Improved bead sort using Fourier Analysis

Paolo Bettelini

October 24, 2023

Contents

1 Introduction

1.1 Abstract

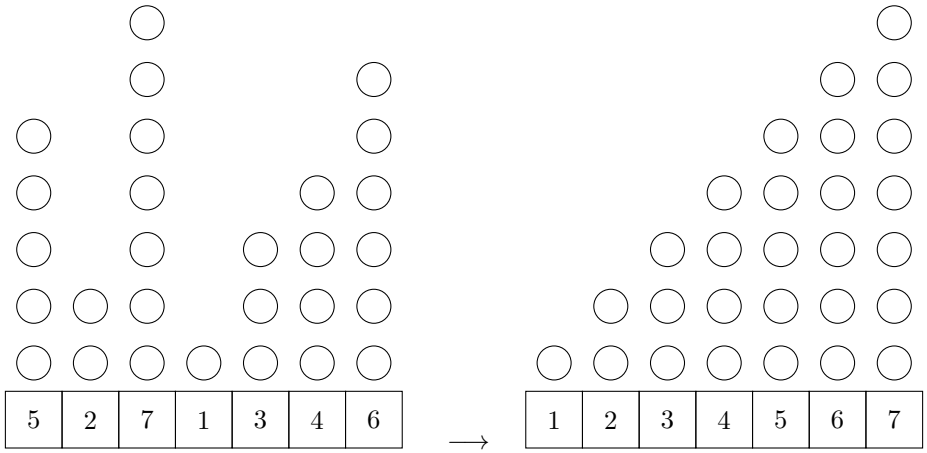
The Bead sort has consistently piqued interest due to its unique attribute of sorting natural numbers without direct comparisons. However, it has typically been considered impractical for practical applications. This paper introduces an endeavor to potentially enhance its efficiency when sorting substantial arrays of numbers in software systems.

The resultant algorithm exhibits improved efficiency, achieving a time complexity of less than $O(N^2)$, particularly when applied to the sorting of extensive arrays containing small natural numbers, ideally with a limited number of distinct elements.

1.2 Bead sort

Bead sort[1] or *gravity sort* is a natural sorting algorithm. This algorithm is primarily used to sort natural numbers, but can be extended to sort the rationals.

The basic idea is to represent the unsorted natural numbers in a matrix (1a), where each columns has n beads. We then shift every bead to the right side of the matrix until they collide with another bead, as if they were affected by gravity. The final step is to count the elements in each columns, forming the final ordered sequence (1b).



(a) Unsorted beads (b) Sorted beads

Figure 1: Beads matrix for $S = 5, 2, 7, 1, 3, 4, 6$

The time complexity of this algorithm, depending on the implementation, ranges from $O(1)$ to $O(P)$ where $P = \sum S_k$.

2 Algorithm

2.1 Matrix representation

Definition 2.1 *Let $a_n \in \mathbb{N}^N$ be the non-empty sequence of N numbers in $\mathbb{N}^*$ to sort, indexed as $(a_1, a_2, \cdots, a_N)$.*

Definition 2.2 *A matrix M with element $m_{i,j}$ is said to be numeric if*

1. $m_{i,j} \in \{0, 1\}$.
2. $m_{i,j} = 0 \implies m_{a,j} = 0, \quad a \leq i$.
3. $m_{i,j} = 1 \implies m_{a,j} = 1, \quad a \geq i$.

Remark. A numeric matrix is a matrix where each of the columns is made of 0s followed by some amount of 1s. Thus, each column represents a number in $\mathbb{N}$ if the 1s are counted. A numeric matrix is a representation of the matrix used in gravity sort.

Definition 2.3 *Let $\tilde{M}$ be the set of all matrices that are numeric.*

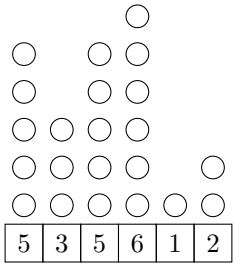
Definition 2.4 *Let $f \colon \mathbb{N}^N \to \tilde{M}$ be a non-injective, surjective function such that $f(a_n)$ is defined as the matrix of size $\max\{a_n\} \times N$ with elements $m_{i,j}$ where*

$$m_{i,j} = \begin{cases} 0 & \max\{a_n\} - i > a_j \\ 1 & \max\{a_n\} - i \leq a_j \end{cases}$$

which is obviously numeric.

Remark. The function f transform a sequence of natural numbers into its corresponding numeric matrix. We are now going to create a function that transforms a numeric matrix given by a sequence into a sinusoidal function. The construction is made such that each row of the matrix represents a frequency starting at $\xi = 1$ from the bottom up. The amount of 1s in the row is the magnitude of the frequency.

Example. Consider the sequence $a_n = (5, 3, 5, 6, 1, 2)$.



$$\longrightarrow \quad 6\sin(t)+5\sin(2t)+4\sin(3t)+3\sin(4t)+3\sin(5t)+\sin(6t)$$

Definition 2.5 *Let $X(t)$ be a time-dependent function*

$$X(t) \triangleq \sum_{k=1}^{N} \sum_{f=1}^{a_k} \sin(ft)$$

Remark. The function $X(t)$ is defined as a naive count of each 1 in the numeric matrix. For each frequency ξ, $X(t)$ will contain the frequency ξ with a magnitude equal to the number of 1s in the row corresponding to ξ. Many numeric matrices correspond to the same function, since the information of the initial order is lost.

It can be noted that computing $X(t)$ at some point $t = k$ has a high time complexity. A computer needs to add a sine function for each 1 in the matrix, meaning more operations as the amount of numbers increase and as the value of the numbers increase. Thus, computing $X(t)$ has a time complexity of $O(N \cdot P)$ where $P = \sum a_n$.

Corollary 2.1 *The function $X(t)$ is equal to*

$$\sum_{k=1}^{N} \frac{\sin(a_k t/2)}{\sin(t/2)} \sin((a_k+1)t/2)$$

for $t \neq 0$.

Proof 2.1 *Note that the naive definition of $X(t)$ contains a geometric series.*

$$\begin{aligned}
\sum_{k=1}^{N}\sum_{f=1}^{a_k} \sin(ft) &= \sum_{k=1}^{N} \Im \sum_{f=1}^{a_k} e^{ift} \\
&= \sum_{k=1}^{N} \Im \left(e^{it} \frac{e^{ita_k} - 1}{e^{it} - 1} \right) \\
&= \sum_{k=1}^{N} \Im \left(e^{it} \frac{e^{ia_k t/2}(e^{ia_k t/2} - e^{-ia_k t/2})}{e^{it/2}(e^{it/2} - e^{-it/2})} \right) \\
&= \sum_{k=1}^{N} \Im \left(e^{it} \frac{e^{ia_k t/2}(2i\sin(a_k t/2))}{e^{it}(2i\sin(t/2))} \right) \\
&= \sum_{k=1}^{N} \Im \left(e^{i(a_k+1)t/2} \frac{\sin(a_k t/2)}{\sin(t/2)} \right) \\
&= \sum_{k=1}^{N} \Im \left((\cos((a_k+1)t/2) + i\sin((a_k+1)t/2)) \frac{\sin(a_k t/2)}{\sin(t/2)} \right) \\
&= \sum_{k=1}^{N} \frac{\sin(a_k t/2)}{\sin(t/2)} \sin((a_k+1)t/2)
\end{aligned}$$

Remark. The time complexity of $X(t)$ can be now reduced to $O(N)$ using this representation.

2.2 Fourier Transform

Definition 2.6 *Let $\hat{X}(\xi)$ be the Fourier Transform of $X(t)$.*

$$\hat{X}(\xi) \triangleq \mathcal{F}\{X(t)\}$$

which is a frequency-dependent function.

Remark. The idea of the algorithm is to apply the Fourier Transform to $X(t)$ in order to retrieve the amount of each frequency in the function, thus retrieving the amount of 1s for each row. The amount of 1s in the row corresponding to the frequency ξ is given by $|\hat{X}(\xi)|$.

Corollary 2.2 *The closed-form for $\hat{X}(\xi)$ is given by*

$$\hat{X}(\xi) = \frac{1}{2\pi} \sum_{k=1}^{N} \int_{0}^{2\pi} \frac{\sin(a_k t/2)}{\sin(t/2)} \sin((a_k+1)t/2) e^{-2\pi i t\xi} \, dt$$

Proof 2.2 *By the representation of* $X(t)$

$$\begin{aligned}\hat{X}(\xi) &= \frac{1}{2\pi}\int_0^{2\pi} e^{-2\pi i t\xi} X(t)\,dt \\ &= \frac{1}{2\pi}\int_0^{2\pi} e^{-2\pi i t\xi} \sum_{k=1}^{N} \frac{\sin(a_k t/2)}{\sin(t/2)} \sin((a_k+1)t/2)\,dt\end{aligned}$$

Since N *is finite, we can apply an interchange of summation and integration*

$$\hat{X}(\xi) = \frac{1}{2\pi}\sum_{k=1}^{N}\int_0^{2\pi} \frac{\sin(a_k t/2)}{\sin(t/2)} \sin((a_k+1)t/2) e^{-2\pi i t\xi}\,dt$$

Remark. The time complexity of computing this function at a point using its closed-form is $O(N)$ and the space complexity $O(1)$. Another approach would be to use the Fast Fourier Transform. The time complexity of the FFT is $O(N\log(N))$, but we would first need to allocate a discrete approximation of $X(t)$ in memory. The space complexity of this operation would be $O(\max\{a_n\})$.

2.3 Sorting Algorithm

The amount of 1s in each row, computed using the Fourier Transform, can be used to generate the sorted sequence of numbers. The principle is the same as the one used during gravity sort: counting the 1s in each column of the sorted numeric matrix gives the ordered sequence.

For all $\xi \in \mathbb{N}^*$, the frequencies given by $|\hat{X}(\xi)|$ are integers, and they follow the property $|\hat{X}(\xi)| \geq |\hat{X}(\xi+1)|$. The first value is always $|\hat{X}(1)| = N$, since every number in a_n produces the frequency 1 of the first row of the numeric matrix (because the sequence is non-empty, every element is at least 1).

In order to retrieve the sorted numbers, we can gradually increase the frequency ξ by 1 starting at $\xi = 1$. Let $k = |\hat{X}(\xi)| - |\hat{X}(\xi+1)|$. Whenever $k \neq 0$ we consider ξ to be the next elements in the list of ordered integers. The element ξ will be the next in the ordered sequence, and it will appear k many times. Repeat this process until all numbers have been extracted ($\xi = \max(S)$ or $|\hat{X}(\xi)| = 0$).

Example. Consider the sequence $a_n = (5, 3, 5, 6, 1, 2)$.
Then, $X(t) = 6\sin(t) + 5\sin(2t) + 4\sin(3t) + 3\sin(4t) + 3\sin(5t) + \sin(6t)$ and

$$f(a_n) = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

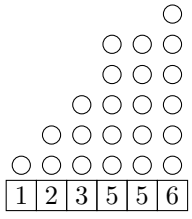
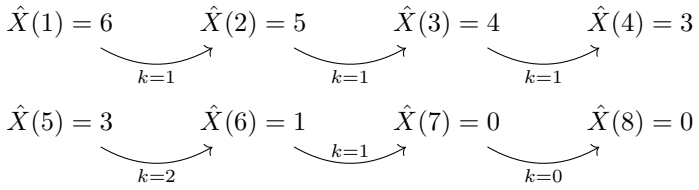


Figure 2: Sorted numerical matrix of a_n

The values of $|\hat{\xi}|$ are easy to guess by looking at $X(t)$.



Note that $\hat{X}(\xi) = 0$ for $\xi > 6$. We are looking for the cases where $k \neq 0$.

1. At frequency 1, the value 6 decreases to 5, meaning that the first sorted element is 1.
2. At frequency 2, the value 5 decreases to 4, meaning that the next sorted element is 2.
3. At frequency 3, the value 4 decreases to 3, meaning that the next sorted element is 3.
4. At frequency 5, the value 3 decreases to 1, meaning that the next sorted elements are the number 5 repeated two times ($k = 3 - 1 = 2$).
5. At frequency 6, the value 1 decreases to 1, meaning that the last sorted element is 6.

It is important to note that duplicate numbers, such as 5 in this case, collapse under the same point in $|\hat{X}(\xi)|$.

3 Space and time complexities

The ordered elements need to be retrieved from $\max\{a_n\}$ values of $|\hat{X}(\xi)|$. Let A be the average distance between two consecutive sorted elements in a_n and D be the amount of distinct elements in a_n.
Note that $D \cdot A \approx \max\{a_n\}$ for an homogeneous distribution.

When using a closed-form for $\hat{X}$, the retrieval has time complexity $O(N \cdot \max\{a_n\})$ if we scan the frequencies linearly. It is possible to optimize this operation using a binary search or similar approaches. The binary search will be used to reach the next point where $|\hat{X}|$ decreases. The binary search will be repeated D times, and for each time it will travel approximately A values. Since computing each value has cost $O(N)$, and the cost of a binary search is $O(\log(N))$, the final time complexity is $O(D \cdot N \log(A))$. The space complexity is always $O(1)$.

4 Appendix

The time complexity of $O(D \cdot N \log(A))$, where A is the average distance between two consecutive sorted elements and D is the amount of distinct elements in the array, can be sometimes advantageous. Of course, the value A needs to be gussed to perform the retrieval of the values properly.

An essential characteristic of this algorithm is its case-independence. The initial arrangement of the array, whether in the best case, average case, or worst case scenarios, does not influence its performance. Furthermore, it is worth noting that the algorithm can be parallelized, allowing for its execution across multiple processors.

The following benchmark evaluates the sorting performance of arrays featuring a uniform random distribution. The initial benchmark (3) focuses on natural numbers drawn from the set $\{1, 2, 3\}$, whereas the subsequent benchmark (4) examines natural numbers within the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. In both scenarios, a linear scan with a constant value of $A = 1$ is employed.

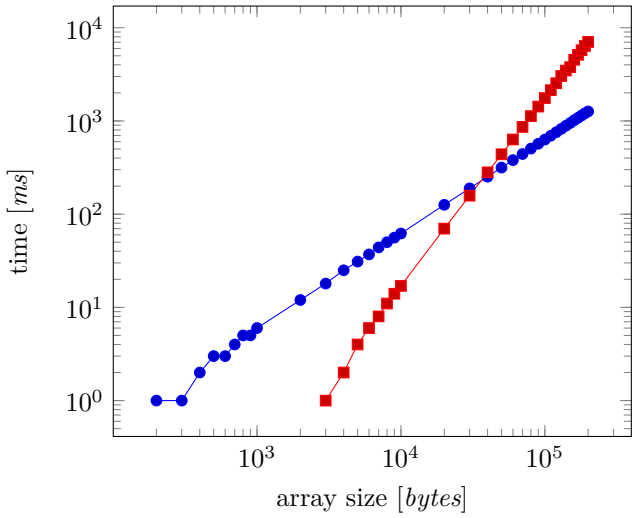


Figure 3: Benchmark 1: gravity (blue) vs naive quicksort (red)

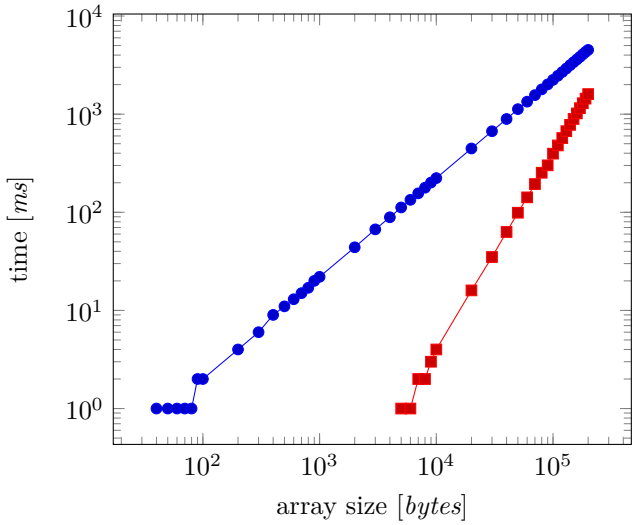


Figure 4: Benchmark 2: gravity (blue) vs naive quicksort (red)

References

[1] J.J Arulanandham, Cristian Calude, and Michael Dinneen. *Bead–Sort: A Natural Sorting Algorithm.* Jan. 2002.

List of Figures