Cycle-Sort: A Linear Sorting Method

Article *in* The Computer Journal · April 1990

DOI: 10.1093/comjnl/33.4.365 · Source: DBLP

Bruce K. Haddon
University of Colorado Boulder

Cycle-Sort: A Linear Sorting Method

B. K. HADDON[*]

Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado 80309-0425, USA

Two sorting algorithms are described,* general_cycle_sort*, and* special_cycle_sort*, based upon the decomposability of a permutation into a product of cyclic permutations. Under the conditions in which these algorithms are applicable, the sorting can be accomplished in linear time.

Received May 1987, revised December 1987; further additions, June 2016.

1. INTRODUCTION

The two sorting algorithms to be presented here have applications in rather specialised situations, but situations that are not uncommon in practice. They both operate in linear time, and sort 'in place.'

The first of the algorithms, *general_cycle_sort*, is useful where the following conditions apply:

(1) The keys are integers drawn from a reasonably small set (or can be mapped into such a set):

(2) The number of occurrences of each key is known.

These circumstances can arise when the elements to be sorted are subject to some earlier processing pass, in which the keys are examined and a histogram counting the occurrences of each key is accumulated. For example, the keys could indicate membership of one of a small number of sets. Programs that classify symbols (*e.g.*, compilers), analyse data (*e.g.*, cluster analysis), play games (*e.g.*, form groups of like moves), *etc*., can all generate the appropriate circumstances.

To make this clear, consider a compiler that lists, after other processing, all symbols collected together by type. Each type can be denoted within the compiler by some small ordinal, and the number of occurrences of declarations of symbols of each type can be counted as the source program is compiled. The conditions for *general_cycle_sort* are then met, and the sorting can be accomplished in linear time.

It frequently happens that in specific sorting applications, each key occurs just once. For example, if transaction records are numbered as they are input, with the object of returning them to input order after some other processing, then the keys for re-sorting to that order constitute such a case. This is the situation in which the second method, *special_cycle_sort*, may be used.

Overview of method

An array of elements to be sorted can be viewed as a permutation of the elements of the sorted order desired. An arbitrary permutation can be represented as a product of cyclic permutations, or 'cycles' (Ref. 4, p. 66). A cycle can be represented by a vector, written in the form:

$$(a_1, a_2, a_3, \dots a_c)$$

where c is the length of the cycle, and the cycle described is the one where the element at a_2 moves to position a_1, and the element at a_3 moves to position a_2, and so on, and the element at a_1 moves to position a_c.

A permutation can be represented by a function $p(j)$, say, where j is the position of the element in the array to be sorted, that is to be at position $p(j)$ when it is sorted. Sorting consists of determining $p(j)$, which can be computed by finding the value of j (uniquely) in one of cycles in the product forming $p(j)$, and taking as the value of $p(j)$ the value (circularly) to the left of j in that cycle.

The cycle sorting method consists of finding a representative array element in each cycle of the permutation of the unsorted array, and then performing the cyclic permutation specified by that cycle, for each of the cycles in the permutation's product. The result is then the desired sorted array.

2. THE GENERAL CASE

The first of the algorithms is named *general_cycle_sort*. It makes use of a histogram that counts the occurrences of each of the keys in the records that are to be sorted. The histogram is converted to a 'cumulative' form at the beginning of processing, and is used thereafter to track the position into which the record with the corresponding key is to be placed as the sort proceeds.

The original version of this paper was published in *The Computer Journal* **33** (4), pp 365 - 367, October 1990.
http://comjnl.oxfordjournals.org/content/33/4/365.full.pdf+html

[*] Now at Paladin Software International, 1506 Chambers Drive, Boulder, CO 80305-7002 USA. Bruce.Haddon@colorado.edu

2.1. The algorithm

For the moment, it will be assumed that the keys of the elements to be sorted is the set of integers 1 .. *m*, where the cardinality *m* is of a size that the appropriate arrays may be declared (this assumption will be examined further below). In the following discussion, and in the algorithm shown in Fig. 1, the array to be sorted will be of the type described by these (Pascal) declarations:

```
const n = ... { the size of the array to be sorted };
       m = ... { cardinality of the key set }
type array_size = 1 .. n;
     element_count = 0 .. n;
     key_set = 1 .. m;
     array_element =
         record key: key_set;
             element_body: ...
         end;
     sort_array = array [array_size] of array_element;
     histogram= array [key_set] of element_count;
```

The *sort_array* is initialised to contain the elements to be sorted, and the histogram is initialised to contain the counts of the elements with keys of each value in the *key_set*.

When the *general_cycle_sort* is called, the initialisation and loop of lines (1)-(3) of Fig. l compute a new, cumulative, histogram, which represents (one below, to simplify the Pascal type declarations) the place in the sort array where the first element for each key value is to be placed, *i.e.*, for key value *k*, *p*[*k*](+ 1) is the place in the array *a* where the first element with that key value should be placed.

The remainder of the procedure body, lines (4)-(25), is a loop searching for elements that are to be placed at a position below (*i.e.*, at positions of lesser index value) in the array than where they currently are. The element at position *i* has a key value *k*, and this element should be at the position indexed by *j*, where *j* = p[k] + 1. The element is too far up the array if $i > j$. (The test at line (8) is for $i \geq j$—the case $i = j$ will be discussed below—the test $i \neq j$ at line (11) restricts consideration to $i > j$ in lines (12)-(19).

The **repeat** loop starting at line (13) takes the element that is out of place, and puts it at the position determined by *p*[*k*], saving the element that was there, in the interim. The entry *p*[*k*] is then incremented by one, to indicate where the next element with a key value of *k* is to be placed. The loop continues until an element that belongs in position *i* is found, and the cycle is closed.

If an element is found in position *i* that belongs in position *i* (a unit cycle, the case $i = j$), the corresponding *p*[*k*] is incremented, so that when another element with a key value of *k* is found, it will be placed following the one that was just examined.

```
procedure general_cycle_sort(
          var a: sort_array; n: array_size;
          var h: histogram; m: key_set);
      var i, j: array.size;
          k: key.set;
          p: histogram;
          save, temp: array_element;
      begin
 (1)      p[1] := 0;
 (2)      for k := 1 to m - 1 do
 (3)          p[k + 1] := p[k] + h[k];
 (4)      for i := 1 to n do
 (5)      begin
 (6)          k := a[i].key;
 (7)          j := p[k] + 1;
 (8)          if i ≥ j then
 (9)          begin
(10)              if i ≠ j then
(11)              begin
(12)                  save := a[i];
(13)                  repeat
(14)                      temp := a[j];
(15)                      a[j] := save;
(16)                      save := temp;
(17)                      p[k] := j;
(18)                      k := save.key;
(19)                      j := p(k)+1
(20)                  until i = j { repeat };
(21)                  a[i] := save
(22)              end { if i ≠ j };
(23)              p[k] := i
(24)          end { if i ≥ j }
(25)      end { for i := 1 to n }
      end { procedure general_cycle_sort };
```

Figure 1

2.2. An application

In an investigation being made by this author of a program playing the game of Mastermind,[1] all moves gaining the same scoring response must be sorted into a 'partition' that collects all equivalent moves. As each move is evaluated, the scoring response is mapped into a set of integers, and the corresponding entry of a histogram is incremented. After all the moves feasible at each stage have been evaluated, the sorting into partitions is accomplished with *general_cycle_sort*.

For the standard game, using six colours of pegs, with four pegs in each pattern, a move can be any of the possible 1296 (6^4) patterns. It can be shown that the score can be mapped into the set of integers 1 .. 16. Thus in the Pascal declarations above, *n* is 1296, and *m* is 16. The *element_body* for this application is the vector of four colours used in the pattern, hence the following declarations are needed:

```
type colour= (black, blue, green, red, yellow, white);
     array_element =
         record key: key_set;
             element_body: array [1 .. 4] of colour;
         end;
```

3. A SPECIAL CASE

The conditions for the special case are those that make p just the identity mapping, corresponding to just a single occurrence of each key, and the keys run from 1 to the number of items to be sorted. Thus the type *key_set* is the same as *array_size*.

Assuming this, *general_cycle_sort* can be modified, removing the variable k and the array p, and the references to them. The resulting procedure, *special_cycle_sort*, is shown in Fig. 2. *special_cycle_sort* has the same time complexity as *general_cycle_sort*, but requires no additional storage for the histogram (or its cumulative form).

```
     procedure special_cycle_sort
                 (var a: sort_array; n: array_size);
          var  i, j: array_size;
               save, temp: array_element;
          begin
 (1)           for i := 1 to n do
 (2)           begin
 (3)                j := a[i].key;
 (4)                if $i \neq j$ then
 (5)                begin
 (6)                     save := a[i];
 (7)                     repeat
 (8)                          temp := a[j];
 (9)                          a[j] := save;
(10)                          save := temp;
(11)                          j := save.key;
(12)                     until i = j { repeat };
(13)                     a[i] := save;
(14)                end { if $i \neq j$ };
(15)           end {for i: = 1 to n }
          end { procedure special_cycle_sort };
```

Figure 2

3.1. An application

In the ‘garbage collection’ method previously reported,[3] an entry was made in a rolling table for each region of unallocated memory. The table is ‘rolled’ between entries being made, thus at the end of the scan of the memory, the table is not in memory address order. In this application, use was made of the fact that each new entry is generated in address order as memory is scanned from lower addresses to higher addresses, and unused space in the entry was used to hold the ordinal of the entry. Thus at the end of the scan, this ordinal field satisfies the requirements for *special_cycle_sort*, and the table could be reorganised into the required order in linear time.

4. CORRECTNESS, AND REQUIREMENTS

An informal discussion of the correctness of the cycle sorts starts with the observation that once a cycle has been entered, it can be seen that all the elements of the cycle are placed in their proper respective positions.

The algorithms must necessarily find all cycles, since either a cycle will be a unit cycle, or it must contain at least one element that is at a ‘higher’ position in the array that it should be, for if it did not, we would have the condition that:

$$a_1 < a_2 < a_3 < ... < a_c < a_1,$$

which is not possible. The algorithm passes over all pairs for which the ‘ < ’-relation holds, and enters it when the inverse condition is found. Hence all cycles are found. Moving the elements around a cycle decomposes it into a product of unit cycles, and, as discussed above, elements of unit cycles are not moved, $p[k]$ is simply incremented. Hence elements are not moved a second time.

A formal proof of the correctness of the cycle sorts can be derived by observing that they are effectively modifications of the procedure *permute* of Ref. 2, where, in *general_cycle_sort*, p is the inverse function of the function f of that paper. The amount of moving of data is reduced in the cycle sorts over *permute* by following the cycle, rather than doing a series of exchanges as each element of the cycle is rediscovered by a linear scan. The changes to p as the cycle is traversed are just the changes needed to make p be the inverse of f for the new permutation generated by moving elements around their cycle.

4.1. Time requirements

At worst, each element in the sort array is touched twice by the procedures—once by the scan controlled by i, and once when the cycle containing the element is traversed. The time cost of the cycle traversal of each of the algorithms is therefore $\boldsymbol{O}(n)$. *general_cycle_sort* requires additionally the initialisation of the histogram into cumulative form, which is an operation of $\boldsymbol{O}(m)$.

The time complexity of the algorithms is therefore bounded by $\boldsymbol{O}(m + n)$. For practical applications, this is linear in n.

4.2. Space requirements

The caller to *general_cycle_sort* must provide the space for the array to be sorted, and for the histogram. The major storage requirement of the *general_cycle_sort* procedure is the cumulative histogram array that is declared locally.

If there is no necessity to preserve the caller's histogram, the procedure can be modified by renaming the parameter h to p, replacing the declaration for the local histogram by two variables s and v of type *element_count*, and replacing lines (1) - (3) by:

```
(1)   s := 0;
(2)   for k := 1 to m do
(3a)  begin
(3b)       v := p[k];
(3c)       p[k] := s;
(3d)       s := s + v
(3e)  end;
```

The actual size of the histogram is, of course, dependent upon the cardinality of *key_set*, which, in turn, is determined by the particular problem.

CONCLUSION

The cycle sort techniques are not universally applicable, but in those places where they can be used, they offer the advantage of a linear time complexity for little or no additional storage. In a sense, the *cycle_sort*s can be regarded to be variants of the class of radix sorts, with the cycles being the buckets.

It is interesting to note that since efficient general sorting methods are of a time complexity of $\boldsymbol{O}(n \log n)$, and that the cycle sorting methods use a histogram of key occurrences to create conditions in which the sorting may be accomplished in a time complexity of $\boldsymbol{O}(n)$, then the histogram must represent, on the average, $\log n$ bits of information per element to be sorted, or $(\log n)/m$ per entry.

In applications where it is possible to create the needed histogram, the actual time required to do this is itself of $\boldsymbol{O}(n)$, hence the entire job of creating the histogram and performing a cycle sort may be accomplished in linear time.

Acknowledgements

Professors H. Jordan and W. M. Waite, of the Department of Electrical and Computer Engineering[*] at the University of Colorado at Boulder, and an anonymous referee, made several helpful suggestions, based on earlier drafts, regarding the presentation of the material.

REFERENCES

1. L. H. Ault, *The Official Mastermind Handbook*. A Signet Book, The New American Library, NJ, pp. 136 (1977).
2. A. J. W. Duijvestijn, *Correctness proof of an in-place permutation*. *Bit* 12, 318-324 (1972).
3. B. K. Haddon and W. M. Waite, A compaction procedure for variable-length storage elements. *The Computer Journal* **10** (2), 162-165 (1967).
4. W. Ledermann, *Introduction to the Theory of Finite Groups*. Oliver and Boyd, London, p. 174 (1961).

[*] Now the Department of Electrical, Computer and Energy Engineering at the University of Colorado at Boulder.

Appendix

Since the original publication of these algorithms, focus has moved from Pascal as a programming language of record (as it moved from the earlier use of Algol) to C, C++, and more notably, Java. This appendix re-presents both versions of the cycle sorts using Java notation.

In doing so, some of the surrounding assumptions have been changed, to accommodate the zero-based indexing normally used in Java. The principal implications of this are that the set of keys is now assumed to be the set $0 .. m - 1$, and the array to be sorted has indexes in the range $0 .. n - 1$, but that the count of elements is still in the range $0 .. n$. This has the additional effect that the array $p[j]$ (in `generalCycleSort`) now points at the actual next location for the key value j, rather than one less than that (as was done in the Pascal version). Another effect is that there is no longer a type definition to define these ranges. That the values are so limited become the preconditions of the methods.

Since Java arrays carry their *length* as a property, there is no need to represent these values with additional parameter values.

Finally, although the method and class names follow the normal Java conventions, the variable names from the Pascal version have been kept, even though they are rather more attenuated than in the usual Java approach.

The *array_element* **record** of the Pascal version is replaced by the Java class definition:

```
public class ArrayElement
{
public int key;
private Body body;

private class Body
{
    // …
}
}
```

```
public void generalCycleSort(
                ArrayElement[] a, int[] h)
{
    final int n = a.length;
    final int m = h.length;
    int[] p = new int[m];
    p[0] = 0;
    for( int k = 0; k != m - 1; ++k )
        p[k + 1] = p[k] + h[k];

    for( int i = 0; i != n; ++i )
    {
        int k = a[i].key;
        int j = p[k];
        if( i >= j )
        {
            if( i != j )
            {
                ArrayElement save = a[i];
                do
                {
                    ArrayElement temp = a[j];
                    a[j] = save;
                    save = temp;
                    p[k] = j + 1;
                    k = save.key;
                    j = p[k];
                } while( i != j );
                a[i] = save;
            }
            p[k] = i + 1;
        }
    }
}
```

Figure 3

```
public void specialCycleSort(
                ArrayElement[] a)
{
    final int n = a.length;
    for( int i = 0; i != n; ++i )
    {
        int j = a[i].key;
        if( i != j )
        {
            ArrayElement save = a[i];
            do
            {
                ArrayElement temp = a[j];
                a[j] = save;
                save = temp;
                j = save.key;
            } while( i != j );
            a[i] = save;
        }
    }
}
```

Figure 4