1982

Exploiting partial order with quicksort

R. Geoff Dromey
University of Wollongong

EXPLOITING PARTIAL ORDER WITH QUICKSORT

R. Geoff Dromey

Department of Computing Science,
The University of Wollongong,
Post Office Box 1144,
Wollongong, N.S.W. 2500
Australia.

ABSTRACT

The widely known Quicksort algorithm does not attempt to actively take advantage of partial order in sorting data. A relatively simple change can be made to the Quicksort strategy to give a bestcase performance of n, for ordered data, with a smooth transition to O(n log n) for the random data case. An attractive attribute of this new algorithm (Transort) is that its performance for random data matches that for Sedgewick's claimed best implementation of Quicksort.

EXPLOITING PARTIAL ORDER WITH QUICKSORT

R. Geoff Dromey

Department of Computing Science,
The University of Wollongong,
Post Office Box 1144,
Wollongong, N.S.W. 2500
Australia.

1. Introduction

The Quicksort algorithm [1] is an elegant, efficient and widely used internal sorting method. The algorithm has been extensively studied by a number of workers [1 to 7] but there have been relatively few real improvements to the algorithm beyond those initially suggested by its inventor C.A.R. Hoare [1]. The work of Sedgewick [5,6] contains a very comprehensive and useful perspective on the implementation and analysis of Quicksort.

One weakness of the Quicksort algorithm is that it does not attempt to take advantage of partial order that may be present initially in a given data set or that may develop during the course of the sort. In the discussion which follows it will be shown how a relatively simple change can be made to the Quicksort strategy to allow it to take advantage of partial order with consequent gains in performance.

2. Design Principles

Any change to Quicksort that attempts to take advantage of partial order should be a refinement that does not adversely influence its performance for the more general random data case.

If we are to adhere to this principle then any mechanism we propose should probably degenerate naturally and smoothly into the standard Quicksort divide and conquer partitioning strategy in the case where random data must be processed. As it turns out there are a number of ways we can do this. We will present one of the simplest of these methods.

Recalling the basic Quicksort algorithm, it sorts a data set a[1...n] by first rearranging the elements of the data set such that

> a[1...k] $\leq$ x $\leq$ a[k+1...n] for some k in $1 \leq k \leq n$, for some elements x of the data set, and for a given ordering relation. After the initial partitioning step the same procedure is recursively applied to the two segments a[1...k] and a[k+1...n]. Recursive partitioning ceases when segments of size one are encountered.

An attempt can be made to take advantage of partial order in the data by preceding each partitioning stage by a call to a mechanism that makes a sort check on the current segment a[l...r]. Naively this could simply involve a run through the current segment until an element was encountered that was out of order or until it was established that the segment was sorted. Such a strategy does not however dovetail naturally with the partitioning mechanism of Quicksort. A slightly more sophisticated strategy must therefore be adopted.

3. Transort Strategy

A way to check on the order status of a given segment is to proceed first from the left end, and then from the right end of the segment, while elements are encountered that are respectively in non descending and non ascending order. Two possibilities can result from these prepartitioning sortchecks.

(a) The sort check extending from the left will establish that the segment a[l...r] is already sorted and requires no further processing.

(b) More usually two ordered segments a[l...i−1] and a[j+1...r] will be established at either end of the current segment being sorted, a[l...r].

When situation (ii) applies we can take advantage of the partial order established in four ways:

(i) in offsetting the cost of partitioning the current segment

(ii) in selecting the partitioning value for the current segment

(iii) in detecting situations where an insertion sort can be more effectively employed to complete the sort of the current segment

(iv) in reducing the re examination of partially ordered segments established in the current call to the mechanism in the subsequent recursive calls to the mechanism.

We are now in a position to examine in more detail how advantage can be taken of the information gained from the prepartitioning sort checks.

3.1. Preconditioning

The prepartitioning phase first establishes (when a[l...r] is not sorted) the largest *i and smallest j* such that the following relations hold:

A1: $\forall s,t((l \le s < t \le i-1 \le r) \supset (a[s] \le (a[t]))$

A2: $\forall s,t((l \le j+1 \le s < t \le r) \supset (a[s] \le a[t]))$.

In the case where it is established that $i > r$ is true it is implied that the complete segment a[l...r] is sorted in non descending order and consequently no further processing is required.

When $i < r$ it is implied that the complete segment a[l...r] is not sorted and so further processing is required. The relations (A1) and (A2) do not establish anything about the relativity between the two ordered sequences a[1...i−1] and a[j+1...r]. It is possible that some or all of the values in the ordered sequence a[j+1...r] are "less" in value (as defined by the chosen ordering relation) than some or all of the values in the ordered sequence a[1...i−1]. Consequently we cannot directly apply the partitioning mechanism employed in Quicksort.

What is required is the application of a preconditioning mechanism that establishes relativity between the two sequences and that subsequently allows application of the partitioning mechanism of Quicksort should that be necessary (i.e. if $i < j$). The pre-conditioning mechanism must establish the following relations:

A3: $\forall s,t((l \le s \le i-1 < j+1 \le t \le r) \supset (a[s] \le a[t]))$

A4: $\forall s((l \le s < p) \wedge (l \le p \le i-1)) \supset (a[s] \le a[s+1]))$

A5: $\forall t((q < t \le r) \wedge (j+1 \le q \le r)) \supset (a[t-1] \le a[t]))$

The essential part of the preconditioning mechanism implemented in Pascal can take the form:

```
{assert: A1 ^ A2}

p := i; q := j;
lc := l; rc := r;
repeat
 p := p-1; q := q+1;
 if a[p] > a[q] then
    exchange (a[p], a[q])
 else
   begin
    lc := p; rc := q
   end
 until (lc := p) or (rc := q)
   .
   .
   .
{assert: A3 ^ A4 ^ A5}
```

where the "exchange" mechanism simply exchanges the array elements a[p] and a[q].

When the preconditioning phase is completed it is then possible to proceed with the partitioning of the segment a[i...j] provided an appropriate choice of partitioning element x is made.

3.2. Selection of the Partitioning Element

In Quicksort implementations one of three methods is usually used to select the partitioning element:

(i) the first element of the segment a[l] is selected

(ii) the middle element of the segment a[(l+r) div 2] is selected

(iii) the median of a[l], a[r], and a[(l+r) div 2] is selected.

As demonstrated first by Singleton [4], and later by Sedgewick [5,6], the median of three method is to be preferred because it reduces the number of comparisons to complete the sort on average by approximately 9%.

A consequence of the preconditioning mechanism outlined above is that the median of three method of selecting the partitioning element fits naturally into the mechanism for the Transort algorithm. To appreciate this we need to take into account that the following two relations must also hold after the preconditioning phase:

A6: $\forall s,t((p+1 \leq s < t \leq i-1) \supset (a[s] \geq a[t]))$

A7: $\forall s,t((j+1 \leq s < t \leq q-1) \supset (a[s] \geq a[t]))$.

After the preconditioning phase we are left with the situation that the relativity between a[p] and a[p+1] and also between a[q] and a[q-1] has not been established. Applying the steps:

```
if a[p] ≤ a[p+1] then p := p+1;

if a[q-1] ≤ a[q] then q := q-1
```

enables us to establish the additional relations:

A8: $\forall s((l \leq s, p \leq i-1) \supset (a[p] \geq a[s]))$

A9: $\forall t((j+1 \leq t, q \leq r) \supset (a[q] \geq a[t]))$

which are needed to make a suitable choice for the partitioning element x while still taking advantage of the information gained during the sort checking and preconditioning steps.

At this point we have established that a[p] is greater than or equal to all the elements in a[l...i-1] and a[q] is less than or equal to all the elements in a[j+1...r]. Additionally from A1...A5 we can conclude that a[p] ≤ a[q]. The median of a[p], a[q] and a[m] (where m:= (p+q) div 2) can then be found for subsequent use as the partitioning element of the current segment a[l...r]. Because a[p] ≤ a[q] is already established the median of three selection is simpler than the methods employed by Sedgewick [5,6]. At most two additional comparisons are needed to find the median of a[p], a[m], and a[q]. We then have the option of moving the a[m] element out of the bounds of the segment a[i...j] that remains to be partitioned.

With the appropriate choice of partitioning element made we can then apply a conventional partitioning mechanism to the segment a[i...j].

3.3. Taking Advantage of Partial Order in the Current Segment

Sedgewick [5,6] has shown that Quicksort's performance can be improved by initially ignoring small segments (of size ≤ 9) generated during the course of the sort. When the recursive mechanism terminates an insertion sort can be applied to the complete data set resulting from the Quicksort. This strategy is to be preferred because of the efficiency of insertion sort over Quicksort for sorting small or localized data sets. The same idea could be applied to the present algorithm. However we can go one step further in this case. For some data sets where there is a high degree of partial order present we find that on occasions the i and j indices are close together after the sort check even though the segment currently being processed is much larger than would normally be left for insertion sorting according to Sedgewick's criteria. In these cases it is appropriate to terminate the recursive partitioning in favour of subsequent insertion sorting. Hence the proximity of i and j after the sort check rather than the size of the segment can provide an effective criterion for the subsequent application of an insertion sort i.e.

A3 $\wedge$A4 $\wedge$A5 $\wedge\, j - i \leq 9$ "subsequent sorting by insertion"

3.4. Taking Subsequent Advantage of Partial Order

Partitioning of the segment a[ℓ...r] establishes the following relation:

A10: $(j < i) \wedge \forall s,t((l \leq s < i) \wedge (j < t \leq r)\ (a[s] \leq a[t]))$.

At this point with Quicksort implementations (allowing for some minor variations) the partitioning mechanism is recursively applied to the two segments a[l...j] and a[i...r]. Instead of proceeding directly with this approach we can take advantage of the fact that the segments a[l...p] and a[q...r] are ordered with the following restrictions on p and q:

$$l \leq p < j$$

$$i < q \leq r.$$

Consequently in recursive calls to the mechanism the sort check can attempt to extend the ordered segment a[l...p] above p and the segment a[q...r] below q *without* re examining the subsegments that were previously established to be ordered. This can on average reduce the cost of processing segments when there is a degree of partial order present. To accomplish these savings information gained from the sort check of the current segment must be transmitted in the recursive calls that process the two resulting subsegments (i.e. via the values of p and q established in the current segment).

4. Implementation

The implementation of Transort follows closely in structure that of Quicksort. The differences in implementation mostly follow directly from the discussion in the previous section. We will present the overall structure of the implementation in Pascal in a modular format and clarify details that are not obvious or that do not follow directly from the discussion and assertions established in the preceding section. Details of the accompanying insertion sort are not given.

```
procedure transort(var a:nelements; l,r, lsort, rsort:integer);
var i, j, p, m, q, x:integer; sorted:boolean;
begin
 {assert: cutoff ≥ 4}

 if rsort - lsort ≥cutoff then begin

   {assert: a[l...lsort] ordered ^ a[rsort...r]ordered}

   sortcheck(a,lsort,rsort,i,j,sorted);

   {assert: A1 ^ A2}

   if (not sorted) and (j-i) ≥ cutoff)then begin

      precondition(a,l,r,i,j,p,q);

      {assert: (A3 ^ A4 ^ A5) ^ (A6 ^A7) ^(A8 ^A9)}

      m := (p+q)div 2;

      medianofthree(a,p,m,q,l,r,i,x);

      {assert: x is median of a[p], a[m] and a[q]}

      partition(a,i,j,x);

      {assert: A10}

      transort(a,l,i,p,j);

      transort(a,i,r,i,q)

      {assert: a[l...j]sorted ^ a[i...r]sorted a[1...r]sorted}
      end
   end
end
```

The implementation of the sortcheck procedure is straightforward in that its role is simply to extend the left ordered segment above lsort to i-1 and to extend the right-ordered segment below rsort to j+1. The indices i and j will then mark the first out of order elements in both instances.

The precondition procedure implementation follows directly from the discussion in sections 3.1 and 3.2.

Perhaps rather surprisingly considerable care should be taken with the implementation of the median of three method for selecting the partitioning element x. The simplest way to implement the median of three method is to rearrange the elements p,m, and q such that

$$a[p] \leq a[m] \leq a[q].$$

The value stored at position m can then be selected as the partitioning element x. The advantage of this strategy is that reverse ordered data is then sorted using essentially only 2n comparisons. However further gains in the general case can be obtained by moving the partitioning element "out" of the segment to be partitioned before carrying out the partitioning step. This can be accomplished by moving the a[i] element into a[m] and starting the partitioning at position i+1. This however disturbs the reverse ordered case. To bring symmetry back into the problem and therefore allow reverse ordered data still to be handled in essentially 2n comparisons, partitioning at the right hand end of the data must be started at j-1 with subsequent compensation for this change after the central partitioning step.

Incorporating these ideas the steps in the median of three mechanism are:

```
begin
  if a[m] > a[q] then begin
    x := a[q];
    a[q] := a[m];
    q := q+1 - q div r
  end
  else begin
    if a[p] > a[m] then begin
      x := a[p];
      a[p] := a[m];
      p := p-1 + i div p
    end
    else x := a[m]
  end;
 a[m] := a[i]
end
```

One final comment about the median of three selection method is that the approach suggested by Sedgewick [6] should not be used particularly for data that has a high degree of partial order as it can be shown to frequently lead to degenerate partitioning sequences with resultant poor performance. This effect is not noticeable for his implementation with random data.

For the implementation of the partitioning mechanism the procedure suggested by Sedgewick could be directly applied [6]. We will instead suggest an alternative implementation which has a cleaner structure. A problem encountered with most of the partitioning algorithms (including Sedgewick's) that have been suggested for use with Quicksort [5,6] is that they must either compensate for, or protect against an exchange being made *after* the indices i and j delimiting the partitions have crossed over. By moving the very first partition extension steps outside the driving loop the problem disappears since the driving loop guard will protect against such exchanges. Adopting this idea the central steps in the partitioning mechanism become:

```
begin
  isave := i; jsave := j;
  repeat i := i + 1 until a[i] ≥x;
  repeat j := j - 1 until a[j] ≤x;
  {invariant: (isave≤ i) ∧ ∀ s(isave ≤ s < i)  (a[s] ≤ x))
  ∧ (j ≤jsave) ∧ ∀ t((j < t≤ jsave)  (a[t] ≥ x)).
  while i < j do
   begin
     exchange (a[i], a[j]);
     repeat i := i + 1 until a[i] ≥ x;
     repeat j := j - 1 until a[j] ≤ x
   end
{assert: (j < i)  ∧∀ s,t((l ≤ s < j)  ∧ (i < t ≤ r)  (a[s] ≤a[t])).
```

This mechanism can be made slightly more efficient by using the loop condition " i < j-1" rather than "i < j" and including an additional test that will allow swapping of a[i] and a[j] if necessary when the loop terminates with i=j-1.

In most implementations of partitioning mechanisms [5,6] the partitioning element x is put in its "sorted" place after the partitioning step is completed. In applying this idea to the present implementation we want to ensure that it is done in such a way that reverse ordered data still gets sorted with essentially only 2n comparisons. To do this we must first save the i and j positions *before* partitioning commences (the variables isave and jsave are used for this purpose). When the partitioning mechanism described above has terminated there will still be one element a[jsave] that has not been taken into account. If it belongs in the segment a[i..r] then all that is necessary is to move the element a[j] to position isave and place the partitioning element x, into position j. In the other instance where a[jsave] belongs in the partition a[l..j] the changes are slightly more involved to allow for the reverse ordered data case. The mechanism that will smoothly handle both these situations is:

```
         if a[jsave] > x then begin
              a[isave] := a[j];
              a[j] := x;
              j := j-1
           end
         else begin
              a[isave] := a[jsave];
              a[jsave] :=  a[i];
              a[i] := x;
              i := i+1
           end
      {assert: a10}
```

These additions to the partitioning mechanism allow us to derive all the benefits of Sedgewick's partitioning mechanism by excluding the partitioning element from the present and subsequent partitioning stages. Furthermore the special case of reverse ordered data is handled in the least costly way because of the retainment of symmetry by the mechanism. All the design information needed to implement the algorithm is now complete.

5. Evaluation and Performance

The Transort algorithm is more complex than Quicksort in that it requires two additional mechanisms; one for sort checking and a second for pre-conditioning. These additional mechanisms are however simple and so do not significantly add to the complexity of the algorithm or to its running cost even in the general random data case.

The attractive attributes of the algorithm rest on the fact that an ordered segment of length j is always "sorted" in j-1 comparisons rather than of order jlogj comparisons. This means that sorted data is always handled in the most natural and efficient way. Reverse ordered data is sorted with essentially 2n comparisons (ie n comparisons to reverse the order of the data and another 2(n/2) = n comparisons to detect that the two partitions are sorted). A data set of n equal values is "sorted" with n-1 comparisons.

Data with a high degree of partial order present takes somewhere between n and O(logn) comparisons to complete the sort. To characterize Transort's behaviour for partially ordered data it is necessary to have a measure for partial order. Two fairly obvious and intuitive definitions that can be used are:

(i) Percentage Out of Order

The percentage out of order is the percentage of elements that need to be removed from a list to leave the remaining elements in order [8].

(ii) Percentage Out of Position

The percentage out of position is the percentage of elements in the set that are not occupying the respective ordinal positions that would place them in order.

For a given percentage of disorder we would expect that less effort on average is needed to complete the sort for the out of position measure. With the out of order definition for say 100 elements it is possible for only one element to be out of order with 100% of the elements out of position. Another measure that could be considered is the sum of the distances of each element from its final position normalized by the element count. Unfortunately the latter measure does not appear to lend itself to a simple randomized generative procedure for producing data sets with predefined characteristics.

Ideally to characterize the behaviour of Transort we should attempt to derive an analytic expression that captures the average behaviour of the algorithm as a function of the percentage out of order (or out of position). To do this it is necessary to average over all possible configurations for a given percentage out of order. Unfortunately such a scheme will not lead to a recurrence relation that is easily solvable by the techniques applied by Sedgewick in his analysis of Quicksort [5]. Empirical simulation results are therefore presented in Table 1 to give some flavour of the performance of Transort as a function of the degree of disorder in the data. To give relativity to the presentation, results for Sedgewick's "best" implementation of Quicksort [ref. 5 program 8.2] are also given. Results are presented for both out of order and out of position measures of disorder.

A count of the comparisons and the sum of comparisons and data element assignments were used to characterize the performance. A similar measure to the latter has been claimed by Cook and Kim [8] to more accurately reflect the amount of effort needed to complete the sorts. The results are presented for 1000 data elements. Each result represents a sum over thirty randomized runs. In obtaining the results a cut off of 9 has been used for the insertion sort.

There have recently been several other sorting methods proposed for handling partially ordered data. The most elegant of these is Dijkstra's Smoothsort [9]. It has the advantage over Transort in that its worst case behaviour is O(nlogn) and its best case behaviour is O(n). Unfortunately Transort still has worst case behaviour of $O(n^2)$ although the data set yielding the worst case is considerably more obscure than the worst cast data set for Quicksort. Dijkstra has not characterized the behaviour of his algorithm as a function of the degree of disorder in the data either analytically or empirically. A similar relativity between Smoothsort and Transort as between Heapsort and Quicksort may be expected. On the basis that random data was the norm Transort could be expected to give the better average performance.

The other two methods proposed, a natural mergesort suggested by Knuth [7] and tested by Harris [10], and an algorithm due to Cook and Kim [8] both exhibit behaviour for small degrees of disorder that may be slightly better than that for Transort. Unfortunately their behaviour for the random data case is considerably worse (particularly Cook and Kim's method) than either Quicksort or Transort. This behaviour diminishes their practical value. Furthermore, unlike Transort and Smoothsort, they are not in place sorting methods.

6. Conclusions

Transort is a competitive alternative as an internal sorting method. It is able to take advantage of partial order in a data set and yield performance measures comparable to other alternatives. Special cases, such as sorted data, reverse ordered data, and equal-keyed data are handled naturally and efficiently. At the same time its performance for random data matches the performance of the claimed best and most practical implementation of Quicksort.

7. Acknowledgements

I wish to thank Philip Maker and Michael Shepanski for helpful discussions during the course of this work.

References

1. C.A.R. Hoare, "Quicksort", Computer J. 5, (1962), 10-15.
2. M.N. Van Emden, Increasing the Efficiency of Quicksort, Comm. ACM, 13, (1970), 563-567.
3. R.S. Scowan, Quickersort, (Algorithm 271), Comm. ACM, 8, (1965) 669-670.
4. R.C. Singleton, An Efficient Algorithm for Sorting with Minimal Storage (Algorithm 347) Comm. ACM 12, (196 185-187.
5. R. Sedgewick, Quicksort, (Ph.D Thesis), Stanford University, (1975), Report Number STAN-CS-75-492.
6. R. Sedgewick, Implementation of Quicksort, Comm. ACM, 21, (1978) 847-857.
7. D.E. Knuth, The Art of Computer Programming, Vol 3, Sorting and Searching, Addison-Wesley, Mass. (1972).
8. C.R. Cook, and D.J. Kim, Best Sorting Algorithm for Nearly Sorted Lists, Comm. ACM, 23, (1980), 20-624.
9. E.W. Dijkstra, Smoothsort, an Alternative for Sorting in situ. Science of Comp. Programming 1, (1982), 223-233.
10. J.D. Harris, Sorting Unsorted and Partially Ordered Lists Using the Natural Mergesort, Software Practice and Experience, 11, (1981), 1339-1340.

TABLE I

Empirical Performance for Partially Ordered Data

% Out of Order	0.2%	0.5%	1%	2%	5%	10%	30%	50%	Random
Quicksort	10027	10515	10088	10671	10600	10508	10461	10769	10761
Transort	3090	4903	5624	7412	9912	10292	10849	10860	10751
(% Quicksort)	31%	46%	56%	69%	93%	97%	103%	100%	100%
COMPARISONS									
Quicksort	12380	13514	12938	14507	15375	15515	17131	17929	19089
Transort	4055	6636	7225	10190	14450	15179	17284	17877	18479
(% Quicksort)	32%	49%	55%	70%	93%	97%	100%	99%	96%
COMPARISONS & ASSIGNMENTS									

% Out of Order (Comparisons)	0.2%	0.5%	1%	2%	5%	10%	30%	50%	Random
Quicksort	9465	9861	9571	9753	9341	9570	10166	10304	10761
Transort	2742	4136	4731	5829	6471	7754	9130	9729	10751
(% Quicksort)	28%	41%	49%	59%	69%	81%	89%	94%	100%
% Out-of-Position COMPARISONS & ASSIGNMENTS	0.2%	0.5%	1%	2%	5%	10%	30%	50%	Random
Quicksort	11337	12038	11525	11916	11394	11982	14171	15537	19089
Transort	3389	4947	5563	6786	7634	9429	12736	14777	18479
(% Quicksort)	29%	41%	48%	56%	67%	78%	89%	95%	96%