bsort：一种适用于整数和浮点数、理论上高效的非比较排序算法

Benjam´ın Guzm´an*独立研究人员，墨西哥墨西哥城
bg@benjaminguzman.dev*本研究工作完成于加入亚马逊之前。

arXiv:2603.08929v1 [cs.DS] 9 Mar 2026

摘要—本文介绍了 bsort，这是一种适用于有符号和无符号整数以及浮点值的非比较排序算法。该算法采用源自二进制快速排序的方法，将这些情况统一起来，实现了 O(wn) 的运行时间渐近行为和 O(w) 的辅助空间，其中 w 为元素的字长。对于字长较小的数据类型，该算法效率很高；实证分析表明，其性能可与热门库中经过高度优化的混合算法相媲美。

索引词—**排序算法、位运算、原地排序、浮点值、非比较排序**

I. 引言

排序是计算机科学中的一个基本问题，涉及将输入序列 $(v_1,\ v_2,\ \ldots\ldots,\ v_n)$ 排列为单调非递减或单调非递增序列。通过比较元素对来解决这一问题的算法称为比较排序。一个已被广泛确立的结果是，任何此类算法的最坏情况时间复杂度下界均为 $\Omega(n\ log(n))$ [1]。

非基于比较的算法（如基数排序）为基于比较的方法提供了一个极具吸引力的替代方案，其时间复杂度为线性。原地位级算法（如二进制快速排序 [2],）提供了一种节省内存的解决方案，但仅限于对符号相同的整数值进行排序。处理同时包含正、负整数元素的序列的扩展方法 [3] ，以及单独的非原地方法，已扩展到能够处理浮点值 [4]。本文旨在提出一种统一的算法，使其兼具原地特性，并能够对有符号整数、无符号整数和浮点值进行排序。

II. 二进制快速排序算法

二进制快速排序算法是一种分治算法，它根据数组元素的位来划分数组 $a = [a_1,\ a_2,\ \ldots\ldots a_n]$。该过程从最高有效位（MSB）开始递归执行。

在单次划分步骤中，数组会被拆分成两个子数组 a' 和 a''，它们满足以下不变量：

(i) $a_i \wedge m \neq 0\ \forall a_i \in a'$

(ii) $a_j \wedge m = 0\ \forall a_j \in a''$

其中，$\wedge$ 是按位与运算符，m 是当前迭代中使用的单个位掩码，其定义为 $m = 1 \ll (s - i)$，其中 $\ll$ 是按位左移运算符，$s \in \mathbb{Z}^+$ 是 a 中每个元素的位数，而 $i \in \mathbb{Z}:\ 1 \leqslant i \leqslant s$ 当前迭代，从 1 开始。a_1 在 a_2 之前还是反过来，取决于所需的排序方向（升序或降序）。对一位进行划分后，算法会使用下一有效位对应的掩码递归地对这些子数组进行排序。此过程会持续进行，直到处理完所有 s 位，此时数组便已完全排序。

例如，假设 $a = [5, 7, 1, 6, 3, 4, 0]$，二进制快速排序算法的完整追踪过程如图 1 所示：

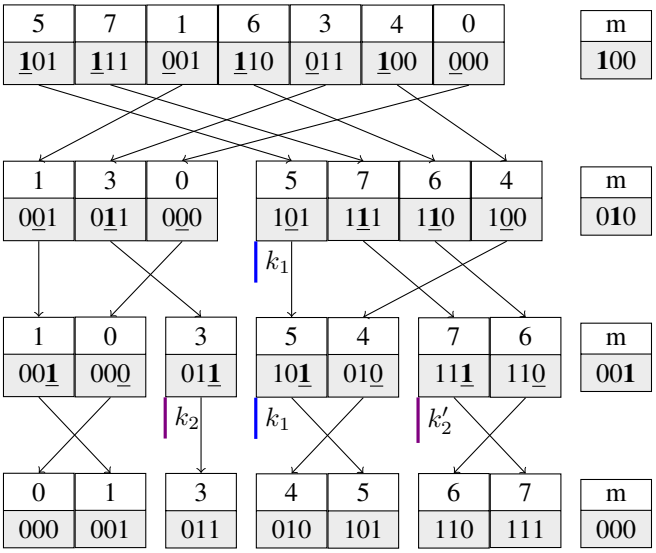


图 1：二进制快速排序算法的追踪过程

bsort 以修改版的二进制快速排序算法作为核心子程序，如算法 1 所述；算法 2 则展示了如何递归使用该子程序对符号相同的整数进行排序。

算法 1： bsort 子过程，用于对数组进行分区

函数*singlePassBSort(a, m, ps, pe, asc)* **为**
  输入： a -待排序的数字数组；
    $a_i \in a \ \forall\ i \in \mathbb{Z}:\ 0 \leqslant i < |a|$
    m - 当前掩码；$\exists k \in \mathbb{N}_0:\ m = 2^k$
    $ps,\ pe$ - 分区的索引
    待排序内容的起始位置（含）和结束位置（不含）；
    $ps,\ pe \in \mathbb{Z}:\ 0 \leqslant ps \leqslant pe < |a|$
    asc - 表示
    顺序为升序
  输出： 一个索引 $k \in [ps,\ pe]$ 使得 $a_i \wedge m = 0, a_j \wedge m = m\ \forall i \in [ps, k), j \in [k, pe]$ 如果
    当且仅当 asc 为真，或者
    $a_i \wedge m = m,\ a_j \wedge m = 0$ 否则
  $k := ps$
  对于 $i \leftarrow ps$ **到** pe，**执行**
    如果 $(asc \wedge \neg(a_i \wedge m)) \vee (\neg asc \wedge (a_i \wedge m))$，**则**
      $swap(a_i, a_k)$
      $k \leftarrow k + 1$
    结束 if
  结束 for
  返回 k
结束

算法 2： 适用于符号相同的 in- 的二进制快速排序整数

函数二分快速排序*(a, m, ps, pe,* 升序*)* **为**
  输入： 与 $singlePassBSort$**相同**
  输出： </style}相同的数组 a，但分区
    起始于 ps，结束于 pe（不包含）
    现在已按升序或降序排列
    顺序取决于 asc
  $k := singlePassBSort(a, m, ps, pe, asc)$
  $m' := m \gg 1$
  如果 $m' = 0$，**则**
    返回 a
  结束 if
  $binaryQuickSort(a, m', ps, k, asc)$
  $binaryQuickSort(a, m', k, pe, asc)$
  返回 a
结束

III. BSORT

标准二进制快速排序算法的按位划分机制以无符号整数表示的字典序为基础。因此，该机制不能直接应用于有符号整数或浮点数等不遵循这一排序原则的数据类型。

证明。考虑一个数组 a，其中包含两个带符号整数，$a = [x,\ y]$，其中 $x < 0$ 且 $y \geqslant 0$。设 w 为以位为单位的字长。在标准的二进制补码表示中，x 的最高有效位为 1，而 y 的最高有效位为 0。

由$singlePassBSort$定义的算法初始划分遍历作用于最高有效位，并根据与掩码 $m = 1 \ll (w-1)$进行按位与的结果对数组进行分区

1) 对于元素 x, $x \wedge m = m$，由于其最高有效位为 1。
2) 对于元素 y, $y \wedge m = 0$，由于其最高有效位为 0。

根据 $singlePassBSort$ 中的逻辑，对于升序排序，比较结果非零的元素会被放入第二个分区（“较大”分区），而结果为零的元素则放入第一个分区。因此，划分步骤会错误地将 x 排在 y 之后。这违反了排序序列所需的单调性不变量，证明该算法无法正确处理混合符号整数。 ■

同样的逻辑也适用于浮点数。根据 IEEE-754标准，最高有效位也是符号位。出于上述相同原因，类似的按位比较会错误地排列负值和正值。此外，其余位的字典序并不直接对应浮点数的数值大小，因此即使在符号相同的分区内，也会导致额外的排序错误。

A. 用于有符号整数的 *bsort*

前面的分析表明，对有符号数调用 $singlePassBSort$ 会颠倒负数分区与非负数分区的相对顺序。例如，在升序排序中，这会导致所有负数错误地排在所有非负数之后。为解决这一问题，我们只需在算法的初始遍历中反转排序方向。算法3将这一过程形式化。

本文中，$a \gg b$ 和 $a \gg b$ 分别表示算术右移和逻辑右移操作。例如，给定一个 8 位整数 $x = -128_{10} = 1000\ 0000_2$，则 $x \gg 1 = -64_{10} = 1100\ 0000_2$ 和 $x \gg 1 = 64_{10} = 0100\ 0000_2$。函数 $sizeof(x)$ 返回操作数 x 的位宽。

算法3： 用于有符号整数的 bsort

函数*bsortSigned(a, asc)* **为**
  输入： a -有符号整数数组
    asc- 表示排序顺序的布尔值
  输出： 同一数组 a，但已排序
  $m := 1 \ll (sizeof(a_0) - 1)$
  $k := singlePassBSort(a, m, 0, |a|, \neg asc)$
  $m' := m \gg 1$
  $binaryQuickSort(a, m', 0, k, asc)$
  $binaryQuickSort(a, m', k, |a|, asc)$
  返回 a
结束

算法3示例：

设输入数据为 $a = [-8,\ -1,\ 3,\ -7,\ 2,\ 6,\ 3]$，$asc = false$，则 $bsortSigned(a,\ asc)$ 的跟踪过程如图2所示。

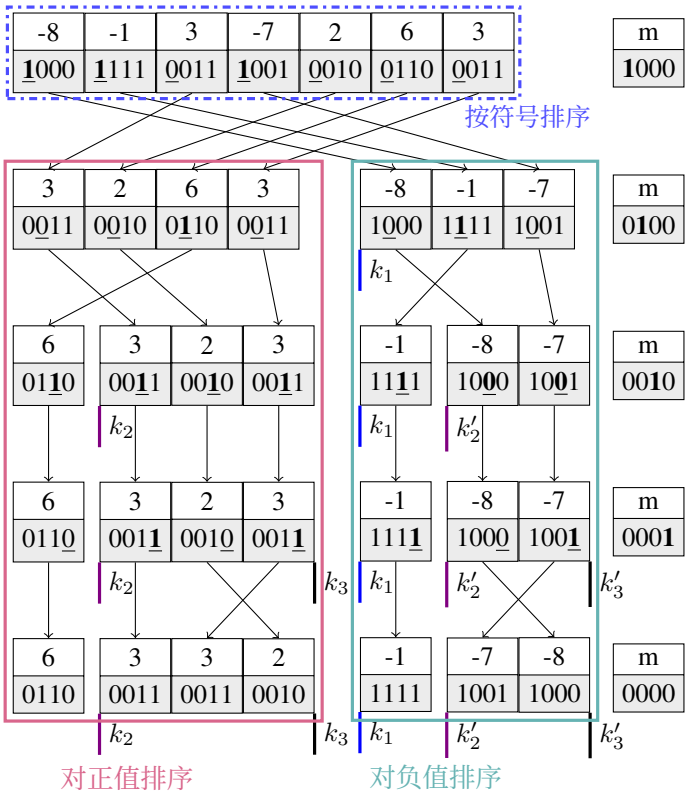


图 2：使用二进制补码表示法对有符号整数应用 bsort 算法的执行过程。

B. 用于浮点值的 *bsort*

定理 1. 设 x 为一个在整数基数 $b > 1$下具有有限小数表示的有理数。则 x 可表示为以下形式：

$$x = s \cdot m \cdot b^p \tag{1}$$

其中，$s \in \{-1, 1\}$ 是符号，$m \in \mathbb{N}_0$ 是尾数，$p \in \mathbb{Z}_0^-$ 是指数。

证明。 任意这样的数 x 都可以分解为其符号$s \in \{-1, 1\}$、整数部分 $a \in \mathbb{N}_0$和小数部分$f \in [0, 1)$。小数部分 f 由有限个 q 数字组成。

我们可以将 x表示为 $x = s \cdot (a + f)$。设由 q 小数位组成的整数为 c。这等价于 $c = f \cdot b^q$，这意味着 $f = c/b^q$。

将其代回 x的表达式中：

$$x = s \cdot \left(a + \frac{c}{b^q}\right) = s \cdot \left(\frac{a \cdot b^q + c}{b^q}\right) = s \cdot (a \cdot b^q + c) \cdot b^{-q}$$

由于 a，b^q，$c \in \mathbb{N}_0$，根据加法和乘法的封闭性，$(a \cdot b^q + c) \in \mathbb{N}_0$。将这个整数记为$m = a \cdot b^q + c$。

此外，设 $p = -q$。由于 $q \in \mathbb{N}_0$，其加法逆元为$p \in \mathbb{Z}_0^-$。因此，我们可以写成：

$$x = s \cdot m \cdot b^p$$

证明完毕。 ■

定理 1 的示例：

- $112.9_{10} = 1 \cdot 1129_{10} \cdot 10^{-1}$
- $-8.348975_{10} = -1 \cdot 8348975_{10} \cdot 10^{-6}$
- $111.11_2 = 1(2^2 \cdot 111_2 + 11_2)2^{-2} = 11111_2 \cdot 2^{-2}$
- $-101.1_2 = -1(2^1 \cdot 101_2 + 1_2)2^{-1} = -1 \cdot 1011_2 \cdot 2^{-1}$
- $4789_{10} = 1 \cdot 4789_{10} \cdot 10^0$

按照定理 1 中所述的表示方式对浮点值进行排序，可通过依次对分量 s、p 和 m 进行多趟排序来完成。

1) 按符号 s 排序：第一趟对整个数组按符号分量排序，将数据划分为两个主要部分：一部分包含所有负数，另一部分包含所有非负数。

2) 按指数 p 排序：接下来，分别对这两个部分按指数 p 排序。这样会形成更细的子分区，其中所有元素的符号和指数都相同。为保持数值顺序正确，负数指数的排序方向必须与正数指数的排序方向相反。例如，要使最终数组按升序排列，负数的指数必须按降序排列。

3) 按尾数 m 排序：由于子分区中的所有元素已经具有相同的符号和指数，按尾数排序即可将它们正确放置在最终的排序位置，因为这等同于对无符号整数进行排序。排序方向也取决于符号。

这一三遍过程通过逐步细化元素的顺序，正确地对整个数组进行排序。形式化证明见定理 2 和 3。

定理 2（排序顺序的必要性）。 *bsortFloat* 算法的正确性依赖于先符号、再指数、后尾数的层次排序顺序。

证明。 我们将证明，除符号 → 指数 → 尾数之外的任何层次排序顺序，都无法对所有输入产生正确的排序数组。一个数的数值首先由其符号决定，然后由指数所决定的大小决定，最后由其尾数加以细化。我们将通过分析违反所需顺序的两种可能方式，用反证法证明这一点。

情况1：符号分量（s）不是主排序键。 假设主排序键是指数（p）或尾数（m）。考虑输入数组$[x, y, z] = [-2.0, 0.0, 0.5]$。根据定理 1，它们的表示为 $x = -1 \cdot 10_2 \cdot 2^0$、$y = 1 \cdot 0_2 \cdot 2^0$ 和$z = 1 \cdot 1_2 \cdot 2^{-1}$。假设我们要以升序对数组进行排序。

如果算法先按尾数排序，就会错误地暗示 $y < z < x$ 因为 $m_y < m_z < m_x$($0_2 < 1_2 < 10_2$)。由于按尾数进行的初始划分会将每个元素置于单元素划分中，随后对每个划分进行的递归排序不会改变它们的顺序。所得数组 $[0.0, 0.5, -2.0]$ 不满足升序排序不变式。

如果算法先按指数排序，就会错误地暗示 $z < x$ 因为 $p_z < p_x$ ($-1 < 0$)。按指数进行的初始划分产生了两个划分，其相对顺序如下：$[z]$ 和 $[x,\ y]$。由于 z 被隔离在其自身的划分中，随后按尾数或符号进行的递归都无法纠正其相对于 x的位置。最终输出$[0.5,\ 0。0,\ -2.0]$同样是错误排序的。

这些情况表明，在有符号数集合中，指数和尾数都无法保持全局单调性。因此，符号位必须作为主排序键；这样可以确保先确定最根本的数值边界（负值与非负值的区别），再比较指数和尾数的大小。

情况 2：符号是主键，但指数（p）不是次级键。这意味着排序顺序必须是符号 → 尾数 → 指数。要对此进行验证，只需考虑符号相同的数。设输入数组为$[x,\ y] = [0.75,\ 2.0]$。同样，假设我们要将数组按升序排序。它们的表示形式分别为 $x = 1 \cdot 11_2 \cdot 2^{-2}$和 $y = 1 \cdot 10_2 \cdot 2^0$。

算法首先将两者正确地放入正数分区。接着按尾数对该分区进行排序。由于 $m_y < m_x$ ($10_2 < 11_2$)，算法将 y 排在 x之前，结果顺序为 $[2.0,\ 0.75]$。由于这些元素现在位于各自的分区中，最后按指数排序不会改变它们的顺序。结果违反了升序排序不变式。

指数决定数值的数量级，而尾数则进一步确定该数量级内的具体数值。因此，算法必须先处理指数，再处理尾数。■

定理 3 (排序顺序的充分性)。 *bsortFloat* 算法结合其子程序，能够正确地对符合定理 *1* 所定义表示方式的浮点数数组进行排序。

证明。 证明分为三个部分，与算法的执行流程相对应：(1) 初始符号分区；(2) 按指数域递归排序；(3) 最后按尾数域排序。

1. 符号排序的正确性： 按符号对浮点数进行分区的逻辑，与为有符号整数确立的方法相同（第 III-A 节）。如该节所示，在升序排序中，符号位的位序与所需的数值顺序相反。若直接进行朴素的升序遍历，非负数就会被错误地排在负数之前。

因此， $bsortFloat$ 采用相同的解决方案：通过调用 *singlePassBSort*并传入 -1升序，在初始轮次中反转排序方向。这样便利用完全相同的机制，正确地将所有负数排在所有非负数之前，从而确立正确的主要顺序。

2. 指数排序的正确性：我们通过归纳证明，*bsortF* 中的递归调用能够根据指数正确地排序每个符号相同的分区。设 w_p 为指数中的位数。设 $P_E(k)$ 为指数排序的归纳假设：

归纳假设 $P_E(k)$**：**处理完指数的 k 最高有效位后（对于 $1 \leqslant k \leqslant w_p$），分区已根据这些 k 位所表示的数值正确排序。

基础情形（$k = 1$）：设算法仅处理最高有效位（b_k）。根据定义， $singlePassBSort$ 会将数组划分为两个集合：$S_0 = \{x | b_k = 0\}$和 $S_1 = \{x | b_k = 1\}$。如果排序方向为升序，S_0 会排在 S_1之前；如果为降序，则反之。在这两种情况下， S_0 与 S_1 的相对顺序都满足该单个位的数值要求。因此，数组相对于$k = 1$ 位已正确划分，且 $P_E(1)$ 成立

归纳步骤： 假设对于某个 $k < w_p$，归纳假设 $P_E(k)$ 成立。这意味着分区已根据指数前 k 位的字典序值正确排序。所有具有相同 k 位前缀的元素现在都位于同一个子分区中。

然后，算法处理第 $(k+1)$ 位。考虑上一步创建的任意一个子分区。*singlePassBSort* 会对这个子分区进行处理，将其分成两个更小的组：一个是 $(k+1)$ 位为 0 的 0 组，另一个是该位为 1 的 1 组。

对于标准升序排序，该过程将 0 组置于 1 组之前。设 x 和 y 分别是来自 0 组和 1 组的任意元素。此时， x 的 $(k+1)$ 位前缀在字典序上小于 y 的前缀。通过将所有此类 x 排在所有此类 y 之前，算法便正确地将排序性质扩展到第 $(k+1)$ 位。

这表明，如果数组已按 k位正确排序，则该算法将按 $k+1$ 位正确排序。因此，如果$P_E(k)$ 为真，则 $P_E(k+1)$ 也为真。

3. 尾数排序的正确性：在指数排序之后，*bsortF* 对每个子分区调用二分快速排序 。在任何此类子分区内，所有元素都具有相同的符号和指数，因此它们的数值顺序仅取决于其尾数。按尾数排序等价于对无符号整数排序，而二分快速排序 对此是正确的。排序方向根据分区的符号选择，从而确保最终顺序正确。

结论：由于数组（1）已按符号正确分区，（2）已按指数正确划分子分区，且（3）每个所得子分区均已按尾数正确排序，因此*bsortFloat* 算法是正确的。■

算法 4 和 5 说明了这一排序过程。需要澄清的是，在这些算法中，$sizeof\ exponent(x)$、 $sizeof\ mantissa(x)$ 和 $sizeof(x)$分别返回指数的位宽、尾数的位宽以及 x的实际值。因此，这些位宽之间的关系定义为 $size\ of(x) = sizeof\ mantissa(x) + sizeof\ exponent(x) + 1$，其中常数项用于计入符号位。

假定符号、指数和尾数按照上述顺序存储（也就是说，

最高有效位对应符号，而最低有效位属于尾数）。不过，即使排列顺序不同，该算法经过一些修改后仍然适用。

算法 4：浮点值的 bsort

函数*bsortFloat(a, asc)* **为**
  输入： a ——一个浮点值数组
    asc——指示排序顺序的布尔值
  输出：同一数组 a，但已排序
  // 按符号排序
  $m := 1 \ll (sizeof(a_0) - 1)$
  $k := singlePassBSort(a, m, 0, |a|, \neg asc)$
  // 按指数和尾数排序
  $m' := m \gg 1$
  $bsortF(a, m', 0, k, asc)$
  $bsortF(a, m', k, |a|, asc)$
  返回 a
结束

算法5：用于浮点值的 bsort，忽略 sign

函数*bsortF(a, m, ps, pe, asc)* **是**
  输入： a -待排序的数字数组；
    $a_i \in a \ \forall \ i \in \mathbb{Z} : \ 0 \leqslant i < |a|$
    m - 当前掩码； $\exists k \in \mathbb{N}_0 : \ m = 2^k$
    $ps, \ pe$ - 待排序分区的起始索引（含）和结束索引（不含）；
    $ps, \ pe \in \mathbb{Z} : \ 0 \leqslant ps \leqslant pe < |a|$
    asc - 表示顺序是否为布尔值是否升序
  输出：一个索引 $k \in [ps, pe]$ 使得 $a_i \wedge m = 0, a_j \wedge m = m \ \forall i \in [ps, k), j \in [k, pe]$ 如果且仅当 asc 为真时，或 $a_i \wedge m = m, a_j \wedge m = 0$ 否则
  // 按指数排序
  $k := singlePassBSort(a, m, ps, pe, asc)$
  // 如果当前掩码是最后一位指数的，那么，现在是时候仅按尾数排序
  如果 $m = (1 \ll sizeof \ mantissa(a_0))$ **那么**
    $m' := m \gg 1$
    $binaryQuickSort(a, m', ps, k, asc)$
    $binaryQuickSort(a, m', k, pe, asc)$
    返回 a
  结束 if
  $m' := m \gg 1$
  $bsortF(a, m', ps, k, asc)$
  $bsortF(a, m', k, pe, asc)$
  返回 a
结束

C. IEEE-754 浮点值的 *bsort*

IEEE-754标准以与定理 1 类似的方式表示浮点值。对于 32 位浮点值，符号位 s 存储在最左侧的位中，指数 p 存储在接下来的 8 位中，最后尾数 m 存储在剩余的 23 位中。64 位、128位或其他任何字长的浮点值 [5]也采用类似的排列方式。图3展示了这种通用排列方式。

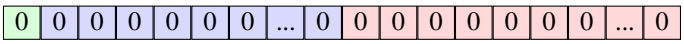


图 3：IEEE-754 浮点值表示。绿色表示符号位，蓝色表示指数位，红色表示尾数位。

然而，定理 1 与 IEEE-754 表示之间存在一些重大差异，因为前者为 $+\infty$、$-\infty$、NaN、$+0$ 和 -0 提供了特定表示，并且还将等于 $2^{k-1} - 1$ 的偏置加到指数上，其中 k 是指数中的位数；尾数则为规格化值包含一个隐式的前导 1，这不会影响所存储位 [5] 的字典序。不过，这些差异不会影响算法 4 和 5 的正确性，其证明可直接由这样一个事实得出：加上一个常数偏移量是一种单调变换，因而天然保留数值顺序。

关于特殊值：

- $+\infty$ 表示为符号位未设置、所有指数位均设置且尾数位均未设置。因此，它会被放入指数位全为 1 的非负数分区中，这些数要么是 $+\infty$，要么是 NaN。因此，$+\infty$在排序数组中的位置意味着，对于数组中的任意元素 $a_i \notin \{+\infty, \ NaN\} \, a, \ a_i < +\infty$ 均成立。

- $-\infty$ 表示为符号位已设置、所有指数位均设置且尾数位均未设置。因此，它会被放入指数位全为 1 的负数分区中，这些数要么是 $-\infty$，要么是 NaN。因此，$-\infty$ 在排序数组中的位置意味着，对于数组中的任意元素 $a_i \notin \{-\infty, \ NaN\}$ 数组$a, \ a_i > -\infty$ 均成立。

- NaN 的符号位可以置位或不置位，所有指数位均置位，且至少有一个尾数位置位。因此，它会被放在数组的两端，其相对顺序由尾数有效载荷决定。- $+0$ 和 -0 的符号位分别不置位和置位，且尾数位和指数位均不置位。因此，-0 会被放入负数分区；由于指数位和尾数位均未置位，在升序排列时它会位于该分区末尾，在降序排列时则位于开头。由此可知，对于任意负元素 a_i，$a_i < -0$ 都成立。同样，对于任意正元素 a_i，$a_i > +0$ 也成立。这种按位方法自然地确保了顺序 $-0 < +0$。

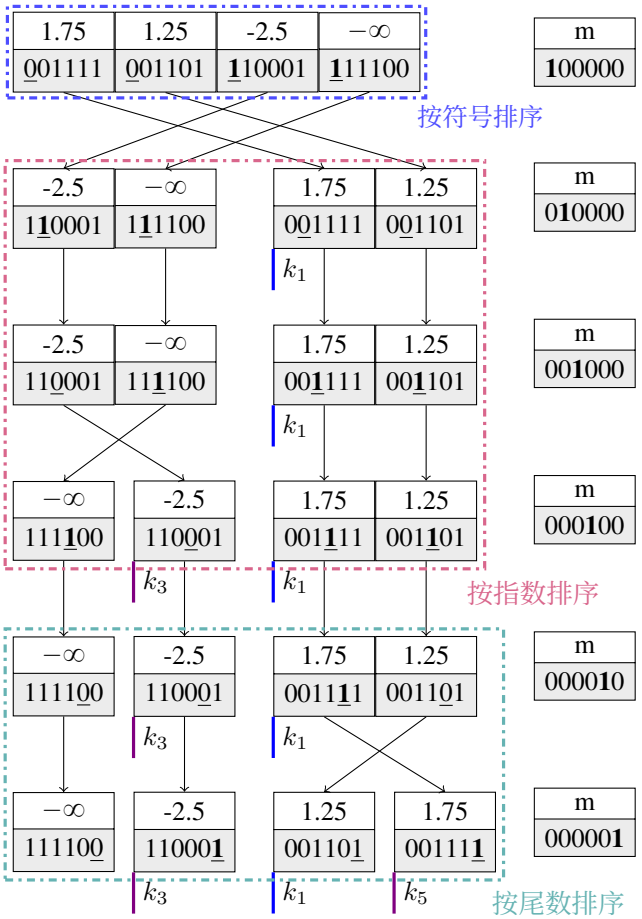


图 4：bsort 对浮点值的追踪过程。为清晰起见，仅展示了相关分区索引(k_1, k_3, k_5)。

示例：

设 $asc = true$, $a = [1.75,\ 1。25,\ -2.5,\ -\infty]$。为简洁明了起见，假设使用 6 位来存储一个浮点值：1 位用于符号位，3 位用于指数（因此偏置为 3），2 位用于尾数；那么$a = [001111_2,\ 001101_2,\ 110001_2,\ 111100_2]$，而 $bsortFloat(a,\ asc)$ 的追踪过程如图4所示。

IV. 分析

A. 理论分析

定理 4. *bsort* 算法在 $O(nw)$时间内对包含 n 个元素、字长为 w 的数组进行排序。

证明。该算法的执行时间由输入的按位划分决定。

- 逐位遍历：对于每一位 $b \in \{0, ..., w-1\}$，算法都会执行一次分区操作（例如$singlePassBSort$）。此过程会对当前分区进行线性扫描。分区中的每个元素都恰好检查一次，以确定其第 b 位的值，因此需要 $O(n_p)$ 次操作，其中 n_p 是当前子分区中的元素数量。
- 每层的总工作量：在递归树的任意深度 d（对应某个特定的位），所有分区中的元素总数恰好为 n。因此，递归的每一层所执行的总工作量为 $O(n)$。
- 总深度：由于算法按照数据的位宽进行递归，递归树的最大深度为 w。

将所有 w 层的工作量求和：

$$T(n) = \sum_{b=1}^{w} O(n) = O(nw) \tag{2}$$

由于对于给定的数据类型，w 是常数（例如，双精度浮点数为 $w = 64$），运行时间渐近行为相对于元素数量 n 呈线性，但为了考虑位深度，形式上表示为 $O(nw)$。■

定理 5. *bsort* 算法以 $O(w)$ 辅助空间运行

证明。 这一结论由两个因素共同决定：

- 原地划分：划分步骤（例如，$singlePassBSort$）采用双指针方法，在原始输入数组中交换元素。此操作只需要常量大小的辅助空间 O(1)，用于存放临时交换变量和指针。
- 递归深度：该算法执行按位分解。对于字长为 w 位的情况，递归树的最大深度为 w。每个栈帧只存储数量固定的变量（指针和位掩码）。因此，递归调用消耗的栈空间为$O(w)$。

由于该算法不会分配与输入规模成正比的额外数据结构 n，且栈深度受限于 w，因此辅助空间的总复杂度为 $O(w)$。所以，我们可以认为 $bsort$ 相对于 n 算法是原地的。

■

B. 实证分析

选择内省排序作为基准进行比较，因为它是 C++ 标准模板库中的标准实现（std::sort）。为了将 bsort 与现代最先进的技术进行比较，测试中纳入了 spreadsort 和基数排序（ska sort），二者均采用非比较式架构。虽然最初考虑了标准 C qsort 实现，但由于其表现始终逊于所有其他受测算法，因此最终结果中未予纳入。这种差距主要是因为 qsort 使用函数指针进行比较，导致编译器无法有效地进行内联和优化，即使在较高的优化级别（-O3）下也是如此 [6]。

qsort 的案例鲜明地展示了理论上的渐近行为与实际性能可能存在显著差异；尽管其渐近界已得到充分确立，但架构开销和编译器限制可能会严重影响它在现实场景中的效率。

算法属性的比较见表 I。

表 I：排序算法的比较分析。

算法	运行时间	内存[+]	原地	比较
内省排序*	$O(n \log(n))$	$O(\log(n))$	Yes	Yes
spreadsort*	$O(n(\frac{k}{s}+s))$	$O(n)$	No	No
ska_sort*	$O(nw)$	$O(d \cdot k)$	Yes	No
快速排序	$O(n \log(n))$	$O(\log(n))$	Yes	Yes
bsort	$O(nw)$	$O(w)$	Yes	No

星号（*）表示混合算法。因此，其运行时间、内存使用量和基于比较的行为会因所执行的具体子程序而异。

bsort 按照本文所述方式实现，即未进行重大修改或优化，采用 C++ 编写，并在配备 48 GB 内存和 Intel i5-8350U 处理器的 64 位 GNU/Linux 环境中，与标准模板库中的内省排序实现（std::sort）、Boost 库中的扩散排序实现（boost::sort::spreadsort）以及经过优化的基数排序实现（ska_sort）进行了比较。不同数据类型对应的字长列表见表 II。为减轻瞬时系统负载等外部因素的影响，并确保统计显著性和一致性，测试使用了 GoogleBenchmark v1.8.0。编译使用 gcc 12.3.1，并采用 -O3 优化标志。

表 II：测试所用各数据类型的字长

数据类型	字长
char	8
短整型	16
int	32
长长整型	64
单精度浮点数	32
双精度浮点数	64

分析基于从多项测试中收集的数据进行。每个测试用例都使用前文提到的四种算法，对大小为 $\{10^5, 10_6, 10_7, 10_8, 10_9, 5 \cdot 10^9\}$ 的数组进行排序，且数组采用特定的数据类型： {字符型、短整型、整型、长长整型、单精度浮点数、双精度浮点数}。本文的源代码（包括实现、测试以及 LaTeX 源文件）已公开发布，网址为 https://benjaminguzman.dev。数据类型与运行时间的比较结果见图 5 和图 6。

从图 5 和图 6 可以看出， *bsort* 表现出稳定的线性扩展。正如上一节对其运行时间的分析所预测的，执行时间会随元素数量 n线性增长。随着 n 增加，bsort 与基于比较的内省排序之间的性能差距仍相对稳定，这证实了按位划分的开销不会在大型数据集上造成不成比例的性能下降。

结果还表明，当位深度较浅时， *bsort*的表现始终优于内省排序；但当位深度较深时，其相对性能会下降。这是

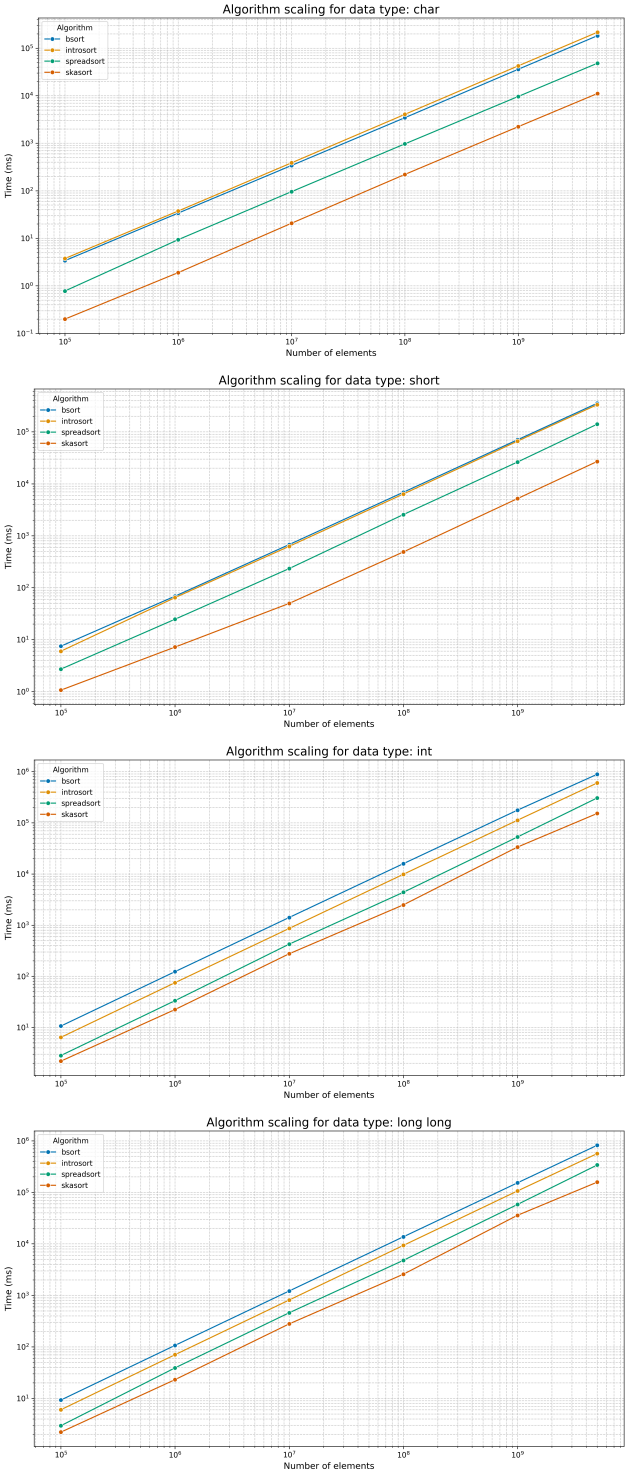


图 5：整数类型字符型、短整型、整型和长长整型的性能比较。

w 因素的直接结果，因为需要更多递归趟才能确定最终顺序。

从理论上讲，在特定条件下，bsort 相较于基于比较的算法具有渐近优势，因为其运行时间受 $O(wn)$ 限制，而高效算法的运行时间通常为

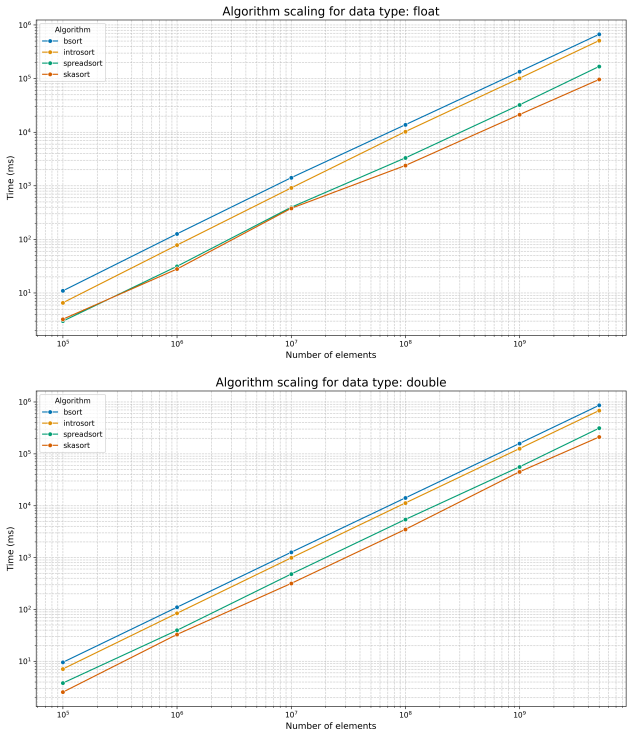


图 6：单精度浮点数和双精度浮点数的性能比较。

$O(n\log(n))$，相对效率可以用比值 $\lim_{n\to\infty} \frac{wn}{nlog(n)}$ =0 和 $\lim_{w\to 0} \frac{wn}{nlog(n)} = 0$ 来刻画。这些极限表明，当数组大小 n 足够大或字长 w 足够小时，bsort 应当严格优于任何基于比较的算法。事实上，我们针对 8 位 char 数据类型的结果证实了这一点。

然而，由于根本性的微体系结构限制，这种理论上的优越性并未转化为实际性能。为单独确定阻碍 bsort 性能的具体瓶颈，在运行基准测试的同一台机器上，使用 Linux 的 perf 工具对该算法进行了性能分析。

实证分析确定了导致 bsort 性能下降的三个主要原因：分支不可预测性、栈污染和指令数量。首先，该算法在分区循环中依赖按位条件检查。对于随机数据而言，这些检查本质上是不可预测的（未命中率接近 50%），从而导致流水线频繁刷新。这一点在图 7 中清晰可见。

其次，bsort 采用僵化的递归结构，由此带来一种隐性代价：寄存器压力和栈污染。不同于会在处理小型数据集时切换到开销较低的迭代方法的混合算法，bsort 会一直递归到最后一位，因此承担了递归函数调用的全部代价。这种反复的上下文切换会导致频繁的寄存器溢出，并占用宝贵的 L1 缓存行来管理栈，而不是存储有效载荷数据。图 8 展示了这种低效性：bsort 的 L1 D-cache 缺失率始终较高涵盖所有数据类型。

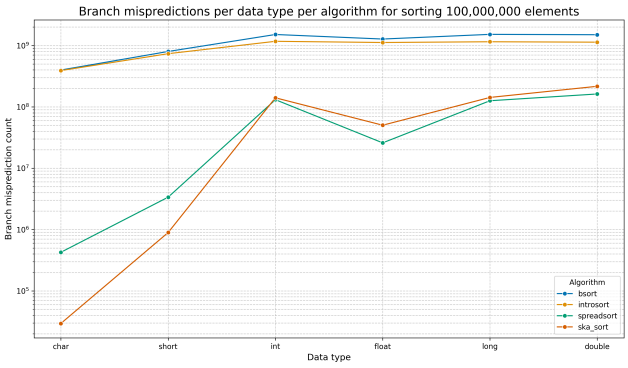


图 7：分支预测错误次数比较

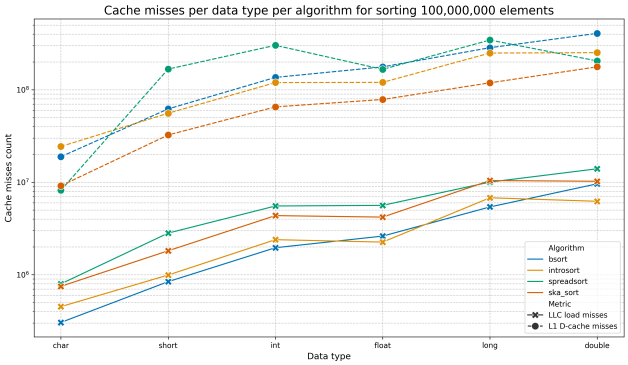


图 8：缓存未命中次数比较

同样的递归结构使 bsort 遍历整个数组 w 次，而内省排序等其他算法通常只需 $O(log(n))$ 次。因此，完成递归所需的大量指令会给处理器带来沉重负担，其执行的指令数显著多于其他算法。例如，bsort 对 64 位整数需要完整扫描数组 64 次，而内省排序处理 10^8 个元素时只需 $\approx$ 26 次。图 9 清楚地展示了这一行为。

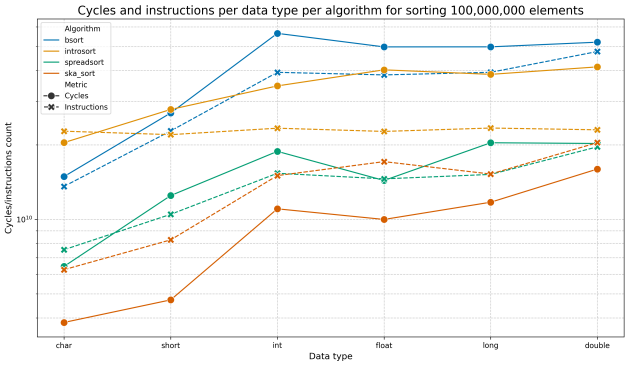


图 9：CPU 周期数和指令数的比较

bsort 与其他算法之间的性能差距还源于算法架构上的根本

差异。如表 I 所述，内省排序、spreadsort 和 ska sort 等最先进的实现都是混合算法；它们会根据分区大小或数据分布自适应地切换策略，以最大限度地提高硬件效率 [7]–[9]。相比之下，bsort 完全依赖同一种按位划分过程，一直细化到最小粒度。

值得注意的是，尽管缺乏混合优化且固有地存在指令开销，bsort 仍展现出有竞争力的性能。在位深范围较浅的情况下（例如 8 位），其性能在统计上优于混合型、缓存优化的内省排序。这表明，虽然当前的单体实现对于较大的字长面临微架构瓶颈，但其核心按位机制仍具有巨大的潜在空间，可用于未来的架构优化。

V. 未来改进

尽管这项工作确立了 $bsort$的基础机制和性能特征，但仍有若干优化方向有待探索。为弥补在较大字长范围内观察到的性能差距，未来版本的算法可改造为混合实现。当分区达到最佳规模时，自适应地切换到非递归的缓存友好型子程序，即可显著减轻 $O(nw)$ 递归深度惩罚。此外，还可通过集成单指令多数据（SIMD）指令来并行执行位掩码操作，采用条件移动指令等无分支分区方法来避免分支预测错误，并利用指令级并行（ILP）重叠执行分区任务，从而提升性能。

这些先进的架构优化目前不在本文的研究范围内，尤其是本研究所用的硬件并不原生支持现代 SIMD 扩展。因此，这些改进的实现与评估将留待后续研究。

VI. 结论

本文提出了 $bsort$，这是一种用于有符号和无符号整数以及浮点值的原地、非比较排序算法，其时间复杂度为 $O(wn)$，空间复杂度为 $O(w)$。形式化证明和实验分析表明， $bsort$ 在较小字长的情形下（例如 8 位整数）尤为有效。然而，研究也发现，对于较大的字长，与最先进的实现相比，该算法因缺乏混合架构且缓存局部性较差，性能受到显著制约。尽管$bsort$为特定数据类型提供了一种内存高效的替代方案，但仍需进一步研究，弥合其理论潜力与实际高性能执行之间的差距。

致谢

我谨向 Kim Garc´ıa 致以深深的谢意，感谢她对我的支持与信任；若没有这些，包括本文在内的许多个人成就都不可能实现。

参考文献

[1] M. L. Fredman，“信息论界在排序中究竟有多好？”《理论计算机科学》，第 1 卷，第 4 期，第 355–361 页，1976 年，访问日期：2021-07-25。 [在线]。可获取：https: //doi.org/ 10 1016 0304 3975 76 90078 5 . / - ( ) - J. L. Bentley 与 R. Sedgewick，[2]“字符串排序与搜索的快速算法”，载于第八届年度 *ACM-SIAM* 离散算法研讨会论文集，1997 年，第 360–369 页。[3] A. V. K. S 与 A. Arya，“Fastbit 基数排序：基数排序的优化版本”，载于*2016* 年第 *11* 届国际计算机工程与系统会议（*ICCES*），2016 年，第 305–312 页。[4] P. Terdiman，“再论基数排序”，Code Corner（博客），2000 年，访问日期：2021-09-26。 [在线]。可获取：https://codercorner.com/ RadixSortRevisited.htm[5] “IEEE 浮点算术标准”，*IEEE 754-2019* 标准（*IEEE 754-2008* 的修订版），第 1–84 页，2019 年。 [在线]。可获取：https://doi.org/ 10 1109 2019 8766229 . / IEEESTD. . S. Meyers，[6]《*Effective STL*》：改进标准模板库使用的具体方法。*50* Addison-Wesley Professional，2001 年。[7] D. R. Musser，“内省排序与选择算法”，软件实践与经验，第 27 卷，第 8 期，第 983–993 页，1997 年 8 月。[在线]。可获取：https://doi.org/10.1002/(SICI)1097-024X(199708) 27 8 3 983 117 3 3 0 2 23 : % C ::AID-SPE % E . .CO; -% M. Skarupke，[8]“我写了一个更快的排序算法”，Probably Dance（博客），2016 年 12 月，访问日期：2021-12-25。 [在线]。可获取：https: // probablydance.com/2016/12/27/i-wrote-a-faster-sorting-algorithm/ [9] S. J. Ross，“spreadsort 高性能通用排序算法”，*C++* 报告，第 14 卷，第 5 期，2002 年，访问日期：2021-12- 25。 [在线]。可获取：https://www.boost.org/doc/libs/latest/libs/sort/ doc/papers/original spreadsort06 2002.pdf

_ _