A LOGICAL INVERTED TAXONOMY OF SORTING ALGORITHMS

S.M. Merritt
School of Computer Science and Information Systems
Pace University
1 Martine Avenue
White Plains, NY 10606
USA
merritt@pacevm.dac.pace.edu

K.K. Lau
Department of Computer Science
University of Manchester
Oxford Road
Manchester M13 9PL
UK
kkl@cs.man.ac.uk

Abstract

Sorting is still one of the most important problems in Computer Science. Work in transformational programming and automatic program synthesis provided the insight that led to Merritt's *inverted taxonomy* of sorting algorithms, a high-level, top-down, conceptually simple and symmetric categorization of sorting algorithms.

More recent work in logic-based program synthesis by Lau has produced a *logical taxonomy* of sorting algorithms. This provides a logical basis for the inverted taxonomy and expands it into a *logical inverted taxonomy* to include distributive sorting algorithms which can be derived along with comparison-based algorithms. The inclusion of distributive algorithms into a unified conceptual framework is new and significant for a comprehensive perspective on sorting algorithms.

In this paper, we describe both the inverted and the logical taxonomies and show how the latter strengthens the latter and expands it into a logical inverted taxonomy of sorting algorithms, a high-level, top-down, symmetrical paradigm for all sorting algorithms.

1 Introduction

The traditional taxonomy of sorting algorithms, which follows that presented in (Knuth 1973), divides sorting algorithms into three categories: insertion, selection and exchange, according to their main operational characteristics. Naturally, the canonical examples of these categories are: *insertion sort*, *exchange sort* and *selection sort* respectively. In this basic classification, more sophisticated algorithms such as *Shell sort*, *heapsort* and *quicksort* are presented as optimization of these basic algorithms; merging is treated as a fourth but limited approach to sorting two already sorted sequences. Figure 1 illustrates the traditional taxonomy.

Although this description of sorting is low-level and bottom-up, it is still widely accepted as standard even though we have become much more serious about top-down design, structured programming, high-level programming languages, algorithm design techniques and software engineering. In (Merritt 1985), Merritt noted the irony that sorting, a classic problem in computer science (and in programming in particular) continues to be understood and presented in such a bottom-up manner. More importantly, she proposed an alternative taxonomy which is based on a higher level, more abstract (and yet conceptually simple) top-down approach to sorting. She called this taxonomy the *inverted taxonomy*.

In this paper, we first give a brief review of the inverted taxonomy. Then we strengthen it by giving it a logical basis and expand it by adding a new category. We call the expanded taxonomy the *logical inverted taxonomy*.

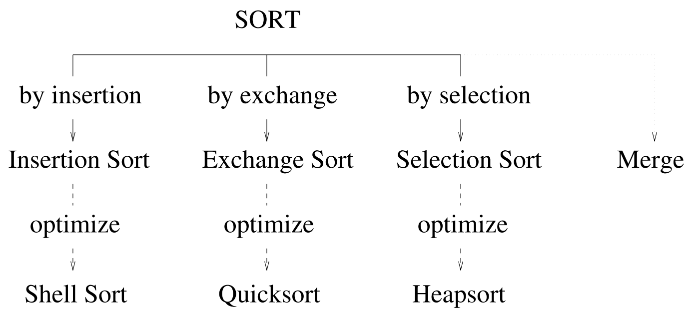

Figure 1: The traditional taxonomy of sorting algorithms.

2 The Inverted Taxonomy of Sorting Algorithms

The alternative taxonomy proposed in (Merritt 1985) was inspired by the work in program synthesis in (Clark and Darlington 1980), (Darlington 1978), (Green and Barstow 1978), and (Barstow 1980). The basis of this classification is to recognize sorting as a ‘split’ and ‘join’ procedure: given a set of things to sort, split the set into two parts; recursively sort each part; and finally join the two parts into a sorted set.

Such a description is top-down and incorporates the principle of stepwise refinement. The sorting problem is decomposed essentially into a ‘split’ and a ‘join’ procedure without prescribing exactly how to split or how to join. Two examples are merge sort and quicksort; the difference between them is in the split and in the join. Green and Barstow give an example similar to that shown in Figure 2.

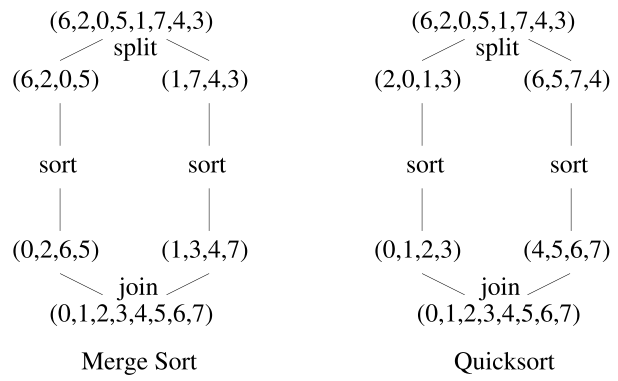

Figure 2: Merge sort and quicksort.

As the example clearly shows, in merge sort the work is done in the joining, not in the splitting (which is trivial); whereas in quicksort the work is done in the splitting, but the joining is trivial. Therefore we claim that all sorts can be divided into two categories: hardsplit/easyjoin and easysplit/hardjoin, of which quicksort and merge sort are the respective canonical examples.

Although it is natural to think about a ‘split’ in terms of equal-size parts, it is easy to see that if merge sort splits off a singleton, it collapses into insertion sort. Similarly, quicksort collapses into selection sort if it splits off a singleton. Moreover, sinking sort (Barstow 1980) and bubble sort can be understood as in-place versions of insertion sort and selection sort respectively. These observations lead to the inverted taxonomy shown in Figure 3.

Since their publication in (Merritt 1985), these observations have been incorporated into at least one elementary textbook (Schneider and Bruell 1987). Also, since then, new work in deriving sorting algorithms in a top-down manner by logical deduction has provided the inverted taxonomy with a logical basis and expanded it.

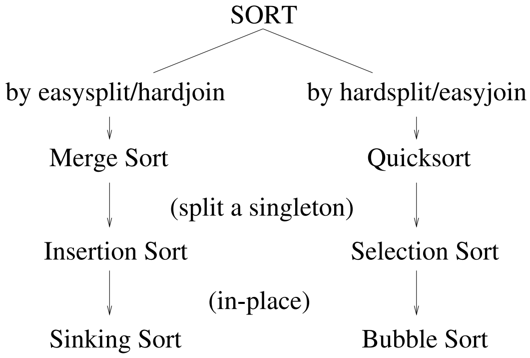

Figure 3: The inverted taxonomy of sorting algorithms.

3 A Logical Basis for the Inverted Taxonomy

A new top-down synthesis of sorting algorithms by Lau (Lau 1989 and 1992), that is logic-based, strengthens the inverted taxonomy by deriving comparison-based sorting algorithms that indeed fall into the two categories of hardsplit/easyjoin and easysplit/hardjoin. Moreover, it expands the taxonomy by deriving distributive algorithms in a symmetric way.

This approach is based on logic programming, so an algorithm is represented by a set of logic clauses, i.e. formulas of the form

$$\begin{array}{ccc} p & \leftarrow & q_1 \\ & \vdots & \\ p & \leftarrow & q_n \end{array}$$

where p is a predicate and $q_1, \cdots, q_n$ are conjunctions of predicates. These clauses are usually recursive.

For example, a possible set of logic clauses for merging two lists is

$$\begin{array}{lcl} merge(x.c,\ y.d,\ x.t) & \leftarrow & x < y \wedge merge(c,\ y.d,\ t) \\ merge(x.c,\ y.d,\ y.t) & \leftarrow & y < x \wedge merge(x.c,\ d,\ t) \end{array} \tag{1}$$

where $h.t$ represents the list whose head is h and whose tail is the list t. Note that for simplicity and clarity we have written $x < y$ and $y < x$ in their common form rather than as predicates.

These clauses embody the logic of merging two lists recursively by taking one element at a time from one of the lists and adding it to the 'current' merged list. The first clause means: If merging the list c and the list $y.d$ gives the list t, and the element x is less than the element y, then merging the list $x.c$ and the list $y.d$ gives the list $x.t$. It expresses the logic that underlies the case where the first element of the first list $x.c$ is added to the 'current' merged list t to form the 'new' merged list.

Similarly, the second clause expresses the logic of the other case where the first element of the second list $y.d$ is added to the 'current' merged list to form the 'new' merged list.

Using these clauses for merging two lists, the clause defining merge sort is

$$sort(a_1|a_2, b) \leftarrow sort(a_1, c) \wedge sort(a_2, d) \wedge merge(c, d, b) \tag{2}$$

where $a|b$ means the concatenation of the lists a and b.

To derive clauses for different sorting algorithms, the method starts from a general definition of sorting using the following logical formula:

$$sort(a, b) \leftrightarrow perm(a, b) \wedge ord(b) \tag{3}$$

where a and b are general lists. The meaning of this definition is: The list b is the sorted version of the list a if and only if b is a permutation of a, and b is ordered. We omit here the other logical formulas which in turn define $perm$ and ord (and other predicates which these formulas may contain).

From this given set of definition formulas, the method can derive different sets of logic clauses, i.e. different sorting algorithms, by logical deduction. The derivation is performed on a logic programming system (Lau and Prestwich 1990) under user-guidance. For each derivation, the user can specify the form of the resulting clauses, in particular the form(s) of the recursive call(s). Thus by specifying different forms for the clauses, the system can be made to derive different clause sets corresponding to different sorting algorithms.

For example, the clause (2) for merge sort can be derived from (3) if the user specifies its form as:

$$sort(a_1|a_2, b) \leftarrow \cdots \wedge sort(a_1, c) \wedge sort(a_2, d) \tag{4}$$

where $(\cdots)$ stands for some unknown conjunction of predicates.

Specifying this form amounts to expressing the design decision to aim for an algorithm that splits the input list a into $a_1|a_2$ (at some pre-specified position) and recursively sorts a_1 and a_2 separately. Logic programming provides a natural framework for expressing such design decisions and for deriving clauses that satisfy them. Incomplete clauses of the form in (4) can be expressed as *goals*, which when *solved* yield the required clauses. Solving a logic programming goal involves instantiating variables or unknowns to other forms or values. For instance, in this example, as a result of the derivation, we get the clause

$$sort(a_1|a_2, b) \leftarrow (perm(c|d, b) \wedge ord(b)) \leftarrow (ord(c) \wedge ord(d)) \wedge sort(a_1, c) \wedge sort(a_2, d)$$

which is (4) in which $(\cdots)$ has been instantiated to

$$(perm(c|d, b) \wedge ord(b)) \leftarrow (ord(c) \wedge ord(d)) \quad . \tag{5}$$

Now the meaning of (5) is: The list b is a permutation of the list $c|d$ and is ordered if c and d are ordered; that is, The list b is a *merge* of the lists c and d'. So we can define (5) as a new predicate $merge$ by

$$merge(c, d, b) \leftrightarrow (perm(c|d, b) \wedge ord(b)) \leftarrow (ord(c) \wedge ord(d))$$

which in effect defines the logic of the merge of two lists. As a result, we get the clause

$$sort(a_1|a_2, b) \leftarrow sort(a_1, c) \wedge sort(a_2, d) \wedge merge(c, d, b)$$

for merge sort. Note that this clause only defines the logic of merge sort but does not say how the merging itself is to be done. To do so, we need to derive the clauses in (1).

To derive clauses for merging two lists, the user can specify their forms as:

$$\begin{array}{lcl} merge(x.c, d, x.t) & \leftarrow & \cdots \wedge merge(c, d, t) \\ merge(c, y.d, y.t) & \leftarrow & \cdots \wedge merge(c, d, t) \end{array} \tag{6}$$

expressing the design decision to aim for an algorithm that takes one element at a time from one of the lists and adding it to the 'current' merged list t. The resulting clauses are precisely those in (1), that is the clauses (6) in which d and c become instantiated to $y.d$ and $x.c$, and the $(\cdots)$ part becomes instantiated to $x < y$ and $y < x$ (in the first and the second clause respectively).

This shows another important feature of the derivation, namely that it is performed in a top-down manner. A goal is decomposed into sub-goals, which are in turn decomposed, and so on, and is solved when all the sub-goals have been solved.

As a result, a family of sorting algorithms has been derived, as shown in Figure 4 in the next section. This tree incorporates a top-down logic-based classification of sorting algorithms which incorporates the inverted taxonomy described in the previous section. Therefore it provides the inverted taxonomy with a logical basis, and thus strengthening it. Moreover, this new tree contains an extra category of algorithms, namely distributive algorithms. We will show that the inverted taxonomy can be naturally expanded to include this new category as well.

4 A Logical Taxonomy of Sorting Algorithms

Returning to the example in the previous section, if in (2), the clause for merge sort, the first sublist is $[a_1]$, i.e. it is a singleton list, then (2) becomes

$$sort(a_1.a_2, b) \leftarrow sort(a_2, d) \wedge merge([a_1], d, b) \quad . \tag{7}$$

This clause now defines insertion sort since merging the lists $[a_1]$ and d is equivalent to inserting the element a_1 into the list d.

Therefore, the synthesis method reveals this logical relationship between merge sort and insertion sort. (Note that this is equivalent to the 'singleton-split' relationship between these algorithms in the inverted taxonomy.) Furthermore, splitting the input list into $a_1|a_2$ is done at a pre-specified position in the input list (usually the mid-point for merge sort). Consequently, merge sort and insertion sort are classified as 'split by position' algorithms.

By deriving other algorithms in a similar manner, we arrive at a logical taxonomy of sorting algorithms which is shown in Figure 4.

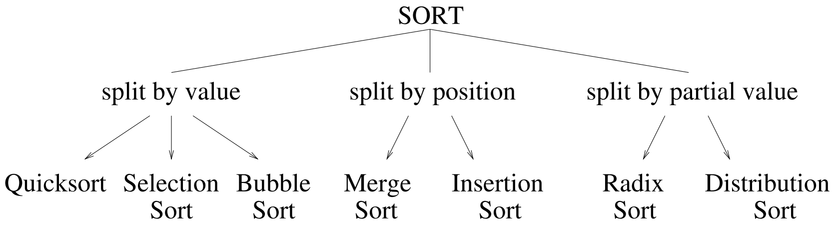

Figure 4: The logical taxonomy of sorting algorithms.

Quicksort, *selection sort* and *bubble sort* are classified as 'split by value' because they split the input list in such a way that the all the elements of the first sublist are less than all the elements of the second sublist. In the clause for *quick sort*, if the first sublist is a singleton list, then the clause defines *selection sort*. This logical relationship is again equivalent to the 'singleton-split' relationship between these algorithms in the inverted taxonomy.

Therefore, this taxonomy largely coincides with the inverted taxonomy. 'Split by position' is clearly an 'easy split'. It is followed by a merge which is clearly a 'hard join'. Similarly, 'split by value' splits the list according to the values of its elements, and is therefore a 'hard split'. It is followed by list concatenation, which is obviously an 'easy join'.

There are two differences between the inverted and the logical taxonomy. The first is that in the latter no algorithms are classified as 'in-place'; this is just not possible in a logic programming framework. So, *sinking sort* does not appear in the taxonomy. However, *bubble sort* is present, and its logical property can be interpreted as being an 'in-place' version of *selection sort*.

The second difference is that the logical taxonomy has an extra category, namely distributive algorithms. This class is called 'split by partial value' because splitting is done according to the lexicographical ordering of the elements of the input list.

The first difference is not significant, and in the next section we will show that the second difference can be eliminated by expanding the inverted taxonomy naturally to include the new category.

5 A Logical Inverted Taxonomy

The logical taxonomy inspires an extension of the generalized split/join paradigm that includes distributive sorting algorithms such as *distribution sort* and *radix sort* (Knuth 1973). The extension divides sorting algorithms into three categories (instead of two); to easysplit/hardjoin and hardsplit/easyjoin we add easysplit/easyjoin. In general, items represented in binary form can be sorted by distributing the numbers into one of two "buckets": one of the buckets is for those numbers with least (most) significant bit 0 and the other is for those numbers with least (most) significant bit 1. Each bucket is then sorted recursively in the next bit. If the distribution is made on the least significant bit, the algorithm is *binary distribution sort* (Knuth 1973). If the distribution is made on the most significant bit, the algorithm is radix exchange sort [8]. These algorithms nicely fall into the split, sort recursively, and join paradigm, and are shown in Figure 5.

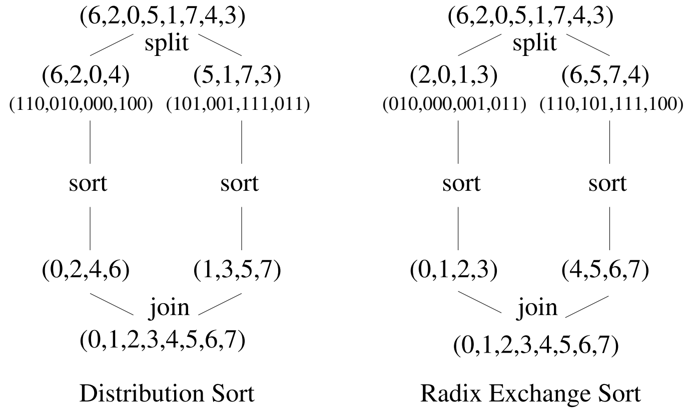

Figure 5: Distribution sort and radix exchange sort.

Other distributive sorting algorithms can be understood as generalizations of the distribution sort. Bucket sort (Aho *et al* 1974) is a distribution sort that is not binary. For example, decimal numbers can be sorted with ten buckets; items are distributed to buckets without compares, and then collected from left to right. Multi-digit decimal numbers can be sorted by a first pass that distributes items based upon the least significant digit, and subsequent passes that distribute on the next most significant digits (Aho *et al* 1974). Address calculation sorting (Knuth 1973) might be understood as a bucket sort in which buckets are intervals of some range of distribution of elements. In the same way that quicksort, for example, is representative of a large class of algorithms of the hardsplit/easyjoin type, including selection sort and heapsort, distribution sort is representative also; binary distribution sort might be considered the canonical example of distributive sorting algorithms.

Clearly, for binary distribution sort, both the "split" and "join" are easy, thereby adding the third category, "easysplit/easyjoin" to the inverted taxonomy. Figure 6 shows the expanded taxonomy, which we call the logical inverted taxonomy, a simple and symmetrical top-down categorization of sorting algorithms, including distributive sorting algorithms.

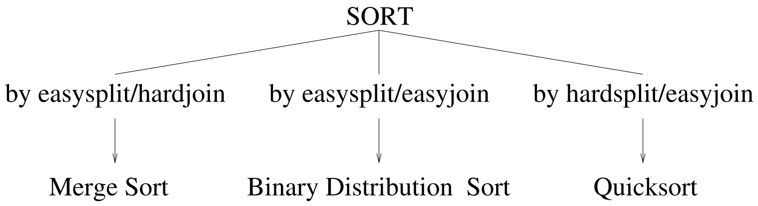

Figure 6: The logical inverted taxonomy of sorting algorithms.

The effective application of distributive sorting algorithms is limited to sets of data that are "predictable", that is, uniformly subordered (multi-digit integers, multi-character codes) or uniformly distributed over a range. In such cases the expected running time is $O(n)$. However, the worst case time for these algorithms is proportional to n^2 (Knuth 1973). It is interesting to note that hybrid algorithms have been developed that are basically distributive (in order to achieve linear expected time), but that invoke a comparison sort as a second phase (in order to insure $O(nlogn)$ worst case time). One such algorithm, *distributive partitioning* (Dobosiewicz 1978) uses quicksort as the second-phase sort. A variation on distributive partitioning, sometimes called *distributive merging* (Van der Nat 1980), uses merge sort. The hybrids can be placed easily in the logical inverted taxonomy and are shown in Figure 7. Note that

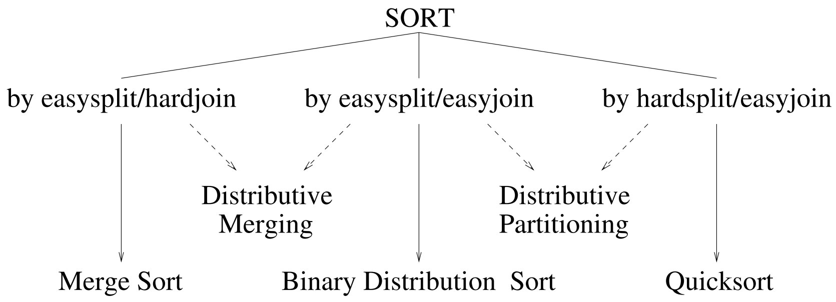

Figure 7: The logical inverted taxonomy with hybrids.

except in the very worst case for the hybrid algorithms, the join (in the case of merge sort) or the split (in the case of quicksort) is not quite as "hard" because of the prior distribution.

6 Conclusion

Sorting is still one of the most important problems in computer science. Work in transformational programming, and automatic program synthesis provided the insight that led to Merritt's inverted taxonomy of sorting algorithms, a high level, top-down, conceptually simple and symmetric categorization of sorting algorithms (Merritt 1985).

More recent work in logic-based program synthesis by Lau has produced a logical taxonomy of sorting algorithms. The logical taxonomy provides a logical basis for the inverted taxonomy, and expands it to include a third category, distributive sorting algorithms, which can be derived along with comparison-based algorithms.

In this paper we show the logical taxonomy of sorting algorithms. The logical taxonomy strengthens the inverted taxonomy by demonstrating the logical derivation: split by value yields the hardsplit/easyjoin category of the inverted taxonomy, and split by position yields the easysplit/hardjoin. Moreover, split by partial value yields a category of distributive sorting algorithms; we note that the new category can be understood as an easysplit/easyjoin expansion of the inverted taxonomy. We also place hybrid algorithms (distributive and comparison-based) into the scheme.

Therefore, the logical taxonomy of sorting algorithms is equivalent to an expansion of the inverted taxonomy into a logical inverted taxonomy, a high-level, top-down, symmetrical paradigm for all sorting algorithms. In particular, the logical inverted taxonomy unifies comparison-based and distributive sorting. This is an important result for educators and practitioners for understanding and teaching about sorting (Merritt 1994).

References

Aho, A.V., Hopcroft, J.E. and Ullman, J.D., (1974), *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Barstow, D.R., (1980), Remarks on "A Synthesis of Several Sorting Algorithms" by John Darlington, *Acta Informatica*, Vol. 13, pp. 225–227.

Clark, K.L. and Darlington, J., (1980), Algorithm Classification through Synthesis, *The Computer Journal* Vol. 23, No. 1, pp. 61–65.

Darlington, J., (1978), A Synthesis of Several Sorting Algorithms, *Acta Informatica*, Vol. 11, pp. 1–30.

Dobosiewicz, W., (1978), Sorting by Distributive Partitioning, *Information Processing Letters*, Vol. 7, No. 1, pp. 1–6.

Dromey, R.G., (1987), Derivation of Sorting Algorithms from a Specification, *The Computer Journal*, Vol. 30, No. 6, pp. 512–518.

Green, C. and Barstow, D., (1978), On Program Synthesis Knowledge, *Artificial Intelligence*, Vol. 10, pp. 241–279.

Knuth, D.E., (1973), *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*, Addison-Wesley.

Lau, K.K., (1989), A Note on Synthesis and Classification of Sorting Algorithms, *Acta Informatica*, Vol. 27, pp. 73–80.

Lau, K.K. and Prestwich, S.D., (1990), Top-down Synthesis of Recursive Logic Procedures from First-order Logic Specifications, in D.H.D. Warren and P. Szeredi, editors, *Proc. 7th Int. Conf. on Logic Programming*, pp. 667–684, MIT Press.

Lau, K.K., (1992), Top-down Synthesis of Sorting Algorithms, *The Computer Journal*, Vol. 35, pp. A001–A007.

Merritt, S.M., (1982), The Role of High Level Specification in Programming by Transformation: Specification and Transformation by Parts, Ph.D. dissertation, Dept of Computer Science, New York University.

Merritt, S.M., (1985), An Inverted Taxonomy of Sorting Algorithms, *Comm. ACM*, Vol. 28, No. 1, pp. 96–99.

Merritt, S.M., (1994), An Expanded Taxonomy of Sorting Algorithms, *Computer Science Education*, Vol. 5, pp. 103–110.

Schneider, G.M. and Bruell, S.C., (1987), *Advanced Programming and Problem Solving with Pascal*, John Wiley & Sons.

Smith, D.R., (1985), Top-down Synthesis of Divide-and-conquer Algorithms, *Artificial Intelligence*, Vol. 27, pp. 43–96.

Smith, D.R., (1985), The Design of Divide-and-conquer Algorithms, *Science of Computer Programming*, Vol. 5, pp. 37–58.

Van der Nat, M., (1980), Fast Sorting Algorithm, a Hybrid of Distributive and Merging Sort, *Information Processing Letters*, Vol. 10, No. 3, pp. 163–167.