arXiv:2603.05676v1 [cs.DS] 5 Mar 2026

如何在冰箱中排序：简单的熵敏感严格原地排序算法

Ofek Gila[0009−0005−5931−771X]、Michael T. Goodrich[0000−0002−8943−191X] 和 Vinesh Sridhar[0009−0009−3549−9589]

加州大学尔湾分校，美国{
ogila,goodrich,vineshs1}@uci.edu

摘要。虽然现代通用计算系统拥有充足的内存，但嵌入式计算机系统（例如冰箱中的系统）仍然受到内存限制；因此，这类嵌入式系统促使人们需要严格原地算法，即除输入所占用的内存之外，仅使用 $O(1)$ 的额外内存。本文首次提出了基于比较的排序算法，它们是严格原地的，并且其运行时间在输入数组的基于自然段的熵 $\mathcal{H}(A)$ 方面达到最优；该数组大小为 n。具体而言，我们描述了两种极其简单的范式，用于在 $O(n(1+\mathcal{H}(A)))$ 时间内实现严格原地的基于栈的自然归并排序算法。

1 引言

嵌入式计算机系统是为更大设备中的特定功能而设计的，由处理器和有限的内存组成。它们见于家用电器、车辆和医疗设备等多种应用。与通用计算机不同，嵌入式系统中有限的内存往往难以扩展；因此，嵌入式系统是严格原地算法的天然应用领域，这类算法除输入所占用的内存外，仅使用 $O(1)$的额外内存。此外，采用非易失性主存（非易失性主存储器）的嵌入式系统 [27]对严格原地算法有额外需求，因为这些存储器若被多次覆写就会退化，例如当必须维护栈之类的数据结构时。尽管如此，我们仍希望能够在嵌入式系统中实现快速算法，因为它们往往具有实时性能要求。

例如，在算法的超越最坏情况分析 $[22,23]$, 中，我们关注那些运行时间能够依据为问题实例所定义的、用以刻画预期输入典型性质的参数或度量而优于最坏情况界的算法。例如，假设给定一个数组 $A=[x_1,x_2,\ldots,x_n]$，其元素来自一个全序。如果我们令$\mathcal{R}=\{R_1,R_2,\ldots,R_{\rho(A)}\}$ 是将 A 划分为一组极大递增或递减运行段（即由连续元素构成的子序列）的划分，并令 r_i

表示 A 中第 i个运行段的大小 $|R_i|$（因此 $\sum_{i=1}^{\rho(A)} r_i = n$），则 A 的基于自然段的熵 $\mathcal{H}(A)$定义如下：[1]

$$\mathcal{H}(A) = \sum_{i=1}^{\rho(A)} (r_i/n) \log(n/r_i).$$

事实证明，研究人员已经证明，任何基于比较的排序算法的运行时间都存在下界 $\Omega(n(1 + \mathcal{H}(A)))$，并且有若干排序算法的运行时间为 $O(n(1 + \mathcal{H}(A)))$；参见例如 [1,3,4,12,17,20,24,25]。有趣的是，这些算法都是基于栈的自然归并排序算法：我们从运行段出发，并实现一种使用栈来决定合并哪些运行段的归并排序。事实上，其中两种算法 TimSort 和 PowerSort 已在实践中被广泛使用，例如作为 Python、Java 及其他主要编程语言中的默认排序算法。正如其描述所示，这些算法维护一个运行段栈，并根据运行段的长度和/或位置决定何时合并它们，以利用输入数据的预排序性。遗憾的是，这些算法都不是严格原地的，即除了用于输入数组的 n 个内存单元之外，它们还使用 $\omega(1)$ 个额外内存单元。事实上，仅其栈的大小就可达 $\Omega(\log n)$。因此，这些算法都不适用于嵌入式系统——在这类系统中，我们希望除输入之外只维护规模为 $O(1)$ 的信息量，而这正是本文的主题。另一方面，已知的严格原地排序算法（如堆排序）都不是熵敏感的。

先前相关工作。 现有的基于栈的自然归并排序算法通常会考察栈顶的 k 个运行段，并按照某种规则合并相邻的运行段（使用标准的有序列表合并算法），其中 $k \geq 2$ 是一个取决于算法的常数。Munro 和 Wild [20] 提出了 PeekSort 和 PowerSort，它们采用基于 2 的幂的规则，其运行时间达到实例最优，即 $O(n(1+\mathcal{H}(A)))$。Auger、Jug´e、Nicaud 和 Pivoteau [1,2] 研究了广受欢迎的 TimSort 算法 [21], ；该算法基于栈的合并规则大致以斐波那契数列为基础。他们证明，TimSort 也具有实例最优性。[2] 其他运行时间相同的基于栈的自然归并排序算法包括 Jug´e 的自适应 ShiversSort [17]、Gelling、Nebel、Smith 和 Wild [12] 的 Multiway PowerSort，以及 Buss 和 Knop 的 α-MergeSort [4] 。Takaoka 的 MinimalSort [25], 、Schou 和 Wang 的 PersiSort [24], ，以及 Barbay 和 Navarro 的自适应搜索树排序 [3] 采用了不同的方法，但仍实现了实例最优性。此外，TimSort 和 PowerSort 已成为许多流行软件库（包括 Python、Java 和 Swift 中的软件库）的标准排序算法 [12]。

实现基于栈的归并排序算法并使其严格原地，主要面临两个挑战。第一个挑战已被广泛研究，

[1] 除非另有说明，本文中所有对数均以2为底。[2] CPython 中 Python 的参考实现最近已将 TimSort 替换为 PowerSort [12]。

即如何原地合并两个有序子数组。幸运的是，已有方法可以严格原地执行线性时间归并；参见例如 [5–7,10,11,15, 16, 18, 26]。然而，即便有了原地合并方法，我们仍须克服实现严格原地的基于栈的归并排序的第二个挑战——栈本身 ——其深度可达 $\Omega(\log n)$。

我们的结果。 我们定义了一种简单的回溯算法，可将任何基于栈的归并排序算法实现为严格原地算法。在该算法中，我们只在栈上维护常数个运行段的长度，外加至多 $O(1)$个额外元数据。如果需要确定栈中更深处某个运行段的长度，只需沿数组向后遍历即可将其恢复。此外，如果用于合并运行段的算法是稳定的 [6, 7, 10, 15]（即经过构造，使相等元素在 A输入数组中的相对顺序保持不变），那么最终得到的原地自然归并排序也是稳定的。

如果对基于栈的自然归并排序算法 $\mathcal{S}$应用回溯算法后，其运行时间 $\mathcal{S}$ 仅增加一个常数因子，我们就称该算法为可回溯的 。诚然，这一定义掩盖了我们分析回溯算法时的一些重要细节。在第 3 节中，我们给出主要结果，证明 PowerSort [20] 是可回溯的。在附录 A 和 C 中，我们证明其他几种基于栈的归并排序也具有可回溯性，包括 c-Adaptive ShiversSort、ShiversSort、 α-StackSort 和 2-MergeSort [4, 17]。在 B 节中，我们给出一个否定结果，并证明著名的自适应排序算法 TimSort[3] [1, 2, 21] 和 α-MergeSort [4] 并非可回溯的。

这些否定性结果促使我们提出第二种简单的算法，称为跳回算法，它也可用于将几乎所有基于栈的归并排序实现为原地算法。在该算法中，我们只在栈上维护常数个运行段的长度，以及至多 $O(1)$ 个元数据，并使用简单的位编码方法，将运行段的长度原地编码在运行段本身中。如果需要确定栈中更深处某个运行段的长度，我们会在 $O(\log n)$ 时间内从其位编码中解码出长度，继而跳转到该运行段的起始位置（也可能重复这一查找过程，以定位前一个运行段）。在第 4 节中，我们证明，利用跳回算法，可以将任意基于栈的自然归并排序算法实现为原地算法，同时只会使其运行时间增加一个较小的常数因子；我们将这一性质称为可跳跃。将我们的主要贡献与关于基于栈的归并排序算法和原地归并算法运行时间的已知结果相结合，可推出首批严格原地排序算法；相对于基于自然段的熵，这些算法的运行时间是实例最优的。

我们用第 E 节中的实验来支持我们的理论结果。

[3] 我们证明，尽管存在一个漏洞，TimSort 的原始版本仍然是可回溯的 [14]。事实上，该漏洞涉及未能分配足够的内存来维护运行段栈。参见 C 节。

2 预备知识

让我们从基于栈的合并排序算法的一些预备知识开始。设A 是一个包含 n 个元素的输入数组。我们将运行段定义为 A中元素的非递减或严格递减子序列。所有递减运行段都可以在对 A 的初始线性时间扫描中轻松原地翻转，从而变为递增运行段，且不影响稳定性，因此，不失一般性，我们只考虑非递减运行段。

我们可以将所有基于栈的归并排序算法分为两个阶段。第一阶段是主循环，在此阶段，我们从左向右扫描 A ，并将新的运行段推入栈中。每种算法都定义了一种归并策略 $\mathcal{P}$，并以此区别于其他算法。每当我们将一个新的运行段推入栈中，且 $\mathcal{P}$ 的不变式遭到破坏时，就会根据 $\mathcal{P}$ 对栈中相邻的运行段执行一系列合并，直到不变式恢复。我们称之为合并序列。处理完 A 中的所有运行段后，主循环结束，我们会对栈中剩余的运行段执行收尾阶段：反复合并栈顶的两个运行段，直至只剩下一个运行段，即最终的有序列表。我们约定，越靠近栈顶的运行段，其标签索引越小，因此栈顶的运行段的标签为 R_1。我们用 r_i 表示运行段 R_i 的长度，即 $r_i = |R_i|$。算法 1 给出了通用的基于栈的归并排序伪代码。参见图 1。

算法 1 采用归并策略的通用基于栈的归并排序算法，$\mathcal{P}$

```
输入：大小为 n 的数组 A 。归并策略 P。
输出：已排序的数组 A′。
R ← A 的游程分解
S ← ∅                                                  ▷ the stack of runs
当 R非空时 执行                                          ▷ main loop
    将下一个运行段从 R 推入 S
    当 S 违反 P的不变式时执行                            ▷ algorithm-specific merge policy
        根据 P合并 S 中的一对相邻运行段
当 |S| > 1 时执行                                       ▷ collapse phase
    从 S 弹出 R1 和 R2 ，并将 MERGE(R1, R2)推入 S
返回弹出(S)
```

为便于表述，在提及某个基于栈的归并排序时，我们仍可能使用“栈”这一术语，尽管我们的算法并未显式维护 S的全部内容。我们将在全文中按需回顾特定基于栈的归并排序的合并策略。

2.1 浅层归并排序

有趣的是，大多数实例最优归并排序采用的归并策略只考虑栈顶的常数个运行段 k。Buss 和

Knop [4] 也有类似的观察，这促使他们提出了k-*aware* 归并排序的概念。如果基于栈的归并排序的 k-*aware* 合并条件基于栈顶 k 个运行段的运行段长度，并且只合并栈顶 k 个条目中的运行段，那么该排序就是 k-*aware* 的。就我们的目的而言，我们还可以允许合并条件基于运行段的其他元数据或属性，只要能在常数时间内获取，例如它们在数组中的位置；此外也可以基于其他全局数据，前提是所有这些信息都能放入 $O(1)$个内存单元中。我们将这种放宽后的定义称为*almost*-k-*aware*。任何 k-aware算法显然都是almost-k-aware的。

这些定义提出了一种非常简单的方法，可使几乎所有熵敏感算法实现原地操作：用一种“浅栈”替换我们的栈 S ，它只维护顶部的 k 个运行段。当然，随着运行段被合并，那些长度已被遗忘的其他运行段会进入 S 的顶部 k 个位置。我们必须确定这些被遗忘运行段的长度，以检验归并策略的不变式是否仍被违反。本文提出了两种简单且高效的算法来完成这一任务：回溯算法和跳回算法。它们分别在第 3 节和第 4 节中介绍。如果我们能够使用浅栈高效地实现某种基于栈的归并排序 $\mathcal{S}$，并使其原地运行，那么就称 $\mathcal{S}$ 为*浅层* 归并排序。我们在表 1 中总结了结果以及已知算法的相关性质。

3 回溯算法

本节定义回溯算法，并证明多种基于栈的归并排序都是可回溯的，也就是说，对它们应用回溯算法，增加的运行时间不会超过一个常数因子。回溯算法按如下方式实现浅栈：每当我们需要比较并确定某个运行段的长度时，就从栈中位于它上方的运行段的开头开始向后遍历。见图 2。

3.1 回溯过程与停止条件

我们会继续向前遍历，直到满足以下任一条件：(1) 确定所讨论运行段的长度，或 (2) 遍历足够远，能够确认给定的比较是否为假。我们

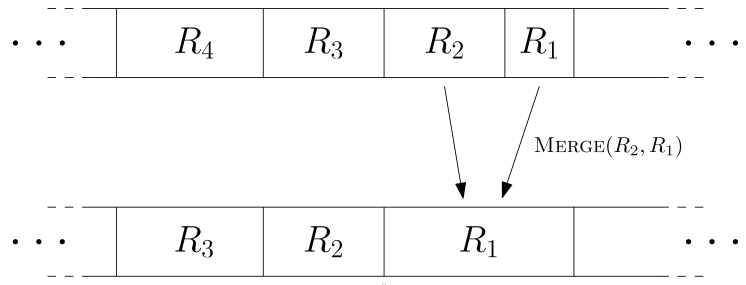

图 1：根据归并策略执行的归并步骤， $\mathcal{P}$。合并后的运行段采用较高运行段的标签。合并会降低位于这两个合并运行段下方的所有运行段的标签。

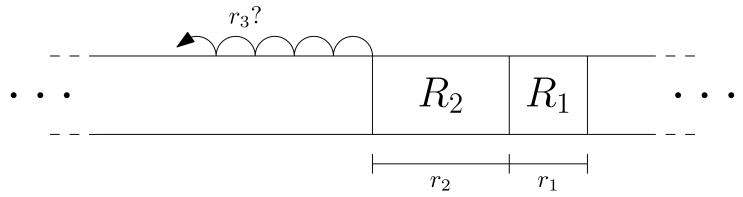

图 2：将回溯算法应用于几乎 3 感知归并排序。我们目前知道标记为 R_1 和 R_2 的运行段长度。数组中（即我们的运行段“栈”中） R_2 左侧可能有 $O(\log n)$ 个运行段，但我们不维护有关它们的任何信息。每当需要知道 R_3 的长度时，我们就从 R_2 的起点开始向后遍历。令人惊讶的是，在几种基于栈的归并排序中，这种策略仅会使运行时间增加一个常数因子。

将 (2) 称为我们的停止条件。它因算法而异。如果我们达到停止条件，就会忘记此前遍历了多远，并在下次调用回溯算法时从头开始重试。

在本节余下部分，我们将证明，两种实例最优算法 PowerSort 和 c-Adaptive ShiversSort 都是可回溯的。将它们与原地归并算法结合，就能得到首个原地实例最优归并排序算法。

我们将初始游程分解 $\mathcal{R}$ 中的运行段称为原始运行段，将合并产生的运行段称为中间运行段。设 m 为所有中间运行段大小之和；根据定义，所有原始运行段的大小之和为 n 。对于任何基于栈的归并排序算法 $\mathcal{S}$，其运行时间$T_{\mathcal{S}}(n) \geq m + n$。这是因为，对于所有中间运行段 i，我们都必须花费 $|i|$ 时间来合并形成 i 的两个运行段。此外，任何排序算法都必须花费 n 时间来读取输入。如果我们能够证明，将回溯算法应用于 $\mathcal{S}$ 后，在任意输入序列上增加的时间不超过 $O(m + n)$，那么我们就证明了 $\mathcal{S}$ 是可回溯的。

观察 1. 在执行任何基于栈的归并排序时，无需回溯，始终可以得知最顶端运行段 r_1 的长度。 $\mathcal{S}$。

算法	Shannon 最优	可回溯	可跳跃	浅层
PowerSort [20]	✓	✓	✓	✓
c-自适应 ShiversSort [17]	✓	✓	✓	✓
TimSort [1, 21]	✓	✗	✓	✓
α-栈排序 [4]	✗	✓	✓	✓
2-MergeSort [4]	✗	✓	✓	✓
α-合并排序 ($\varphi < \alpha < 2$) [4, 13]	✓	✗	✓	✓
ShiversSort [4]	✗	✓	✓	✓

表 1：我们将研究结果应用于所选基于栈的归并排序后的总结。

引理 1. 任何基于栈的归并排序的折叠阶段，都可以使用回溯算法以原地方式实现，且运行时间只会增加一个常数倍。 $\mathcal{S}$

证明基于这样一个事实：回溯以找回最后两个运行段的代价，与合并它们的代价成正比。根据上述引理，本文余下部分只考虑每种基于栈的归并排序的主循环。

3.2 PowerSort 可回溯

本节中，我们将说明，实例最优的基于栈的归并排序 PowerSort 是可回溯的：回溯成本可以由归并过程中已经付出的成本来「支付」。具体来说，我们将分析成功归并的回溯成本，确定一个停止条件，并利用该停止条件计算失败归并的回溯成本。

Munro 和 Wild 的 PowerSort [20] 为每个运行段分配一个幂值，该值与栈中位于其上方的运行段共同计算。这些幂值用于决定是否应合并栈中的第二和第三个运行段。为了理解运行段如何被赋予幂值，可以想象在输入数组上方叠加一棵完美二叉树，并将数组中的每个索引与一个二叉树节点关联。[4] 数组中的每个索引都关联一个幂值，该索引的幂值就是对应树节点的深度。中心索引的幂值为 0，第一和第三四分位点的幂值为 1，依此类推，直到 $\lfloor \log n \rfloor$。每个运行段关联的幂值，是从其中点到下一个运行段中点之间所有索引中最小的幂值；我们将其称为运行段的幂区间 I。图 3 展示了幂值的示例及其计算方法。请注意， [20]对幂值的定义略有不同，通常比我们的定义大 1。不过，最终得到的算法相同。我们在算法 2 中简要概述了 PowerSort 的归并策略。

应用回溯算法的一大障碍是，Munro 和 Wild 的原始算法不会在发生合并时重新计算幂值。由于原始 PowerSort 中可能会同时存储多达 $\Omega(\log n)$ 个幂值

[4] 为简便起见，我们假设输入的大小为 $n = 2^a - 1$，其中 a，不过该结果对一般的 n 也成立。

算法 2 PowerSort 的归并策略（参见算法 1 中的算法 1）

```
1: 如果 |S| > 1，则
2:     p2 ← NodePower(R2, R1)                    ▷ p3 already known
3:     当 |S| > 2 且 p3 > p2 时
4:         R2 ← Merge(R3, R2)
5:         p3 ← NodePower(R3, R2)
6:     p3 ← p2
```

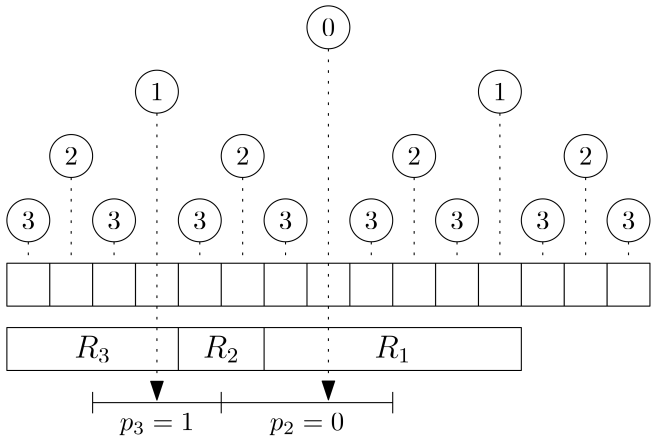

图 3：数组中的每个索引都关联一个幂值，其中索引的幂值是对应树节点的深度（如其上方所示）。每个运行段都关联一个幂值，该幂值是从其中点到下一个运行段中点之间所有索引的最小幂值，即其幂区间 I。幂区间 I_3 和 I_2 定义了幂值 p_3 和 p_2。它们分别显示在对应运行段的下方。 R_1作为顶层运行段，尚未分配幂值。

算法，因此维护浅栈就必须在运行段上升至栈顶时重新计算幂值。重新计算的幂值可能与原始值不相等，从而导致浅层 PowerSort 的行为不同于标准 PowerSort。幸运的是，我们证明情况并非如此，实时计算的幂值与其原始值一致。

引理 2. *PowerSort* 的合并步骤（算法 *2* 第 *4* 行）不会改变第二个运行段的幂值，也不会改变任何已有运行段的幂值。

证明. 见图 4。一次成功的合并会合并运行段 R_2 和 R_3，其中 $p_3 > p_2$。令 I_2' 表示合并后 R_2的幂区间。由于合并只会向左扩展 I_2，扩展到原先属于 I_3 的区域，而 I_3 中的所有索引的幂值均为 $\geq p_3 > p_2$，因此 R_2 的幂值保持为 p_2。类似地， R_4的区间向右扩展至 I_3，只纳入幂值为 $\geq p_3 \geq p_4$的索引，因此其幂值不变。更深层的运行段不受影响。□

由引理 2 可知，PowerSort 几乎具备 3 感知性，因为只需知道其当前 幂区间，我们就能正确地重新计算所有幂值。给定最顶端 3 个运行段的长度以及最顶端运行段的位置，我们可以在常数时间内计算幂区间。根据观察 1，我们始终知道 r_1。通过回溯，恢复标签为 R_2 和 R_3 的运行段长度，最多需要 $r_2 + r_3$ 时间。如果合并条件满足，那么 R_2 和 R_3 的合并成本等于回溯成本，因此在这种情况下，总回溯成本至多为 m。由此，我们证明了成功的合并能够“支付”自身的回溯成本。接下来需要界定失败的

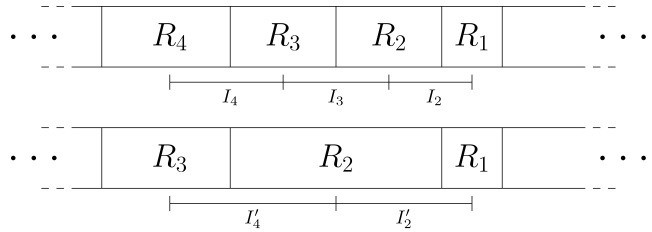

图 4：合并运行段 R_2 和 R_3 会使 I_2 和 I_4 延伸至原先属于 I_3的部分。不过，合并 R_2 和 R_3 时，$p_3 > p_2$ 和 $p_3 \geq p_4$。因此，I'_4 和 I'_2 的幂值不会改变。

合并的回溯成本。为此，我们接下来为回溯版 PowerSort 推导一个停止条件。首先，我们给出关于幂的以下观察。

观察 2。序列中的每个幂值 p 都均匀分布。两个幂值相同的索引 $p > 0$ 之间至少相隔 $n/2^p$ 个元素。[5]

观察 3。任意一个幂值为 p 的索引，与其任一侧最近的幂值小于 p 的索引之间的距离均为 $n/2^{p+1}$ 。因此，任意包含 $n/2^{p+1}$ 个索引的范围内，要么包含幂值为 p 的索引，要么包含幂值小于 p的索引。

这两点都可以直接从我们对索引处隐含持有的幂值的描述中得出。

观察 4。大小至少为 $n/2^p$ 的任何运行段，其幂值至多为 p。

这一观察源于这样一个事实：该运行段的右半部分（其长度至少为 $n/2^{p-1}$）属于它的幂区间；同时也源于观察 3 的第二部分。重新整理后，可得如下结论。

观察 5。大小为 r 的运行段，其幂值必定不超过 $p \leq$ 对数 $\frac{n}{r}$ 。 –

如果能证明长运行段的幂值总是较小，而短游程的幂值总是较大，那将很方便。然而，如果短游程出现在幂值较小的索引处，其幂值可能会任意小。这类“幸运”的短游程可能会因合并比较失败而产生过高的回溯成本，因为最终的合并成本可能无法与该回溯成本成比例。尽管如此，我们证明了整个主循环中的总回溯成本很小。

观察 6. 考虑算法归并步骤中对运行段 R_3 和 R_2 进行归并。R_3 和 R_1 的中点之间的距离至少为 $n/2^{p_3+1}$，其中 p_3 是归并前 R_3 的幂。

[5] 此处以及其他观察中的距离，可能会因具体输入大小 n 而相差至多一。

证明。 R_3 和 R_1 的中点之间的索引范围包含幂区间 I_3 和 I_2。由于发生了归并， $p_3 > p_2$。根据观察 3 的第一部分，这两个幂对应的索引之间的距离至少为 $n/2^{p_3+1}$。证明基于这样一个事实：包含 p_2 和 p_3 的节点分别位于 I_2 和 I_3 中。□

引理 3. 如果 $r_3 \geq n/2^{p_2}$，那么 $p_3 \leq p_2$。

证明。 可直接由观察 4 得出。 □

引理 4. 在 *PowerSort* 的主循环中，只要栈中至少有两个条目，我们就始终知道 r_2。

证明。 任何新的 R_2 要么先前被标记为 R_1，其长度已知（观察 1）；要么是合并的结果。□

根据观察 1 和引理 4，在检验 PowerSort 的合并条件时，我们始终知道 r_1 和 r_2。因此，引理 3 定义了我们的停止条件，并表明失败的合并所产生的回溯成本至多为 $n/2^{p_2}$。

引理 5。 *PowerSort* 中失败合并的总回溯成本为 $O(m+n)$。

证明。 考虑两个运行段 q 和 s，其中 s在栈中恰好位于 q 上方。它们各自的幂分别为 p_q 和 p_s。假设 q 当前的标签为 R_3， s的标签为 R_2，并且我们即将测试它们之间的合并条件。假设我们发现 $|q| > n/2^{p_s}$；根据引理 3，合并条件不成立，因此我们退出回溯算法并终止合并序列。我们可以将这次 $n/2^{p_s}$ 的回溯成本计入哪一项?

我们考虑两种情况。在情况 1 中， q 之后不再参与失败的合并。那么我们可以将这项成本计入 q 本身，因为 $|q| > n/2^{p_s}$。对于所有运行段 q，这至多使回溯成本增加 $(m+n)$，因为 q 可能是原始运行段，也可能是中间运行段。

在情况 2 中， q 还会参与另一次失败的合并。这意味着 q 必须再次取得标签 R_3 ，这又意味着 s 会在主循环期间合并。当 s 合并时， q 的标签为 R_4 ，而 s 的标签为 R_3。根据观察 6， R_3 和 R_1 的中点之间的距离至少为 $n/2^{p_s+1}$。这意味着 $r_1/2 + r_2 + r_3/2 \geq n/2^{p_s+1}$，因此 $r_1 + 2r_2 + r_3 \geq n/2^{p_s}$。为简便起见，我们将其上界估计为 $2(r_1 + r_2 + r_3)$。

我们可以将 $n/2^{p_s}$ 的成本计为当前标签为 R_1、 R_2 和 R_3 的运行段大小的两倍。这三个运行段都以这种方式计费一次。对于 R_2 和 R_3，这是因为它们随即被合并，因而被销毁。对于 R_1，请记住，我们假设 q 不满足其合并条件，因此合并序列随后终止。合并序列结束时，我们会将一个新的运行段压入栈中，并将标签为 R_1 的运行段移至 R_2。这个运行段不可能再向上回到 R_1，因此只会按 R_1 计费一次。

因此，这种情况会对主循环中遇到的每个运行段最多计入其长度的两倍，且至多计入三次（即分别标记为 R_1、 R_2 和 R_3 时）。这会使失败的合并的总回溯成本额外增加 $O(m+n)$。□

我们已经证明，对于所有合并，回溯成本有界于 $O(m+n)$。结论如下。

定理1。 *PowerSort* 可回溯。

推论1。 我们可以使用回溯算法实现 *PowerSort*，从而得到一种原地、稳定的排序算法，其运行时间为 $\Theta(n(1+\mathcal{H}(A)))$。

证明。 这由 PowerSort、稳定原地归并算法和回溯算法的性质可得。 □

4 跳回算法

本节将介绍我们的跳回算法。这是一种通用方法，可将所有已知的基于栈的自然归并排序算法实现为原地算法且无需使用栈，但代价是牺牲稳定性。唯一需要的假设是，基于栈的归并排序算法 $\mathcal{S}$ 几乎具备 k 感知能力。重要的是，如果 $\mathcal{S}$ 的运行时间为 $O(n(1+\mathcal{H}(A)))$，那么我们原地实现的 $\mathcal{S}$ 跳回版本的运行时间也将为 $O(n(1+\mathcal{H}(A)))$。

假设给定一个大小为 n 的输入数组 A，并令 $\lambda = \lceil \log n \rceil + 1$ (λ 是我们在整个跳回算法中使用的一个参数）。首先，我们将短 运行段（即大小至多为 3λ 的运行段）移至 A 的末尾。然后，我们使用标准的原地归并排序算法对它们进行排序；例如，参见 [6–10, 15]。我们可以使用附录D中描述的划分算法，以线性时间原地完成这次移动。见图5。

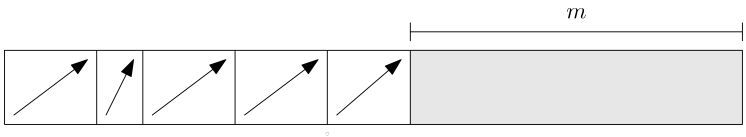

图5：通过划分将所有短运行段移至数组末尾后的结果。灰色块左侧的所有运行段大小均为 $> 3\lambda$。灰色块中的元素原本属于大小为 $\leq 3\lambda$ 的运行段，其顺序可能未得到保留。

接着，我们将给定的基于栈的自然归并排序算法 $\mathcal{S}$ 应用于其余的较长运行段，只做一处改动——我们不再像原先定义 $\mathcal{S}$ 的方式那样使用完整栈来存储运行段大小，而是使用大小为 k 的浅栈，并采用位编码，在运行段自身中编码每个运行段的大小。

引理6. 任何较长运行段的长度，都可以使用可逆原地编码方案编码在该运行段本身中；该方案可在 $O(\lambda) = O(\log n)$ 时间内完成编码、解码和逆转。

该引理的证明引入了两种新的编码方案，我们称之为枢轴编码和标记编码。当运行段中有足够多的不同元素时，前者适用；后者则适用于一般情况。在这两种方案下，我们都将运行段的长度编码在该运行段最后的 $\lambda+1$ 个元素中。我们将在第5节讨论这些编码方案。参见图6。

请回想一下，我们已经将所有短游程移到了数组末尾。利用引理6，我们可以使用跳回算法，只需进行两处修改，便能对剩余的长运行段原地实现几乎-k-感知算法 $\mathcal{S}$。

1. 每当添加一个新的运行段 q，而浅栈中最底部的运行段 s 不再适合放入时，我们就使用引理6中的算法对 s 的大小进行编码 。此步骤耗时 $O(\lambda)$，其开销可计入刚刚加入栈中的运行段 q，因为它的大小必定至少为 $3\lambda+1$。2. 类似地，每当合并一个运行段后浅栈不再满时，我们就将数组中更深处的运行段加入浅栈：先解码其大小，再反转 编码，从而恢复原始运行段。此步骤同样耗时 $O(\lambda)$，其开销可计入合并操作。

通过这两项改动，我们可以高效地维护浅栈中的所有信息，而不会渐近地增加算法的运行时间， $\mathcal{S}$。剩下的就是对所有剩余的短 运行段进行排序。

所有长度至多为 3λ 的短运行段最初都被移到了数组末尾。我们将数组的这部分记为 A'。根据数组香农熵的定义， $n\mathcal{H}(A) = \sum_{i=1}^{\rho(A)} r_i \log(n/r_i)$，由此可知

$$\sum_{i=1}^{\rho(A')} r_i \log(n/r_i) \geq \sum_{i=1}^{\rho(A')} r_i \log(n/3\lambda).$$

但 $\lambda = \lceil \log n \rceil + 1$；因此，若令 $m = \sum^{\rho(A')}{}_{i=1} \; r_i$，那么它们的熵之和为 $\Omega(m\log\ n)$。因此，将所有长度至多为 3λ 的运行段在 $O(m\log\ m)$ 时间内排序，其代价可由它们对原始数组熵的贡献来“支付”。此外，我们还需花费 $O(n)$ 时间将所有此类运行段移至数组末尾。（见附录D）。如上所述，对 A 中的长运行段排序所需的时间在渐近意义上与 $\mathcal{S}$ 相同。因此，我们有以下结论：

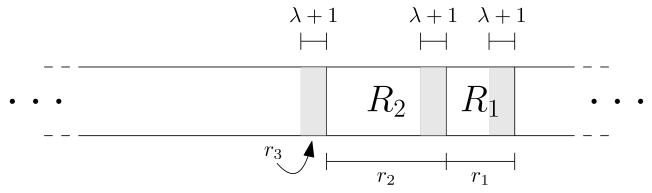

图 6：我们在每个运行段的最后 $\lambda+1$ 个元素中编码运行段长度（以阴影标示），从而能够在 $O(\log n)$ 时间内计算出运行段的长度。这免去了通过回溯算法向前回溯的需要，但代价是失去稳定性。

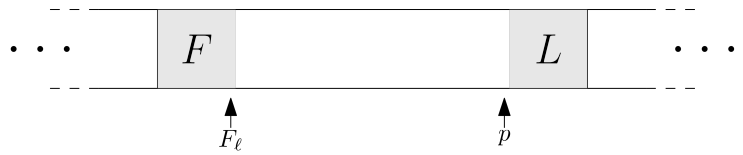

图 7：这里展示了判断一个运行段是否可枢轴化的测试。我们将 F 和 L 定义为该运行段的第一个和最后一个 λ 单元格。我们的枢轴 p 是紧邻 L 之前的元素，而 F_ℓ 是 F的最后一个元素。当且仅当 $F_\ell < p$时，该运行段可枢轴化。

定理 2. 给定一个大小为 n 的数组 A，以及一个基于栈的自然归并排序算法 $\mathcal{S}$，该算法对于常数 $k \geq 2$ 是几乎-k-感知的，我们可以在 $O(n(1+\mathcal{H}(A)) + T_{\mathcal{S}}(n))$ 时间内将 A 原地实现，其中 $\mathcal{S}$ 的运行时间为 $T_{\mathcal{S}}(n)$。

5 我们的位编码方法

本节介绍我们如何在运行段内部原地编码其大小。与 Kuszmaul 和 Westover 等人此前提出的编码方法不同 [19], ，我们的方法不对元素本身及其表示作任何假设，只要求元素可比较且可复制。回想一下，我们定义 $\lambda = \lceil \log n \rceil + 1$。这样定义是因为，每个运行段的大小都可以用 $\lceil \log n \rceil$ 位表示，此外还需要一位来指定所用的编码方法。具体而言，设 $b_1 b_2 \dots b_{\lceil \log n \rceil}$ 为大小 r 的长 运行段 R的大小的一种二进制编码，并令 b_λ 表示所用的编码方法，即，$b_\lambda = 0$ 表示枢轴编码 方法， $b_\lambda = 1$ 表示标记编码方法。我们的两种方法都能在运行段 R内部高效地对 $\boldsymbol{b}$ 进行原地编码。

我们的枢轴编码方法。 我们的第一种编码方法是一种简单的编码方法，我们称之为枢轴编码 方法。设 R 为一个长 运行段，并设 F和 L分别为 R中的前 λ个元素和后 λ个元素。此外，设 p 为 R 中紧邻 L之前的元素，我们称之为枢轴 （即 p是距 R末尾 $\lambda + 1$ 个单元处的元素）。如果 F的最后一个元素 F_ℓ严格小于我们的枢轴 p，则称运行段 R 是可枢轴化 的。见图7。

那么假设 R 是可枢轴化的，这意味着 F 中的每个元素都严格小于 L 和 p 中的每个元素。在我们的枢轴编码方案中对 $\boldsymbol{b}$ 进行编码很容易：我们将 L的每个元素 $L[i]$视为对应于比特 b_i，并且如果 $b_i = 0$，则交换 $L[i]$ 和 $F[i]$。因此，如果 $L[i] < p$，我们就知道 $b_i = 0$；如果 $L[i] \geq p$，我们就知道 $b_i = 1$。见图8。对此类编码进行编码、读取和还原可在 $O(\log n)$时间内完成。

我们的标记编码方法。 对于一个长运行段 R，若其不可枢轴化，即$F_\ell \geq p$，则不仅必须满足 $F_\ell = p$，而且 F 与 L之间的全部元素（记为 M）也都必须等于 p。在这种情况下，我们的枢轴编码方法不起作用，因此我们使用另一种编码方法，称为标记编码方法。

对于我们的标记编码方法，我们将遇到的前两个不同元素指定为标记，分别记为 m_1 和 m_2，其中 $m_1 < m_2$，并使用 $O(1)$内存存储它们。这些标记实际上来自第一个运行段本身，因为每个运行段必须至少包含两个不同的元素。

由于 R 不可枢轴化，我们知道 $F_\ell = p$，因此 M 中的所有元素都等于 p。首先，我们将 L 的副本存储在 M 的前 λ 个单元中。接下来，我们仅使用两个标记，在 L 中编码 $\boldsymbol{b}$，令 $L[i]$ 对应位 b_i；如果 $b_i = 0$，则设置 $L[i] \leftarrow m_1$；如果 $b_i = 1$，则设置 $L[i] \leftarrow m_2$。最后，我们将枢轴值 p 设为 m_2。读取此编码的方式与读取枢轴编码方法相同，因为如果 $L[i] < p$，则 $b_i = 0$，而如果 $L[i] \geq p$，则 $b_i = 1$，这一点仍然成立。读取编码后，如果必须撤销编码，我们只需利用 M 中的副本，将 L 恢复为其原始值；然后通过将 F_ℓ 复制到其前 λ 个元素以及原始枢轴位置 p，来清理 M。参见图9。

6 结论

我们展示了如何实现基于栈的自然归并排序算法，使其成为严格原地算法，同时保留其原有的渐近运行时间；其中还证明，若干这类算法（但并非全部）可以通过简单的回溯算法实现为严格原地算法。我们在跳回中使用的比特编码方法是不可变的，也就是说，无需修改任何元素中的任何比特，这与以往的编码方法不同。

参考文献

1. Auger, N., Jug´e, V., Nicaud, C., Pivoteau, C.：论TimSort的最坏情况复杂度（2019年） 2. Auger, N., Nicaud, C., Pivoteau, C.：归并策略：从归并排序到TimSort（2015年12月 01212839 3），研究报告 hal- 。Barbay, J., Navarro, G.：论排列压缩与自适应排序。《理论计算机科学》**513**，109–123（2013年）

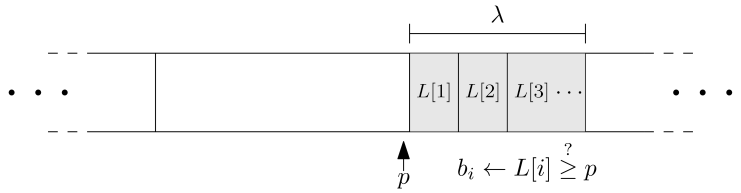

图 8：解码过程，适用于我们的枢轴编码方案和标记编码方案。最后的 λ 个元素编码了我们的比特串 $\boldsymbol{b}$。从末尾起位于 $\lambda + 1$ 个元素处的元素p决定了对应于每个 $L[i]$的比特。解码时，如果 $L[i] \geq p$，那么 b_i 为 1；否则为 0。

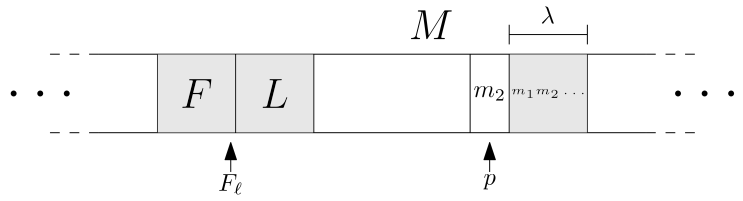

图 9：变换运行段以应用标记编码。我们将块 L 复制到紧接着 F之后的位置，其余元素均属于 M。与枢轴编码一样，我们使用数组的 p 和最后的 λ 个元素来编码 $\boldsymbol{b}$。不过，这次我们通过标记 m_1 和 m_2来实现，而不是使用 F 和 L的元素。我们将 p的元素替换为 m_2，使解码过程与枢轴编码的解码过程相同。

4. Buss, S., Knop, A.：稳定归并排序策略。载于：ACM-SIAM离散算法研讨会（SODA）。第1272–1290页。SIAM（2019年） 5. Chen, J.C.：一种简单的原地归并算法。《信息处理快报》**98**(1)，34–40（2006年） 6. Chen, J.：优化稳定原地归并。《理论计算机科学》**302**(1)，191–210（2003年） 7. Dalkilic, M.E., Acar, E., Tokatli, G.：一种简单的基于洗牌的稳定原地归并算法。《计算机科学进展》**3**，1049–1054（2011年） 8. Edelkamp, S., Wei, A.：使用快速归并排序的最坏情况高效排序。载于：2019年算法工程与实验会议（ALENEX）论文集。第1–14页（2019年） 9. Edelkamp, S., Weiß, A., Wild, S.：QuickXsort：理论与实践中的快速排序方案。《算法学》**82**(3)，509–588（2020年） 10. Ellis, J., Markov, M.：借助完美洗牌的原位稳定归并。《计算机杂志》**43**(1)，40–53（2000年） 11. Geffert, V., Katajainen, J., Pasanen, T.：渐近高效原地归并。《理论计算机科学》**237**(1)，159–181（2000年） 12. Gelling, W.C., Nebel, M.E., Smith, B., Wild, S.：Multiway PowerSort。载于：算法工程与实验研讨会（ALENEX）。SIAM（2023年） 13. Ghasemi, E., Jug´e, V., Khalighinejad, G., Yazdanyar, H.：快速增长自然归并排序中的疾驰。《算法学》**87**(2)，242–291（2025年） 14. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., H¨ahnle, R.：OpenJDK的java.utils.collection.sort()已损坏：好的、坏的与最坏情况。载于：计算机辅助验证。Springer国际出版公司，Cham（2015年） 15. Huang, B.C., Langston, M.A.：常数额外空间下的快速稳定归并与排序*。《计算机杂志》**35**(6)，643–650（1992年12月） 16. Huang, B.C., Langston, M.A.：实用原地归并。Commun. ACM **31**(3)，348–352（1988年3月） 17. Jug´e, V.：自适应Shivers排序：一种替代排序算法。ACM Transaction on Algorithms **20**(4)（2024年8月） 18. Katajainen, J., Pasanen, T., Teuhola, J.：实用原地归并排序。Nordic J. of Computing **3**(1)，27–40（1996年3月） 19. Kuszmaul, W., Westover, A.：使用独占读写内存的原地并行划分算法。arXiv预印本 arXiv:2004.12532（2020年）

20. Munro, J.I., Wild, S.：近乎最优归并排序：能够根据已有运行段进行最优自适应的快速实用排序方法。载于：第26届欧洲算法研讨会（ESA）。第63:1–63:16页。LIPIcs（2018）21 2025 22. Peters, T.：Timsort 描述，检索于 4 月。Roughgarden, T.：超越最坏情况分析。《ACM通讯》**62**(3), 88–96（2019）23. Roughgarden, T.：超越算法的最坏情况分析。剑桥大学出版社（2021）24. Schou, J.K.R., Wang, B.：PersiSort：一种基于持久性理解自适应排序的新视角。载于：加拿大计算几何会议（2024）25. Takaoka, T.：部分解与熵。载于：Kr´alovic, R., Niwinski, D.（编）《2009年计算机科学数学基础会议》。第700–711页。施普林格（2009）26. Wong, J.K.：一些简单的原地归并算法。《BIT 数值数学》**21**, 157–166（1981）27. Zhang, Y., Swanson, S.：非易失性主存中的应用程序性能研究。载于：第31届 IEEE 大容量存储系统与技术研讨会（2015）

一种 c-Adaptive ShiversSort 是可回溯的

本节中，我们将证明，近期提出的实例最优基于栈的归并排序c-Adaptive ShiversSort [17], 具有可回溯性。其归并策略见算法 3。作者将 c 定义为一个可调参数，以便进行更精细的运行时间分析。不过，这些细节与我们的工作无关。

算法 3 c-Adaptive ShiversSort 的归并策略（见算法 1）

1: $\ell_1 \leftarrow \lfloor \log(r_1/c) \rfloor$ （分别为 ℓ_2, ℓ_3）
2: **当** $|S| \geq 3$ 且 $\ell_3 \leq \max\{\ell_2, \ell_1\}$ **时**
3: $\quad R_2 \leftarrow \textsc{Merge}(R_3, R_2)$ ▷ *removes R_3 from S*

我们首先证明，主循环期间的所有成功归并都能“支付”执行这些归并所需的回溯成本。假设算法 3 第 2 行的条件成立。根据观察 1，我们已经知道 r_1。执行此操作的回溯成本至多为 $r_2 + r_3$。这恰好等于合并标签为 R_2 和 R_3的运行段的成本。因此，所有成功归并的回溯成本总和不超过主循环期间的合并成本总和，而后者至多为 m。接下来需要界定条件不成立时的回溯成本。我们先确定 c-Adaptive ShiversSort 的停止条件。

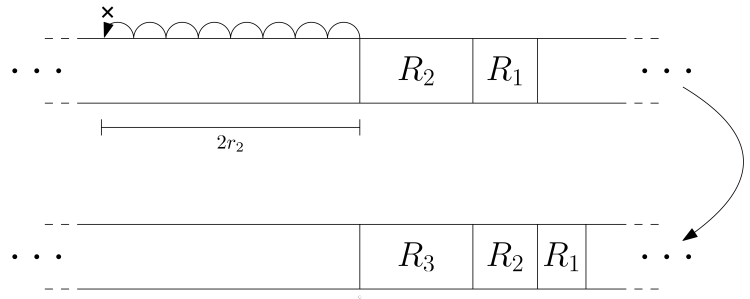

图10：在此图中，我们展示了 c-Adaptive ShiversSort 中合并条件不成立时会发生什么。不失一般性，假设 $\ell_2 = \max\{\ell_2, \ell_1\}$。我们回溯了 $2 \times r_2$ 步，但仍未找到 r_3。根据引理7，合并条件必定为假。因此，我们继续执行主循环，将一个新的运行段压入栈中。

引理7。 若 $r_3 > 2 \times \max\{r_1, r_2\}$，则 $\ell_3 > \max\{\ell_1, \ell_2\}$。

证明。 不失一般性，设 $r_2 = \max\{r_1, r_2\}$。那么 $\ell_2 \geq \ell_1$。由于$r_3 > 2r_2$，以下结论成立。

$$\ell_3 \geq \lfloor \log(2r_2/c) \rfloor = \lfloor \log(r_2/c) + 1 \rfloor = \lfloor \log(r_2/c) \rfloor + 1 > \ell_2$$

□

上述引理给出了将回溯算法应用于 c-Adaptive ShiversSort 时的停止条件：如果回溯 $2 \times \max\{r_1, r_2\}$ 步后仍未找到 r_3，那么我们就知道算法 3 第 2 行的合并条件不成立。见图 10。

引理 8. 失败的合并所产生的回溯成本为 $O(m + n)$。

证明。 如上所述，合并条件不成立时，我们会花费 $2 \times \max\{r_1, r_2\}$ 进行回溯。由于之后不会发生合并，我们无法立即“补回”这部分成本。为简单起见，我们可以将此成本的上界设为 $2 \times (r_1 + r_2)$。因此，每次失败的合并检查所产生的回溯成本，至多为标签为 R_1 和 R_2 的运行段长度之和的两倍。

为了完成论证，我们证明在整个主循环中，每个运行段分别至多在一个合并序列中被标记为 R_1 和 R_2。考虑发生条件失败时，标签为 R_1 或 R_2 的任意运行段 q。假设游程分解非空，我们将一个新的运行段压入栈中，然后重复主循环。这会分别将 q 重新标记为 R_2 或 R_3。由于该算法从不合并 R_1 和 R_2，因此不可能从标签 R_2 上升到 R_1。通过合并，可以将标签从 R_3 改为 R_2。不过，这样做会移除参与合并的组成运行段，并用一个新的合并后运行段取代它们。因此，每个运行段分别至多在一个合并序列中被标记为 R_1 和 R_2。

所有原始运行段和中间运行段的总和为 $m+n$。每个运行段 q 的回溯成本最多计入两次（一次作为 R_1，另一次作为 R_2），每次计为 $2|q|$。因此，失败的合并的总回溯成本至多为 $4(m + n)$。□

我们已经证明，在 c-Adaptive ShiversSort 的主循环中，成功归并和失败的合并所产生的回溯成本为 $O(m+n)$。结合引理 1，我们得出以下结论。

定理 3. *c-Adaptive ShiversSort 是可回溯的。*

进一步检查后发现，我们的证明依赖于 c-Adaptive ShiversSort 的两个关键性质。首先，我们只会合并 R_3 和 R_2，绝不会检查 R_3 以下的运行段。其次，失败的合并条件所产生的回溯成本，至多为已知运行段数量的常数倍。第一个性质使我们能够轻松证明，成功归并的回溯成本由后续的合并承担。第二个性质则是一个较宽松的停止条件，使我们能够界定失败的合并的回溯成本。事实上，只需稍作修改，上述证明就适用于任何 几乎-k-感知的归并排序，只要它满足以下两个性质：

1. 在主循环期间，只有 R_k 和 R_{k-1} 会被合并。

2. 如果对于某个常数 α， $r_k > \alpha \sum_{i=1}^{k-1} r_i$ 成立，则不会发生合并。

推论 2. 原始的 *ShiversSort* 和 α-*StackSort* 是可回溯的。

证明。由于 ShiversSort 和 α-StackSort [4] 都是近似 2 感知的，它们平凡地满足第一条性质。ShiversSort 的合并条件 $2^{\lfloor \log r_2 \rfloor} \leq r_1$ 以 $\alpha = 2$ 满足第二条性质。α-StackSort 同样以其自身参数 α 满足该性质。因此，这两种基于栈的归并排序都是可回溯的。□

B TimSort 不可回溯

人们很容易试图将我们的回溯分析扩展到 TimSort。遗憾的是，在本节中我们通过给出一个输入实例来证明 TimSort 不可回溯：采用回溯算法实现的 TimSort 的渐近性能劣于标准的非原地 TimSort。这促使我们提出第 4 节中的跳回算法，它几乎适用于所有基于栈的归并排序。在 B.1 节中，我们用类似的论证表明 α- MergeSort 同样不可回溯。

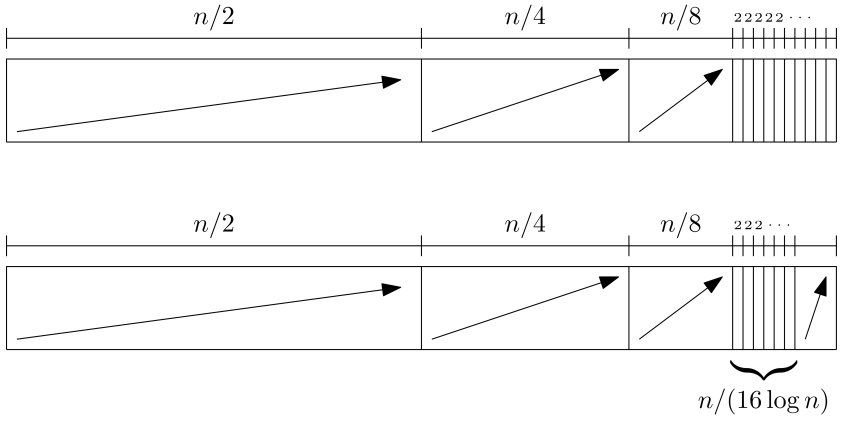

图 11：我们为 TimSort 回溯实现构造的最坏情况输入序列。上方的数组表示我们分析的第一个版本。大小为 2 的一系列 $n/16$ 运行段会导致发生 $\Theta(\log n)$ 阶段，使回溯成本膨胀至$\Omega(n \log n)$。下方的数组将大小为 2 的运行段数量减少到 $n/(16 \log n)$。回溯成本仍为 $\Omega(n \log n)$，而数组的香农熵则变为常数。因此，在第二个序列上，采用回溯算法的 TimSort 的渐近性能将劣于标准 TimSort。

我们的反例考虑一个长度为 n 的数组，其中包含以下预先存在的有序段。前 $n/2$ 个元素构成一个运行段，接下来的 $n/4$ 个元素构成一个运行段，再接下来的 $n/8$ 个元素构成一个运行段。其余元素构成一个

算法 4 TimSort 的归并策略（参见算法 1 的算法 1）

```
1:  当 真 时执行
2:      如果 |S| > 3 且 r_1 > r_3 则                          ▷ merge condition #1
3:          R_2 ← MERGE(R_3, R_2)
4:      否则，如果 |S| > 2 且 r_1 ≥ r_2 则                    ▷ merge condition #2
5:          R_1 ← MERGE(R_2, R_1)
6:      否则如果 |S| > 3 并且 r_1 + r_2 ≥ r_3 则              ▷ merge condition #3
7:          R_1 ← MERGE(R_2, R_1)
8:      否则如果 |S| > 4 并且 r_2 + r_3 ≥ r_4 则              ▷ merge condition #4
9:          R_1 ← MERGE(R_2, R_1)
10:     else
11:         跳出
```

由 $n/16$ 个运行段组成的序列，每个运行段的大小为 2。为简单起见，假设 n 可以被这些运行段整除。见图 11。我们的证明利用了以下关键观察，这些观察可直接从 TimSort 的归并策略推导得出（见算法 4）。

观察 7. 如果 *TimSort* 中的某个合并序列产生一个运行段栈，且这些运行段的长度构成严格递减的 *2* 的幂序列，则该合并序列终止。

观察 8. 我们必须知道 r_2，才能验证在合并序列结束时所有合并条件均为假。

观察 7 有两个显著影响。首先，开头的三个运行段，即长度分别为 $n/2$、$n/4$和 $n/8$的运行段，只会在主循环的最末尾被合并。其次，反复添加大小为 2 的运行段会使它们坍缩为 2 的幂。例如，将前四个运行段加入栈会产生运行段长度序列 $\{n/2, n/4, n/8, 2\}$。这不会产生任何合并。然而，再添加下一个运行段会产生如下栈：$\{n/2, n/4, n/8, 2, 2\}$，它会根据算法 4 的合并条件#2坍缩为$\{n/2, n/4, n/8, 4\}$。我们用如下定义对此加以推广。

定义1。设 *TimSort* 在我们的输入实例上运行的阶段 i定义如下。考虑栈上一组大小为

$$\{n/2, n/4, n/8, 2^{i-1}, 2^{i-2}, \ldots, 8, 4, 2\}.$$

由于它们的大小是递减的 *2* 的幂，根据观察 *7*，不会发生合并。在主循环的下一次迭代中，会添加一个额外的运行段。我们称这是阶段 i 的开始。运行段长度如下：

$$\{n/2, n/4, n/8, 2^{i-1}, 2^{i-2}, \ldots, 8, 4, 2, 2\},$$

这会导致合并条件*#2*被反复应用。对于 $i < \log(n/8)$，根据观察 *7*，当栈的大小为四时，合并序列终止，此时运行段长度如下：

$$\{n/2, n/4, n/8, 2^i\}.$$

此外，令 $k = O(1)$ 表示浅栈的大小。根据回溯算法的定义，我们只保留栈顶的 k 个元素，其余元素都会被遗忘。

观察 9. 在阶段 i 开始时，栈中有 $i+3$ 个运行段。

引理 9. 在 *TimSort* 的主循环中，总共会经历 $\Theta(\log n)$ 个阶段。

证明. 前三个运行段的总长度为 $7n/8$。然后，所有其余运行段中还剩下 $n/8$ 个元素。在阶段 i 结束时，最后一个运行段的大小为 2^i。根据定义，对所有满足 $1 \le i < \log(n/8)$ 的 i 都会出现这种情况。于是，i 至多为 $\log(n/8) - 1$。因此，该算法总共经历 $\Theta(\log n)$ 个阶段。□

推论 3. 存在 $\Theta(\log n)$ 个阶段，使得 $i > k$。

证明. 这是由 k 是常数这一事实得出的。 □

引理 10. 在所有满足 $i > k$ 的阶段 i，应用回溯算法会产生 $\Omega(n)$ 的回溯成本。

证明. 根据观察 9，在该阶段开始时，栈中有 $i+3$ 个运行段。由于 $i > k$，这意味着底部三个大小分别为 $n/2$、$n/4$ 和 $n/8$ 的运行段已被遗忘。当合并序列终止时，栈中有四个运行段。这意味着大小为 $n/8$ 的运行段取得标签 R_2。根据观察 8，我们必须知道 r_2，才能确定合并序列确实已经结束。由 $r_2 = n/8 \in \Omega(n)$ 可得结论。□

根据推论 3 和引理 10，在此输入实例下，TimSort 的总回溯成本为 $\Omega(n \log n)$。然而，如前所述，我们的反例输入实例的熵为 $\Omega(\log n)$。因此，标准 TimSort 的运行时间也为 $\Omega(n \log n)$。

对我们的反例稍作修改，就能得到所需结果。不要使用大小为 2 的 $n/16$ 个运行段，而是将其减少到 $n/(16 \log n)$ 个运行段。让剩余元素组成一个大小为 $n/8 - n/(8 \log n)$ 的单一运行段。见图 11。这个新数组的香农熵为常数，因此标准 TimSort 的运行时间为线性时间。然而，对这个变体进行类似分析可知，应用回溯算法会产生 $\Omega(n \log(n/\log n)) \subseteq \Omega(n \log n)$ 的回溯成本。由此得出以下结论。

定理 4。 *TimSort* 不可回溯。

B.1 附加反例

我们可以用与上述类似的论证证明，α-MergeSort 不可回溯。

首先，让我们来看看 α-MergeSort 的归并策略。

算法 5 α-归并排序的归并策略（见算法 1 的算法 1）

```
1: 当 r_2 < αr_1 或 r_3 < αr_2 执行
2:     如果 r_3 < r_1 则
3:         R_2 ← MERGE(R_3, R_2)
4:     else
5:         R_1 ← MERGE(R_2, R_1)
```

定理5. 存在一些输入序列，使用标准的 α-*MergeSort* 可在 $O(n)$ 时间内完成排序，而使用通过回溯算法实现的 α-*MergeSort* 则需要 $\Omega(n\log\, n)$ 时间。因此， α-*MergeSort* 不可回溯。

证明。 我们的证明与 TimSort 的证明类似；事实上，当 $\alpha = 2$ 时，同一个反例也成立。对于 α 的其他取值，也存在类似的序列，其中三个原始运行段的大小为 n/α， n/α^2， n/α^3。不失一般性，在此证明中我们假设 $\alpha = 2$ ，并使用与 TimSort 相同的反例。

我们注意到，与 TimSort 类似，只有第一个 while 循环条件（第 1 行）会成立，而 if 语句中的检查（第 2 行）始终会失败，因此只有第 5 行中运行段 R_3 和 R_2 之间的合并会发生。因此，该算法对这个输入执行的合并与 TimSort 完全相同。与 TimSort 一样，问题出在某个阶段合并完成后，此时栈中恰好剩下四个运行段： $\{n/2, n/4, n/8, 2^i\}$。在第二个 while 循环条件处，我们会检查 $r_3 < 2r_2$；实现回溯算法时，必须至少完整加载 R_3 ，才能确认该条件不成立。因此，回溯成本至少为 $n/8$ （实际上会达到 $3n/8$），我们的 TimSort 分析仍然成立。□

C 可回退归并排序的其他证明

在本附录中，我们给出证明，说明其他一些归并排序算法也具有可回退性。

C.1 原始 TimSort

我们在 B 节中证明了，现代TimSort算法 [1, 14] 不具有可回退性。令人意外的是，本节将证明，移除第四个条件后的原始 TimSort算法 [21], 具有可回退性。有关该算法的归并策略，请参见算法6。本节将证明以下定理。

定理6. 原始 *TimSort* 具有可回退性。

和前面一样，我们先分析成功归并的回溯成本。

引理11。 执行一次合并时，回溯成本最多由后续合并成本的 *2* 倍支付。

算法6 原始 TimSort 合并策略（参见算法 1 中的算法 1）

```
1: 当 真时执行
2:     如果 |S| > 3 且 r_1 > r_3 则                          ▷ merge condition #1
3:         R_2 ← MERGE(R_3, R_2)
4:     否则，如果 |S| > 2 且 r_1 ≥ r_2 则                    ▷ merge condition #2
5:         R_1 ← MERGE(R_2, R_1)
6:     否则，如果 |S| > 3 且 r_1 + r_2 ≥ r_3 则              ▷ merge condition #3
7:         R_1 ← MERGE(R_2, R_1)
8:     else
9:         跳出
```

证明。 我们通过讨论三种条件各自的回溯成本和合并成本来证明这一点。根据观察 1，在整个算法过程中，我们始终知道 r_1。为简单起见，假设每次开始合并序列中的迭代时，我们只知道 r_1。要检验合并条件 1，我们必须花费 r_2 时间，仅仅从 R_1 的末尾走到 R_2 的末尾，因为我们假设并不知道 r_2。如果该条件成立，我们还要额外花费 r_3 时间走回去。因此，对 R_2 和 R_3 的合并恰好足以支付回溯成本。

现在讨论条件 2。我们知道，刚才已经测试过条件 1，但未能满足。我们又花费了 r_2 时间向前遍历，以找到 R_3的末尾。然而，由于条件 1 未能满足，我们只花了 r_1 时间遍历便停止了。这样一来，我们收回了 r_2。因此，无需额外的回溯成本即可判断条件 2。我们花费了 $r_1 + r_2$ 时间回溯，最终合并了 R_1 和R_2，所以这次合并再次恰好抵消了回溯成本。

最后，我们讨论条件 3。测试条件 1 和 2 并确认它们均未满足，需要付出 r_1+r_2回溯成本。还需要额外的 r_3 工作来确认 $r_1 + r_2 \geq r_3$。因此，我们总共花费 $r_1 + r_2 + r_3$ 时间回溯。由于条件成立，我们知道 $r_1 + r_2 \geq r_3$，所以回溯成本至多为合并 R_1 和 R_2的合并成本的两倍。

现在我们研究失败的合并的回溯成本。

观察 10. 在原始 *TimSort* 中，三个合并条件全部失败所产生的回溯成本至多为 $2(r_1 + r_2)$。

证明。 我们再次假设我们只知道 r_1。如前所述，条件 1 失败需要花费 $r_1 + r_2$ 的时间进行回溯，而条件 2 失败不会产生回溯成本。最后，条件 3 失败需要额外花费 $r_1 + r_2$ 的时间在 R_3 上进行回溯。□

观察 11. 每个唯一游程作为 R_1 至多一次、作为 R_2 至多一次地促成所有合并条件失败

证明。 如果运行段为 R_1，下一次迭代中添加的新运行段会将其推至R_2。在进行合并之前，同一个运行段无法回退到 R_1。同样地，如果运行段为 R_2，新运行段会将其推至 R_3。如果不进行合并，它无法回退到 R_2或 R_1 中的任何一个位置。□

因此，每个运行段所贡献的回溯成本最多为其大小的 4 倍（作为 R_1 时为其大小的 2 倍，作为 R_2时也为其大小的 2 倍）。每个原始运行段和中间运行段都可能出现这种情况，因此失败的合并产生的总回溯成本为 $O(m+n)$。因此，原始 TimSort 是可回溯的。

C.2 2-MergeSort

2019年，Buss 和 Knop 提出了几种新颖自然的基于栈的归并排序，分别称为 2-MergeSort 和 α-MergeSort [4]。在 B.1 节中，我们证明了 α-MergeSort（不要与 Auger 等人 [2] 提出的 α-StackSort 混淆）不是可回溯的。这里，我们证明 2-MergeSort 是可回溯的。2-MergeSort 的归并策略见算法 7。

算法 7 2-MergeSort 的合并策略（见算法 1 的算法 1）

```
1: 当 |S| ≥ 3 且 r2 < 2r1 时
2:     如果 r3 < r1，则
3:         R2 ← MERGE(R3, R2)                ▷ removes R3 from S
4:     else
5:         R1 ← MERGE(R2, R1)                ▷ removes R2 from S
```

定理7. *2-MergeSort* 是可回溯的。

证明。 与往常一样，我们先讨论发生合并时的回走代价。根据观察 1，我们始终知道 r_1。如果发生合并，则条件 $r_2 < 2r_1$ 成立，我们为此付出回溯成本 r_2。如果 $r_3 < r_1$，我们还会产生额外的回溯成本 r_3。由于我们合并了 R_3 和 R_2，此次合并恰好抵消了 $r_2 + r_3$的回溯成本。如果$r_3 \not< r_1$，情况则对称。

现在我们讨论未发生合并时的相关成本。这发生在初始比较未通过时，因此 $r_2 \geq 2r_1$。所以，我们产生回溯成本$2r_1$。我们注意到，在主循环中，这种情况对于每个唯一游程最多只会发生一次。每当失败发生时，我们都会压入一个新的运行段，将 R_1 移至 R_2。要返回栈中的某个位置，这个运行段就必须合并，从而将其销毁。因此，总回溯成本为 $O(m+n)$。□

D 原地游程划分

在本附录中，为完整起见，我们描述一种简单的划分算法。例如，该算法可用于将数组 A 中的所有短游程移动到 A 的末端，同时将长游程移动到该末端区域之前，而不会破坏长游程。

我们可以将长运行段中的每个元素表示为 0，将短运行段中的每个元素表示为 1，从而抽象出这个问题。假设 A 是一个包含 n 个元素的数组，

其中每个元素都是 0 或 1，且 A 中有 $n-m$ 0 个 0 和 m 1 个 1。我们的目标是将短运行段移至 A 的末尾，这等价于将 A 中的 0 和 1 分开，使 $n-m$ 0 个 0 位于 m 1 个 1 之前。我们需要稳定地移动 0，也就是说，不能破坏长运行段；但我们并不在意 1 的稳定性，通常我们的算法也不会保持 1 的稳定性，也就是说，不会保持短运行段的稳定性。只需在遍历 A 的过程中维护两个指针，一个指向下一个 0，另一个指向此前遇到的第一个 1，即可在 $O(n)$ 时间内原地完成此操作。参见算法 8。

在我们的设定中，我们事先不知道哪些元素是 0，哪些是 1。不过，由于这些标签由每个运行段的长度决定，而且我们的算法会对 A 进行一次遍历，因此我们可以在遇到每个运行段时扫描它，以确定它是由 0 还是 1 组成的运行段。对于每个运行段，我们最多扫描两次，每个指针各扫描一次，因此会额外增加 $O(n)$ 的时间。

算法 8 原地运行划分

```
令 A[i] 为第一个满足 A[i] = 1 的单元格
令 A[j] 为第一个满足 A[j] = 0 且 j > i 的单元格
交换 A[i] 和 A[j]
当 i < n 执行
    令 A[i] 为下一个满足 A[i] = 1 的单元格
    令 A[j] 为下一个满足 A[j] = 0 且 j > i 的单元格
    交换 A[i] 和 A[j]
```

注意，该算法会以保持数组中原有顺序的方式将 0 向前移动， A。因此，我们有如下结论：

定理 8。 给定一个包含 n 个单元的数组 A，其中 m 个单元存储 *1*，$n-m$ 个单元存储 *0*，我们可以在原地稳定地将 *0* 划分到前 $n-m$ 个单元中，并在 A 中以 $O(n)$ 时间完成。

E 实验评估回溯成本

我们通过实验比较若干基于栈的归并排序与利用回溯算法生成的原地版本的性能，从而验证理论结果。我们还将这些算法与内置 C++实现的内省排序 std::sort，以及近期提出的原地排序算法 QuickXsort 的性能进行了比较。

E.1 实验设置

我们在 Ubuntu 22.04.5 LTS（Linux 内核 5.15.0-135-generic）上运行了 C++实验。硬件包括两颗 Intel® Xeon® X5680 处理器（@ 3.33GHz，总计 24 MiB L3 缓存）和 94 GiB 内存。该

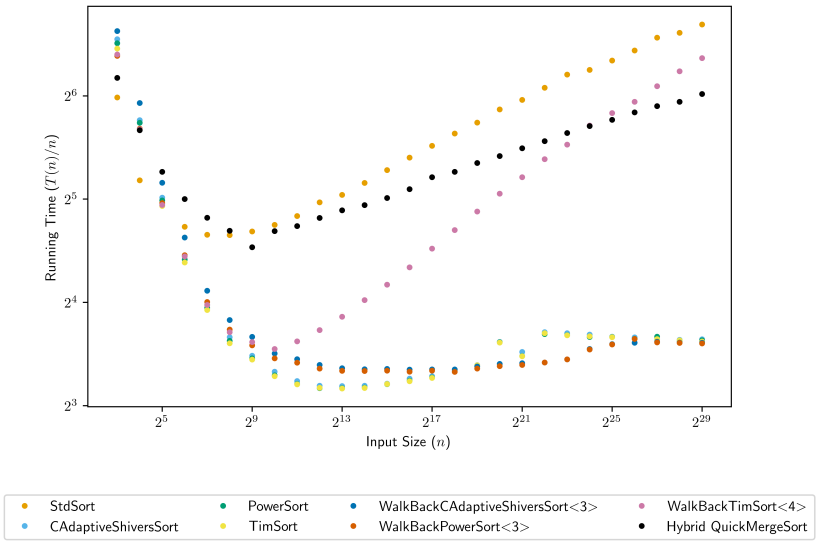

图 12：如 B 节所述，在我们的 TimSort 反例输入上，归一化运行时间的比较。请注意，对于此输入，香农熵为常数。因此，我们将运行时间按 n 缩放。与我们的基于栈的归并排序不同，std::sort 和混合 QMS 并非实例最优，因此无法利用输入的自然顺序。此外，尽管我们所有可回溯的原地归并排序版本都能利用输入的自然顺序，回溯式 TimSort 却不能。这证实了我们在 B 节中的证明：TimSort 不可回溯，并且需要第 4 节所述的编码技术。

代码使用 GCC 11.4.0 版本和 -O3 -march=native 优化标志进行编译。我们使用 Google Benchmark 1.8.3 版本运行基准测试，它是一个面向 C++ 的微基准测试框架。我们测试了两类输入：（1）如 B 节所述的 TimSort 反例输入；（2）使用梅森旋转算法 19937 随机数生成器打乱互异整数而形成的均匀输入分布。许多 QuickXsort 变体由 Edelkamp、Weiß 和 Wild [9], 以及 Edelkamp 和 Weiß[8]提出。我们选择了混合快速归并排序（混合 QMS）变体，它是 [8]中所述最具竞争力的变体。我们按照 [1]中的算法 3 实现了 TimSort。

为简便起见，所有标准的基于栈的归并排序都使用 C++ 标准库中的 std::merge，而原地版本则使用 std::inplace_merge。尽管前者保证能在 $O(n)$ 时间内运行，但 C++ 的原地归并算法在最坏情况下的时间复杂度为 $O(n\log\ n)$，不过我们的实验表明并未达到这一最坏情况。另一点是，我们的输入都不包含重复元素，尽管初步

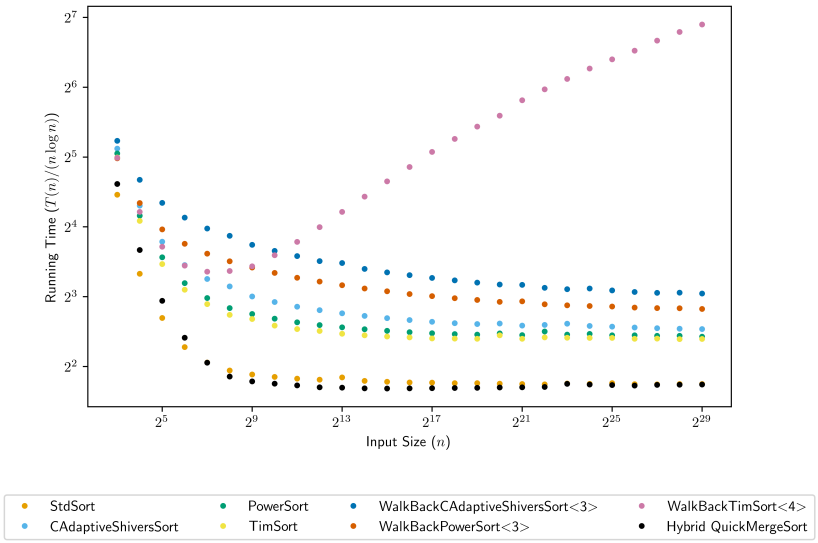

图 13：均匀随机输入下归一化运行时间的比较。由于均匀随机输入的香农熵预期为$\Theta(\log\ n)$，我们将运行时间除以 $n\log\ n$ 进行缩放，使得任何在香农熵意义下最优的算法都应呈现为常数。正如预期，std::sort（StdSort）和混合 QMS 针对这类输入都进行了良好优化。令人欣慰的是，不仅所有基于栈的归并排序都呈现为常数，我们所有原地版本的可回溯 归并排序也都呈现为常数。唯一未呈现为常数的归并排序是 TimSort 的原地版本；如 B 节所示，它不可回溯。

实验表明，这对算法的运行时间没有显著影响。我们的所有代码都将在网上公开。

E.2 结果

在图 12 中，我们比较了各算法在 B 节和 B.1 节所述的 TimSort 反例输入上的表现。该输入的香农熵为常数，因此，任何相对于香农熵达到实例最优的算法都应在线性时间内运行；将运行时间除以n后，其曲线应呈现为常数。事实上，我们看到，所有标准的基于栈的归并排序都能利用输入的自然顺序，曲线看起来是常数；而 C++ 的 std::sort 函数（使用内省排序）和 QuickXsort 并非实例最优，因此表现不佳。令人欣慰的是，应用我们简单的回溯算法后，原地版本的可回溯 归并排序（PowerSort 和 c-Adaptive ShiversSort）似乎仍像其非原地版本一样保持输入敏感。相比之下，正如我们在 B 节中证明的那样，TimSort 的原地版本不可回溯，因此表现不佳。

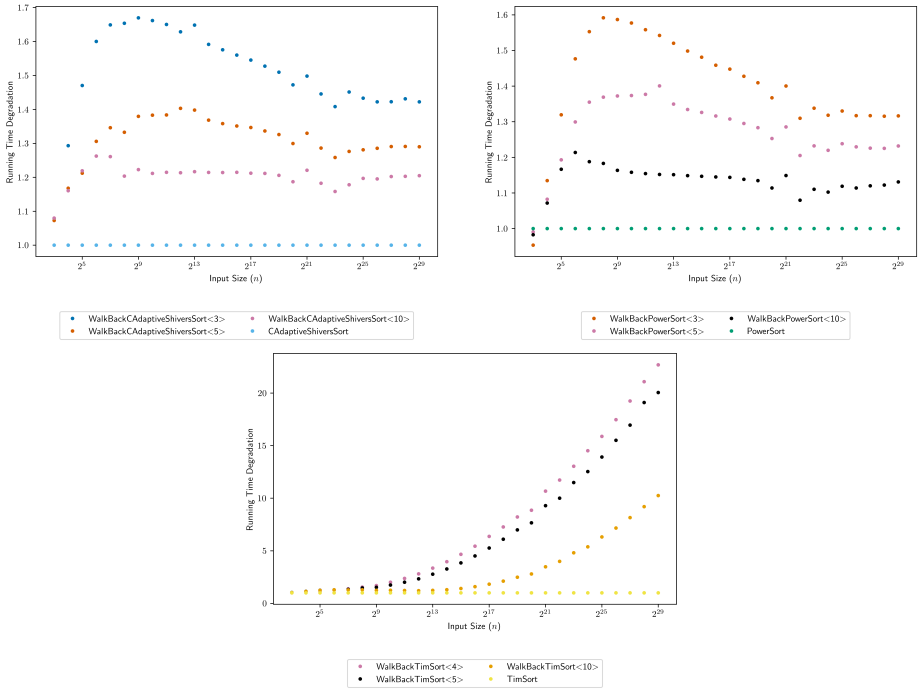

图14：在均匀随机输入上，将每种算法的不同原地版本与其标准版本进行比较。每种算法均相对于标准版本进行测量，标准版本归一化为 1。尽管 c-Adaptive ShiversSort 和 PowerSort 的原地版本与标准版本的差距仅为较小的常数因子，但 TimSort 的原地版本在渐近意义上的性能明显更差。

在图13中，我们比较了各算法在均匀随机输入上的表现。此类输入的香农熵预期为 $\Theta(\log n)$，因此我们将运行时间除以 $n\log\ n$，这样任何 $O(n\log\ n)$ 时间算法的曲线都应呈现为常数。事实上，正如预期，所有算法（尤其是采用回溯算法实现的 TimSort 版本除外）看起来都是常数。在这类输入上，回溯式 TimSort 的表现似乎差于 $O(n\log\ n)$，这一点颇为有趣。

F 补充实验

F.1 不同栈大小的影响

在上一节中，我们给出了每种原地变体所需的最小栈大小：k。对于 PowerSort 和 c-Adaptive ShiversSort，这个值是 3；而对于

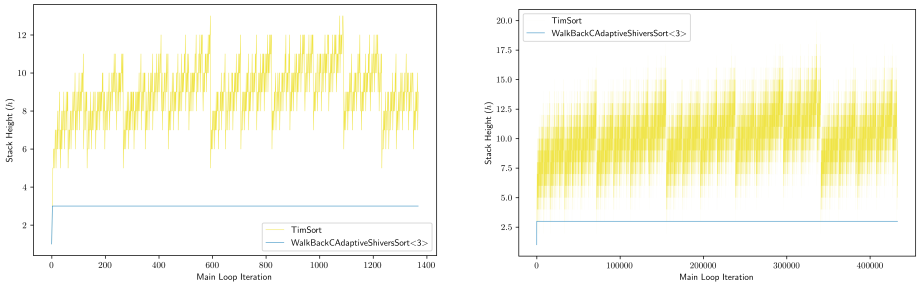

图 15：此处我们跟踪 TimSort 在主循环每次迭代中、新运行段刚被压入栈之后的栈高度（h） S ，针对大小为$n = 2^{20}$的数组。左侧使用 B 节所述的 TimSort 反例输入，右侧使用均匀随机输入。

TimSort 则是 4。现在，我们将每种原地算法的栈大小从 k 增至 5、再增至 10 时的表现，与标准版本进行比较（可以认为标准版本的栈大小最高可达 $\Theta(\log n)$）。

正如预期，我们基于栈的自然归并排序的所有原地版本都会随着栈大小的增加而表现更好，并在栈大小不受限制时表现最佳。可遍历归并排序尤其表现出色，与标准版本相比仅相差一个很小的常数因子。即便栈大小最小仅为 3，PowerSort 和 c- Adaptive ShiversSort 的原地版本相对于标准版本的性能因子也分别在 1.7 和 1.6 以内。这进一步证实，我们简单的回溯算法不仅有效，而且很可能因其非常缓存友好而受益。相比之下，尽管回溯算法具有缓存友好性，TimSort 的原地版本显然仍无法跟上标准版本。即使允许存储栈顶的 10 个运行段，它也只能在一段时间内跟上标准版本，随后便开始偏离。见图 14。这与我们在 B 节中的结果直接吻合。

F.2 栈高度变化

在图 15 中，我们绘制了 TimSort 在主循环每次迭代期间的栈高度（h）：新运行段被压入栈后立即记录；S 对于大小为 $n = 2^{20}$ 的数组，分别展示了我们的 TimSort 不良示例（左）和均匀随机输入（右），并将其与原地版本 c-Adaptive ShiversSort 的栈大小进行比较。从两幅图中可以清楚地看到，TimSort 的栈大小经常以难以预测的方式显著波动，而 c-Adaptive ShiversSort 的原地版本则具有非常稳定的栈大小，上限为 3。这些简单的图表展示了我们的原地版本具有良好的可预测性，这可能使其实现更不容易出错。