In-place sort algorithm by tournament

Shuji Kaya

1. Overview

This document's purpose is a presentation that a sort algorithm is different from ever issued. I named it "Section top sort".

Cardinal sort method is a tournament bracket to choose a champion, and repetition of tournament for next any ranks until the end of data.
Ever known sort algorithms by tournament always demand memory area for tournament result besides the sorting array. 2)3) However, this sorting doesn't need memory area except for the sorting array basically. It memorizes tournament result inside of the sorting array. It uses O(log N) stack for recursive call. But deleting recursive call is possible by modification.
It is classified into in-place sort, and unstable sort. Basic sort method is exchange. It becomes hybrid of exchange and selection in best speed up version.

Speed performance : minimum time O(N log N), maximum time O(N^2), average time O(N^2).
I have tried speed up alteration. This document includes these trials.
I guess that both maximum and average time become O(N log N) in best one, but I have not got the evidence yet. Even fastest one, it is slower than quicksort.

2. Detailed explanation

2-1. Condition

* Sort data must be in an array, accessed by index number.. The index number must start from 0. The index number is essential of this sort. this sorting can't treat non-array structure.
* The array length and 2^N (N:natural number) equal to or first exceed value from array length are used in the process. These are necessary in advance.
* Random access of each element in the array is necessary. Whole array stored in memory at once is not necessary.

2-2. Method

Thinking a tournament bracket above sort data array laid in horizontal courses.
Undermost layer first rounds stand on sections of 2 elements. Array is divided into these 2 elements sections from the head to the tail.
Next layer second rounds stand on sections of 4 elements. Array is divided into these 4 elements sections from the head to the tail.
Similarly, the bracket continues up to the final round. Sections of 8 elements, sections of 16 elements, ... until sections of 2^N elements, the array is divided into stratification.
The tournament match content is comparison between elements. If later element wins former element, these are exchanged. After the match, these are arranged.
After the all first rounds, in all sections of 2 elements, winner (the top of the section) sits former (head) location.
As second rounds, in all sections of 4 elements, the match between top (head) of 2-elements sections plays. In all sections of 4 elements, the top of the section goes to the head of the section.
In the same way, in all sections of all layers, the two low layer section tops play the match to decide high layer top.

After the final round match of the tournament, the champion (element to be sorted first) sits the head of the array.
Champion graduates from the sorting matches.
The top element of rest is chosen and is set at the head of rest by similar tournament. Runners-up is sorted.

Repeat similar tournament matches until whole rank is decided. When no pending rank remains, the sort is completed.

When deciding runners-up and after, whole new tournament round matches are not necessary.
In all sections of 2^k (k = 1, 2, 3 ... N-1) size, the top element in it already sits the head of it.
By dividing the rest into sections as large as possible and playing the matches between top of these sections, the rank element is decided.
The expression "top element in it already sits" is not always right. The relation is sometimes broken.
So the renaturation of this relation is necessary. The detail method will be described later.

Program list of this sorting is shown below. (All program list in this document is C-language)

First function is the match play process.
Input argument 'superior' is former index number and 'inferior' is later index number.
This routine sets each element to the proper location in these.

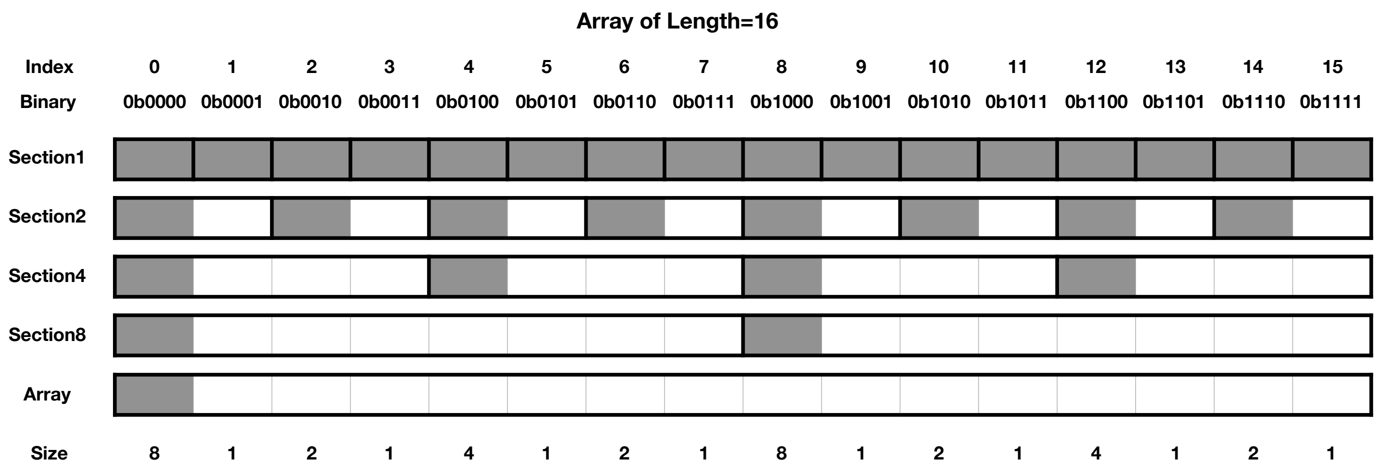
```
void    top_match_for_location(unsigned int superior, unsigned int inferior) {

        if (inferior >= amount)          //  If index number exceeds array length (amount), do nothing.
                return;

        if (win(inferior, superior)) {              // Comparison process
                exchange(inferior, superior);       // Exchange process
                down_to_the_section(inferior);      // Rearrange process to set top element
                                                    //   at head location in the section.
        }
}
```

Without exchange, the rule "Section top element sits head location" is held.
At the superior location with exchange, the rule is clear to be held, because the newcomer element surpasses the top until then. Only at the inferior location with exchange, the rearrange process is necessary to execute.

Second function is the process to get maximum section size (amount of element) that regards the location (index number) as the top.
Below figure is an array of 16 elements.



This array is divided into sections of 2, 4, until 8. These tops are set to the head (colored location).
The maximum size easily comes out from the binary of index number. Seeing binary from least significant bit, first 1 value is the maximum size. For example, when index binary is 0b???1000, the location is the top of 0b1000 (8) elements section. When index is odd number 0x?????1, it is the top of 1 element section.
When index is 0, the result 2^(N-1) is impossible to calculate from index. The result is calculated from 2^N. In fact, this routine is not called with argument 0, but infinite loop happens at the case. The branch is added for safety.

```
unsigned int    section_size_from_top(unsigned int section_top) {

        unsigned int    section_size;

        if (section_top == 0) {
                return  (maximum_size / 2);             // maximum_size = 2^N (>= amount)
        }
        else {
                section_size = 1;

                while ((section_top & section_size) == 0) {
                        section_size *= 2;
                }
        }
        return  section_size;
}
```

Third function is main process of sorting.
It is the same description as the explanation so far.
Sorting is completed by 2 loops. One is to decide champion (0th index). Other is to decide later rank (1st index and after).

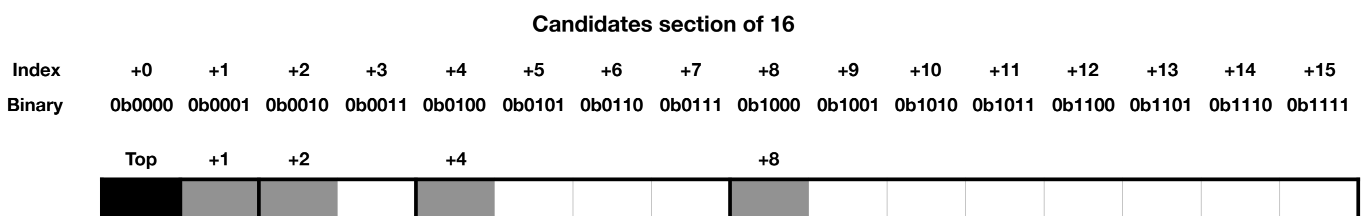
```
void    section_top_sort() {

        unsigned int    size, distance, non_sorted, index;

        for (size = 2, distance = 1; distance < amount; size *= 2, distance *= 2) {
                for (index = 0; index < amount; index += size) {
                        top_match_for_location(index, index + distance);
                }
        }

        for (non_sorted = 1; non_sorted < amount; non_sorted++) {
                for (index = non_sorted; index < amount; ) {
                        index += section_size_from_top(index);
                        top_match_for_location(non_sorted, index);
                }
        }
}
```

Last function is rearrange process to set the top element to the head location in the section.
Below figure is the case that new element from former location comes section of 16 elements.



Black element is an newcomer.
Just the same as the process of runner-up and after rank, the section is divided into sections as large as possible. But the size calculation is more simple. Candidate locations are calculated adding numbers from 1, and numbers repeating double, until the section size.
(That is equivalent to searching from bottom in binary search tree.)
If there is an exchange in a match, this routine is called in recursive.

```
void    down_to_the_section(unsigned int section_top_location) {

        unsigned int    distance;
        unsigned int    section_size = section_size_from_top(section_top_location);

        for (distance = 1; distance < section_size; distance *= 2) {
                top_match_for_location(section_top_location, section_top_location + distance);
        }
}
```

This sort all particular program list is end here. General part of sorting is omitted.

More concrete behavior of this sorting is repeated below.
(1) Divide the array into sections of 2 elements from front to back. Play matches between next elements in sections. Set winner (top) to front (head), loser (bottom) to back (tail) in these sections.
(2) Divide the array into sections of 4 elements [including two (1)-sections] from front to back. Play the match between top elements of next (1)-sections in it. Because winner was set to head by (1)-matches, play the match between elements at head location of (1)-sections. Top of 4 elements is set to head location.
If no exchange accrues, (1)-rule is kept. If an exchange accrues, the promotion element keeps (1)-rule, but the demotion element can break (1)-rule of the section. In (1)-section of demotion element, play (1)-match between next elements to keep (1)-rule.
(3) Divide the array into sections of 8 elements [including two (2)-sections] from front to back. Play matches between top elements of (2)-sections in it. Because winner was set head by (2)-matches, play matches between elements at head location of (2)-sections. Top of 8 elements is set to head location.
If no exchange accrues, (1)-rule and (2)-rule are kept. If an exchange accrues, the promotion element keeps (1)(2)-rule, but the demotion element can break (1)(2)-rule of the section. In (1)-section of the demotion element, play (1)-match. And in (2)-section of the demotion element, play (2)-match. (1)(2)-rule are kept again. If an exchange accrues again by (2)-match, play (1)-match in (1)-section of the demotion element, to keep (1)-rule.
(4) Repeat the same process until 2^N size section (2^N >= array length), that covers whole array. Champion sits the head location of array. And in each section of every 2^k (k= 1 ... N-1) size layer, the top element of the section sits at the head location.
(5) Champion graduates the matches. The matches to choose the top of the rest elements continue.
After champion element, section of 1 element (top itself only), section of 2 elements (its head is top), section of 4 elements (its head is top) ... follow.
Runner-up element is the strongest of these rest section tops.
If an exchange accrues in runner-up decision matches, in the demotion element section, plays matches of rule remake process like (1)(2)(3). Repeat it until no exchange. The rule "The section top sits at head location" is kept by that.
The maximum section size regarding the location as the top, is calculated from index number binary.
(6) Set following rank elements to the fitted location by repeating the same way as previous.
Divide the unsettled part to sections as large as possible, play matches between tops and set the winner to the foremost location. Repeat it until sort finishes.

Thinking the case of a draw. In a match, elements between player elements are no regard. If a draw element with either player exists between player elements, the draw element order becomes upside down. In conclusion, this sorting is unstable.

If draw elements must follow the original order, prepare the same size index array as data, and set order in it. Whenever an exchange accrues, exchange the same location of this index array. When a match is draw, compare the number in this index array and judge the original front side element as the winner. It is just same as another unstable sort.

3. Characteristics

First theme is execution time.
When dividing the time into 3 parts, 1.comparison process, 2.exchange process, 3.other sort process, other sort process is too small to affect without doubt. Comparison count and exchange count are considerable points.
The measured result is shown chapter 5. Below conjecture comes out.
Comparison count : minimum = (N log2 N)/2, maximum = (N-1)*N/2, average = O(N^2). (The formula is hard to figure.)
Exchange count : minimum = 0, maximum = (N-1)*N/2, average = (N-1)*N/4
When amount N is very large, larger count conceivably effects performance.
Minimum time O(N log N), maximum time O(N^2), average time O(N^2) are conclusion.

The shorter section rearrangement process time is, rephrasing, the fewer exchange count is, the sorting ends faster. In faster case, comparisons are omitted by a law "The element that wins the section's top clearly wins all elements in the section".
Saying it with initial data distribution, the former strong element locates and the later weak element locates, the sorting ends faster. The best case is that originally all element follows sort order. The worst case is just the opposite order. The worst case needs the same time as bubble sort.

Second theme is memory.
Sort array is the only thing to occupy memory through the process. No extra work area is necessary.
But log2 N stack area is clearly consumed by recursive call.

Last theme is merit.

* Elements are sorted from the array's head. The fixed range is clear in all moments in sorting. This sorting fits to partial sort, like the aim is only top 100. And this sorting can output sorted elements at the moment fixed, before the sort is completed.

* This sorting will have good performance in parallel computing, I think. A real tournament is almost done in parallel, that is the evidence. And the beginning of this sorting resembles to sorting network algorithm. I guess this sorting may be more suitable for real CPU than network algorithms, when computing thread amount is very few compared with the sorted item amount. This sorting can terminate the sorting if the data are in order. This may help to avoid waste of threads.

* This sorting will have good relation with cache, I think, in the case whole data can't exist in memory at once and disk/network communication in sorting is necessary. The same element sits candidate location and it is used in comparison repeatedly. And if exchange arises, referred elements in successive comparison and exchange, is commonly in narrow range nearby.

4. Alteration

This sorting basic version speed is slow. Trying speed up of the version.
I have tried to make 5 versions. This chapter is an archive of these trials.

4-1. Less conflict

The main time loss is when an element demotion accrues. "The top of the section sits the head of it" rule rearrangement process is it. The fewer this element demotion count is, successive comparison and exchange count become fewer. In conclusion, the sorting ends faster.
Thinking decrement way of element demotion time. It is to stop the exchange in every match and only memorize the winner location. Exchange is limited only one time between the strongest element and the top location at last.
"down_to_the_section" process changes like below.

```
void    down_to_the_section_1exchange_only(unsigned int section_top_location) {

        unsigned int    distance;
        unsigned int    really_top, index;
        unsigned int    section_size = section_size_from_top(section_top_location);

        really_top = section_top_location;
        for (distance = 1; distance < section_size; distance *= 2) {
                index = section_top_location + distance;
                if (index >= amount)
                        break;
                if (win(index, really_top))
                        really_top = index;
        }

        if (section_top_location != really_top) {
                exchange(really_top, section_top_location);
                down_to_the_section_1exchange_only(really_top);
        }
}
```

When deciding runners-up and after, same alteration like this is possible. I omit the program list because the difference is just same.
By these alterations, sort method becomes hybrid of exchange and selection.

In this C-language program list, "goto" statement can take the place of recursive call “down_to_the_section_1exchange_only". It releases this sorting from recursive stack consumption.
The dirty program list that derogates from structure programming, is omitted.

As shown in measurement of chapter 5, minimum count stays the same, but maximum/average count improve. Basic version speed O(N^2) is multiplication of elements amount N and O(N) time per one element. I presume that this modification time per one element becomes O(log N), because recursive rearrangement is cut. The speed becomes O(N log N), I guess. But I have not got the confirmation from measure result. For example, maximum count data of 8 elements are "5 4 6 3 7 2 1 0", "6 5 4 3 7 2 1 0". These are hard to find the rule. I can't think out the formula of maximum case. Average case, too.

Other trial later are hard to say speed up. It is the present best version of this sorting algorithm.

4-2. Full section exchange

There is another way to keep the rule "Top element of the section sits the head of it". It is to exchange the whole sections, not only the head elements. However, it can apply to only the match between the same size sections. The possible point of this modification is limited.
I omit the detailed points and program list, because it is not enough to explain.

When array length is not 2^N, it is necessary to fill weakest elements after the array tail until 2^N. This extra huge memory necessity defeats this modification practicability. 2^(N-1)+1 length array sort speed is the same as 2^N length array. In many cases, speed clearly becomes slow. Even a 2^N length array sort, comparison count deceases, but exchange count increases. Apparent improvement from chapter 4-1 is out of hope.
Maximum/average case is hard to think out like above try. (Maximum count data of 8 elements is "3 2 7 1 6 5 4 0".)

4-3. Two top

This sorting algorithm uses the rule "Top element of the section sits the head of it". How does one more additional rule change the speed? I try to add a rule "Second top element of the section sits the center of it".
I omit detailed explanation because these deviate from the essence of this thesis.
As shown in measurement in chapter 5, both comparison/exchange count increase through minimum/maximum/average than chapter 4-1. Some results are no good, because the program still remains some bugs (in the case runner-up location is out of array).

The conclusion is, increasing rule amount gives effect to the sort speed slower.
Maximum/average case is hard to think out like above tries. (Maximum count data of 8 elements are "5 4 7 6 1 0 3 2", "5 4 7 6 2 1 3 0", "5 4 7 6 3 2 1 0", "6 5 7 4 1 0 3 2", "6 5 7 4 2 1 3 0", "6 5 7 4 3 2 1 0", "7 6 5 4 1 0 3 2", "7 6 5 4 2 1 3 0", "7 6 5 4 3 2 1 0".)

4-4. Set next

Thinking the data that the champion locates at the head of second half part of the array. After some advance in this sorting, champion at the second half section head is exchanged with an element X at first half section head (whole array's head). X comes to the second half section head. Later step, X will be compared with each section tops in first half. But X has been already first half section top. So these comparisons are wasteful.
Like this example, the previous tournament record is lost by exchange, and sometimes just same comparison as past is doubled. It is the cause of speed down. The solution is, to take an area of match results in memory, to record in, and to check these before playing matches. Huge memory is necessary for it and this sorting loses merit, in-place.
If limiting only the case X at this chapter top, it can be modified without additional memory. The way is to stop jumping X to the second half, to shift X to the next location of the sorted area in first half. Instead of X, substitution element that doesn't break the rule, jumps to second half head.
I omit detailed modification points because these deviate from the essence of this thesis.

As shown in measurement in chapter 5, comparison count decreases, but exchange count increases from chapter 4-1. Exchange needs writing to device. Normally writing occupies more time than reading. I presume it is not an improvement of speed-up.
Maximum/average case is hard to think out like above tries. (Maximum count data of 8 elements is "7 5 6 4 3 1 2 0".)

4-5. 'Both sides now'

Thinking opposite direction process. In the section of 2 elements, the step to set the top element to the head location is, also the step to set the bottom element to the tail location. In the section of 4 elements, the step to set the top element to the head location do not overlap the step to set the bottom element to the tail location. These can execute in parallel. Similarly, moreover section top setting step doesn't overlap bottom setting step and these can execute in parallel.
However, top/bottom rearrange process in the section sometimes overlaps. This process is very complex. In my opinion, this may be the most complex and long sorting procedure.
I omit the explanation because it transcends my ability. I have made a program to pass all my test data. But I can't decide it the completed final one or not.
As shown in measurement in chapter 5, normally both comparison/exchange count increase from chapter 4-1. It is not an improvement of speed-up. Maximum/average case is hard to think out like above tries. (Maximum count data of 8 elements are "7 5 6 2 4 3 1 0", "7 5 6 3 4 1 2 0", "7 6 4 2 5 3 1 0", "7 6 4 3 5 1 2 0".)

When the number of array elements is limited to 4, it is the fastest sort. Comparison/exchange ends until 5 count. In the section of 4 elements, top/bottom rearrange process is not necessary. Top/bottom relation in the section of 2 elements is held without operation. Even 8 elements array result, both comparison/exchange count are still less than 4-1. Some scope for consideration may rest.

4-6. Other ideas

There are some other ideas, these clearly do not improve theoretical performance. I have not examined by program. But these may be effective in some case. Below writings are memorandum.

Until now, elements included one section is always continuous. However, section can be consisted from discrete elements. One way is to scan index number binary from MSB, instead of LSB. Thinking 8 data sort as an example. Index 0b0xx and index 0b1xx are arranged in a same section of 2 elements. (xx is the same number.) Theoretical performance is just the same. But the time distribution by data order is changed. Real data does not agree uniformly distribution. There is something deviation in it, like strong elements gather in one district and weak elements gather in another. By adjustment of the partition to interpret index number binary, actual sort time may be possible to shorten in some cases.

Instead of champion from array top, a way to select tailender from array tail is also conceivable. It changes only sort direction, like ascending or descending switch. Sort speed is just same. In basic version, champion always stays foremost of data after the final match of tournament tree. If choosing tailender, unless the data amount is 2^N, tailender doesn't stay very back after the final tournament match. (The final match is skipped because of out of range.)
If tail side sorted data is necessary earlier while sorting, this sorting can accommodate the case.

5. Time measurement

The whole program list used in operation verification and speed measurement is placed at below location.

http://boogieshake.web.fc2.com/sort/section_top_sort.c

The measurement result is shown in the table from next page. Detailed evaluation is omitted because essential points are already mentioned previously.
Until 1 - 12 amount data, the whole permutation are made and sorted. These minimum/maximum/average agree with theoretical values.
100/512/1000 amount data minimum/maximum/average time are the result of only some sample data. These differ from theoretical values.

Quicksort value in gcc is shown at the last of this table. Only comparison count can be measured as a library usage. Because it is clear that this sorting loses at present phase, I don’t examine it moreover.

Compare and exchange count measurement in Sorting

Kind	Data amount	Total line	Compare			Exchange		
			minimum	maximum	average	minimum	maximum	average
Section Top Sort [Basic]	1	1	0	0	0.000000	0	0	0.000000
	2	2	1	1	1.000000	0	1	0.500000
	3	6	3	3	3.000000	0	3	1.500000
	4	24	4	6	5.333333	0	6	3.000000
	5	120	8	10	9.333333	0	10	5.000000
	6	720	9	15	13.000000	0	15	7.500000
	7	5040	11	21	18.333334	0	21	10.500000
	8	40320	12	28	23.333334	0	28	14.000000
	9	362880	20	36	31.333334	0	36	18.000000
	10	3628800	21	45	37.666668	0	45	22.500000
	11	39916800	23	55	46.333333	0	55	27.500000
	12	479001600	24	66	54.000000	0	66	33.000000
	100	100	3451	4263	3901.920000	2040	2893	2482.380000
	512	512	65535	107675	102484.134766	60513	70675	65489.023438
	1000	1000	65535	405052	391165.900000	65535	267105	250076.896000
Section Top Sort [Less conflict]	1	1	0	0	0.000000	0	0	0.000000
	2	2	1	1	1.000000	0	1	0.500000
	3	6	3	3	3.000000	0	3	1.500000
	4	24	4	6	5.333333	0	6	3.000000
	5	120	8	10	9.000000	0	8	3.933333
	6	720	9	14	12.033334	0	11	5.900000
	7	5040	11	20	16.995237	0	15	7.677778
	8	40320	12	27	21.298412	0	20	10.465873
	9	362880	20	35	28.061905	0	22	11.408995
	10	3628800	21	39	31.463148	0	27	13.755274
	11	39916800	23	46	37.599262	0	30	15.753829
	12	479001600	24	54	42.750179	0	36	19.101740
	100	100	1074	1249	1161.850000	302	377	336.980000
	512	512	10430	11294	10900.849609	2364	2584	2477.164062
	1000	1000	24502	26346	25445.341000	5061	5421	5245.574000
Section Top Sort [Full exchange]	1	1	0	0	0.000000	0	0	0.000000
	2	2	1	1	1.000000	0	1	0.500000
	3	6	4	5	4.666667	0	6	3.000000
	4	24	4	5	4.666667	0	6	3.000000
	5	120	12	22	18.380952	0	23	11.771429
	6	720	12	22	18.380952	0	23	11.771429
	7	5040	12	22	18.380952	0	23	11.771429
	8	40320	12	22	18.380952	0	23	11.771429
	9	362880		Out of time			Out of time	
	10	3628800		Out of time			Out of time	
	11	39916800		Out of time			Out of time	
	12	479001600		Out of time			Out of time	
	100	100	1112	1618	1358.940000	338	605	469.910000
	512	512	9899	10830	10446.103516	2845	3737	3253.757812
	1000	1000	24211	26457	25560.028000	6060	8135	7197.247000

Kind	Data amount	Total line	Compare			Exchange		
			minimum	maximum	average	minimum	maximum	average
Two Top Sort	1	1	0	0	0.000000	1	1	1.000000
	2	2	2	2	2.000000	1	2	1.500000
	3	6	4	5	4.666667	1	3	2.166667
	4	24	7	7	7.000000	2	7	4.333333
	5	120	10	13	12.000000	1	8	4.733333
	6	720		NG			NG	
	7	5040	14	21	18.009524	2	17	9.357143
	8	40320	17	27	22.842857	3	23	12.766667
	9	362880	26	38	34.001587	4	25	14.512698
	10	3628800		NG			NG	
	11	39916800		NG			NG	
	12	479001600		NG			NG	
	100	100		NG			NG	
	512	512	29664	36859	33064.343750	17361	22830	19948.535156
	1000	1000		NG			NG	
Section Top Sort [Set Next]	1	1	0	0	0.000000	0	0	0.000000
	2	2	1	1	1.000000	0	1	0.500000
	3	6	2	3	2.666667	0	3	1.500000
	4	24	4	6	4.833333	0	6	3.000000
	5	120	5	10	8.000000	0	9	4.733333
	6	720	8	13	10.633333	0	14	7.055556
	7	5040	11	19	14.904762	0	20	9.830159
	8	40320	12	24	18.524603	0	26	13.112698
	9	362880	12	34	25.753483	0	27	13.474956
	10	3628800	17	37	28.135573	0	33	16.468307
	11	39916800	22	43	32.998617	0	41	20.324998
	12	479001600	23	50	37.265566	0	49	24.923850
	100	100	970	1134	1049.19000	402	522	464.920000
	512	512	9772	10588	10194.183594	3241	3681	3429.621094
	1000	1000	22781	24627	23875.151000	6881	7576	7248.754000
Both sides now sort	1	1	0	0	0.000000	0	0	0.000000
	2	2	1	1	1.000000	0	1	0.500000
	3	6	3	3	3.000000	0	3	1.500000
	4	24	5	5	5.000000	0	5	2.666667
	5	120	9	9	9.000000	0	8	4.133333
	6	720	11	13	11.933333	0	12	6.011111
	7	5040	14	18	16.142857	0	15	7.557143
	8	40320	17	23	20.571429	0	19	9.917460
	9	362880	23	33	29.185185	0	22	12.194709
	10	3628800	25	37	32.605926	0	27	14.482081
	11	39916800	28	44	38.304185	0	30	16.427016
	12	479001600	31	53	44.475075	0	37	19.599656
	100	100	1156	1344	1257.900000	320	417	373.320000
	512	512	11786	13679	12766.183594	2627	3207	2920.757812
	1000	1000	28525	32808	30573.943000	5800	7148	6328.246000

Kind	Data amount	Total line	Compare			Exchange		
			minimum	maximum	average	minimum	maximum	average
***gcc q-sort**	1	1	0	0	0.000000			
(For comparison)	2	2	1	1	1.000000			
	3	6	2	3	2.666667			
	4	24	3	6	4.916667			
	5	120	4	10	7.716667			
	6	720	5	15	11.050000			
	7	5040	6	21	14.907143		Can't count	
	8	40320	14	29	19.171354			
	9	362880	16	36	22.888112			
	10	3628800	18	47	27.061262			
	11	39916800	20	55	31.299797			
	12	479001600	22	70	35.746590			
	100	100	603	699	641.710000			
	512	512	4416	5064	4588.541016			
	1000	1000	9661	10708	10003.768000			

6. Conclusion

This document content is a sort algorithm explanation of my idea. It includes fundamental version and some improvement trials of speed up. Since this sorting is possible to be different from a previously announced one and to be useful in some purpose - partial sort, parallel sort, etc., it is documented and shown here.

Remain problem is, maximum and average time mathematical expression of this sorting fastest version. Presently it is only a conjecture. Providing evidence is necessary.
As future development, against the sentence that this sorting has affinity with parallel computing, actual algorithm and program in parallel process are necessary to prepare and to confirm.

7. References

Because I am not a scholar, all referred article is limited to that everyone can see in internet.
I referred known sort algorithm and technical terms from 1). I searched the papers including ‘sort’ in title from Cornell University Library, and checked recently sort algorithm usage tendency from 4) 5).

1) Wikipedia "Sorting algorithm" article and all its related article
2) Wikipedia "Tournament sort" article
3) arXiv.org > cs > arXiv:1402.2712 "Dynamic Partial Sorting"
4) arXiv.org > cs > arXiv:1609.04471 "Sort Race"
5) arXiv.org > cs > arXiv:1511.03404 "Comparison of parallel sorting algorithms"