**Sorting Network**

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*.

**Odd-even mergesort**

Given sorted two lists, odd-even mergesort sort odd-indexed entries (first, third, fifth, …) and the even-indexed entries (second, fourth, sixth,…) separately (independently), then perform comparison-switch per pair of keys to completely sort the list. It assume 1 as the first index.

**Example**

Suppose we are given the following list of numbers:

2 7 6 3 9 4 1 8

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

2 3 6 7 1 4 8 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 3 2 4 6 7 8 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

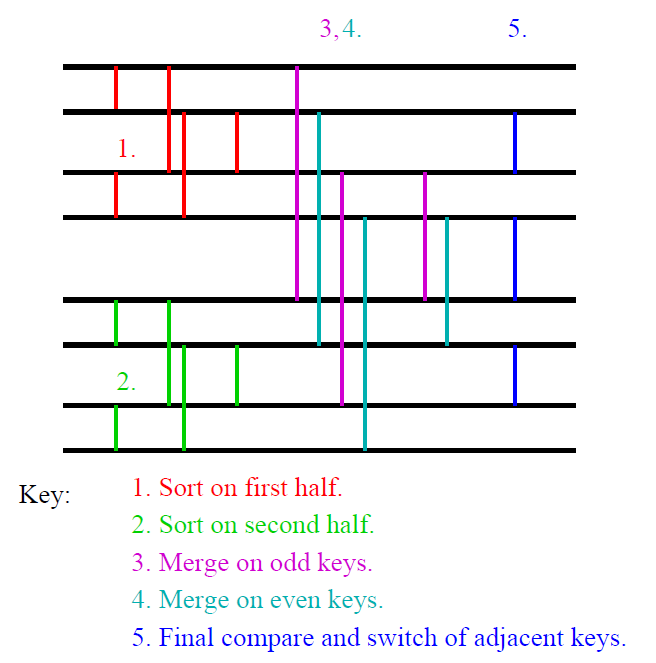
the entries in the second half, then sort the entries in odd positions, then sort the entries in even

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 2m-th element with the (2m+ 1)-st for m = 1, 2, · · · , (n/2) − 1.



**Algorithm *odd-even mergesort*(*n*)**

**Input:** sequence *a*0, ..., *an*-1 (*n* a power of 2)

**Output:** the sorted sequence

**Method:** if *n*>1 then

1. apply *odd-even mergesort*(*n*/2) recursively to the two halves *a*0, ..., *an*/2-1 and *an*/2, ..., *an*-1 of the sequence

2. *odd-even merge*(*n*)

**Java code**: the length n of the array must be a power of 2

**public class** OddEvenMergeSorter **implements** Sorter

{

**private int**[] a;

**public void** sort(**int**[] a)

{

**this**.a=a;

oddEvenMergeSort(0, a.length);

}

/\* sorts a piece of length n of the array starting at position lo \*/

**private void** oddEvenMergeSort(**int** lo, **int** n)

{

**if** (n>1)

{

**int** m=n/2;

oddEvenMergeSort(lo, m);

oddEvenMergeSort(lo+m, m);

oddEvenMerge(lo, n, 1);

}

}

/\* lo is the starting position and n is the length of the piece to be merged,

r is the distance of the elements to be compared \*/

**private void** oddEvenMerge(**int** lo, **int** n, **int** r)

{

**int** m=r\*2;

**if** (m<n)

{

oddEvenMerge(lo, n, m); // even subsequence

oddEvenMerge(lo+r, n, m); // odd subsequence

**for** (**int** i=lo+r; i+r<lo+n; i+=m)

compare(i, i+r);

} **else**

compare(lo, lo+r);

}

**private void** compare(**int** i, **int** j)

{

**if** (a[i]>a[j])

exchange(i, j);

}

**private void** exchange(**int** i, **int** j)

{

**int** t=a[i];

a[i]=a[j];

a[j]=t;

}

} // end class OddEvenMergeSorter

**Conclusions**

There are other sorting networks that have a complexity of O(n log(n)2), too, e.g. bitonic sort and shellsort. However, oddeven mergesort requires the fewest comparators of these. The following table shows the number of comparators for n = 4, 16, 64, 256 and 1024.

