Quicker than Quicksort

- A very fast sorting algorithm -

Prof. Dr. W. P. Kowalk
kowalk@informatik.uni-oldenburg.de
Unversity Oldenburg, Ammerländer Heerstraße 114-118
D-26111 Oldenburg, Germany

Abstract

We describe a fast sorting algorithm and its properties compared to other sorting algorithms. We also compare implementations in different languages, C and Java, and explain, why the results are substantial distinct.

Introduction

Sorting is a classical task in computer science, although today the most standard algorithms seem to have a sufficient performance. However, from a theoretical point of view one might ask, whether there are better sorting algorithms than the existing ones and how improvements can be gained. We will show such an algorithm and consider its properties as well as the reasons for its behavior in this paper.

Basics of the algorithm

The main idea for this sorting algorithm is the following: we distribute the elements into subsets and proceed recursively. To find adequate subsets, we raise a statistic of the data (which we call a distribution) and sort them into corresponding parts of always the same array.

Our assumptions are the followings, given formally in Java:

1. We sort arrays of data sets, where there is an integer key:
   ```
   class DataSet {...; int key;}
   DataSet Field[];
   ```
2. We require some auxiliar arrays to determine the ranges of the subsets.
   ```
   int FromInd[]; // Number or index of elements in array
   int ToInd[];   // Index of elements in array
   ```
3. We assume that there is a function that can map the key into a subrange. In the implementation below we compute the minimum (`Min`) and maximum (`Max`) key and estimate their difference as the range. Depending on the number of subranges (`Ranges`) we compute a range index by the formula
   ```
   double factor = ((double)(Ranges-1.0))/(Max-Min);
   int rangeIndex =  factor * key;
   ```

From these assumptions the algorithm is canonical. Let N be the number of elements to be sorted. The algorithm starts by finding the smallest and biggest key (O(N)), then it computes the `Faktor` (O(1)), and now it counts the number of elements that belong to each subrange `k` (O(N)) on `FromInd[k+1]`. In the next step the first indices of the corresponding subranges in the total array are computed (O(Ranges)) and the bounds of the corresponding subranges are copied to another array (O(Ranges)).

The sorting process consists of putting the element found in the current subrange at `FromInd[index]` into its corresponding subrange, the element at that place in that subrange into its corresponding subrange, etc, until finally an element is to be put in the current subrange; then the algorithm increments `FromInd[index]` and proceeds until this subrange is exhausted; all elements of this subrange are then in this subrange, which is sorted recursively. Then the next subrange is handled in the same way. Complexity on this level is of course O(N), since each element is touched once.

Adding complexities we get: $3 \times O(N)+2 \times O(Ranges)+O(1)$. Thus complexity on each level of recursion is O(N), while the number of levels is $\log_{Ranges}(N)$. Since `Ranges` is usually very large, this logarithm yields a very small factor.

The algorithm

The next program shows the executable algorithm in Java.

```
void DistributionSort(DataSet [] Field, int from, int to) {
  int FromInd[] = new int[Ranges+1];  // Array for characters, including 0 for empty character
  int ToInd[] = new int[Ranges+1];    // set counters to 0
  int Min = Field[from].key;          // Minimum key
  int Max = Field[from].key;          // Maximum key
  for(int Ind=from;Ind<=to;Ind++) {   // find min and max key
    int key = Field[Ind].key;
    Min = Math.min(Min,key);
    Max = Math.max(Max,key);
  }
  if(Max<=Min) return;                // Min, Max are least and biggest key
  double factor = ((double)(Ranges-1.0))/(Max-Min); // key*factor is index range
  for(int Ind=0;Ind<=Ranges;Ind++) FromInd[Ind]=0;  // set counters to zero
  int Index=0;
  for(Index=from;Index<=to;Index++) // FromInd[i] holds number of elements in range {i-1}
    FromInd[1+(int)((Field[Index].key - Min ) * factor)]  += 1;
  FromInd[0] = from;
  for(int Ind=1;Ind<=Ranges;Ind++) // FromInd[i] holds index of subrange i
    FromInd[Ind] += FromInd[Ind-1];
  System.arraycopy(FromInd,0,ToInd,0,Ranges+1);    //  ToInd  == FromInd
  for(int Bereich=0;Bereich<Ranges;Bereich++) {    // for all subranges
    int First = FromInd[Bereich];                  // First Index of current subrange
    int Last = ToInd[Bereich+1]-1;                 // Last  Index of current subrange
    for(int Ind=First;Ind<=Last;Ind++) {           // Exchange elements from this subrange to Dest.
      int To = (int)((Field[Ind].key-Min)*factor); // comp. dest. of element in Field[Ind]
      while(To != Bereich) {                       // Until Dest. == current subrange
        DataSet Pointer = Field[FromInd[To]];      // Shift element to Destination
        Field[FromInd[To]] = Field[Ind];           // ..
        Field[Ind] = Pointer;                      // ..
        FromInd[To]++;                             // Increment Destination Index
        To = (int)((Field[Ind].key-Min)*factor);   // comp. dest. of element in Field[Ind]
    } } // All elementa are sorted into the current subrange
    First = ToInd[Bereich];                        // First is index of current subrange
    if(First<Last) {                               // if there is more than one element
      if(Last-First < DirektSortAnzahl)            // if number of elements is very small
        selection(Field,First,Last);               // use direct sorting algorithm
      else                                         // else
        DistributionSort(Field,First,Last);        // Sort this subrange with DistributionSort
} } }
```

Analysis

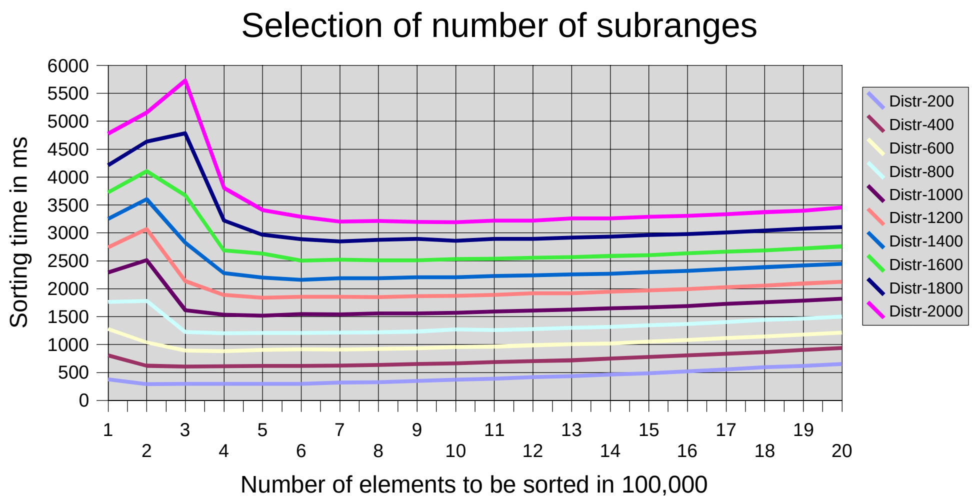
The algorithm has been tested carefully so that we will assume that it works correctly. However, the performance is of particular interest.

A first question is the optimum selection of the free parameters of our algorithm. There are two: The number of subranges and the number of elements that are to be sorted directly. The latter we will set to 20, although there are some reasons to increase this slightly, with however little influence on the sorting time.

The number of subranges has much more influence on the performance. The following figure shows this. We measured the sorting time for different number of elements to be sorted and different number of subranges. All times measured in this paper were derived from averaging sorting time of ten or more randomly generated fields with the corresponding number of elements. The programs run on a DELL Latitude C800 with 1 GHz. All programs were developed with JBuilder 7.

The results show, that there is a maximum execution time, and that there are several relativ minimums. The least minimums are at a range from about 200 subranges for 200,000 elements to about 700 subranges for 2,000,000 elements. An approximative formula for this can be like:

$$\text{number subranges} \approx 0.5 \cdot \sqrt{\text{number of elements}}.$$



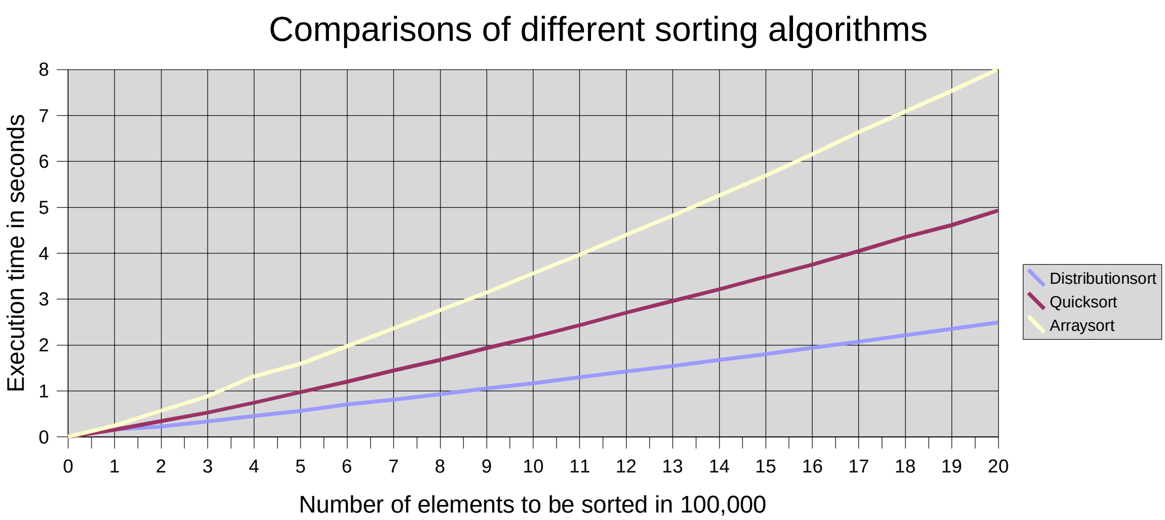
The square root is taken here since this seems to be adequate for the problem. If all subranges are equally distributed over the field (or the distribution of elements ist completely linear), then we get exactly two levels of recursion. E.g. in case of 1 Million elements we have about 2000 elements in each subrange of the first level and 4 elements in the second subrange, which is then sorted directly. From this we see that the number of elements which are to be sorted directly can vary very much without any influence on the speed of the algorithm.

This following table shows the number of subranges depending on the number of elements to be sorted.

Elements	200000	400000	600000	800000	1000000	1200000	1400000	1600000	1800000	2000000
Subranges	223,61	316,23	387,3	447,21	500	547,72	591,61	632,46	670,82	707,11

With this number of subranges one gets a suboptimal execution time with our new algorithm, Distribution Sort.

Now we compare Distribution Sort's execution time with other algorithms. We have implemented Quicksort and Arraysort, where the latter is the standard sorting algorithm of Java, which uses the interface `Comparator` to compute the order of the elements. The following diagram shows execution time against number of elements; for each array length we have sampled ten randomly generated fields and averaged the measured execution times.



The first question is how good is our algorithm compared to Quicksort. The main surprise is that the algorithm is about twice as quick as Quicksort, when programmed in Java. We also implemented this algorithm in C and found an increase of speed of about 5%. That is much less. What's the reason for this?

Analysis of Quicksort (which has been done somewhere else carefully) shows that the main reason for Quicksort's performance compared to e.g. Heapsort lies in the checks of array bounds. Quicksort compares only keys, but never array bounds, while e.g. Heapsort seems to check array bounds more often than keys. However, in Java, each array access also checks against array bounds, so that Quicksort is impeded when implemented in Java. But any way, our algorithm shows to be faster in both, C like languages with no array bound check as well as Java.

The diagram also shows that the Arraysort algorithm of Java is less fast. The reason is that they implemented Mergesort, since this is a stable algorithm, which neither Quicksort nor Distribution Sort is. However, Mergesort is usually less fast.

Other key types

Our algorithm works very fine with integer key types. For other key types, e.g. text, their may be some problems.

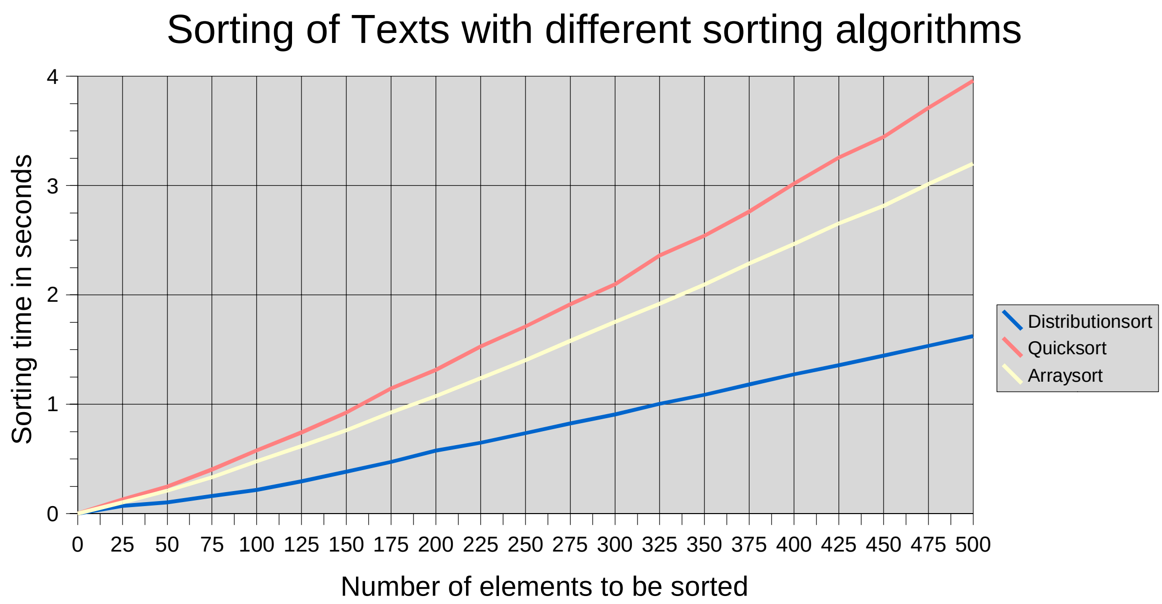
Well, text is not really critical in this algorithm, since we can use a simple method, namely compare first, second, third etc. letters of the words. Here, even the index arrays can be chosen fixed, since it seems to be appropriate to use for each letter its own subrange. Even better, one can take a mapping, which maps all characters into a number, where different characters that are ordered identically can be mapped to the same number, e.g. lower case and upper case letters or special letters (German Ä,ä as A,a, French é,ê as e etc.). Also, no minimum and maximum count is required.

The following algorithm concentrates on the principles, where a function `getInt` maps the n-th character to a number from 1 to `CharNr;` 0 is chosen for the empty character.

```
void Distribute1(int from, int to, int index) {
  int FromInd[] = new int[CharNr+2];  // Array for characters, including 0 for empty character
  for(int i=0;i<=CharNr+1;i++) FromInd[i]=0; // set counters to 0
  int count=0;
  for(int i = from;i<=to;i++) {
    if(Field[i].length>=index) count++;      // count elements with String length >= index
    FromInd[Field[i].getInt(index)+1]++;     // count destination subrange of this text
  }
  if(count==0) return;    // only empty strings, thus no sorting required
  FromInd[0] = from;      // Init pointer to first index of array
  for(int i=1;i<=CharNr+1;i++) FromInd[i] += FromInd[i-1]; // Make Distribution
  int ToInd[] = new int[CharNr+2];                       // New Field for Indices
  System.arraycopy(FromInd,0,ToInd,0,CharNr+2); // Init this Field
  for(int subrange=0;subrange<CharNr+1;subrange++) {     // for all subrange
    int Start = FromInd[subrange];                       // First Index of this subrange
    int Last = ToInd[subrange+1]-1;                      // Last Index of this subrange
    for(int Ind=Start;Ind<=Last;Ind++) {                 // Exchange all elements into its subrange
      int To = (int)((Field[Ind].getInt(index)));        // To is index of Destination subrange
      while(To != subrange) {                            // while Dest. subrange != subrange
        int ToIndex = FromInd[To];                       // ToIndex point to Dest. index
        DataSet Pointer = Field[ToIndex];                // aux. pointer of element
        Field[ToIndex] = Field[Ind];                     // copy element to destination
        Field[Ind] = Pointer;                            // copy dest. element
        FromInd[To]++;                                   // Increment index of dest. subrange
        To = (int)((Field[Ind].getInt(index)));          // To is index of Destination subrange
     } }
    // All elements are in the current subrange
    Start = ToInd[subrange];
    if(Start<Last) {  // if not yet completely sorted
      if(Last-Start < DirectSortCount) { // Sort direct, if too less elements
        Select(Start,Last);
      } else {
        Distribute1(Start,Last,index+1); // Sort this subrange with DistributionSort
} } } }
```

In the first loop the algorithm counts the number of key texts the length of which is not bigger than `index`. If there is none the procedure is finished, since no sorting is required. This is critical for correct termination of the program.

Performance measurement showed that this program is more than twice as fast as Quicksort, if implemented in Java. We used randomly generated texts with random lengths between 1 and 50 symbols, with 90 different symbols. The length of the arrays differed from 25,000 to 500,000, where again for each array length ten samples have been averaged.



Conclusion

Our algorithm is twice as good as Quicksort, although this depends on the implementation language. It can be used for sorting of records with text keys or numerical keys.

An important result of the analysis shows that Java seems to impede Quicksort since Quicksort's functionality does not require array bounds checks, while Java tests each array access against bounds. Thus the conclusion of this observation is that the speed of any algorithm depends critically on the programming language it is implemented in.