Magnetosort: A Bidirectional Counting Strategy for Fast Integer Sorting

Ignacio Alfredo Savi Gualco

October 17, 2025

Abstract

Magnetosort is a counting-based integer sorting algorithm that separates negative and positive values into two frequency "rails" and reconstructs the sorted array in linear time with respect to the data size plus the value range. It is especially effective when the numeric range is bounded or when the number of distinct values is small relative to the input size. In controlled experiments, Magnetosort outperformed qsort by 3.7× on 5 million integers and by 5.6× on 50 million integers, while producing identical sorted outputs. This paper presents the algorithm, explains why it works, analyzes its complexity, discusses memory trade-offs and when to use it, and provides reference implementations in C, C++, Python, and additional languages.

Resumo (Português)

Magnetosort é um algoritmo de ordenação baseado em contagem que separa valores negativos e positivos em duas "trilhas" de frequências e reconstrói o vetor ordenado em tempo linear no tamanho dos dados mais o alcance de valores. Ele é especialmente eficaz quando a faixa de valores é limitada ou quando o número de valores distintos é pequeno em relação ao tamanho da entrada. Em experimentos controlados, o Magnetosort superou o qsort em 3,7× com 5 milhões de inteiros e em 5,6× com 50 milhões, produzindo o mesmo resultado ordenado. Este artigo apresenta o algoritmo, explica seu funcionamento, analisa a complexidade, discute limitações de memória, e fornece implementações em C, C++, Python e outras linguagens.

Resumen (Español)

Magnetosort es un algoritmo de ordenación basado en conteo que separa valores negativos y positivos en dos "rieles" de frecuencias y reconstruye el arreglo ordenado en tiempo lineal respecto al tamaño de los datos más el rango de valores. Es especialmente eficaz cuando el rango numérico está acotado o cuando el número de valores distintos es pequeño en relación con el tamaño de la entrada. En experimentos controlados, Magnetosort superó a qsort en 3,7× con 5 millones de enteros y en 5,6× con 50 millones. Este trabajo presenta el algoritmo, explica su funcionamiento, analiza su complejidad, discute las compensaciones de memoria y proporciona implementaciones en C, C++, Python y otros lenguajes.

1 Introduction and Motivation

General-purpose comparison sorts, such as Quicksort, Mergesort, or Introsort, have an average time complexity of $O(n \log n)$ and perform well for arbitrary data types and orderings. However, for integer data with a bounded range or heavy duplication, distribution-based approaches can achieve linear-time behavior in practice. Magnetosort is a practical variant of counting sort designed specifically for signed integers.

The algorithm is structured as follows:

- It counts the frequencies of negative and positive numbers separately.
- It tracks zero values explicitly.
- It reconstructs the final sorted output in ascending order by iterating through the stored counts.

This split "magnetic" view—a negative rail pulling from the left and a positive rail pulling from the right, with zeros in the middle—makes handling negative values clear and avoids overwriting issues when duplicate elements are present.

2 How and Why Magnetosort Works

The core idea of Magnetosort is to determine the final position of elements through frequency counting rather than direct comparison.

1. A single pass over the input array is performed to find the minimum negative value (`min_neg`), the maximum positive value (`max_pos`), and the count of zeros (`zero_count`). This step takes $O(n)$ time.
2. Two counting arrays are allocated: `neg_counts`, where `neg_counts[i]` stores the frequency of $-i$, and `pos_counts`, where `pos_counts[i]` stores the frequency of $+i$.
3. A second pass fills these counting arrays, also in $O(n)$ time.
4. The final array is reconstructed by writing the numbers back in their correct order based on the counts. Negatives are emitted from `min_neg` up to -1, followed by all zeros, and finally positives from 1 up to `max_pos`.

Why it's fast: This approach avoids element-wise comparisons entirely. The work is proportional to the data size plus the integer range covered. Furthermore, memory access is predominantly linear and sequential, which is highly cache-friendly.

3 Algorithm and Correctness

3.1 Pseudocode

1. **Scan:** Iterate through the input array `A` to compute:
   - `min_neg` = $\min(\{v \in A \mid v < 0\})$, default 0 if none.
   - `max_pos` = $\max(\{v \in A \mid v > 0\})$, default 0 if none.
   - `zero_count` = number of zero elements.
2. **Allocate:**
   - `neg_counts` of length $|\text{min\_neg}| + 1$.
   - `pos_counts` of length $\text{max\_pos} + 1$.
3. **Count:**
   - For each $v < 0$ in `A`: `neg_counts[`$-v$`]++`.
   - For each $v > 0$ in `A`: `pos_counts[`v`]++`.
4. **Reconstruct:**
   - For i from $|\text{min\_neg}|$ down to 1: output $-i$, `neg_counts[`i`]` times.
   - Output `zero_count` zeros.
   - For i from 1 to `max_pos`: output $+i$, `pos_counts[`i`]` times.

3.2 Correctness

The correctness of the algorithm follows directly from its construction. By counting the exact multiplicity of each distinct integer value and then writing them back to the array in their natural ascending order (most negative to most positive), the final array is guaranteed to be sorted.

4 Analysis: Complexity, Memory, and Stability

Time Complexity The time complexity is $O(n + R)$, where n is the number of elements in the input array and R is the range of values, defined as $R = |\text{min\_neg}| + \text{max\_pos} + 2$.

Memory Complexity The space complexity is $O(R)$ for the counting arrays. For 32-bit integers, this amounts to approximately $4R$ bytes. For very large or sparse value ranges, the memory requirement can become a limiting factor.

Stability The presented version of Magnetosort is not stable, as elements with the same value are not guaranteed to maintain their original relative order. A stable variant would require cumulative counts and an auxiliary output array. For primitive integers, stability is often not required.

When Magnetosort wins: The algorithm is most effective when the integer range R is moderate relative to n. It excels in scenarios with many duplicates or naturally bounded domains, such as IDs, discretized measurements, or pixel intensities.
When not to use: If the value range is extremely large or sparse relative to n, a dense counting array becomes impractical. In such cases, a sparse histogram (e.g., a hash map) can be used to count frequencies, followed by sorting the distinct keys. This leads to a complexity of $O(n + m \log m)$, where m is the number of distinct values.

5 Experimental Results

Tests were conducted using an online C compiler environment to compare the provided C implementation of Magnetosort against the standard library's `qsort` function. The input data was generated with a value range narrower than the number of elements to ensure a high frequency of duplicates.

- **5,000,000 integers:**
  - Magnetosort: 0.291597 s
  - `qsort`: 1.067445 s
  - **Speedup: $\sim$3.66$\times$**
- **50,000,000 integers:**
  - Magnetosort: 2.268838 s
  - `qsort`: 12.707359 s
  - **Speedup: $\sim$5.60$\times$**

Exact timings may vary based on compiler settings, CPU architecture, cache performance, and memory bandwidth. However, the relative performance advantage is consistent when the range R is small to moderate.

6 Practical Guidance and Variants

Magnetosort should be used when the bounds on the integer domain are known and sufficient memory is available for $O(R)$ counters. For larger or unknown ranges, consider the following alternatives:

- **Sparse Counting:** Use a hash map to store counts, then sort the distinct keys. This is effective when the number of distinct values is much smaller than the range.
- **Bucket Strategy:** The range can be divided into segments, or buckets, to limit peak memory usage.
- **Parallelization:** The counting phase is highly parallelizable. The input can be split into chunks, processed concurrently, and the resulting counts can be reduced into a final histogram.

7 Reference Implementations

7.1 C (Reference)

```c
void magnetosort(int* list, int size) {
    if (size <= 1) return;

    int min_neg = 0, max_pos = 0;
    int zero_count = 0;

    for (int i = 0; i < size; i++) {
        int val = list[i];
        if (val < 0) {
            if (val < min_neg) min_neg = val;
        } else if (val > 0) {
            if (val > max_pos) max_pos = val;
        } else {
            zero_count++;
        }
    }

    long long range_neg = (min_neg == 0) ? 0 : (long long)abs(min_neg) + 1;
    long long range_pos = (long long)max_pos + 1;

    int* neg_counts = NULL;
    if (range_neg > 0) {
        neg_counts = (int*)calloc(range_neg, sizeof(int));
    }
    int* pos_counts = (int*)calloc(range_pos, sizeof(int));
    if ((range_neg > 0 && neg_counts == NULL) || pos_counts == NULL) {
        perror("Failed to allocate counting grids");
        free(neg_counts);
        free(pos_counts);
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < size; i++) {
        int val = list[i];
        if (val < 0) {
            neg_counts[-val]++;
        } else if (val > 0) {
            pos_counts[val]++;
        }
    }
```

```
    int idx = 0;
    if (range_neg > 0) {
        for (long long i = range_neg - 1; i > 0; i--) {
            int c = neg_counts[i];
            while (c--) list[idx++] = -(int)i;
        }
    }
    for (int i = 0; i < zero_count; i++) list[idx++] = 0;
    if (range_pos > 0) {
        for (long long i = 1; i < range_pos; i++) {
            int c = pos_counts[i];
            while (c--) list[idx++] = (int)i;
        }
    }

    free(neg_counts);
    free(pos_counts);
}
```

Listing 1: Reference C implementation of Magnetosort.

7.2 C++17

```
#include <vector>
#include <algorithm>
#include <cstdint>

void magnetosort(std::vector<int>& a) {
    if (a.size() <= 1) return;

    int min_neg = 0, max_pos = 0;
    int zero_count = 0;
    for (int v : a) {
        if (v < 0) min_neg = std::min(min_neg, v);
        else if (v > 0) max_pos = std::max(max_pos, v);
        else zero_count++;
    }

    size_t range_neg = (min_neg < 0) ? static_cast<size_t>(-(int64_t)min_neg) + 1 : 0;
    size_t range_pos = static_cast<size_t>(max_pos) + 1;

    std::vector<int> neg_counts(range_neg, 0);
    std::vector<int> pos_counts(range_pos, 0);

    for (int v : a) {
        if (v < 0) neg_counts[static_cast<size_t>(-v)]++;
        else if (v > 0) pos_counts[static_cast<size_t>(v)]++;
    }

    size_t idx = 0;
    if (range_neg > 0) {
        for (size_t i = range_neg - 1; i > 0; --i) {
            int c = neg_counts[i];
            while (c--) a[idx++] = -static_cast<int>(i);
        }
    }
    for (int i = 0; i < zero_count; ++i) a[idx++] = 0;
    if (range_pos > 0) {
        for (size_t i = 1; i < range_pos; ++i) {
            int c = pos_counts[i];
```

```cpp
            while (c--) a[idx++] = static_cast<int>(i);
        }
    }
}
```

Listing 2: C++17 implementation using `std::vector`.

7.3 Python

```python
def magnetosort(arr):
    n = len(arr)
    if n <= 1:
        return arr

    min_neg = 0
    max_pos = 0
    zero_count = 0
    for v in arr:
        if v < 0:
            if v < min_neg: min_neg = v
        elif v > 0:
            if v > max_pos: max_pos = v
        else:
            zero_count += 1

    range_neg = (-min_neg) + 1 if min_neg < 0 else 0
    range_pos = max_pos + 1

    neg_counts = [0] * range_neg if range_neg > 0 else []
    pos_counts = [0] * range_pos

    for v in arr:
        if v < 0: neg_counts[-v] += 1
        elif v > 0: pos_counts[v] += 1

    i = 0
    if range_neg > 0:
        for val in range(range_neg - 1, 0, -1):
            c = neg_counts[val]
            if c:
                arr[i:i+c] = [-val] * c
                i += c
    if zero_count:
        arr[i:i+zero_count] = [0] * zero_count
        i += zero_count
    if range_pos > 0:
        for val in range(1, range_pos):
            c = pos_counts[val]
            if c:
                arr[i:i+c] = [val] * c
                i += c
    return arr
```

Listing 3: Python implementation using dense lists.

7.4 Additional Languages

Implementations for Java, Go, and Rust are also provided to demonstrate the portability of the concept. The logic remains identical across all languages. *(Due to brevity, the code for Java, Go, and Rust is omitted from this version but follows the same pattern as C/C++.)*

8 Conclusion

Magnetosort is a practical, simple, and very fast approach to sorting integers when the value range is bounded. Its performance advantage over general-purpose comparison sorts grows with larger inputs, especially in the presence of heavy duplication. The primary trade-off is its memory requirement, which is proportional to the integer range rather than the number of elements. For applications involving very large or sparse domains, a sparse counting variant or a traditional comparison sort may be preferable.

Acknowledgment

While this document was formatted with AI assistance, the core idea, concept, and implementation of the Magnetosort algorithm originate from the human author.

A Reproducibility Tips

- Compile C/C++ implementations with high optimization levels (e.g., `-O3 -march=native`) where possible.
- Ensure the value range used in performance tests is appropriate for the available system RAM.
- Verify correctness with an `is_sorted` check after each sort operation.
- For Python performance on large arrays, consider using NumPy for vectorized operations or Cython/Numba to accelerate the counting loops.