Batcher's Algorithm

Prof. Michel Goemans

Perhaps the most restrictive version of the sorting problem requires not only no motion of the keys beyond compare-and-switches, but also that the plan of comparison-and-switches be fixed in advance. In each of the methods mentioned so far, the comparison to be made at any time often depends upon the result of previous comparisons. For example, in HeapSort, it appears at first glance that we are making only compare-and-switches between pairs of keys, but the comparisons we perform are not fixed in advance. Indeed when fixing a headless heap, we move either to the left child or to the right child *depending* on which child had the largest element; this is not fixed in advance. A *sorting network* is a fixed collection of comparison-switches, so that all comparisons and switches are between keys at locations that have been specified from the beginning. These comparisons are not dependent on what has happened before. The corresponding sorting algorithm is said to be *non-adaptive.*

We will describe a simple recursive non-adaptive sorting procedure, named Batcher's Algorithm after its discoverer. It is simple and elegant but has the disadvantage that it requires on the order of $n(\log n)^2$ comparisons. which is larger by a factor of the order of $\log n$ than the theoretical lower bound for comparison sorting. For a long time (ten years is a long time in this subject!) nobody knew if one could find a sorting network better than this one. Then Ajtai, Komlós, and Szemerédi proved the existence of very complex networks that can sort with $cn \log n$ comparisons for a very large constant c. Their procedure is extremely complicated, and for most practical purposes Batcher's algorithm is more useful.

The idea behind Batcher's algorithm is the following claim (which at first glance looks incredible): If you sort the first half of a list, and sort the second half separately, and then sort the odd-indexed entries (first, third, fifth, ...) and the even-indexed entries (second, fourth, sixth, ...) separately, then you need make only one more comparison-switch per pair of keys to completely sort the list. We will prove this below.

For the remainder of this section, we will assume that the length of our list, n, is a power of 2. Let's see what happens for a particular list of length 8. Suppose we are given the following list of numbers:

$$2 \quad 7 \quad 6 \quad 3 \quad 9 \quad 4 \quad 1 \quad 8$$

We wish to sort it from least to greatest. If we sort the first and second halves separately we obtain:

$$2 \quad 3 \quad 6 \quad 7 \quad 1 \quad 4 \quad 8 \quad 9$$

Sorting the odd-indexed keys $(2, 6, 1, 8)$ and then the even-indexed keys $(3, 7, 4, 9)$ while leaving them in odd and even places respectively yields:

$$1 \quad 3 \quad 2 \quad 4 \quad 6 \quad 7 \quad 8 \quad 9$$

This list is now almost sorted: doing a comparison switch between the keys in positions (2 and 3), (4 and 5) and (6 and 7) will in fact finish the sort.

This is no accident: given any list of length 8, if we sort the entries in the first half, then sort the entries in the second half, then sort the entries in odd positions, then sort the entries in even positions and lastly perform this same round of exchanges (second with third, fourth with fifth, and sixth with seventh), the list will end up sorted. Furthermore, and even more incredibly, the same fact holds for *any* list whose length is a multiple of 4 (as we shall see below); in that case, in the final step, we sort the 2ℓ-th element with the $(2\ell+1)$-st for $\ell = 1, 2, \cdots, (n/2) - 1$.

Theorem 1. *For any list of length n, where n is a multiple of 4, first sorting separately the first and second halves, then sorting separately the odd-indexed keys and the even-indexed keys, and finally comparing-and-switching the keys indexed 2ℓ and $2\ell+1$ for $\ell = 1, 2, \cdots, (n/2) - 1$ results in a sorted list.*

Proof. After the two halves of the list have been sorted separately, it is obvious that for all i between 1 and n except for 1 and $\frac{n}{2}+1$, the $(i-1)$-st element of the list is less than the i-th. Call the $(i-1)$-st key the predecessor of the i-th. Note that 1 and $n/2+1$ are both odd. Every even-indexed key has as its predecessor a number smaller than itself (since any even-indexed key and its predecessor are in the same half), so the ℓ-th smallest even-indexed key must be larger than at least ℓ odd-indexed keys (look at its predecessor as well as the predecessors of the $\ell - 1$ even-indexed keys that are smaller than it). Similarly, every odd-indexed key (with the possible exception of two keys, namely the 1st and $(\frac{n}{2}+1)$-st) has as its predecessor a number smaller than itself, so the $\ell+1$-st smallest odd-indexed key must be larger than at least $\ell - 1$ even-indexed keys.

If we denote by k_i the i-th indexed key after sorting the first and second halves and the even and odd halves, we have just argued that

$$k_{2\ell-1} \leq k_{2\ell}$$

and

$$k_{2\ell-2} \leq k_{2\ell+1},$$

for any appropriate ℓ. Since we have sorted the even indexed keys and the odd indexed keys, we also know that $k_{2\ell-2} \leq k_{2\ell}$ and that $k_{2\ell-1} \leq k_{2\ell+1}$. Thus, if we group the elements in pairs $(k_{2\ell}, k_{2\ell+1})$ for each appropriate ℓ, we see that both elements of a pair are greater or equal to both elements of the previous pair. To finish the sort after sorting the odds and evens, it is therefore only necessary to compare the $\ell+1$-st smallest odd-indexed key $(k_{2\ell+1})$ to the ℓ-th smallest even-indexed key $(k_{2\ell})$ for each appropriate ℓ to determine its rank completely; this is precisely what the final step does. □

If we start with a list whose size is a power of two, we can use this idea to create longer and longer sorted lists by repeatedly sorting the first and second halves and then the odd and even entries followed by one additional round of comparison-switches. This task is slightly easier than it looks, as a result of a second observation: If you first sort the first and second halves, then the first and second halves of the odds and evens are already sorted. As a result, when sorting the odds and evens here, we only need to merge the two halves of the odd indexed keys and the two halves of the even indexed keys. We have shown how to merge 2 sorted lists of length p while performing only $2p-1$ comparisons, but this requires an *adaptive* algorithm. Instead, Batcher's procedure proceeds recursively and has two different components: a sorting algorithm that sorts completely disordered lists and a merging algorithm that sorts lists whose first and second halves have already

been sorted. We will call these algorithms SORT and MERGE. Let us emphasize that MERGE only sorts if the first half and second half are already sorted. If not, it is unclear what MERGE does. Our plan when executing Batcher's procedure is to successively merge smaller lists to get larger ones, but we use Batcher's merging algorithm rather than the simple merge, to make it non-adaptive.

It follows that Batcher's Algorithm can be written in the following recursive form (provided n is a power of 2 larger than 2):

- SORT$(x_1, \ldots, x_n)$ calls:

  SORT$(x_1, \ldots, x_{n/2})$, then SORT$(x_{n/2+1}, \ldots, x_n)$, and then MERGE$(x_1, \ldots, x_n)$.

- MERGE$(x_1, \ldots, x_n)$ calls:

  MERGE(x_i, for i odd), then MERGE(x_i for i even), and then COMP(x_2, x_3), COMP$(x_4, x_5), \cdots$ COMP(x_{n-2}, x_{n-1}).

- COMP(x_i, x_j) means:

  compare the key in the position i with the one in position j and put the larger one in position j, the smaller one in position i.

In the definition of MERGE, notice that if the first and second halves of $(x_1, \ldots, x_n)$ are already sorted then the first and second halves of both (x_i, for i odd) and (x_i for i even) are already sorted as well, and therefore will become sorted after their MERGE calls. For the base case, we should say that

$$\text{SORT}(x_1, x_2) = \text{MERGE}(x_1, x_2) = \text{COMP}(x_1, x_2).$$

We now turn to the question: how many steps does this algorithm take? Let $S(n)$ denote the number of comparisons needed to sort n items, and let $M(n)$ denote the number of comparisons needed to merge two sorted lists of $n/2$ items each. Then we have

$$S(n) = 2S(n/2) + M(n),$$

$$M(n) = 2M(n/2) + n/2 - 1$$

with the initial conditions $S(2) = M(2) = 1$. If we ignore the "-1" term (which is certainly fine if we're aiming for an upper bound on $S(n)$), the second recurrence relation has the solution

$$M'(n) = \frac{n}{2} \log n.$$

You can verify this by induction. Therefore $M(n) \leq \frac{n}{2} \log n$. We can use this to show that $S(n) \leq \frac{n}{2} (\log n)^2$. We will see another way to count the number of comparisons below.

We now examine the Batcher procedure from the bottom up rather from the top down as we have done so far. We have seen that the procedure for sorting n items reduces to two applications of the procedure for sorting $n/2$ items followed by two applications of the procedure for merging $n/2$ items followed by a single round of comparison-switches. The procedures for sorting-merging $n/2$ items similarly involve procedures for sorting-merging $n/4$ items, which involve procedure for sorting $n/8$ items, and so on, until we get down to sorting single pairs. From the opposite point of view, we start from an unsorted list, sort into pairs, then apply several rounds of these sorts to create sorted groups of size four, then sorted groups of size eight, and so on until we have sorted all

n keys. We can use this description of Batcher's Algorithm to deduce what comparison switches are involved in sorting 2^j keys in this way.

We will describe rounds of comparisons; a round will be a set of comparison-switches which use each key at most once and hence can be performed simultaneously; this is for example useful for a parallel implementation of a sorting algorithm.

The first step is to arrange odd-even adjacent pairs of keys into ordered pairs. This can be done in one round.

$$(1,2),(3,4),(5,6),\ldots$$

For brevity, we write $(1,2)$ for $\textsc{Comp}(x_1,x_2)$.

To sort into groups of four we then want to sort the odd and even-indexed keys into pairs by applying a similar round to the odd and even-indexed keys separately and in parallel; we then perform the final step for sorting into groups of four, by doing the comparisons $(a,a+1)$ for $a \equiv 2 \pmod 4$.

$$(1.3),(2,4),(5,7),(6,8),\ldots$$

$$(2,3),(6,7),\ldots$$

Producing sorted groups of size eight from what we now have then involves repeating these last two rounds on odds and evens, and then doing the final round for sorting into groups of eight (see upcoming figure).

$$(1,5),(2,6),(3,7),(4,8),(9,13)\ldots$$

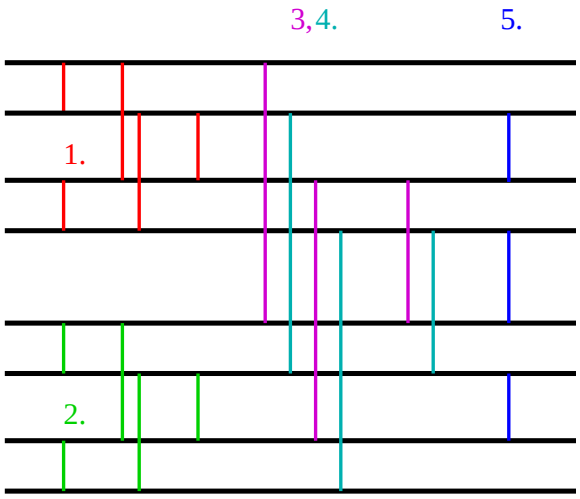
$$(3,5),(4,6),(11,13),(12,14),\ldots$$

$$(2,3),(4,5),(6,7),(10,11),\ldots$$

Each merging step here involves one more round of comparisons than the previous one, because we must repeat the previous round on odds and evens and add a final round. Producing sorted 8's from sorted 4's takes 3 rounds of comparisons (see upcoming figure); in general, getting sorted 2^k's from sorted 2^{k-1}'s takes k rounds of comparisons.

To actually sort $n = 2^k$ keys in this way involves making them into sorted 2's, then sorted 4's, then sorted 8's, etc,. until one has sorted 2^k's. This takes $1+2+3+\ldots+k$ rounds of comparisons, which is $k(k+1)/2$ rounds. Since each round takes no more than $n/2$ comparisons and $k = \log n$, we need approximately $n(\log n)^2/4$ comparisons in this algorithm. In fact, when n is sufficiently large, it can be shown that the number of comparisons required becomes strictly less than $n(\log n)^2/4$.

We schematically represent Batcher's algorithm as a sorting network in the following way. We draw an horizontal line (wire) for each key to be sorted and the convention is that processing goes from left to right. A compare-and-switch is represented by a vertical line segment between the corresponding wires; the larger key is output on the top wire (to the right of the comparator, as processing goes from left to right) and the smaller key on the bottom wire. Inputs enter the sorting network on the left (with the ith key on the ith wire from the top) and outputs can be read on the right side. The comparisons made in each round of Batcher's algorithm are drawn closely to each other. Here is such a representation for Batcher's algorithm on 8 inputs; the color coding refers to sorting in groups of size 8 from groups of size 4.



Key:
1. Sort on first half.
2. Sort on second half.
3. Merge on odd keys.
4. Merge on even keys.
5. Final compare and switch of adjacent keys.

Batcher's algorithm is not difficult to implement. If one can write recursive code, the initial description we gave can be coded directly. Otherwise, one can explicitly construct the rounds of comparisons that are to be made as described above.