UW Aberystwyth, Department of Computer Science

CS21020

Program Design, Data Structures and Algorithms

Assignment Two Part 1 - Assessment and Solution

This is Part 1 of the second of two CS21020 practical assignments. It will count for 15% of the total marks for the course. You should spend of the order of fifteen hours of effort on this assignment.

This assignment consists of five tasks. The percentage that any task contributes to the whole assignment is shown in brackets to the right of the description of the task.

Marking criteria - ringed items apply to you.

1. Were example outputs from intermediate sort states (a) machine produced (b) explained (c) sufficient to illustrate the method?
2. Did the complexity analysis show understanding of the issues involved, or did it just appear to be copied from other sources?
3. The metrics for measuring program activity in sorting are, most naturally, counts for the number of swaps, moves and compare operations involving array elements. Were (a) separate complexity arguments given for each? (b) were the counts obtained ? (c) was it explained how the count were obtained, and were they obtained reliably, with minimal program alteration? (see Appendix E.)
4. Were (a)best (b)average (random) (c) worst sort cases discussed, in terms of complexity theory, and run results?
5. Were appropriate growth functions matched against the counter values found? Was it appreciated that the different counters can have different growth rates?
6. Was there an attempt to verify that the growth functions and matched counter values corresponded to theoretical expectations?
7. Were adequate data samples used? Was it made clear how the data were generated?
8. Were significant program alterations of the given source (if any) made clear?
9. Was the core of the assignment kept to within the suggested page limit (12 pages)?
10. Did the solution demonstrate any insights obtained (any conclusions) from the sorting experiments?

** H. Holstein, 19th May, 1997*

1 Assessment

2 Examiner's comments

(Examiner's comments continued)

3 Part I - Statement of Task

3.1 Assignment Overview

This assignment is concerned with the practical evaluation of the complexity of **two** out of five sorting algorithms provided, **one of which should be the Quicksort algorithm**. Ada code is provided for four of the algorithms for experimentation. A secondary concern is Ada reuse - the sort package is written in a way that allows arrays of different data types to be sorted, according to a user-defined "less than" operator.

Testing the efficiency of a sort routine requires more than runs on a few arbitrary data samples. Consult the lecture notes and read the standard course texts, and others, to find out the expected theoretical performance behaviour of the two algorithms, what are suitable data sets for testing the various algorithms, and in particular, on what data sets they may perform well or badly.

On the basis of reference sources (which you should quote), choose suitable metrics for monitoring the amount of computation done in an algorithm. Implement appropriate counters, added to the code provided. Verify (or disagree with !) the theoretical estimates.

Limit answers to Part I to *at most* 12 pages (normal spacing).

3.2 Detailed task description

1. Show example outputs from intermediate states of the two sort routines to illustrate the way these routines work. [20]

2. By consulting the literature on sorting, summarise the main results for the time complexity of the two algorithms. [20]

3. Run examples of the two sort routines provided, to obtain an experimental verification (or otherwise!) of the theoretically derived complexities. Explain the basis for choosing the data, and how it was generated. [40]

   Note. If a counter has a value C_N when sorting N data items, and the counter is supposed to grow as $O(g(N))$, then the ratio

   $$C_N/g(N)$$

   should "settle down" to a fairly constant value for large enough N. This ratio should be computed for various data samples in order to verify a complexity formula experimentally.

4. Experiment with *one* of [20]

   (a) the Shell Sort with different sequences of diminishing increments;

   (b) the Quick Sort with a different pivotal strategy;

   (c) the Quick Sort, with different cutoffs and switches to different algorithms below the cutoff;

   (d) sorting of alphabetic and float type items, and of records containing a key field.

4 Solution concerning Quick Sort

1. **Show example outputs from intermediate states of the two sort routines to illustrate the way these routines work.**

   Quicksort is a divide and conquer algorithm. It partitions a given array with respect to a 'pivot', such that values to the left are less than or equal to the pivot, and values to the right are greater or equal to the pivot.

   This places the pivotal element in its correct position, from which it never has to move again. The problem of sorting becomes that of recursively sorting the left and right partitions.

   Ideally, we would like a pivotal strategy that neatly partitions the array into two equal halves at every stage. In practice a fast 'hit and miss' method is used, which is likely to be reasonable on average. The particular pivotal strategy used here, the 'median of 3', chooses from the key values of the middle and two end ends of the array segment. Partition details will be given in the next section.

   As an example, 16 array elements were sorted by quicksort (with the 'Cutoff' parameter reset from 10 to 2, to allow us to examine short array sequences). The array partition indexes were captured from print statements placed at the entry and exit of the recursive quicksort routine, as shown in Appendix A.

   The entry traces, clearly showing the diminishing ranges, were:

```
Q_Sort called on index range   1  16
Q_Sort called on index range   1   8
Q_Sort called on index range   1   1
Q_Sort called on index range   3   8
Q_Sort called on index range   3   4
Q_Sort called on index range   6   8
Q_Sort called on index range   6   6
Q_Sort called on index range   8   8
Q_Sort called on index range  10  16
Q_Sort called on index range  10  13
Q_Sort called on index range  10  10
Q_Sort called on index range  12  13
Q_Sort called on index range  15  16
```

   The exit traces (subarrays of length 2 or 1 are not traced), clearly showing longer and longer sorted ranges, were:

```
Sorting now complete in index range   6   8
Sorting now complete in index range   3   8
Sorting now complete in index range   1   8
Sorting now complete in index range  10  13
Sorting now complete in index range  10  16
Sorting now complete in index range   1  16
```

These sequences can be understood in terms of a binary 'call' tree, in which each node (representing the call to quicksort over a given index range), has two children, corresponding to the two recursive calls made.

```
               1     -       16

            /                   \
         1 - 8      (9)        10-16
        /     \               /       \
       1  (2) 3-8          10-13 (14) 15-16
             /   \        /     \
         3-4 (5) 6-8  10 (11) 12-13
                /   \
              6 (7) 8
```

Entry traces are obtained from a pre-order traversal of the tree, while exit traces follow a post-order traversal.

Appendix B shows a program run illustrating (with minimal subsequent editing) actual data movement for the descending sequence -1 to -16, to be sorted into the ascending order -16 to -1. The values considered by the 'median of 3' are highlighted by an asterisk ($*$). The pivotal value is chosen from among these three. The result after partitioning with respect to the pivot (placed in brackets) is shown. It is to be noted that all values to the left of the pivot are less than or equal to it, and all values to the right are greater or equal to it, as required. The entry and exit traces shown above now appear interleaved in the appropriate order.

The partition strategy is discussed in the next section.

2. **By consulting the literature on sorting, summarise the main results for the time complexity of the two algorithms.**

The behaviour of quicksort depends critically on its partition strategy. This is briefly summarised.

The 'median of 3' strategy sorts, in situ, the middle and two end elements. Thus from

```
 -1  -2  -3  -4  -5  -6  -7  -8  -9 -10 -11 -12 -13 -14 -15 -16
```

we obtain

```
-16  -2  -3  -4  -5  -6  -7 (-8) -9 -10 -11 -12 -13 -14 -15  -1
```

with pivot = -8. Partitioning with respect to the pivot begins with a preliminary step (for programming convenience) of swapping the pivot with the right end but one element:

```
-16  -2  -3  -4  -5  -6  -7 -15  -9 -10 -11 -12 -13 -14 (-8) -1
                             ^                             ^
```

The first and the last two elements are now in their right partition. It follows that partitioning can proceed by considering just elements

```
     -2  -3  -4  -5  -6  -7 -15  -9 -10 -11 -12 -13 -14.
      ^                                              ^

      left                                           right
```

Left and right pointers now advance towards the middle, stopping when an element in the wrong partition is found, or on being equal to the pivot. Such elements are swapped over. In the present example, pairs (-14,-2), (-13,-3), (-12,-4), (-11,-5), (-10,-6) (-9,-7) are swapped in this process, leading to

```
-14 -13 -12 -11 -10  -9 -15  -7  -6  -5  -4  -3  -2.
                      ^       ^

                      l       r
```

Further movement of the pointers causes them to cross over:

```
-14 -13 -12 -11 -10  -9 -15  -7  -6  -5  -4  -3  -2.
                         ^    ^

                         r    l
```

The crossing over is the signal for the pointer movement to stop. The left pointer now points to an element (-7) not less than the pivot, so it belongs to the right partition, *and can therefore be swapped with* the pivotal element. On restoring the pivotal element, we obtain

```
-14 -13 -12 -11 -10  -9 -15 (-8) -6  -5  -4  -3  -2  -7 -1
```

The required partition (which agrees with Appendix B) is now complete.

In general, partitioning an array segment of l elements will require every element to undergo a comparison. Thus there are $O(l)$ comparisons. An element may or may not be swapped. If, on average, half the comparisons require swaps, then we still have $O(l)$ swaps. At best, only $O(1)$ swaps are required (in the median of 3). The number of data moves apart from swaps is constant per partition, and so is $O(1)$.

With reference to the recursion tree, the height of the tree corresponds to the number of levels of partition. For N items, the optimal and average tree height is $O(\log N)$.

In the worst case, an array segment of l elements is partitioned systematically, not near the middle, but at its extreme left or right. In that case, the successive recursion levels yield partitions of length $N-2, N-4, N-6, \ldots, 2$, giving $N/2$, or $O(N)$, levels.

The number of comparisons, *on any level*, involves all elements, (except previously found pivots), and so requires $O(N)$ work. Similarly, the average and worst swap counts are $O(N)$.

The number of moves is $O(1)$ per partition. Since there are $1+2+4+8+\ldots+N/2 = O(N)$ partitions on all levels combined in the case of optimal balancing, or $1+1+\ldots = O(N)$ for complete unbalancing, in either case there are $O(N)$ moves in the entire algorithm.

The best case for swaps is $O(1)$ per partition. Therefore $O(N)$ is also the best count for the swaps in the entire algorithm.

The case of equal (i.e. repeated) data is similar to the best case, except that values equal to the pivot (all are) have to be swapped with *every* pointer advancement. Although swapping values with equals is wasteful, this strategy does ensure that the left and right pointers advance at the same towards the centre, ensuring optimal partitioning. Thus, there are $O(N)$ swaps per level.

Taking the product of the number of levels, and the amount of work per level, we arrive at the work complexity summarised in the table:

Table 1. Theoretical quicksort complexities

	moves	swaps	comparisons
best	$O(N)$	$O(N)$	$O(N \log N)$
equal	$O(N)$	$O(N \log N)$	$O(N \log N)$
average	$O(N)$	$O(N \log N)$	$O(N \log N)$
worst	$O(N)$	$O(N)$	$O(N^2)$

Paradoxically, the case of systematic worst partitioning leads to only $O(1)$ swaps per partition (or $O(N)$ in total), since the pivot will be next to the first or last element, leaving no room for swaps either side of the pivot.

3. **Run examples of the two sort routines provided, to obtain an experimental verification (or otherwise!) of the theoretically derived complexities. Explain the basis for choosing the data, and how it was generated.**

Trial runs were performed with quicksort to emulate its best, average and worst performance.

For the **best performance**, I have taken sorted data of distinct values. In that case optimal partitioning takes place, and the action of 'hiding and restoring' the pivot retains the original sequence, with no systematic swapping having to be performed.

As second best, I have taken the case of **equal data**. This is similar to the sorted case, but requires maximal swapping.

For **average performance**, I have used pseudo random integers generated in the manner of quadratic probing in hashing, from the formula

$$(i - c)^2 \bmod d, i = 1 \ldots N \tag{1}$$

where d is a prime number just bigger than N, and c is an offset (taken as $\lfloor N/4 \rfloor$) to avoid an initially ascending sequence. The values of c and d, chosen according to the value of N, and the first three sequences, are shown in Appendix C.

For **worst performance**, I have constructed a sequence which leads to a succession of unbalanced partitions. To do this, I started with the sequence 1 2 3 and tried to reverse the partition algorithm so as to generate a sequence of unbalanced partitions of lengths 5,7,9, ...

If 1 2 3 was the left partition of 1 2 3 (4) 5, with 4 as the pivot, how would the 5 elements have to be ordered before partitioning?

The median of 3 strategy would need to find 1 4 5, in some order, in positions 1 3 5. The simplest case is 1 2 (4) 3 5. The action of "hiding the pivot" leads to 1 2 3 (4) 5. Applying left and right pointers to the sequence 2 3 leaves the right pointer at 3 and the left pointer crossing over to (4). The pivot is then swapped with itself, yielding partitioned sequences 1 2 3 and 5, as required.

In general, suppose we have a sequence of key values in the range $1 \ldots (2n - 1)$ which leads to a succession of unbalanced partitions. Append key values $2n$ and $2n + 1$, and swap position $(n + 1)$ - then middle one - with position $2n$. Then we obtain a new sequence of $2n+1$ values which, on partitioning, will reproduce the previous unbalanced sequence of $2n - 1$ values.

Starting with the sequence 1 2 3, we can build up longer and longer unbalanced sequences, adding two elements at a time. During the running of quicksort, such an unbalanced sequence of length N will be partitioned into an unbalanced sequence of length $N-2$, the pivot, and a partition of length 1. Successive production of unbalanced sequences of lengths $N-2, N-4, N-6, \ldots 3$ will result in an overall $O(N^2)$ performance.

The unbalanced sequences can be automatically generated. The Ada code and the first few sequences are given in Appendix C. Poor quicksort!

For **complexity metrics**, I counted the number of comparisons, moves and swaps, by placing counters in the compare, move and swap routines in which these operations are carried out. The counts were initialised to zero at the beginning of a sort run. The code is shown in Appendix D. Apart from changing the "<" operator to "/", no change had to be made any of the sort routines, and no count increments will be missed, in any of the sort routines.

In counting key comparisons, swaps and moves, I have followed Aho *et al.*, Weiss, Kruse, Knuth (see Appendix G). Kruse suggests counting item assignments - in that case, a swap would count as 3 assignments, and a move as 1 assignment.

Not all parts of a sorting algorithm involve key comparisons. It is therefore expedient to include other metrics, such as, for example, data movement (swaps and moves), for which we can verify the theoretical complexities. In the end, however, it is the metric with the biggest count growth in terms of the data size N that dominates the run time of the algorithm.

The tables below give the **results of the computational experiments**. Raw counts are given, as well as the counts normalised with respect to the appropriate theoretical complexity growth function from Table 1. If the function is chosen correctly, the normalised counts should remain fairly constant as the data size N varies over a large range. A check was made that allegedly sorted data was indeed sorted. The code is in Appendix E.

Table 2. Best case (sorted data).

N	moves	swaps	compares	moves/N	swaps/N	compares/$N \log_2 N$
15	29	2	31	1.93	.133	.530
31	63	6	97	2.03	.194	.632
63	131	14	261	2.08	.222	.696
127	267	30	653	2.10	.236	.735
255	539	62	1565	2.11	.243	.768
511	1083	126	3645	2.12	.247	.791
1023	2171	254	8317	2.12	.248	.811
2047	4347	510	18685	2.12	.249	.829
4095	8699	1022	41469	2.12	.250	.840
8191	17403	2046	91133	2.12	.250	.854
16383	34811	4094	198653	2.12	.250	.864

Table 3. Equal data case (repeated data).

N	moves	swaps	compares	moves/N	swaps/$N \log_2 N$	compares/$N \log_2 N$
15	29	8	31	1.93	.136	.530
31	63	32	97	2.03	.208	.632
63	131	96	261	2.08	.255	.696
127	267	256	653	2.10	.289	.735
255	539	640	1565	2.11	.314	.768
511	1083	1536	3645	2.12	.334	.791
1023	2171	3584	8317	2.12	.349	.811
2047	4347	8192	18685	2.12	.363	.829
4095	8699	18432	41469	2.12	.374	.840
8191	17403	40960	91133	2.12	.385	.854
16383	34811	90112	198653	2.12	.393	.864

Table 4. Reverse data case (descending data).

N	moves	swaps	compares	moves/N	swaps/$N \log_2 N$	compares/$N \log_2 N$
15	39	11	41	2.60	.188	.699
31	85	28	133	2.74	.182	.867
63	174	70	373	2.76	.186	.995
127	351	160	969	2.76	.180	1.09
255	704	346	2401	2.76	.170	1.18
511	1409	724	5757	2.76	.158	1.25
1023	2818	1486	13469	2.75	.145	1.32
2047	5635	3016	30913	2.75	.133	1.37
4095	11268	6082	69865	2.75	.124	1.42
8191	22533	12220	155925	2.75	.115	1.46
16383	45062	24502	344389	2.75	.107	1.50

Table 5. Random data case.

N	moves	swaps	compares	moves/N	swaps/$N \log_2 N$	compares/$N \log_2 N$
15	45	4	45	3.00	.068	.768
31	107	19	138	3.45	.124	.899
63	208	62	371	3.30	.165	.990
127	389	171	817	3.06	.193	.919
255	799	390	2015	3.13	.191	.988
511	1642	902	5151	3.21	.196	1.12
1023	3389	2049	10861	3.31	.200	1.06
2047	6756	4594	23821	3.30	.203	1.05
4095	13211	10275	52711	3.23	.209	1.00
8191	26981	22322	114479	3.29	.210	1.08
16383	53327	48948	250685	3.26	.213	1.09

Table 6. Worst data case .

N	moves	swaps	compares	moves/N	swaps/N	compares/N^2
15	38	6	66	2.53	.400	.293
31	78	22	290	2.52	.710	.302
63	158	54	1122	2.51	.857	.283
127	318	118	4322	2.50	.929	.268
255	638	246	16866	2.50	.965	.259
511	1278	502	66530	2.50	.982	.255
1023	2558	1014	264162	2.50	.991	.252
2047	5118	2038	1052642	2.50	.996	.251
4095	10238	4086	4202466	2.50	.998	.251
8191	20478	8182	16793570	2.50	.999	.250
16383	40958	16374	67141602	2.50	.999	.250

Discussion

In all cases, the theoretical complexities from Table 1 are convincingly verified, because the normalised counts for moves, swaps and compares remain essentially constant as N approaches large enough values. Comparisons systematically have the highest counts as a function of N. Comparisons therefore dominate the quicksort algorithm.

Results for sorted data. This corresponds to the "best" case in Table 1, since very little data movement has to take place. The median of 3 will always find the optimal pivot, and partitioning does not involve swapping.

Results for equal data. This case is similar to the one above, except that quicksort wastes a lot of effort in swapping equal values. This results in the left and right pointers moving in one step at a time, ensuring optimal partition in the middle. Wasting swaps saves the algorithm - it remains $O(N \log N)$.

Results for reverse and random data. I take the random data example to be indicative of the average performance of quicksort. The compares count for reverse data is 50% higher than for random data - reverse data appears to be a slightly 'hard' problem for quicksort.

Results for worst data. The construction of this data convincingly demonstrates that quicksort can degrade to an $O(N^2)$ complexity in the compares count. The algorithm was noticeably slow in this case, taking several minutes to complete the final sort case. The move and swap counts are $O(N)$ as predicted.

Conclusion. Quicksort experiments verified that the normal performance is that of an $O(N \log N)$ algorithm, but that data sequences can be constructed for which the performance degrades to $O(N^2)$.

4. **Experiment with *one* of**

   (a) ...

   (b) **the Quick Sort, with different cutoffs and switches to different algorithms below the cutoff;**

I will give some results on using different cutoffs. I used only one work metric:

$$W = (\text{Moves} + 3 * \text{Swaps} + \text{Compares})/(N \log_2 N).$$

For a given value of N, the values of W are obtained for each of the ten cutoffs

$$2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.$$

Normalisation by the factor $N \log_2 N$ allows results for different values of N to be compared more readily.

The values of W were obtained for the random data samples, using the insertion sort and Shell sort after cutoff. The results are summarised in Tables 7 and 8.

Table 7. Quicksort with insertion sort at different cutoff values. (Random data.)

Cutoff	2	4	8	16	32	64	128	256	512	1024
$N = 15$	3.02	2.37	$\underline{1.74}$							
31	2.50	2.12	$\underline{2.01}$	2.10						
63	2.45	2.23	$\underline{2.07}$	2.17	3.24					
127	2.43	2.13	$\underline{1.96}$	2.06	2.76	3.75				
255	2.35	2.14	$\underline{1.98}$	2.04	2.43	3.24	6.33			
511	2.41	2.20	$\underline{2.08}$	2.14	2.56	3.12	4.43	12.7		
1023	2.29	2.12	$\underline{2.00}$	2.04	2.35	3.13	5.93	10.1	17.0	
2047	2.24	2.08	$\underline{1.98}$	2.01	2.32	3.07	4.90	9.62	16.9	35.3
4095	2.21	2.07	$\underline{1.97}$	2.01	2.31	2.98	4.66	8.49	14.9	31.5
8191	2.18	2.05	$\underline{1.97}$	1.99	2.25	2.94	4.40	7.77	14.5	29.5
16383	2.18	2.05	$\underline{1.97}$	2.00	2.23	2.87	4.29	7.43	14.5	25.7

Table 8. Quicksort with Shell sort at different cutoff values. (Random data.)

Cutoff	2	4	8	16	32	64	128	256	512	1024
$N = 15$	4.04	3.40	$\underline{2.66}$							
31	3.82	3.43	3.17	$\underline{3.03}$						
63	4.02	3.76	3.49	3.28	$\underline{3.01}$					
127	4.18	3.86	3.59	3.45	3.30	$\underline{3.16}$				
255	4.24	4.01	3.79	3.64	3.51	3.44	$\underline{3.38}$			
511	4.41	4.20	4.01	3.87	3.74	3.72	3.67	$\underline{3.49}$		
1023	4.39	4.20	4.04	3.90	3.80	3.76	3.71	3.70	$\underline{3.69}$	
2047	4.42	4.25	4.10	3.98	3.90	3.82	3.79	$\underline{3.78}$	3.80	3.84
4095	4.46	4.31	4.17	4.06	3.98	3.91	3.89	3.89	$\underline{3.88}$	3.94
8191	4.49	4.35	4.23	4.12	4.04	3.98	3.95	$\underline{3.95}$	3.96	4.04
16383	4.53	4.41	4.28	4.19	4.11	4.07	4.04	$\underline{4.03}$	4.04	4.08

Discussion. In Table 7, the minimum work appears consistently around a cutoff of 8 (figures underlined). This confirms that the cutoff value of 10, recommended by Weiss, seems to have a sound basis. The high work counts at larger cutoffs indicates that the insertion sort is much less efficient than the quicksort. Giving too much work to the insertion sort raises the computational cost of the composite algorithm.

I repeated the experiment with a Shell sort - not expecting any improvement over the above, except that the work counts at the higher cutoffs should not climb so steeply, because Shell sort is more efficient at longer sequences than a pure insertion sort. Results are given in Table 8.

The immediate observation is that at low cutoff values, the Shell sort W values are much higher than for insertion sort (by about a factor 2). Therefore, quicksort + Shell short is wasteful compared to quicksort + insertion sort. The underlined minimum work counts occur at higher cutoff values than for insertion sort, consistent with the arguments presented in the last but one paragraph. Unlike the insertion sort case, the minima occur at cutoffs that depend on N.

Conclusion. Quicksort performs marginally better in terms of operation counts, when its operation is cut off at sequences of length 8 or less. The cutoff value seems fairly insensitive to N. It is therefore cheap to implement.

Another metric which could have been used is a count of the number of recursive quicksort calls, by placing a counter at the procedure entry. Although not implemented, it would have shown that large numbers of calls (of the order of $N/2$) are saved by terminating recursion before sequences below the cutoff length are reached. The overhead is a single call to the insertion sort routine, to 'tidy up' what was abandoned by quicksort. Reducing a large number of calls will be beneficial on runtime.

5 Appendix A - Entry and exit traces

```
procedure Q_Sort( A: in out Input_Data ) is ...
begin
       new_line; put(" Q_Sort called on index range");
       PRINT_INDEX(A'First);
       PRINT_INDEX(A'Last);
    if A'Length > Cutoff then  -- Cutoff set at 2
       --
       -- partition statements
       --
       Q_Sort( A( A'First .. Pivot_Index-1 ) );
       Q_Sort( A( Pivot_Index+1 .. A'Last  ) );
       new_line; put(" Sorting now complete in index range");
       PRINT_INDEX(A'First); PRINT_INDEX(A'Last);
    end if;
end Q_Sort;
```

6 Appendix B - A trace for sorting 16 elements

```
 Q_Sort called on index range   1  16 with key values

  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10 -11 -12 -13 -14 -15 -16
   *                           *                               *
 pivot value chosen from (-1  -8  -16) was  -8.
 After partitioning  1  16:
 -16 -14 -13 -12 -11 -10  -9 -15( -8)  -6  -5  -4  -3  -2  -7  -1

 Q_Sort called on index range   1   8 with key values
 -16 -14 -13 -12 -11 -10  -9 -15
   *           *               *
 pivot value chosen from (-16 -12 -15) was -15.
 After partitioning  1   8: -16(-15) -13  -9 -11 -10 -14 -12

 Q_Sort called on index range   3   8 with key values
 -13  -9 -11 -10 -14 -12
   *           *       *
 pivot value chosen from (-13 -10 -12) was -12.
 After partitioning  3   8: -13 -14(-12) -10  -9 -11

 Q_Sort called on index range   6   8 with key values
 -10  -9 -11
   *   *   *
 pivot value chosen from (-10  -9 -11) was -10
 After partitioning   6   8: -11(-10)  -9)
```

```
Sorting now complete in index range   6   8 Result :
-11 -10  -9
Sorting now complete in index range   3   8 Result :
-13 -14 -12 -11 -10  -9
Sorting now complete in index range   1   8 Result :
-16 -15 -13 -14 -12 -11 -10  -9

Q_Sort called on index range  10  16 with key values
 -6  -5  -4  -3  -2  -7  -1
  *           *           *
pivot value  chosen from ( -6  -3  -1) was  -3.
After partitioning 10  16: -6  -5  -4  -7( -3)  -2  -1

Q_Sort called on index range  10  13 with key values
 -6  -5  -4  -7
  *   *       *
pivot value from ( -6  -5  -7) was  -6.
After partitioning 10  13: -7( -6)  -4  -5
Sorting now complete in index range  10  13 Result :
 -7  -6  -4  -5
Sorting now complete in index range  10  16 Result :
 -7  -6  -4  -5  -3  -2  -1
Sorting now complete in index range   1  16 Result :
-16 -15 -13 -14 -12 -11 -10  -9  -8  -7  -6  -4  -5  -3  -2  -1
```

The pairs (-13 -14) and (-4 -5) are not yet in the correct order. This is a consequence of 'Cufoff' being set to 2, and is corrected in the given code be a final sweep with the insertion sort algorithm, which is very efficient on an 'almost sorted' array.

7 Appendix C - Random and worst data

Table C1. Constants used in equation (1) to generate random integers.

N	c	d	N	c	d
15	3	17	1023	255	1031
31	7	37	2047	511	2053
63	15	67	4095	1023	4099
127	31	131	8191	2047	8191
255	63	257	16383	4095	16411
511	125	521			

Random data sequences for $N = 15, 31, 63$ are, from equation (1),

```
N = 15    C =  3    D = 17
   4   1  0  1  4  9 16  8  2 15 13 13 15  2  8
```

```
N = 31    C =  7    D = 37
  36  25 16  9  4  1  0  1  4  9 16 25 36 12 27  7
  26  10 33 21 11  3 34 30 28 28 30 34  3 11 31

N = 63    C = 15    D = 67
  62  35 10 54 33 14 64 49 36 25 16  9  4  1  0  1
   4   9 16 25 36 49 64 14 33 54 10 35 62 24 55 21
  56  26 65 39 15 60 40 22  6 59 47 37 29 23 19 17
  17  19 23 29 37 47 59  6 22 40 60 15 39 65 26
```

Note that every value can appear twice.

```
-- initialisation for worst case
  for I in A'First .. A'Last loop
    A( I ) :=  I;
  end loop;

  for I in 2 .. (A'Last-1)/2 loop
    Mid := I+1;
    A(2*I) := A(Mid);
    A(Mid) := 2*I;
    A(2*I+1) := 2*I+1;
  end loop;
-- end worst case
```

The first few sequences are:

```
                      1                          N =  1
                   1  2  3                       N =  3
                1  2  4  3  5                    N =  5
             1  2  4  6  5  3  7                 N =  7
          1  2  4  6  8  3  7  5  9              N =  9
       1  2  4  6  8 10  7  5  9  3 11           N = 11
    1  2  4  6  8 10 12  5  9  3 11  7 13        N = 13
                      ^
                    pivot
```

Starting with a sequence for any odd value of N, *all* the previous cases (down to the cutoff) will be generated during quicksort, leading to $O(N^2)$ performance.

8 Appendix D - Implementation of counters

The routines for compare, Swap and Move were modified to include counters. Whenever one of these routines is called, counter incrementation takes place. The sort routine is not changed at all (except that all references to "<", where counted key comparisons are involved, are renamed to "/").

```
    function "/" ( Left, Right: Input_Type ) return Boolean is
--  renamed "less than" operator, to intercept for
--  counting.
--
--  Others comparison operators (>, >=, <=) are
--  defined in terms of this operator, and so will
--  automatically trip the counter.
    begin
      Comps := Comps + 1;
      return ( Left < Right );
    end "/";

    procedure Swap( Left, Right: in out Input_Type ) is
        Temp : Input_Type;
      begin
        Temp := Left;
        Left := Right;
        Right := Temp;
        -- update counter
        Swaps := Swaps + 1;
      end Swap;

    procedure Move( Left: out Input_Type; Right: in Input_Type ) is
      begin
        Left := Right;
        -- update counter
        Moves := Moves + 1;
      end Move;

    procedure Initialise_Counts is
    begin
      Moves := 0;  Swaps := 0;  Comps := 0;
    end;
```

9 Appendix E - Checking sorted status

The following function was used after every call to a sorting routine, to ensure that data was actually sorted. The function Check_Sorted returns zero if any array elements are found out of sequence. Otherwise, it returns the highest number of equal items in the sorted sequence. For distinct data, this is 1, for equal data, this is N, for my pseudo random data use, it is 2.

```
    function Check_Sorted( A: Input_Data ) return NATURAL is
        Tmp : Boolean;
        Sequence, Longest_Sequence : Natural := 1;
    begin
        for I in A'First..A'Last - 1 loop
```

```
          Tmp :=  A(I+1) < A(I);
          if A(I+1) = A(I) then
             Sequence := Sequence + 1;
          else
             if Sequence > Longest_Sequence then
                Longest_Sequence := Sequence;
             end if;
             Sequence := 1;
          end if;
          exit when Tmp;
        end loop;
        if TMP then
           return(0);
        else
           if  Sequence > Longest_Sequence then
               Longest_Sequence := Sequence;
           end if;
           return(Longest_Sequence);
        end if;
    end Check_Sorted;
```

10 Appendix F - Acknowledgments

I thank Richard Huss of Advisory and Robert Woodall for help on several points of Ada generics.

11 Appendix G - References

DE Knuth . Sorting and Searching. The Art of Computer Programming, Volume 3, Addison Wesley, 1973.

Mark Allan Weiss . Data Structures and Algorithm Analysis in Ada Benjamin/Cummings Pubs. Co., 1993.

AV Aho, JE Hopcroft and JE Ullman . Data structures and algorithms. Addison Wesley, 1983.

RL Kruse . Data Structures and Program Design. Prentice Hall, 1984.