Knuth's 0-1-Principle

Janis Voigtländer

May 19th, 2020

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

Many Solutions:

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

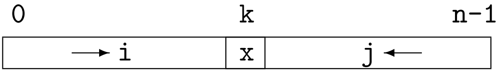
Quicksort

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.

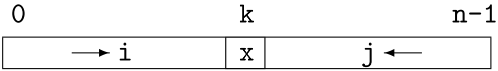
Realisation:



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:



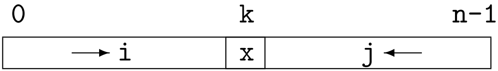
Example:

2	15	7	9	12	4	11

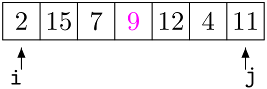
Quicksort

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:



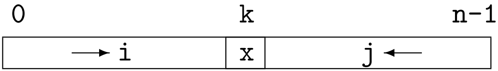
Example:



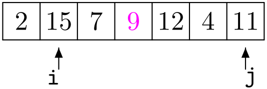
Quicksort

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:



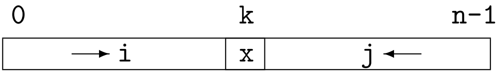
Example:



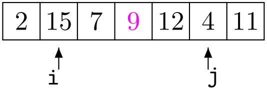
Quicksort

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:



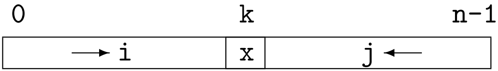
Example:



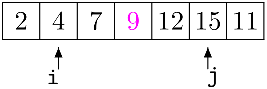
Quicksort

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:



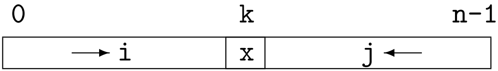
Example:



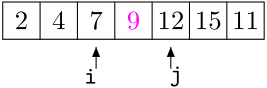
Quicksort

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:



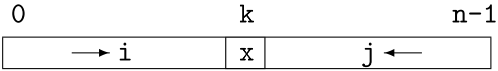
Example:



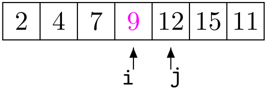
Quicksort

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:



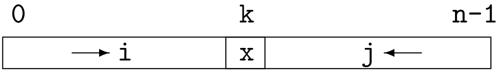
Example:



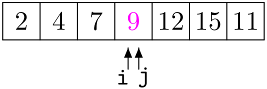
Quicksort

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:



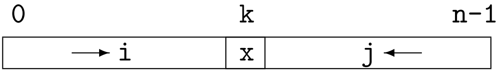
Example:



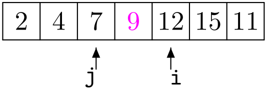
Quicksort

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:



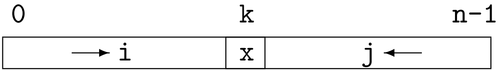
Example:



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:



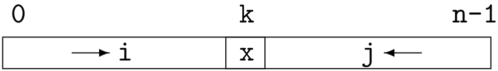
Example:

2	4	7	9	12	15	11

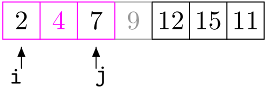
Quicksort

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:



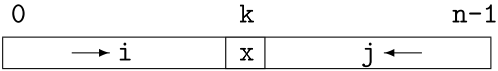
Example:



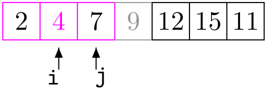
Quicksort

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:



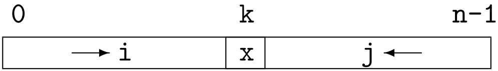
Example:



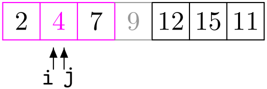
Quicksort

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:



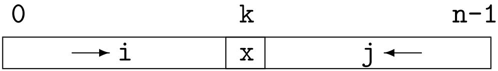
Example:



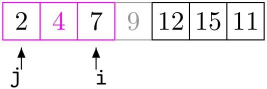
Quicksort

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:



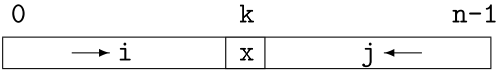
Example:



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):

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

- 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):

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

- 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:

$$cswap :: (\tau, \tau) \to (\tau, \tau)$$

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:

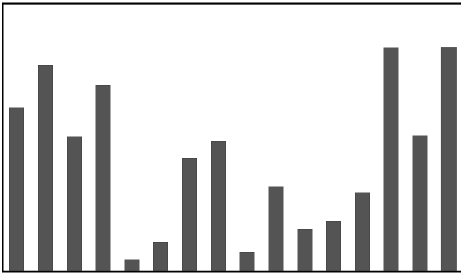
$$\mathit{cswap} :: (\tau, \tau) \rightarrow (\tau, \tau)$$

Bitonic Sort

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

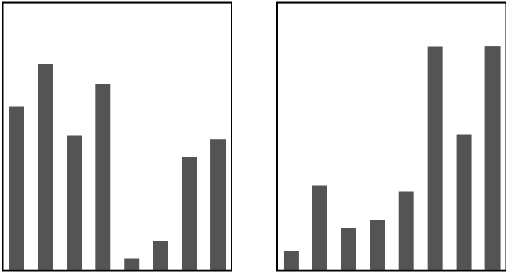
Bitonic Sort

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



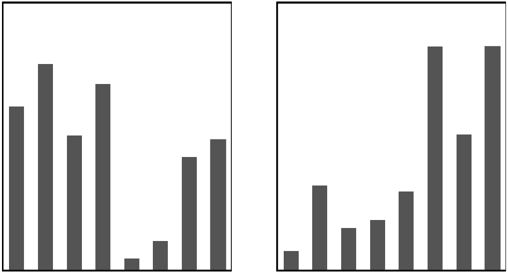
Bitonic Sort

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



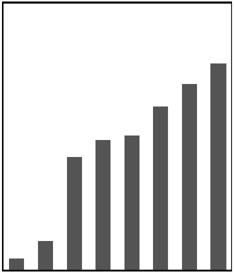
Bitonic Sort

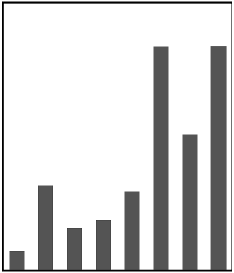
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively



Bitonic Sort

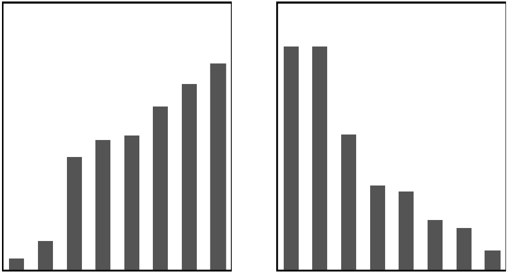
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively





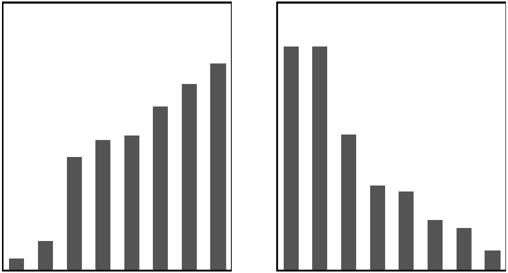
Bitonic Sort

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



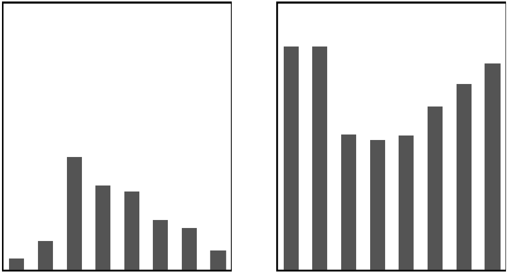
Bitonic Sort

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.



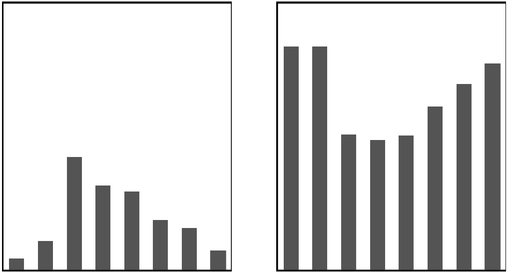
Bitonic Sort

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.



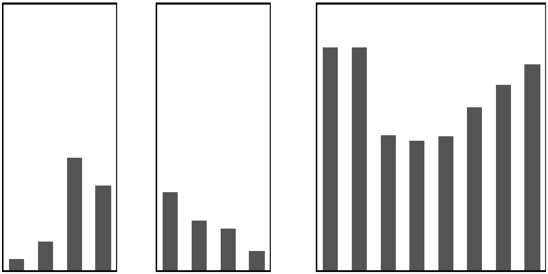
Bitonic Sort

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.



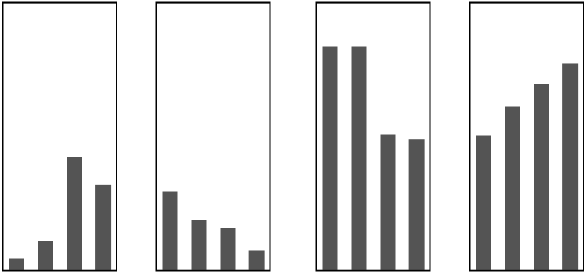
Bitonic Sort

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.



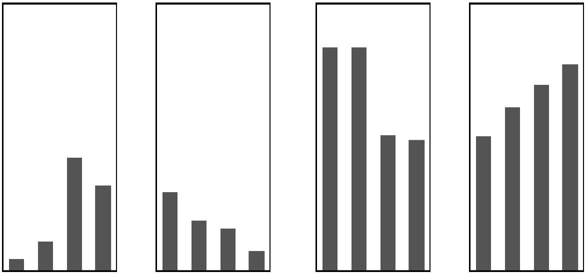
Bitonic Sort

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.



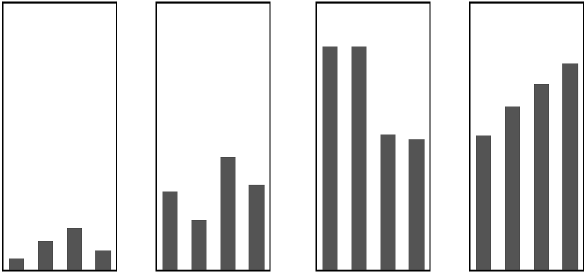
Bitonic Sort

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.



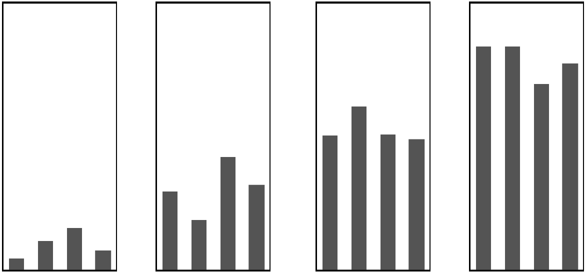
Bitonic Sort

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.



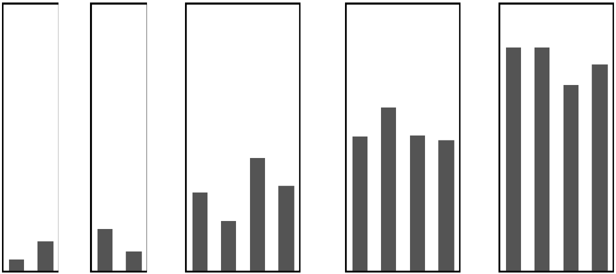
Bitonic Sort

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.



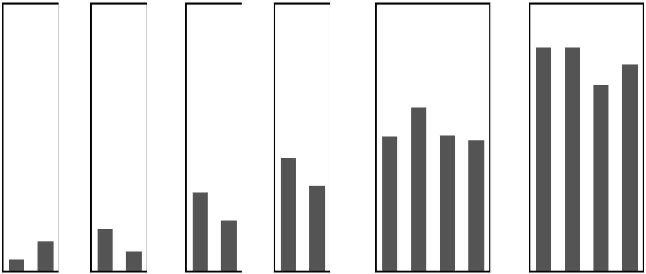
Bitonic Sort

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.



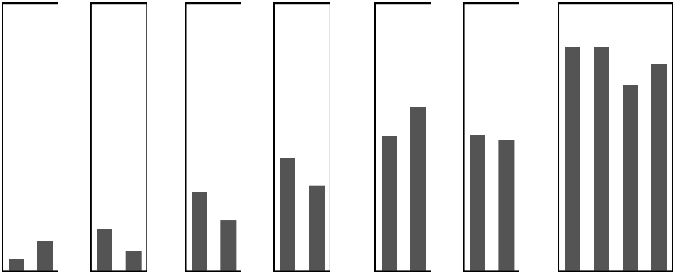
Bitonic Sort

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.



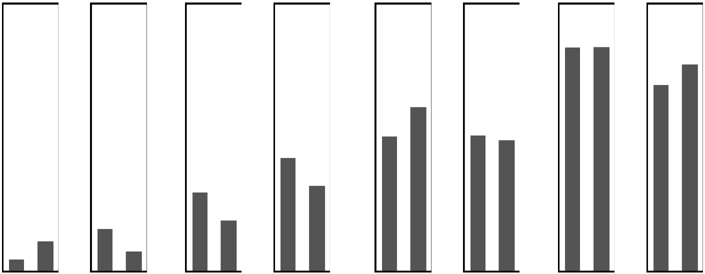
Bitonic Sort

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.



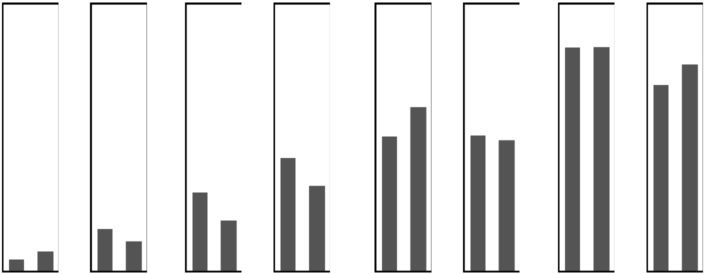
Bitonic Sort

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.



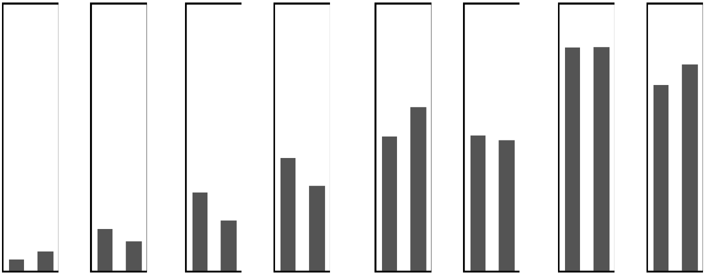
Bitonic Sort

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.



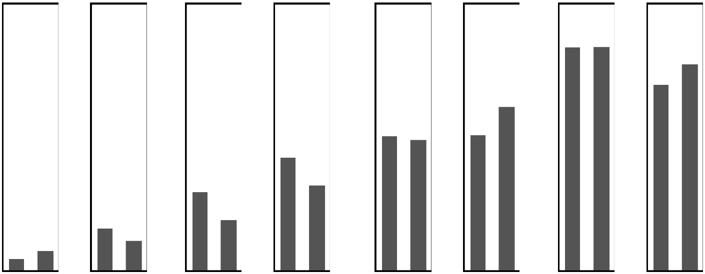
Bitonic Sort

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.



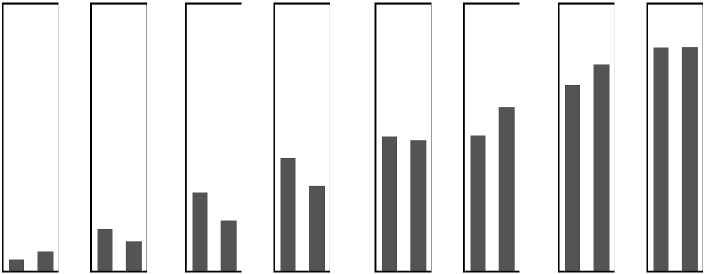
Bitonic Sort

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.



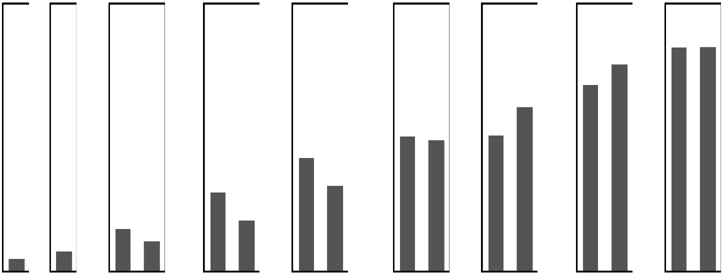
Bitonic Sort

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.



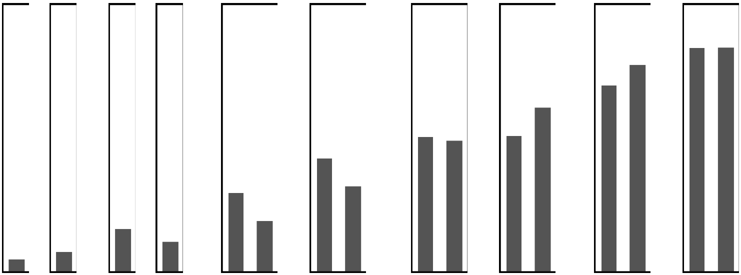
Bitonic Sort

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.



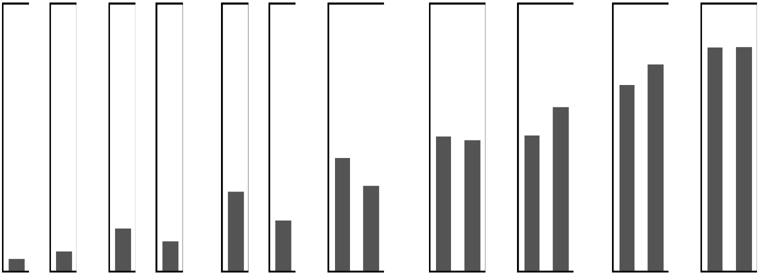
Bitonic Sort

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.



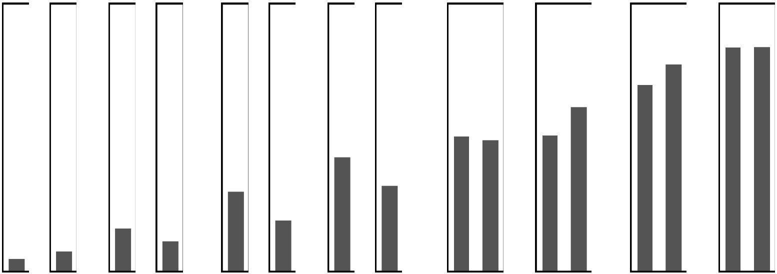
Bitonic Sort

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.



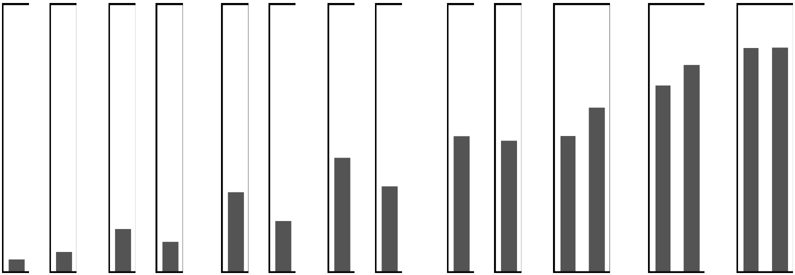
Bitonic Sort

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.



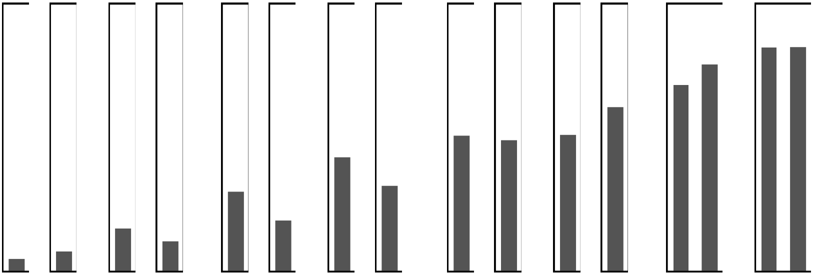
Bitonic Sort

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.



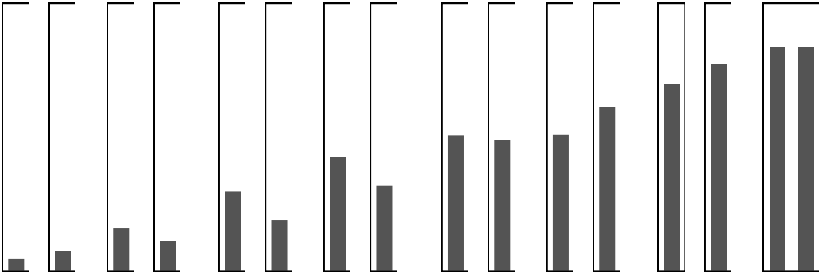
Bitonic Sort

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.



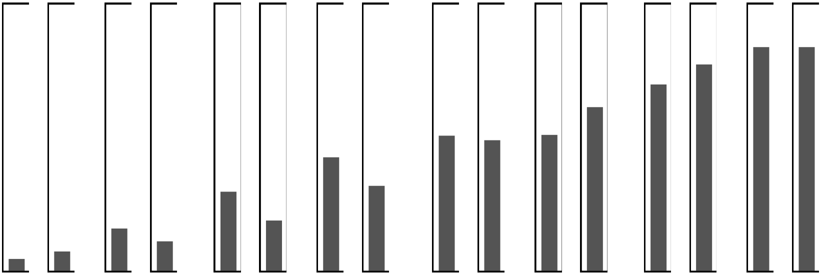
Bitonic Sort

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.



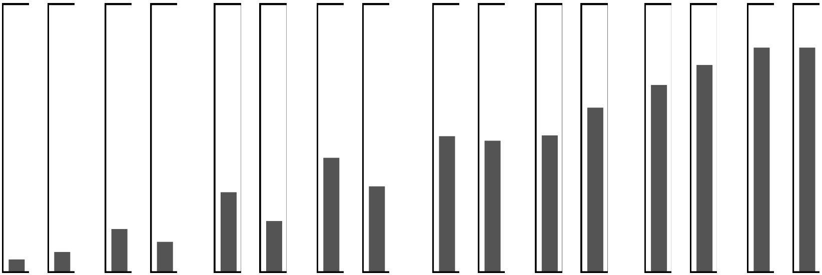
Bitonic Sort

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.



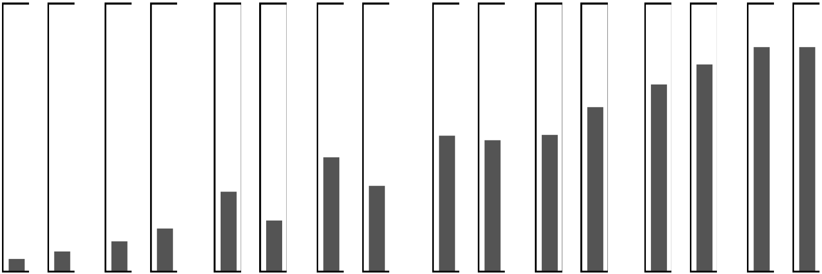
Bitonic Sort

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.



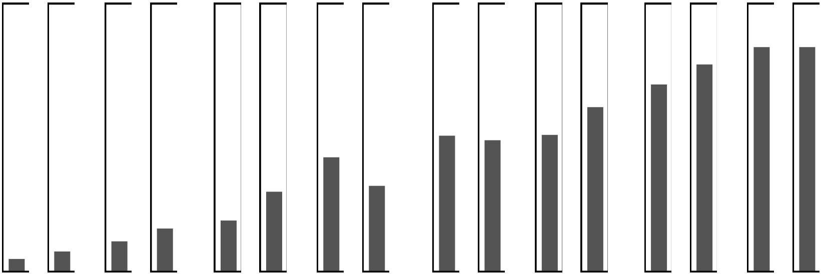
Bitonic Sort

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.



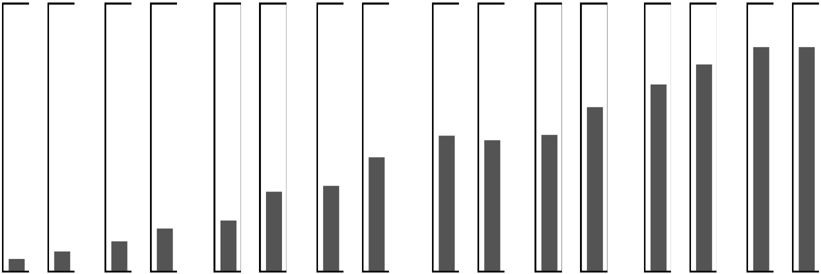
Bitonic Sort

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.



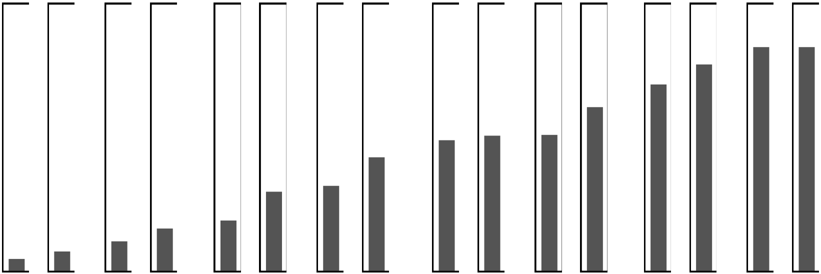
Bitonic Sort

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.



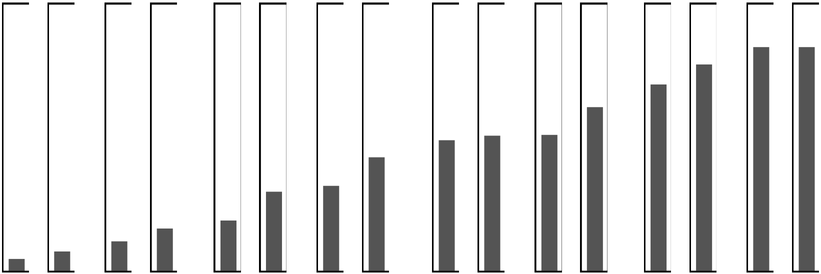
Bitonic Sort

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.



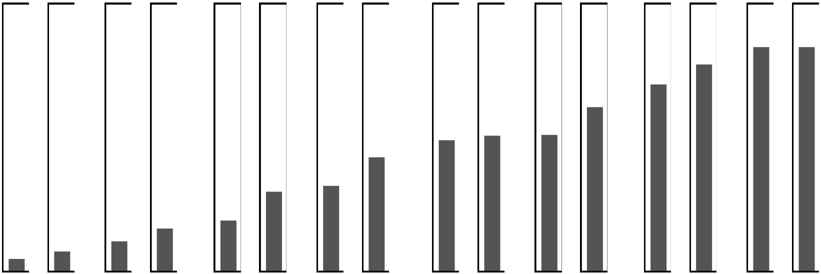
Bitonic Sort

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.



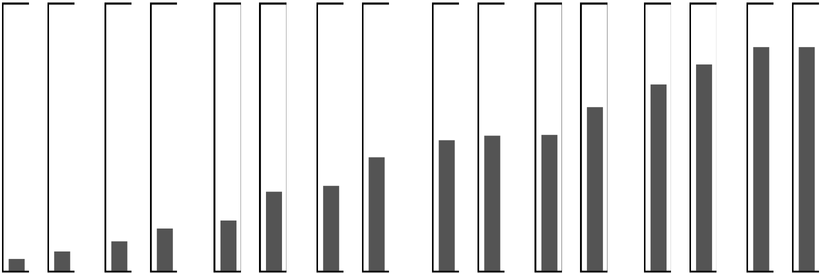
Bitonic Sort

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.



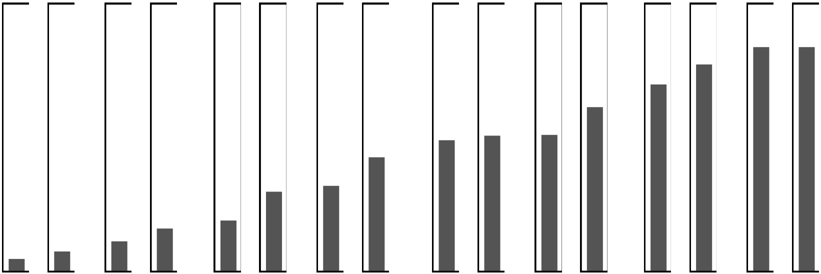
Bitonic Sort

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.



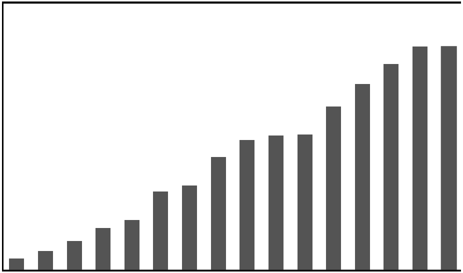
Bitonic Sort

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

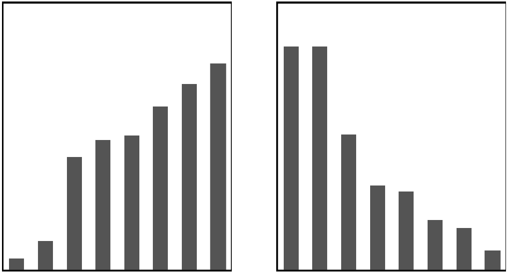
Bitonic Sort

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

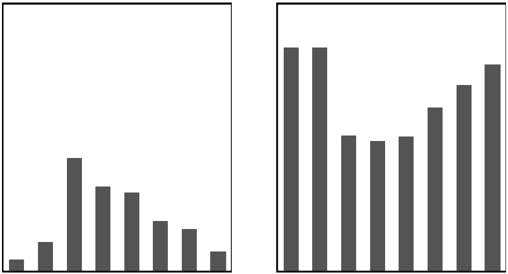
Bitonic Sort

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.



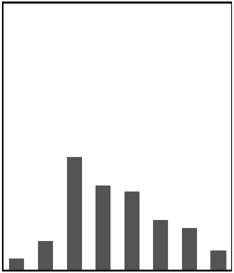
Bitonic Sort

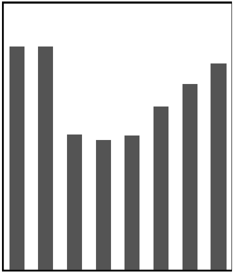
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.



Bitonic Sort

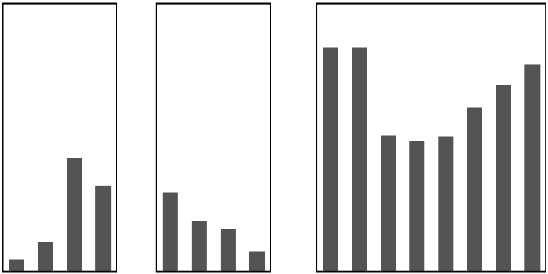
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.





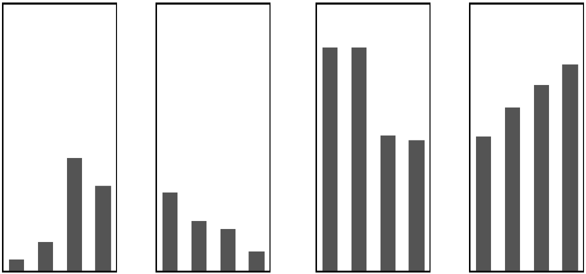
Bitonic Sort

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.



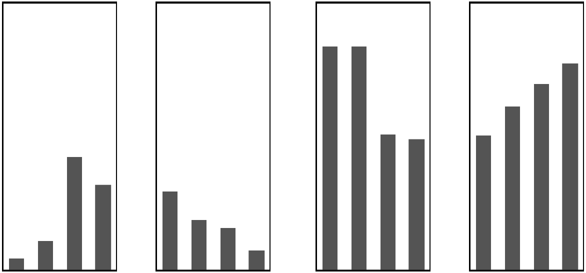
Bitonic Sort

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.



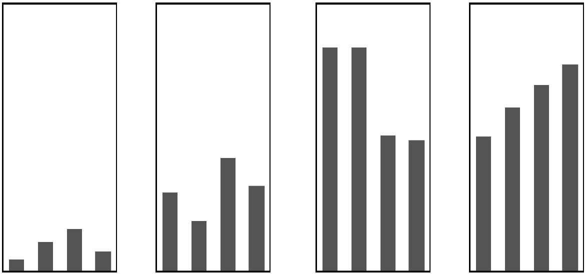
Bitonic Sort

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.



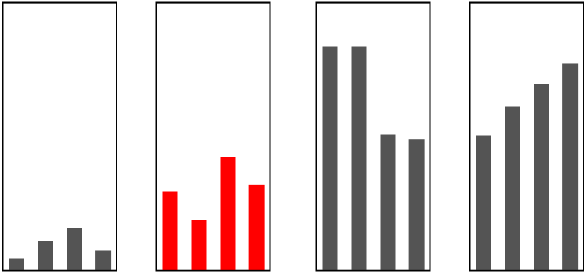
Bitonic Sort

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.

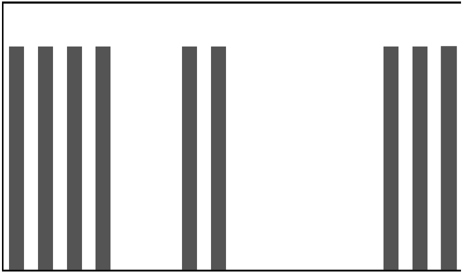


Knuth's 0-1-Principle [Knuth 1973]

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

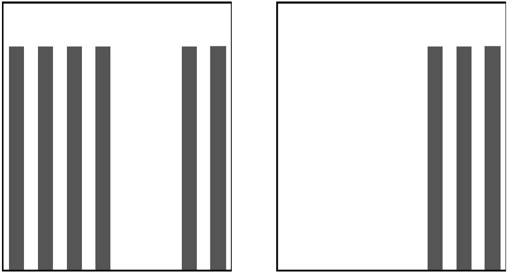
Bitonic Sort

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



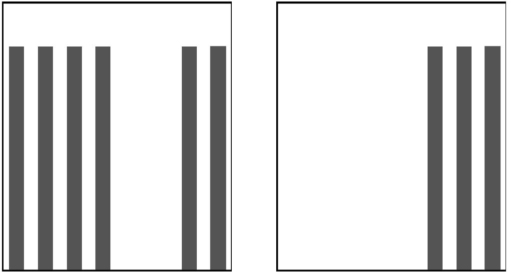
Bitonic Sort

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



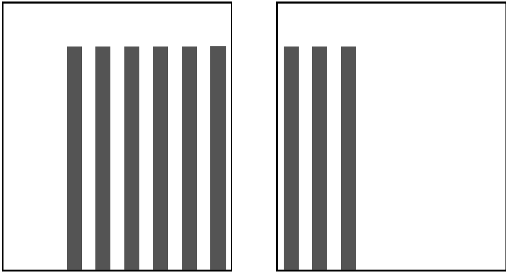
Bitonic Sort

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



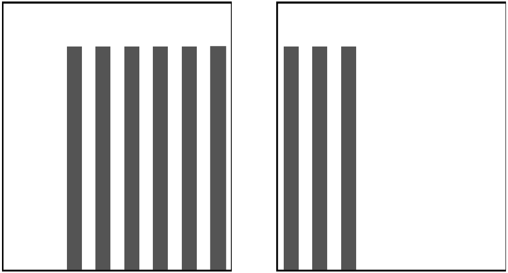
Bitonic Sort

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



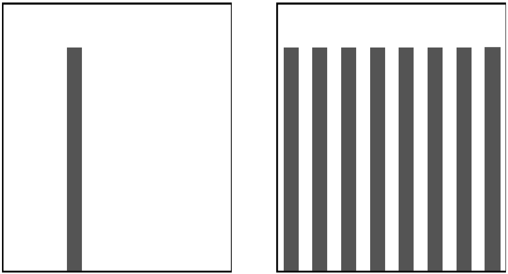
Bitonic Sort

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.



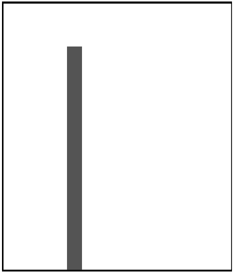
Bitonic Sort

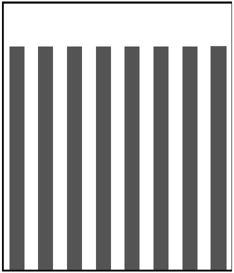
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.



Bitonic Sort

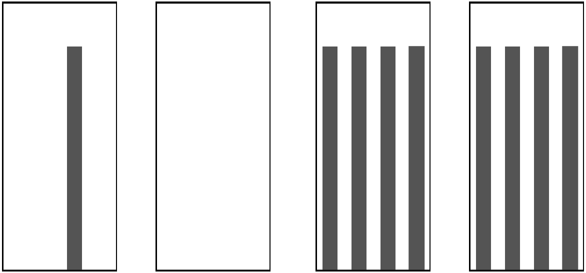
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.





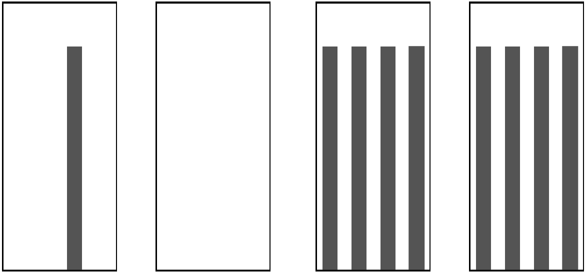
Bitonic Sort

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.



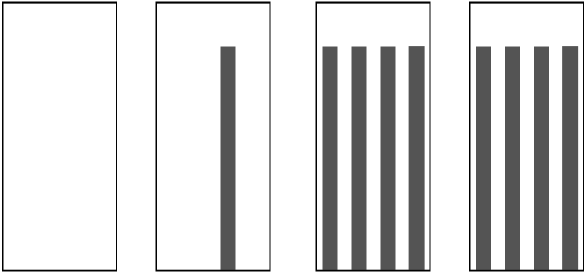
Bitonic Sort

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.



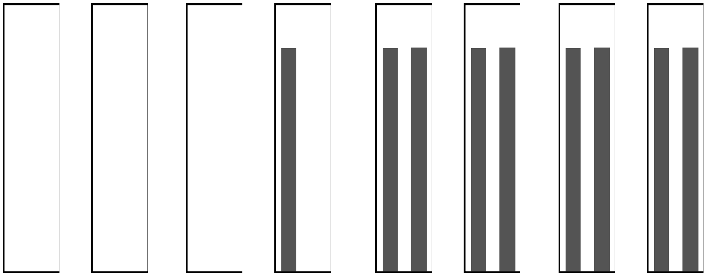
Bitonic Sort

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.



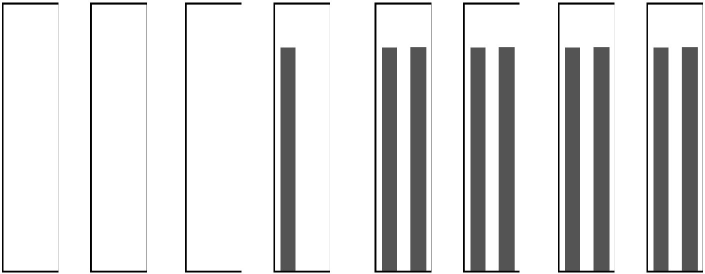
Bitonic Sort

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.



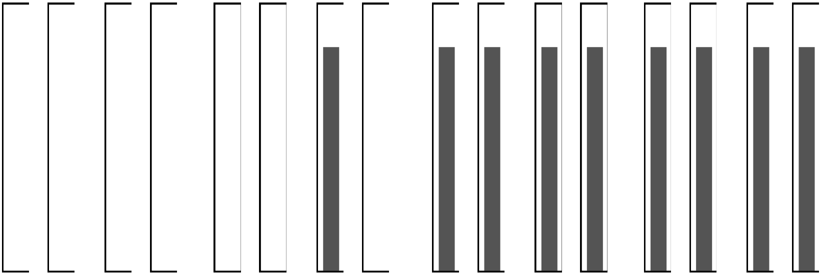
Bitonic Sort

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.



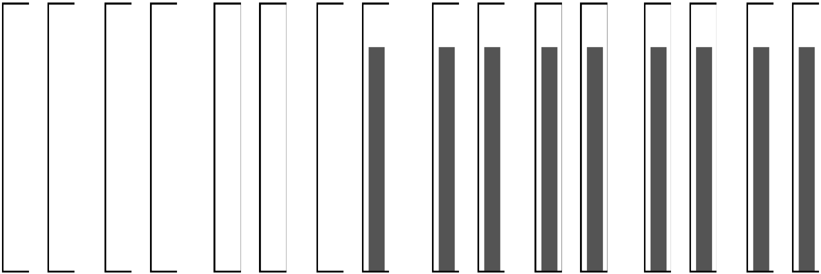
Bitonic Sort

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.



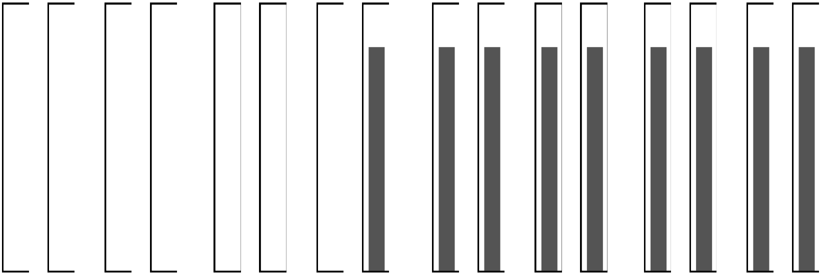
Bitonic Sort

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.



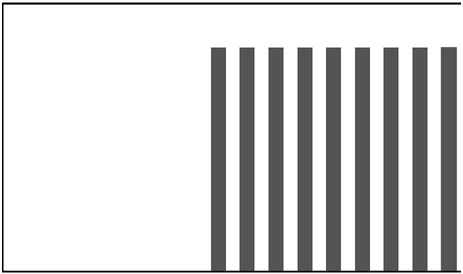
Bitonic Sort

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.



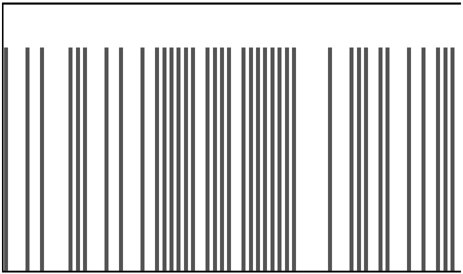
Bitonic Sort

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.



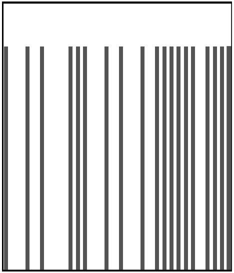
Bitonic Sort

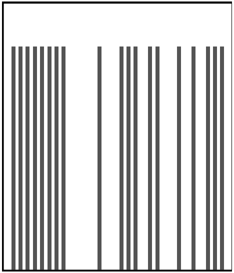
1. Split the input list into two sublists of equal length.



Bitonic Sort

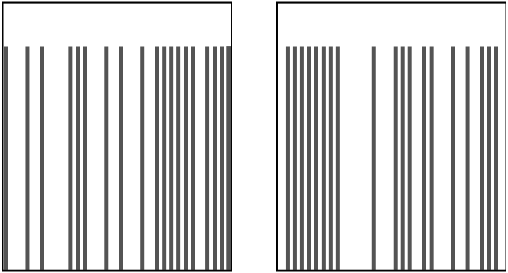
1. Split the input list into two sublists of equal length.





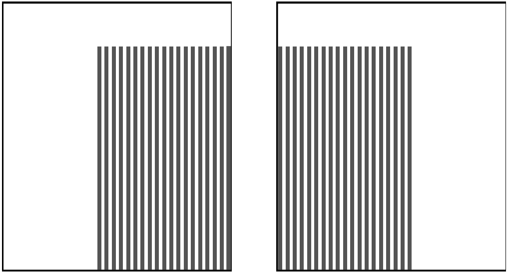
Bitonic Sort

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



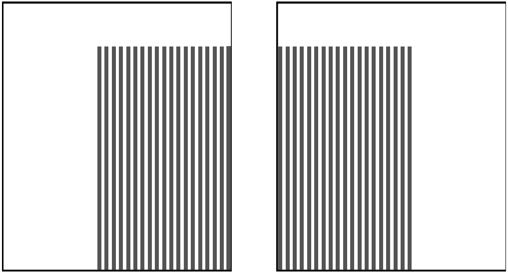
Bitonic Sort

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



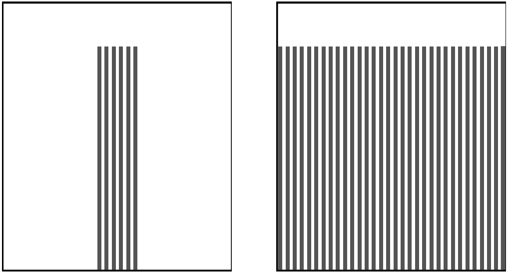
Bitonic Sort

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.



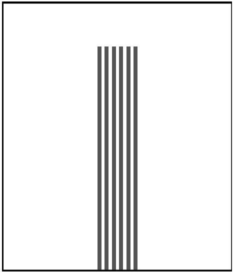
Bitonic Sort

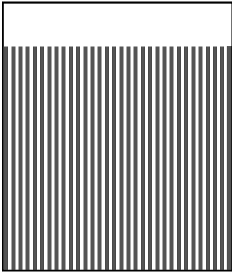
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.



Bitonic Sort

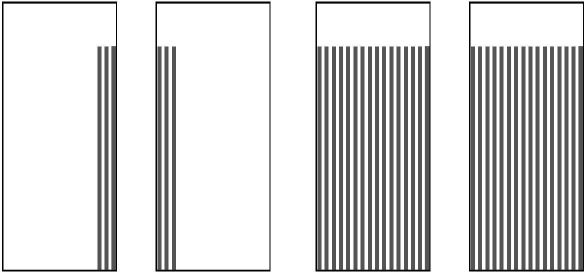
1. Split the input list into two sublists of equal length.
2. Sort the two sublists recursively, the second one in reverse order.
3. Merge the sorted sublists as follows:
   3.1 Apply *cswap* to pairs of elements at corresponding positions.
   3.2 Split each of the resulting lists in the middle.





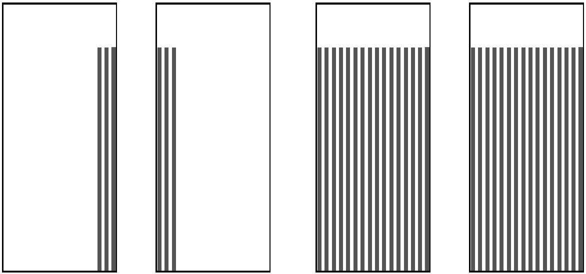
Bitonic Sort

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.



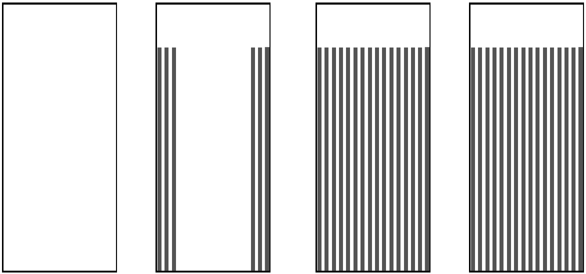
Bitonic Sort

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.



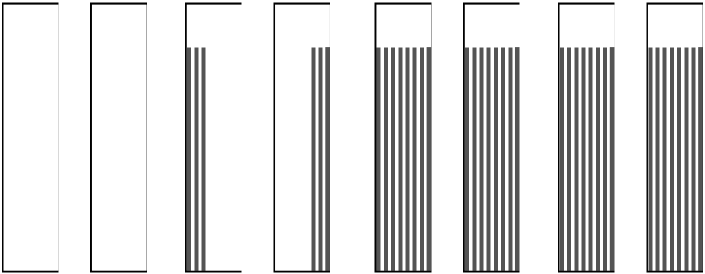
Bitonic Sort

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.



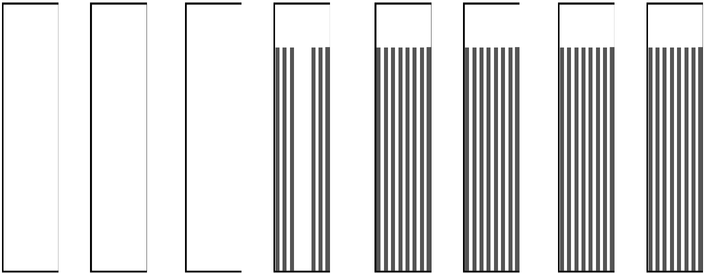
Bitonic Sort

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.



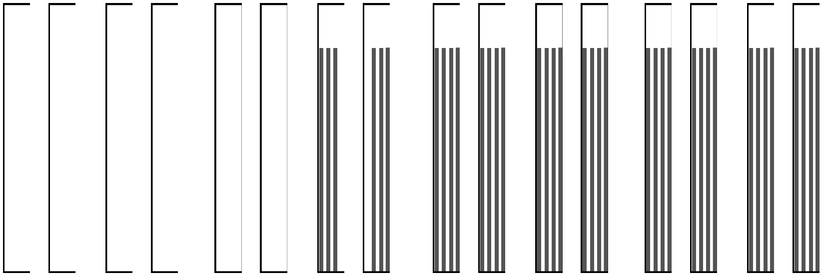
Bitonic Sort

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.



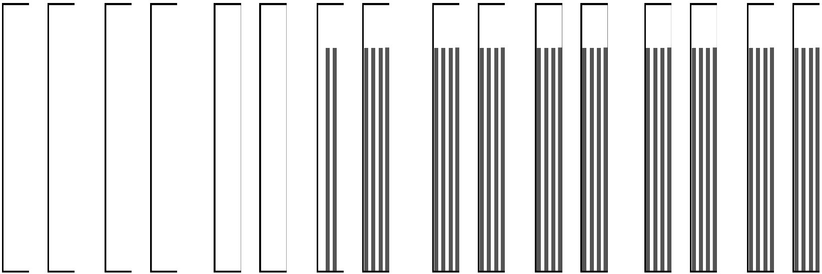
Bitonic Sort

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

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

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

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

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

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

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

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

If for every xs :: [Bool], `sort` g xs gives the correct result, then for every xs :: [Int], `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

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

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

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

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

Using a Free Theorems Generator

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

Using a Free Theorems Generator

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

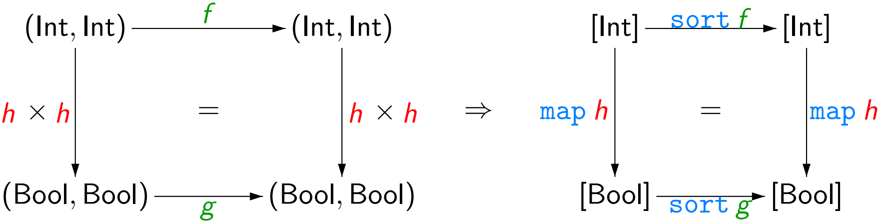
Output:

```
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)}
```

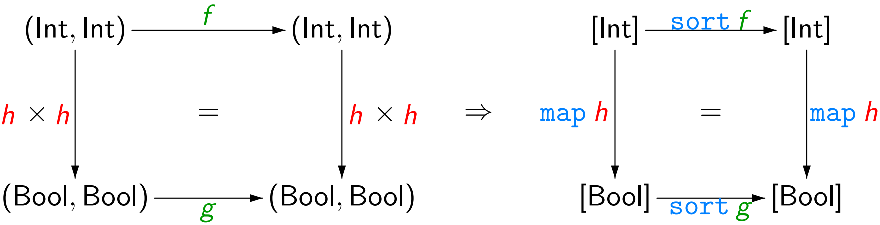
More Specific (and Intuitive)

For every `sort` :: $((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$,
f :: (Int, Int) $\to$ (Int, Int), g :: (Bool, Bool) $\to$ (Bool, Bool), and
h :: Int $\to$ Bool:



More Specific (and Intuitive)

For every `sort` :: $((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$,
f :: (Int, Int) $\to$ (Int, Int), g :: (Bool, Bool) $\to$ (Bool, Bool), and
h :: Int $\to$ Bool:



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

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

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 $\neg P(us, vs)$.

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

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:

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

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

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

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]"

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 $\neg Q(vs)$.

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.

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]"

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:

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

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

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:

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

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

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

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

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

If for every xs :: [Bool], $\texttt{sort}\ g\ xs$ gives the correct result, then for every xs :: [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 *cswap* $:: (\alpha, \alpha) \rightarrow (\alpha, \alpha)$ in the case of sorting.

References I

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.

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

References II

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