1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

Complexity and Guarantee of Permutation Assertion for Sorting

Pei-Chi Wu

Department of Information Management
Department of Computer Science and Information Engineering
National Penghu Institute of Marine & Management Technology
300 Liu-Ho Road, Makung City, Penghu, Taiwan 880, R.O.C.
Email: pcwu@npit.edu.tw

ABSTRACT

Assertions for sorting functions are widely used as an example in discussions of software specification and fault-tolerance. Complete checking for sorting needs two assertions: *order* and *permutation*. The order assertion checks whether the output array monotonically increases (or decreases). The permutation assertion checks whether the output array contains all the elements in the input array. The order assertion is simple and fast, whereas permutation assertion is considered to be time-consuming. This paper gives a detailed analysis of the time complexity of the permutation assertion, which is shown $O(n \log n)$. This complexity is no simpler than that of sorting. This paper then presents a method that guarantees the correctness of a sorting procedure by replacing assignments with swap operations on any two array elements. The technique can be applied to sorting algorithms that are in-place. This shows that complete checking for sorting can be achieved in linear time.

Keywords: complete assertions, time complexity, software engineering, assignments, swap operations, fault-tolerant software.

1. INTRODUCTION

The sorting problem is well-defined and has practical importance. It is also used in many software engineering textbooks to introduce the concept of program assertions. Sorting assertions are widely referred to in discussions of software specification [2, 6, 7] and fault-tolerance [8, 10]. A set of assertions for a program is said to be *complete*, if satisfying the set of assertions guarantees the correctness of the program. Complete checking for sorting needs two assertions: *order* and *permutation* [6, 9]. Let a sorting function be denoted as *Sort*: array $\rightarrow$ array. The order assertion checks whether the output array monotonically increases (or decreases). The permutation assertion checks whether the output array contains all the elements in the input array, i.e., whether the output array is a permutation of the input array.

Since run-time assertions take overhead, their efficiency is important. Order assertion is simple and fast: its time complexity is $O(n)$. On the other hand, permutation assertion is considered to be time-consuming and rarely used at run-time [8]. The concept of check-sum assertion was addressed in Randell [8]. Since check-sum assertion with order assertion is incomplete, its power to detect errors is further discussed in Saxena and McCluskey [9], where the complexity of permutation assertion is also said to be $O(n \log n)$, but no further analysis is given. Because permutation assertion inefficiency is the only reason to adopt an incomplete sorting assertion, a detailed analysis of permutation assertion complexity may be needed. Here we give such a detailed analysis. We present an $O(n \log n)$ algorithm for permutation assertion and a lower bound of $O(n \log n)$, showing the exact complexity of permutation assertion to be $O(n \log n)$. This complexity is no simpler than that of sorting, so it is not attractive to perform permutation assertion at run-time.

Permutation assertion is central to guaranteeing the correctness of a sorting procedure. One may ask: "Why did permutation assertion fail in a sorting procedure?" The reason is that many sorting procedures directly assign values to the array elements, which may arbitrarily put an erroneous value on an array element. Such operations are very dangerous. This paper presents a method that guarantees the correctness of a sorting procedure by replacing these assignments with swap operations on any two array elements. The technique can be applied to sorting algorithms that are in-place, including insertion sort, selection sort, quick sort, and heap sort. This shows that complete checking for sorting can be achieved in linear time.

2. AN O(n log n) ALGORITHM

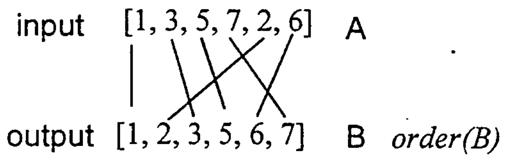
Permutation assertion has also been called the *multiplicity equality* [6] of two arrays. Consider the array A=[1, 4, 3, 5, 9, 4] and the sorted array B=[1, 3, 4, 4, 5, 9], B = *Sort*(A). The permutation assertion not only checks whether all distinct elements of B, i.e., {1, 3, 4, 5, 9}, are included in A, but also checks whether there are the same number of copies of each distinct element in both A and B. Let *ArrayBag* be a conversion of an array to a *bag* (also called a *multi-set*). That is, *permutation*(A, B) holds if and only if *ArrayBag*(A) = *ArrayBag*(B). Since order assertion is very simple, we assume that the assertion *order*(B) is executed and already holds before the permutation assertion is executed, and B thus monotonically increases (or decreases). Adding this pre-condition is more realistic when discussing the time complexity of permutation assertion, because it may lead to a simpler and more efficient algorithm. In addition, it is almost meaningless to execute permutation assertion after the *order*(B) assertion has already failed.

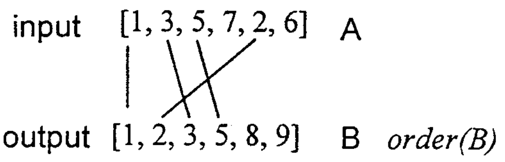
The following shows the Algorithm *permutation-assert*. Each element of A is checked to see whether it is also in *tempB*, a copy of B. Each element in *tempB* is "marked" when it is matched with an element in A, such that it will never match other element in A. For each element in A, the algorithm applies a binary search of array *tempB*. A binary search takes $O(\log n)$ for array size n. The time complexity of the algorithm is then $O(n \log n)$.

```
Algorithm permutation-assert (A, B).
Input. arrays A and B, where order(B) holds and n =
|A|, the size of A.
Output. whether B is a permutation of A.
Begin
   tempB := B; {* array copy *}
   for i from 1 to n do
      index := binary_search(A[i], tempB);
      {* Find a non-marked elements of tempB
            whose value is A[i].
         Return index of A[i] in tempB; return 0
            otherwise.
      *}
      if (index>0) then
         mark tempB[index];
         {* This element will never match again in
               the binary search. *}
      else
         return false;
      end if
   end do
   return true;
End of Algorithm.
```

3. A LOWER BOUND

The problem of permutation assertion can be formulated as a process of identifying the relationship between the input and the output arrays of a sorting function. An element in A that relates to an element in B is indicated by a line between them. There are no lines between unmatched elements. Figure 1 shows a relationship where the permutation assertion holds. The output array is correct if the relation is *one-to-one* and *onto*. Figure 2 shows an example which output is incorrect: Although the output array is already ordered, the input elements 6 and 7 are missing in the output. The elements 8 and 9 are also erroneously inserted to the output array.


input [1, 3, 5, 7, 2, 6] A
output [1, 2, 3, 5, 6, 7] B order(B)
Figure 1. An example of the relationship between the input and output arrays.


input [1, 3, 5, 7, 2, 6] A
output [1, 2, 3, 5, 8, 9] B order(B)
Figure 2. An example of relationship where the permutation assertion has failed.

The arrays in Figures 1 and 2 contain only distinct numbers. Figure 3 shows two relationships where there are two 3's in the input. The outputs may or may not preserve the order of the input. This property is called the *stability* [3] of sorting. To simplify the analysis, these two relationships will be treated as two "decisions." That is, all numbers in the input array are treated as distinct.

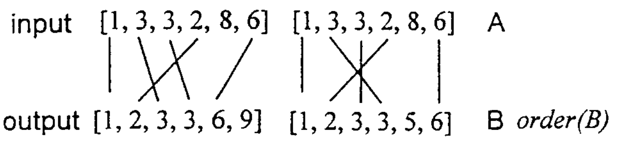

input [1, 3, 3, 2, 8, 6] [1, 3, 3, 2, 8, 6] A
output [1, 2, 3, 3, 6, 9] [1, 2, 3, 3, 5, 6] B order(B)

Figure 3. Two examples of relationships containing two 3's.

Based on this formulation, the permutation assertion is then a decision problem requiring selection of a candidate from all possible relationships (leaves of the decision tree). The time complexity of permutation assertion can be determined by the height of the decision tree, which is a logarithm of the number of decision tree leaves. Consider a pair of input and output arrays. Step 1) Choose as r elements of the input that are also found in the output: in total there are $C(n, r)$ combinations, $0 \le r \le n$. Step 2) For these r elements, choose r positions from the output array (there are n positions): the combination is $C(n, r)$. For example, the arrays in Figure 2, $n = 6$, $r = 4$, $C(n, r) = C(6, 4) = 15$. Step 3) Assign these r elements to r positions; because B is already ordered, this assignment is unique.

Let $PERM(n)$ be the number of possible relationships for array size n. Since the number of combinations in Step 1 and 2 are both $C(n, r)$, $PERM(n)$ is then the summation of $C(n, r)\cdot C(n, r)$, $0 \le r \le n$.

$$PERM(n) = \sum_{r=0}^{n} C(n,r)\cdot C(n,r)$$
$$= \sum_{r=0}^{n} C(n,r)\cdot C(n,n-r) = C(2n,n)$$

The following is an example for $n = 2$:

Let array $A = [4, 3]$.
$PERM(n) = C(2, 2)\cdot C(2, 2) + C(2, 1)\cdot C(2, 1) + C(2, 0)\cdot C(2, 0)$
$= 1 + 4 + 1 = 6.$

Let X represent any erroneous number inserted to the output array. There are 6 cases as shown below:

$r = 0$: $B =$ [X, X].
$r = 1$: $B =$ [3, X], [X, 3], [4, X], [X, 4].
$r = 2$: $B =$ [3, 4].

The following is the detailed computation of the complexity order of $PERM(n)$. It shows that the lower bound of the permutation assertion is $O(n \log n)$:

$$C(2n,n) = (2n)!/n!n!$$

Stirling's formula: $\log n! \approx (n + 1/2)\log n - n + \delta$

$$O(\log C(2n,n)) = O(\log((2n)!) - 2\log(n!)) = O(n\log n)$$
$$\therefore O(PERM(n)) = O(n\log n)$$

Which is more complex: sorting or permutation assertion? The decision tree for sorting has $n!$ leaves and the logarithm of $n!$ is also $O(n \log n)$ [3]. Table 1 shows the first 12 numbers of $C(2n, n)$ and $n!$. When $1 \le n \le 6$, $C(2n, n) > n!$. However, when $n \ge 7$, $C(2n, n) < n!$. Permutation assertion seems to be simpler than sorting when considered this perspective.

Table 1. First 12 numbers of C(2n, n) and n!.

n	$C(2n, n)$	$n!$
1	2	1
2	6	2
3	20	6
4	70	24
5	252	120
6	924	720
7	3432	5040
8	12870	40320
9	48620	362880
10	184756	3628800
11	705432	39916800
12	2704156	479001600

4. ArraytobeSorted: An Array with Swap Operations

To support safe operations on an array, a new array type called *ArraytobeSorted* is proposed. *ArraytobeSorted* provides swap operations on any two array elements, and allows only one way of writing an element. A series of swap operations on an *ArraytobeSorted* makes a permutation of the array. Thus, the permutation assertion holds immediately. Simply applying order assertion can then guarantee the correctness of the sorting procedure.

Figure 4 shows the interfaces of array (`Array`) and *ArraytobeSorted* (`ArraytobeSorted`). All these codes are in C++ [1] class templates, where the type parameter `T` represents the element type of the array. Template `Array` is a definition of an ordinary array. The `operator[]()` returns a reference to an array element. Each element can be read and written via the returned reference. Template

ArraytobeSorted provides swap() to swap any two array elements, operator[]() to get the value of an element, and size() to get the number of elements in an array. An ArraytobeSorted is created by giving an ordinary array to its constructor ArraytobeSorted(). The given ordinary array is not copied: only a reference is kept in the constructed ArraytobeSorted. Thus, procedures that operate on ArraytobeSorted change the ordinary array. Except using swap(), there is no other way to write an element.

```
template<class T>
class Array
{
public:
  Array(int s);
  Array(Array& a);
  T& operator[] (int i) const;
  int size() const;
};

template<class T>
class ArraytobeSorted
{
public:
  ArraytobeSorted(Array<T>& a) : A(a) { }
  int size() const { return A.size(); }
  T operator[] (int i) const
    { return A[i]; }
  void swap(int i, int j)
    {
      T temp;
      temp=A[i];
      A[i]=A[j];
      A[j]=temp;
    }
private:
  Array<T>& A;
};
```

Figure 4. The interfaces of Array and ArraytobeSorted.

Figure 5 shows an example of using the ArraytobeSorted interface. Procedure selection_sort is an implementation of the selection sort algorithm [5]. The inner loop of selection_sort finds a minimal element from an unordered part of an input array. The minimum found is put in min and its array index is put in index. The procedure then swaps the minimum with the first element using swap(i, index).

```
template<class T>
void selection_sort(ArraytobeSorted<T>& A)
{
  int size = A.size();
  for(int i=0; i<size; i++)
  {
    T min = A[i];
    int index=i;
    for(int j=i+1; j<size; j++)
      if (A[j]<min) { min=A[j]; index=j; }
    A.swap(i, index);
  }
}
```

Figure 5. Code of selection_sort using ArraytobeSorted.

5. APPLICABILITY TO SORTING ALGORITHMS

The technique introduced in Section 4 can be applied to many sorting algorithms. For example, the codes given in [5] for *quick sort* and *heap sort* already use swap (or called "exchange") operations on two array elements. Using ArraytobeSorted on these algorithms is straightforward.

The technique is also applicable to *insertion sort* algorithm. The insertion sort algorithm inserts an element into the array each iteration. There are two steps: 1) find the proper position for the new element, and 2) insert the element into the array. Step 1 applies a sequential or binary search. Step 2 needs to shift each element that is greater than the inserted element by one position. This step is usually done using a series of assignments on array elements, which may cause problems such as duplicated or missing elements. Fortunately, this step can also be done using a series of swap operations on array elements. Figure 6 shows the insertion_sort code. The code uses sequential search and swapping elements during searching.

```
template<class T>
void insertion_sort(ArraytobeSorted<T>& A)
{
 int size = A.size();
 for(int i=1; i<size; i++)
 {
  int index=i;
  for(int j=i-1; j>=0; j--)
   if (A[j]<A[index])
    break;
   else
   {
    A.swap(index,j);
    index=j;
   }
 }
}
```

Figure 6. Code of `insertion_sort` using `ArraytobeSorted`.

This technique cannot be applied to sorting algorithms that need additional storage. For example, *bucket sort* and *radix sort* use buckets to store elements, and *merge sort* uses a temporary array when merging two sorted arrays. The array elements are temporarily stored in these additional storage and later moved to the array. These operations cannot be conducted using `ArraytobeSorted`.

Table 2 summarizes the applicability of the technique to various sorting algorithms. The technique can be applied to sorting algorithms that are in-place, including insertion sort, selection sort, quick sort, and heap sort.

Table 2. Applicability of swapping to various sorting algorithms.

Algorithm	Ap.	Reasons
insertion sort	√	shift data by a series of swapping
selection sort	√	swap the minimum with first element
bucket sort	×	need a separate storage for buckets
radix sort	×	need a separate storage for buckets
merge sort	×	need a temporary storage
quick sort	√	directly using swapping
heap sort	√	directly using swapping

Ap. = Applicability

6. Conclusions

The exact time complexity of permutation assertion has been proven to be $O(n \log n)$ by giving an $O(n \log n)$ algorithm and an $O(n \log n)$ lower bound. This paper has also presented an array interface which guarantees swapping elements to be the only way to change an array. Using this kind of arrays in a sorting procedure, the permutation assertion holds immediately, and simply applying order assertion guarantees the correctness of the sorting procedure. This shows that complete checking for sorting can be achieved in linear time. This technique can be applied to algorithms such as insertion sort, selection sort, quick sort, and heap sort.

References

[1] M. A. Ellis, and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[2] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice-Hall, 1991.

[3] E. Horowitz and S. Sahni, *Fundamentals of Data Structures in Pascal*, Computer Science Press, 1984.

[4] C. L. Liu, *Introduction to Combinatorial Mathematics*, McGraw-Hill, New York, 1968.

[5] U. Manber, *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, 1989.

[6] Z. Manna and J. Waldinger, *The Logical Basis for Computer Programming, Volume I: Deductive Reasoning*, Reading, MA: Addison-Wesley, 1985.

[7] H. A. Partsch, *Specification and Transformation of Programs: A Formal Approach to Software Development*, Springer-Verlag, 1990.

[8] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. on Software Engineering*, Vol. 1, No. 2, pp. 22-232, June 1975.

[9] N. R. Saxena and E. J. McCluskey, "Linear Complexity Assertions for Sorting," *IEEE Trans. on Software Engineering*, Vol. 20, No. 6, pp. 424-431, June 1994.

[10] I. Sommerville, *Software Engineering*, 4th Ed., Addison-Wesley, 1992.