Knitting Sort: A Novel Stable and Inplace Sorting Technique (An Extension of Exchange Sorting with Bitonic Sorting Network)

Parveen Kumar and Nitin Gupta
Department of Computer Science and engineering, National Institute of Technology Hamirpur, Himachal Pradesh, India
parveen.csed@gmail.com, nitin3041@gmail.com

Abstract— One of the most frequent operations performed on database is searching. To perform this operation we have different kinds of searching algorithms, some of which are Binary Search, Linear Search, Quadratic search[1], but these and all other searching algorithms work only on data, which are previously sorted. An efficient algorithm is required in order to make the searching algorithm fast and efficient. Sorting is an algorithm that arranges all elements of an array, orderly. Sorting Technique is frequently used in a large variety of important applications to arrange the data in ascending or descending order. Several Sorting Algorithms of different time and space complexity are exist and used[9], but they are not suitable because some are stable but not inplace such as merge sort, and other are inplace but not stable such as Quick sort. Two approaches of 2-way merging (odd-even merging and bitonic merging ) have been proposed by Batcher[16]. k-way bitonic sort based on 2-way merging(i.e. two monotonic sequences of n numbers merging is completed by k m-way bitonic sorters[12] and m k-key bitonic sorters, where 2n=km and “k-key means k numbers”) has been proposed by Nakatani et al.[17] and Batcher[18].Since the early work of Batcher, much work has been done on parallel processing system, multi-access memory systems, and so on. This research paper presents a new sorting algorithm named as Knitting Sort which is stable as well as Inplace also. In this algorithm the technique of exchanging elements is used. It is shown in this paper that by various manners of comparisons and swapping the array is sorted. The technique of bitonic sorting network is used to implement this algorithm. The Knitting Sort is implemented on an array to sort it in ascending or descending order. The average case time complexity for this algorithm is O(n).

Index Terms— Algorithms, Bitonic sequence, Comparator, Sorting Network, Swapping, Reverse Cleaner, Monotonic, Array.

I. INTRODUCTION

In computer science, a sorting algorithm is an efficient algorithm which performs an important task that puts elements of a list in a certain order or arranges a collection of items into a particular order. Sorting data has been developed to arrange the array values in various ways for a database. For instance, sorting will order an array of numbers from lowest to highest or from highest to lowest, or arrange an array of strings into
II. Literature Review

In computer science, each sorting algorithm is better in some situation and has its own advantages. For example the insertion sort is preferable to the quick sort for small files and for almost-sorted files. To measure the performance of each sorting algorithm, the most important factor is runtime that a specific sort uses to execute a data. Because the fastest algorithm is the best algorithm, it pays to know which Sorting algorithm is the fastest. Several sorting algorithms are discussed as, Bubble Sort, Exchange two adjacent elements if they are out of order. Repeat until array is sorted. Selection Sort, Find the smallest element in the array, and put it in the proper place. Swap it with the value in the first position. Repeat until array is sorted.(starting at the second position and advancing each time). Insertion Sort, Scan successive elements for an out-of-order item, then insert the item in the proper place. Quick Sort, Partition the array into two segments. In the first segment, all elements are less than or equal to the pivot value. In the second segment, all elements are greater than or equal to the pivot value. Finally, sort the two segments recursively. Merge Sort, Start from two sorted runs of length 1, merge into a single run of twice the length. Repeat until a single sorted run is left. Merge sort needs N/2 extra buffer. Performance is second place on average, with quite good speed on nearly sorted array. Shell Sort, Sort every Nth element in an array using insertion sort. Repeat using smaller N values, until N = 1. On average, Shell sort is fourth place in speed. Shell sort may sort some distributions slowly.

Due to the constructions of Batcher [16]:
\[ T(2n) \leq T(n) + 1 \]  

This and Inequality (1) imply that:
\[ T(2^j) = j. \]  

Nakatani et al. [17] established that:
\[ T(i, j) \leq T(i) + T(j). \]  

The only prior technique that constructs Bitonic sorters of any width is due to Batcher and Liszka [18]. They show that:
\[ T(n) \leq \max \left( \left\lceil \frac{n}{2} \right\rceil, T \left( \left\lceil \frac{n}{2} \right\rceil \right) \right) + 2 \]

This and a straightforward induction imply:
\[ T(n) \leq 2 \lceil \log(n) \rceil - 1. \]

The first, non-trivial, lower bound on T(n) is due to Levy and Litman [14]. They showed that for every n that is not a power of two:
\[ \lceil \log(n) \rceil + 1 \leq T(n). \]

This result, combined with Equality (3), yields the surprising corollary that T is not monotonic. For example, \( T(15) \geq 5 \times 4 = T(16) \). As said, main result is the exact value of T(n). Namely, for every n:
\[ T(n) = 2 \lceil \log(n) \rceil - \lfloor \log(n) \rfloor. \]

In other words,
\[ T(n) = \begin{cases} \log(n) & \text{where } n \text{ is a power of two} \\ \lceil \log(n) \rceil + 1 & \text{otherwise} \end{cases} \]

Another model of oblivious computation, called min–max networks, was studied by Levy and Litman [14].
Let $T'(n)$ denote the minimal depth of a min–max network that sorts all Bitonic sequences of $n$ keys. The exact value of $T'(n)$ is almost known, as implied by the following arguments. The same reachability argument implies Inequality (1) also for min–max networks; therefore:

$$[\log(n)] \leq T'(n).$$  

(8)

Since every comparator network can be translated to a min–max network of the same depth, it follows that for every $n$:

$$T'(n) \leq T(n).$$

(9)

Inequalities (8), (9) imply that for every $n$:

$$[\log(n)] \leq T'(n) \leq [\log(n)] + 1.$$  

There are certain cases in which the exact value of $T'(n)$ is known, as listed below. The exact value of $T'(n)$ for other cases is yet unknown.

• $T'(n) = \log(n)$ when $n$ is a power of two. This follows from Inequalities (3), (8) and (9).

• $T'(n) = [\log(n)] + 1$, for every odd $n$. Levy and Litman established that $[\log(n)] + 1 \leq T'(n)$, for every odd $n$.

Inequality (9) provides the matching upper bound.

• $T'(n) = [\log(n)]$ for $n \in (10 \cdot 2N)$. This was established in.

• $T'(n) = [\log(n)]$ for $n \in (6 \cdot 2N)$, as shown in the next paragraph.

A min–max network, presented in [15], which is a Bitonic sort of 6 keys and of depth 3. Hence, $T'(6) = 3$. Due to this network, $T'(6 \cdot 2i) = 3 + i$ as follows. The techniques of Nakatani [17] and Batcher [16] are applicable also to min–max networks; hence, Inequalities (2) and (4) hold also for $T'$. Together with Inequality (8), we get that $T'(6 \cdot 2i) = 3 + i = [\log(6 \cdot 2i)]$. As discussed in [14], the above examples imply that min–max networks are sometimes strictly faster than comparator networks. Namely, there are infinitely many $n$'s with $T'(n) < T(n)$; this holds at least for any $n$ of the form $n=6\cdot2^i$ or $n=10\cdot2^i$. The work of this paper can be generalized in two directions. One direction considers the same computational problem but under a different model of computation. The above discussion on $T'$ follows this direction. Another direction keeps the same model of computation but considers harder computational problems. A natural generalization of our problem is the problem of sorting “multitonic sequences”, studied by Seiferas.

III. COMPARISON NETWORKS

Sorting networks are comparison networks that always sort their inputs, so it makes sense to begin the discussion with comparison networks and their characteristics.

A comparison network is composed solely of wires and comparators. A comparator, shown in Figure 1, is a device with two inputs, $x$ and $y$, and two outputs, $x'$ and $y'$, that performs the following function:

$$x' = \min(x, y),$$

$$y' = \max(x, y).$$

Because the pictorial representation of a comparator in Figure 1(a) is too bulky to represent, the convention of drawing comparators as single vertical lines shall be adopted, as shown in Figure 1(b). Inputs appear on the left and outputs on the right, with the smaller input value appearing on the top output and the larger input value appearing on the bottom output. Thus comparator can think as sorting its two inputs. It is assumed that each comparator operates in $O(1)$ time. In other words, It is assume that the time between the appearance of the input values $x$ and $y$ and the production of the output values $x'$ and $y'$ is a constant. A wire transmits a value from place to place. Wires can connect the output of one comparator to the input of another, but otherwise they are either network input wires or network output wires. Throughout this paper. It is assumed that a comparison network contains $n$ input wires $a_1, a_2, \ldots, a_n$ through which the values to be sorted enter the network, and $n$ output wires $b_1, b_2, \ldots, b_n$, which produce the results computed by the network. Also, the input sequence $\{a_1, a_2, \ldots, a_n\}$ and the output sequence $\{b_1, b_2, \ldots, b_n\}$ referring to the values on the
a comparison network, which is a set of comparators interconnected by wires[3]. A comparison network on \( n \) inputs is a collection of \( n \) horizontal lines with comparators stretched vertically. Note that a line does not represent a single wire, but rather a sequence of distinct wires connecting various comparators. The top line in Figure 2, for example, represents three wires: input wire \( a_1 \), which connects to an input of comparator \( A \); a wire connecting the top output of comparator \( A \) to an input of comparator \( C \); and output wire \( b_1 \), which comes from the top output of comparator \( C \). Each comparator input is connected to a wire that is either one of the network's \( n \) input wires \( a_1, a_2, \ldots, a_n \) or is connected to the output of another comparator. Similarly, each comparator output is connected to a wire that is either one of the network's \( n \) output wires \( b_1, b_2, \ldots, b_n \) or is connected to the output of another comparator. The main requirement for interconnecting comparators is that the graph of interconnections must be acyclic: if a path is traced from the output of a given comparator to the input of another to an output to an input, etc., the path traced must never cycle back on itself and go through the same comparator twice. Thus, as in Figure 2, a comparison network with network inputs on the left and network outputs on the right, data move through the network from left to right.

Figure 2: (a) A 4-input, 4-output comparison network, which is in fact a sorting network. At time 0, the input values shown appear on the four input wires. (b) At time 1, the values shown appear on the outputs of comparators \( A \) and \( B \), which are at depth 1. (c) At time 2, the values shown appear on the outputs of comparators \( C \) and \( D \), at depth 2. Output wires \( b_1 \) and \( b_2 \) now have their final values, but output wires \( b_3 \) and \( b_4 \) do not. (d) At time 3, the values shown appear on the outputs of comparator \( E \), at depth 3. Output wires \( b_3 \) and \( b_4 \) now have their final values.

Each comparator produces its output values only when both of its input values are available to it. In Figure 2(a), for example, suppose that the sequence \( \{ 9, 5, 2, 6 \} \) appears on the input wires at time 0. At time 0, then, only comparators \( A \) and \( B \) have all their input values available. Assuming that each comparator requires one unit time to compute its output values, comparators \( A \) and \( B \) produce their outputs at time 1; the resulting values are shown in Figure 2(b). Note that comparators \( A \) and \( B \) produce their values at the same time, or "in parallel."

Now, at time 1, comparators \( C \) and \( D \), but not \( E \), have all their input values available. One time unit later, at time 2, they produce their outputs, as shown in Figure 2(c). Comparators \( C \) and \( D \) operate in parallel as well. The top output of comparator \( C \) and the bottom output of comparator \( D \) connect to output wires \( b_1 \) and \( b_2 \), respectively, of the comparison network, and these network output wires therefore carry their final values at time 2. Meanwhile, at time 2, comparator \( E \) has its inputs available, and Figure 2(d) shows that it produces its output values at time 3. These values are carried on network output wires \( b_3 \) and \( b_4 \), and the output sequence \( \{ 2, 5, 6, 9 \} \) is now complete[4].

Under the assumption that each comparator takes unit time, the "running time" of a comparison network is defined, that is, the time it takes for all the output wires to receive their values once the input wires receive theirs. Informally, this time is the largest number of comparators that any input element can pass through as it travels from an input wire to an output wire. More formally, the depth of a wire is defined as follows. An
input wire of a comparison network has depth 0. Now, if a comparator has two input wires with depths \(dx\) and \(dy\), then its output wires have depth \(\max(dx, dy) + 1\). Because there are no cycles of comparators in a comparison network, the depth of a wire is well defined, and the depth of a comparator is defined to be the depth of its output wires. Figure 2 shows comparator depths. The depth of a comparison network is the maximum depth of an output wire or, equivalently, the maximum depth of a comparator.

A. Sorting Network

A sorting network is a comparison network for which the output sequence is monotonically increasing (that is, \(b_1 \leq b_2 \leq \ldots \leq b_n\) for every input sequence)[11]. Of course, not every comparison network is a sorting network, but the network of Figure 2 is. To see why, observe that after time 1, the minimum of the four input values has been produced by either the top output of comparator \(A\) or the top output of comparator \(B\). After time 2, therefore, it must be on the top output of comparator \(C\). A symmetrical argument shows that after time 2, the maximum of the four input values has been produced by the bottom output of comparator \(D\). All that remains is for comparator \(E\) to ensure that the middle two values occupy their correct output positions, which happens at time 3.

A comparison network is like a procedure in that it specifies how comparisons are to occur, but it is unlike a sorting network, which happens at time 3.

A comparison network is like a procedure in that it specifies how comparisons are to occur, but it is unlike a sorting network, which happens at time 3.

A comparison network is like a procedure in that it specifies how comparisons are to occur, but it is unlike a sorting network, which happens at time 3.

Lemma 1

If a comparison network transforms the input sequence \(a = \{a_1, a_2, \ldots, a_n\}\) into the output sequence \(b = \{b_1, b_2, \ldots, b_n\}\) then for any monotonically increasing function \(f\), the network transforms the input sequence \(f(a) = \{f(a_1), f(a_2), \ldots, f(a_n)\}\) into the output sequence \(f(b) = \{f(b_1), f(b_2), \ldots, f(b_n)\}\).

Proof: First prove the claim that if \(f\) is a monotonically increasing function, then a single comparator with inputs \(f(x)\) and \(f(y)\) produces outputs \(f(\min(x, y))\) and \(f(\max(x, y))\). Then use induction to prove the lemma.

To prove the claim, consider a comparator whose input values are \(x\) and \(y\). The upper output of the comparator is \(\min(x, y)\) and the lower output is \(\max(x, y)\). Suppose now apply \(f(x)\) and \(f(y)\) to the inputs of the comparator, as is shown in Figure 3. The operation of the comparator yields the value \(\min(f(x), f(y))\) on the upper output and the value \(\max(f(x), f(y))\) on the lower output. Since \(f\) is monotonically increasing, \(x \leq y\) implies \(f(x) \leq f(y)\). Consequently, the identities:

\[
\begin{align*}
\min(f(x), f(y)) &= f(\min(x, y)) \\
\max(f(x), f(y)) &= f(\max(x, y))
\end{align*}
\]

Figure 3: The operation of the comparator in the proof of Lemma 1. The function \(f\) is monotonically increasing.

Thus, the comparator produces the values \(f(\min(x, y))\) and \(f(\max(x, y))\) when \(f(x)\) and \(f(y)\) are its inputs, which completes the proof of the claim. Now induction can be used on the depth of each wire in a general comparison network to prove a stronger result than the statement of the lemma: if a wire assumes the value \(a_i\) when the input sequence \(a\) is applied to the network, then it assumes the value \(f(a_i)\) when the input sequence \(f(a)\) is applied. Because the output wires are included in this statement, proving it will prove the lemma. For the basis, consider a wire at depth 0, that is, an input wire \(a_i\). The result follows trivially: when \(f(a)\) is applied to the network, the input wire carries \(f(a_i)\). For the inductive step, consider a wire at depth \(d\), where \(d \geq 1\). The wire is the output of a comparator at depth \(d\), and the input wires to this comparator are at a depth strictly less than \(d\). By the inductive hypothesis, therefore, if the input wires to the comparator carry values \(a_i\) and \(a_j\) when the input sequence \(a\) is applied, then they carry \(f(a_i)\) and \(f(a_j)\) when the input sequence \(f(a)\) is applied. By our earlier claim, the output wires of this comparator then carry \(f(\min(a_i, a_j))\) and \(f(\max(a_i, a_j))\). Since they carry \(\min(a_i, a_j)\) and \(\max(a_i, a_j)\) when the input sequence is \(a\), the lemma is proved[20].

Theorem 1: (Zero-one principle)

If a comparison network with \(n\) inputs sorts all \(2^n\) possible sequences of 0's and 1's correctly, then it sorts all sequences of arbitrary numbers correctly[20].

Proof: Suppose for the purpose of contradiction that the network sorts all zero-one sequences, but there exists a sequence of arbitrary numbers that the network does not correctly sort. That is, there exists an input sequence \(\{a_1, a_2, \ldots, a_n\}\) containing elements \(a_i\) and \(a_j\) such that \(a_i < a_j\), but the network places \(a_j\) before \(a_i\) in
the output sequence. We define a monotonically increasing function \( f \) as follows. The word \( H \) of 0's in the output sequence when \( \{ a_1, a_2, ..., a_n \} \) is input, it follows from Lemma 1 that it places \( f(a_j) \) before \( f(a_i) \) in the output sequence when \( \{ f(a_1), f(a_2), ..., f(a_n) \} \) is input. But since \( f(a_j) = 1 \) and \( f(a_i) = 0 \), the contradiction is obtained that the network fails to sort the zero-one sequence \( \{ f(a_1), f(a_2), ..., f(a_n) \} \) correctly.

**Lemma 2**

Let each of \( q, r, s \) and \( t \) be either binary or undefined and let \( q + r \leq s + t \). Then \( \min(q, r) \leq \min(s, t) \) and \( \max(q, r) \leq \max(s, t) \). The following two lemmas concern the splitting of Bitonic words. Although the proofs of these lemmas are tedious, they are straightforward and therefore omitted.

**Lemma 3**

Let \( v \) be a Bitonic word and let \( v = (x, y) \). Then:

a. \( x \cdot y = v \) or \( x \cdot y = v \).

b. \( x \) is descending–ascending.

c. \( y \) is ascending–descending.

d. If \( v \) is 1-heavy then \( y \) is 1-heavy and \( 1 \leq y_1 + y_{|y|} \).

e. If \( x, y \) is ascending and \( n^0(x) \geq n^0(y) \); if, in addition, \( v \) is 1-heavy then \( x_1 + x_{|x|} \leq y_1 + y_{|y|} \).

**Lemma 4.**

Let \( v \) be descending–ascending word and let \( v = (x, y) \). Then:

a. \( y \) is ascending.

b. \( x_1 + x_{|x|} \leq y_1 + y_{|y|} \).

c. If \( x, y \) is \( LS(v) \) then \( x_{|x|} \leq y_1 \).

Define \( 0^k \) and \( 1^k \) as the words of width \( k \) that contain only 0's and only 1's, respectively. Define \( 0^k \dot{=} \{ 0^k | k \in \mathbb{N} \} \) and \( 1^k \dot{=} \{ 1^k | k \in \mathbb{N} \} \).

**Lemma 5.**

Let \( w \) be 1-heavy and Bitonic, let \( w = (a, b) \) and let \( b = (c, r) \). Then \( r \in 1^* \).

Proof. By Lemma 2(c, d), the word \( b \) is ascending–descending and \( 1 \leq b_1 + b_{|b|} \). Therefore, \( b \) is either ascending or descending. Again, by Lemma 2(d), \( b \) is 1-heavy. This clearly implies that \( r \in 1^* \).

**B. A Bitonic Sorting Network**

The first step in construction of an efficient sorting network is to construct a comparison network that can sort any bitonic sequence: a sequence that monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing[2]. For example, the sequences \{ 1, 4, 6, 8, 3, 2 \}, \{ 6, 9, 4, 2, 3, 5 \}, and \{ 9, 8, 3, 2, 4, 6 \} are all bitonic. As a boundary condition, it can be said that any sequence of just 1 or 2 numbers is bitonic. The zero-one sequences that are bitonic have a simple structure. They have the form \( 0^1 \) or \( 1^1 \) for some \( i, j, k \geq 0 \). Note that a sequence that is either monotonically increasing or monotonically decreasing is also bitonic[13][14][15]. The bitonic sorters that are constructed is a comparison network that sorts bitonic sequences of 0's and 1's.

**C. Bitonic Sorters Of Odd Width**

This section proves that, for every \( k \), \( C(2k - 1) \) is a Bitonic sorter[5][6]. The proof is indirect and builds on the similarity of \( C(2k - 1) \) to \( B(2k) \) and on the fact that \( B(2k) \) is a Bitonic sorter. In both networks, \( B(2k) \) and \( C(2k - 1) \), it is easy to verify that the key (if there is any) coming out of \( D \) or \( U \) is a minimal key or a maximal key, respectively. Hence, the core of our proof is the claim that the output of \( M \) is sorted. To this end, we show that, for every Bitonic (not necessarily 1-heavy) word \( w \) of width \( 2k - 1 \), there is a Bitonic word \( w' \) of width \( 2k \) with the following property. When \( C(2k - 1) \) processes \( w \) and \( B(2k) \) processes \( w' \), the sub-networks \( MC(2k - 1) \) and \( MB(2k) \) receive the same vector, \( MB(2k) = MC(2k - 1) \); hence, these two sub-networks produce the same vector. Since \( B(2k) \) is a Bitonic sorter, the output of \( MB(2k) \) is sorted.

Hence, the same holds for \( MC(2k - 1) \). To construct the above \( w' \), let \( t : \{ 0, 1 \}^* \rightarrow \{ 0, 1 \}^* \) be defined as follows. The word \( t(w) \) is derived from \( w \) by inserting a single 0 bit. This bit is inserted in the longest interval of 0's in \( w \). If \( w \) has several intervals of 0's of maximal length, the bit is inserted in the last interval. For example: \( t(00010010) = 000010010, t(00111000) = 00111000, t(11) = 110 \). For a word \( w \), let \( H^i(w) \) denote the number of 0's at the head of \( w \). Formally, \( H^i(w) \) is the largest \( i \) such that \( 0^i \) is a prefix of \( w \). Similarly, let \( H^i(w) \) be the largest \( i \) such that \( 1^i \) is a prefix of \( w \).
Lemma 6.
Let \( w \) be a Bitonic word and let \((x, y) = 5 \downarrow (w)\). Then either \( 5 \downarrow (t(w)) = (t(x), y) \) or \( 5 \downarrow (t(w)) = (\leftrightarrow(t(x)), y) \).

**Proof.** Let \( w' = t(w) \) and let \((x', y') = 5 \downarrow (w')\). We need to show that \( y' = y \) and that \( x' = t(x) \) or \( x' = \leftrightarrow(t(x)) \).

Clearly, \(|w'| = |w| + 1\) and \(n^i(w') = n^i(w) + 1\). By straightforward arithmetic \(|x'| = |x| + 1\) and \(|y'| = |y|\). We consider the following cases which are conclusive but not disjoint:

**Case 1:** \( w \) is empty. This case is trivial.

**Case 2:** \( w = 1^0 1^1 1^k \) for some \( i, j, k \geq 0 \).

By Lemma 3, \( y \) and \( y' \) are ascending. Clearly:
\[
n^i(y') = \min(\max(i, k), |y|) = n^i(y).
\]
Therefore, \( y = y' \). The facts that \( 5 \downarrow \) and \( 5 \downarrow \) are isometric and that \( y = y' \) imply that \( n^i(x') = n^i(x) \) and \( n^j(x') = n^j(x) + 1 \). By Lemma 4, both \( x \) and \( x' \) are descending–ascending. Hence, it remains to show that \( H^i(x) = H^i(x') \). In fact, both these numbers are \( \min(i, k) \).

**Case 3:** \( w = 0^i 1^0 1^k \) for some \( i, j, k \geq 0 \).

We first consider the words \( x \) and \( x' \). By duality and Lemma 3, these words are ascending. Clearly:
\[
H^i(x) + 1 = \min(\max(i, k), |x|) + 1 = \min(\max(i, k) + 1, |x'|) = H^i(x').
\]
(Not that the latter happens when \( x \in 1^* \); in the current case it implies that \( w \in 1^* \). Next, consider the words \( y \) and \( y' \). By previous arguments, \( y \) and \( y' \) are isometric. By Lemma 2(c), \( y \) and \( y' \) are ascending–descending. It remains to show that \( H^i(\leftrightarrow(y)) = H^i(\leftrightarrow(y')) \). In fact, both these numbers are \( \min(i, k) \).

D. The Reverse-cleaner

A bitonic sorter is composed of \( n/2 \) stages, each of which is called a **Reverse-cleaner**. Each reverse-cleaner is a comparison network of depth 1 in which input line \( i \) is compared with line \((n - i) + 1\) for \( i = 1, 2, ..., n/2 \). (It is assumed that \( n \) is even.) Figure 4 shows REVERSE-CLEANER[8], the Reverse-cleaner with 8 inputs and 8 outputs.

![Figure 4: The comparison network REVERSE-CLEANER[8]. A different sample zero-one input and output values are shown. The input is assumed to be bitonic. A Reverse-cleaner ensures that every output element of the top half is at least as small as every corresponding output element of the bottom half. Moreover, when a bitonic sequence of 0's and 1's is applied as input to a reverse-cleaner, the reverse-cleaner produces an output sequence in which smaller values are in the top half, larger values are in the bottom half, and both halves are bitonic. The next lemma proves these properties of reverse-cleaners](image)

**Lemma 7**

If the input to a reverse-cleaner is a bitonic sequence of 0's and 1's, then the output satisfies the following properties: both the top half and the bottom half are bitonic, every element in the top half is at least as small as corresponding element of the bottom half[8].

**Proof:** The comparison network REVERSE-CLEANER[\( n \)] shown in Figure 5 compares inputs \( i \) and \((n - i) + 1\) for \( i = 1, 2, ..., n/2 \). Without loss of generality, suppose that the input is of the form 00 ... 011 ... 100 ... 0. (The situation in which the input is of the form 11 ... 100 ... 011 ... 1 is symmetric.) There are three possible cases depending upon the block of consecutive 0's or 1's in which the reverse cleaner shows different behavior.
Figure 5: The possible comparisons in REVERSE-CLEANER[n]. The input sequence is assumed to be a bitonic sequence of 0's and 1's, and without loss of generality, it is assumed that it is of the form 00...011...100...0. We can think of the $n$ inputs as being divided into two halves such that for $i = 1, 2, ..., n/2$, inputs $i$ and $n-i+1$ are compared. (a) Case in which the sequence is already bitonic clean. (b) Case in which the input sequence is random. (c) Case in which the is bitonic clean but in reverse order. For all cases, every element in the top half of the output is at least as small as every element in the bottom half.

E. Knitting Sort Algorithm

In the proposed algorithm, First the number of elements must be even. If the input number is odd then introduced a new element in the array are assign $+\infty$ to it.

If(No. of Element $\%$ 2 $\neq$ 1)
{
    N$\leftarrow$N+1;
    A[N]$\leftarrow+\infty$;
}

when the number of elements would become even. Then the algorithm will work as follow.

Step 1: Compare 1st element with the Nth element, then 2nd element with N-1th element, Then 3rd element with N-2th element, and process will goes on till the first half is not compared with the second half.

Step 2: compare 1st element with 2nd element, Then 3rd element is compared with the 4th element and so on till (N-1)th element with Nth element. And simultaneously 2nd element with 3rd element, Then 4th element with 5th element and so on till (N-2)th element with (N-1)th element.

In the Knitting sort Algorithm there are two steps, First step is used to decrease the number of comparisons and decrease the complexity of the algorithm when the given array is already sorted in reverse order. The algorithm will also take minimum number of comparison when the array is already sorted in reverse order i.e.
O(1), without first step the algorithm will show the worst case complexity O(n^2) which is logically equivalent to bubble sort that's why this part is included to reduce the complexity and time consumption when array is sorted in decreasing order. Second Step is used to sort the given array followed by the first step. The algorithm will also take minimum number of comparison when the array is already sorted i.e. O(1).

Now, A concrete Algorithm is proposed as,

```
ALGORITHM:
KNITTING SORT(int array(),int low,int high)
{
    If(No. of Element % 2 !=1)
    {
        N=N+1;
        A[N]←+∞;
    }
    Low=1;
    High=n;
    while(low<high)
    {
        if(a[low]>a[high])
        {
            Exchange(a[low]←→ a[high])
        }
        Low←low+1;
        High←high+1;
    }
    while(!=Sort)
    {
        for(i=1 to n by +2 and j=2 to n-1 by +2)
        {
            if(a[i]>a[i+1])
            {
                Exchange(a[i]←→ a[i+1])
            }
            if(a[j]>a[j+1])
            {
                Exchange(a[j]←→ a[j+1])
            }
        }
    }
}
```

Diagrammatically, the implementation steps are shown in Figure 6 as:

![Diagram](image-url)
F. Performance Graph of Knitting Sort

![Performance Graph of Knitting Sort](image)

Figure 7. Performance of Knitting Sort

G. Algorithm Complexity

Step 1: \( n \)

Step 2: \( n \)

Total Complexity: \( n+n=2n \approx O(n) \)

IV. EXPERIMENTAL TEST AND RESULT

An array of 10 elements is used to test the algorithm which is shown in the Figure 8, then the Knitting sort algorithm is applied in 3 different cases, firstly when the array is random i.e. not sorted at any manner, secondly when the array is sorted in reverse order and lastly when the array is already sorted, also the no. of steps algorithm takes are mentioned in table with the respected values of comparison and swapping.

**ARRAY:**

<table>
<thead>
<tr>
<th>Index (i) values</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12</td>
<td>10</td>
<td>6</td>
<td>99</td>
<td>29</td>
<td>39</td>
<td>199</td>
<td>20</td>
<td>19</td>
<td>56</td>
</tr>
</tbody>
</table>

Figure 8. Array used to implement Knitting Sort

**A. Array sorted in reverse order**

<table>
<thead>
<tr>
<th>Array Steps</th>
<th>199</th>
<th>99</th>
<th>56</th>
<th>39</th>
<th>29</th>
<th>20</th>
<th>19</th>
<th>12</th>
<th>10</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>10</td>
<td>12</td>
<td>19</td>
<td>20</td>
<td>29</td>
<td>39</td>
<td>56</td>
<td>99</td>
<td>199</td>
</tr>
</tbody>
</table>

**B. Array is not sorted**

<table>
<thead>
<tr>
<th>Array Steps</th>
<th>12</th>
<th>10</th>
<th>6</th>
<th>99</th>
<th>29</th>
<th>39</th>
<th>199</th>
<th>20</th>
<th>19</th>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>6</td>
<td>12</td>
<td>29</td>
<td>39</td>
<td>99</td>
<td>20</td>
<td>19</td>
<td>56</td>
<td>199</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>12</td>
<td>29</td>
<td>39</td>
<td>20</td>
<td>19</td>
<td>56</td>
<td>99</td>
<td>199</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>10</td>
<td>12</td>
<td>29</td>
<td>20</td>
<td>19</td>
<td>39</td>
<td>56</td>
<td>99</td>
<td>199</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>10</td>
<td>12</td>
<td>29</td>
<td>19</td>
<td>39</td>
<td>56</td>
<td>99</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>10</td>
<td>12</td>
<td>19</td>
<td>20</td>
<td>29</td>
<td>39</td>
<td>56</td>
<td>99</td>
<td>199</td>
</tr>
</tbody>
</table>
C. Array is already sorted

TABLE III. FOR ALREADY SORTED ELEMENTS

<table>
<thead>
<tr>
<th>Array Steps</th>
<th>6</th>
<th>10</th>
<th>12</th>
<th>19</th>
<th>20</th>
<th>29</th>
<th>39</th>
<th>56</th>
<th>99</th>
<th>199</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>10</td>
<td>12</td>
<td>19</td>
<td>20</td>
<td>29</td>
<td>39</td>
<td>56</td>
<td>99</td>
<td>199</td>
</tr>
</tbody>
</table>

D. Comparison with various sorting techniques at Average Case time analysis

TABLE IV. COMPARISON WITH OTHER SORTING TECHNIQUES

<table>
<thead>
<tr>
<th>Steps Taken (No. of elements)</th>
<th>Knitting Sort</th>
<th>Merge Sort</th>
<th>Bubble Sort</th>
<th>Insertion Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
<td>30</td>
<td>100</td>
<td>30</td>
</tr>
<tr>
<td>20</td>
<td>9</td>
<td>80</td>
<td>400</td>
<td>80</td>
</tr>
<tr>
<td>50</td>
<td>21</td>
<td>250</td>
<td>2500</td>
<td>250</td>
</tr>
<tr>
<td>100</td>
<td>49</td>
<td>600</td>
<td>10000</td>
<td>600</td>
</tr>
<tr>
<td>200</td>
<td>137</td>
<td>1400</td>
<td>40000</td>
<td>1400</td>
</tr>
<tr>
<td>500</td>
<td>417</td>
<td>4000</td>
<td>250000</td>
<td>4000</td>
</tr>
<tr>
<td>1000</td>
<td>785</td>
<td>9000</td>
<td>1000000</td>
<td>9000</td>
</tr>
</tbody>
</table>

Figure 9. Comparison with other sorting techniques

# Due to conveniently compare the various sorting technique in Bar chart Bubble Sort is not shown in it. It will make the Graph Chart inconvenient to compare.

V. CONCLUSION

Searching is very important problem in database query optimization. To make searching convenient they must be sorted. There are various sorting techniques which will be helpful in various ways, according to the need. The problem with already existing sorting techniques are that either they are not stable or they are not inplace, and If they have both features (inplacability and stability) then their time complexity is very high. In this paper, A new algorithm is presented and implemented which is stable and inplace algorithm to sort an ordered list of items with stability and having inplace convenience with Average case complexity $O(n)$. Comparators and bitonic sorting network is used to implement the algorithm. An experimental result shown in Figure 7 and performance graph shown in Figure 9 comparing the performance of Knitting Sort with
bubble sort[7][10] and Merge Sort is also shown[19]. This constitute a significant improvement over the unstable and not inplace sorting techniques or algorithms available in the literature.

REFERENCES

[3] OPTIMAL CONCLUSIVE SETS FOR COMPARATOR NETWORKS Guy Evena, Tamir Levi b, Ami Litman b a School of Electrical Engineering, Tel-Aviv University, Tel-Aviv 69978, Israel b Faculty of Computer Science, Technion, Haifa 32000, Israel , Theoretical Computer Science 410 (2009) 1369_1376
[19] "Name : Jariya Phongsai Class : MAC 286. Data Structure "
http://www.eportfolio.lagcc.cuny.edu/scholars/doc_f09/eP_f09/Jariya.0%20Phongsai...