Binary Spaghetti Sort and The Binary Spaghetti Heads Index

This paper was downloaded from TechRxiv (<https://www.techrxiv.org>).

LICENSE

CC BY-SA 4.0

SUBMISSION DATE / POSTED DATE

23-02-2022 / 07-03-2022

CITATION

Calderone, Alberto (2022): Binary Spaghetti Sort and The Binary Spaghetti Heads Index. TechRxiv. Preprint.
<https://doi.org/10.36227/techrxiv.19224645.v1>

DOI

[10.36227/techrxiv.19224645.v1](https://doi.org/10.36227/techrxiv.19224645.v1)

Binary Spaghetti Sort and The Binary Spaghetti Heads Index

Alberto Calderone, Ph.D.
sinnefa@gmail.com

February 2022

Abstract

Sorting items is a fundamental problem in computer science and algorithms design. Sorting algorithms can be divided in two categories, comparison based and non-comparison based, depending on the way they perform the sorting. Non-comparison based algorithms work on the nature of numbers, for instance, the fact that they are represented as decimal numbers. No algorithm so far, to the best of my knowledge, exploits the binary representation of numbers and, at the same time, bitwise operations to sort items. In this work, I present a stable non-comparison based algorithm (inspired by Spaghetti Sort) which sorts lists of $\mathbb{N}^+$ numbers using binary numbers to create an index which guarantees stable sorting. Furthermore, because of its nature, I show how this algorithm can be assembled into an electrical circuit to sort numbers in parallel.

Contact: sinnefa@gmail.com

1 Introduction

The idea of sorting items automatically is one of the fundamental problems in computer science and a large variety of algorithms exist. Sorting is a problem that has been studied for many years, as early as the year 1974 we can find a classification of sorting algorithms. Alfred Aho, John Hopcroft and Jeffrey Ullman (1974) classified algorithms according to complexity, operations needed, stability, resources and recursion.[1].

Sorting algorithms can be comparison based and non-comparison based, depending on how they operate. Examples of comparison based sorting algorithms are Quick Sort, Bubble Sort, Merge Sort and others. On the other hand, non-comparison based algorithms are interesting to study as they are not bound to $\Omega(n * \log(n))$ [2]. Examples of non-comparison based algorithms are Bucket Sort, Radix Sort, Counting Sort and Pigeonhole Sort[3]. In this work, I introduce a stable non-comparison based algorithm inspired by the Spaghetti Sort [4]. To the best of my knowledge, the algorithm presented in this article is the

only sorting algorithm so far, comparison based and non, which exploits binary numbers and bitwise operations.

Spaghetti sort is an algorithm which sorts items in linear time with an analog approach. The algorithm was proposed by A. K. Dewdney in Scientific American[4]. It is a speculation, as it assumes properties of the real world. Preparing Spaghetti takes $O(1)$ time. Tapping them on the table is one operation, and then, your hand can find rods in parallel. Spaghetti sort inspired my solution to sort numbers using binary numbers and operations. Even though this algorithm is not the Spaghetti sort algorithm proposed by Dewdney, it can be seen as the first attempt to translate this idea into code, which could possibly run in parallel, provided that bits are processed simultaneously with a digital circuit.

This paper consists of the following sections: Introduction, The Algorithms, Example, The Electrical Circuit, Comparisons and Results, Conclusions.

2 The Algorithm

Let's assume we have an array of values we want to sort $H = \{s_1, s_2, \dots, s_m\}$ of length m we want to create a set of column vectors $H^b = \{s_1^b, s_2^b, \dots, s_m^b\}$ (The upper b stands for binary representation). To this end, we can represent numbers as long rods ending with 1s. Let's take a generic spaghetti s , if we make it into a series of zeros and one 1 we can see it like a column vector $s^b = (0_1, 0_2, 0_3, \dots, 0_{n-1}, 1)$ where n is the value of s . If we apply this conversion to each element in H we end up with rods of different lengths - if numbers are different - so we also want, as a final step, to add as many 0s as we need to each s^b to make all these vectors of the length of the maximum in H . The collection of these vectors with one 1 forms H^b . These ones will allow us to recover which values are to be inserted in the sorted final array preserving their order in the starting array, a property which makes this algorithm stable.

Let $r = \max(H)$ and $c = \text{length}(H)$. If we denote bits (1/0) with the letter $\beta_{i,j}$ where i is the position in s^b and j the position in H and we then join the vectors in H^b in a matrix M we obtain a structure like the following. We call all this process *Spaghetization*. As an example, the element $\beta_{2,3}$ is the second bit of the third row in the matrix M .

$$M_{r,c} = \begin{pmatrix} \beta_{1,1} & \beta_{1,2} & \cdots & \beta_{1,c} \\ \beta_{2,1} & \beta_{2,2} & \cdots & \beta_{2,c} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{r,1} & \beta_{r,2} & \cdots & \beta_{r,c} \end{pmatrix}$$

Each column of the matrix M is a spaghetti and all spaghetti are aligned at the top of the matrix. Each column is a spaghetti which has one 1 at the row which corresponds to its values in H . The rows of this matrix can be read as binary numbers of length c , thus, they contain 1s in correspondence to the elements in the vectors H .

By turning each rows r in an integer value we obtain a *Binary Spaghetti Heads Index* a data structure we can process with a loop of $\oplus$ (XORs) and shifts - shifts which guarantee the algorithm stability. If we go through the *Binary Spaghetti Heads Index BSHI*, we can use the 1s to pick, in order, values in the starting array H , similarly to the hand in the Spaghetti Sort.

$$BSHI = \left\{ \sum_{k=1}^c \beta_{1,k} * 2^{(c-k+1)}, \sum_{k=1}^c \beta_{2,k} * 2^{(c-k+1)}, \dots, \sum_{k=1}^c \beta_{r,k} * 2^{(c-k+1)} \right\}$$

The *Binary Spaghetti Heads Index* can be seen as a binary row sums and thus it might contain elements with values 0, which can be ignored. The entire algorithm: *spaghetization*, creation of the the *Binary Spaghetti Heads Index* and sorting is shown in the following box.

Algorithm 1 Binary Spaghetti Sort

Require: values, array of $\mathbb{N}^+$ numbers

```

length ← len(values)
sorted ← [0] * length
maximum ← max(values)
minimum ← min(values)
BSHI ← [0] * (maximum - minimum + 1)

for i, n in enumerate(values) do
    /* Shifting is power of 2 */
    BSHI[n - minimum] + = 1 << (length - i - 1)
end for

/* To reverse ordering scan the BSHI in reverse */
index ← 0
for xor in BSHI do
    while xor > 0 do
        k ← int(Log2(xor))
        sorted[index] ← values[length - k - 1]
        xor ← xor ⊕ (1 << k)
        index ← index + 1
    end while
end for
return sorted

```

3 Example

We want to sort the array $x = [3, 2, 3, 5]$. First we do the spaghettization. For example the number 3 becomes $(0, 0, 1)$ but as the maximum is 5 we obtain $(0, 0, 1, 0, 0)$. We can spare a row as the minimum is 2 so, in the end, number 3 becomes: $(0, 1, 0, 0)$, the first column in the following matrix.

$$M_{4,4} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We can now compute the *Binary Spaghetti Heads Index* $BSHI$ by turning each row starting from the bottom into a decimal number and obtain:

$$BSHI = \{1, 0, 10, 4\}$$

The $BSHI$ is the core of the algorithm. We can create it in $O(n)$. We can loop through its elements to find the indexes of the elements in x in the right order. As an example if we take 1, if we make it binary we get $(0, 0, 0, 1)$ which only has one 1 in position 0 which tells us we can get the element $length - 0 - 1$ - assuming we start counting elements index from 0 - in x . Differently, for number 10 in $BSHI$ we get $(1, 0, 1, 0)$ so we need to perform two $\oplus$ to mask the bits one at a time and get to the indices 1 and 3, which point at elements $length - 1 - 1$ and $length - 3 - 1$ respectively in x . Processing rows from bottom to top return $x = [5, 3, 3, 2]$ while processing them from top to bottom return $x = [2, 3, 3, 5]$. The $BSHI$ can be processed bidirectionally.

Figure 1: Example. Spaghettization, Binary Spaghetti Heads Index and Sorting.

4 The Electrical Circuit

The *Spaghetti Sort Algorithm* uses binary numbers to create an index which can be processed with bitwise operations. Because of its characteristics, it can be translated into an electrical circuit. This will result in a sorting circuit which does not use comparisons and is stable, numbers keep their position. Even though going into the details of this circuit goes beyond the purpose of this paper, I briefly describe the key point. The full circuit in logisim [5][6] is available in the GitHub repository linked at the end of this paper. A video explanation is available here: YouTube Channel

Figure 2: The main circuit. In this portion of the main circuit we can see the digits above, as an example, and how the input binary values from 1 to 5 go through a *spaghettization* process using shift. The four digits on the left are the input, the four digits on the right are the sorted output. The red LEDs represent the *Binary Spaghetti Index*.

Figure 3: The while loop translated into a circuit. Through a cascade of shifts and XORs we select one bit set to 1 at a time. We use a counter to keep track of where the number should go. The counter is both an input and an output to cascade more than one of these components. A series of plexers are used to shunt numbers to the right output

5 Comparisons and Results

The complexity of the *Binary Spaghetti Sort* algorithm is proportional to the length of the list but also to the numbers in the list. *Spaghettization* costs $O(n)$. Then, on the one hand, the more unique values we have in the list, the slower the sorting step is as the non zero elements in *BSHI* grows. On the other hand, if the list has lots of repetitions, extracting all the 1s in the binary values inside the *BSHI* makes the algorithm faster. The complexity can be proportional to the number of unique values in the list to sort. In the worst case, we would have to extract 1s from a binary string of only 1s and in that case sorting would require $\lfloor \log_2(n) + 1 \rfloor$ shift but, we would not have to go through the entire *BSHI*. Complexity can be proportional either to one or the other thus it is linear. The following graphs shows performances (Average of 10 iterations) of *Binary Spaghetti Sort* (SS) against other algorithms implemented in searchsort 0.1.6 by Kalanithi available at (<https://pypi.org/project/searchsort/> and <https://github.com/M-Kalanithi-14>). Radix sort either with base 10 or 2 has almost identical performances.

Tests were performed with code developed in Python 3.8 on a home Linux machine with an Intel(R) Core(TM) i7-6500U CPU 2.50GHz and 8GB RAM using the package time to trace execution time and tracemalloc for memory usage. As these algorithms have linear complexity I used log-scale for the y axis.

Figure 4: Small arrays. With uniformly random numbers from 1 to the length of the array performances are similar to those of Bucket Sort. Pigeonhole Sort (PS), Count Sort (CS), Bucket Sort (BS), Radix Sort (RS), Binary Spaghetti Sort (SS).

Figure 5: Large arrays. With uniformly random numbers from 1 to the length but with very large arrays performances are still similar to those of Bucket Sort but the index grows - in this example I did not use a sparse structure, which is recommended in this case. Pigeonhole Sort (PS), Count Sort (CS), Bucket Sort (BS), Radix Sort (RS), Binary Spaghetti Sort (SS).

Figure 6: Small arrays, large numbers. With uniformly random numbers from 3000 to the 5000 performances are amongst the best. Pigeonhole Sort (PS), Count Sort (CS), Bucket Sort (BS), Radix Sort (RS), Binary Spaghetti Sort (SS).

Figure 7: Large arrays, large numbers. With uniformly random numbers from 3000 to the 5000 and large arrays performances are still good but memory usage starts to increase. Pigeonhole Sort (PS), Count Sort (CS), Bucket Sort (BS), Radix Sort (RS), Binary Spaghetti Sort (SS).

6 Conclusions

In this paper I introduced the *Binary Spaghetti Sort*, a non-comparison based sorting algorithm which uses binary numbers and operations. The algorithm is based of a Spaghettization process, a step which can be seen as a matrix of binary numbers. I also introduced the *Binary Spaghetti Heads Index* a data structure which is the translation of a binary matrix into an ordered index which, when processed as binary numbers, preserves the position and frequency of elements in a list.

Due to the nature of the algorithm I point out three facts. First, the *BSHI* might contain a large number of empty values thus, if memory is an issue, a sparse representation is recommended when dealing with enormous arrays. Second, this algorithms heavily relies on large numbers thus Python 3.8[7] is recommended as it supports arbitrarily long integers. Third, being binary strings the representation of integer numbers, long binaries with many 1s might render the algorithm heavy.

In conclusion, this algorithm is unique as it uses binary numbers and bit-wise operations. Because of this, it can also be turned into an electrical circuit which sorts numbers in parallel without comparisons. Finally, from a coding point of view, the algorithm has performances comparable to other well-known algorithms.

An implementation in Python and a Logisim-evolution circuit are available at: <https://github.com/Sinnefa/Binary-Spaghetti-Sort-Binary-Spaghetti-Heads-Index>

References

- [1] Aho A., Hopcroft J., and Ullman J. (1974) The Design and Analysis of Computer Algorithms, Addison Wesley .
- [2] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), "8", Introduction To Algorithms (2nd ed.), Cambridge, MA: The MIT Press, p. 165, ISBN 0-262-03293-7.
- [3] Black, Paul E. "Dictionary of Algorithms and Data Structures". NIST.
- [4] Dewdney, A. K. (June 1984), "On the spaghetti computer and other analog gadgets for problem solving", Scientific American, vol. 250, no. 6, pp. 19–26.
- [5] Logisim: A graphical system for logic circuit design and simulation. (2002) "Journal of Educational Resources in Computing"
- [6] Science of Computing Suite (SOCS): Resources for a Breadth-First Introduction. (2004) With Lynn Ziegler. Technical Symposium on Computer Science Education (SIGCSE)
- [7] Van Rossum, G., & Drake, F. L. (2009). Python 3 Reference Manual. Scotts Valley, CA: CreateSpace.