x_2 和 y_2 应用引理 17 时，我们会找到该 run。否则，$T[s..e]$ 完全包含在 $T[a_2 + 1..a_3 - 1]$ 中，我们无需报告该 run。 □

现在我们描述如何使用 $\mathcal{O}(n \log \sigma)$ 次比较和 $\mathcal{O}(n \log \sigma + n \log^* n)$ 时间计算所有 runs。我们再次使用序列 $\sigma_t = 2^{2^{\lceil \log \log n \rceil - t}}$，其中 $t = 0$、1、……、$\lceil \log \log n \rceil$。我们观察到$\sigma_{t-1} = (\sigma_t)^2$，，并按照 t 的取值分阶段进行。在第 t^{th} 阶段，我们的目标是计算长度至少为 σ_t 且小于 $(\sigma_t)^2$ 的 runs。需要强调的是，这个条件取决于 run 的长度，而不是其周期。我们将整个 $T[1..n]$ 划分为长度为 $(\sigma_t)^2$，的块，并将第 k^{th} 个块记为 B_k。长度小于 $(\sigma_t)^2$ 的 run 完全包含在某两个相邻块内 $B_i B_{i+1}$，而且总存在一对相邻块，使得该 run 既不包含这对块中的第一个位置，也不包含最后一个位置（除非第一个位置分别是 $T[1]$，或最后一个位置是 $T[n]$）。因此，我们依次考察每一对 $B_1 B_2$、$B_2 B_3$ 等。我们首先用 $\Delta = \sigma_t/8$ 和 $\tilde{\sigma} = (\sigma_t)^{1/4}/\log(\sigma_t)$ 应用引理 13，以求得 $T[1..n]$ 中对应片段的一个 $(\sigma_t/8)$-近似 LZ 分解；随后使用引理 19 计算所有长度至少为 σ_t 的 runs，但可能有些 run 完全包含在尾部中。对于计算出的 runs，我们舍弃包含块对第一个或最后一个位置的那些（除非第一个位置分别是 $T[1]$，或最后一个位置是 $T[n]$）。这样一来，报告的每个 run 不仅是相对于该块对的 run，也是相对于整个 $T[1..n]$ 的 run。如果通过这种方式没有报告某个长度至少为 σ_t 且小于 $(\sigma_t)^2$ 的 run，那么它必定完全包含在某个尾部中。

我们无法总是负担得起对所有块对应用引理 13 和 19。因此，我们必须停用某些块。在当前阶段 t，对于每个长度至少为 $T[s..e]$ 的尾部 Δ，

我们停用阶段 $t+3$ 中所有包含在 $T[s+1..e-1]$内的块对。根据第 3.2 节中的类似推理，如果某个尾部在应用引理 19 时产生了 $e-s+1$ 次比较并耗费了相应时间，那么它会永久停用字符串中的 $\Omega(e-s+1)$ 个位置，因此，调用引理 13 和引理 19 所需的总时间和比较次数上界为 $\mathcal{O}(n)$（此外，引理 13 还需要额外的总时间 $\mathcal{O}(n\log^* n)$）。每当我们应用引理 13 时，都会将所有长度至少为 Δ 的尾部加入列表 $\mathcal{L}$，并为每个尾部标注其前一次出现的位置。算法终止后，$\mathcal{L}$ 包含所有阶段中的足够长的尾部。我们已经证明，引理 19 所需的总时间上界为 $\mathcal{O}(n)$，因此，$\mathcal{L}$ 中尾部的总长度至多为 $\mathcal{O}(n)$。

如果当前阶段中对引理 13 的任何一次调用检测到 $\sigma > \tilde{\sigma}$，或者如果 $\tilde{\sigma} < 256$，我们就立即改为对当前阶段中每一对 B_iB_{i+1} 块应用引理 18，这需要 $\mathcal{O}(n\log\ \sigma)$ 时间（因为块对的长度关于 $\tilde{\sigma}$ 呈多项式关系）。同样，在对 B_iB_{i+1}应用引理 13 后，我们会丢弃所有包含 B_iB_{i+1}的第一个或最后一个位置的 runs（除非第一个位置分别是 $T[1]$ 或最后一个位置是 $T[n]$）。该过程结束后，我们已计算出所有 runs，但可能还遗漏了列表 $\mathcal{L}$尾部中被完全包含的某些 runs。我们可能报告了一些重复周期运行，接下来将其过滤掉。截至目前报告的 runs 数量为 $r = \mathcal{O}(n\log\ \sigma)$[4]。我们再用 $= \mathcal{O}(n\log\ \sigma)$ 时间对它们进行排序，例如使用基数排序，并删除重复项；这额外需要 $\mathcal{O}(n+r)$ 时间。目前为止的运行时间为 $\mathcal{O}(n\log\ \sigma)$。

5.1 从先前出现位置复制周期运行

最后，我们必须计算那些完全包含在 $\mathcal{L}$ 中某个尾部里的 run。考虑这样的一个 run $\langle r_s, r_e, p\rangle$，并设 $\mathcal{L}$ 中有一个尾部 $T[s..e]$，满足 $s < r_s$ 和 $r_e < e$。如果有多个尾部符合这一条件，则令 $T[s..e]$ 为其中使 e 最大的那个。在 $\mathcal{L}$ 中，我们为 $T[s..e]$ 标注了其前一次出现 $T[s-d..e-d]$。注意，$\langle r_s - d, r_e - d, p\rangle$ 也是一个 run。因此，如果按适当的顺序计算这些 runs，我们就可以直接从各自的前一次出现位置复制缺失的 runs。为此，我们为每个位置 $i \in [1, n]$ 标注：

- 一个由所有已计算出的 runs $\langle i, e, p\rangle$ 构成的列表，按结束位置 e 递增排序。我们已经为消除重复项对这些 runs 进行了排序，并且可以在 $\mathcal{O}(n)$ 时间内为所有位置添加标注。

- 一对 (e^*, d^*)，其中 $e^* = d^* = 0$ 如果不存在满足 $s < i < e$ 的尾部 $T[s..e]$。否则，在所有满足 $s < i < e$ 的尾部 $T[s..e]$ 中，我们选择使 e 最大的那个。令 $T[s-d..e-d]$ 为它的前一次出现，则使用 $e^* = e$ 和 $d^* = d$。如前所述，$\mathcal{L}$ 中所有尾部的总长度为 $\mathcal{O}(n)$，因此我们只需扫描每个尾部，并在必要时更新其中每个位置的注释对。

注意，如果某个位置标注为 $(0, 0)$，那么没有任何从位置 i开始的 run 完全包含在某个尾部中，因此我们已经用从 i开始的 runs 的完整列表标注了位置 i。现在，我们按递增顺序逐一处理位置 $i \in [1, n]$。根据归纳假设，在处理 i时，我们已经用从 j开始的 runs 的完整列表标注了每个 $j < i$。因此，我们的目标是补全 i 的列表，使其包含所有从 i开始的 runs。如果 i 标注为 $(0, 0)$，那么该列表已经完整。否则，i 标注为 (e, d)，每个缺失的 run $\langle i, e_r, p\rangle$ 都满足 $e_r < e$，而 $i-d$的标注列表已经包含 run $\langle i-d, e_r-d, p\rangle$（这是由于 $T[i-1..e_r+1] = T[i-d-1..e_r-d+1]$

[4] 更仔细的分析会发现，它是 $\mathcal{O}(n)$，但这对证明并非必要

以及归纳假设）。对于位置 $i-d$的标注列表中的每个 run $\langle i-d, r_e-d, p\rangle$，我们都将 run $\langle i, e_r, p\rangle$ 插入 i的标注列表。我们采用归并的方式执行这一步，从两个列表中最短的 runs 开始，将它们交错合并。当我们即将插入一个满足 $e_r \geq e$的 run $\langle i, e_r, p\rangle$ 时，就不再插入它并中止。因此，处理 i 所需的时间与从位置 i开始的 runs 数量呈线性关系。根据 runs 定理 [8], ，runs 的总数小于 n，因此这一步的总时间为 $\mathcal{O}(n)$。

除了第 5.1 节中的新步骤外，复杂度分析的过程与第 3.2 节完全相同。因此，我们证明了：

定理 5.1。 对于包含来自一般无序字母表的 σ 个不同符号的长度为 n 的字符串，计算所有 *run* 可通过 $\mathcal{O}(n \log \sigma)$ 次比较和 $\mathcal{O}(n \log \sigma + n \log^* n)$时间实现。

5.2 计算 runs 的最终改进

现在的目标是调整最终算法以检测所有 run。我们不能再一检测到平方就停止，也不能简单地停用较早出现的块对。然而，定理 2.2 实际上能够在 $\mathcal{O}(\ell_i + r_i)$ 时间内报告 $T[i\cdot\tau..i\cdot\tau+\ell_i)$ 和 $T[(i+1)\cdot\tau-r_i..(i+1)\cdot\tau)$ 中的所有 run，而且即使这些片段不是平方自由的，我们也不必终止算法。因此，我们确实可以停用完全包含在 $T[i\cdot\tau..i\cdot\tau+\ell_i)$ 和$T[(i+1)\cdot\tau-r_i..(i+1)\cdot\tau)$ 中的任何块对。接下来，我们还会停用完全包含在 $T[i_j+\tau..i_{j+1}+\tau)$ 的最长前缀中的块对，该前缀等于 $T[i_{j'}+\tau..i_{j'+1}+\tau)$，其中 $j'<j$。将此前缀的长度记为 ℓ，我们将 $T[i_j+\tau..i_j+\ell)$ 视为尾部，并将其添加到列表 $\mathcal{L}$(并用 $i_{j'}$ 标注)。添加到 $\mathcal{L}$ 的所有片段的总长度仍为 $\mathcal{O}(n)$。

定理 5.2。 在一般无序字母表中，计算一个包含 σ 个不同符号、长度为 n 的字符串中的所有 *runs*，可通过 $\mathcal{O}(n \log \sigma)$ 次比较和 $\mathcal{O}(n\log \sigma)$ 时间实现。

参考文献

[1] J. Allouche 和 J. O. Shallit。《无处不在的普鲁厄-图埃-莫尔斯序列》。载于 C. Ding、T. Helleseth 和 H. Niederreiter 编，《序列及其应用——*SETA 1998* 论文集，新加坡，*1998*年*12*月*14*日至*17*日》，《离散数学与理论计算机科学》，第 1–16 页。施普林格，1998年。[2] A. Amir、E. Eisenberg 和 A. Levy。近似周期性。载于 O. Cheong、K. Chwa 和 K. Park 编，《算法与计算——第*21*届国际研讨会 *ISAAC 2010*，济州岛，韩国，*2010*年*12*月*15*日至*17*日，论文集，第一部分》，《计算机科学讲义》第 6506 卷，第 25–36 页。施普林格，2010年。[3] A. Amir、E. Eisenberg 和 A. Levy。近似周期性。《信息与计算》，241:215–226，2015年。[4] A. Amir 和 A. Levy。近似周期检测与校正。载于 L. Calder´on-Benavides、C. N. Gonz´ález-Caro、E. Ch´avez 和 N. Ziviani 编，《字符串处理与信息检索——第*19*届国际研讨会 *SPIRE 2012*，卡塔赫纳-德印第亚斯，哥伦比亚，*2012*年*10*月*21*日至*25*日，论文集》，《计算机科学讲义》第 7608 卷，第 1–15 页。施普林格，2012年。

[5] A. Apostolico 和 D. Breslauer。一种用于检测字符串中所有平方的最优 $\mathcal{O}(\log \log n)$ 时间并行算法。*SIAM J. Comput.*，25(6):1318–1331，1996年。[6] A. Apostolico 和 R. Giancarlo。参数化字符串中的周期性与重复。*Discret.Appl. Math.*，156(9):1389–1398，2008年。[7] A. Apostolico 和 F. P. Preparata。字符串中重复的最优离线检测。*Theor.Comput. Sci.*，22:297–315，1983年。[8] H. Bannai、T. I、S. Inenaga、Y. Nakashima、M. Takeda 和 K. Tsuruta。“runs”定理。*SIAM J. Comput.*，46(5):1501–1514，2017年。[9] D. R. Bean、A. Ehrenfeucht 和 G. F. McNulty。符号串中的可避免模式。*Pacific Journal of Mathematics*，85(2):261–294，1979年。[10] M. A. Bender 和 M. Farach-Colton。重新探讨最近公共祖先问题。载于 G. H. Gonnet、D. Panario 和 A. Viola 编，《*LATIN 2000*：理论信息学，第四届拉丁美洲研讨会，乌拉圭埃斯特角城，*2000*年*4*月*10*日至*14*日，论文集》，《计算机科学讲义》第 1776 卷，第 88–94 页。施普林格，2000年。[11] J. Berstel 和 L. Boasson。部分词与 Fine-Wilf 定理。*Theor. Comput.Sci.*，218(1):135–141，1999年。[12] F. Blanchet-Sadri、D. Bal 和 G. Sisodia。图连通性、部分词与 Fine-Wilf 定理。《信息与计算》，206(5):676–693，2008年。[13] F. Blanchet-Sadri 和 R. A. Hegstrom。重新探讨部分词与 Fine-Wilf 定理。*Theor. Comput. Sci.*，270(1-2):401–419，2002年。[14] F. Blanchet-Sadri、S. Simmons、A. Tebbe 和 A. Veprauskas。阿贝尔周期、部分词及 Fine-Wilf 定理的推广。《*RAIRO*理论信息学与应用》，47(3):215–234，2013年。[15] D. Breslauer。高效字符串算法。博士论文，哥伦比亚大学，1992年。[16] S. Brlek 和 S. Li。有限词中平方的数量。*CoRR*，abs/2204.10204，2022年。[17] M. G. Castelli、F. Mignosi 和 A. Restivo。关于三个周期的 Fine-Wilf 定理及 Sturmian 词的推广。*Theor. Comput. Sci.*，218(1):83–94，1999年。[18] R. Cole 和 R. Hariharan。字符串匹配精确复杂度的更紧上界。*SIAM J. Comput.*，26(3):803–856，1997年。[19] R. Cole、R. Hariharan、M. Paterson 和 U. Zwick。字符串匹配精确复杂度的更紧下界。*SIAM J. Co mput.*，24(1):30–45，1995年。[20] S. Constantinescu 和 L. Ilie。阿贝尔周期的 Fine-Wilf 定理。*EATCS* 通报，89:167–170，2006年1月。[21] T. M. Cover 和 J. A. Thomas。信息论基础。威利，第二版，2006年。[22] M. Crochemore。一种计算词中重复的最优算法。*Inf. Process.Lett.*，12(5):244–250，1981年。

[23] M. Crochemore. 变换器与重复。*Theor. Comput. Sci.*, 45(1):63–86, 1986年。[24] M. Crochemore and L. Ilie. 字符串中的极大重复。*J. Comput. Syst. Sci.*, 74(5):796– 807, 2008年。[25]M. Crochemore, L. Ilie, and L. Tinta."runs"猜想。*Theor. Comput. Sci.*, 412(27):2931–2941, 2011年。[26] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, R. Kundu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen. 通过非交叉最长公共扩展查询，在一般字母表上近乎最优地计算runs。载于 S. Inenaga, K. Sadakane, and T. Sakai 编，字符串处理与信息检索——第*23*届国际研讨会，*SPIRE 2016*，日本别府，*2016*年*10*月*18*日至*20*日，论文集，计算机科学讲义第9954卷，第22–34页，2016年。[27] J. D. Currie. 模式避免：主题与变奏。*Theor. Comput. Sci.*, 339(1):7–18, 2005年。[28] A. Deza, F. Franek, and A. Thierry. 一个字符串中能包含多少个双平方？*Discret.Appl. Math.*, 180:52–69, 2015年。[29] J. Duval, T. Lecroq, and A. Lefebvre. 在无序字母表上以线性时间计算无边框共轭词。*Theor. Comput. Sci.*, 522:77–84, 2014年。[30] J. Ellert and J. Fischer. 一般有序字母表上的线性时间runs。载于 N. Bansal, E. Merelli, and J. Worrell 编，第*48*届自动机、语言与编程国际研讨会（*ICALP 2021*），第198卷，莱布尼茨国际信息学论文集（*LIPIcs*），第63:1–63:16页，德国达格施图尔，2021年。施洛斯·达格施图尔——莱布尼茨信息学中心。¨ur Informatik。[31] M. Farach. 大型字母表上的最优后缀树构造。载于第*38*届计算机科学基础年度研讨会，*FOCS '97*，美国佛罗里达州迈阿密海滩，*1997*年*10*月*19*日至*22*日，第137–143页。IEEE计算机学会，1997年。[32] N. J. Fine and H. S. Wilf. 周期函数的唯一性定理。载于美国数学学会会刊，第16卷，第109–114页，1965年。[33] J. Fischer, S. Holub, T. I, and M. Lewenstein. 超越runs定理。载于 C. S. Iliopoulos, S. J. Puglisi, and E. Yilmaz 编，字符串处理与信息检索——第*22*届国际研讨会，*SPIRE 2015*，英国伦敦，*2015*年*9*月*1*日至*4*日，论文集，计算机科学讲义第9309卷，第277–286页。施普林格，2015年。[34] J. Fischer, T. I, D. K¨oppl, and K. Sadakane. 借助空间高效后缀树进行 Lempel-Ziv 分解。*Algorithmica*, 80(7):2048–2081, 2018年。[35] A. S. Fraenkel and J. Simpson. 一个字符串中能包含多少个平方？*J. Comb. Theory,Ser. A*, 82(1):112–120, 1998年。[36] F. Franek and Q. Yang. 字符串中runs最大数量的渐近下界。*Int. J. Found. Comput. Sci.*, 19(1):195–203, 2008年。

[37] H. N. Gabow. 带链接操作的加权匹配与最近公共祖先数据结构。载于 D. S. Johnson 编，第一届*ACM-SIAM*离散算法研讨会论文集，*1990*年*1*月*22*日至*24*日，美国加利福尼亚州旧金山，第434–443页。SIAM，1990年。[38]Z. Galil and J. I. Seiferas. 时间与空间最优字符串匹配。*J. Comput. Syst. Sci.*, 26(3):280–294, 1983年。[39] P. Gawrychowski, T. Kociumaka, W. Rytter, and T. Walen. 一般字母表字符串中更快的最长公共扩展查询。载于 R. Grossi and M. Lewenstein 编，第*27*届组合模式匹配年度研讨会，*CPM 2016*，以色列特拉维夫，*2016*年*6*月*27*日至*29*日，LIPIcs第54卷，第5:1–5:13页。施洛斯·达格施图尔——莱布尼茨信息学中心，2016年。[40] M. Giraud. 字符串中的runs并没有那么多。载于 C. Mart´ín-Vide, F. Otto, and H. Fernau 编，语言与自动机理论及应用，第二届国际会议，*LATA 2008*，西班牙塔拉戈纳，*2008*年*3*月*13*日至*19*日。修订论文，计算机科学讲义第5196卷，第232–239页。施普林格，2008年。[41]M. Giraud. runs与微型极大周期重复数量的渐近行为。《信息与计算》，207(11):1221–1228, 2009年。[42] G. Gourdel, T. Kociumaka, J. Radoszewski, W. Rytter, A. M. Shur, and T. Walen. 保序模型中的字符串周期。《信息与计算》, 270, 2020年。[43] D. Gusfield.《字符串、树和序列上的算法——计算机科学与计算生物学》。剑桥大学出版社，1997年。[44] Y. Han and M. Thorup. 在期望 O($n\sqrt{\log \log n}$) 时间和线性空间内进行整数排序。载于*FOCS*，第135–144页。IEEE计算机学会，2002年。[45] S. Holub. 丢失位置的前缀频率。*Theor. Comput. Sci.*, 684:43–52, 2017年。[46] J. Hong and G. Chen. 高效的在线重复检测。*Theor. Comput. Sci.*, 407(1- 3):554–563, 2008年。[47] L. A. Idiatulina and A. M. Shur. 周期性部分词与随机二分图。《信息学基础》, 132(1):15–31, 2014年。[48] L. Ilie. 关于一个词中平方数量的简短说明。*Theor. Comput. Sci.*, 380(3):373–376, 2007年。[49]J. Justin. 关于 Castelli、Mignosi、Restivo 的一篇论文。《*RAIRO*理论信息学与应用》, 34(5):373–377, 2000年。[50] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen. 部分词的周期性引理。《信息与计算》，283:104677, 2022年。[51] R. M. Kolpakov, G. Bana, and G. Kucherov. mreps：高效灵活地检测 DNA 中的串联重复。《核酸研究》, 31(13):3672–3678, 2003年。[52] R. M. Kolpakov and G. Kucherov. 线性时间内查找一个词中的极大重复。载于第*40*届计算机科学基础年度研讨会，*FOCS '99*，*1999*年*10*月*17*日至*18*日，美国纽约州纽约，第596–604页。IEEE计算机学会，1999年。

[53] S. R. Kosaraju. 字符串中平方的计算（初步版本）。载于 M. Crochemore 和 D. Gusfield 编，组合模式匹配，第*5*届年度研讨会，*CPM 94*，美国加利福尼亚州阿西洛马，*1994*年*6*月*5*日至*8*日，论文集，计算机科学讲义第807卷，第146–150页。施普林格，1994年。[54] D. Kosolobov. 在线平方检测。*CoRR*，abs/1411.2022，2014年。[55] D. Kosolobov. Lempel-Ziv分解可能比计算所有runs更难。载于 E. W. Mayr 和 N. Ollinger 编，理论计算机科学专题第*32*届国际研讨会（*STACS 2015*），莱布尼茨国际信息学会议录第30卷，第582–593页，德国达格施图尔，2015年。Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik。[56] D. Kosolobov. 带回溯的在线重复检测。载于 F. Cicalese、E. Porat 和 U. Vaccaro 编，组合模式匹配——第*26*届年度研讨会，*CPM 2015*，意大利伊斯基亚岛，*2015*年*6*月*29*日至*7*月*1*日，论文集，计算机科学讲义第9133卷，第295–306页。施普林格，2015年。[57] D. Kosolobov. 一般字母表上的周期运行计算。*Inf. Process. Lett.*，116(3):241–244，2016年。[58] D. Kosolobov. 在无序字母表上寻找最左关键因子分解。*Theor.Comput. Sci.*，636:56–65，2016年。[59] S. Li、J. Pachocki 和 J. Radoszewski. 关于有限词中 k幂的最大数量的札记。*CoRR*，abs/2205.10156，2022年。[60] M. Maekawa. 一种用于去中心化系统中互斥的$\sqrt{N}$ 算法。*ACM Trans.Comput. Syst.*，3(2):145–159，1985年。[61] M. G. Main 和 R. J. Lorentz. 一种在字符串中查找所有重复的 $\mathcal{O}(n\log\ n)$算法。*J. Algorithms*，5(3):422–432，1984年。[62] W. Matsubara、K. Kusano、H. Bannai 和 A. Shinohara. 一系列周期运行丰富字符串。载于 A. Dediu、A. Ionescu 和 C. Mart´ın-Vide 编，语言与自动机理论及应用，第三届国际会议，*LATA 2009*，西班牙塔拉戈纳，*2009*年*4*月*2*日至*8*日，论文集，计算机科学讲义第5457卷，第578–587页。施普林格，2009年。[63] W. Matsubara、K. Kusano、A. Ishino、H. Bannai 和 A. Shinohara. 字符串中周期运行最大数量的新下界。载于 J. Holub 和 J. Zd´arek 编，*2008* 年布拉格字符串学会议论文集，捷克共和国布拉格，*2008* 年 *9* 月*1*日至*3*日，第140–145页。布拉格字符串学俱乐部、计算机科学与工程系、电气工程学院、布拉格捷克理工大学，2008年。[64] Y. Matsuoka、T. Aoki、S. Inenaga、H. Bannai 和 M. Takeda. 子串一致等价关系下的广义模式匹配与周期性。*Theor. Comput. Sci.*，656:225–233，2016年。[65] O. Merkurev 和 A. M. Shur. 流中的周期运行搜索。载于 N. R. Brisaboa 和 S. J. Puglisi 编，字符串处理与信息检索——第*26*届国际研讨会，*SPIRE 2019*，西班牙塞哥维亚，*2019*年*10*月*7*日至*9*日，论文集，计算机科学讲义第11811卷，第203–220页。施普林格，2019年。

[66] O. Merkurev 和 A. M. Shur. 流中的最大指数计算。*Algorithmica*，84(3):742–756，2022年。[67] S. J. Puglisi、J. Simpson 和 W. F. Smyth. 一个字符串中可以包含多少个runs? *Theor.Comput. Sci.*，401(1-3):165–171，2008年。[68] W. Rytter. 字符串中的runs数量：对线性上界的改进分析。载于 B. Durand 和 W. Thomas 编，*STACS 2006*，理论计算机科学专题第*23*届年度研讨会，法国马赛，*2006*年*2*月*23*日至*25*日，论文集，计算机科学讲义第3884卷，第184–195页。施普林格，2006年。[69] A. M. Shur 和 Y. V. Gamzova. 部分词与周期的交互性质。*Izvestiya: Mathematics*，68(2):405–428，2004年4月。[70] A. M. Shur 和 Y. V. Konovalova. 关于部分词的周期。载于 J. Sgall、A. Pultr 和 P. Kolman 编，*2001*年计算机科学数学基础，第*26*届国际研讨会，*MFCS 2001*，捷克共和国玛丽亚温泉市，*2001*年*8*月*27*日至*31*日，论文集，计算机科学讲义第2136卷，第657–665页。施普林格，2001年。[71] J. Simpson. 修正的帕多万字与词中周期运行的最大数量。*Australas.J Comb.*，46:129–146，2010年。[72] A. Thierry. 证明长度为 n 的词包含少于 $1.5n$ 个不同平方。*CoRR*，abs/2001.02996，2020年。.[73] A. Thue. 关于无限字符序列。*Norske Vid. Selsk. Skr., I Mat.–Nat. Kl.,* 克里斯蒂安尼亚，7:1–22，1906年。[74] R. Tijdeman 和 L. Zamboni. 适用于任意周期的 Fine-Wilf字。*Indagationes Mathemat-icae*，14(1):135–147，2003年。[75] J. Vuillemin. 统一视角下的数据结构。*Commun. ACM*，23(4):229–239，1980年。