A New Sorting Algorithm with filling to the left and right

N. Vasilev and A. Bosakova-Ardenska

***Abstract*— This paper presents an algorithm for sorting by using of LIT (left inversions table). The algorithm is named LR. The time complexity of the proposed algorithm analytically evaluated. Two approaches for acceleration of LR are presented. The proposed algorithm and its two improvements are implemented in C++. Experimental comparisons are done between LR and some known algorithms, and between LR and its two modifications. The experiments show that LR is faster than "bubble sort" and "LtoRA" algorithms but it is slower than the algorithms "insertion sort" and "selection sort". The experiments also show that for rows in which there is a large number of the repetitions, the modification "LR – repeat" is faster than the original algorithm, "Bubble sort", "Selection sort" and the modification "LRA – minimax". The algorithm "LR minimax" is faster than algorithm LR in all cases (when the row has large or small number of repetitions).**

***Index Terms*— Sorting algorithm, Left Inversions Table, Insertion sort, Selection sort**

I. INTRODUCTION AND AIMS

THIS paper is a continuation of the work in article [7]. The two papers are part of a research on some sorting methods and the possibility for their improvement.

It is considered that near 25% of the work of the computer systems is used for sorting of information [1]. This shows how important it is to find good sorting methods and algorithms. There are many methods and algorithms for sorting and they are studied widely [1,2,3,4,5,6]. This doesn't mean that everything in this area is finished and nothing new and better could be found, especially considering the characteristics of the given row.

The aim of this paper is:

- to propose and investigate a method for sorting of rows by a left inversions table (LIT) with left and right filling which is an improvement of the sorting methods proposed in [7];
- to evaluate the complexity of the proposed method;
- to make a program for the proposed method (algorithm) for row sorting by LIT with right and left filling;
- to evaluate and compare experimentally the proposed method: sorting by LIT with right and left filling.

The methods for row sorting considered in [7] are:

- sorting by LIT with filling from left to right;
- sorting by LIT with filling from right to left.

They are based on the proven assertion: the table of the left inversions by positions of a given row a_j (j = 1,2,…,n) uniquely defines the sorted ascending or descending row.

LIT of a given row is the sequence of numbers in the j-th position in which the number of the elements $\mathbf{d_j}$ is written, left from $\mathbf{a_j}$ (**j**-th element, **j** = 1,2,...,**n**) and lager than it.

The steps for sorting by LIT are the following:

1) constructing the LIT of the given row by counting the larger elements from the left of every element in the row;
2) constructing the searching row.

Constructing the searching row begins with the element in the first position of the given row, continues with the second element and so on, until the element in the n-th position. The position for recording of the elements and the number of moved elements depending on the desired sorting, and the direction of moving are shown in Table 1.

TABLE 1. POSITION FOR RECORD OF AJ, J = 1,2,...,N, AND NUMBER OF MOVED ELEMENTS FOR SORTING WITH FILLING FROM LEFT TO RIGHT AND FROM RIGHT TO LEFT

Filling	Position for record of a_j		Number of the moved elements	
	Row		Row	
	Ascending	Descending	Ascending	Descending
To the right	$j - d_j$	$d_j + 1$	d_j	$j - d_j - 1$
To the left	$n - d_j$	$n - (j - d_j) + 1$	$j - d_j - 1$	d_j

TABLE 2. TABLE OF LEFT INVERSIONS OF THE GIVEN ROW

Position **j** in the given row	1	2	3	4	5	6	7	8	9
Value of the element $\mathbf{a_j}$	60	40	70	20	50	30	10	90	80
$\mathbf{d_j}$ – num. of the elements from the left bigger then $\mathbf{a_j}$	0	1	0	3	2	4	6	0	1

Naiden Borisov Vasilev was head of department of Computer Systems and Technologies in Technical University Sofia, branch in Plovdiv, Bulgaria. (e-mail: mnvasilev@yahoo.com).

Atanaska Dimitrova Bosakova-Ardenska works in University of Food Technologies, Plovdiv in Bulgaria. She is associated professor in department of Computer Systems and Technologies (e-mail:a_bosakova@uft-plovdiv.bg).

Example

Sort in an ascending and a descending order the following row: 60, 40, 70, 20, 50, 30, 10, 90, 80.

We construct the LIT of the row (Table 2).

	a	b
	60	60
	40 60	60 40
	40 60 70	70 60 40
	20 40 60 70	70 60 40 20
	20 40 50 60 70	70 60 50 40 20
	20 30 40 50 60 70	70 60 50 40 30 20
	10 20 30 40 50 60 70 90	70 60 50 40 30 20 10
	10 20 30 40 50 60 70 80 90	90 80 70 60 50 40 30 20 10
Positions:	01 02 03 04 05 06 07 08 09	01 02 03 04 05 06 07 08 09

Fig.1. Ascending (a) and descending (b) rows of the given row sorted from left to right

Figure 1a shows the ascending row and figure 1b – the descending row with filling from left to right. The moved elements are underlined.

The number of the moves for constructing the ascending row is 17. The number of the moves for constructing the descending row is 19.

If t_{11} is the time for execution of the comparison operation on the computer, t_{21} – the time for execution of the increment and record operations, t_{22} – the time for execution of the record operation (move to the right), then:

– the time for constructing the ascending row by sorting with right filling and the descending row by sorting with left filling, without record operations in temporary positions of the array and intermediate saving of the elements will be:

$$T_r^a = T_l^d = t_{11}n(n-1)/2 + (t_{21} + t_{22})\sum_{j=1}^{n} d_j$$

– the time for constructing the descending row by sorting with right filling and the ascending row by sorting with left filling, without record operations in temporary positions of the array and intermediate saving of the elements will be:

$$T_r^d = T_l^a = t_{11}n(n-1)/2 + t_{21}\sum_{j=1}^{n} d_j + t_{22}\sum_{j=1}^{n}(j - d_j - 1)$$

As seen above, the complexity of the proposed algorithms is quadratic. We will look for ways to reduce the number of comparisons and moves.

II. Reducing the Number of Comparisons

We will consider two ways for reducing the number of comparisons proposed in [7]. These methods decrease the value of the first addend in the equations above for some rows with certain properties. In other words, the efficiency of the proposed methods depends on the characteristics of the given row and sometimes they do not reduce the number of comparisons.

Approach 1. Comparisons with the current minimal and current maximal elements

When comparing the elements for constructing the LIT, two fields are used. In these fields the minimal (a_{min}) and the maximal (a_{max}) values are written among the elements with which the current element is compared. When we count the left elements larger than a_j, first we compare it with the minimal and the maximal values.

If $a_j < a_{min} = a_i$, i=1,2,...,j-1 the value d_j will be j-1 because all elements to the left of a_j are larger than it. The comparisons of a_j with the left elements are not done.

If $a_j \geq a_{max} = a_i$, i=1,2,...,j-1 the value d_j will be 0 because to the left of a_j there are not elements larger than it. The comparisons of a_j with the left elements are not done.

This approach is effective for rows in which the small and the large elements are in the end of the rows. It is most effective (0 comparisons) for the rows, which can be split from left to right into two rows – an ascending one and a descending one, for which the smallest element of the ascending row is larger than the largest element of the descending row. The ascending and the descending rows are also such rows. The number of these rows is 2^{n-1}. (The values in LIT for such rows are: 0 in the first position and 0 or j-1 in the other positions.)

If the minimal and the maximal elements are in the first two position of the given row, the number of the comparisons will not be reduced.

This approach adds at most 2n comparisons. First, the element a_j is compared to element a_{j-1}. If $a_j > a_{j-1}$ then a_j will be compared to a_{max} only. If $a_j < a_{j-1}$ then a_j will be compared to a_{min} only. If $a_j \geq a_{min}$ or $a_j < a_{max}$ there will be only one more comparison. Otherwise the added comparisons will be 2. If $a_j = a_{j-1}$ then comparisons to a_{min} and a_{max} are not done. For avoiding repeated comparisons, the approach 2 is used.

Approach 2. Avoid repeated comparisons

If the row has repeated elements, it is desirable to avoid repeated comparisons.

If $a_i = a_j$, i<j, i=1,2,...,n-1, j=2,3,...,n, then the value d_i+d_t is assigned to d_j, where d_t is the current value of d_j. Thus, one comparison and one summation are added, but (i-1) repeated comparisons are avoided for determination of d_j.

This approach adds n(n-1)/2 comparisons. It is efficient when the number of the repeated elements is large, the same elements are near and they are to the right in the given row.

III. Reducing the Number of the Moves

To reduce the number of the moves we will unite both methods (algorithms): filling from left to right and filling from right to left [7]. In this case, the record of each element $\mathbf{a_j}$ (j = 1,2, .., n) will be done to the left or right of the row depending on the result of the comparison of the values $\mathbf{d_j}$ and $\mathbf{j-d_j-1}$.

If $\mathbf{d_j \leq j-d_j-1}$ the moves for ascending row will be done to the **right** and for descending row will be done to the **left**.

If $\mathbf{d_j > j-d_j-1}$ the moves for ascending row will be done to the **left** and for descending` row will be done to the **right**.

Thus, the number of the moves for sorting the row will be minimal. The size of the array for saving the sorted row must be **(2n-1)**. The first element will be written in position **n**. By filling of the array we must keep current indices of nearest left (**L**) and nearest right (**R**) vacant positions in the array.

Position for record $\mathbf{p_j}$ of the current element $\mathbf{a_j}$ is:

$\mathbf{p_j = R-d_j}$ when $\mathbf{d_j \leq j-d_j-1}$ and $\mathbf{p_j = L+j-d_j-1}$ when $\mathbf{d_j > j-d_j-1}$ for ascending row;

$p_j = L+d_j$ when **$d_j \le j-d_j-1$** and **$p_j = R-(j-d_j-1)$** when **$d_j > j-d_j-1$** for descending row.

We will construct the ascending row of the row in the example above.
j=1; the first element a_1 is written in position $p_1 = n = 9$.
j=2; L=8, R=10; $d_2 = 1 > 2-1-1 = 0 = 2-d_2-1$; a_2 is written to the left; $p_2 = 8+2-1-1 = 8$.
j=3; L=7, R=10; $d_3 = 0 < 3-0-1 = 2 = 3-d_3-1$; a_3 is written to the right; $p_3 = 10-0 = 10$.
j=4; L=7, R=11; $d_4 = 3 > 4-3-1 = 0 = 4-d_4-1$; a_4 is written to the left; $p_4 = 7+4-3-1 = 7$.
j=5; L=6, R=11; $d_5 = 2 = 5-2-1 = 2 = 5-d_5-1$; a_5 is written to the right; $p_5 = 11-2 = 9$.

The elements with indices 1 (60) and 3 (70) are moved to the right. They are written in positions 10 and 11.
j=6; L=6, R=12; $d_6 = 4 > 6-4-1 = 1 = 6-d_6-1$; a_6 is written to the left; $p_6 = 6+6-4-1 = 7$.

The element with index 4 (20) is moved to the left. It is written in position 6.
j=7; L=5, R=12; $d_7 = 6 > 7-6-1 = 0 = 7-d_7-1$; a_7 is written to the left; $p_7 = 5+7-6-1 = 5$.
j=8; L=4, R=12; $d_8 = 0 < 8-0-1 = 7 = 8-d_8-1$; a_8 is written to the right; $p_8 = 12-0 = 12$.
j=9; L=4, R=13; $d_9 = 1 < 9-1-1 = 7 = 9-d_9-1$; a_9 is written to the right; $p_9 = 13-1 = 12$.

The element with index 8 is moved to the right. It is written in position 13.

Figures 2a and 2b present the construction of the ascending and the descending rows sorted by LIT with filling to the left and right. The moved elements are underlined.

					60								
				40	60								
				40	60	70							
			20	40	60	70							
			20	40	50	60	70						
		20	30	40	50	60	70						
	10	20	30	40	50	60	70						
	10	20	30	40	50	60	70	90					
	10	20	30	40	50	60	70	80	90				
Positions:	05	06	07	08	09	10	11	12	13				
					a								

					60								
					60	40							
				70	60	40							
				70	60	40	20						
			70	60	50	40	20						
			70	60	50	40	30	20					
			70	60	50	40	30	20	10				
		90	70	60	50	40	30	20	10				
	90	80	70	60	50	40	30	20	10				
Positions:	05	06	07	08	09	10	11	12	13				
					b								

Fig.2. Ascending (a) and descending (б) orders of the given row sorted by LIT with filling to the left and right

We will note that:
- if $d_j = 0$, a_j is written to the right after the last element in the row;
- if dj = j-1, aj is written to the left before the first element in the row.

In both cases, moves of elements are not executed.
The number of the moves for constructing the ascending and the descending rows is the same and its value is 4.
The results are summarized in Table 3.

TABLE 3. POSITIONS FOR RECORDING AND DIRECTION OF FILLING BY SORTING TO THE LEFT AND RIGHT

Row	Direction of filling		Position for record of a_j	
	$d_j \le j-d_j-1$	**$d_j > j-d_j-1$**	**$d_j \le j-d_j-1$**	**$d_j > j-d_j-1$**
Asc.	To the right	To the left	**$R-d_j$**	**$L+j-d_j-1$**
Desc.	To the left	To the right	**$L+d_j$**	**$R-(j-d_j-1)$**

IV. EVALUATION OF THE COMPLEXITY OF THE ALGORITHM FOR SORTING BY LIT WITH FILLING TO THE LEFT AND RIGHT

The operations which are used in the proposed algorithm are:
1) compare for constructing of the LIT (for counting of the larger elements); the elements of the array are compared;
2) record (increment) for counting of the larger elements;
3) compare to determine the direction of the minimal move (left or right); the first operand is number and the second one is an expression which contains subtraction (see table 3);
4) record (move one position to the left or to the right) of the elements of the array to construct the current sorted row;
5) record (move) of the elements in temporary positions in the array for constructing the current sorted row; the record position is a result of addition/subtraction (see table 3);
6) record (increment) of nearest vacant left (L) and right (R) positions in the array for sorted row.

Number of the compares for count the bigger elements (left inversions) by position is:
$C = n(n-1)/2$.
Number of the records (increments) by counting the bigger elements is:

$$R_{lr} = \sum_{j=1}^{n} d_j.$$

His values [7] are:
- minimal– **0** (the row is ascending);
- average– **$n(n-1)/4$**;
- maximal– **$n(n-1)/2$** (the row is descending).

The number of compares to determine the direction of minimal move (left or right) is equal to **n-1**.
The number of records (moves one position to left or right) is:

$$S_{lr} = \sum_{j=1}^{n} \min(d_j, j - d_j - 1)$$

The minimal number of the moves is 0. For rows which has d_j= 0 or j-1, j = 1,2,..,n, operations move aren't done. Every such row can be divided (from left to right) in two rows: one increasing row and one decreasing row like the smallest element in the increasing row is bigger than the biggest element in the decreasing row. Ascending and descending rows are such rows. When sort increasing row the fill up are to the right only. When sort decreasing row the fill up are to the left only. The number of these rows is 2^{n-1}. For n=4, the number of these rows is 2^{4-1}=8. These rows have LIT: 0000, 0100, 0020, 0003, 0120, 0103, 0023 and 0123. (See table 4.)

The maximal number of the moves is: $\mathbf{(n^2-2n)/4}$ when **n** is even and $\mathbf{(n^2-2n+1)/4}$ when **n** is odd. This value is obtained when the LIT are: 0, 0 or 1, 1, 1 or 2, 2, 2 or 3,..., n/2-1 , n/2-1 or n/2 if **n** is even and 0, 0 or 1, 1, 1 or 2, 2, 2 or 3,..., (n-1)/2-1 or (n-1)/2, (n-1)/2 if **n** is odd. The number of these rows is $2^{n/2}$ if **n** is even and $2^{(n-1)/2}$ if **n** is odd. When n = 4, the number of these rows is 4. The LIT of these rows are: 0011, 0012, 0111 and 0112. The maximal number of the moves is 2. (See table 4.)

The number of records (moves) of the elements in temporary positions in the array for constructing of the current sorted row is always **n** (a new array).

The number of records (increments) of the nearest left (L) vacant position and nearest right (R) vacant position in the array is **n**.

We note that the number of the operations (including the moves) for constructing the ascending and the descending row is the same.

The time for constructing the sorted row with filling to the left and right will be:

$$T_{lr} = t_{11}n(n-1)/2 + t_{21}\sum_{j=1}^{n} d_j + t_{12}(n-1) + t_{22}\sum_{j=1}^{n} \min(d_j, j-d_j-1) + t_{23}n + t_{21}n$$

t_{11} is the time for comparing two elements of the array;

t_{21} is the time for incrementing and recording;

t_{12} is the time for comparing with the first operand being an number, and the second one being a value of the operation subtraction;

t_{22} is the time for recording (moving to the left or right) of a element of the array;

t_{23} is the time for additions/subtractions and recording of the result.

The minimal and maximal time for constructing the sorted row by LIT with filling to the left and right will be:

$$\mathbf{T_{lr\_min} = t_{11}n(n-1)/2 + t_{12}(n-1) + t_{23}n + t_{21}n = K}$$

$$\mathbf{T_{lr\_max} = K + \max(t_{21}n(n-1)/2, t_{21}n^2/4 + t_{22}(n^2-2n)/4)}$$

$$\mathbf{T_{lr\_max} = K + \max(t_{21}n(n-1)/2, t_{21}(n^2-1)/4 + t_{22}(n^2-2n+1)/4)}$$

The first expression for T_{lr_max} is for n even and the second expression is for n odd.

$n^2/4$ is maximal number of recordings (increments) for the rows with maximal number of moves for n even. $(n^2 - 1)/4$ is maximal number of recordings (increments) for the rows with maximal number of moves for n odd.

The time T_{lr_min} is for the ascending row.

The time T_{lr_max} is for the rows with LIT: 0,1,1,2,2,3,.....,n/2-1,n/2-1, n/2 when **n** is even and 0, 1, 1, 2, 2, 3,.....,(n-1)/2-1, (n-1)/2-1, (n-1)/2, (n-1)/2 when **n** is odd or for the descending row.

The number of operations in the second operand of the addend "max" is equal to the number of operations in the first operand: n(n-1)/2. So, the value of max will be determined by the ratio of the values of t_{21} and t_{22}.

Table 4 shows the relationship between operations "move" in the discussed methods for sorting for n = 4.

The first column contains all rows (permutations) of the elements of the set {1,2,3,4}.

In the second column the tables of the left inversions (LIT) for every row are constructed.

In each cell of the third column the moves to the right are sequentially written and summed for the first, second, third and fourth elements of the row (in the cell of the first column) for constructing the ascending row with filling to the right. It can be seen that the numbers (digits) in the second and third column are the same.

In each cell of the fourth column the moves to the left are sequentially written and summed for the first, second, third and fourth elements of the row (in the cell of the first column) for constructing the ascending row with filling to the left. It is seen that the sum of the corresponding digits in the third and fourth columns is equal to the position number of digits minus one.

TABLE 4. MOVES IN THE THREE METHODS FOR SORTING BY LIT FOR N=4.

Row (n=4)	LIT	Moves		
		To the right	To the left	Left and right
1234	0000	0+0+0+0=0	0+1+2+3=6	0+0+0+0=0
1243	0001	0+0+0+1=1	0+1+2+2=5	0+0+0+1=1
1342	0002	0+0+0+2=2	0+1+2+1=4	0+0+0+1=1
2341	0003	0+0+0+3=3	0+1+2+0=3	0+0+0+0=0
1324	0010	0+0+1+0=1	0+1+1+3=5	0+0+1+0=1
1423	0011	0+0+1+1=2	0+1+1+2=4	0+0+1+1=2
1432	0012	0+0+1+2=3	0+1+1+1=3	0+0+1+1=2
2431	0013	0+0+1+3=4	0+1+1+0=2	0+0+1+0=1
2314	0020	0+0+2+0=2	0+1+0+3=4	0+0+0+0=0
2413	0021	0+0+2+1=3	0+1+0+2=3	0+0+0+1=1
3412	0022	0+0+2+2=4	0+1+0+1=2	0+0+0+1=1
3421	0023	0+0+2+3=5	0+1+0+0=1	0+0+0+0=0
2134	0100	0+1+0+0=1	0+0+2+3=5	0+0+0+0=0
2143	0101	0+1+0+1=2	0+0+2+2=4	0+0+0+1=1
3142	0102	0+1+0+2=3	0+0+2+1=3	0+0+0+1=1
3241	0103	0+1+0+3=4	0+0+2+0=2	0+0+0+0=0
3124	0110	0+1+1+0=2	0+0+1+3=4	0+0+1+0=1
4123	0111	0+1+1+1=3	0+0+1+2=3	0+0+1+1=2
4132	0112	0+1+1+2=4	0+0+1+1=2	0+0+1+1=2
4231	0113	0+1+1+3=5	0+0+1+0=1	0+0+1+0=1
3214	0120	0+1+2+0=3	0+0+0+3=3	0+0+0+0=0
4213	0121	0+1+2+1=4	0+0+0+2=2	0+0+0+1=1
4312	0122	0+1+2+2=5	0+0+0+1=1	0+0+0+1=1
4321	0123	0+1+2+3=6	0+0+0+0=0	0+0+0+0=0

In each cell of the fifth column the moves to the left and right are sequentially written and summed for the first, second, third and fourth elements of the row (in the cell of the first column) for constructing the ascending row with filling to the left and right. It is seen that the value of each digit is equal to the value of the smaller of the corresponding digits in the third and fourth columns. The smaller digit determines the direction of the filling.

The maximal number of moves for sorting with filling to the left and right is more than two times smaller than the sorting with filling from left to right.

For example, when n=1000, the maximal number of the moves

for constructing the ascending row with filling from left to right is 499500, and the average is 249750. When the filling is to the left and right the maximal number of the moves is 249500. This can be seen in table 4. When n = 4, these values are respectively 6, 3 and 2.

In comparison with sorting by LIT with filling from left to right [7] sorting by LIT with filling to the left and right use twice more memory and has additional operations: **n-1** comparisons to determine the minimal number of moves; **n** recordings of the nearest left (**L**) or nearest right (**R**) vacant positions.

V. Experimental results

The proposed methods for sorting which use LIT are:

- filling to the left and right;
- filling to the left and right and comparisons with the current minimal and current maximal elements;
- filling to the left and right with avoiding repeated comparisons.

These methods are implemented in C++. We will name them LR, LR minimax, and LR repeat, respectively.

The aims of the experimental work is the following:

1. To compare the execution times of the proposed algorithms with some known algorithms. We use the algorithms: Bubble, Insertion sort, Selection sort and LtoRA [7].
2. To verify the influence of the proposed improvements (LR minimax, LR repeat) on the execution time of the investigated algorithm (LR).

For experiments a computer system is used with processor Intel Celeron E3300 2,5GHz, RAM 2,96 GB. The elements of the rows are generated by the functions rand() and srand().

The number of the elements of the rows for these experiments is from 2000 to 8000. The elements are integer numbers. Each algorithm sorts 10 times 4 different rows with number of the elements 2000, 3000, 4000, 5000, 6000, 7000 and 8000. The execution time is obtained after averaging the corresponding execution times.

Experiment 1. Sorting by the algorithms: LtoRA, LR, LR minimax, LR repeat, Bubble, Insertion sort and Selection sort integer numbers from 0 to 32767 (a low repetition rate for the elements).

The average times (ms) are shown in Table 5 and Fig.3.

In parentheses is shown the number of operations for sorting the row with algorithms LR, LR minimax and LR repeat. The number of operations is counted with a program.

Working time of the program cannot be considered as a reliable estimation because computers work in a multiprogramming mode and there is no guarantee that the tested program is not interrupted, which could increase its execution time. Also, the execution time is influenced by the memory organization.

As a result, the time for sorting of the same rows can be different. This is why, experimenting with the algorithms LR, LR repeat, LR minimax, first the number of operations for sorting the rows is counted and the obtained values are used as criteria for correct time results. Fig.5 shows result of counting the operations for sorting the row with 4000 elements with algorithm LR repeat (low repetition rate for the elements).

TABLE 5. AVERAGE TIMES (MS) FOR SORTING ROWS WITH LOW REPETITION RATE FOR THE ELEMENTS

Elem. number / Algorithm	2000	3000	4000	5000	6000	7000	8000
LtoRA	19,9	38,6	78	123,5	177,8	236	311,1
LR	15,2 (8527436)	35,5 (19113969)	68,8 (34150391)	104,7 (53386151)	154,5 (76723619)	207,5 (10428 6373)	273,4 (13615 8382)
LR mini max	15,1 (8512421)	35,5 (19091956)	67,3 (34118041)	104,6 (53353315)	153 (76674177)	206 (10423 6432)	271,9 (13610 7945)
LR repeat	21,5 (9322687)	45,8 (20867827)	81 (36877231)	120,5 (56958461)	174,7 (81033635)	239,3 (10985 6137)	304,9 (14248 2171)
Bubble	46,2	106,1	182,5	285,6	415,1	562,6	742,5
Insert. sort	7,5	9,1	15,5	31,1	43,3	56,3	78
Select. sort	9,1	12,2	25,2	43,1	59,1	78	104,6

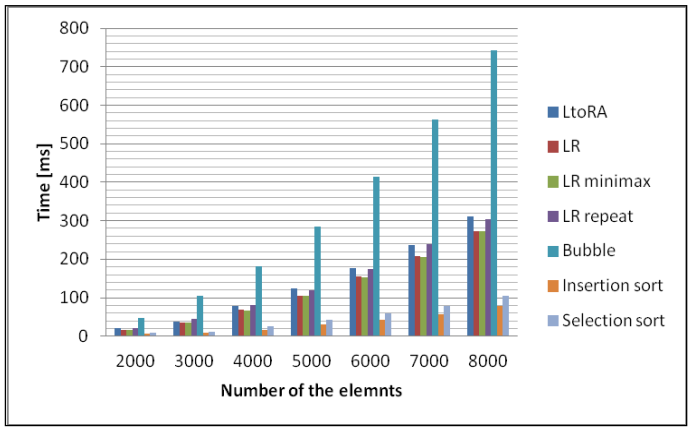


Fig.3 Average times for sorting the rows with low repetition rate for the elements

The experiments show that:

1) algorithm LR is faster than algorithms LtoRA and Bubble sort but it is slower than algorithms Insertion sort and Selection sort;
2) algorithm "LR minimax" reduced the number of operations (the time) compared with the operations (the time) of the algorithm LR;
3) algorithm "LR repeat" increases the number of operations (the time) compared with the operations (the time) of algorithm LR; the number of added operations "compare" is equal to the not inversed and equal pairs of elements in the row; if there are no equal elements, no comparisons are avoided.

Experiment 2. Sorting by the algorithms: LtoRA, LR, Bubble, Insertion sort and Selection sort for integer numbers with high repetition rate for the elements – 10 repetitions on average.

The average times (ms) are shown in Table 6 and Fig.4.

TABLE 6. AVERAGE TIMES (MS) FOR SORTING ROWS WITH HIGH REPETITION OF THE ELEMENTS

Elem. no / Algorithm	2000	3000	4000	5000	6000	7000	8000
LtoRA	20	43,3	76,5	122	177,7	239,2	308
LR	15,3 (8521 897)	32,5 (1918 3809)	63,8 (3392 7700)	106,2 (5314 1336)	154,5 (7660 5673)	209,1 (10403 4350)	272 (13638 8051)
LR minimax	15,3 (8481 391)	32,5 (1907 2369)	63,7 (3379 7053)	106,1 (5293 6231)	153 (7638 5661)	209 (10374 3277)	270,5 (13609 9578)
LR repeat	6 (2969 425)	7,5 (6665 277)	20,2 (1171 2696)	32,5 (1843 2564)	51,4 (2657 8738)	70,4 (3615 8709)	90,2 (4738 2641)
Bubble	44,7	106	181	285,5	415,3	554,8	747
Insertion sort	4,5	12	18,6	31	43,3	60,9	78
Selection sort	7,5	13,9	25	37	49,8	79,5	101,6

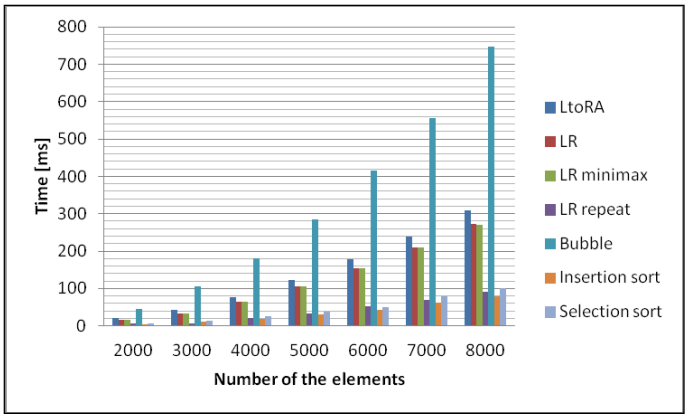


Fig.4 Average times for sorting the rows with a high repetition rate for the elements

The experiments show that:

1) algorithm LR is faster than algorithms LtoRA and Bubble sort but it is slower than algorithms Insertion sort and Selection sort;
2) algorithm "LR minimax" reduces the number of operations (the time) compared to the operations (the time) of the algorithm LR;
3) algorithm "LR repeat" is faster than algorithms LtoRA, LR minimax, Bubble and Selection sort but it is slower than Insertion sort; when the number of the repeated elements is large, the algorithm "LR repeat" is very effective. The number of operations (the time) compared to those of algorithm LR is reduced considerably.

The experiments show that the algorithm "LR repeat" outperforms the algorithm LR when the average number of repetitions in the row is greater than or equal to 2.

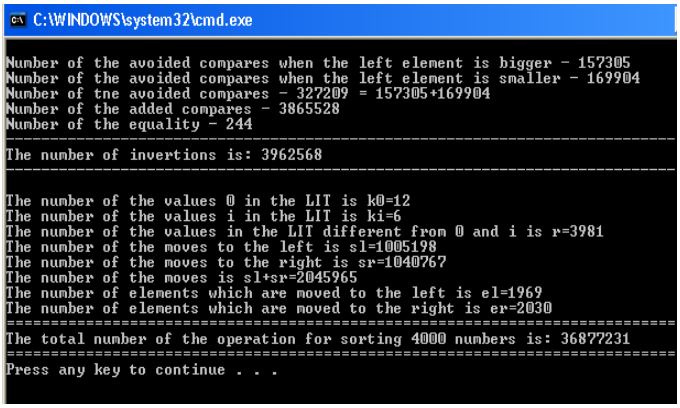


Fig.5. Number of operations for sorting the row with 4000 elements with algorithm "LR repeat" (a low repetition rate for the elements)

VI. CONCLUSION

This paper proposes the algorithm for sorting by LIT (Left Inversions Table) with filling to the left and right. The complexity of the proposed algorithm is evaluated. Its minimal and maximal complexities are derived. An experimental comparison of the proposed algorithm LR with algorithm LtoRA and some known algorithms (Bubble sort, Insertion sort, Selection sort) is also done. Two modifications of the algorithm LR are proposed and realized: "LR – comparisons with the current minimal and maximal values" and "LR with avoiding repeated comparisons".

The experiments show that:

a) algorithm LR is faster than algorithms "Bubble sort" and LtoRA but it is slower than algorithms "Insertion sort" and "Selection sort";
b) algorithm "LR repeat" is faster than algorithms LR for rows with number of repetitions larger than 2;
c) obviously there is an average number of repetition for which the algorithm "LR repeat" will be faster than algorithm "Insertion sort";
d) algorithm "LR minimax" is faster than algorithms LR;

Finally, algorithm "LR minimax" can be used to sort any row, while using the algorithm "LR repeat" needs a preliminary estimate of the average number of repetitions in the given row.

The future work will continue with:

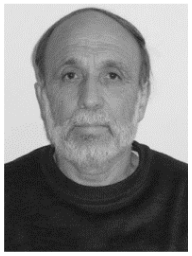
1) developing methods for quick estimation of the average number of repetitions in the row;
2) unification and applying both the improvements;
3) developing methods (algorithms) with smaller number of the comparisons and moves;
4) development and implementation of good parallel algorithms for sorting.

REFERENCES

[1] Knuth D., The art of computer programming, V3. Sorting and Searching, Addison Wesley Publishing Company, 1973.
[2] Stoichev St., Synthesis and analysis of algorithms, BPS, Sofia, 2003 (in Bulgarian).
[3] Nakov P., P. Dobrikov, Programming=++Algorithms, TopTeam Co, 2003 Наков (in Bulgarian).
[4] Sedgewick R, Algorithms in C, Addison-Wesley, 1990.
[5] Harris S., J. Ross, Beginning Algorithms, 2005.

[6] Wirth N., Algorithms+data structures=programs, Prentice-Hall, 1976.
[7] N. Vasilev, A. Bosakova-Ardenska, Algorithms for sorting by left inversions table, International Review on Computers and Software (IReCoS), vol 7 n2 – Part A, 2012, ISSN 1828-6003, pp 642-650.

BIOGRAPHIES



NAIDEN B. VASILEV is associate professor in department of "Computer Systems and Technologies" at Technical University of Plovdiv. He is receives Ph.D. in 1976. His research interests include: parallel algorithms, discrete mathematics and music.

ATANASKA D. BOSAKOVA-ARDENSKA was born in 1980. She received the M.Sc. degree of Computer Systems and Technologies at Technical University of Sofia, Plovdiv branch 2004. She receives Ph.D. in 2009 with thesis "Parallel information processing in image processing systems". From 2010 she is assistant in department of Computer Systems and Technologies in University of Food Technologies. From 2014 she is associated professor by "Synthesis and Analysis of Algorithms" in department of Computer Systems and Technologies in University of Food Technologies in Plovdiv, Bulgaria. She is member of USB (Union of Scientist in Bulgaria) and head of Club of Young Scientists in Plovdiv (USB – Plovdiv in Bulgaria). Her research interests include: parallel algorithms, sorting algorithms, image processing, MPI (Message Passing Interface), C++ programming.