Appl. Math. Lett. Vol. 4, No. 2, pp. 57–59, 1991
Printed in Great Britain

0893-9659/91 $3.00 + 0.00
Pergamon Press plc

Aesthetic Sorting

JEFFREY K. UHLMANN

Information Technology, Naval Research Laboratory

(Received July 1990)

Abstract. The problem of sorting a sequence of numbers is considered from the perpective of mathematical aesthetics. In particular, a sorting algorithm is developed in which comparisons and exchages of index values are performed implicitly using ordinary arithmetic operators rather than computerish *if-then* instructions.

Sorting is one of the most common processes performed by computers and, consequently, one of the most analyzed in computer science. For computer scientists the evaluation of a given sorting algorithm consists of determining how the number of fundamental operations which must be performed scales with the number of items to be sorted. In particular, scaling is usually measured in terms of the number of comparisons and exchanges (or *swaps*) an algorithm requires. For the mathematician, however, such mundane considerations may seem unimportant since they are motivated by the purely practical constraints of actual computing hardware. In other words, the mathematician who has no concern with the limitations of computers might adopt a completely different measure for sorting algorithms. In this article we will consider such a measure and develop an algorithm that is tailored to the tastes of the mathematically inclined.

The problem of sorting consists of finding a permutation of a set of n numbers $\{a_1, \ldots, a_n\}$, such that for any two elements a_i and a_j, $i < j$, $a_i \leq a_j$. A straightforward computer algorithm for doing so is the following:

```
(1)   for i = 1 to n
(2)       for j = i to n
(3)           if a_i > a_j then
(4)               temp ← a_i
(5)               a_i ← a_j
(6)               a_j ← temp
(7)           end
(8)       end
(9)   end.
```

The principle behind this algorithm consists of successively assigning the smallest number in the set to the indexed element a_1, the next smallest to a_2, and so on. A cursory examination of the steps reveals that the number of comparisons (step (3)) scales as $O(n^2)$, and the number of swaps (steps (4)–(6)) could be of the same order. By these measures, the algorithm does not compare favorably to others for which the number of required comparisons and swaps is $O(n \log n)$. For our purposes, however, the above algorithm is superior to those other more efficient algorithms

whose full descriptions tend to be rather unsightly. In other words, we are more concerned with the aesthetic qualities of algorithms than with how efficiently they can be executed by a computer.

Although our algorithm is straightforward, it has a very distinctive computerish flavor. In particular, mathematicians rarely phrase operations using *if-then* conditions. Furthermore, the need for the *temp* variable as a placeholder when swapping seems rather cumbersome. In fact, the entire approach of making explicit comparisons and swaps is certainly not very imaginative nor elegant. The question then is whether we can achieve the same results using standard mathematical operators and notation.

The reason for comparing a_i and a_j in step (3) is to determine the respective positions of the lesser and greater of the two values so that the lesser can then be assigned to indexed element a_i and the greater to a_j. In other words, what we really want is to make the assignments $a_i \leftarrow \min(a_i, a_j)$ and $a_j \leftarrow \max(a_i, a_j)$. Of course, to use min() and max() functions would be no less *deus ex machinae* than to assume a compare-and-swap() function. However, the recognition of the relationship between these operations provides an indication of how we might proceed. Specifically, the problem reduces to the expression of min() and max() in terms of standard arithmetic operations. Fortunately, this can be achieved by verifying the following:

$$\min(a_i, a_j) \equiv \frac{1}{2}(a_i + a_j - |a_i - a_j|),$$
$$\max(a_i, a_j) \equiv \frac{1}{2}(a_i + a_j + |a_i - a_j|).$$

Thus, we can rewrite our sorting algorithm without an explicit comparison step as follows:

```
(1)     for i = 1 to n
(2)         for j = i to n
(3)             temp ← a_i
(4)             a_i ← (a_i + a_j - |a_i - a_j|) / 2
(5)             a_j ← (temp + a_j + |temp - a_j|) / 2
(6)         end
(7)     end.
```

The result is surely less pedestrian than the original, but the *temp* variable has now become an even more prominent sacrifice to the constraints of sequentialism. How we can rid ourselves of this burden is not so obvious. After some thought, though, one should realize that any invertible operator can solve our problem. For example, the following steps can be performed to exchange the values of a_i and a_j:[1]

$$a_i \leftarrow a_i \text{ XOR } a_j$$
$$a_j \leftarrow a_i \text{ XOR } a_j$$
$$a_i \leftarrow a_i \text{ XOR } a_j,$$

where the XOR function is defined by the following truth table:

XOR	0	1
0	0	1
1	1	0

[1] A similar "trick" employed by computer scientists is to swap the contents of two variables, stored in binary of course, using the exclusive-or (XOR) function as follows:

The advantage of the self-inverting XOR function on computers is that it avoids the possible overflow problems associated with addition/subtraction on fixed-length binary variables.

$$a_i \leftarrow a_i + a_j$$
$$a_j \leftarrow a_i - a_j$$
$$a_i \leftarrow a_i - a_j .$$

The most appealing feature of this result is that it not only eliminates a superfluous variable, it also eliminates the clutter of redundant operations in our sorting algorithm. Specifically, we obtain the following sufficient steps for sorting:

```
(1)     for i = 1 to n
(2)         for j = i to n
(3)             a_i ← a_i + a_j
(4)             a_j ← (a_i ± |a_i - 2a_j|) / 2
(5)             a_i ← a_i - a_j
(6)         end
(7)     end,
```

where the '±' in step (4) signifies that a '+' or a '−' can be used to sort the elements in ascending or descending order, respectively.

Information Technology—Code 5579, Naval Research Laboratory, Washington, D.C. 20375