Knuth's 0-1-Principle and Beyond

Janis Voigtländer

University of Bonn, Germany

(visiting NII Tokyo)

February 26th, 2010

The Sorting Problem

Task: Given a list and an order on the type of elements of this list, produce a sorted list (with same content)!

Example:

12	7	9	8	4	6

$\mapsto$

4	6	7	8	9	12

The Sorting Problem

Task: Given a list and an order on the type of elements of this list, produce a sorted list (with same content)!

Example:

$$\boxed{12 \mid 7 \mid 9 \mid 8 \mid 4 \mid 6} \mapsto \boxed{4 \mid 6 \mid 7 \mid 8 \mid 9 \mid 12}$$

Many Solutions:

- Quicksort
- Insertion Sort
- Merge Sort
- Bubble Sort
- ...

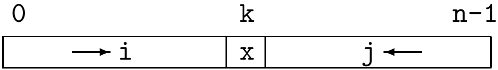
Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

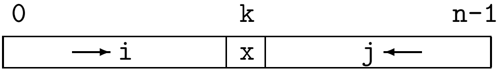
Realisation:



Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



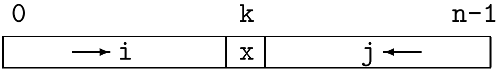
Example:

2	15	7	9	12	4	11

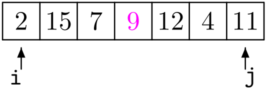
Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



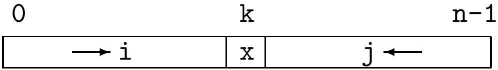
Example:



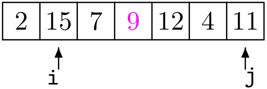
Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



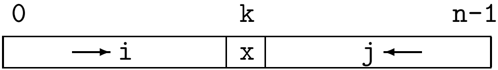
Example:



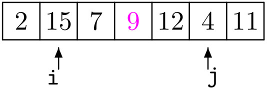
Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



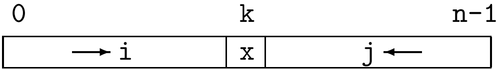
Example:



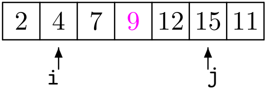
Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



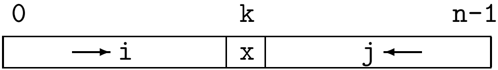
Example:



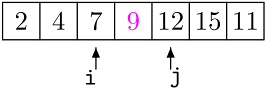
Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



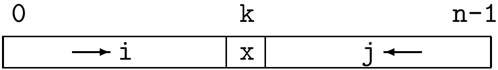
Example:



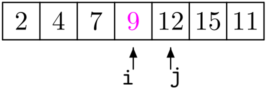
Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



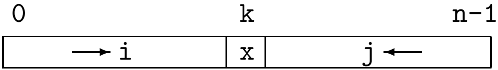
Example:



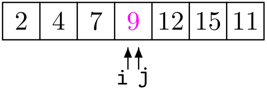
Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



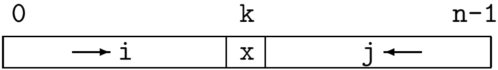
Example:



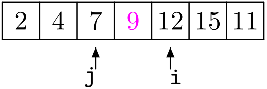
Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



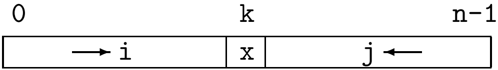
Example:



Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



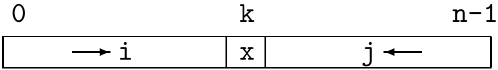
Example:

2	4	7	9	12	15	11

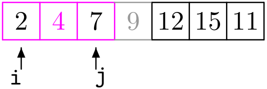
Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



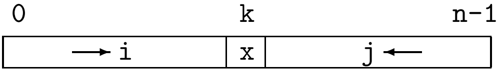
Example:



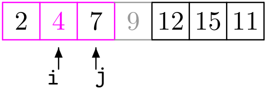
Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



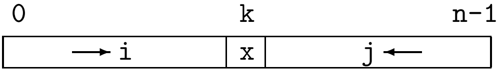
Example:



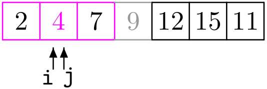
Quicksort

1. Choose an element x from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than x, and
   - one containing all elements greater or equal to x.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element x, and
   - the sorted second sublist.

Realisation:



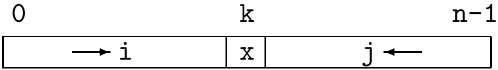
Example:



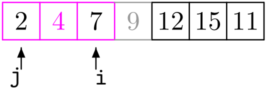
Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



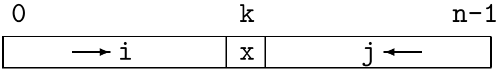
Example:



Quicksort

1. Choose an element `x` from the input list.
2. Partition the remaining elements into two sublists:
   - one containing all elements smaller than `x`, and
   - one containing all elements greater or equal to `x`.
3. Sort the two sublists recursively.
4. The ouput list is the concatenation of:
   - the sorted first sublist,
   - the element `x`, and
   - the sorted second sublist.

Realisation:



Example:

2	4	7	9	12	15	11

Alternatives

Note:

- The Quicksort algorithm uses the following as key operation (to drive the partitioning):

$$\textit{compare} :: (\tau, \tau) \rightarrow \mathsf{Bool}$$

Alternatives

Note:

- The Quicksort algorithm uses the following as key operation (to drive the partitioning):

$$\textit{compare} :: (\tau, \tau) \rightarrow \text{Bool}$$

- The same is true for algorithms like Insertion Sort, Merge Sort, ...

Alternatives

Note:

- The Quicksort algorithm uses the following as key operation (to drive the partitioning):

$$\mathit{compare} :: (\tau, \tau) \to \mathsf{Bool}$$

- The same is true for algorithms like Insertion Sort, Merge Sort, ...

But: Knuth also considered a more restricted class of sorting algorithms, based instead on the following operation:

$$\mathit{cswap} :: (\tau, \tau) \to (\tau, \tau)$$

Alternatives

Note:

- The Quicksort algorithm uses the following as key operation (to drive the partitioning):

$$\mathit{compare} :: (\tau, \tau) \rightarrow \mathsf{Bool}$$

- The same is true for algorithms like Insertion Sort, Merge Sort, ...

But: Knuth also considered a more restricted class of sorting algorithms, based instead on the following operation:

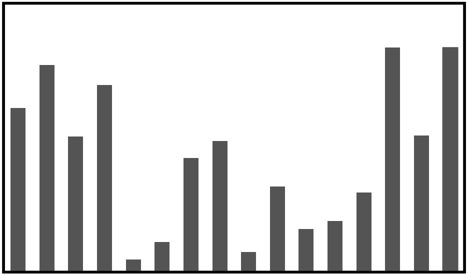
$$\mathit{cswap} :: (\tau, \tau) \rightarrow (\tau, \tau)$$

Bitonic Sort

1. Split the input list into two sublists of equal length.

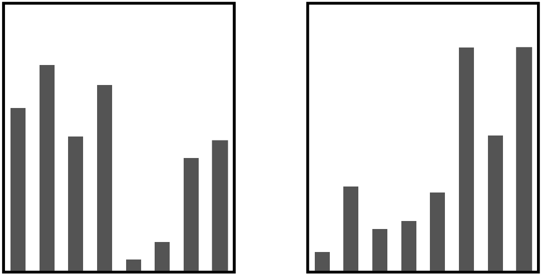
Bitonic Sort

1. Split the input list into two sublists of equal length.



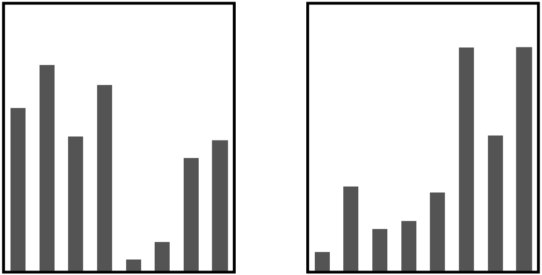
Bitonic Sort

1. Split the input list into two sublists of equal length.



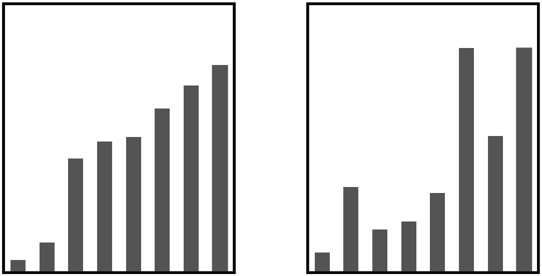
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively



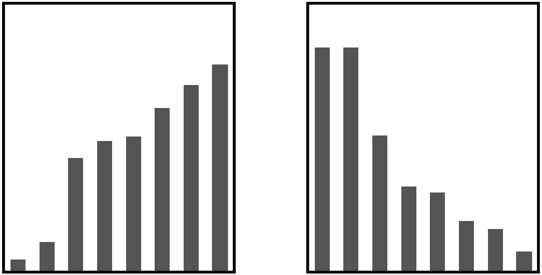
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively



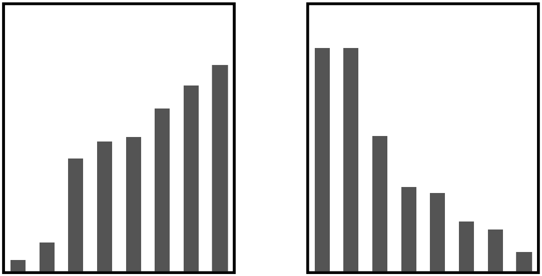
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.



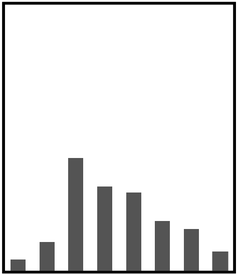
Bitonic Sort

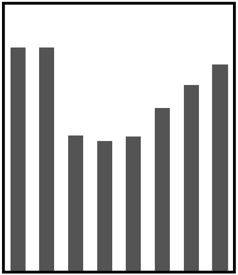
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.



Bitonic Sort

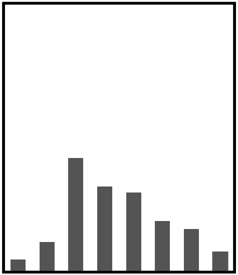
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.

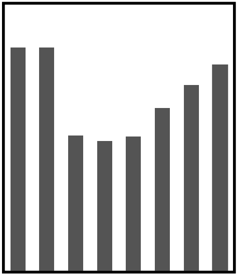




Bitonic Sort

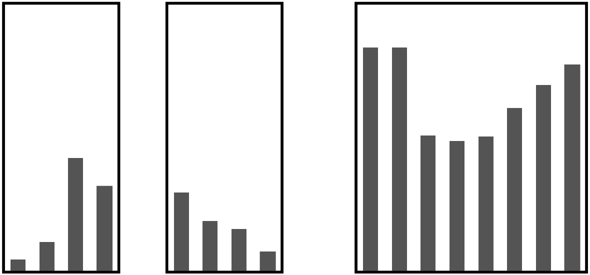
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
    3.1 Apply *cswap* to pairs of elements at corresponding positions.
    3.2 Split each of the resulting lists in the middle.





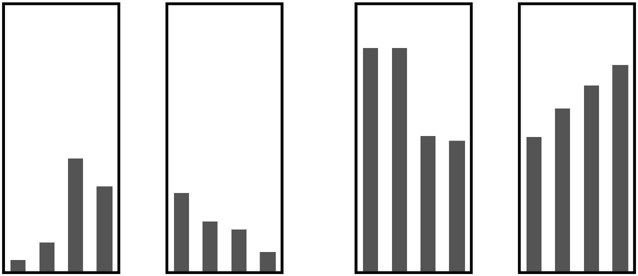
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.



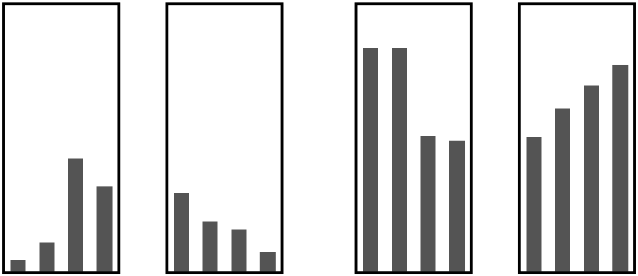
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.



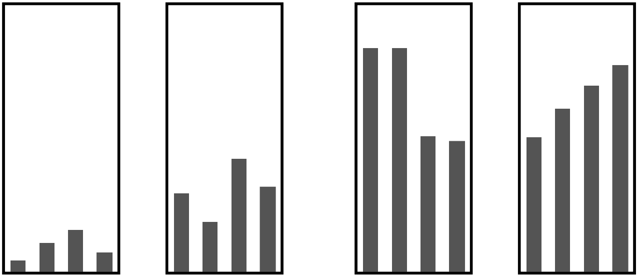
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



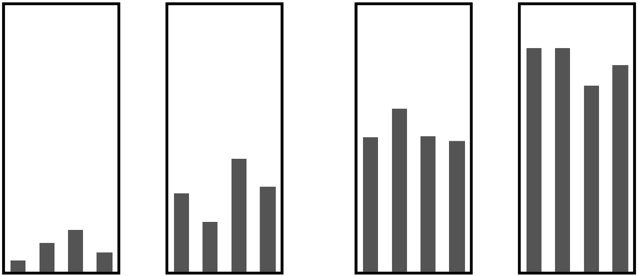
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



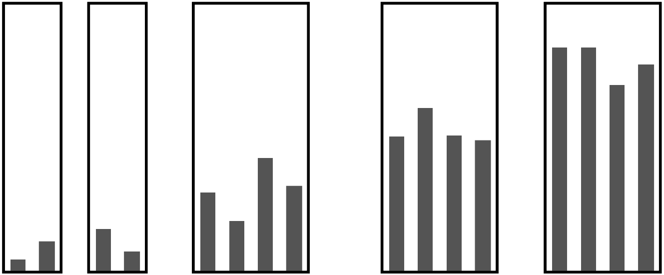
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



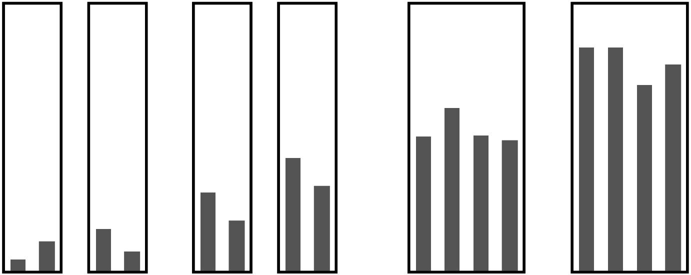
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



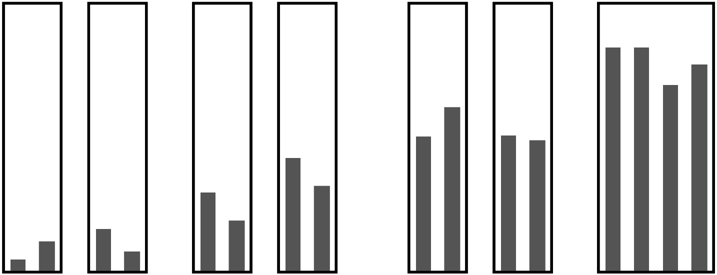
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



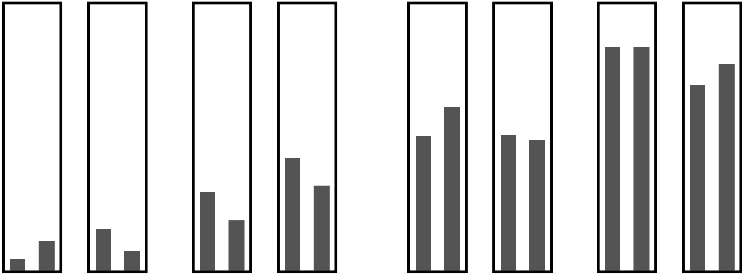
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



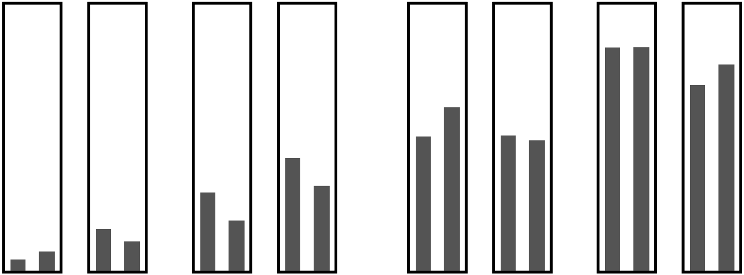
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



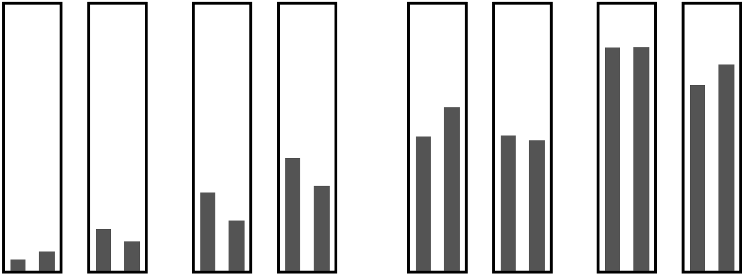
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



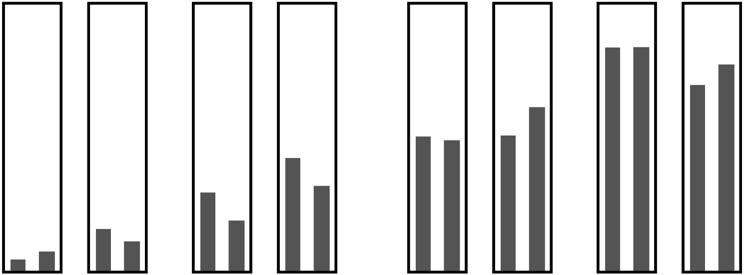
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
    3.1 Apply *cswap* to pairs of elements at corresponding positions.
    3.2 Split each of the resulting lists in the middle.
    3.3 Merge the resulting pairs of lists recursively.



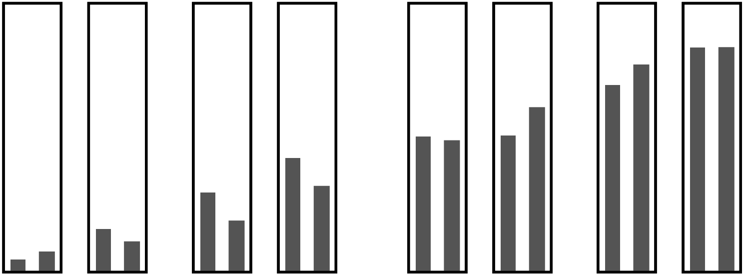
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



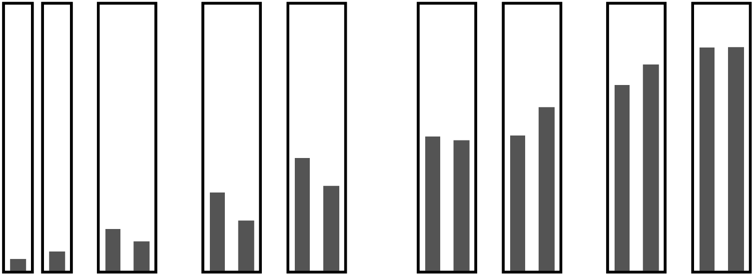
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
    3.1 Apply *cswap* to pairs of elements at corresponding positions.
    3.2 Split each of the resulting lists in the middle.
    3.3 Merge the resulting pairs of lists recursively.



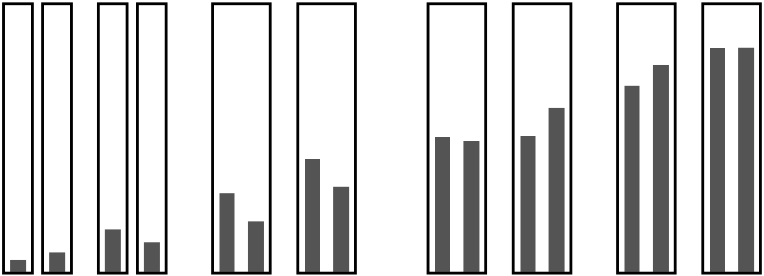
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



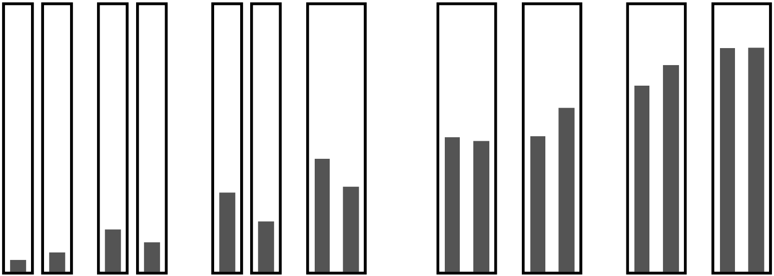
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



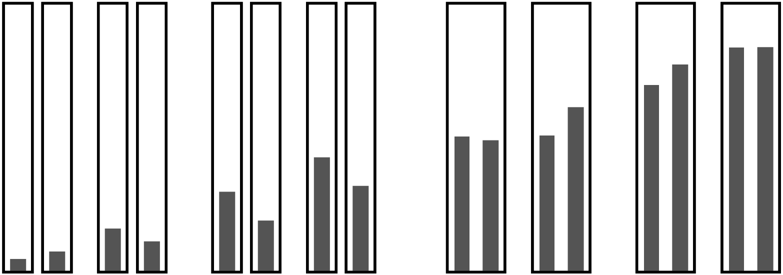
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



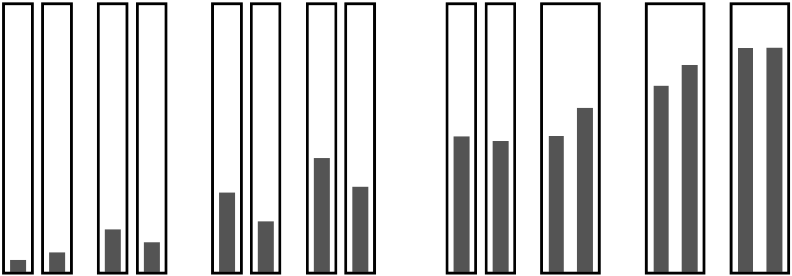
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



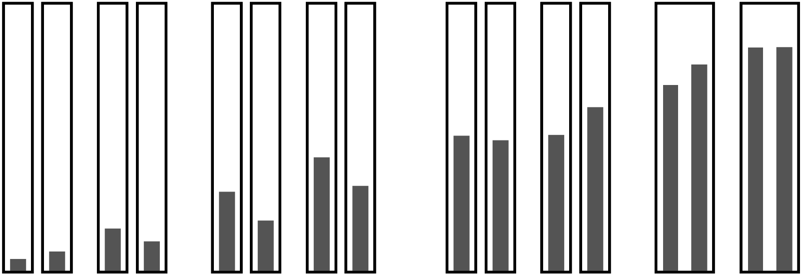
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   - 3.1 Apply *cswap* to pairs of elements at corresponding positions.
   - 3.2 Split each of the resulting lists in the middle.
   - 3.3 Merge the resulting pairs of lists recursively.



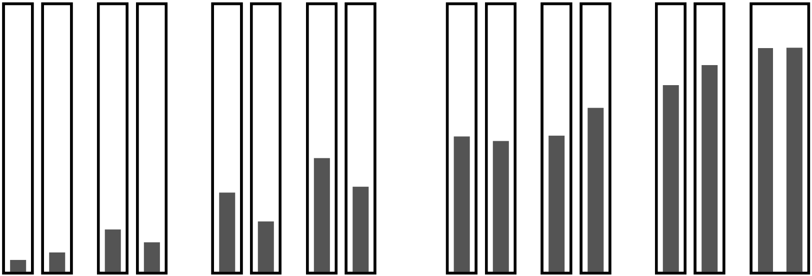
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
    3.1 Apply *cswap* to pairs of elements at corresponding positions.
    3.2 Split each of the resulting lists in the middle.
    3.3 Merge the resulting pairs of lists recursively.



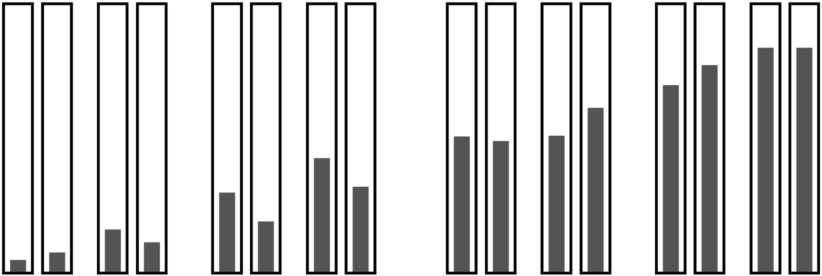
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



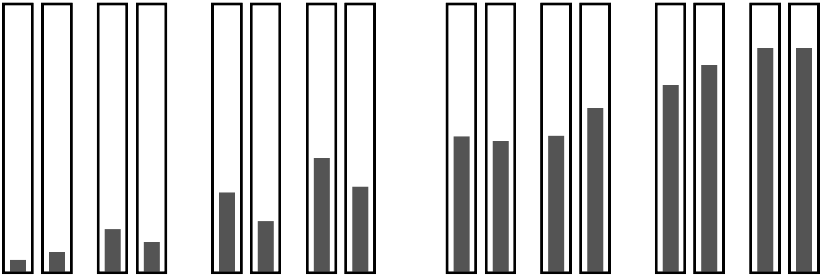
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



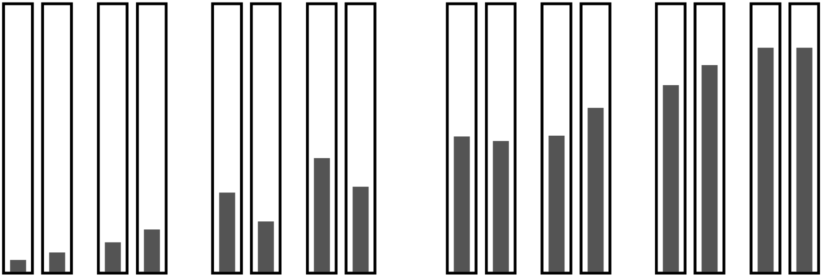
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



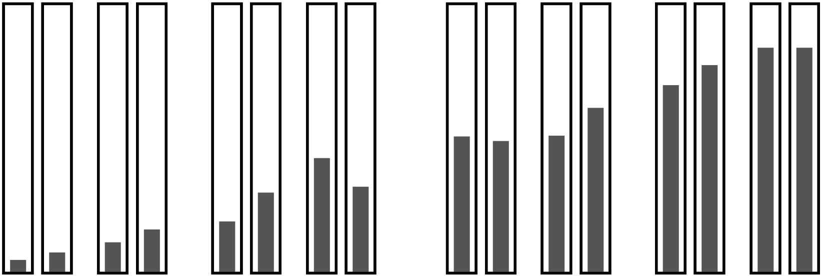
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



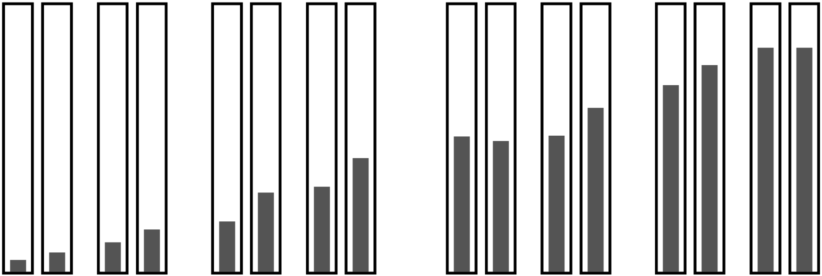
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



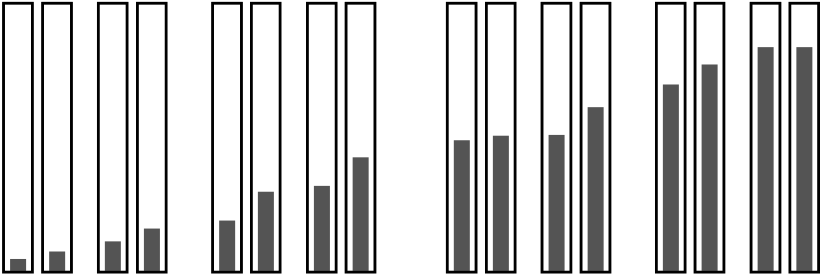
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



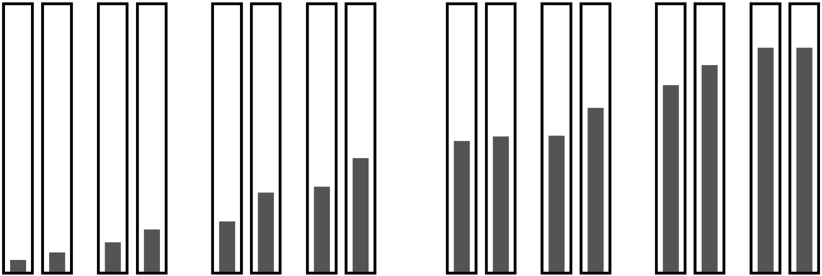
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



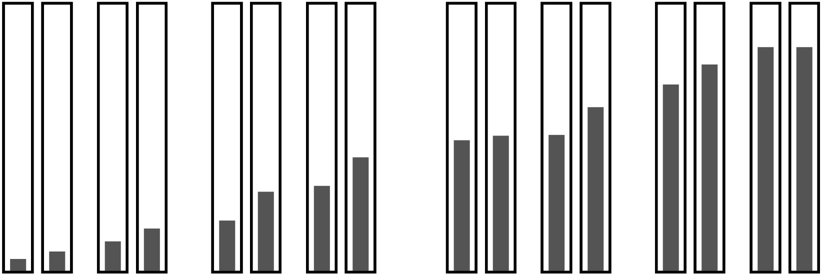
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
    3.1 Apply *cswap* to pairs of elements at corresponding positions.
    3.2 Split each of the resulting lists in the middle.
    3.3 Merge the resulting pairs of lists recursively.



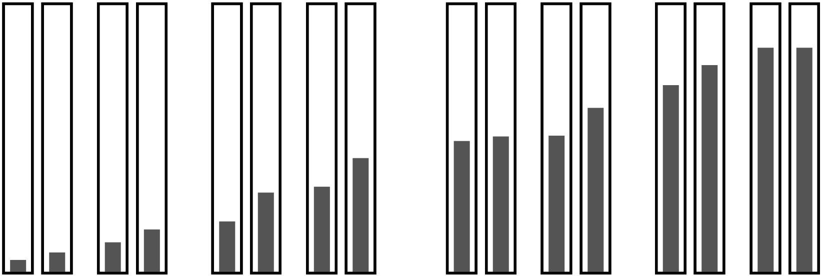
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



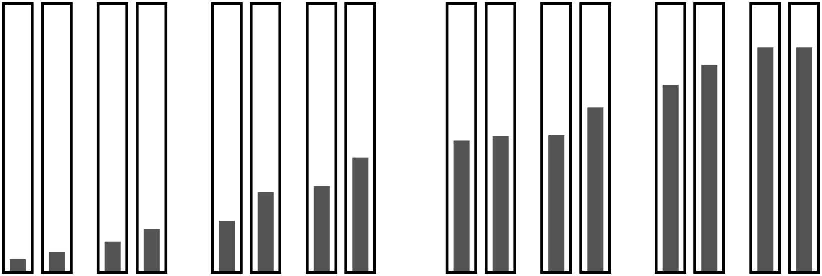
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



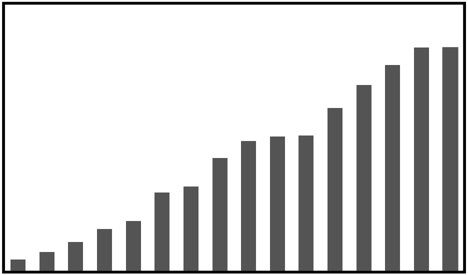
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.
   3.4 Concatenate the results.



Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.
   3.4 Concatenate the results.



Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.
   3.4 Concatenate the results.

Note:
- works only for lists whose length is a power of two

Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.
   3.4 Concatenate the results.

Note:
- works only for lists whose length is a power of two
- complexity is $O(n \cdot \log(n)^2)$

Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.
   3.4 Concatenate the results.

Note:
- works only for lists whose length is a power of two
- complexity is $O(n \cdot \log(n)^2)$
- particularly suitable for hardware and parallel implementations

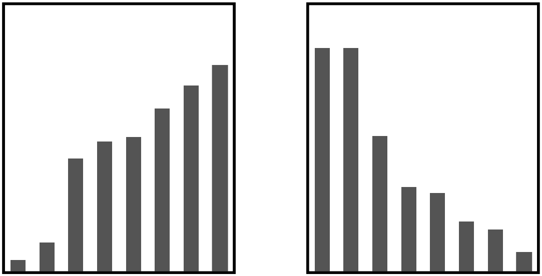
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.
   3.4 Concatenate the results.

Note:
- works only for lists whose length is a power of two
- complexity is $O(n \cdot \log(n)^2)$
- particularly suitable for hardware and parallel implementations
- correctness is not obvious

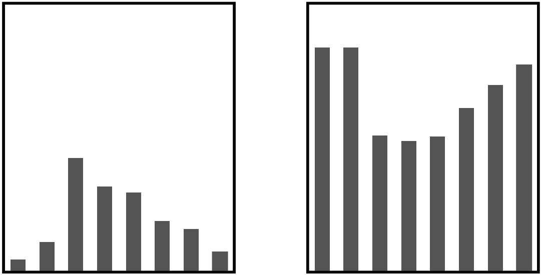
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.



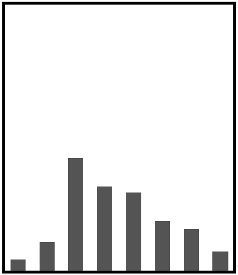
Bitonic Sort

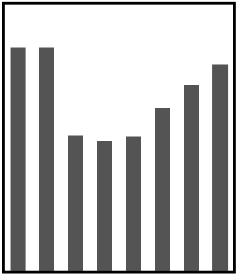
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.



Bitonic Sort

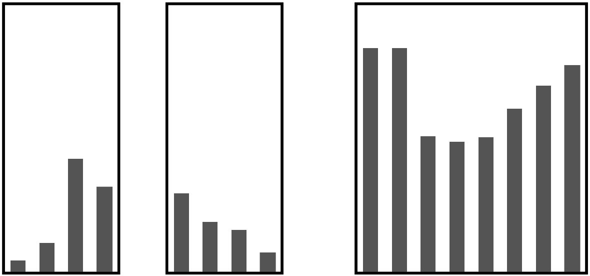
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.





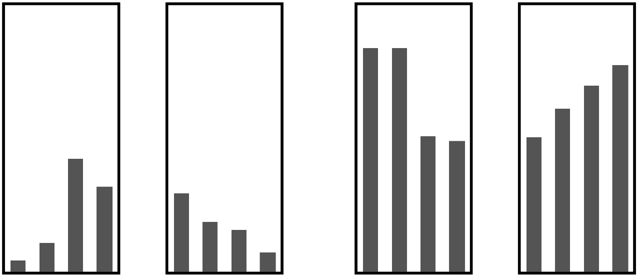
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.



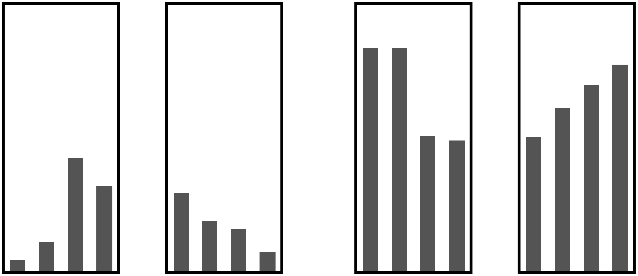
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.



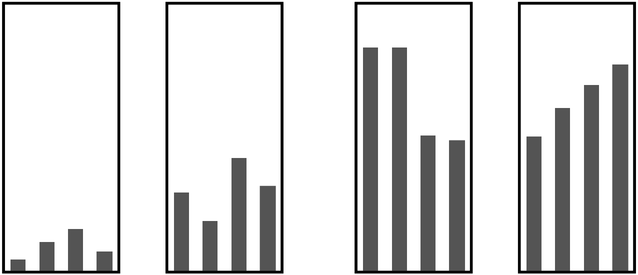
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



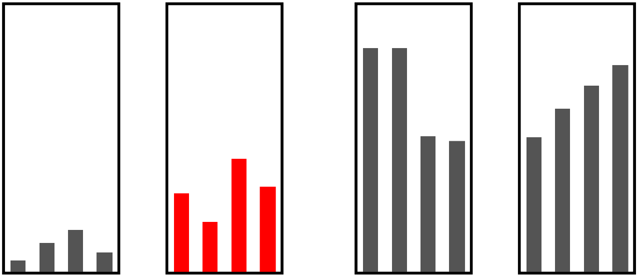
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.

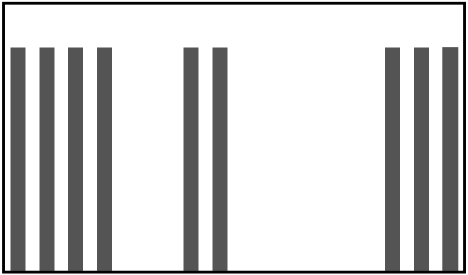


Knuth's 0-1-Principle [Knuth 1973]

Informally: If a comparison-swap algorithm sorts Booleans correctly, it sorts integers correctly as well.

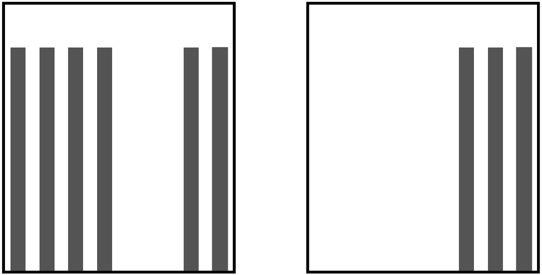
Bitonic Sort

1. Split the input list into two sublists of equal length.



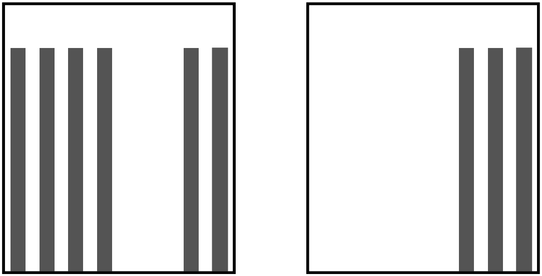
Bitonic Sort

1. Split the input list into two sublists of equal length.



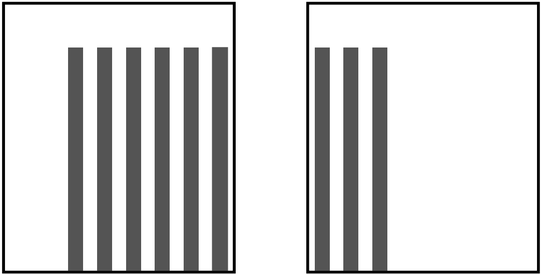
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.



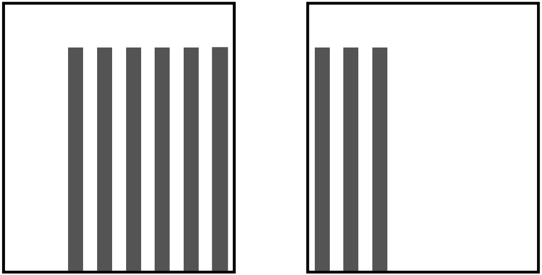
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.



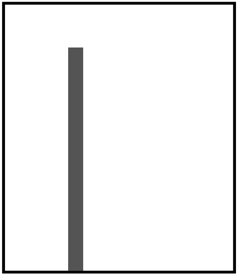
Bitonic Sort

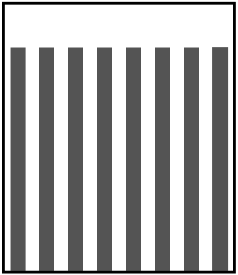
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
    3.1 Apply *cswap* to pairs of elements at corresponding positions.



Bitonic Sort

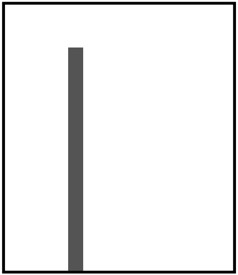
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.

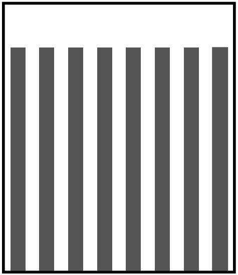




Bitonic Sort

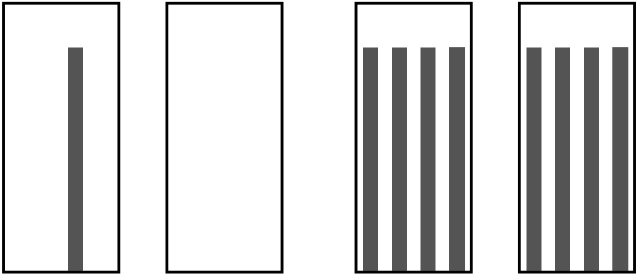
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.





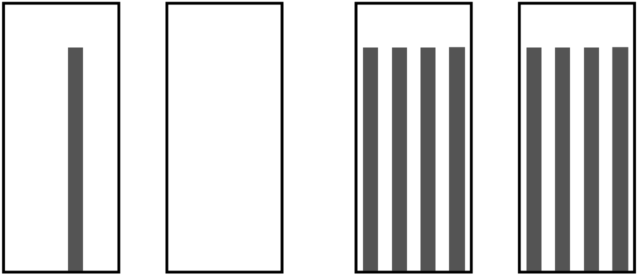
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.



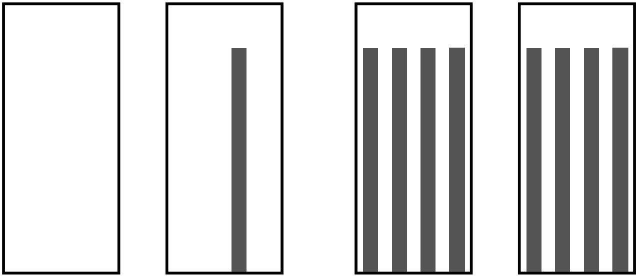
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



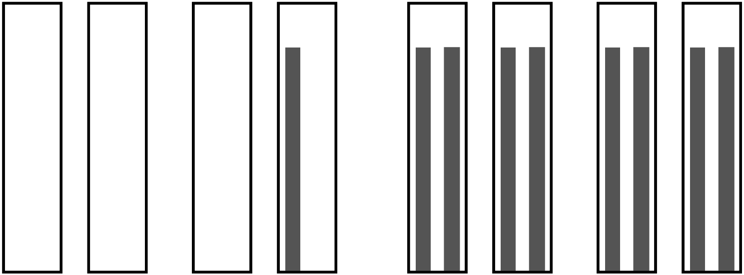
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



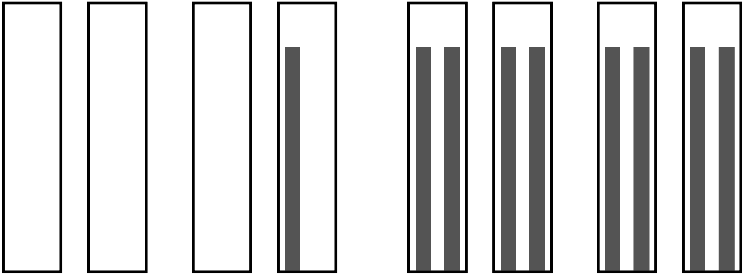
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



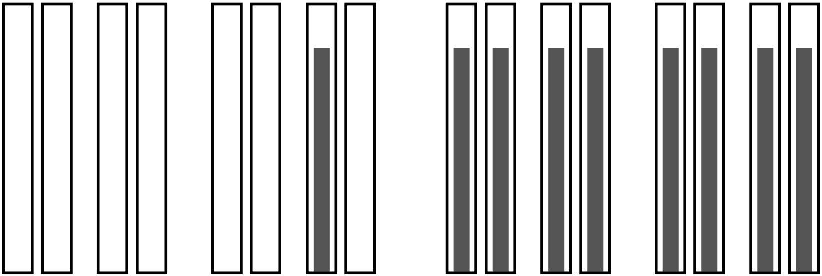
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



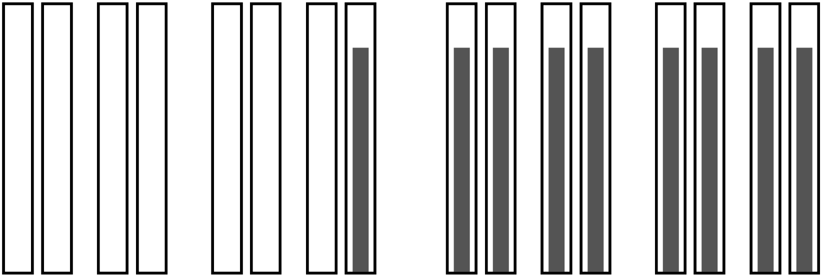
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



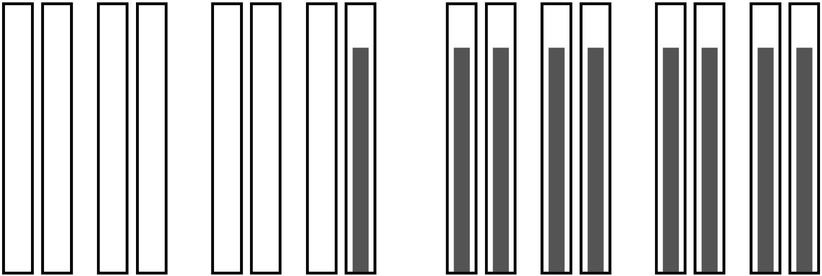
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



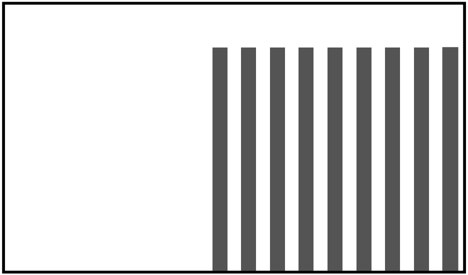
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.
   3.4 Concatenate the results.



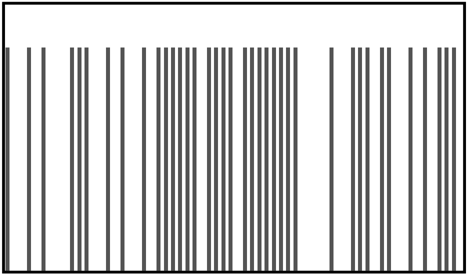
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.
   3.4 Concatenate the results.



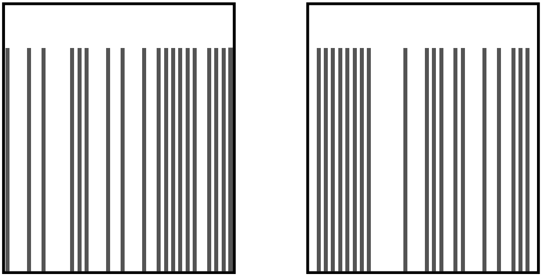
Bitonic Sort

1. Split the input list into two sublists of equal length.



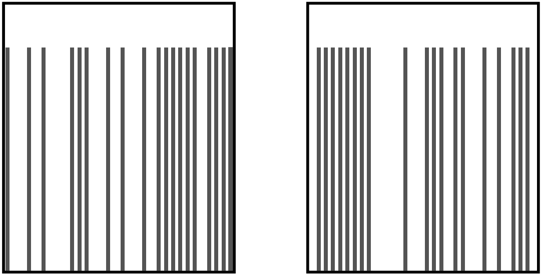
Bitonic Sort

1. Split the input list into two sublists of equal length.



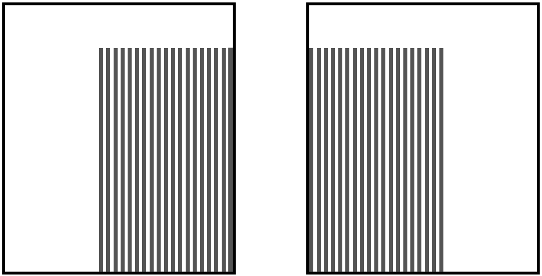
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.



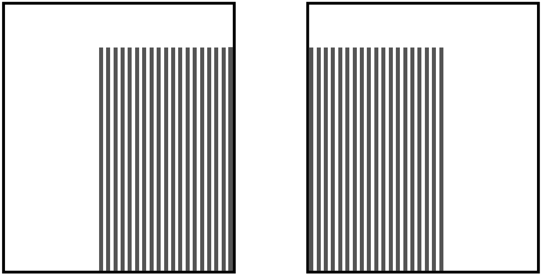
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.



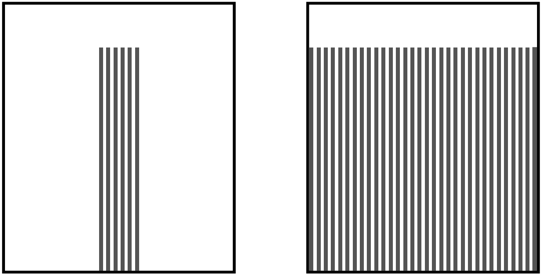
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
    3.1 Apply *cswap* to pairs of elements at corresponding positions.



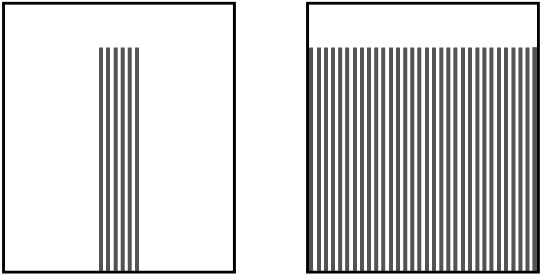
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.



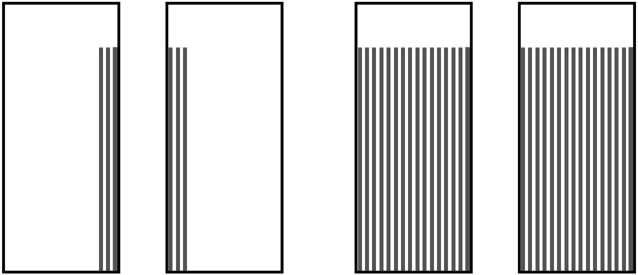
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
    3.1 Apply *cswap* to pairs of elements at corresponding positions.
    3.2 Split each of the resulting lists in the middle.



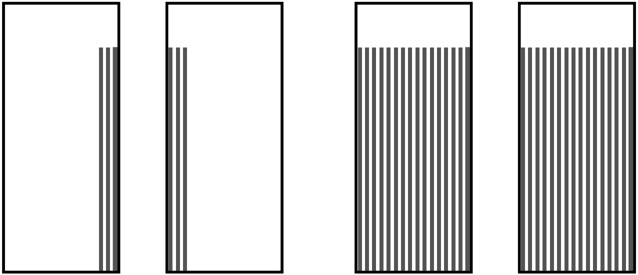
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.



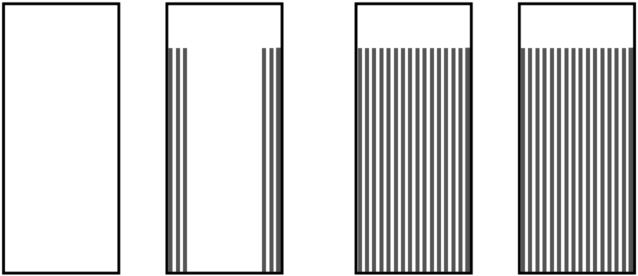
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
    3.1 Apply *cswap* to pairs of elements at corresponding positions.
    3.2 Split each of the resulting lists in the middle.
    3.3 Merge the resulting pairs of lists recursively.



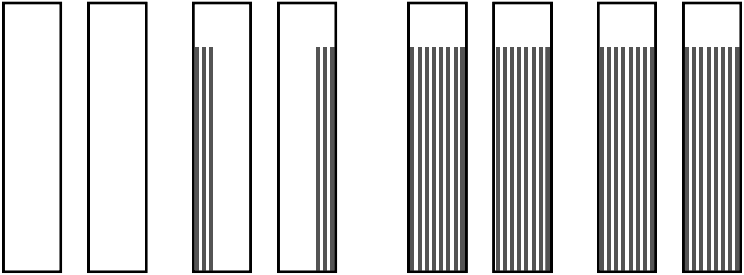
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



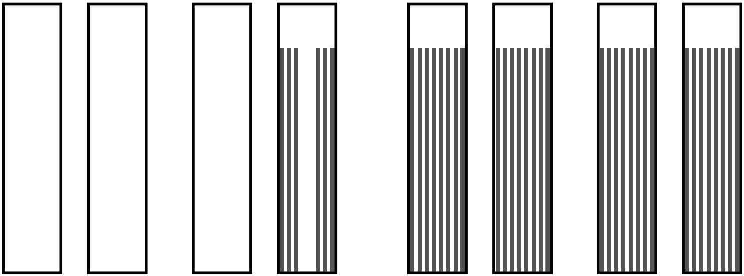
Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.



Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.
   3.3 Merge the resulting pairs of lists recursively.

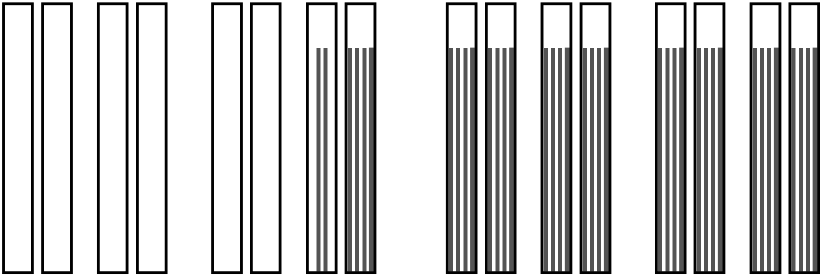


Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
    3.1 Apply *cswap* to pairs of elements at corresponding positions.
    3.2 Split each of the resulting lists in the middle.
    3.3 Merge the resulting pairs of lists recursively.

Bitonic Sort

1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
    3.1 Apply *cswap* to pairs of elements at corresponding positions.
    3.2 Split each of the resulting lists in the middle.
    3.3 Merge the resulting pairs of lists recursively.



Knuth's 0-1-Principle [Knuth 1973]

Informally: If a comparison-swap algorithm sorts Booleans correctly, it sorts integers correctly as well.

Knuth's 0-1-Principle [Knuth 1973]

Informally: If a comparison-swap algorithm sorts Booleans correctly, it sorts integers correctly as well.

Formally: ???

Knuth's 0-1-Principle [Knuth 1973]

Informally: If a comparison-swap algorithm sorts Booleans correctly, it sorts integers correctly as well.

Formally: Use Haskell.

Knuth's 0-1-Principle [Knuth 1973]

Informally: If a comparison-swap algorithm sorts Booleans correctly, it sorts integers correctly as well.

Formally: Use Haskell. Let

$$\texttt{sort} :: ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$$

Knuth's 0-1-Principle [Knuth 1973]

Informally: If a comparison-swap algorithm sorts Booleans correctly, it sorts integers correctly as well.

Formally: Use Haskell. Let

$$\texttt{sort} :: ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$$

$$f :: (\mathsf{Int}, \mathsf{Int}) \to (\mathsf{Int}, \mathsf{Int})$$
$$f\ (x, y) = \mathit{if}\ x > y\ \mathit{then}\ (y, x)\ \mathit{else}\ (x, y)$$

$$g :: (\mathsf{Bool}, \mathsf{Bool}) \to (\mathsf{Bool}, \mathsf{Bool})$$
$$g\ (x, y) = (x\ \&\&\ y, x \,||\, y)$$

Knuth's 0-1-Principle [Knuth 1973]

Informally: If a comparison-swap algorithm sorts Booleans correctly, it sorts integers correctly as well.

Formally: Use Haskell. Let

$$\texttt{sort} :: ((\alpha, \alpha) \rightarrow (\alpha, \alpha)) \rightarrow [\alpha] \rightarrow [\alpha]$$

$$\begin{aligned} & f :: (\mathsf{Int}, \mathsf{Int}) \rightarrow (\mathsf{Int}, \mathsf{Int}) \\ & f\ (x, y) = \mathit{if}\ x > y\ \mathit{then}\ (y, x)\ \mathit{else}\ (x, y) \end{aligned}$$

$$\begin{aligned} & g :: (\mathsf{Bool}, \mathsf{Bool}) \rightarrow (\mathsf{Bool}, \mathsf{Bool}) \\ & g\ (x, y) = (x \ \&\&\ y, x \ ||\ y) \end{aligned}$$

If for every $xs :: [\mathsf{Bool}]$, $\texttt{sort}\ g\ xs$ gives the correct result, then for every $xs :: [\mathsf{Int}]$, $\texttt{sort}\ f\ xs$ gives the correct result.

Knuth's 0-1-Principle [Knuth 1973]

Informally: If a comparison-swap algorithm sorts Booleans correctly, it sorts integers correctly as well.

Formally: Use Haskell. Let

$$\texttt{sort} :: ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$$

$$f :: (\mathsf{Int}, \mathsf{Int}) \to (\mathsf{Int}, \mathsf{Int})$$
$$f\ (x, y) = \mathit{if}\ x > y\ \mathit{then}\ (y, x)\ \mathit{else}\ (x, y)$$

$$g :: (\mathsf{Bool}, \mathsf{Bool}) \to (\mathsf{Bool}, \mathsf{Bool})$$
$$g\ (x, y) = (x\ \&\&\ y, x\ ||\ y)$$

If $\forall xs :: [\mathsf{Bool}], ys = \texttt{sort}\ g\ xs.\ P(xs, ys) \wedge Q(ys)$,
then $\forall xs :: [\mathsf{Int}], ys = \texttt{sort}\ f\ xs.\ P(xs, ys) \wedge Q(ys)$,
where $P(xs, ys) :=$ xs and ys contain the same elements in the same multiplicity
$Q(ys) :=$ ys is sorted

Free Theorems [Reynolds 1983, Wadler 1989]

Here just a magic black box.

Free Theorems [Reynolds 1983, Wadler 1989]

Here just a magic black box.

Input: `sort::((a,a)->(a,a))->[a]->[a]`

Free Theorems [Reynolds 1983, Wadler 1989]

Here just a magic black box.

Input:
```
sort::((a,a)->(a,a))->[a]->[a]
```

Output:
```
forall t1,t2 in TYPES, h::t1->t2.
 forall f::(t1,t1)->(t1,t1).
  forall g::(t2,t2)->(t2,t2).
   (forall (x,y) in lift_{(,)}(h,h).
     (f x,g y) in lift_{(,)}(h,h))
   ==> (forall xs::[t1].
         map h (sort f xs) = sort g (map h xs))
```

```
lift_{(,)}(h,h)
 = {((x1,x2),(y1,y2)) | (h x1 = y1)
                         && (h x2 = y2)}
```

```
map h []     = []
map h (x:xs) = (h x):(map h xs)
```

More Specific (and Intuitive)

For every $\texttt{sort} :: ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$,
$f :: (\mathsf{Int}, \mathsf{Int}) \to (\mathsf{Int}, \mathsf{Int})$, $g :: (\mathsf{Bool}, \mathsf{Bool}) \to (\mathsf{Bool}, \mathsf{Bool})$, and
$h :: \mathsf{Int} \to \mathsf{Bool}$:

$$\begin{tikzcd}
(\mathsf{Int}, \mathsf{Int}) \arrow[r, "f"] \arrow[d, "h \times h"'] \arrow[dr, phantom, "="] & (\mathsf{Int}, \mathsf{Int}) \arrow[d, "h \times h"] \\
(\mathsf{Bool}, \mathsf{Bool}) \arrow[r, "g"'] & (\mathsf{Bool}, \mathsf{Bool})
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
[\mathsf{Int}] \arrow[r, "\texttt{sort}\ f"] \arrow[d, "\texttt{map}\ h"'] \arrow[dr, phantom, "="] & [\mathsf{Int}] \arrow[d, "\texttt{map}\ h"] \\
[\mathsf{Bool}] \arrow[r, "\texttt{sort}\ g"'] & [\mathsf{Bool}]
\end{tikzcd}$$

More Specific (and Intuitive)

For every $\texttt{sort} :: ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$,
$f :: (\mathsf{Int}, \mathsf{Int}) \to (\mathsf{Int}, \mathsf{Int})$, $g :: (\mathsf{Bool}, \mathsf{Bool}) \to (\mathsf{Bool}, \mathsf{Bool})$, and
$h :: \mathsf{Int} \to \mathsf{Bool}$:

$$\begin{tikzcd}
(\mathsf{Int}, \mathsf{Int}) \arrow[r, "f"] \arrow[d, "h \times h"'] \arrow[dr, phantom, "="] & (\mathsf{Int}, \mathsf{Int}) \arrow[d, "h \times h"] \\
(\mathsf{Bool}, \mathsf{Bool}) \arrow[r, "g"'] & (\mathsf{Bool}, \mathsf{Bool})
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
[\mathsf{Int}] \arrow[r, "\texttt{sort}\ f"] \arrow[d, "\texttt{map}\ h"'] \arrow[dr, phantom, "="] & [\mathsf{Int}] \arrow[d, "\texttt{map}\ h"] \\
[\mathsf{Bool}] \arrow[r, "\texttt{sort}\ g"'] & [\mathsf{Bool}]
\end{tikzcd}$$

If f and g are as defined before, then the precondition is fulfilled for any h of the form $h\ x = n < x$ for some $n :: \mathsf{Int}$.

Knuth's 0-1-Principle [Knuth 1973]

Informally: If a comparison-swap algorithm sorts Booleans correctly, it sorts integers correctly as well.

Formally: Use Haskell. Let

$$\texttt{sort} :: ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$$

$$f :: (\mathsf{Int}, \mathsf{Int}) \to (\mathsf{Int}, \mathsf{Int})$$
$$f\ (x, y) = \mathit{if}\ x > y\ \mathit{then}\ (y, x)\ \mathit{else}\ (x, y)$$

$$g :: (\mathsf{Bool}, \mathsf{Bool}) \to (\mathsf{Bool}, \mathsf{Bool})$$
$$g\ (x, y) = (x\ \&\&\ y, x \mid\mid y)$$

If $\forall xs :: [\mathsf{Bool}], ys = \texttt{sort}\ g\ xs.\ P(xs, ys) \wedge Q(ys)$,
then $\forall xs :: [\mathsf{Int}], ys = \texttt{sort}\ f\ xs.\ P(xs, ys) \wedge Q(ys)$,
where $P(xs, ys) :=$ xs and ys contain the same elements in the same multiplicity
$Q(ys) :=$ ys is sorted

Proof of “P on [Bool] implies P on [Int]”

Recall: $P(xs, ys) :=$ *xs* and *ys* contain the same elements in the same multiplicity

Proof of "P on [Bool] implies P on [Int]"

Recall: $P(xs, ys) :=$ xs and ys contain the same elements in the same multiplicity

Given: $\forall xs :: [\text{Bool}], ys = \texttt{sort}\ g\ xs.\ P(xs, ys)$

Proof of "P on [Bool] implies P on [Int]"

Recall: $P(xs, ys) :=$ xs and ys contain the same elements in the same multiplicity

Given: $\forall xs :: [\text{Bool}], ys = \texttt{sort}\ g\ xs.\ P(xs, ys)$

To prove: $\forall xs :: [\text{Int}], ys = \texttt{sort}\ f\ xs.\ P(xs, ys)$

Proof of "P on [Bool] implies P on [Int]"

Recall: $P(xs, ys) :=$ xs and ys contain the same elements in the same multiplicity

Given: $\forall xs :: [\mathrm{Bool}], ys = \texttt{sort}\ g\ xs.\ P(xs, ys)$

To prove: $\forall xs :: [\mathrm{Int}], ys = \texttt{sort}\ f\ xs.\ P(xs, ys)$

Assume there exist $us :: [\mathrm{Int}]$ and $vs = \texttt{sort}\ f\ us$ with $\neg P(us, vs)$.

Proof of "P on [Bool] implies P on [Int]"

Recall: $P(xs, ys) :=$ xs and ys contain the same elements in the same multiplicity

Given: $\forall xs :: [\text{Bool}], ys = \texttt{sort}\ g\ xs.\ P(xs, ys)$

To prove: $\forall xs :: [\text{Int}], ys = \texttt{sort}\ f\ xs.\ P(xs, ys)$

Assume there exist $us :: [\text{Int}]$ and $vs = \texttt{sort}\ f\ us$ with $\neg P(us, vs)$. Then there is a smallest integer n such that the multiplicities of n in us and vs are not the same.

Proof of "P on [Bool] implies P on [Int]"

Recall: $P(xs, ys) :=$ xs and ys contain the same elements in the same multiplicity

Given: $\forall xs :: [\text{Bool}], ys = \texttt{sort}\ g\ xs.\ P(xs, ys)$

To prove: $\forall xs :: [\text{Int}], ys = \texttt{sort}\ f\ xs.\ P(xs, ys)$

Assume there exist $us :: [\text{Int}]$ and $vs = \texttt{sort}\ f\ us$ with $\neg P(us, vs)$. Then there is a smallest integer n such that the multiplicities of n in us and vs are not the same. Then for $h\ x = n < x$ the multiplicities of False in ($\texttt{map}\ h\ us$) and ($\texttt{map}\ h\ vs$) are different.

Proof of "P on [Bool] implies P on [Int]"

Recall: $P(xs, ys) :=$ xs and ys contain the same elements in the same multiplicity

Given: $\forall xs :: [\text{Bool}], ys = \texttt{sort}\ g\ xs.\ P(xs, ys)$

To prove: $\forall xs :: [\text{Int}], ys = \texttt{sort}\ f\ xs.\ P(xs, ys)$

Assume there exist $us :: [\text{Int}]$ and $vs = \texttt{sort}\ f\ us$ with $\neg P(us, vs)$. Then there is a smallest integer n such that the multiplicities of n in us and vs are not the same. Then for $h\ x = n < x$ the multiplicities of False in ($\texttt{map}\ h\ us$) and ($\texttt{map}\ h\ vs$) are different. But this is in contradiction to the precondition with:

$$xs = \texttt{map}\ h\ us$$
$$ys = \texttt{sort}\ g\ (\texttt{map}\ h\ us)$$

Proof of "P on [Bool] implies P on [Int]"

Recall: $P(xs, ys) :=$ xs and ys contain the same elements in the same multiplicity

Given: $\forall xs :: [\text{Bool}], ys = \texttt{sort}\ g\ xs.\ P(xs, ys)$

To prove: $\forall xs :: [\text{Int}], ys = \texttt{sort}\ f\ xs.\ P(xs, ys)$

Assume there exist $us :: [\text{Int}]$ and $vs = \texttt{sort}\ f\ us$ with $\neg P(us, vs)$. Then there is a smallest integer n such that the multiplicities of n in us and vs are not the same. Then for $h\ x = n < x$ the multiplicities of False in ($\texttt{map}\ h\ us$) and ($\texttt{map}\ h\ vs$) are different. But this is in contradiction to the precondition with:

$$xs = \texttt{map}\ h\ us$$
$$ys = \texttt{sort}\ g\ (\texttt{map}\ h\ us) = \texttt{map}\ h\ (\texttt{sort}\ f\ us)$$

Proof of "P on [Bool] implies P on [Int]"

Recall: $P(xs, ys) :=$ xs and ys contain the same elements in the same multiplicity

Given: $\forall xs :: [\mathrm{Bool}], ys = \texttt{sort}\ g\ xs.\ P(xs, ys)$

To prove: $\forall xs :: [\mathrm{Int}], ys = \texttt{sort}\ f\ xs.\ P(xs, ys)$

Assume there exist $us :: [\mathrm{Int}]$ and $vs = \texttt{sort}\ f\ us$ with $\neg P(us, vs)$. Then there is a smallest integer n such that the multiplicities of n in us and vs are not the same. Then for $h\ x = n < x$ the multiplicities of False in ($\texttt{map}\ h\ us$) and ($\texttt{map}\ h\ vs$) are different. But this is in contradiction to the precondition with:

$$xs = \texttt{map}\ h\ us$$
$$ys = \texttt{sort}\ g\ (\texttt{map}\ h\ us) = \texttt{map}\ h\ (\texttt{sort}\ f\ us) = \texttt{map}\ h\ vs$$

Proof of “Q on [Bool] implies Q on [Int]”

Recall: $Q(ys) := ys$ is sorted

Proof of "Q on [Bool] implies Q on [Int]"

Recall: $Q(\mathit{ys}) := \mathit{ys}$ is sorted

Given: $\forall \mathit{xs} :: [\mathrm{Bool}], \mathit{ys} = \texttt{sort}\ g\ \mathit{xs}.\ Q(\mathit{ys})$

Proof of "Q on [Bool] implies Q on [Int]"

Recall: $Q(ys) := ys$ is sorted

Given: $\forall xs :: [\text{Bool}], ys = \texttt{sort}\ g\ xs.\ Q(ys)$

To prove: $\forall xs :: [\text{Int}], ys = \texttt{sort}\ f\ xs.\ Q(ys)$

Proof of "Q on [Bool] implies Q on [Int]"

$$\begin{aligned} \text{Recall: } & Q(ys) := ys \text{ is sorted} \\ \text{Given: } & \forall xs :: [\text{Bool}], ys = \texttt{sort}\ g\ xs.\ Q(ys) \\ \text{To prove: } & \forall xs :: [\text{Int}], ys = \texttt{sort}\ f\ xs.\ Q(ys) \end{aligned}$$

Assume there exist us :: [Int] and $vs = \texttt{sort}\ f\ us$ with $\neg Q(vs)$.

Proof of “Q on [Bool] implies Q on [Int]”

Recall: $Q(ys) := ys$ is sorted

Given: $\forall xs :: [\mathrm{Bool}], ys = \texttt{sort}\ g\ xs.\ Q(ys)$

To prove: $\forall xs :: [\mathrm{Int}], ys = \texttt{sort}\ f\ xs.\ Q(ys)$

Assume there exist $us :: [\mathrm{Int}]$ and $vs = \texttt{sort}\ f\ us$ with $\neg Q(vs)$.
Then there are $n < m$ such that an m occurs in vs before an n.

Proof of "Q on [Bool] implies Q on [Int]"

Recall: $Q(ys) := ys$ is sorted

Given: $\forall xs :: [\text{Bool}], ys = \texttt{sort}\ g\ xs.\ Q(ys)$

To prove: $\forall xs :: [\text{Int}], ys = \texttt{sort}\ f\ xs.\ Q(ys)$

Assume there exist us :: [Int] and $vs = \texttt{sort}\ f\ us$ with $\neg Q(vs)$.
Then there are $n < m$ such that an m occurs in vs before an n.
Then for $h\ x = n < x$ a True occurs in ($\texttt{map}\ h\ vs$) before a False.

Proof of "Q on [Bool] implies Q on [Int]"

$$
\begin{aligned}
\text{Recall: } & Q(ys) := ys \text{ is sorted} \\
\text{Given: } & \forall xs :: [\text{Bool}], ys = \texttt{sort}\ g\ xs.\ Q(ys) \\
\text{To prove: } & \forall xs :: [\text{Int}], ys = \texttt{sort}\ f\ xs.\ Q(ys)
\end{aligned}
$$

Assume there exist us :: [Int] and $vs = \texttt{sort}\ f\ us$ with $\neg Q(vs)$.
Then there are $n < m$ such that an m occurs in vs before an n.
Then for $h\ x = n < x$ a True occurs in ($\texttt{map}\ h\ vs$) before a False.
But this is in contradiction to the precondition with:

$$
\begin{aligned}
xs &= \texttt{map}\ h\ us \\
ys &= \texttt{sort}\ g\ (\texttt{map}\ h\ us)
\end{aligned}
$$

Proof of "Q on [Bool] implies Q on [Int]"

Recall: $Q(ys) := ys$ is sorted

Given: $\forall xs :: [\text{Bool}], ys = \texttt{sort}\ g\ xs.\ Q(ys)$

To prove: $\forall xs :: [\text{Int}], ys = \texttt{sort}\ f\ xs.\ Q(ys)$

Assume there exist $us :: [\text{Int}]$ and $vs = \texttt{sort}\ f\ us$ with $\neg Q(vs)$.
Then there are $n < m$ such that an m occurs in vs before an n.
Then for $h\ x = n < x$ a True occurs in $(\texttt{map}\ h\ vs)$ before a False.
But this is in contradiction to the precondition with:

$$xs = \texttt{map}\ h\ us$$
$$ys = \texttt{sort}\ g\ (\texttt{map}\ h\ us) = \texttt{map}\ h\ (\texttt{sort}\ f\ us)$$

Proof of "Q on [Bool] implies Q on [Int]"

Recall: $Q(ys) := ys$ is sorted

Given: $\forall xs :: [\mathrm{Bool}], ys = \texttt{sort}\ g\ xs.\ Q(ys)$

To prove: $\forall xs :: [\mathrm{Int}], ys = \texttt{sort}\ f\ xs.\ Q(ys)$

Assume there exist $us :: [\mathrm{Int}]$ and $vs = \texttt{sort}\ f\ us$ with $\neg Q(vs)$.
Then there are $n < m$ such that an m occurs in vs before an n.
Then for $h\ x = n < x$ a True occurs in $(\texttt{map}\ h\ vs)$ before a False.
But this is in contradiction to the precondition with:

$$xs = \texttt{map}\ h\ us$$
$$ys = \texttt{sort}\ g\ (\texttt{map}\ h\ us) = \texttt{map}\ h\ (\texttt{sort}\ f\ us) = \texttt{map}\ h\ vs$$

Knuth's 0-1-Principle [Knuth 1973]

Informally: If a comparison-swap algorithm sorts Booleans correctly, it sorts integers correctly as well.

Formally: Use Haskell. Let

$$\texttt{sort} :: ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$$

$$f :: (\mathsf{Int}, \mathsf{Int}) \to (\mathsf{Int}, \mathsf{Int})$$
$$f\ (x, y) = \mathit{if}\ x > y\ \mathit{then}\ (y, x)\ \mathit{else}\ (x, y)$$

$$g :: (\mathsf{Bool}, \mathsf{Bool}) \to (\mathsf{Bool}, \mathsf{Bool})$$
$$g\ (x, y) = (x\ \&\&\ y, x \mathbin{||} y)$$

If for every $xs :: [\mathsf{Bool}]$, $\texttt{sort}\ g\ xs$ gives the correct result, then for every $xs :: [\mathsf{Int}]$, $\texttt{sort}\ f\ xs$ gives the correct result.

And Beyond?

- Knuth's 0-1-Principle allows to reduce algorithm correctness, for comparison-swap sorting, for inputs over an infinite range to correctness over a finite range of values.

And Beyond?

- Knuth's 0-1-Principle allows to reduce algorithm correctness, for comparison-swap sorting, for inputs over an infinite range to correctness over a finite range of values.

- Free theorems allow for a particularly elegant proof of this principle. (This was not my idea: [Day et al. 1999]!)

And Beyond?

- Knuth's 0-1-Principle allows to reduce algorithm correctness, for comparison-swap sorting, for inputs over an infinite range to correctness over a finite range of values.

- Free theorems allow for a particularly elegant proof of this principle. (This was not my idea: [Day et al. 1999]!)

- Can we do something similar for other algorithm classes?

And Beyond?

- Knuth's 0-1-Principle allows to reduce algorithm correctness, for comparison-swap sorting, for inputs over an infinite range to correctness over a finite range of values.
- Free theorems allow for a particularly elegant proof of this principle. (This was not my idea: [Day et al. 1999]!)
- Can we do something similar for other algorithm classes?
- Good candidates: algorithms parametrised over some operation, like $\mathit{cswap} :: (\alpha, \alpha) \to (\alpha, \alpha)$ in the case of sorting.

Parallel Prefix Computation

Given: inputs $x_1, \ldots, x_n$ and an associative operation $\oplus$

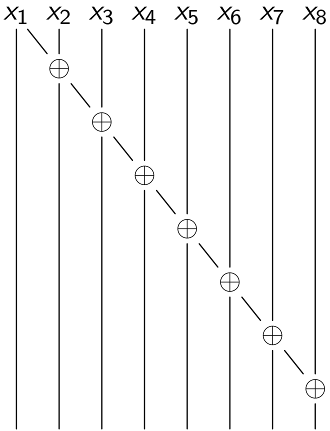
Task: compute the values $x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n$

Parallel Prefix Computation

Given: inputs $x_1, \ldots, x_n$ and an associative operation $\oplus$

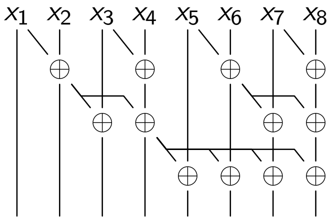
Task: compute the values $x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n$

Solution:



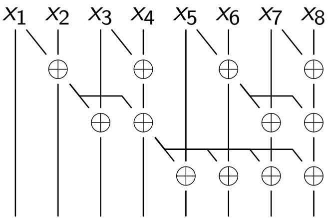
Parallel Prefix Computation

Alternative:

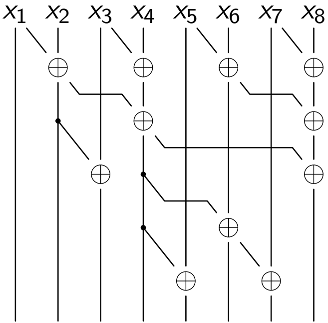


Parallel Prefix Computation

Alternative:



Or:



Or: ...

In Haskell

Functions of type:

$$\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

In Haskell

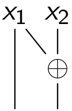
Functions of type:

$$\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

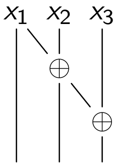
For example, à la [Sklansky 1960]:

$$\begin{aligned}
&\texttt{sklansky} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
&\texttt{sklansky}\ (\oplus)\ [x] = [x] \\
&\texttt{sklansky}\ (\oplus)\ xs\ \ = us +\!\!+\, vs \\
&\qquad \textbf{where}\ t \qquad\quad = ((\texttt{length}\ xs) + 1)\ `\texttt{div}`\ 2 \\
&\qquad\qquad\quad (ys, zs) = \texttt{splitAt}\ t\ xs \\
&\qquad\qquad\quad us \qquad = \texttt{sklansky}\ (\oplus)\ ys \\
&\qquad\qquad\quad vs \qquad = [(\texttt{last}\ us) \oplus v \mid v \leftarrow \texttt{sklansky}\ (\oplus)\ zs]
\end{aligned}$$

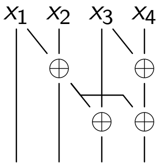
Sklansky's Method Visualised



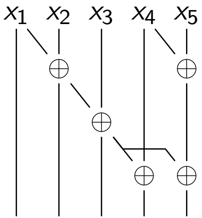
Sklansky's Method Visualised



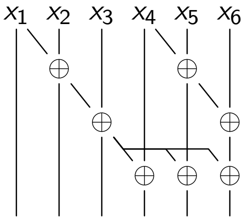
Sklansky's Method Visualised



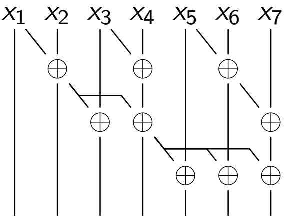
Sklansky's Method Visualised



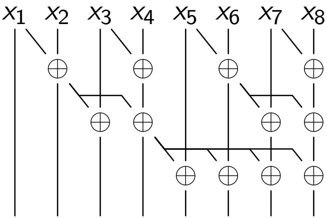
Sklansky's Method Visualised



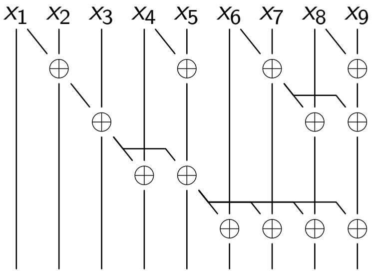
Sklansky's Method Visualised



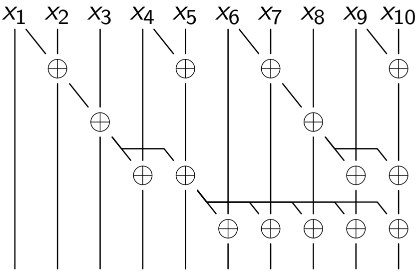
Sklansky's Method Visualised



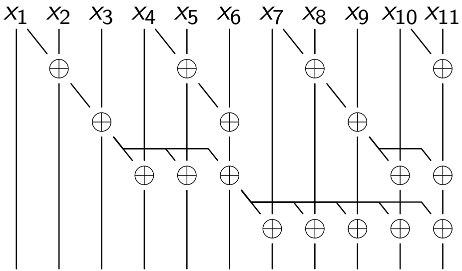
Sklansky's Method Visualised



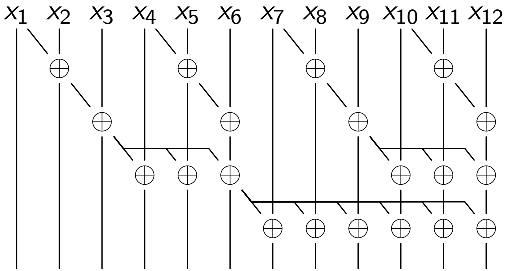
Sklansky's Method Visualised



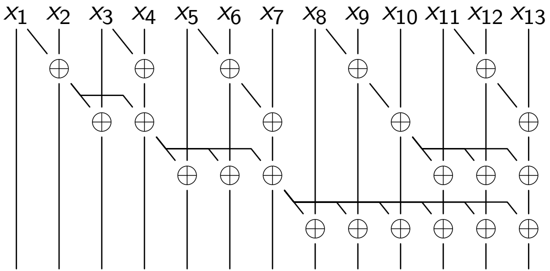
Sklansky's Method Visualised



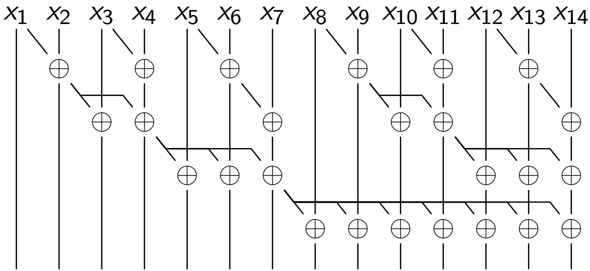
Sklansky's Method Visualised



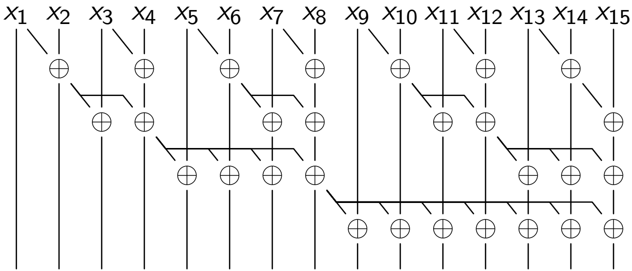
Sklansky's Method Visualised



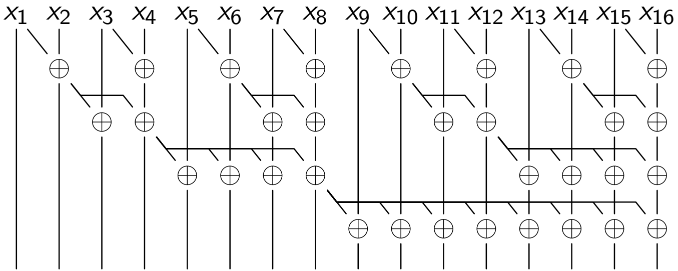
Sklansky's Method Visualised



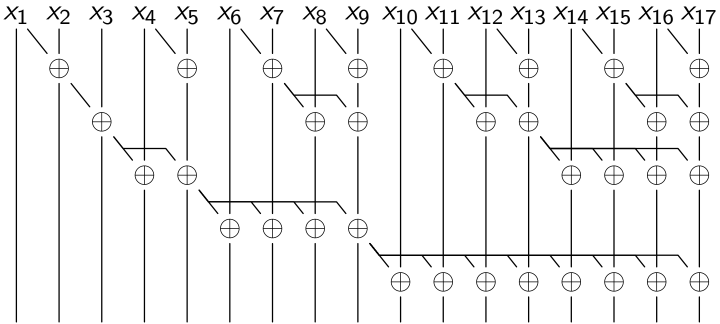
Sklansky's Method Visualised



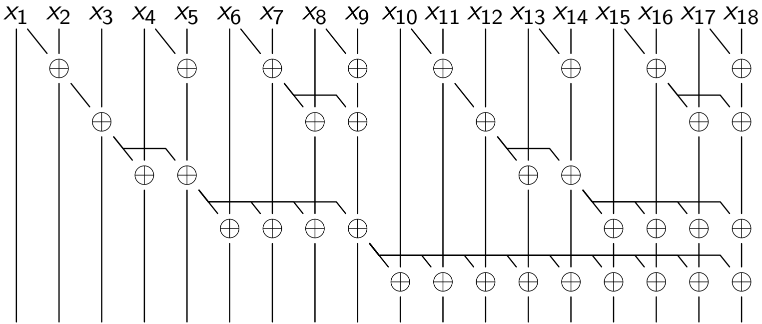
Sklansky's Method Visualised



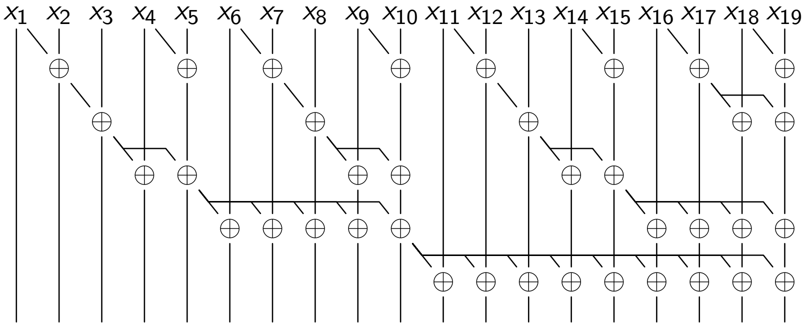
Sklansky's Method Visualised



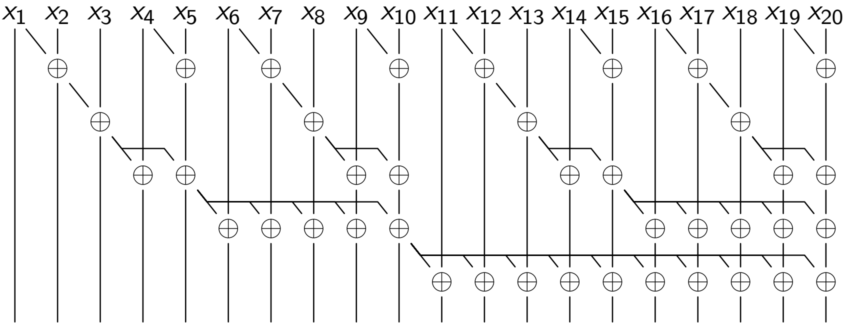
Sklansky's Method Visualised



Sklansky's Method Visualised



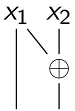
Sklansky's Method Visualised



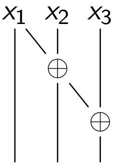
Or, à la [Brent & Kung 1980] (code follows [Sheeran 2007])

$$
\begin{aligned}
&\texttt{brentKung} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
&\texttt{brentKung}\ (\oplus)\ [x] = [x] \\
&\texttt{brentKung}\ (\oplus)\ xs\ = odds\ (riffle\ (par\ (unriffle\ (evens\ xs)))) \\
&\quad \textbf{where}\ evens\ [\,] \qquad\qquad = [\,] \\
&\qquad\qquad evens\ [x] \qquad\qquad = [x] \\
&\qquad\qquad evens\ (x : y : zs) = [x, x \oplus y] +\!\!+\ evens\ zs \\
&\qquad\qquad unriffle\ [\,] \qquad\qquad = ([\,],[\,]) \\
&\qquad\qquad unriffle\ [x] \qquad\qquad = ([x],[\,]) \\
&\qquad\qquad unriffle\ (x : y : zs) = (x : xs, y : ys) \\
&\qquad\qquad\quad \textbf{where}\ (xs, ys) = unriffle\ zs \\
&\qquad\qquad par\ (xs, ys) = (xs, \texttt{brentKung}\ (\oplus)\ ys) \\
&\qquad\qquad riffle\ ([\,],[\,]) \qquad\qquad = [\,] \\
&\qquad\qquad riffle\ ([x],[\,]) \qquad\qquad = [x] \\
&\qquad\qquad riffle\ (x : xs, y : ys) = x : y : riffle\ (xs, ys) \\
&\qquad\qquad odds\ (x : xs) = x : evens\ xs
\end{aligned}
$$

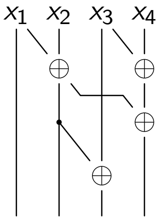
Brent & Kung's Method Visualised



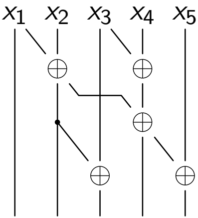
Brent & Kung's Method Visualised



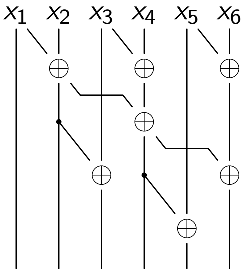
Brent & Kung's Method Visualised



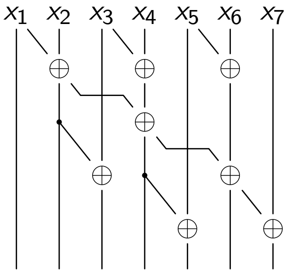
Brent & Kung's Method Visualised



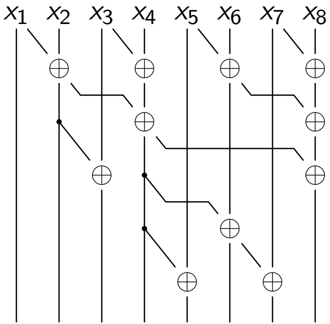
Brent & Kung's Method Visualised



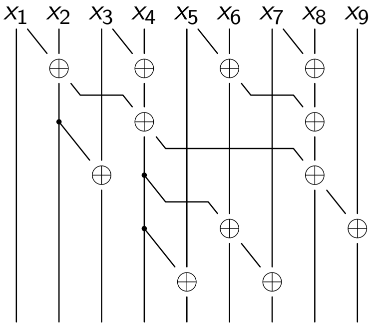
Brent & Kung's Method Visualised



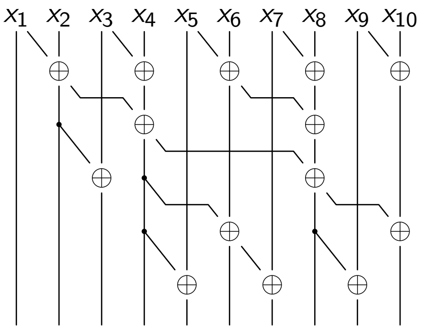
Brent & Kung's Method Visualised



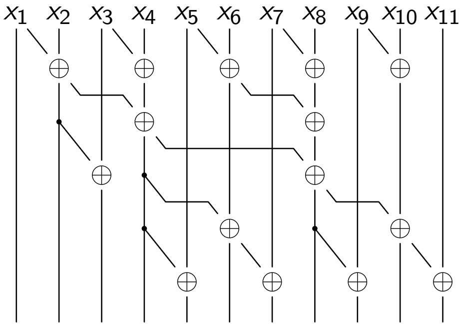
Brent & Kung's Method Visualised



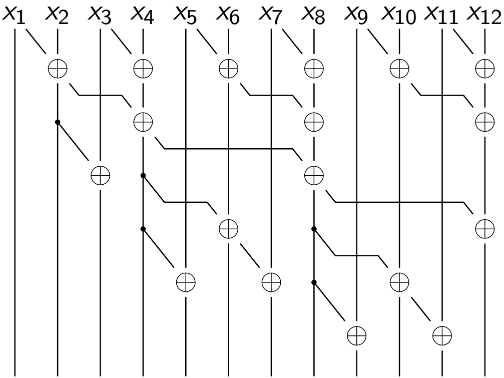
Brent & Kung's Method Visualised



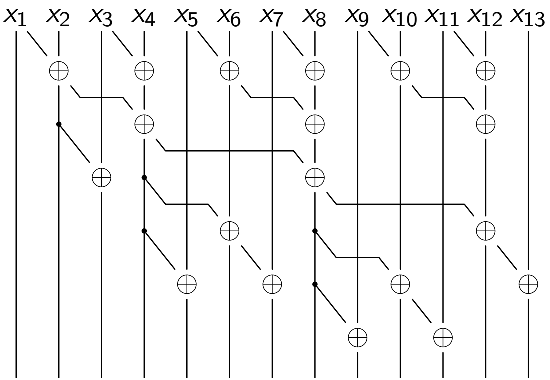
Brent & Kung's Method Visualised



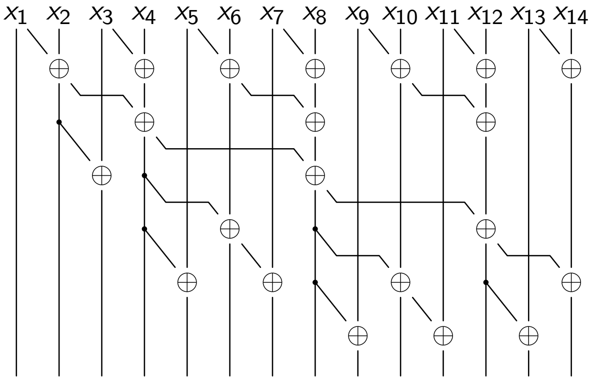
Brent & Kung's Method Visualised



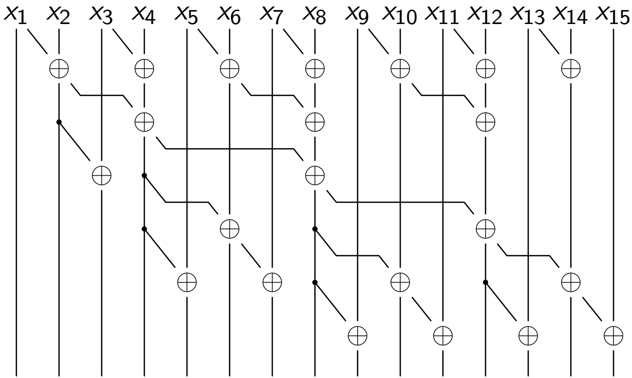
Brent & Kung's Method Visualised



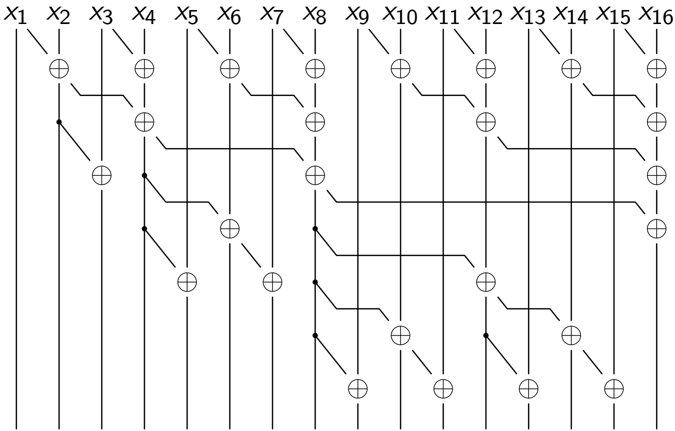
Brent & Kung's Method Visualised



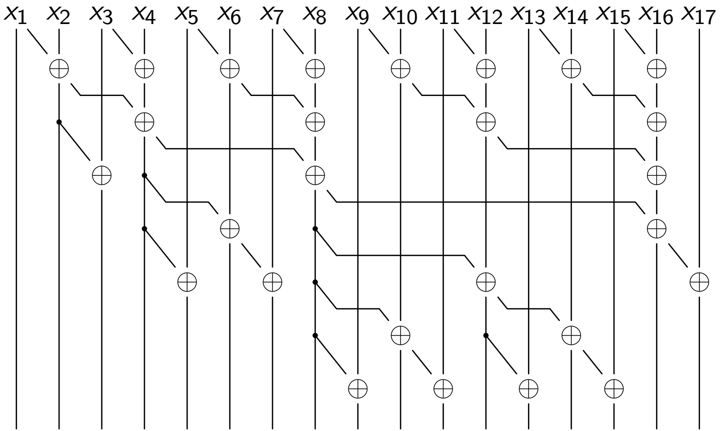
Brent & Kung's Method Visualised



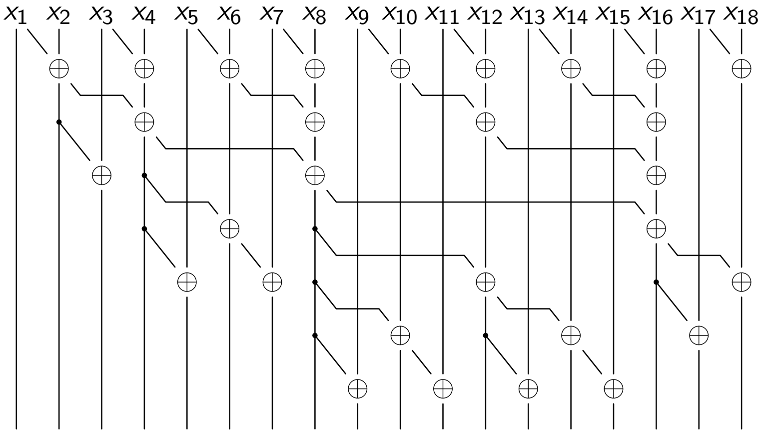
Brent & Kung's Method Visualised



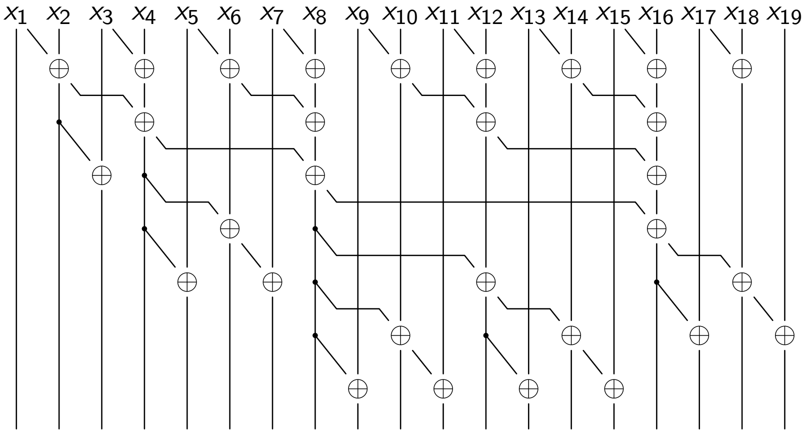
Brent & Kung's Method Visualised



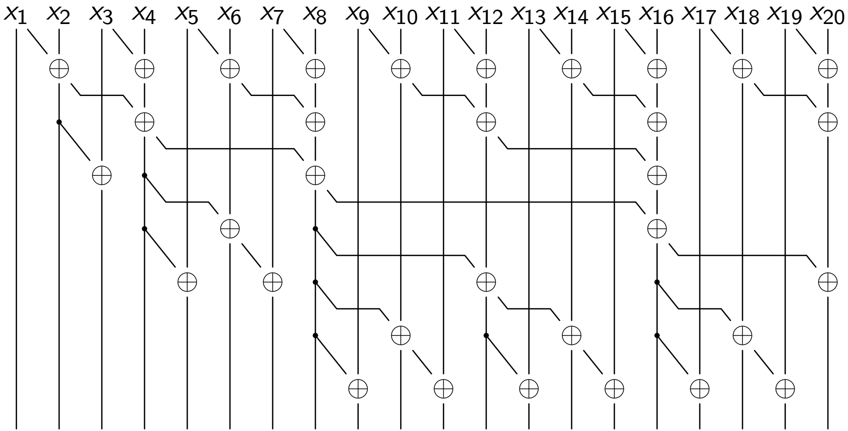
Brent & Kung's Method Visualised



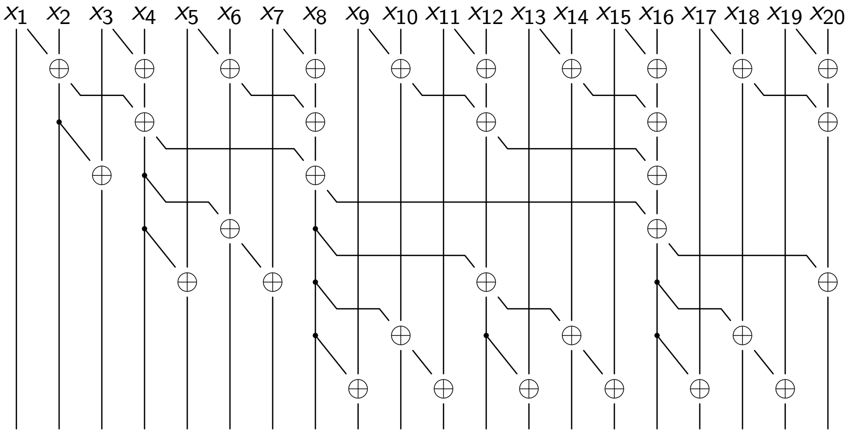
Brent & Kung's Method Visualised



Brent & Kung's Method Visualised



Brent & Kung's Method Visualised



Wanted: reasoning principles, verification techniques, systematic testing approach, ...

Investigating Particular Instances Only

Knuth's 0-1-Principle

If a comparison-swap algorithm sorts correctly on the Booleans, it does so on arbitrary totally ordered value sets.

Investigating Particular Instances Only

Knuth's 0-1-Principle

If a comparison-swap algorithm sorts correctly on the Booleans, it does so on arbitrary totally ordered value sets.

A Knuth-like 0-1-Principle ?

If a parallel prefix algorithm is correct (for associative operations) on the Booleans, it is so on arbitrary value sets.

Investigating Particular Instances Only

Knuth's 0-1-Principle

If a comparison-swap algorithm sorts correctly on the Booleans, it does so on arbitrary totally ordered value sets.

A Knuth-like 0-1-Principle ?

If a parallel prefix algorithm is correct (for associative operations) on the Booleans, it is so on arbitrary value sets.

Unfortunately not !

A Knuth-like 0-1-2-Principle [V. 2008]

Given: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$
\begin{aligned}
&\texttt{scanl1}\ (\oplus)\ (x : xs) = go\ x\ xs \\
&\qquad \textbf{where}\ go\ x\ [\,] \qquad\quad = [x] \\
&\qquad\qquad\quad\ go\ x\ (y : ys) = x : (go\ (x \oplus y)\ ys)
\end{aligned}
$$

A Knuth-like 0-1-2-Principle [V. 2008]

Given: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$\begin{aligned}
&\texttt{scanl1}\ (\oplus)\ (x : xs) = go\ x\ xs \\
&\qquad \textbf{where}\ go\ x\ [\,] \qquad\quad = [x] \\
&\qquad\qquad\quad go\ x\ (y : ys) = x : (go\ (x \oplus y)\ ys)
\end{aligned}$$

$$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

A Knuth-like 0-1-2-Principle [V. 2008]

Given: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$
\begin{aligned}
&\texttt{scanl1}\ (\oplus)\ (x : xs) = go\ x\ xs\\
&\quad \textbf{where}\ go\ x\ [\,] \qquad\quad = [x]\\
&\qquad\qquad\ \ go\ x\ (y : ys) = x : (go\ (x \oplus y)\ ys)
\end{aligned}
$$

$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

data Three = Zero | One | Two

A Knuth-like 0-1-2-Principle [V. 2008]

Given: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$
\begin{aligned}
&\texttt{scanl1}\ (\oplus)\ (x : xs) = go\ x\ xs \\
&\qquad \textbf{where}\ go\ x\ [\,] \qquad\quad = [x] \\
&\qquad\qquad\quad\ go\ x\ (y : ys) = x : (go\ (x \oplus y)\ ys)
\end{aligned}
$$

$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

data Three = Zero | One | Two

Theorem: If for every $xs :: [\text{Three}]$ and associative $(\oplus) :: \text{Three} \to \text{Three} \to \text{Three}$,

$$\texttt{candidate}\ (\oplus)\ xs \quad = \quad \texttt{scanl1}\ (\oplus)\ xs\,,$$

then the same holds for every type τ, $xs :: [\tau]$, and associative $(\oplus) :: \tau \to \tau \to \tau$.

Why 0-1-2? And How?

A question: What can `candidate` $:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$
do, given an operation $\oplus$ and input list $[x_1, \ldots, x_n]$?

Why 0-1-2? And How?

A question: What can `candidate` $:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$ do, given an operation $\oplus$ and input list $[x_1, \ldots, x_n]$?

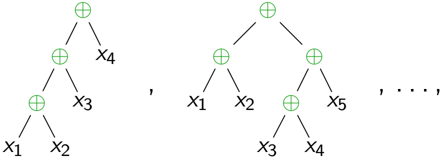
The answer: Create an output list consisting of expressions built from $\oplus$ and $x_1, \ldots, x_n$. Independently of the α-type !

Why 0-1-2? And How?

A question: What can `candidate` $:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$ do, given an operation $\oplus$ and input list $[x_1, \ldots, x_n]$?

The answer: Create an output list consisting of expressions built from $\oplus$ and $x_1, \ldots, x_n$. Independently of the α-type !

Among these expressions, there are good ones:

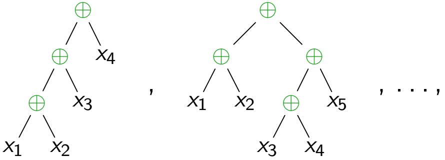


Why 0-1-2? And How?

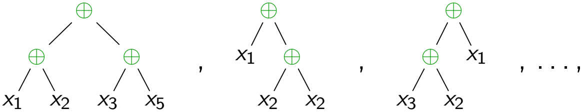
A question: What can `candidate` $:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$ do, given an operation $\oplus$ and input list $[x_1, \ldots, x_n]$?

The answer: Create an output list consisting of expressions built from $\oplus$ and $x_1, \ldots, x_n$. Independently of the α-type !

Among these expressions, there are good ones:

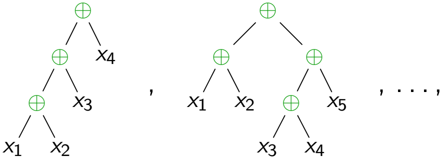


bad ones:

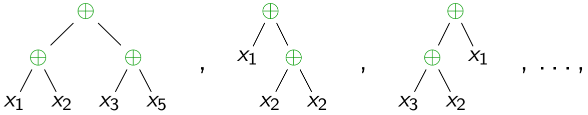


Why 0-1-2? And How?

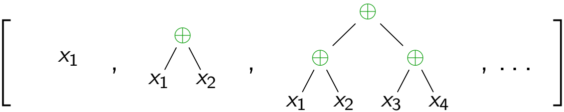
Among these expressions, there are good ones:



bad ones:



and ones in the wrong position:



That's How!

Let

$\oplus_1$	Zero	One	Two
Zero	Zero	One	Two
One	One	Two	Two
Two	Two	Two	Two

and

$\oplus_2$	Zero	One	Two
Zero	Zero	One	Two
One	One	One	Two
Two	Two	One	Two

That's How!

Let

$\oplus_1$	Zero	One	Two
Zero	Zero	One	Two
One	One	Two	Two
Two	Two	Two	Two

and

$\oplus_2$	Zero	One	Two
Zero	Zero	One	Two
One	One	One	Two
Two	Two	One	Two

If `candidate` $(\oplus_1)$ is correct on each list of the form

$$[(\text{Zero},)^* \ \text{One}\ (,\text{Zero})^* \ (,\text{Two})^*]$$

That's How!

Let

$\oplus_1$	Zero	One	Two
Zero	Zero	One	Two
One	One	Two	Two
Two	Two	Two	Two

and

$\oplus_2$	Zero	One	Two
Zero	Zero	One	Two
One	One	One	Two
Two	Two	One	Two

If `candidate` $(\oplus_1)$ is correct on each list of the form

$$[(\text{Zero},)^* \ \text{One}\ (,\text{Zero})^* \ (,\text{Two})^*]$$

and `candidate` $(\oplus_2)$ is correct on each list of the form

$$[(\text{Zero},)^* \ \text{One},\text{Two}\ (,\text{Zero})^*]$$

then `candidate` is correct for associative $\oplus$ at arbitrary type.

A Knuth-like 0-1-2-Principle [V. 2008]

Given: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$\begin{aligned}
&\texttt{scanl1}\ (\oplus)\ (x : xs) = go\ x\ xs\\
&\qquad \textbf{where}\ go\ x\ [\,] \qquad\quad = [x]\\
&\qquad\qquad\quad go\ x\ (y : ys) = x : (go\ (x \oplus y)\ ys)
\end{aligned}$$

$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

data Three = Zero | One | Two

Theorem: If for every xs :: [Three] and associative $(\oplus)$ :: Three $\to$ Three $\to$ Three,

$$\texttt{candidate}\ (\oplus)\ xs \quad = \quad \texttt{scanl1}\ (\oplus)\ xs\,,$$

then the same holds for every type τ, $xs :: [\tau]$, and associative $(\oplus) :: \tau \to \tau \to \tau$.

Using a Free Theorem (Generator)

Input: `candidate :: (a -> a -> a) -> [a] -> [a]`

Output:
```
forall t1,t2 in TYPES, f :: t1 -> t2.
  forall g :: t1 -> t1 -> t1.
    forall h :: t2 -> t2 -> t2.
      (forall x :: t1. forall y :: t1.
         f (g x y) = h (f x) (f y))
      ==> (forall z :: [t1].
                map f (candidate g z)
                 = candidate h (map f z))
```

Rephrased

For every choice of types τ_1, τ_2 and functions $f :: \tau_1 \to \tau_2$, $(\otimes) :: \tau_1 \to \tau_1 \to \tau_1$, and $(\oplus) :: \tau_2 \to \tau_2 \to \tau_2$:

$$\begin{tikzcd}
\tau_1 \times \tau_1 \arrow[r, "\otimes"] \arrow[d, "f \times f"'] \arrow[dr, phantom, "="] & \tau_1 \arrow[d, "f"] & & {[\tau_1]} \arrow[r, "\texttt{candidate}\ (\otimes)"] \arrow[d, "\texttt{map}\ f"'] \arrow[dr, phantom, "="] & {[\tau_1]} \arrow[d, "\texttt{map}\ f"] \\
\tau_2 \times \tau_2 \arrow[r, "\oplus"'] & \tau_2 \arrow[ur, phantom, "\Rightarrow"] & & {[\tau_2]} \arrow[r, "\texttt{candidate}\ (\oplus)"'] & {[\tau_2]}
\end{tikzcd}$$

A Knuth-like 0-1-2-Principle [V. 2008]

Given: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$\begin{aligned}
&\texttt{scanl1}\ (\oplus)\ (x : xs) = go\ x\ xs \\
&\qquad \textbf{where}\ go\ x\ [\,] \qquad\quad = [x] \\
&\qquad\qquad\quad go\ x\ (y : ys) = x : (go\ (x \oplus y)\ ys)
\end{aligned}$$

$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

data Three = Zero | One | Two

Theorem: If for every xs :: [Three] and associative $(\oplus)$ :: Three $\to$ Three $\to$ Three,

$$\texttt{candidate}\ (\oplus)\ xs \quad = \quad \texttt{scanl1}\ (\oplus)\ xs\,,$$

then the same holds for every type τ, $xs :: [\tau]$, and associative $(\oplus) :: \tau \to \tau \to \tau$.

Decomposing the 0-1-2-Principle

Proposition 1: If `candidate` $(\oplus_1)$ is correct on each list of the form $[(\mathrm{Zero},)^* \ \mathrm{One}\ (,\mathrm{Zero})^* \ (,\mathrm{Two})^*]$ and `candidate` $(\oplus_2)$ is correct on each list of the form $[(\mathrm{Zero},)^* \ \mathrm{One}, \mathrm{Two}\ (,\mathrm{Zero})^*]$, then for every $n \geq 0$,
`candidate` $(+\!\!+)$ $[[k] \mid k \leftarrow [0..n]] \ = \ [[0..k] \mid k \leftarrow [0..n]]$ $(*)$.

Decomposing the 0-1-2-Principle

Proposition 1: If `candidate` $(\oplus_1)$ is correct on each list of the form $[(\mathrm{Zero},)^* \ \mathrm{One}\ (,\mathrm{Zero})^* \ (,\mathrm{Two})^*]$ and `candidate` $(\oplus_2)$ is correct on each list of the form $[(\mathrm{Zero},)^* \ \mathrm{One}, \mathrm{Two}\ (,\mathrm{Zero})^*]$, then for every $n \geq 0$,
`candidate` $(+\!\!+)\ [[k] \mid k \leftarrow [0..n]] \ = \ [[0..k] \mid k \leftarrow [0..n]] \ \ (*)$.

Proposition 2: If for every $n \geq 0$, $(*)$ holds, then `candidate` is correct for associative $\oplus$ at arbitrary type.

Decomposing the 0-1-2-Principle

Proposition 1: If candidate $(\oplus_1)$ is correct on each list of the form $[(\text{Zero},)^* \ \text{One} \ (,\text{Zero})^* \ (,\text{Two})^*]$ and candidate $(\oplus_2)$ is correct on each list of the form $[(\text{Zero},)^* \ \text{One},\text{Two} \ (,\text{Zero})^*]$, then for every $n \geq 0$,
$\texttt{candidate}\ (+\!\!+)\ [[k] \mid k \leftarrow [0..n]] \ = \ [[0..k] \mid k \leftarrow [0..n]] \ \ (*)$.

Proposition 2: If for every $n \geq 0$, (*) holds, then candidate is correct for associative $\oplus$ at arbitrary type.

$$
\begin{tikzcd}
\tau_1 \times \tau_1 \arrow[r, "\otimes"] \arrow[d, "f \times f"'] \arrow[dr, phantom, "="] & \tau_1 \arrow[d, "f"] \\
\tau_2 \times \tau_2 \arrow[r, "\oplus"'] & \tau_2
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
{[\tau_1]} \arrow[r, "\texttt{candidate}\ (\otimes)"] \arrow[d, "\texttt{map}\ f"'] \arrow[dr, phantom, "="] & {[\tau_1]} \arrow[d, "\texttt{map}\ f"] \\
{[\tau_2]} \arrow[r, "\texttt{candidate}\ (\oplus)"'] & {[\tau_2]}
\end{tikzcd}
$$

Decomposing the 0-1-2-Principle

Proposition 1: If `candidate` $(\oplus_1)$ is correct on each list of the form $[(\mathrm{Zero},)^*\ \mathrm{One}\ (,\mathrm{Zero})^*\ (,\mathrm{Two})^*]$ and `candidate` $(\oplus_2)$ is correct on each list of the form $[(\mathrm{Zero},)^*\ \mathrm{One}, \mathrm{Two}\ (,\mathrm{Zero})^*]$, then for every $n \geq 0$,
`candidate` $(+\!\!+)\ [[k] \mid k \leftarrow [0..n]] \ = \ [[0..k] \mid k \leftarrow [0..n]]$ (*).

Proposition 2: If for every $n \geq 0$, (*) holds, then `candidate` is correct for associative $\oplus$ at arbitrary type.

$$\begin{tikzcd}
{[\mathrm{Int}] \times [\mathrm{Int}]} \arrow[r, "+\!\!+"] \arrow[d, "f \times f"'] \arrow[dr, phantom, "="] & {[\mathrm{Int}]} \arrow[d, "f"] \\
\tau_2 \times \tau_2 \arrow[r, "\oplus"'] & \tau_2
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
{[[k] \mid k \leftarrow [0..n]]} \arrow[d, phantom, "::"] & \\
{[[\mathrm{Int}]]} \arrow[r, "\mathtt{candidate}\ (+\!\!+)"] \arrow[d, "\mathtt{map}\ f"'] \arrow[dr, phantom, "="] & {[[\mathrm{Int}]]} \arrow[d, "\mathtt{map}\ f"] \\
{[\tau_2]} \arrow[r, "\mathtt{candidate}\ (\oplus)"'] & {[\tau_2]}
\end{tikzcd}$$

What Else?

- For parallel prefix computation, formalisation available in Isabelle/HOL [Böhme 2007].

What Else?

- For parallel prefix computation, formalisation available in Isabelle/HOL [Böhme 2007].

- There is still an interesting story to tell behind how "0-1-2", $\oplus_1$, $\oplus_2$, ... were found.

What Else?

- For parallel prefix computation, formalisation available in Isabelle/HOL [Böhme 2007].
- There is still an interesting story to tell behind how "0-1-2", $\oplus_1$, $\oplus_2$, ... were found.
- For which other algorithm classes can one play the same trick?

References I

G.E. Blelloch.
Prefix sums and their applications.
In J.H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 35–60. Morgan Kaufmann, 1993.

S. Böhme.
Much Ado about Two. Formal proof development.
In *The Archive of Formal Proofs*.
`http://afp.sf.net/entries/MuchAdoAboutTwo.shtml`, 2007.

A. Bove and T. Coquand.
Formalising bitonic sort in type theory.
In *Types for Proofs and Programs, TYPES 2004, Revised Selected Papers*, volume 3839 of *LNCS*, pages 82–97. Springer-Verlag, 2006.

References II

R.P. Brent and H.T. Kung.
The chip complexity of binary arithmetic.
In *ACM Symposium on Theory of Computing, Proceedings*, pages 190–200. ACM Press, 1980.

N.A. Day, J. Launchbury, and J.R. Lewis.
Logical abstractions in Haskell.
In *Haskell Workshop, Proceedings*, 1999.

P. Dybjer, Q. Haiyan, and M. Takeyama.
Verifying Haskell programs by combining testing, model checking and interactive theorem proving.
Information & Software Technology, 46(15):1011–1025, 2004.

References III

D.E. Knuth.
The Art of Computer Programming, volume 3: Sorting and Searching.
Addison-Wesley, 1973.

J.C. Reynolds.
Types, abstraction and parametric polymorphism.
In *Information Processing, Proceedings*, pages 513–523.
Elsevier Science Publishers B.V., 1983.

M. Sheeran.
Searching for prefix networks to fit in a context using a lazy functional programming language.
Hardware Design and Functional Languages, 2007.

References IV

J. Sklansky.
Conditional-sum addition logic.
IRE Transactions on Electronic Computers, EC-9(6):226–231, 1960.

J. Voigtländer.
Much ado about two: A pearl on parallel prefix computation.
In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008.

P. Wadler.
Theorems for free!
In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.

Excerpt from Formal Proof — Where Associativity is Used

Let $xs :: [\tau_2]$ with length $(n + 1)$.

Excerpt from Formal Proof — Where Associativity is Used

Let $xs :: [\tau_2]$ with length $(n+1)$. Then for

$$f = \texttt{foldl1}\ (\oplus) \circ \texttt{map}\ (xs\,!!)$$

the precondition of

$$\begin{tikzcd}
{[\mathrm{Int}] \times [\mathrm{Int}]} \arrow[r, "++"] \arrow[d, "f \times f"'] \arrow[dr, phantom, "="] & {[\mathrm{Int}]} \arrow[d, "f"] \\
\tau_2 \times \tau_2 \arrow[r, "\oplus"'] & \tau_2
\end{tikzcd}
\quad\Rightarrow\quad
\begin{tikzcd}
{[[k] \mid k \leftarrow [0..n]] :: [[\mathrm{Int}]]} \arrow[r, "\texttt{candidate}\ (++)"] \arrow[d, "\texttt{map}\ f"'] \arrow[dr, phantom, "="] & {[[\mathrm{Int}]]} \arrow[d, "\texttt{map}\ f"] \\
{[\tau_2]} \arrow[r, "\texttt{candidate}\ (\oplus)"'] & {[\tau_2]}
\end{tikzcd}$$

Excerpt from Formal Proof — Where Associativity is Used

Let $xs :: [\tau_2]$ with length $(n+1)$. Then for

$$f = \texttt{foldl1}\ (\oplus) \circ \texttt{map}\ (xs\,!!)$$

the precondition of

$$\begin{tikzcd}
{[\mathrm{Int}] \times [\mathrm{Int}]} \arrow[r, "+\!\!+"] \arrow[d, "f \times f"'] \arrow[dr, phantom, "="] & {[\mathrm{Int}]} \arrow[d, "f"] \\
\tau_2 \times \tau_2 \arrow[r, "\oplus"'] & \tau_2
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
{[[k] \mid k \leftarrow [0..n]]} \arrow[d, phantom, "::"] & \\
{[[\mathrm{Int}]]} \arrow[r, "\texttt{candidate}\ (+\!\!+)"] \arrow[d, "\texttt{map}\ f"'] \arrow[dr, phantom, "="] & {[[\mathrm{Int}]]} \arrow[d, "\texttt{map}\ f"] \\
{[\tau_2]} \arrow[r, "\texttt{candidate}\ (\oplus)"'] & {[\tau_2]}
\end{tikzcd}$$

is fulfilled, provided $\oplus$ is associative.

Excerpt from Formal Proof — Where Associativity is Used

Let $xs :: [\tau_2]$ with length $(n+1)$. Then for

$$f = \texttt{foldl1}\ (\oplus) \circ \texttt{map}\ (xs\,!!)$$

the precondition of

$$\begin{tikzcd}
{[\mathrm{Int}] \times [\mathrm{Int}]} \arrow[r, "+\!\!+"] \arrow[d, "f \times f"'] \arrow[dr, phantom, "="] & {[\mathrm{Int}]} \arrow[d, "f"] \\
\tau_2 \times \tau_2 \arrow[r, "\oplus"'] & \tau_2
\end{tikzcd}
\Rightarrow
\begin{tikzcd}
{[[k] \mid k \leftarrow [0..n]]} \arrow[d, phantom, "::"] & \\
{[[\mathrm{Int}]]} \arrow[r, "\texttt{candidate}\ (+\!\!+)"] \arrow[d, "\texttt{map}\ f"'] \arrow[dr, phantom, "="] & {[[\mathrm{Int}]]} \arrow[d, "\texttt{map}\ f"] \\
{[\tau_2]} \arrow[r, "\texttt{candidate}\ (\oplus)"'] & {[\tau_2]}
\end{tikzcd}$$

is fulfilled, provided $\oplus$ is associative.
Hence, then:

$$\begin{aligned}
& \texttt{map}\ f\ (\texttt{candidate}\ (+\!\!+)\ [[k] \mid k \leftarrow [0..n]]) \\
= \quad & \texttt{candidate}\ (\oplus)\ (\texttt{map}\ f\ [[k] \mid k \leftarrow [0..n]])
\end{aligned}$$

Excerpt from Formal Proof — Where Associativity is Used

Let $xs :: [\tau_2]$ with length $(n+1)$. Then for

$$f = \texttt{foldl1}\ (\oplus) \circ \texttt{map}\ (xs\,!!)$$

the precondition of

$$\begin{tikzcd}
{[\mathrm{Int}] \times [\mathrm{Int}]} \arrow[r, "+\!\!+"] \arrow[d, "f \times f"'] \arrow[dr, phantom, "="] & {[\mathrm{Int}]} \arrow[d, "f"] \\
\tau_2 \times \tau_2 \arrow[r, "\oplus"'] & \tau_2
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
{[[k] \mid k \leftarrow [0..n]]} \arrow[d, phantom, "::"] & \\
{[[\mathrm{Int}]]} \arrow[r, "\texttt{candidate}\ (+\!\!+)"] \arrow[d, "\texttt{map}\ f"'] \arrow[dr, phantom, "="] & {[[\mathrm{Int}]]} \arrow[d, "\texttt{map}\ f"] \\
{[\tau_2]} \arrow[r, "\texttt{candidate}\ (\oplus)"'] & {[\tau_2]}
\end{tikzcd}$$

is fulfilled, provided $\oplus$ is associative.
Hence, then:

$$\begin{aligned}
& \texttt{map}\ f\ (\texttt{candidate}\ (+\!\!+)\ [[k] \mid k \leftarrow [0..n]]) \\
= \; & \texttt{candidate}\ (\oplus)\ (\texttt{map}\ f\ [[k] \mid k \leftarrow [0..n]]) \\
= \; & \texttt{candidate}\ (\oplus)\ xs
\end{aligned}$$