**Sorting Networks**

When sorting lists of some specific size, sorting networks are often employed. These are very simple devices that only do *compare-exchange* operations. A compare-exchange operation on the pair \(<x, y>\) is merely the following code.

\[
\text{if } x > y \text{ then } \{ t = x; \ x = y; \ y = t \}
\]

Consider the network example in figure 1.

![Figure 1 - A Four-Element Sorting Network](http://www.cs.uky.edu/~lewis/essays/algorithms/sortnets/sort-net.html)

Here is how it works. The pairs of blue dots are modules that perform compare-exchange operations. Thus, in the first column, the pairs of numbers \(<3, 8>\) and \(<6, 2>\) enter the modules where the 6 and 2 are exchanged while the 3 and 8 remain in their original positions. Tracing the numbers as they go through the network provides the following table.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Note that the first column of modules exchanged the \(<6, 2>\) pair, the second column exchanged the \(<3, 2>\) pair, then the pair \(<8, 6>\) was exchanged, and finally the pair \(<6, 3>\) was exchanged.

Now that we know *how* it works, we need to explore *why* it works and what strategy motivated the design of the network. Understanding why generalizations of the above sorting network are correct is nontrivial.

Networks similar to that of figure 1 are derived from a strange mergesort algorithm developed by K. E. Batcher in the 1960's. Thus it is known as *Batcher's Odd-Even Mergesort* algorithm. The main sorting algorithm is exactly the same as the standard mergesort presented in figure 2.

The only difference is in the merge. In our first presentation of the merge, we shall assume that the two lists to be merged are both of the same size and that this size is a power of two. We shall need two auxiliary methods or algorithms in our new merge. One is a shuffle operation on lists exactly like that used with playing cards and the other is the compare exchange operation. The specifications for these appear as figure 3 along with the specifications for merging.

**Figure 3 - Specifications**

The strategy behind Batcher's odd-even merge is to merge the elements in the odd-numbered positions of lists a and b. That is, merge:

\[ a[1], a[3], \ldots, a[n-1] \text{ with } b[1], b[3], \ldots, b[n-1]. \]

Then merge the elements of a and b that occupy the even numbered positions. After this, the two recently merged lists are shuffled and adjacent items are checked for proper order. Here is an example:

\[ a = 5, 9, 14, 25 \quad b = 3, 12, 19, 32 \]
merge $a_o$ = the odd elements of $a$ with those of $b$

$a_o = 5, 14 \quad b_o = 3, 19 \quad c_o = 3, 5, 14, 19$

merge $a_e$ = the even elements of $a$ with those of $b$

$a_e = 9, 25 \quad b_e = 12, 32 \quad c_e = 9, 12, 25, 32$

shuffle $c_o$ with $c_e$

$c = 3, 9, 5, 12, 14, 25, 19, 32$

Note that at this point exchanging adjacent elements of $c$ provides us with a perfectly sorted list. The recursive algorithm appears in figure 4.

merge($a$, $b$, $c$, $n$)
if $n = 1$ then
  
    
  compex($c$, 1, 2)}
else

  $\{a_o = a[1], a[3], ..., a[n-1]; a_e = a[2], a[4], ..., a[n]$
    
  $b_o = b[1], b[3], ..., b[n-1]; b_e = b[2], b[4], ..., b[n]$
    
  merge($a_o$, $b_o$, $c_o$, $n/2)$
    
  merge($a_e$, $b_e$, $c_e$, $n/2)$
    
  shuffle($c_o$, $c_e$, $c$, $n/2)$
  
  for $i = 2$ to $n-2$ in steps of 2 do compex($c$, $i$, $i+1$)

Figure 4 - Batcher's Odd-Even Merge

The proof of correctness for this strange merge involves showing that each of the elements in $c_o$ and $c_e$ are larger than a certain number of elements of $c$. Since both $a[1]$ and $b[1]$ are in $c_o$, then we know that the first element of $c_o$ is the smallest on the list. For $c_o[i]$ and $c_e[i-1]$, an intricate counting argument shows that each is larger than $2i-3$ elements and thus should be in either position $2i-2$ or $2i-1$.

Deriving the complexity of this merge is not so difficult. We note that the shuffle and the compare-exchange operations take $O(n)$ operations and solve this recurrence relation for merging two lists of size $n$.

$$\text{Merge}(n, n) = 2*\text{Merge}(n/2, n/2) + n$$

Since $n$ is a power of two, we may rewrite and solve the equation as shown below.

$$\text{Merge}(2^k, 2^k) = 2*\text{Merge}(2^{k-1}, 2^{k-1}) + 2^k$$

$$= 2*(2*\text{Merge}(2^{k-2}, 2^{k-2}) + 2^{k-1}) + 2^k$$
\[
= 2^{2k-2} \text{Merge}(2^{k-2}, 2^{k-2}) + 2^k + 2^k \\
= 2^k + ... + 2^k \\
= k^2 \\
= n \log n
\]

This results in a complexity of \(O(n \log^2 n)\) for the mergesort algorithm using the odd-even merge. This is still much better than \(O(n^2)\).

Two important observations lead from the odd-even mergesort to sorting networks.

- **All of the rearranging of the list to be sorted is done by the compare-exchange operations.**

- **By keeping track of where the a, b, and c list elements are on the original list, the sort can be done inplace.**

So, if we note the order of the compare-exchange operations during a sort, we can easily construct a sorting network and even build a sorting circuit. The compare-exchange operations for a list of size four are:

- merge the first two elements: \texttt{compex(a, 1, 2)}
- merge the last two elements: \texttt{compex(a, 3, 4)}
- merge the odd elements: \texttt{compex(a, 1, 3)}
- merge the even elements: \texttt{compex(a, 2, 4)}
- shuffle and correct: \texttt{compex(a, 2, 3)}

This produces the sorting network of figure 1. Omitting the operations that involve element 4 provides the sorting network for lists of size three shown in figure 5. It is easy to verify that this is correct.

![Figure 5 - A Three-Element Sorting Network](http://www.cs.uky.edu/~lewis/essays/algorithms/sortnets/sort-net.html)

If we wish to perform the odd-even mergesort on lists which are not powers of two in length, we do exactly the same thing that we would do in order to mergesort such a list. Consider the operations that we would go through to sort a list of size five.
divide the list into (a[1],a[2],a[3]) and (a[4],a[5])

divide the first list into (a[1],a[2]) and (a[3])

merge a[1] and a[2]: \texttt{compex(a, 1, 2)}
\textit{now (a[1], a[2]) is sorted}
form odd lists (a[1]) and (a[3])

merge a[1] and a[3]: \texttt{compex(a, 1, 3)}
shuffle (a[1],a[3]) and (a[2]) to get (a[1],a[2],a[3])
correct (a[1],a[2],a[3]): \texttt{compex(a, 2, 3)}

\textit{now (a[1], a[2], a[3]) is sorted}

merge a[4] and a[5]: \texttt{compex(a, 4, 5)}

\textit{now (a[1], a[2], a[3]) and (a[4], a[5]) are sorted}
form odd lists (a[1],a[3]) and (a[4])

form odd lists (a[1]) and (a[4])

merge a[1] and a[4]: \texttt{compex(a, 1, 4)}
shuffle (a[1],a[4]) and (a[3]) to get (a[1], a[3], a[4])
correct (a[1],a[3],a[4]): \texttt{compex(a, 3, 4)}

\textit{now (a[1], a[3], a[4]) is sorted}
form even list (a[2], a[5])

merge a[2] and a[5]: \texttt{compex(a, 2, 5)}

\textit{now (a[1], a[3], a[4]) and (a[2], a[5]) are sorted}
shuffle (a[1],a[3],a[4]) and (a[2],a[5]) to get (a[1],a[2],a[3],a[5],a[4])
correct (a[1],a[2],a[3],a[4],a[5]): \texttt{compex(a, 2, 3); compex(a, 4, 5)}

Extracting the compare-exchange operations provides both a straight line algorithm and a sorting network for lists of five elements. The network is shown in figure 6.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{sort-net.png}
\caption{A Five-Element Sorting Network}
\end{figure}