Sorting Algorithms

- rearranging a list of numbers into increasing (or decreasing) order.

Potential Speedup

The worst-case time complexity of mergesort and the average time complexity of quicksort are both $O(n \log n)$, where there are $n$ numbers.

$O(n \log n)$ is, in fact, optimal for any sequential sorting algorithm without using any special properties of the numbers.

Hence, the best parallel time complexity we can expect based upon a sequential sorting algorithm but using $n$ processors is

$$\text{Optimal parallel time complexity} = \frac{O(n \log n)}{n} = O(\log n)$$

A $O(\log n)$ sorting algorithm with $n$ processors has been demonstrated by Leighton (1984) based upon an algorithm by Ajtai, Komlós, and Szemerédi (1983), but the constant hidden in the order notation is extremely large.

An $O(\log n)$ sorting algorithm is also described by Leighton (1994) for an $n$-processor hypercube using random operations.

Akl (1985) describes 20 different parallel sorting algorithms, several of which achieve the lower bound for a particular interconnection network.

But, in general, a realistic $O(\log n)$ algorithm with $n$ processors is a goal that will not be easy to achieve. It may be that the number of processors will be greater than $n$. 
Rank Sort

The number of numbers that are smaller than each selected number is counted.

This count provides the position of the selected number in the list; that is, its “rank” in the list.

Suppose there are \( n \) numbers stored in an array, \( a[0] \ldots a[n-1] \).

First \( a[0] \) is read and compared with each of the other numbers, \( a[1] \ldots a[n-1] \), recording the number of numbers less than \( a[0] \).

Suppose this number is \( x \). This is the index of the location in the final sorted list.

The number \( a[0] \) is copied into the final sorted list \( b[0] \ldots b[n-1] \), at location \( b[x] \).

Actions repeated with the other numbers.

Overall sequential sorting time complexity of \( O(n^2) \) (not exactly a good sequential sorting algorithm!).

Sequential Code

```c
for (i = 0; i < n; i++) { /* for each number */
    x = 0;
    for (j = 0; j < n; j++) /* count number of nos less than it */
        if (a[i] > a[j]) x++;
    b[x] = a[i]; /* copy number into correct place */
}
```

(This code will fail if duplicates exist in the sequence of numbers.)
Parallel Code

Using $n$ Processors

One processor allocated to one of the numbers

Processor finds the final index of one numbers in $O(n)$ steps.

With all processors operating in parallel, the parallel time complexity $O(n)$.

In `forall` notation, the code would look like

```c
forall (i = 0; i < n; i++) { /* for each number in parallel*/
    x = 0;
    for (j = 0; j < n; j++) /* count number of nos less than it */
        if (a[i] > a[j]) x++;
    b[x] = a[i]; /* copy number into correct place */
}
```

The parallel time complexity, $O(n)$, is better than any sequential sorting algorithm.

We can do even better if we have more processors.
Using $n^2$ Processors

Comparing one selected number with each of the other numbers in the list can be performed using multiple processors:

$n - 1$ processors are used to find the rank of one number

With $n$ numbers, $(n - 1)n$ processors or (almost) $n^2$ processors needed.

A single counter is needed for each number.

Incrementing the counter is done sequentially and requires a maximum of $n$ steps.

Total number of steps would be given by $1 + n$. 

Figure 9.1 Finding the rank in parallel.
Reduction in Number of Steps

A tree structure could be used to reduce the number of steps involved in incrementing the counter:

Leads to an $O(\log n)$ algorithm with $n^2$ processors for sorting numbers.

The actual processor efficiency of this method is relatively low.
Parallel Rank Sort Conclusions

Rank sort can sort:

\( \text{in } O(n) \) with \( n \) processors

or

\( \text{in } O(\log n) \) using \( n^2 \) processors.

In practical applications, using \( n^2 \) processors will be prohibitive.

Theoretically possible to reduce the time complexity to \( O(1) \) by considering all the increment operations as happening in parallel since they are independent of each other.

\( O(1) \) is, of course, the lower bound for any problem.
Message Passing Parallel Rank Sort

Master-Slave Approach

Requires shared access to the list of numbers.

Master process responds to request for numbers from slaves.

Algorithm better for shared memory

Figure 9.3  Rank sort using a master and slaves.
Compare-and-Exchange Sorting Algorithms

Compare and Exchange

Form the basis of several, if not most, classical sequential sorting algorithms.

Two numbers, say $A$ and $B$, are compared.

If $A > B$, $A$ and $B$ are exchanged, i.e.:

```c
if (A > B) {
    temp = A;
    A = B;
    B = temp;
}
```
Message-Passing Compare and Exchange

One simple way of implementing the compare and exchange is for $P_1$ to send $A$ to $P_2$, which then compares $A$ and $B$ and sends back $B$ to $P_1$ if $A$ is larger than $B$ (otherwise it sends back $A$ to $P_1$):

![Diagram of compare and exchange process]

**Figure 9.4** Compare and exchange on a message-passing system — Version 1.

**Code:**

Process $P_1$

```c
send(&A, P_2);
recv(&A, P_2);
```

Process $P_2$

```c
recv(&A, P_1);
if (A > B) {
    send(&B, P_1);
    B = A;
} else
    send(&A, P_1);
```
Alternative Message Passing Method

For $P_1$ to send $A$ to $P_2$ and $P_2$ to send $B$ to $P_1$.

Then both processes perform compare operations. $P_1$ keeps the larger of $A$ and $B$ and $P_2$ keeps the smaller of $A$ and $B$:

![Diagram of message passing](image)

**Figure 9.5** Compare and exchange on a message-passing system — Version 2.

Code:

Process $P_1$

```c
send(&A, P_2);
recv(&B, P_2);
if (A > B) A = B;
```

Process $P_2$

```c
recv(&A, P_1);
send(&B, P_1);
if (A > B) B = A;
```

Process $P_1$ performs the `send()` first and process $P_2$ performs the `recv()` first to avoid deadlock.

Alternatively, both $P_1$ and $P_2$ could perform `send()` first if locally blocking (asynchronous) sends are used and sufficient buffering is guaranteed to exist - not safe message passing.
Note on Precision of Duplicated Computations

Previous code assumes that the if condition, $A > B$, will return the same Boolean answer in both processors.

Different processors operating at different precision could conceivably produce different answers if real numbers are being compared.

This situation applies to anywhere computations are duplicated in different processors to reduce message passing, or to make the code SPMD.
Data Partitioning

Suppose there are \( p \) processors and \( n \) numbers.

A list of \( n/p \) numbers would be assigned to each processor:

![Diagram showing data partitioning](image-url)

**Figure 9.6** Merging two sublists — Version 1.
Figure 9.7 Merging two sublists — Version 2.
Bubble Sort

The largest number is first moved to the very end of the list by a series of compares and exchanges, starting at the opposite end. The actions are repeated for each number.

The larger numbers move ("bubble") toward one end:

![Figure 9.8 Steps in bubble sort.](image-url)

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1999
Sequential Code

With numbers held in array $a[]$:

```c
for (i = n - 1; i > 0; i--)
    for (j = 0; j < i; j++) {
        k = j + 1;
        if (a[j] > a[k]) {
            temp = a[j];
            a[j] = a[k];
            a[k] = temp;
        }
    }
```

Time Complexity

Number of compare and exchange operations

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

which indicates a time complexity of $O(n^2)$ given that a single compare-and-exchange operation has a constant complexity, $O(1)$. 
Parallel Bubble Sort

The “bubbling” action of one iteration could start before the previous iteration has finished so long as it does not overtake the previous bubbling action:

Figure 9.9 Overlapping bubble sort actions in a pipeline.
Odd-Even (Transposition) Sort

Variation of bubble sort.

Operates in two alternating phases, an *even* phase and an *odd* phase.

**Even phase**

Even-numbered processes exchange numbers with their right neighbor.

**Odd phase**

Odd-numbered processes exchange numbers with their right neighbor.

![Figure 9.10](image-url) Odd-even transposition sort sorting eight numbers.
Odd-Even Transposition Sort Code

Even Phase

\[ P_i, \ i = 0, 2, 4, \ldots, n-2 \text{ (even)} \quad P_i, \ i = 1, 3, 5, \ldots, n-1 \text{ (odd)} \]

\[
\text{recv}(\&A, \ P_{i+1}); \\
\text{send}(\&B, \ P_{i+1}); \\
\text{if} \ (A > B) \ B = A;
\]

\[
\text{recv}(\&A, \ P_{i-1}); \\
\text{send}(\&B, \ P_{i-1}); \\
\text{if} \ (A > B) \ A = B; /* \text{even phase} */
\]

where the number stored in \( P_{i\text{(even)}} \) is \( B \) and the number stored in \( P_{i\text{(odd)}} \) is \( A \).

Odd Phase

\[ P_i, \ i = 1, 3, 5, \ldots, n-3 \text{ (odd)} \quad P_i, \ i = 2, 4, 6, \ldots, n-2 \text{ (even)} \]

\[
\text{send}(\&A, \ P_{i+1}); \\
\text{recv}(\&B, \ P_{i+1}); \\
\text{if} \ (A > B) \ A = B;
\]

\[
\text{recv}(\&A, \ P_{i-1}); \\
\text{send}(\&B, \ P_{i-1}); \\
\text{if} \ (A > B) \ B = A; /* \text{odd phase} */
\]

Combined

\[ P_i, \ i = 1, 3, 5, \ldots, n-3 \text{ (odd)} \quad P_i, \ i = 0, 2, 4, \ldots, n-2 \text{ (even)} \]

\[
\text{send}(\&A, \ P_{i-1}); \\
\text{recv}(\&B, \ P_{i-1}); \\
\text{if} \ (A > B) \ A = B; \\
\text{if} \ (i < n-3) \{ \\
\text{send}(\&A, \ P_{i+1}); \\
\text{recv}(\&B, \ P_{i+1}); \\
\text{if} \ (A > B) \ B = A;
\}
\]

\[
\text{recv}(\&A, \ P_{i+1}); \\
\text{send}(\&B, \ P_{i+1}); /* \text{even phase} */ \\
\text{send}(\&B, \ P_{i+1}); \\
\text{recv}(\&A, \ P_{i-1}); /* \text{odd phase} */ \\
\text{send}(\&B, \ P_{i-1}); \\
\text{recv}(\&A, \ P_{i-1}); \\
\text{if} \ (A > B) \ B = A; \\
\text{if} \ (A > B) \ A = B;
\}
\]
Two-Dimensional Sorting

If the numbers are mapped onto a mesh, other distinct possibilities exist for sorting the numbers.

The layout of a sorted sequence on a mesh could be row by row or snakelike. In a snakelike sorted list, the numbers are arranged in nondecreasing order:

![Snakelike sorted list diagram](image)

**Figure 9.11** Snakelike sorted list.
Shearsort

Requires $\sqrt{n}(\log n + 1)$ steps for $n$ numbers on a $\sqrt{n} \times \sqrt{n}$ mesh.

Odd phase

Each row of numbers is sorted independently, in alternative directions:

- Even rows — The smallest number of each column is placed at the rightmost end and largest number at the leftmost end.
- Odd rows — The smallest number of each column is placed at the leftmost end and the largest number at the rightmost end.

Even phase

Each column of numbers is sorted independently, placing the smallest number of each column at the top and the largest number at the bottom.

After $\log n + 1$ phases, the numbers are sorted with a snakelike placement in the mesh.

Note the alternating directions of the row sorting phase, which matches the final snakelike layout.

(a) Original placement of numbers
(b) Phase 1 — Row sort
(c) Phase 2 — Column sort
(d) Phase 3 — Row sort
(e) Phase 4 — Column sort
(f) Final phase — Row sort

Figure 9.12 Shearsort.
Using Transposition

For algorithms that alternate between acting within rows and acting within columns, we can be limited to rows by transposing the array of data points between each phase.

A transpose operation causes the elements in each column to be in positions in a row.

The transpose operation is placed between the row operations and column operations:

![Diagram](image)

(a) Operations between elements in rows  
(b) Transpose operation  
(c) Operations between elements in rows (originally columns)

**Figure 9.13** Using the transpose operation to maintain operations in rows.

The transposition can be achieved with \(\sqrt{n} (\sqrt{n} - 1)\) communications or \(O(n)\) communications.

A single *all-to-all* routine could be reduce this.
Mergesort

The unsorted list is first divided into half. Each half is again divided into two. This is continued until individual numbers are obtained.

Then pairs of numbers are combined (merged) into sorted list of two numbers. Pairs of these lists of four numbers are merged into sorted lists of eight numbers. This is continued until the one fully sorted list is obtained.

**Figure 9.14** Mergesort using tree allocation of processes.
Analysis

Sequential time complexity is $O(n \log n)$.

There are $2 \log n$ steps in the parallel version but each step may need to perform more than one basic operation, depending upon the number of numbers being processed.

Communication

In the division phase, communication only takes place as follows:

Communication at each step | Processor communication
--- | ---
$t_{\text{startup}} + (n/2)t_{\text{data}}$ | $P_0 \rightarrow P_4$
$t_{\text{startup}} + (n/4)t_{\text{data}}$ | $P_0 \rightarrow P_2; P_4 \rightarrow P_6$
$t_{\text{startup}} + (n/8)t_{\text{data}}$ | $P_0 \rightarrow P_1; P_2 \rightarrow P_3; P_4 \rightarrow P_5; P_6 \rightarrow P_7$

with $\log p$ steps, given $p$ processors. In the merge phase, the reverse communications take place:

$t_{\text{startup}} + (n/8)t_{\text{data}}$ | $P_0 \rightarrow P_1; P_2 \rightarrow P_3; P_4 \rightarrow P_5; P_6 \rightarrow P_7$
$t_{\text{startup}} + (n/4)t_{\text{data}}$ | $P_0 \rightarrow P_2; P_4 \rightarrow P_6$
$t_{\text{startup}} + (n/2)t_{\text{data}}$ | $P_0 \rightarrow P_4$

again $\log p$ steps. This leads to the communication time being

$$ t_{\text{comm}} = 2(t_{\text{startup}} + (n/2)t_{\text{data}} + t_{\text{startup}} + (n/4)t_{\text{data}} + t_{\text{startup}} + (n/8)t_{\text{data}} + \ldots) $$

or:

$$ t_{\text{comm}} \approx 2(\log p)t_{\text{startup}} + 2nt_{\text{data}} $$
Computation

Computations only occurs in merging the sublists.

Merging can be done by stepping through each list, moving the smallest found into the final list first. It takes $2n - 1$ steps in the worst case to merge two sorted lists each of $n$ numbers into one sorted list in this manner.

Therefore, the computation consists of

\[
\begin{align*}
    t_{\text{comp}} &= 1 & P_0; P_2; P_4; P_6 \\
    t_{\text{comp}} &= 3 & P_0; P_2 \\
    t_{\text{comp}} &= 7 & P_0 \\
\end{align*}
\]

Hence:

\[
\begin{align*}
    t_{\text{comp}} &= \log p \\
    &= \sum_{i=1}^{\log p} (2^i - 1) \\
\end{align*}
\]

The parallel computational time complexity is $O(p)$ using $p$ processors and one number in each processor.

As with all sorting algorithms, normally we would partition the list into groups, one group of numbers for each processor.
Quicksort

Sequential time complexity of $O(n \log n)$. The question to answer is whether a parallel version can achieve the time complexity of $O(\log n)$ with $n$ processors.

Quicksort sorts a list of numbers by first dividing the list into two sublists, as in mergesort.

All the numbers in one sublist are arranged to be smaller than all the numbers in the other sublist.

Achieved by first selecting one number, called a pivot, against which every other number is compared.

If the number is less than the pivot, it is placed in one sublist. Otherwise, it is placed in the other sublist.

By repeating the procedure sufficiently, we are left with sublists of one number each. With proper ordering of the sublists, a sorted list is obtained.

**Sequential Code**

Suppose an array `list[]` holds the list of numbers and `pivot` is the index in the array of the final position of the pivot:

```c
quicksort(list, start, end)
{
    if (start < end) {
        partition(list, start, end, pivot)
        quicksort(list, start, pivot-1); /* recursively call on sublists*/
        quicksort(list, pivot+1, end);
    }
}
```

*Partition()* moves numbers in the list between `start` to `end` so that those less than the pivot are before the pivot and those equal or greater than the pivot are after the pivot.

The pivot is in its final position of the sorted list.
Parallelizing Quicksort

**Figure 9.15** Quicksort using tree allocation of processes.
With the pivot being withheld

Figure 9.16  Quicksort showing pivot withheld in processes.
Analysis

Fundamental problem with all of these tree constructions – initial division is done by a single processor, which will seriously limit the speed.

Suppose the pivot selection is ideal and each division creates two sublists of equal size.

Computation

First, one processor operates upon \( n \) numbers. Then two processors each operate upon \( n/2 \) numbers. Then four processors each operate upon \( n/4 \) numbers, and so on:

\[
t_{\text{comp}} = n + n/2 + n/4 + n/8 + \ldots \approx 2n
\]

Communication

Communication also occurs in a similar fashion as for mergesort:

\[
t_{\text{comm}} = (t_{\text{startup}} + (n/2)t_{\text{data}}) + (t_{\text{startup}} + (n/4)t_{\text{data}}) + (t_{\text{startup}} + (n/8)t_{\text{data}}) + \ldots \\
\approx (\log p)t_{\text{startup}} + nt_{\text{data}}
\]

The major difference between quicksort and mergesort is that the tree in quicksort will not, in general, be perfectly balanced

The selection of the pivot is very important to make quicksort operate fast.
Work Pool Implementation

First, the work pool holds the initial unsorted list, which is given to the first processor. This processor divides the list into two parts. One part is returned to the work pool to be given to another processor, while the other part is operated upon again.

Figure 9.17 Work pool implementation of quicksort.
Quicksort on a Hypercube

Complete List Placed in One Processor

The list can be divided into two parts by using a pivot determined by the processor, with one part sent to the adjacent node in the highest dimension.

Then the two nodes can repeat the process, dividing their lists into two parts using locally selected pivots. One part is sent to a node in the next highest dimension.

This process is continued for \( \log d \) steps for a \( d \)-dimensional hypercube.

<table>
<thead>
<tr>
<th>Node</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st step: 000 → 001 (numbers greater than a pivot, say ( p_1 ))</td>
<td></td>
</tr>
<tr>
<td>2nd step: 000 → 010 (numbers greater than a pivot, say ( p_2 ))</td>
<td></td>
</tr>
<tr>
<td>001 → 011 (numbers greater than a pivot, say ( p_3 ))</td>
<td></td>
</tr>
<tr>
<td>3rd step: 000 → 100 (numbers greater than a pivot, say ( p_4 ))</td>
<td></td>
</tr>
<tr>
<td>001 → 101 (numbers greater than a pivot, say ( p_5 ))</td>
<td></td>
</tr>
<tr>
<td>010 → 110 (numbers greater than a pivot, say ( p_6 ))</td>
<td></td>
</tr>
<tr>
<td>011 → 111 (numbers greater than a pivot, say ( p_7 ))</td>
<td></td>
</tr>
</tbody>
</table>
Figure 9.18 Hypercube quicksort algorithm when the numbers are originally in node 000.
Numbers Initially Distributed across All Processors

Suppose the unsorted numbers are initially distributed across the nodes in an equitable fashion but not in any special order.

A $2^d$-node hypercube ($d$-dimensional hypercube) is composed of two smaller $2^{d-1}$-node hypercubes, which are interconnected with links between pairs of nodes in each cube in the $d$th dimension. This feature can be used in a direct extension of the quicksort algorithm to a hypercube as follows.

Steps

1. One processor (say $P_0$) selects (or computes) a suitable pivot and broadcasts this to all others in the cube.

2. The processors in the “lower” subcube send their numbers, which are greater than the pivot, to their partner processor in the “upper” subcube. The processors in the “upper” subcube send their numbers, which are equal to or less than the pivot, to their partner processor in the “lower” cube.

3. Each processor concatenates the list received with what remains of its own list.

Given a $d$-dimensional hypercube, after these steps the numbers in the lower $(d-1)$-dimensional subcube will all be equal to or less than the pivot and all the numbers in the upper $(d-1)$-dimensional hypercube will be greater than the pivot.

Steps 2 and 3 are now repeated recursively on the two $(d-1)$-dimensional subcubes.

One process in each subcube computes a pivot for its subcube and broadcasts it throughout its subcube.

These actions terminate after $\log d$ recursive phases. Suppose the hypercube has three dimensions.

Now the numbers in the processor 000 will be smaller than the numbers in processor 001, which will be smaller than the numbers in processor 010, and so on.
Figure 9.19  Hypercube quicksort algorithm when numbers are distributed among nodes.
Communication Patterns in Hypercube

(a) Phase 1 communication

(b) Phase 2 communication

(c) Phase 3 communication

Figure 9.20 Hypercube quicksort communication.
Pivot Selection

A poor pivot selection could result in most of the numbers being allocated to a small part of the hypercube, leaving the rest idle. This is most deleterious in the first split.

In the sequential quicksort algorithm, often the pivot is simply chosen to be the first number in the list, which could be obtained in a single step or with $O(1)$ time complexity.

One approach – take a sample of a numbers from the list, compute the mean value, and select the median as the pivot. The numbers sampled would need to be sorted at least halfway through to find the median.

We might choose a simple bubble sort, which can be terminated when the median is reached.
Hyperquicksort

Sorts the numbers at each stage to maintain sorted numbers in each processor.

Not only does this simplify selecting the pivots, it eliminates the final sorting operation.

Steps

1. Each processor sorts its list sequentially.

2. One processor (say $P_0$) selects (or computes) a suitable pivot and broadcasts this pivot to all others in the cube.

3. The processors in the “lower” subcube send their numbers, which are greater than the pivot, to their partner processor in the “upper” subcube. The processors in the “upper” subcube send their numbers, which are equal to or less than the pivot, to their partner processor in the “lower” cube.

4. Each processor merges the list received with its own to obtain a sorted list.

Steps 2, 3, and 4 are repeated ($d$ phases in all for a $d$-dimensional hypercube).
Figure 9.21 Quicksort hypercube algorithm with Gray code ordering.
Analysis

Suppose $n$ numbers and $p$ processors in a $d$-dimensional hypercube so that $2^d = p$. Initially, the numbers are distributed so that each processor has $n/p$ numbers. Afterward, the number of numbers at each processor will vary. Let the number be $x$. (Ideally of course, $x = n/p$ throughout.)

The algorithm calls for $d$ phases. After the initial sorting step requiring $O(n/p \log n/p)$, each phase has pivot selection, pivot broadcast, a data split, data communication, and data merge.

Computation — Pivot Selection

With a sorted list, pivot selection can be done in one step, $O(1)$, if there always were $n/p$ numbers. In the more general case, the time complexity will be higher.

Communication — Pivot Broadcast

\[
\frac{d(d-1)}{2} (t_{\text{startup}} + t_{\text{data}})
\]

Computation — Data Split

If the numbers are sorted and there are $x$ numbers, the split operation can be done in $\log x$ steps.

Communication — Data from Split

\[
t_{\text{startup}} + \frac{x}{2} t_{\text{data}}
\]

Computation — Data Merge

To merge two sorted lists into one sorted list requires $x$ steps if the biggest list has $x$ numbers.

Total

The total time is given by the sum of the individual communication times and computation times. Pivot broadcast is the most expensive part of the algorithm.
Odd-Even Mergesort

Based upon odd-even merge algorithm.

Odd-Even Merge Algorithm

Will merge two sorted lists into one sorted list, and this is used recursively to build up larger and larger sorted lists.

Given two sorted lists \(a_1, a_2, a_3, \ldots, a_n\) and \(b_1, b_2, b_3, \ldots, b_n\) (where \(n\) is a power of 2), the following actions are performed:

1. The elements with odd indices of each sequence — that is, \(a_1, a_3, a_5, \ldots, a_{n-1}\), and \(b_1, b_3, b_5, \ldots, b_{n-1}\) — are merged into one sorted list, \(c_1, c_2, c_3, \ldots, c_n\).

2. The elements with even indices of each sequence — that is, \(a_2, a_4, a_6, \ldots, a_n\), and \(b_2, b_4, b_6, \ldots, b_n\) — are merged into one sorted list, \(d_1, d_2, \ldots, d_n\).

3. The final sorted list, \(e_1, e_2, \ldots, e_{2n}\), is obtained by the following:

\[
\begin{align*}
  e_{2i} &= \min\{c_{i+1}, d_i\} \\
  e_{2i+1} &= \max\{c_{i+1}, d_i\}
\end{align*}
\]

for \(1 \leq i \leq n-1\). Essentially the odd and even index lists are interleaved, and pairs of odd/even elements are interchanged to move the larger toward one end, if necessary.

The first number is given by \(e_1 = c_1\) (since this will be the smallest of first elements of each list, \(a_1\) or \(b_1\)) and the last number by \(e_{2n} = d_n\) (since this will be the largest of last elements of each list, \(a_n\) or \(b_n\)).
Figure 9.22  Odd-even merging of two sorted lists.
Figure 9.23 Odd-even mergesort.
Bitonic Mergesort

The basis of bitonic mergesort is the *bitonic sequence*.

**Bitonic Sequence**

A monotonic increasing sequence is a sequence of increasing numbers.

A *bitonic sequence* has two sequences, one increasing and one decreasing.

Formally, a bitonic sequence is a sequence of numbers, \(a_0, a_1, a_2, a_3, \ldots, a_{n-2}, a_{n-1}\), which monotonically increases in value, reaches a single maximum, and then monotonically decreases in value; e.g.,

\[
a_0 < a_1 < a_2, a_3, \ldots, a_{i-1} < a_i > a_{i+1}, \ldots, a_{n-2} > a_{n-1}
\]

for some value of \(i\) (\(0 \leq i < n\)).

A sequence is also bitonic if the preceding can be achieved by shifting the numbers cyclically (left or right).

\[\text{Value} \quad a_0, a_1, a_2, a_3, \ldots, a_{n-2}, a_{n-1}\]

\[a_0, a_1, a_2, a_3, \ldots, a_{n-2}, a_{n-1}\]

(a) Single maximum (b) Single maximum and single minimum

**Figure 9.24** Bitonic sequences.
“Special” Characteristic of Bitonic Sequences

If we perform a compare-and-exchange operation on \( a_i \) with \( a_{i+n/2} \) for all \( i \) (\( 0 \leq i < n/2 \)), where there are \( n \) numbers in the sequence, we get two bitonic sequences, where the numbers in one sequence are all less than the numbers in the other sequence.

Example

Starting with the bitonic sequence

\[
3, 5, 8, 9, 7, 4, 2, 1
\]

we get the sequences shown below

![Figure 9.25 Creating two bitonic sequences from one bitonic sequence.](image)
The compare-and-exchange operation moves the smaller numbers of each pair to the left sequence and the larger numbers of the pair to the right sequence.

Given a bitonic sequence, recursively performing compare-and-exchange operations to subsequences will sort the list.

Eventually, we obtain bitonic sequences consisting of one number each and a fully sorted list.

![Sorting a bitonic sequence.](Figure 9.26)
**Sorting**

To sort an unordered sequence, sequences are merged into larger bitonic sequences, starting with pairs of adjacent numbers.

By a compare-and-exchange operation, pairs of adjacent numbers are formed into increasing sequences and decreasing sequences, pairs of which form a bitonic sequence of twice the size of each of the original sequences.

By repeating this process, bitonic sequences of larger and larger lengths are obtained.

In the final step, a single bitonic sequence is sorted into a single increasing sequence.

*Figure 9.27*  Bitonic mergesort.
Bitonic Mergesort Example

Sorting eight numbers. The basic compare-and-exchange operation is given by a box, with an arrow indicating which output is the larger number of the operation:

**Figure 9.28** Bitonic mergesort on eight numbers.
Phases

The six steps (for eight numbers) are divided into three phases:

Phase 1 (Step 1) Convert pairs of numbers into increasing/decreasing sequences and hence into 4-bit bitonic sequences.

Phase 2 (Steps 2/3) Split each 4-bit bitonic sequence into two 2-bit bitonic sequences, higher sequences at center.

Sort each 4-bit bitonic sequence increasing/decreasing sequences and merge into 8-bit bitonic sequence.

Phase 3 (Steps 4/5/6) Sort 8-bit bitonic sequence (as in Figure 9.27).

Number of Steps

In general, with \( n = 2^k \), there are \( k \) phases, each of 1, 2, 3, …, \( k \) steps. Hence the total number of steps is given by

\[
\text{Steps} = \sum_{i=1}^{k} i = \frac{k(k + 1)}{2} = \frac{\log n (\log n + 1)}{2} = O(\log^2 n)
\]