Institute of Computer Science
The Czech Academy of Sciences

Sort program for real keys with linear time complexity

Marcel Jiřina

Technical report No. V-1272

October 2019

Pod Vodárenskou věží 2, 182 07 Prague 8, phone: +420 266 033 50, fax: +420 286 585 789,
e-mail: marcel@cs.cas.cz

Institute of Computer Science
The Czech Academy of Sciences

Sort program for real keys with linear time complexity

Marcel Jiřina [1]

Technical report No. V-1272

October 2019

Abstract:

In this report we present a program for sorting data structures with sorting keys as real numbers, i.e. of type "real" or "float". The basis of the program is a modification of the countingsort algorithm for reals (instead of integers). It uses a comparison-type sorting for small part of data set given. The time complexity of this part of program can be bounded by linear function of n and thus, the total time complexity is also O(n) for n data items.

Keywords:
Sorting, Real sorting keys, Counting sort.

[1]Institute of Computer Science, The Czech Academy of Sciences, Pod Vodárenskou věží 2, 182 07 Prague 8, Czech Republic, E-mail: marcel@cs.cas.cz

Abstract

In this report we present a program for sorting data structures with sorting keys as real numbers, i.e. of type ”real” or ”float”. The basis of the program is a modification of the countingsort algorithm for reals (instead of integers). It uses a comparison-type sorting for small part of data set given. The time complexity of this part of program can be bounded by linear function of n and thus, the total time complexity is also O(n) for n data items.

Keywords: Sorting; linear time; time complexity.

I. Introduction

The theme of fast sorting with noninteger (real, float) sorting keys is dominated by the quicksort algorithm that exists in uncountable number of versions. Note that a big advantage of the quicksort is a small and constant space complexity. There is also a proof that for rational numbers keys generally the best time complexity of comparison-based sorting is given by $n \log n$ [1].

In this work we consider numerical sorting keys as real (in fact, rational) numbers that can be sometimes the same, sometimes very close one to another. The target of this work is to present a modification of the counting sort for reals as sorting keys. It often happens that several items sorted fall into one cell (bucket). These items are unsorted. We propose to sort them using a comparison sort, e.g. bubble sort or quicksort. It can be found that the number of items faling into one bucket is very small not exceeding several tens in practice. At the same time, also the number of cells with such multiplicities is also limited. Based on upper estimations of multiplicities and their probabilities it can be prooved the linear time complexity of the algorithm.

II. Background

A. the computational complexity

The computational complexity of algorithm is the amount of time, storage or other resources necessary to execute them.

Most often the computational complexity is meant as the $time\ \ complesity$. There is the $space\ \ complexity$ that gives the amount of memory needed. Some other criteria are

Stable sorting.

In place sorting.

The number of data moves.

III. Results

A. The algorithm outline

The algorithm uses exactly n buckets, bins. Similarity to the counting sort is in that to each sample its position is derived from the value of the sorting key. After such an assignment some buckets (bins) are occupied by exactly one sample. Sice that time the relative position of such a sample to other samples is final. Some bins are empty and some bins are occupied with several samples. We call this a multiplicity. This is similar to the bucket sort. Samples in these buckets must be sorted and (with exception of one sample) assigned to nearest bins, one sample a bin. This way a sorted array is obtained. There is mostly very small number of samples in a bin. We found that this small number is limited to at most several tens for very large number of samples of order 10^{15}. The time demand for sorting one sample in a bin with multiple samples can be limited to a constant.

The upper margin of the time complexity to sort all samples in bins with multiple samples is proportional to their total number and a constant mentioned. Thus, the upper margin of the time complexity of the algorithm is proportional to the total number of samples sorted independently of algorithm used for sorting samples in bins with several samples.

B. The algorithm description

In more detail - it is supposed that sorting key has known minimum and maximum, s_m and s_M. Let the value of a key of a sample be κ. The key is of type "real" or "float", in fact a rational number. An index i is computed as

$$i = \lfloor (n-1)\frac{\kappa - s_m}{s_M - s_m} \rfloor \ . \qquad (1)$$

The algorithm consists of four, eventually five loops of fixed length and needs three integer arrays of length n named $rank1$, $rank2$, and $rank$. We suppose array $rank1$ initiated with zeroes. Note that array $rank2$ can be omited later. We call an element of an array a bin.

In the first loop index i is computed for each sample according to (1) and the content of bin i of array $rank1$ is increased by 1. Thus in array $rank1$ an histogram arises. One can find that some bins are empty, lot of them contain 1 corresponding to exactly one particular sample, and some more, the multiplicities of samples.

In the second loop array $rank2$ is filed with partial sums (from the beginning) of values gained in $rank1$ array during the previous loop. Thus, in fact, an empirical distribution function of keys arises.

In the third loop array $rank$ is filled so, that empty bins of $rank1$ array are skipped, and to bins of $rank1$ array with multiplicities a corresponding number of bins in $rank3$ array are assigned. Bins of array $rank3$ are filled with initial ranks of sorted samples (i.e. with pointers to unsorted array of samples). In groups of bins that correspond to multiplicties samples are usually not in the proper oder and must be sorted.

In the fourth loop these groups of bins in array $rank$ are scanned and items are sorted by any comparison based algorithm. These groups are usually small and their sorting fast. It holds even if the bubble sort algorithm is used. This results in that array $rank$ contains indexes of samples ranked according to the sorting key.

During the fifth loop samples can be arranged in the output sequence.

Example 1: Let there be 10 samples to sort as shown in Table I in the second column. The third and fourth column shows computation of indexes of individual samples.

TABLE I
THE FIRST ASSIGNMENT OF RANKS TO SAMPLES.

sample number	sample x	ix= $(n-1)\frac{x-min}{max-min}$	index (first rank)
0	0.263	2.55	2
1	0.582	6.28	6
2	0.407	4.24	4
3	0.088	0.51	0
4	0.815	9	9
5	0.044	0	0
6	0.603	6.25	6
7	0.249	2.39	2
8	0.232	2.19	2
9	0.641	6.95	6
min	0.044		
max	0.815		

Table II shows in the second column the histogram of indexes computed in the previous table. The third column shows in each bin a sum of contents of corresponding bin and of previous bin of array $rank1$. There is an exception, the first bin remains zero. (Thus, in array $rank2$ the empirical distribution function of keys appears.) The number 0 in row 0 of the third column (array $rank2$, first item) says that the group of samples starts at 0. The number of samples is given by the difference of the next bin of array $rank2$ (second item, 2) and the first item (0), that gives 2, then the bucket corresponds to two samples, see the fourth column of Table II. Initially array $rank$ is empty. The indexes of samples (in the original unsorted order) are used again. First, there is sample No. 0 with index 2. Then its number (0) is written to the first free

TABLE II
HISTOGRAM AND FIRST RANKING OF SAMPLES. NOTE TWO MULTIPLICITIES IN THE HISTOGRAM.

bin number	array $rank1$ histogram	array $rank2$ indexes	size of group	array $rank$ samp.no.	before sort sample
0	2	0	0	**3**	**0.088**
1	0	2	2	**5**	**0.044**
2	3	5	3	*0*	*0.263*
3	0	5	0	*7*	*0.249*
4	1	6	1	*8*	*0.232*
5	0	6	0	2	0.407
6	3	9	3	1	0.582
7	0	9	0	6	0.603
8	0	9	0	9	0.641
9	1	10	1	4	0.815

position of array $rank$ that corresponds to number 2 in array $rank2$. In this case it is row number 3. In the sixth column of the Table a value of the item is shown (0.263). This way array $rank$ is filled in. It is apparent, that groups of samples that have the same index, need not be in a proper order. Note that here samples are in the same order as they were in the beginning. The algorithm is stable and remains stable if the algorithm used for sorting groups is also stable. Table III shows the sorting of the first group (rows 0 and 1, in bold), and then the sorting of the second group in rows No. 2, 3, and 4 (in italics). There are no other multiplicities and then we have resultig sequence of numbes of samples. In the last column of the table samples after sorting are shown.

TABLE III
SORTING GROUPS AND RESULTING RANKS IN ARRAY RANK3.

bin No.	arr rank first group sorted index	sample	arr rank second group sorted index	RESULT sample
0	**5**	**0.044**	5	0.044
1	**3**	**0.088**	3	0.088
2	8	0.263	*0*	*0.232*
3	7	0.249	*7*	*0.249*
4	0	0.232	*8*	*0.264*
5	2	0.507	2	0.507
6	1	0.582	1	0.582
7	6	0.603	6	0.603
8	9	0.641	9	0.641
9	4	0.815	4	0.815

Note 1: In the program the number of arrays used is reduced. Instead of $rank1$ and $rank2$ arrays a single array $rank1$ suffice.

Note 2: In the third loop indexes according to (1) are computed again. We found that otherwise a new array would be necessary for storing indexes i computed in the first loop, and, moreover, in the whole, it can take a more time.

Note 3: Samples sorted can each represent a data structure, not only individual floating point number. In such a case it is advantageous to have an array of pointers to samples in sorted order and to leave samples stored as they are. In this case the fifth loop of the algorithm is omitted. Technically, if there is an unsorted array of samples $samples[i],\ i = 1, 2, ...n$, the sorted array is $samples[rank[j]],\ j = 1, 2, ...n$.

C. algorithm implementation

The algorithm written in c++ is attached in the Appendix. It does not use the fifth loop, the sequence of indexes of array sorted is stored in array $rank$. The same is assumed about the quicksort called with name IquickSort. Eventually a bucket sort can be used instead.

When sorting groups, these groups are divided to groups of two samples and to larger groups. Pairs of samples are tested and eventually exchanged. For sorting larger groups an external procedure is called.

IV. Practical implications

A. Space complexity

The algorithm uses an input array of rational numbers of length n and output integer array of the same length of indexes of numbers in the sorted order, see Note 3.

Inside of the algorithm one other working integer array is needed, an array for histogram of sorted numbers. The length of this array is $n + 1$. It is also supposed that the algorithm used for comparison sort of bins with multiplicities has negligible space demand as the bubble sort or the quicksort.

B. Data moves

The other criterion for efficiency of an algorithm is the total number of data moves. A data item, a sample, is moved if its position in array is changed.

The first move for each sample happens in the third loop of the algorithm, where each bin of array $rank$ is assigned to a sample. There is estimated to n moves as a negligible small portion of data remain in the same position in the array.

C. The trick

We proved linear time complexity of the algorithm under assumptions of uniform or exponential distribution of values of sorting keys that are assumed to be ”reals”, in fact, rational numbers. We use histograms with as many bins as there are samples. Both proofs use finding that data with distributions mentioned have in this histogram bins with small number of samples (at most several tenths) even for unrealistically large number n of samples. It can be supposed that a similar behavior appears also for other light-tailed distributions.

Heavy-tailed distributions of sorting keys does not have this feature. With heavy-tailed distributions there appear one or several bins containing nearly all samples that are sorted with comparison sort. It causes finally that sorting takes more time than direct use of the comparison sort alone.

We use the fact that data can be transformed using arctg function. The arctg function applied to keys with Cauchy distribution causes uniform distribution of sorting keys. For beta-prime and other distributions the resulting distributiom may be rather strange. We found that this transform works well also for light-tailed distributions including uniform and exponential. However, it brings no advantage and also no disadvantage (with exception of some computation more).

V. Conclusion

In the sort algorithm presented in this report we use a combination of the counting sort and comparison sort. The essence of the algorithm is start as a counting, ev. bucket sort with the number of bins or buckets equal to the number of samples sorted. Samples in buckets with two samples exchange if necessary. Samples in buckets of three or more samples sort with a comparison sort.

The use of the counting sort allows to sort most of samples by direct assignment of rank to samples and eventually exchange some pairs of samples. This way some 70% samples are ranked properly. The time needed to process this part of data is proportional to their number. Remaining 30% of samples form groups of three or more samples. We speak about the multiplicity. The number ev. probability of appearance of these groups lessens rapidly with their size. We found that the maximal size of a group (bin) is several tens only even for unrealistic number of samples. This allows to state an upper margin for sorting these groups by any method in form $constant \times the\ number\ of\ samples$. Total time needed to sort all samples including those in groups grows linearly with the number of items sorted n. At the same time, the space needed is equal to n, ev. $3n$ including input and output arrays. There is also at most $2n$ data moves, approx. $1.6n$ data moves are more probable in practice.

ACKNOWLEDGMENT

The work was supported (partly) by the long-term strategic development financing of the Institute of Computer Science (RVO:67985807) and in part by the Czech Ministry of Education Youth and Sports in project No. $EF16\_D013/0001787$ ($CZ.02.1.01/0.0/0.0/16\_013/0001787$) Cooperation on experiments at the FNAL, USA.

REFERENCES

[1] Hoare, C. A. R.(1962) Quicksort *The Computer Journal* Vol. 5, No. 1, pp. 10-16. Available at $https://doi.org/10.1093/comjnl/5.1.10$

[2] Quicksort. Available at $http://www.algolist.net/Algorithms/\ Sorting/Quicksort$

APPENDIX 1. THE ALGORITHM.

```
void LS(double* arr, long* rank, long samples, double keymin, double keymax) {
      // uses IquickSort and double computation of position
      long k,j,i,ix,kx;
      double logdd;
      long* rank1  = new long [samples+1];
      logdd=1.0/(keymax-keymin);
      for(k=0;k<=samples;k++) rank1[k]=0;
      for(k=0;k<samples;k++){ // FIRST loop to build histogram
            j=(long)fabs((double)(samples-1)*(arr[k]-keymin)*logdd );
            rank1[j+1]++;
      }
      for(k=1;k<=samples;k++) rank1[k]+=rank1[k-1];// SECOND loop to build sums.
      for(k=0;k<samples;k++) { //THIRD loop fill-in rank.
            ix=(long)(fabs((double)(samples-1)*(arr[k]-keymin)*logdd));
            kx=rank1[ix];// bin where a segment starts
            rank[kx]=k;
            rank1[ix]++;//bin for the next time
      }
      ix=0;kx=rank1[0];
      for(k=0;k<samples-1;k++){ //FOURTH loop sort groups by a comparison sort
            if(k>0) kx=rank1[k]+(rank1[k-1]==ix?0:1);
            i=kx-ix;
            if(i>1) {
                  if(i>2) IquickSort(arr,rank,ix,kx-1);
                  else{
                        j=ix;
                        if(arr[rank[j+1]]<arr[rank[j]]) {//exchange
                              i=rank[j+1];
                              rank[j+1]=rank[j];
                              rank[j]=i;
                        }
                  }
            }
            ix=kx;
      }
} // There are indexes in the rank array such that arr[rank[i]] is arranged in ascending order
```

APPENDIX 2. THE QUICKSORT.

Originally published in [2]. We use and cite it here verbatim with small technical changes as follows.

```
//********** I-QUICKSORT *****************
int Ipartition(double* arr,long* rank,long left,long right){
//http://www.algolist.net/Algorithms/Sorting/Quicksort*/
      long i=left,j=right,tmp;
      double pivot;
      pivot=(arr[rank[left]]+arr[rank[right]])/2;
      while(i<=j){
            while(arr[rank[i]]<pivot)
                  i++;
            while(arr[rank[j]]>pivot)
                  j--;
            if(i<=j){
                  tmp=rank[i];
                  rank[i]=rank[j];
                  rank[j]=tmp;
                  i++;
                  j--;
            }
      };
      return i;
}
void IquickSort(double arr[],long ranks[],long left,long right){ //left, right included!!
      long index=Ipartition(arr,ranks,left,right);
      if(left<index-1)
            IquickSort(arr,ranks,left,index-1);
      if(index<right)
            IquickSort(arr,ranks,index,right);
}
//******************************************
```