Massively Parallel Algorithms
Parallel Sorting

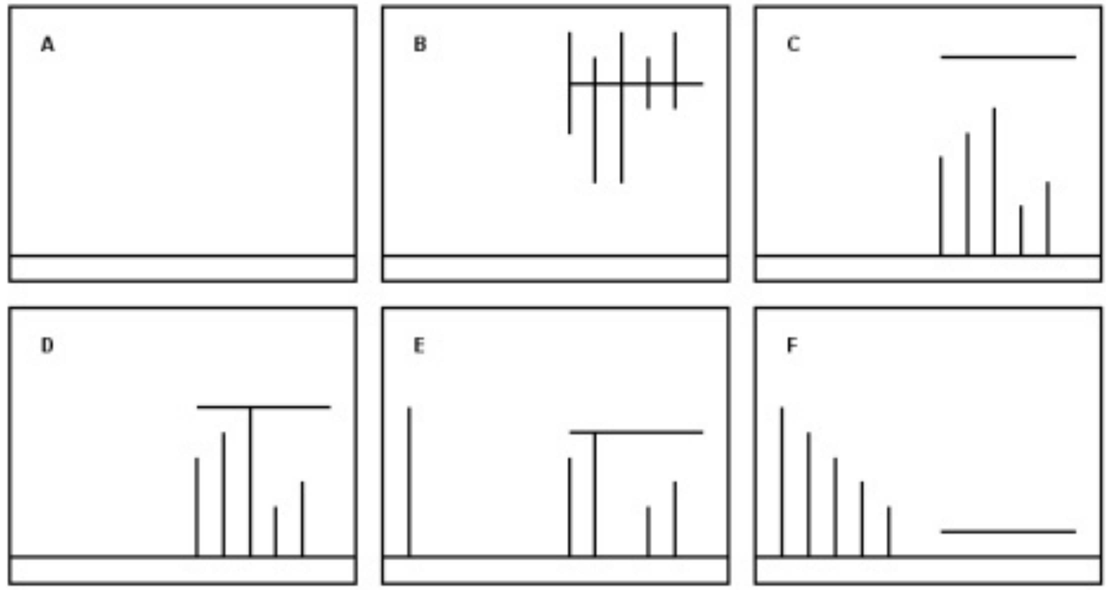
G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

Sorting using Spaghetti in O(1) (?)

- Is O(n) really the lower bound for sorting?
- Consider the following thought experiment:

2. For each number x in the list, cut a spaghetto to length x
   → list = bundle of spaghetti & unary repr.
3. Hold the spaghetti loosely in your hand and tap them on the kitchen table → takes O(1) !
4. Lower your other hand from above until it meets with a spaghetto — this one is clearly the longest
5. Remove this spaghetto and insert it into the front of the output list
6. Repeat

- If we could use this *mechanical* computer, then sorting would be O(1), unless you count the extraction, too :-)



Difficulties With Parallel Implementation of Standard Sequential Algorithms

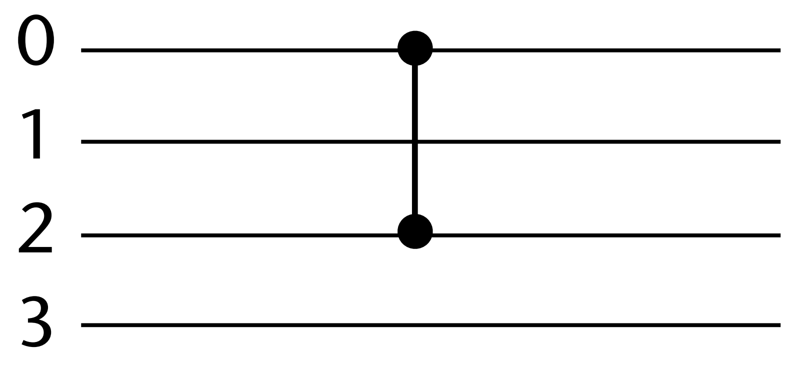
- Insertion sort: considers only one element at a time
- Quicksort:
  - Yes, some parallelism at lower levels of the recursion tree is possible
  - But, would need the *median* as a pivot element $\longrightarrow$ hard to find
  - Otherwise, random pivot element causes very different sub-array sizes
- Heapsort:
  - Only one element at a time
  - Heap (= recursive data structure) is difficult on massively-parallel architecture
- Radix sort:
  - Yes, we've seen that already, works well
  - But, can handle only fixed-length numbers

Assumptions

- In this chapter, we will always assume that $n = 2^k$
- Elements can have any type, for which there is a *comparison* operator

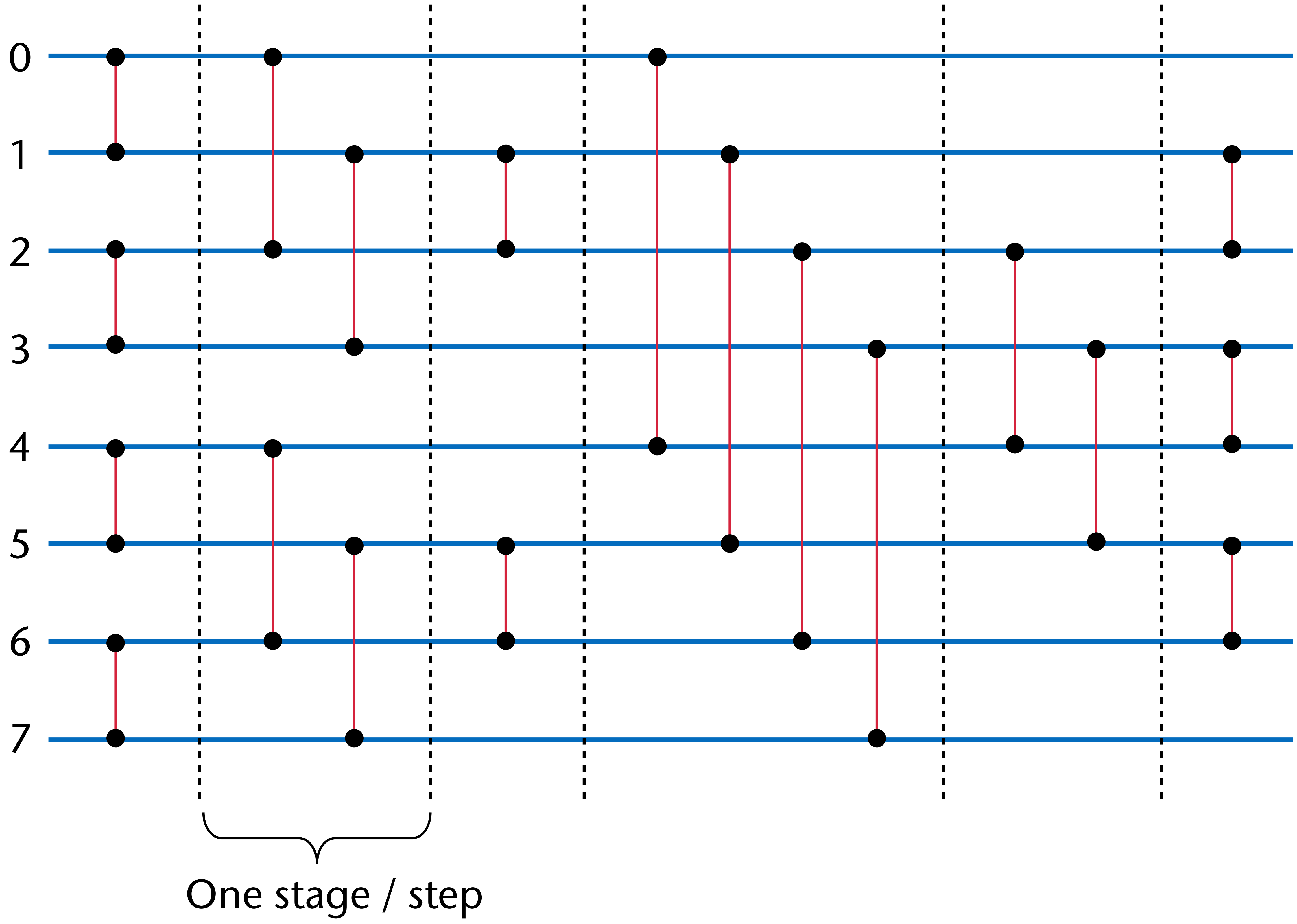
Sorting Networks

- Informal definition of comparator networks:
  - Consist of a bundle of "wires"
  - Each wire i carries a data element D_i (e.g., floats) from left to right
  - Two wires can be connected vertically by a comparator
  - If $D_i > D_j \wedge i < j$ (i.e., wrong order), then D_i and D_j are swapped by the comparator before they move on along the wires



- Observation: every *comparator network* is *data independent*, i.e., the arrangement of comparators and the running time are always the same!
- Goal: find a "*small*" comparator network that performs sorting for *any* input → sorting network

Example



The 0-1 Principle

- Definition (*monotone function*):

  Let A, B be two sets with a total ordering relation, and let $f : A \rightarrow B$ be a mapping. f is called monotone iff $\forall a_1, a_2 \in A: \ a_1 \leq a_2 \Rightarrow f(a_1) \leq f(a_2)$

- Lemma:

  If $f : A \rightarrow B$ is monotone, then, f and *min* are commutative, i.e.

  $$\forall a_1, a_2 \in A: \ f(\min(a_1, a_2)) = \min(f(a_1), f(a_2))$$

  Analogously for the *max*.

- Proof:

  Case $a_1 \leq a_2$: $f(\min(a_1, a_2)) = f(a_1) \underset{f(a_1) \leq f(a_2)}{=} \min(f(a_1), f(a_2))$

  Case $a_2 < a_1$: analogous

Extension to Sequences

- Extension of $f : A \to B$ to sequences over A and B, resp.:

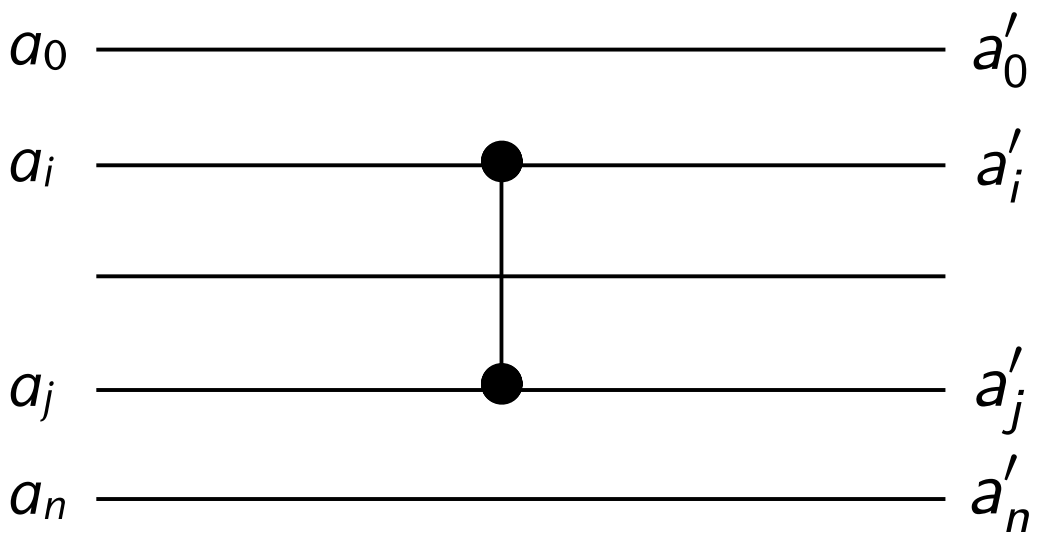
$$f(a_0, \ldots, a_n) = f(a_0), \ldots, f(a_n)$$

- Commutative Lemma for Comparator Networks:
  Let f be a monotone mapping and $\mathcal{N}$ a comparator network.
  Then $\mathcal{N}$ and f are commutative, i.e.

$$\forall n \, \forall a_0, \ldots, a_n : \; \mathcal{N}\big(f(a)\big) = f\big(\mathcal{N}(a)\big)$$

Proof

- Let $a = (a_0, \ldots, a_n)$ be a sequence
- Notation: we write a comparator connecting wires i and j like so: $a' = [i:j](a)$
- Now the following is true:



$$
\begin{aligned}
[i:j]\big(f(a)\big) &= [i:j]\big(f(a_0), \ldots, f(a_n)\big) \\
&= \big(f(a_0), \ldots, \underbrace{\min(\,f(a_i), f(a_j)\,)}_{i}, \ldots, \underbrace{\max(\,f(a_i), f(a_j)\,)}_{j}, \ldots, f(a_n)\big) \\
&= \big(f(a_0), \ldots, f(\,\min(a_i, a_j)\,), \ldots, f(\,\max(a_i, a_j)\,), \ldots, f(a_n)\big) \\
&= f\big(a_0, \ldots, \min(a_i, a_j), \ldots, \max(a_i, a_j), \ldots, a_n\big) \\
&= f\big([i:j](a)\big)
\end{aligned}
$$

- Theorem (0-1 principle):
  Let $\mathcal{N}$ be a comparator network.
  Now, if $\mathcal{N}$ sorts *every* sequence of 0's and 1's, then it also sorts *every* sequence of arbitrary elements!

Proof (by contradiction)

- Assumption: $\mathcal{N}$ sorts all 0-1 sequences, but does not sort sequence a
- Then $\mathcal{N}(a) = b$ is not sorted correctly, i.e. $\exists k : b_k > b_{k+1}$
- Define $f : A \rightarrow \{0,1\}$ as follows: $f(c) = \begin{cases} 0, & c < b_k \\ 1, & c \geq b_k \end{cases}$
- Now, the following holds:

$$f(b) = f\big(\mathcal{N}(a)\big) \underset{\uparrow}{=} \mathcal{N}\big(f(a)\big) = \mathcal{N}(a')$$

f monotone, Commut. Lemma

where a' is a 0-1 sequence.

- But: $f(b)$ is *not* sorted, because $f(b_k) = 1$ and $f(b_{k+1}) = 0$
- Therefore, $\mathcal{N}(a')$ is not sorted as well, in other words, we have constructed a 0-1 sequence that is not sorted correctly by $\mathcal{N}$.

Batcher's Odd-Even-Mergesort

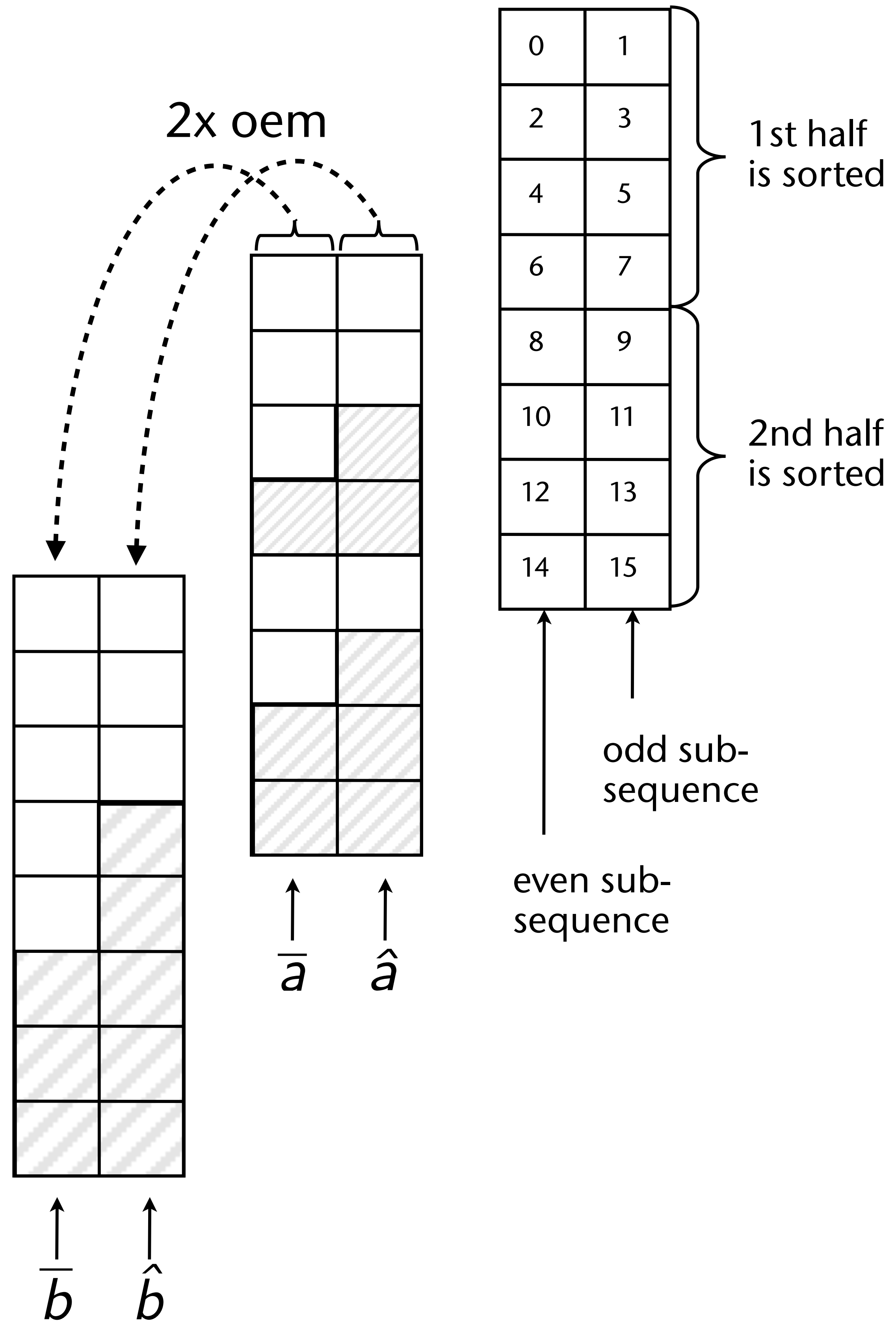
[1968]

- In the following, we'll always assume that the length n of a sequence $a_0,\dots,a_{n-1}$ is a power of 2, i.e., $n = 2^k$
- First of all, we define the sub-routine "odd-even merge":

```
oem( a0,…,an-1 ):
precondition:  a0,…,an/2 -1  and  an/2 ,…,an-1  are both sorted
postcondition: a0,…,an-1  is sorted
if  n = 2:
    compare [a0:a1]                                       (1)
if  n > 2:
    ā ← a0,a2,…,an-2             // = even sub-sequence
    â ← a1,a3,…,an-1             // = odd sub-sequence
    b̄ ← oem( ā )
    b̂ ← oem( â )                                           (2)
    copy b̄ → a0,a2,…,an-2
    copy b̂ → a1,a3,…,an-1
    for i ∈ {1,3,5,…,n-3}                                  (3)
        compare [ai : ai+1]
```

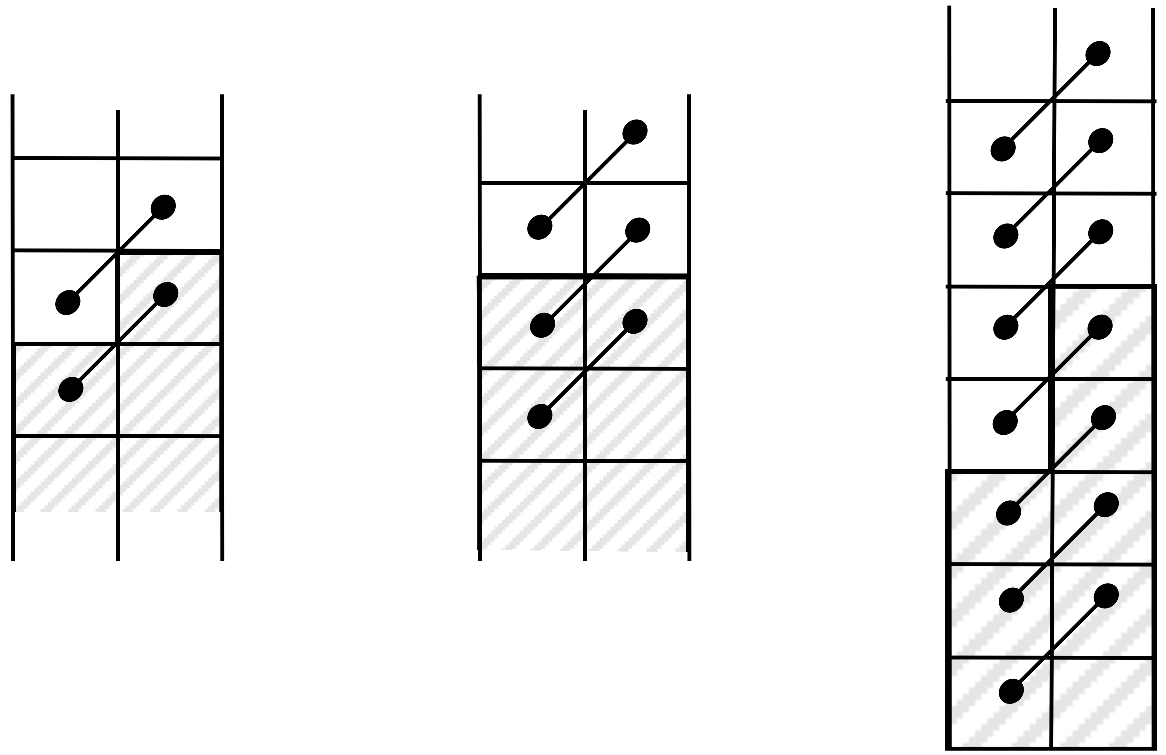
Proof of correctness

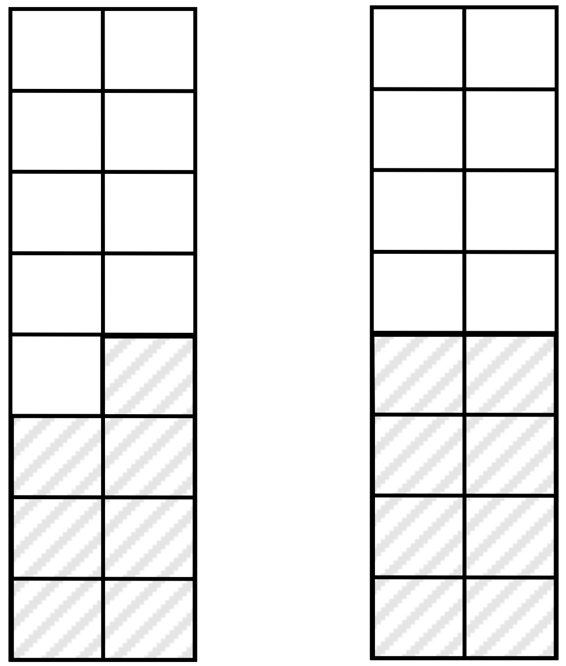
- By induction and the 0-1-principle
- Base case: $n = 2$
- Induction step: $n = 2^k$, $k > 1$
- Consider a 0-1-sequence $a_0,\ldots,a_{n-1}$
- Write it in two columns
- Visualize 0 = white, 1 = grey
- Obviously: both ā and â consist of two sorted halves → preconditon of **oem** is met
- After line (2) in the algo, we have this situation (the odd sub-sequence can have at most two 1's more than the even sub-sequence)



- In loop (3), these comparisons are made, and there can be only 3 cases:
- Afterwards, one of these two situations has been established:
- Result: the output sequence is sorted
- Conclusion:
  every 0-1-sequence (meeting the preconditions) is sorted correctly
- Running time (sequ.) :

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{2} - 1 \in O\left(n \log n\right)$$





- The complete general sorting algorithm:

```
oemSort(a₀,…,aₙ₋₁):
if n = 1:
  return
a₀,…,aₙ/₂ -1  ← oemSort(a₀,…,aₙ/₂ -1)
aₙ/₂ ,…,aₙ₋₁  ← oemSort(aₙ/₂ ,…,aₙ₋₁)
oem(a₀,…,aₙ₋₁)
```

- Running time (sequ.): $T(n) \in O\left(n \log^2 n\right)$

- Note: in a real implementation, no copying is done!

Mapping the Recursion to a Massively-Parallel Architecture

Only FYI

- Load data onto the GPU (global memory)
- Each recursion in the sequential **oem** can be mapped to a stride parameter value, so that sorting can be done *in situ*: recursion $i \longrightarrow$ stride = 2^i ; at that recursion level / iteration, the algo works only on elements that are stride places apart
- The CPU executes the following control program (informal):

```
oemSort(n):
if n = 1 → return
do in parallel:
  oemSort( lower n/2 )
  oemSort( upper n/2 )
oem( n, 1 )
```

```
oem( n, stride ):
if n = 2:
    launch oemBaseCaseKernel(stride)
   // launch N (not n) threads
else:
    oem( n/2, stride*2 )
    launch oemRecursionKernel(stride)
```

N = total size of input , n = #elements the function will actually look at

Only FYI

- The kernel for line (3) of the original function **`oem`**():

```
oemRecursionKernel( stride ):
if tid < stride || tid ≥ n-stride:
    output SortData[tid]       // pass through
else:
    a_i ← SortData[tid]
    a_j ← SortData[ tid+stride ]
    if tid/stride is even:
        output max( a_i, a_j )
    else:
        output min( a_i, a_j )
```

As usual, *tid* = thread ID = 0, …, *n*-1

Only FYI

- Kernel for line (1) of the function **`oem`**():
  - Reminder: this kernel is executed in parallel for each index *tid* = 0, …, *n*-1

```
oemBaseCaseKernel ( stride ):
i = tid                           // = thread ID
if tid/stride is even:            // are we on even/odd side?
    j = i + stride
else:
    j = i - stride
a0 ← SortData[i]                  // SortData = global array
a1 ← SortData[j]
if on even side:
    SortData[i] = min(a0,a1)      // write output back
else:
    SortData[i] = max(a0,a1)
```

Only FYI

- Depth complexity:

$$\frac{1}{2}\log^2 n + \frac{1}{2}\log n$$

- E.g., for 2^{20} elements this amounts to 210 passes

Bitonic Sorting

- Definition "bitonic sequence":

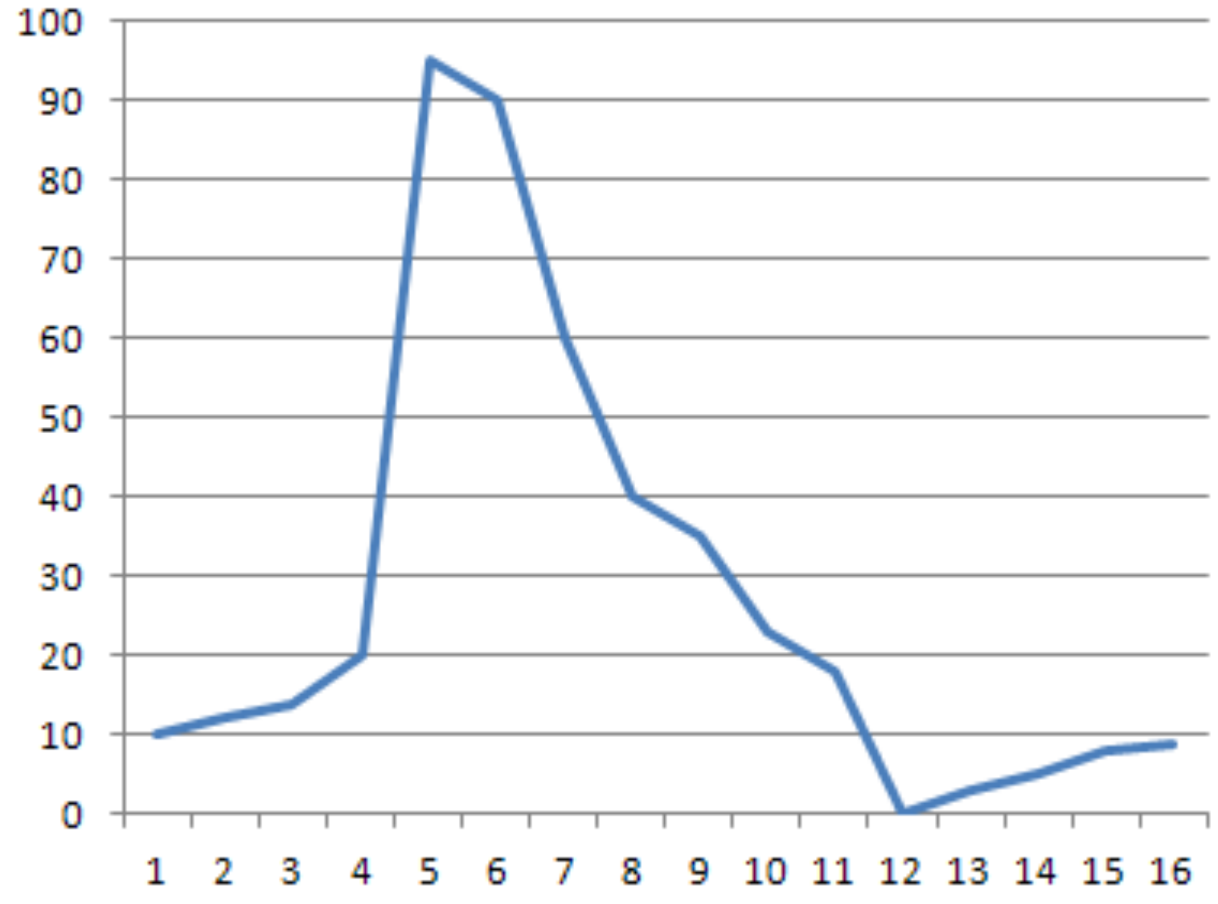
  A sequence of numbers $a_0, \ldots, a_{n-1}$ is bitonic $\Leftrightarrow$ there is an index i such that

  - $a_0, \ldots, a_i$ is monotonically increasing, and
    $a_{i+1}, \ldots, a_{n-1}$ is monotonically decreasing;
  - OR, if there is a cyclic shift of this sequence such that this is the case.

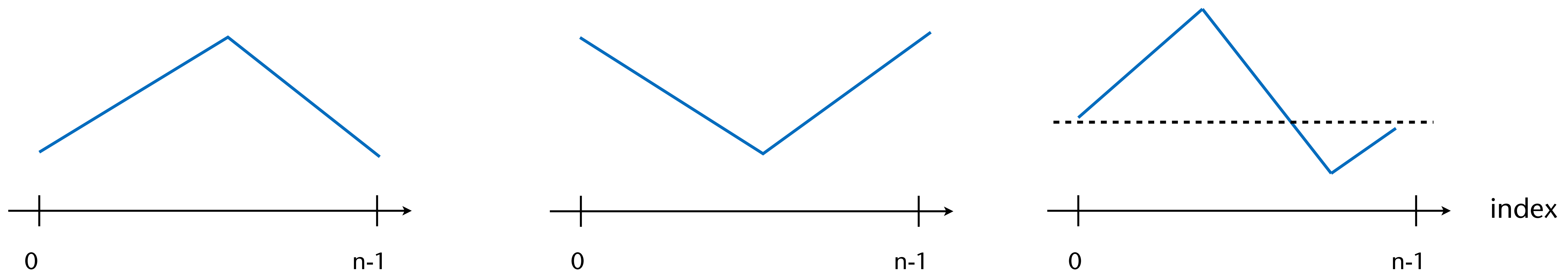
- Because of the second condition (OR), we understand *all index arithmetic* in the following modulo *n*, and/or we assume in the following that the sequence(s) have been cyclically shifted as described above

Examples of bitonic sequences

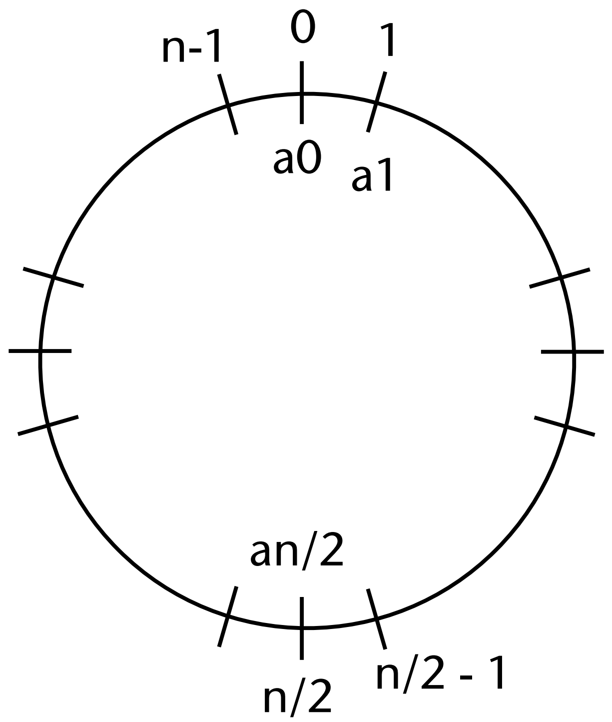
- 0 2 4 8 10 9 7 5 3 ; also: 2 4 8 10 9 7 5 3 0 ; also: 4 8 10 9 7 5 3 0 2 ; …
- 10 12 14 20 95 90 60 40
  35 23 18 0 3 5 8 9
- 1 2 3 4 5
- [ ]
- 0000011111 0000 ;
  111110000 0111111 ;
  1111100000 ; 000011111
- These sequences are NOT bitonic sequences:
  - 1 2 3 1 2 3
  - 1 2 3 0 1 2

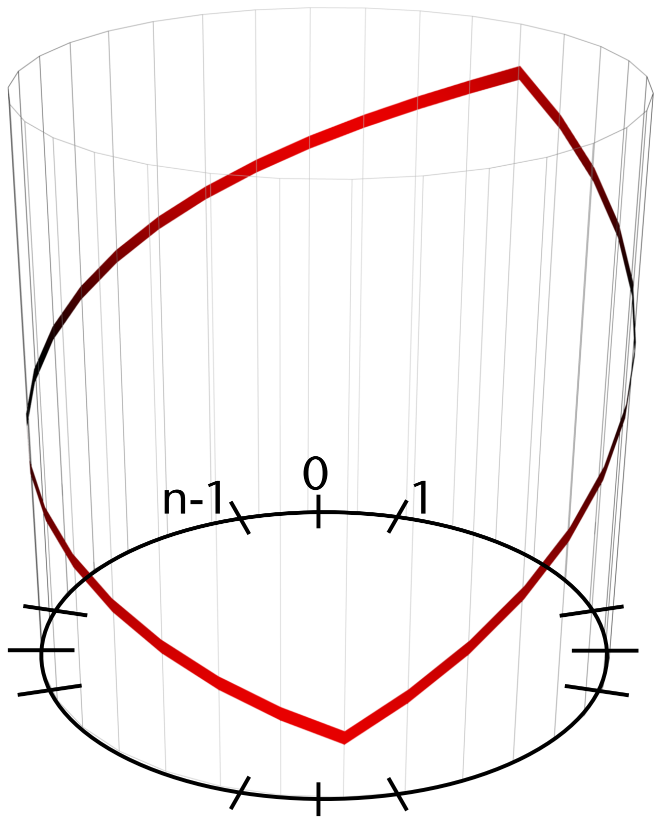


Visual representation of bitonic sequences



- Because of the "modulo" index arithmetic, we can also visualize them on a circle or cylinder
  - Clearly, bitonic sequences have always exactly two inflection points



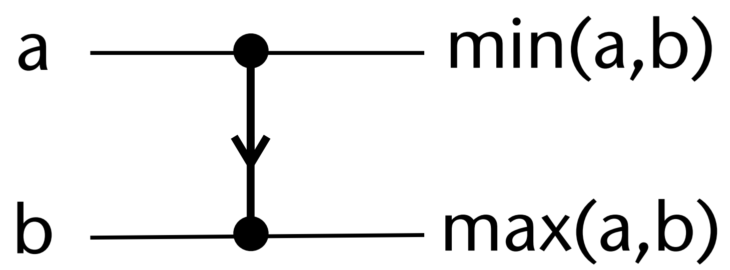


Properties of Bitonic Sequences

- The bitonic property is *invariant* against subset extraction, reversal, flipping
- Any sub-sequence of a bitonic sequence is a bitonic sequence (too)
  - More precisely, assume $a_0, \ldots, a_{n-1}$ is bitonic and we consider some indices $0 \leq i_1 \leq i_2 \leq \ldots \leq i_m < n$
  - Then, $a_{i_0}, a_{i_1}, \ldots, a_{i_m}$ is bitonic, too
- If $a_0, \ldots, a_{n-1}$ is bitonic, then $a_{n-1}, \ldots, a_0$ is bitonic, too
- If we mirror a bitonic sequence "upside down", then the new sequence is bitonic, too
- A bitonic sequence has exactly *one* local minimum and *one* local maximum

Some Notions and Definitions

- More precise graphical notation of a comparator:



- Definition rotation operator:

  Let $\mathbf{a} = (a_0, \ldots, a_{n-1})$, and $j \in [1, n\text{-}1]$.

  We define the rotation operator R_j acting on **a** as

$$R_j\mathbf{a} = (a_j, a_{j+1}, \ldots, a_{j+n-1})$$

- Definition L / U operator:

$$L\mathbf{a} = (\min(a_0, a_{\frac{n}{2}}), \ldots, \min(a_{\frac{n}{2}-1}, a_{n-1}))$$
$$U\mathbf{a} = (\max(a_0, a_{\frac{n}{2}}), \ldots, \max(a_{\frac{n}{2}-1}, a_{n-1}))$$

- Lemma:
  The L/U operators are *rotation invariant*, i.e., for any *j*

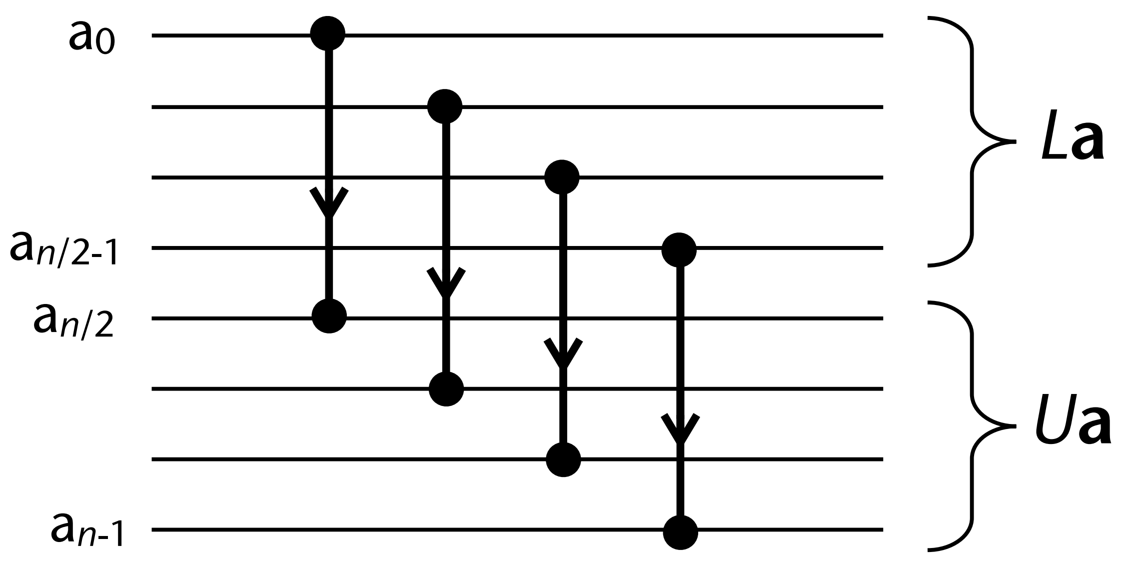
$$L\mathbf{a} = R_{-j} L R_j \mathbf{a}, \quad \text{and} \quad U\mathbf{a} = R_{-j} U R_j \mathbf{a}.$$

  (Remember that indices are always meant mod *n*)

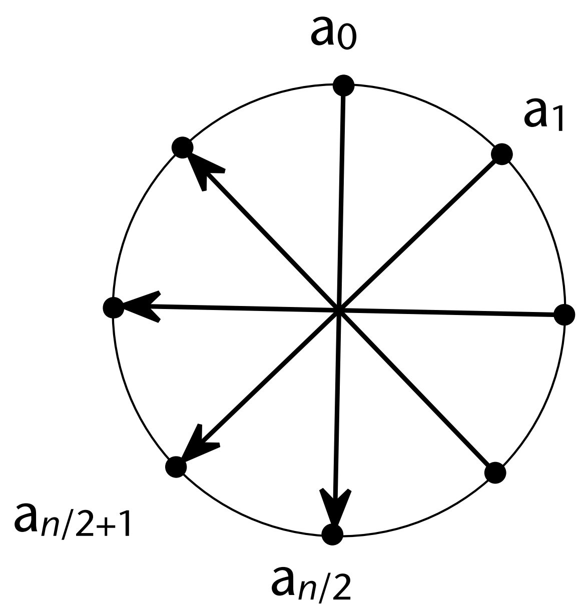
- Proof :
  - We need to show that $R_j L\mathbf{a} = LR_j\mathbf{a}$
  - This is trivially the case:

$$LR_j\mathbf{a} = \left(\min(a_j, a_{j+\frac{n}{2}}), \ldots, \min(a_{\frac{n}{2}-1}, a_{n-1}), \ldots, \min(a_{j-1}, a_{j-1+\frac{n}{2}})\right) = \ldots$$

- Definition half-cleaner:
  is network that takes **a** as input
  and outputs (*L***a** , *U***a**)
- The network that realizes a *half-cleaner*



- Because of the rotation invariance, we can depict a half-cleaner on a circle:
  - It always produces *L***a** and *U***a**, no matter how **a** is rotated around the circle!

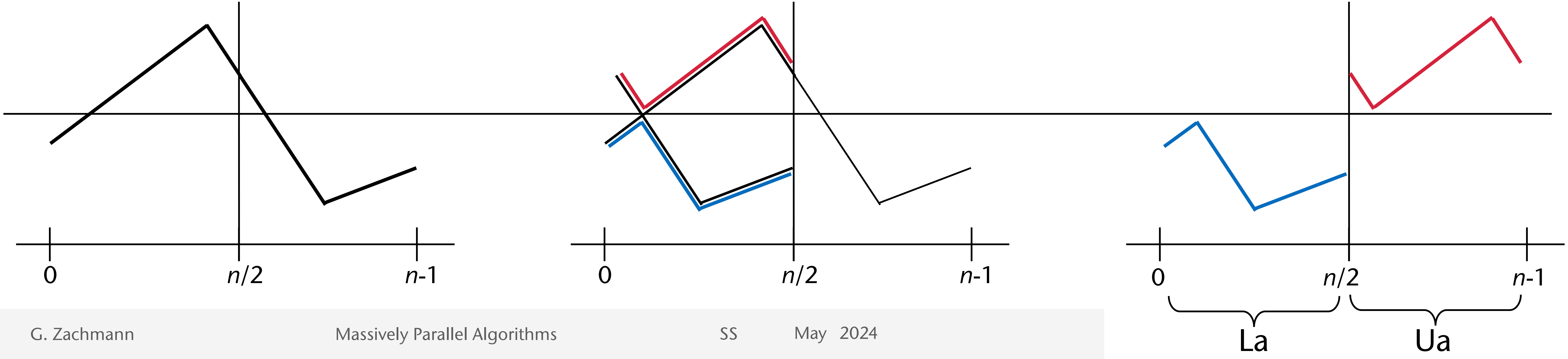


- Theorem 1:
  Given a bitonic input sequence **a**, the output of a half-cleaner has the following properties:

  1. L**a** and U**a** are *bitonic*, too;
  2. $\max\{L\mathbf{a}\} \leq \min\{U\mathbf{a}\}$

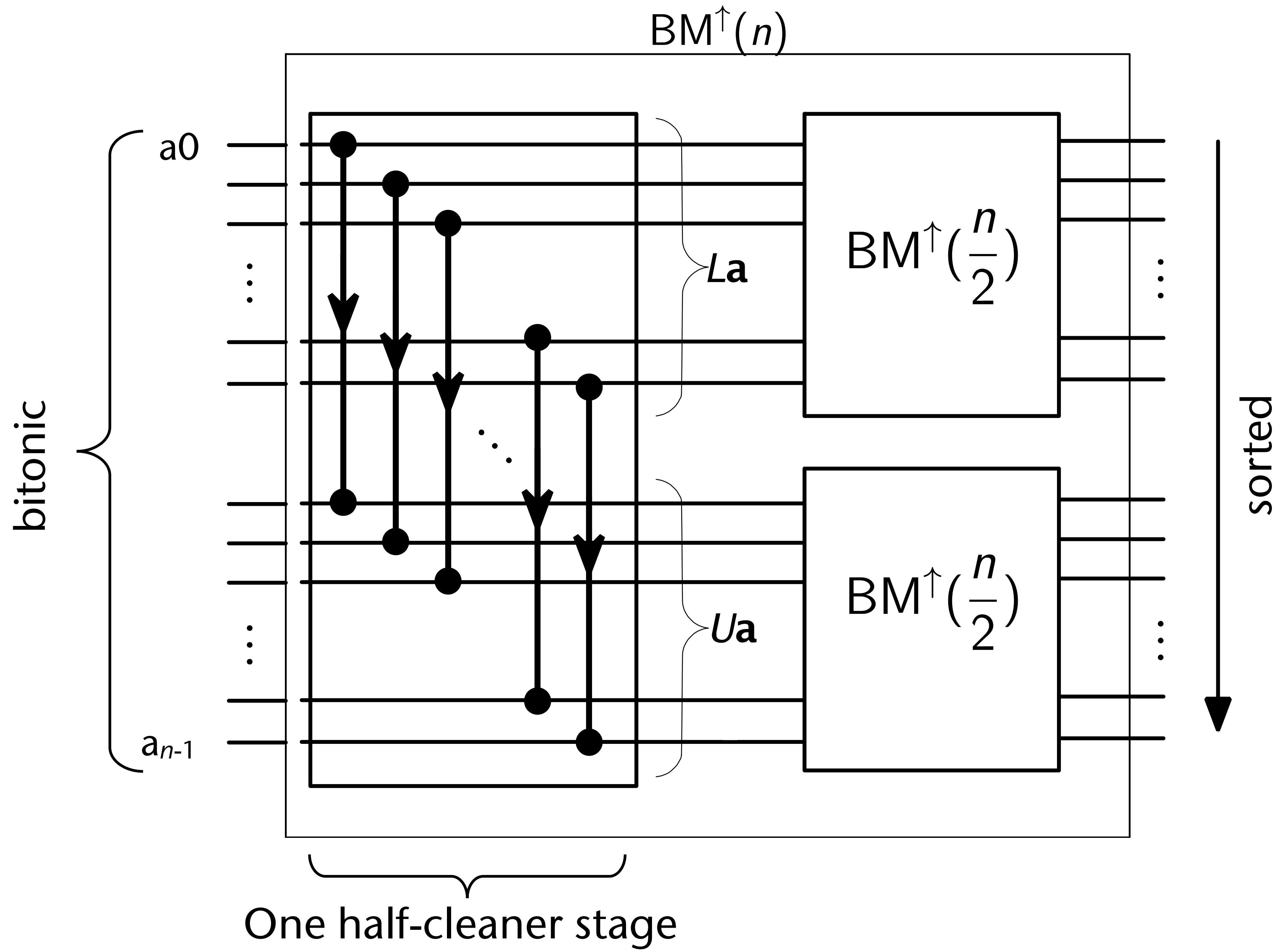
Proof

- The half-cleaner does the following:
  1. Shift (only conceptually) the right half of **a** over to the left
  2. Take the point-wise min/max $\longrightarrow$ *L***a** , *U***a**
  3. Shift *U***a** back to the right
- Because **a** is bitonic, there can be only one "cross-over" point
- By construction, both *L***a** and *U***a** must have length *n*/2
- Property 1 in theorem 1 follows from the sub-sequence property

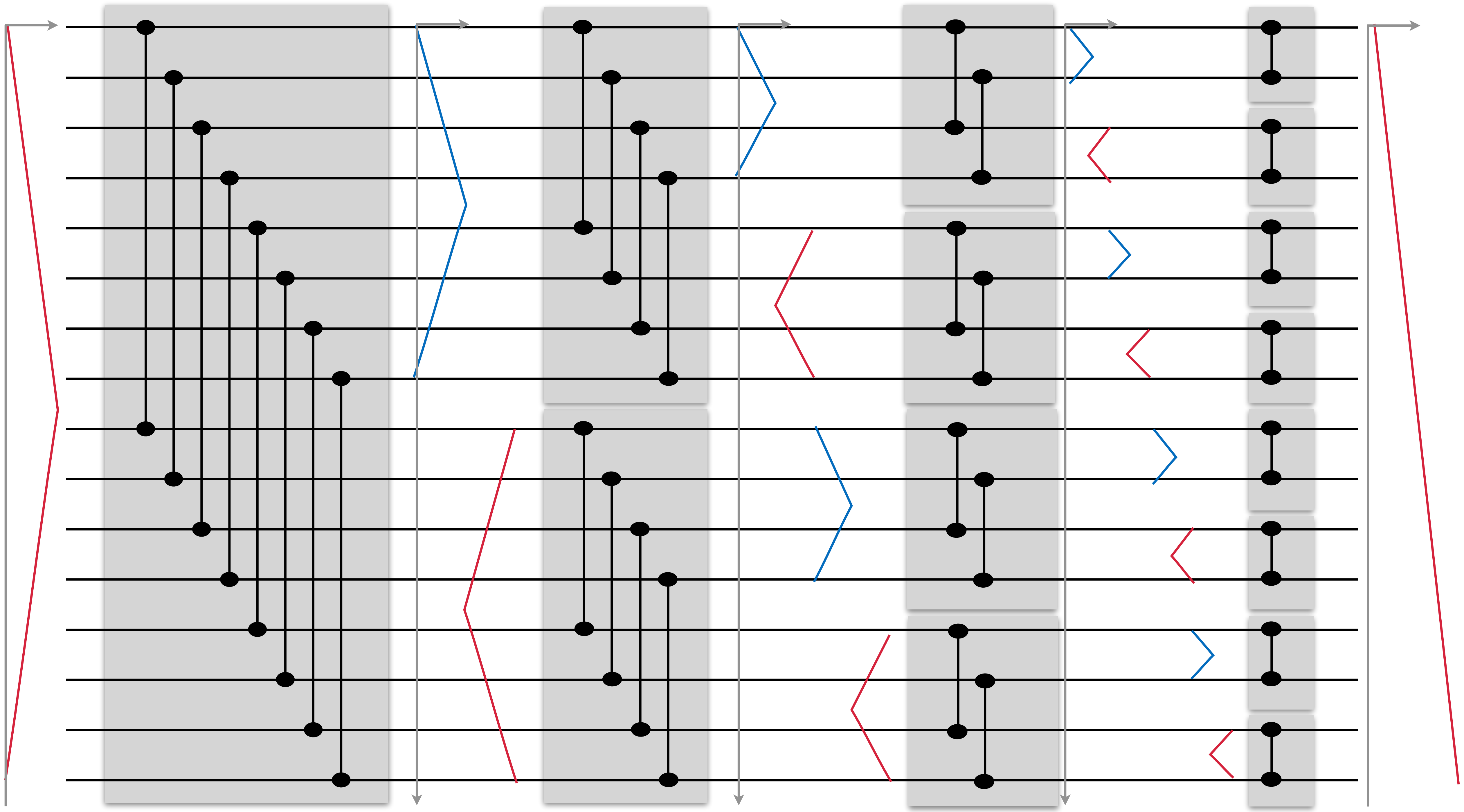


The Bitonic Merger

- The half-cleaner is the basic (and only) building block for the bitonic sorting network!
- The recursive definition of a bitonic merger $BM^{\uparrow}(n)$:
  - Input: bitonic sequence of length *n*
  - Output: sorted sequence in *ascending* order
- Analogously, we can define $BM^{\downarrow}(n)$

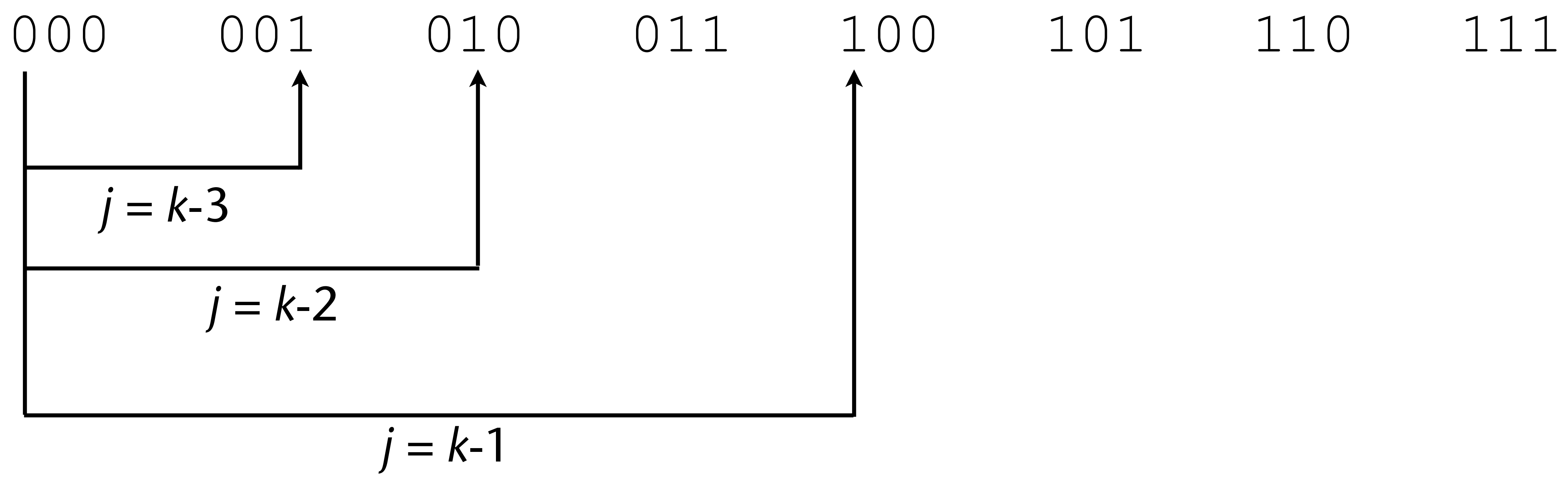


Visualization of a Bitonic Merger



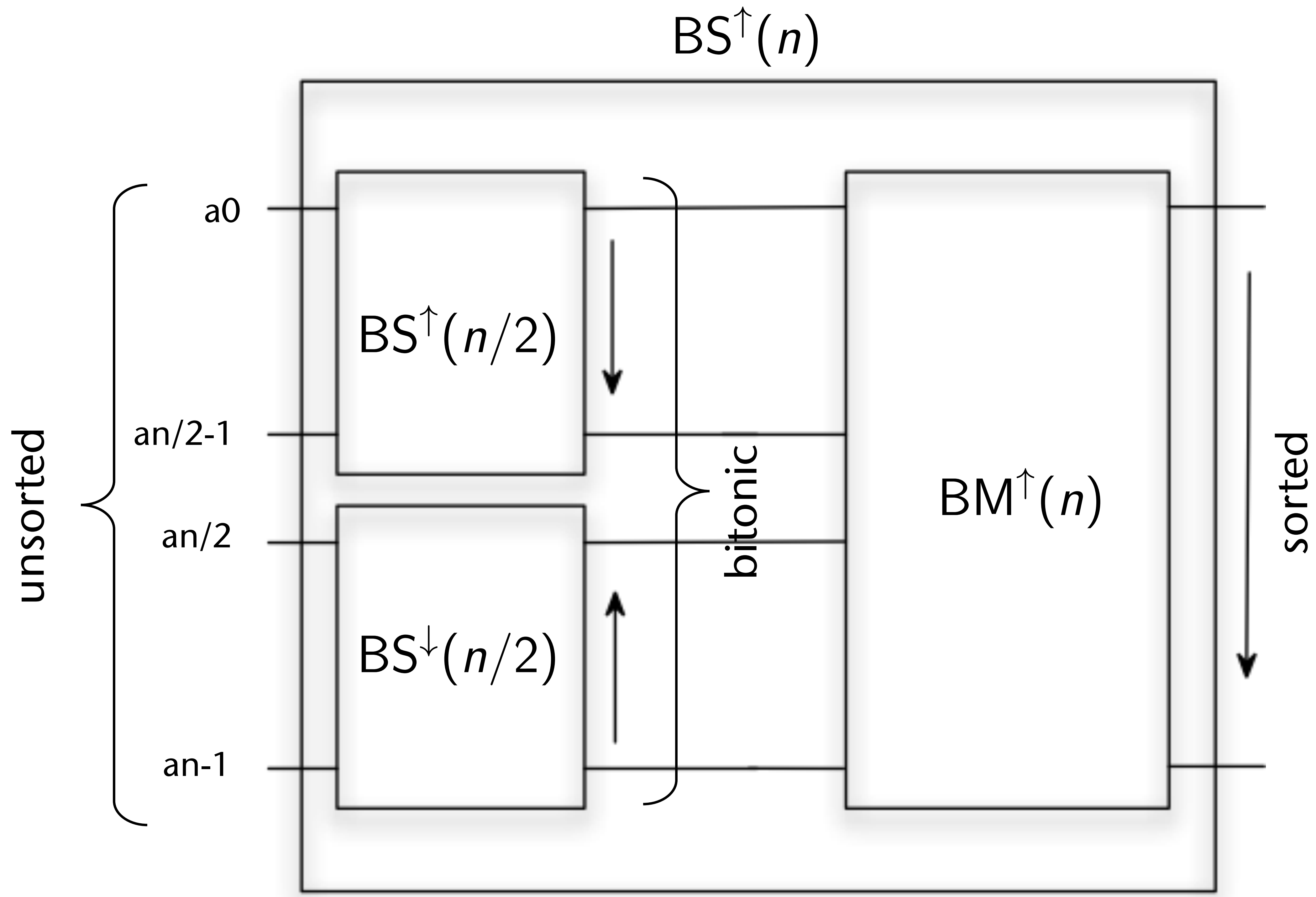
Mapping to a Massively Parallel Architecture

- We have $n = 2^k$ many "lanes" = threads
- At each step, each thread needs to figure out its partner for compare/exchange
- This can be done by considering the ID of each thread (in binary):
  - At step j, $j = 1, \ldots, k$: partner ID = ID obtained by reversing bit $(k\text{-}j)$ of own ID
- Example:

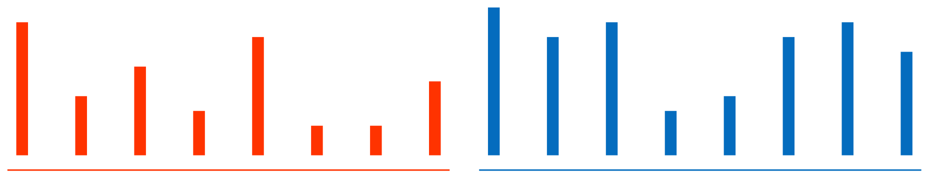


The Bitonic Sorter

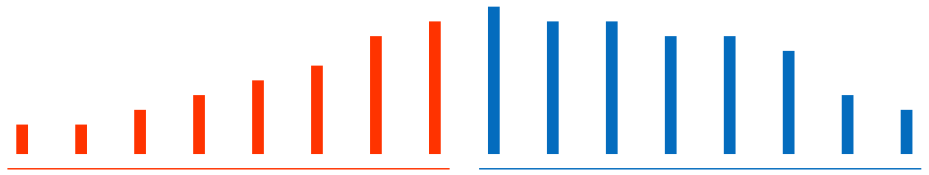
- The recursive definition of a bitonic sorter $BS^{\uparrow}(n)$:



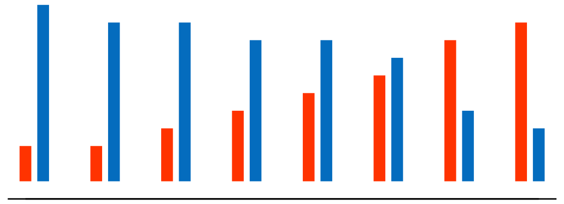
Visualizing Bitonic Sorting



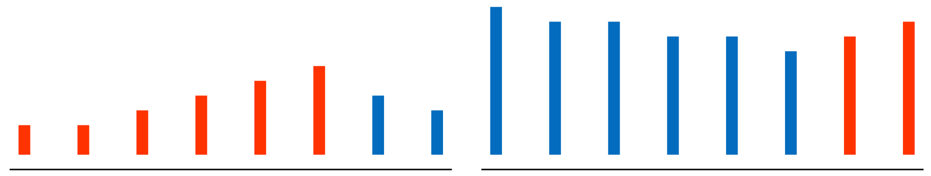
Initial data sequence



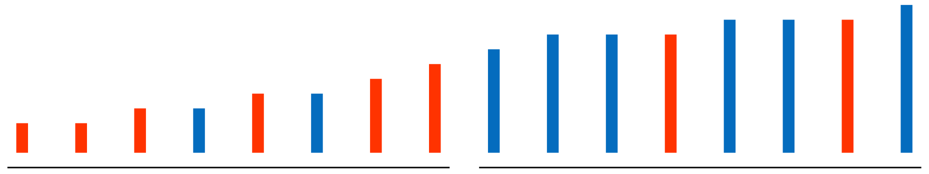
1: Sort array halves in opposite directions to achieve a bitonic sequence



2: Overlap and compare the array halves (half-cleaner)

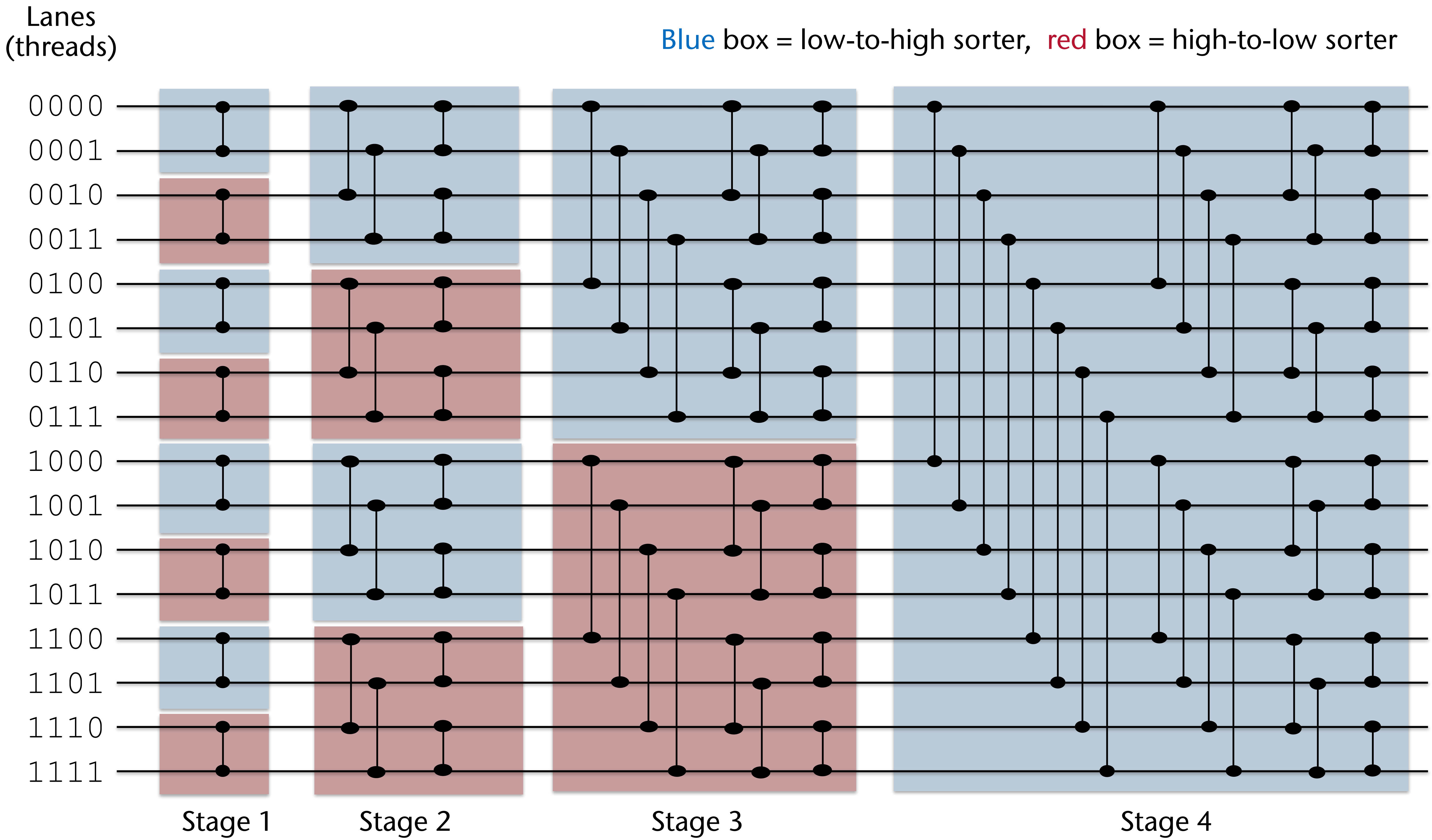


3: Send larger item in each pair to the right



Perform 2 & 3 recursively on each half

Example Bitonic Sorting Network



Example Run

3

7

4

8

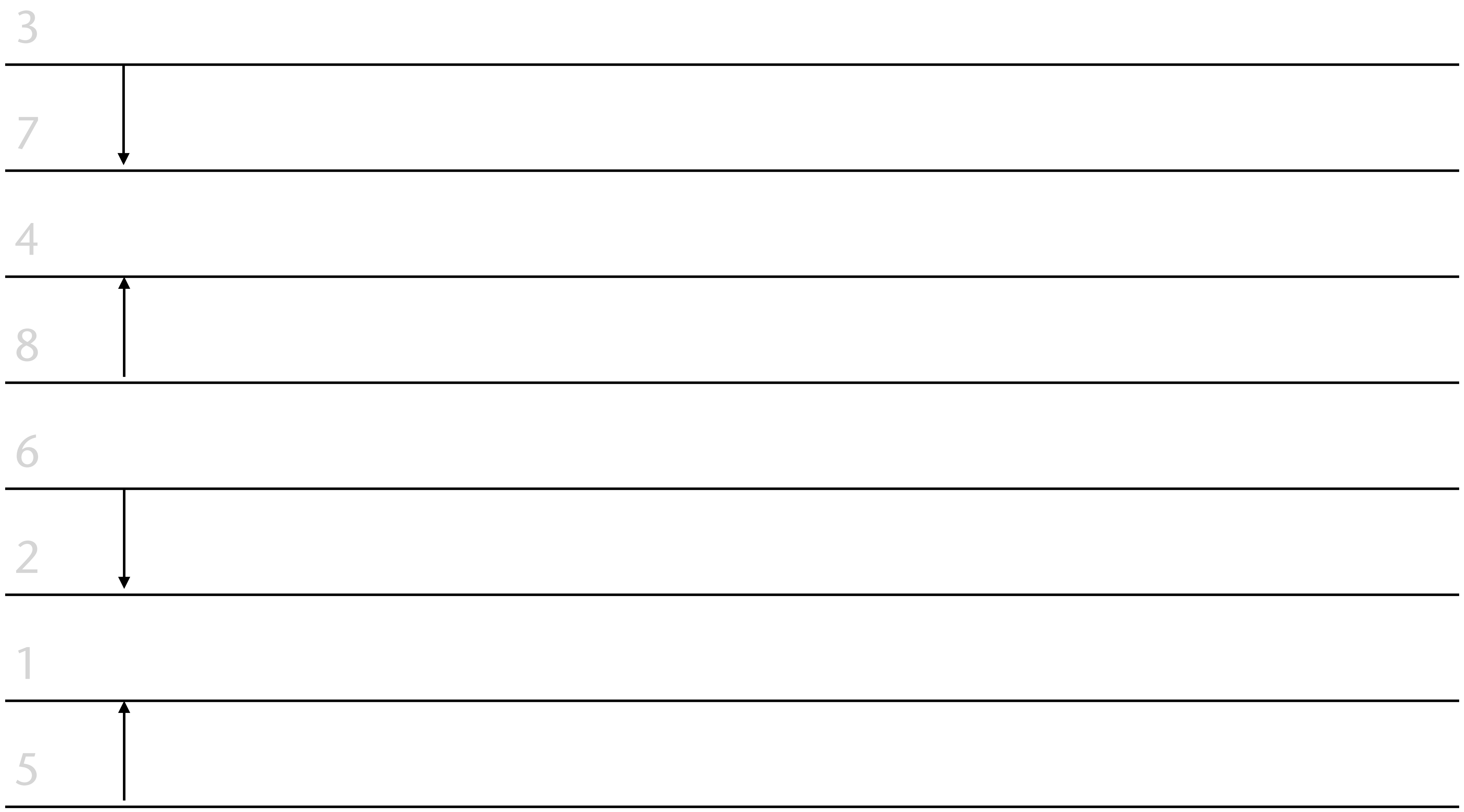
6

2

1

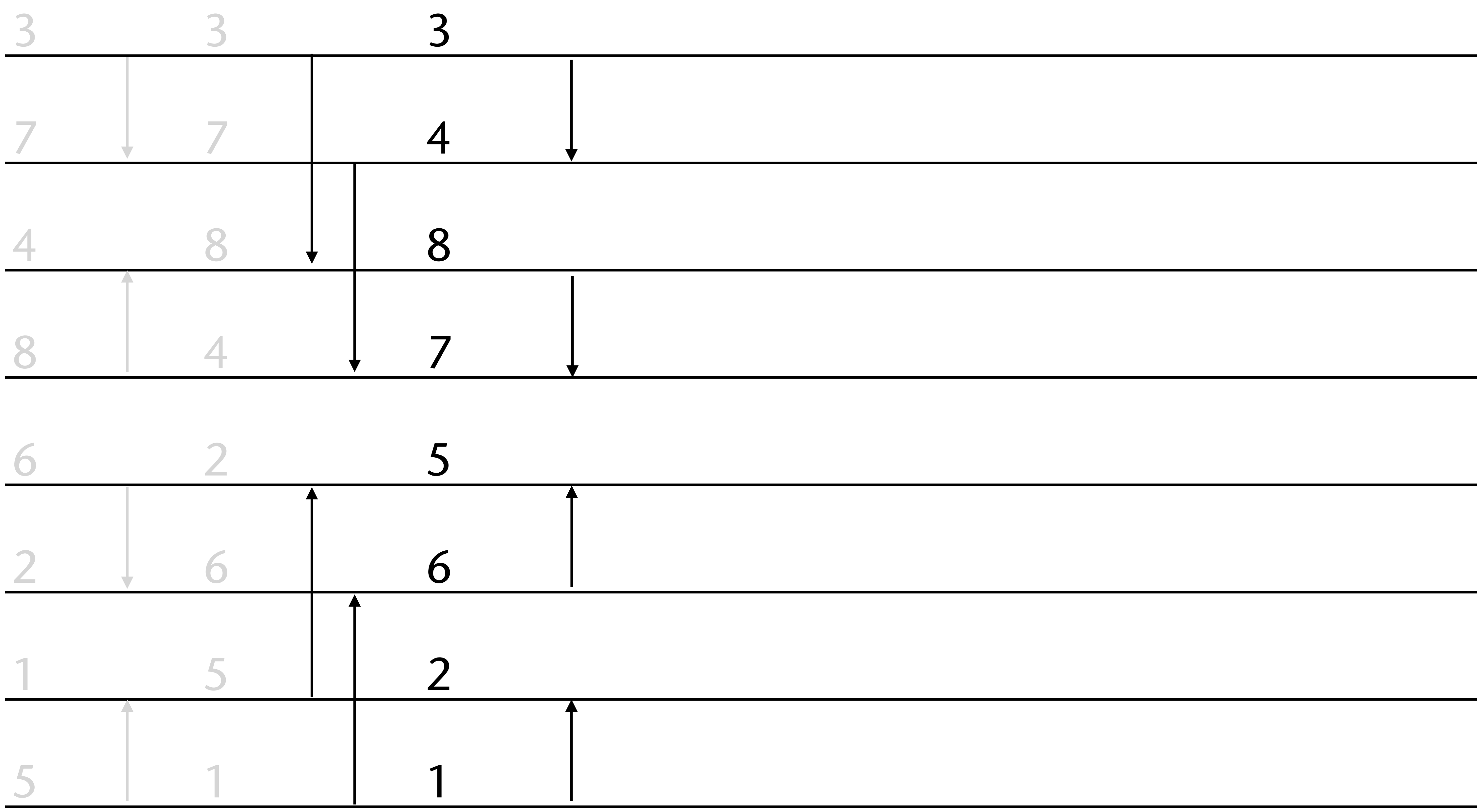
5

8x monotonic lists: (3) (7) (4) (8) (6) (2) (1) (5)
4x bitonic lists: (3,7) (4,8) (6,2) (1,5)

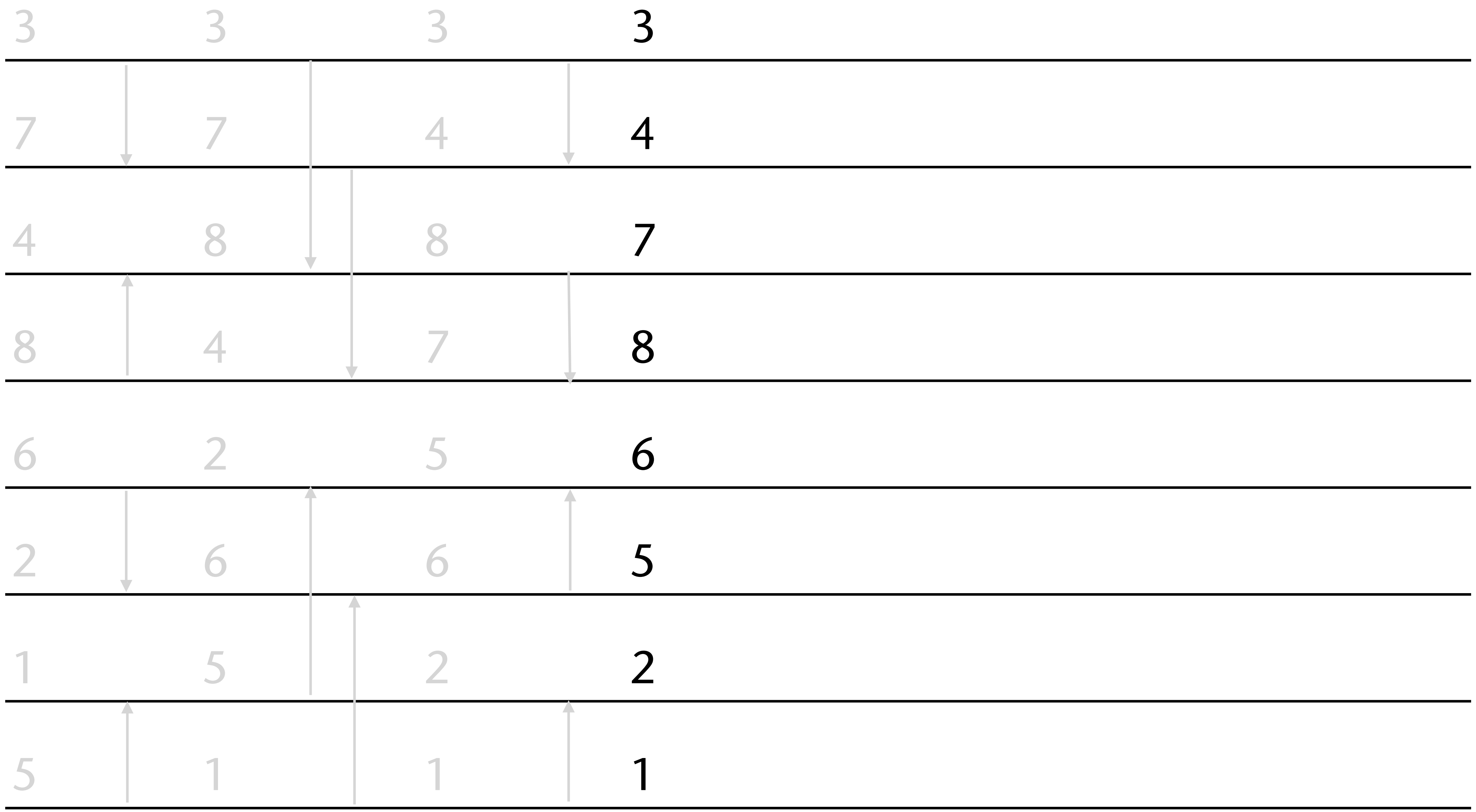


Sort the bitonic lists (each list = 2 elements ⟶ trivially bitonic)

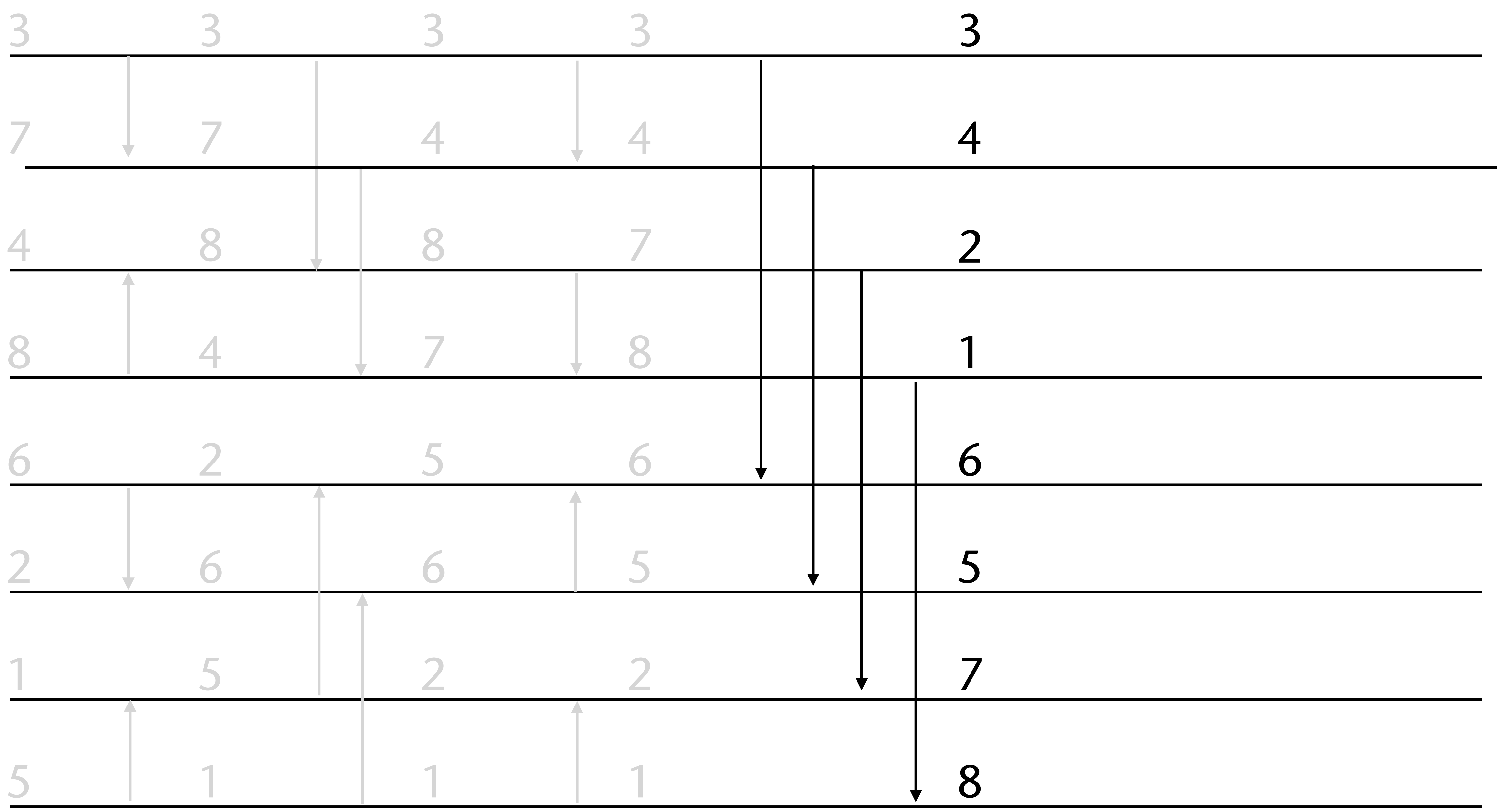
4x monotonic lists: (3,7) (8,4) (2,6) (5,1)
2x bitonic lists: (3,7,8,4) (2,6,5,1)



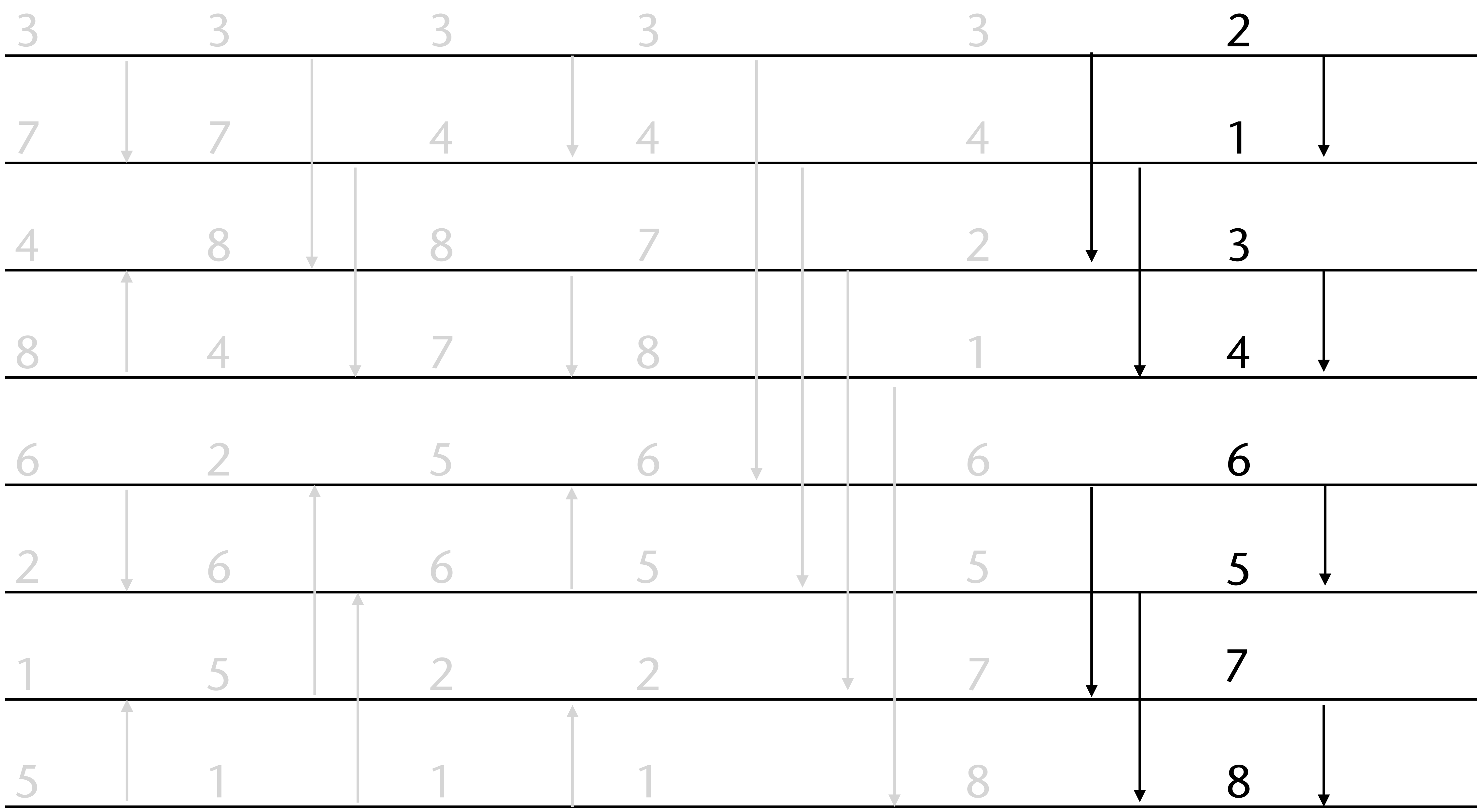
Sort the bitonic lists



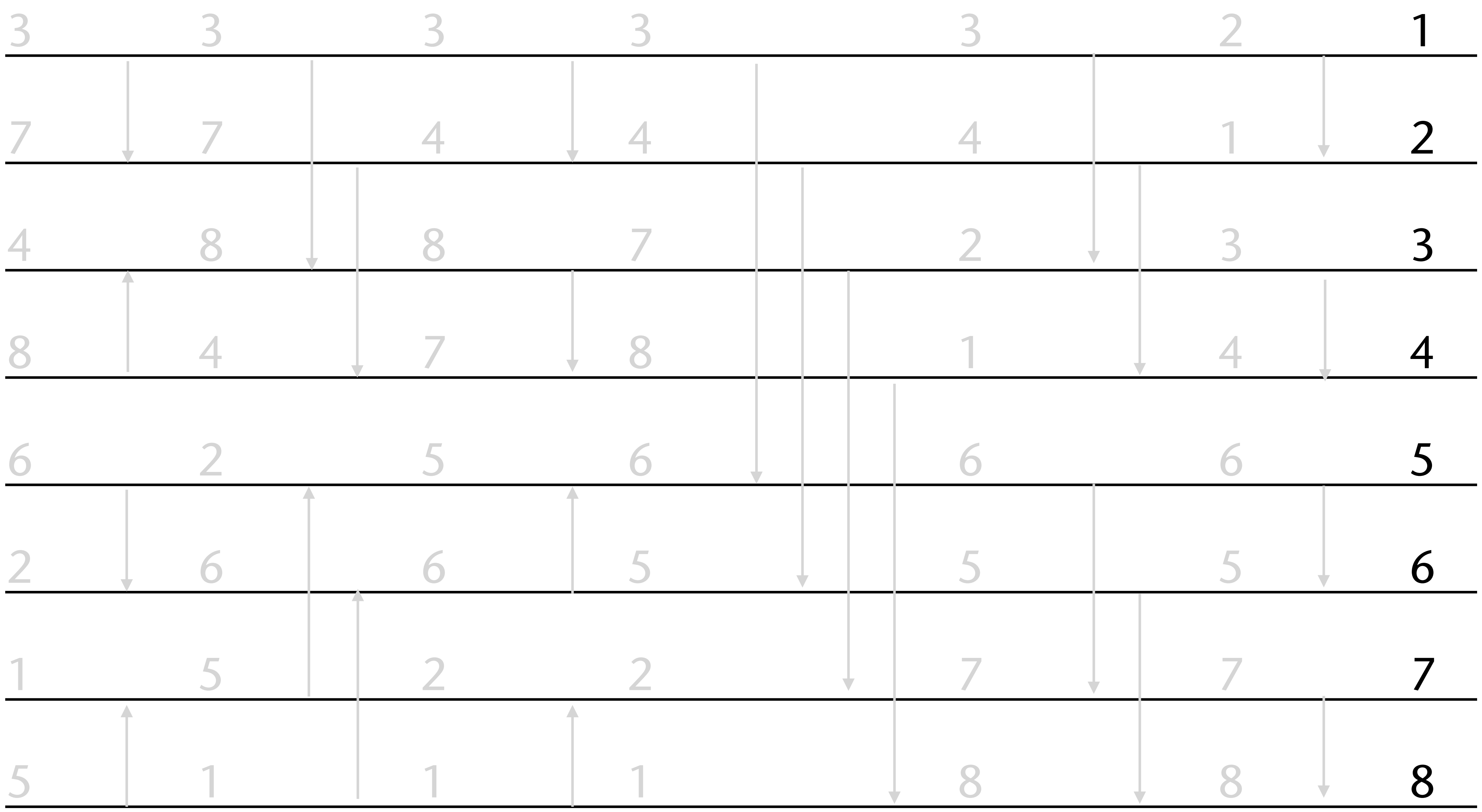
2x monotonic lists: (3,4,7,8) (6,5,2,1)
1x bitonic list: (3,4,7,8, 6,5,2,1)



Sort the bitonic lists



Sort the bitonic lists


3 3 3 3 3 2 1
7 7 4 4 4 1 2
4 8 8 7 2 3 3
8 4 7 8 1 4 4
6 2 5 6 6 6 5
2 6 6 5 5 5 6
1 5 2 2 7 7 7
5 1 1 1 8 8 8

Done!

Complexity of the Bitonic Sorter

- Depth complexity (= parallel time complexity):
  - Bitonic merger: $O(\log n)$
  - Bitonic sorter: $O(\log^2 n)$
- Work complexity of bitonic merger: count #comparators = $C(n)$
  - Recursive equation for C: $C(n) = 2C(\frac{n}{2}) + \frac{n}{2}$, with $C(2) = 1$
  - Overall: $C(n) = \frac{1}{2} n \log n$
- Remark: there must be some redundancy in the sorting network, because we know (from merge sort) that n comparisons are sufficient for *merging* two sorted sequences
- Reason for the redundancy? $\longrightarrow$ because the network is data-*independent*!

Remarks on Bitonic Sorting

- Probably most well-known parallel sorting algo / network
- Fastest algorithm for "small" arrays
- Lower bound on depth complexity for parallel sorting is

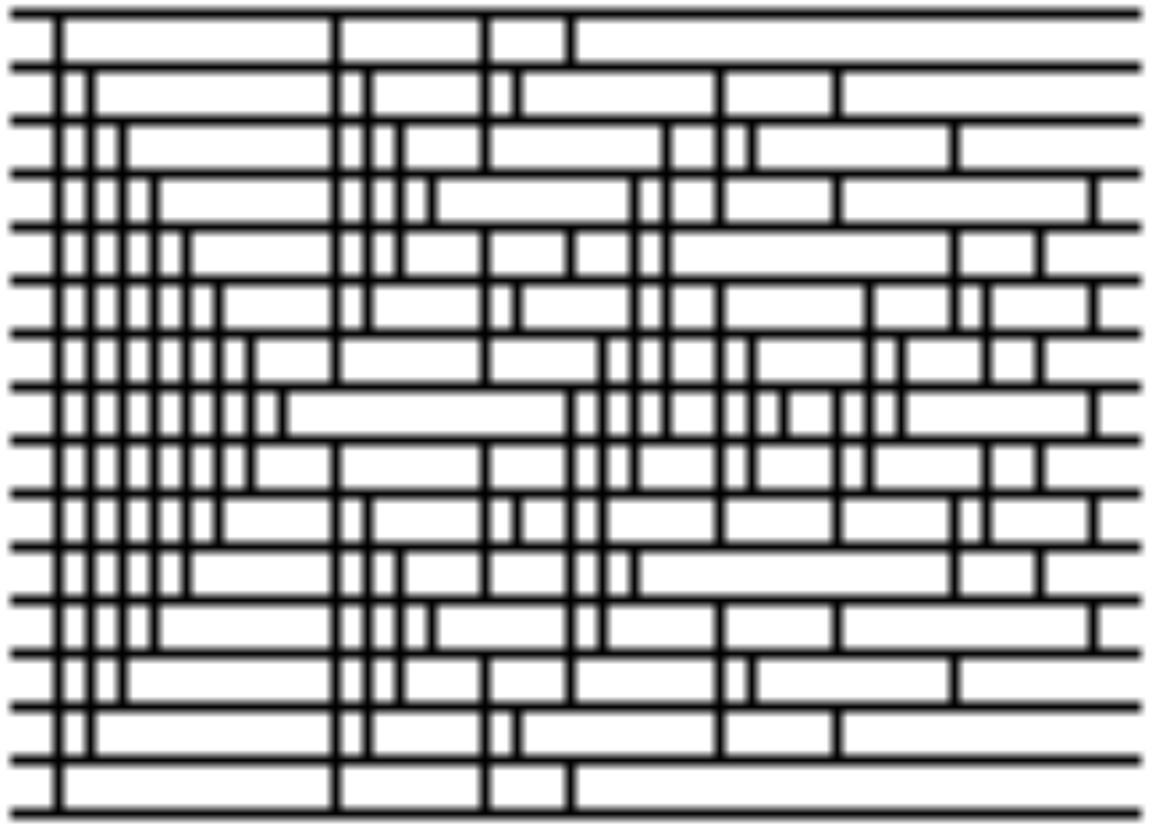
$$\frac{O\left(n \log n\right)}{n} = O\left(\log n\right)$$

assuming we have n processors (in this sense, the bitonic is not optimal)

- A nice property: comparators in a bitonic sorter network only ever compare lanes whose labels (= binary lane number) differ by exactly one bit!
- Consequence for the implementation:
  - One kernel for all threads
  - Each thread only needs to determine which bit of its own thread ID to "flip" $\longrightarrow$ gives the "other" lane with which to compare
- Hence, bitonic sorting is sometimes pictured as well-suited for a log(n)-dimensional hypercube parallel architecture:
  - Each node of the hypercube = one processor
  - Each processor is connected directly to log(n) many other processors
  - In each step, each processor talks to one of its direct neighbors

Optimal Sorting Networks

- Optimal = minimal depth
- Known up to depth 11 [2013], and depth 40 [2014]
- Example: optimal depth $d = 9$ for $n = 16$



- Would it improve performance on the GPU??

Adaptive Bitonic Sorting

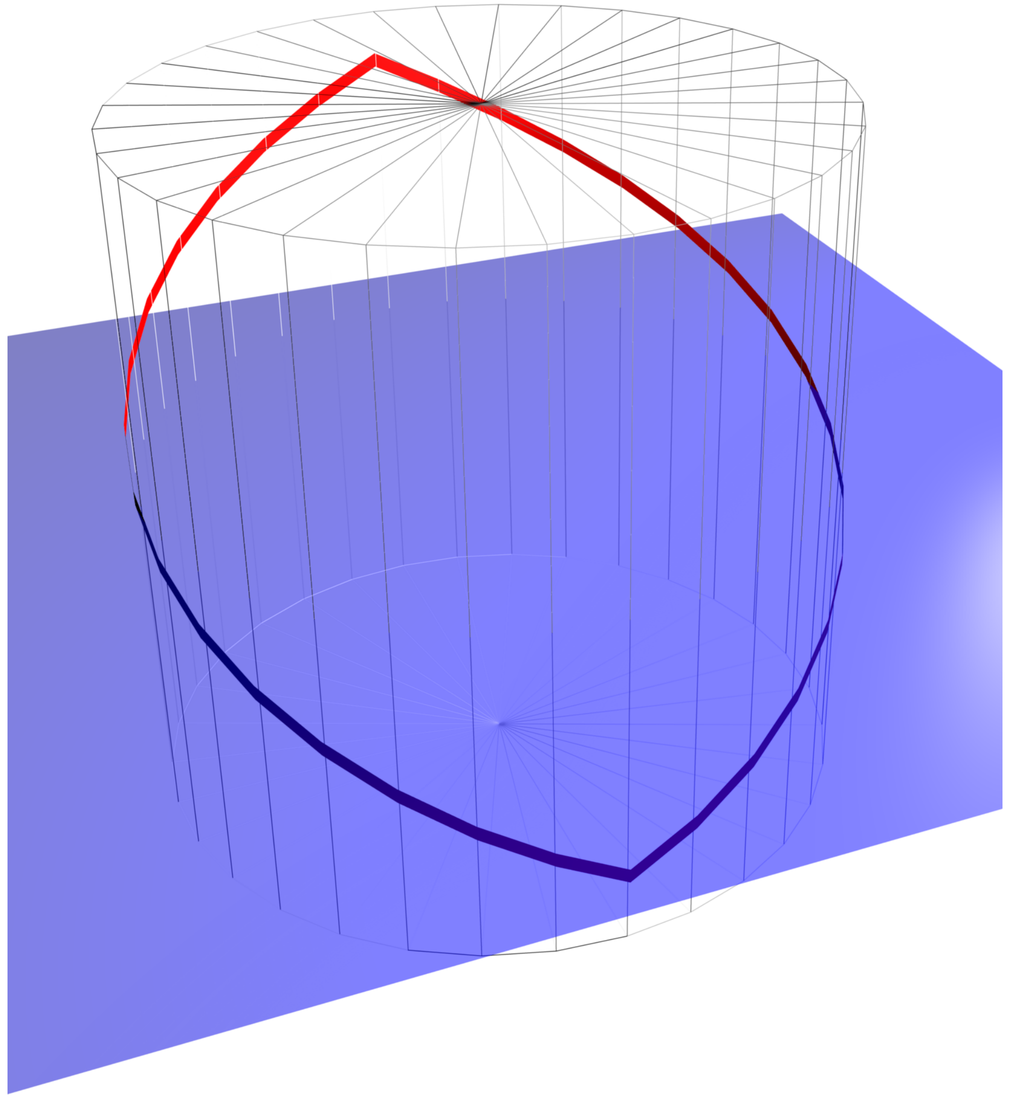
- Theorem 2:
  Let **a** be a bitonic sequence.
  Then, we can always find an index q such that

$$\max\left(a_q, \ldots, a_{q+\frac{n}{2}-1}\right) \leq \min\left(a_{q+\frac{n}{2}}, \ldots, a_{q-1}\right)$$

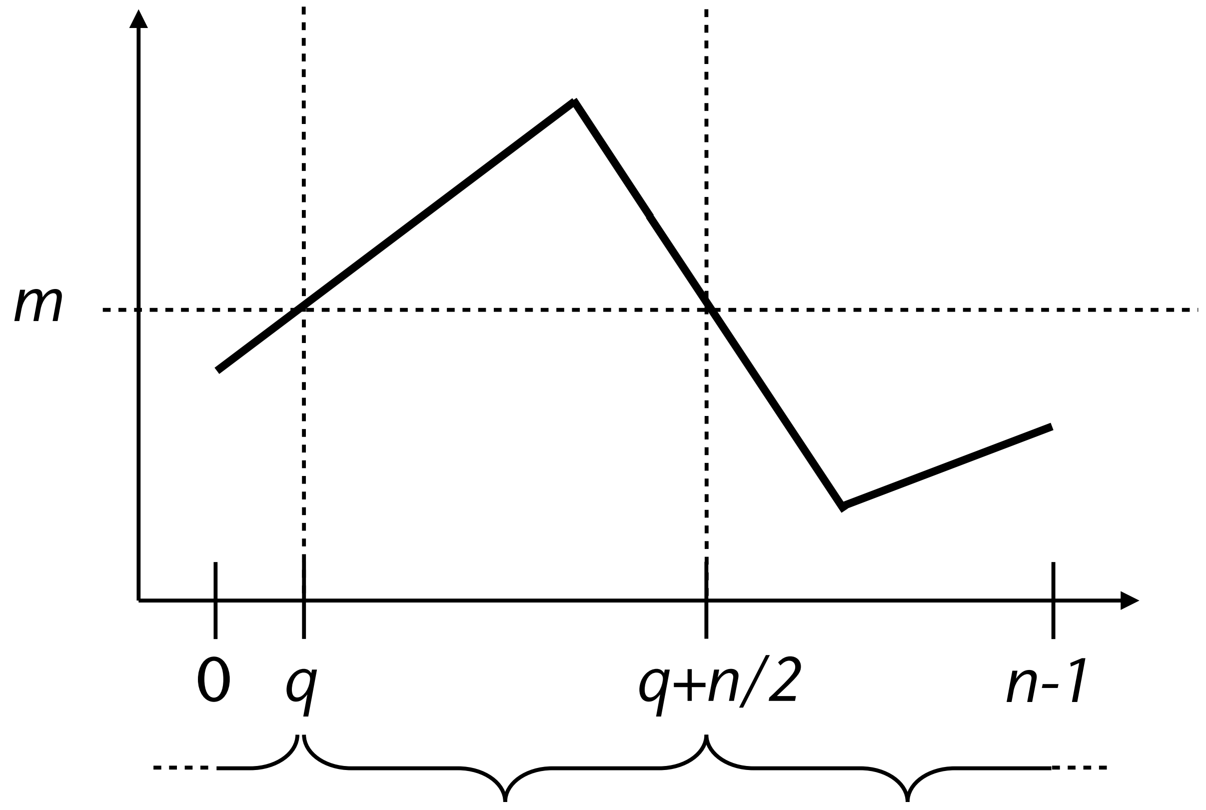
- This can be turned into an adaptive bitonic merger (ABM)

Sketch of Proof

- Assume (for sake of simplicity) that all elements in **a** are distinct
- Imagine the bitonic sequence as a "line" on a cylinder
- Since **a** is bitonic $\longrightarrow$ only two inflection points $\longrightarrow$ each horizontal plane cuts the sequence at exactly 2 points, and both sub-sequences are contiguous (under index arithmetic using modulo!)
- Use the median *m* as "cut plane" $\longrightarrow$ each sub-sequence has length *n/2*, and max("lower sequ.") ≤ *m* ≤ min("upper sequ.")
- The index of *m* is exactly index *q* in Theorem 2
- The two halves must be *L***a** and *U***a** , resp.



- Visualization of the theorem:



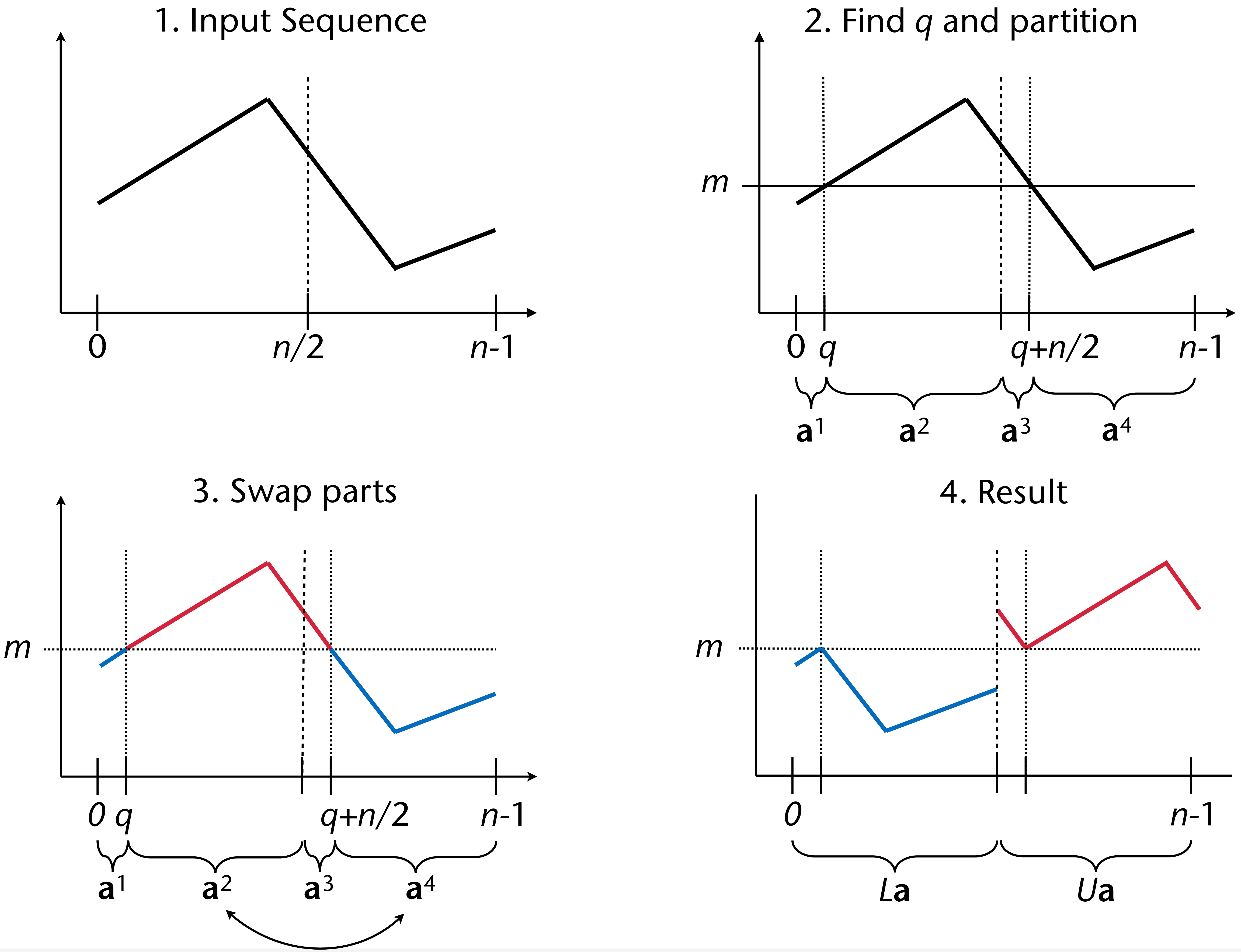
- Theorem 3:
  Any bitonic sequence **a** can be partitioned into four sub-sequences (a1, a2, a3, a4) = **a**, such that

$$|\mathbf{a}^1| + |\mathbf{a}^2| = |\mathbf{a}^3| + |\mathbf{a}^4| = \frac{n}{2} \quad , \quad |\mathbf{a}^1| = |\mathbf{a}^3| \quad , \quad |\mathbf{a}^2| = |\mathbf{a}^4|$$

and

$$\text{either} \quad (L\mathbf{a}, U\mathbf{a}) = (\mathbf{a}^1, \mathbf{a}^4, \mathbf{a}^3, \mathbf{a}^2) \qquad \text{or} \quad (L\mathbf{a}, U\mathbf{a}) = (\mathbf{a}^3, \mathbf{a}^2, \mathbf{a}^1, \mathbf{a}^4)$$

Visual "Proof"



Complexity

- Finding the median in a bitonic sequence $\longrightarrow$ $O(\log n)$ steps
- Remark: this algorithm is no longer data-*independent*!
- Depth complexity: $\longrightarrow$ exercise / research
- Work complexity of the adaptive bitonic merger:
  - Number of comparisons

$$C(n) = 2C(\frac{n}{2}) + \log(n) = \sum_{i=0}^{k-1} 2^i \log(\frac{n}{2^i}) = 2n - \log n - 2$$

  - This is optimal!
  - Needs a trick to avoid actually copying the subsequences
    - Otherwise the total complexity of an ABM(n) would be O(n log n)
  - Trick = *bitonic tree* (see orig. paper for details)

How to find the median in a bitonic sequence

- We have
$$\text{median}(a) = \min(U\mathbf{a})$$
or
$$\text{median}(a) = \max(L\mathbf{a})$$
(depending on the definition of the median)
- Finding the minimum in a bitonic sequence takes $\log(n)$ steps

Overall Algorithm for Adaptive Bitonic Sorting

- Same as bitonic sorting, except we replace the half cleaner by
  1. Finding the median, and
  2. Swapping subsequences (only conceptually)

```
adaptiveBitonicSort( a0,..., an-1):
do parallel:
  sort a0,..., an/2-1  ascending
  sort an/2,..., an-1  descending
adaptiveBitonicMerge( a0,..., an-1)
```

```
adaptiveBitonicMerge( a0,..., an-1):
precond.: a0,..., an-1 is bitonic
find index q of median
swap subsequences as per theorem 2 and proof
do parallel:
  adaptiveBitonicMerge( a0,..., an/2-1)
  adaptiveBitonicMerge( an/2,..., an-1)
```

Topics for Master Theses

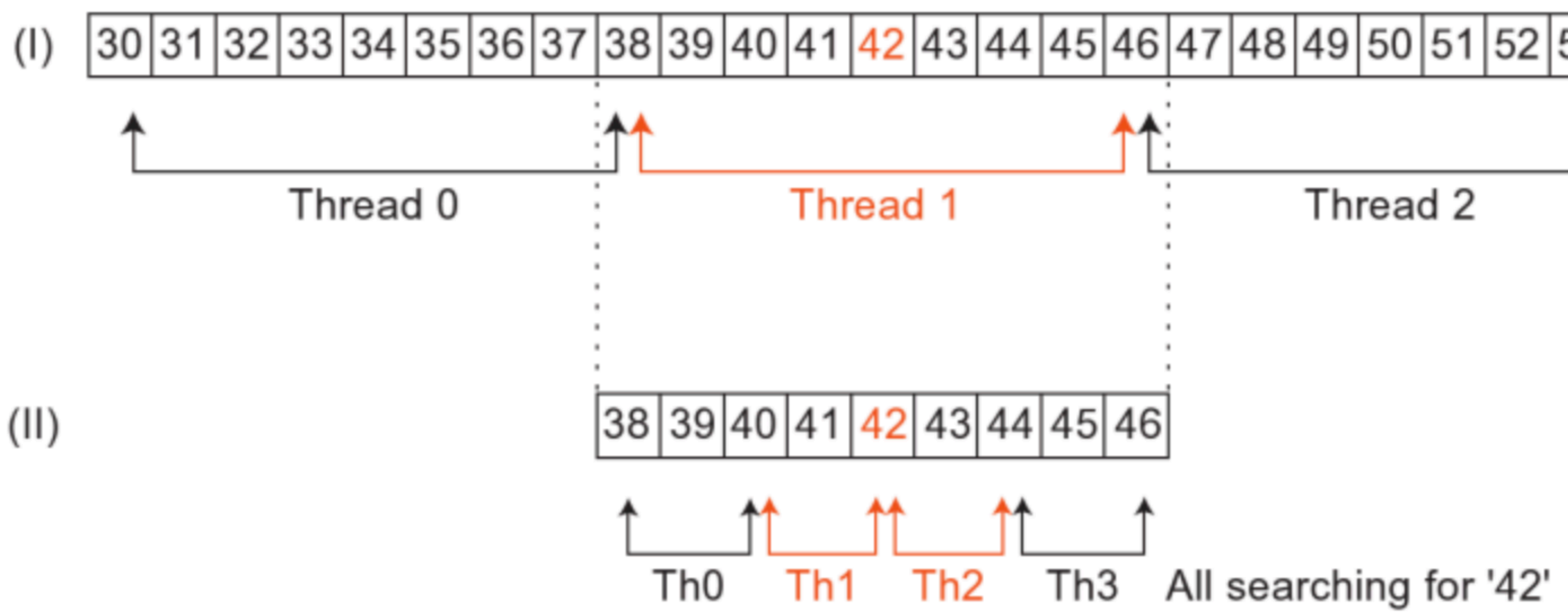
- Lots of different parallel sorting algorithms
- What is the performance of Adaptive Bitonic Sorting using CUDA?
- Do you love algorithms?
  - Thinking about them?
  - Proving properties?
  - Implementing them super-fast?
- Then we should talk about a possible master's thesis topic! 😀

Application: Searching

- Given a sorted array (should be really large, i.e., $\gg$1m elements)
- How to utilize the GPU for searching for keys in the array?
- Trivial solution, if you have a huge number of search requests:
  - Batch all requests into one multi-query
  - Each thread processes one request, doing binary search on the array for "their" key
    - Memory requests will be totally un-coalesced
  - Response time = similar to response for CPU-based search
  - Hardware utilization: usually, some threads will be finished early
  - Throughput: slower than CPU, since all threads must wait before next multi-query

P-Ary Parallel Search

- Given a sorted array, A, of n elements, and one key q
- With p threads, choose p pivot elements (not just one)
- Each thread i loads A[$i * n/p$] into shared memory
- Each thread i compares A[$i * n/p$] ≤ q ≤ A[$(i+1) * n/p$]
  - (For the last thread, use a sentinel element)
- Repeat with the bracket containing q (if any)



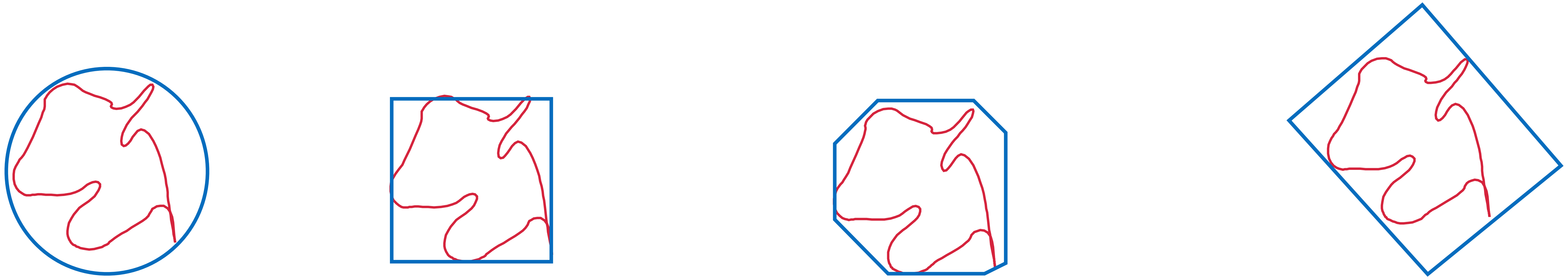
- Complexity: $O\left(\log_p n\right) = O\left(\frac{\log n}{\log p}\right)$

- A (potential) practical optimization: re-arrange data to match access patterns
  - If data can be re-arranged, move the p pivot elements of the first iteration to the front of the array $\longrightarrow$ coalesced memory access among the p threads
  - For each of the p segments, again move the p pivot elements *within that segment* to a contiguous segment in the array, etc.
  - Only useful, of course, if a huge number of queries are to be made

Application: BVH Construction

- Bounding volume hierarchies (BVHs): very important data structure for accelerating geometric queries
- Applications: ray-scene intersection, collision detection, spatial data bases, etc.
  - Database people usually call it "R-tree" ...

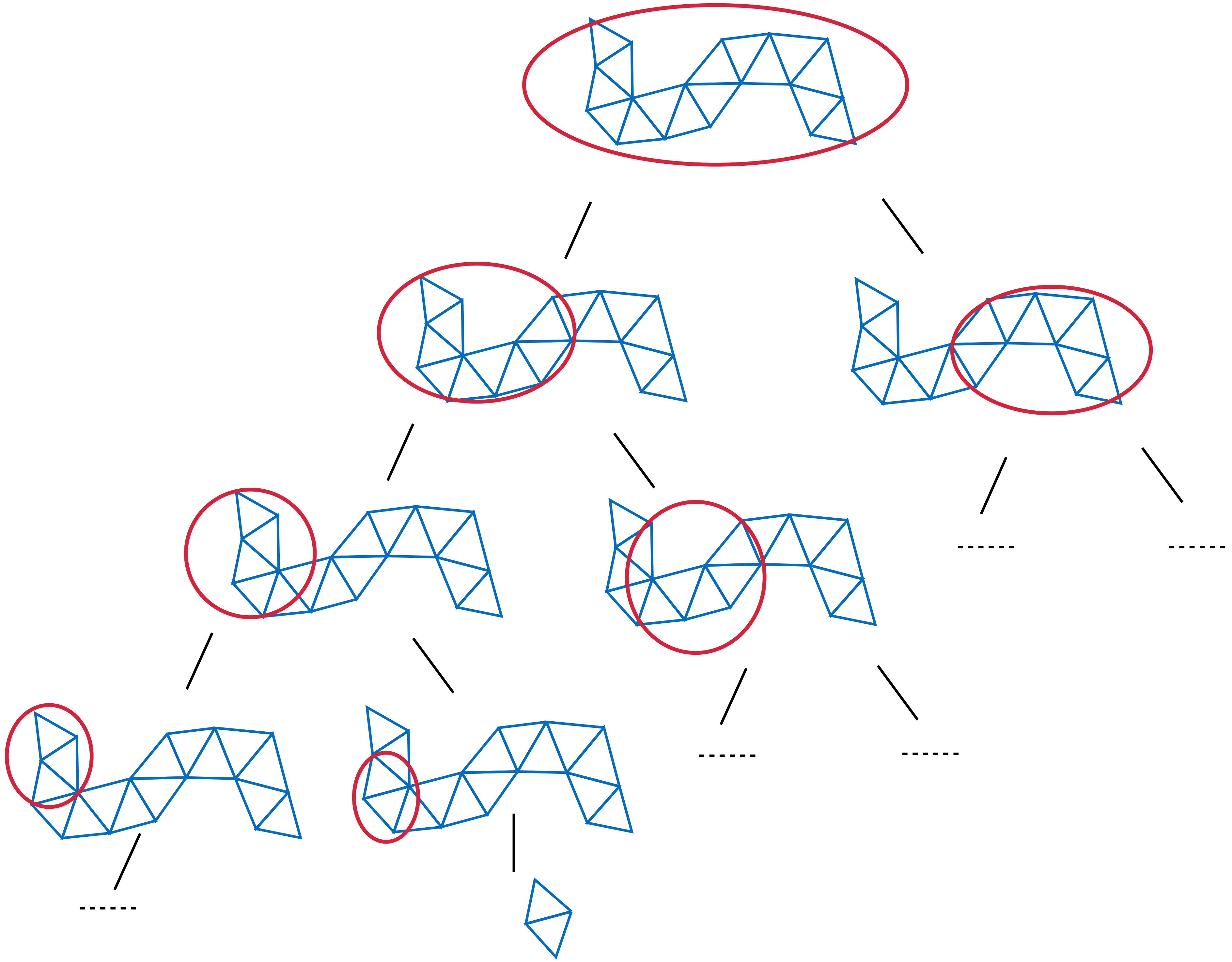
- Frequently used types of bounding volumes (BVs):



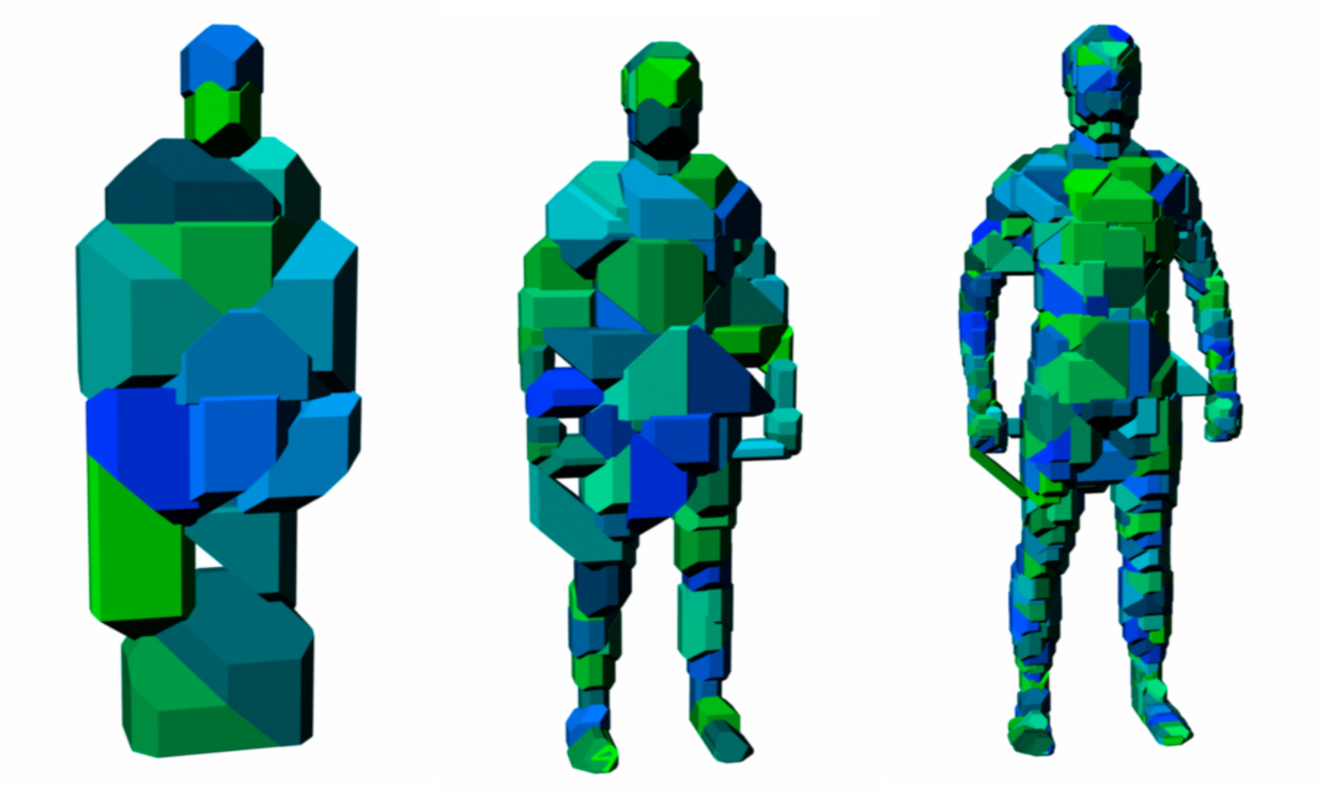
Sphere　　Box, AABB (R*-trees)　　k-DOPs / Slabs　　OBB (oriented bounding box)

The Notion of Bounding Volume Hierarchies

- Schematic example:



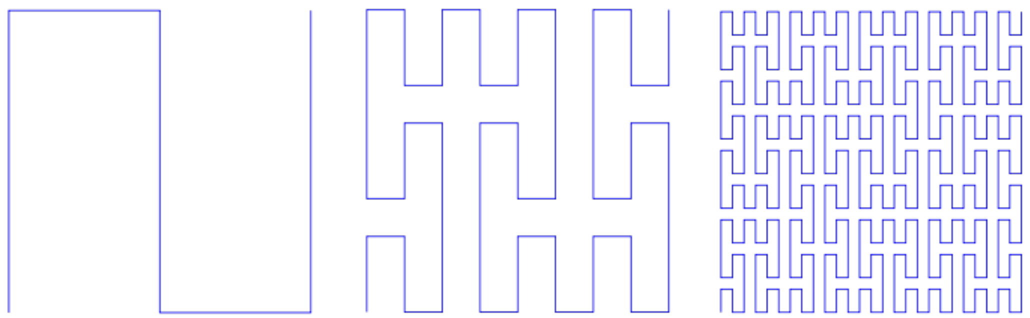
- Three levels of a k-DOP BVH:



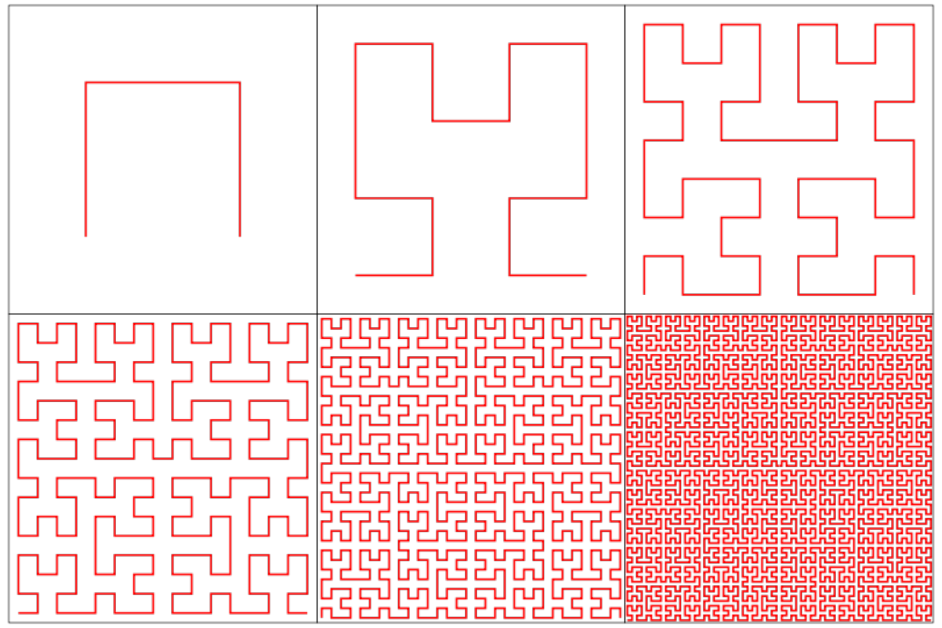
Parallel Construction of BVHs

- First idea: linearize 3D points/objects by a space-filling curve
- Definition curve:
  A curve (with endpoints) is a continuous function with its *domain* in the unit interval $[0, 1]$ and its *range* in some d-dimensional space.
- Definition space-filling curve:
  A space-filling curve is a curve with a range that covers the entire 2-dimensional unit square (or, more generally, an n-dimensional hypercube).

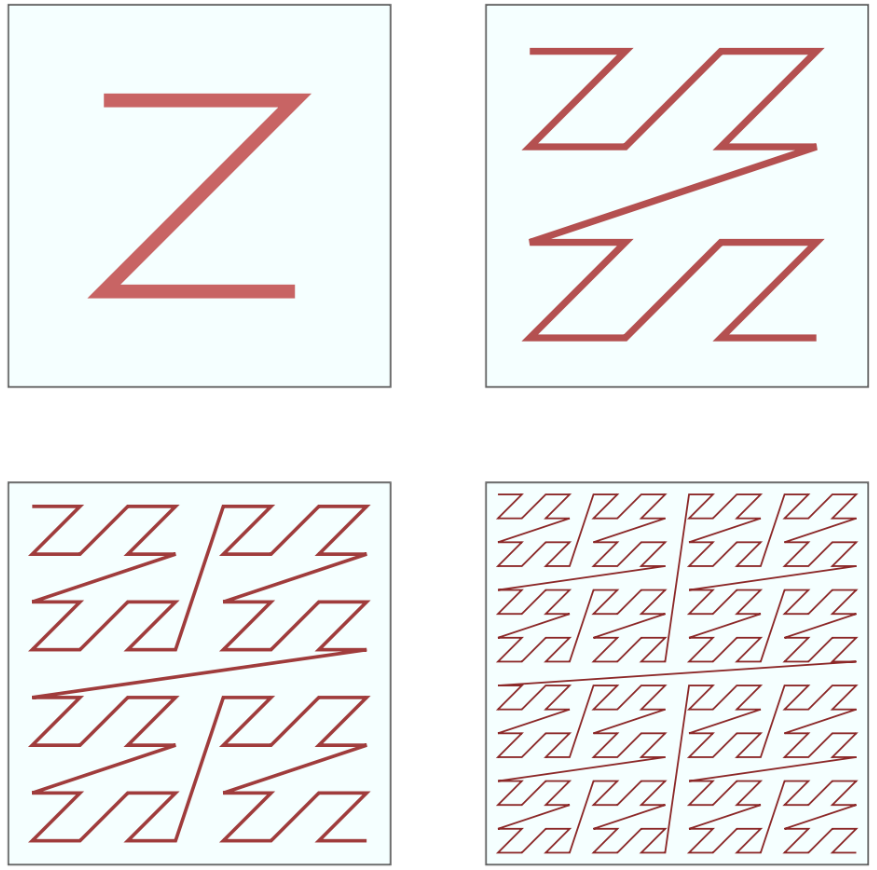
Examples of Space-Filling Curves (or, Rather, Approximations)



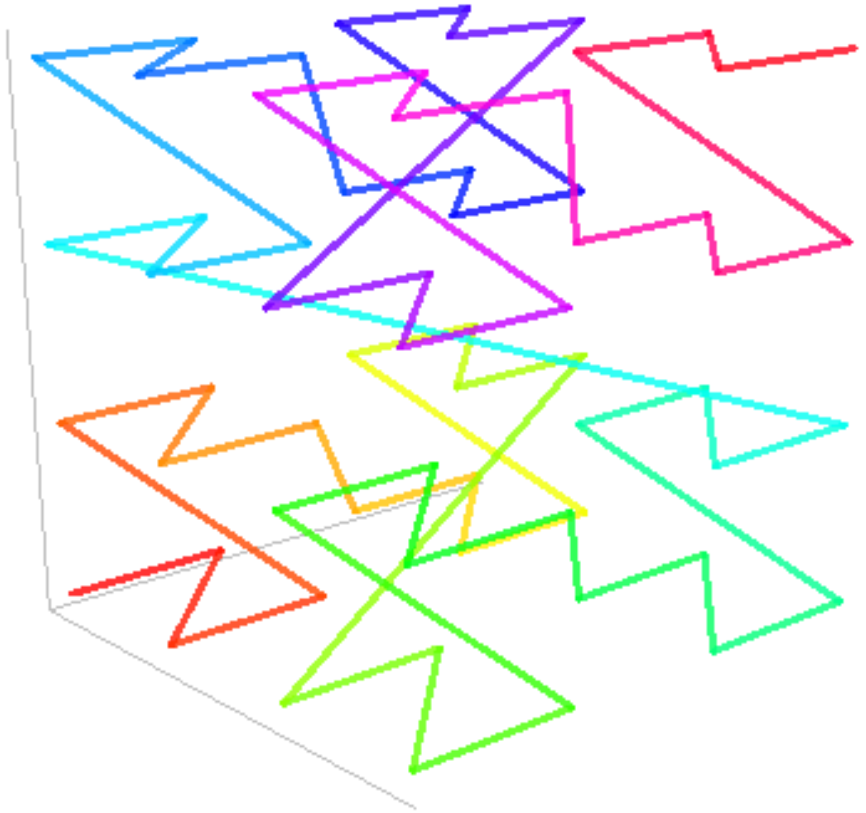
Peano curve



Hilbert curve

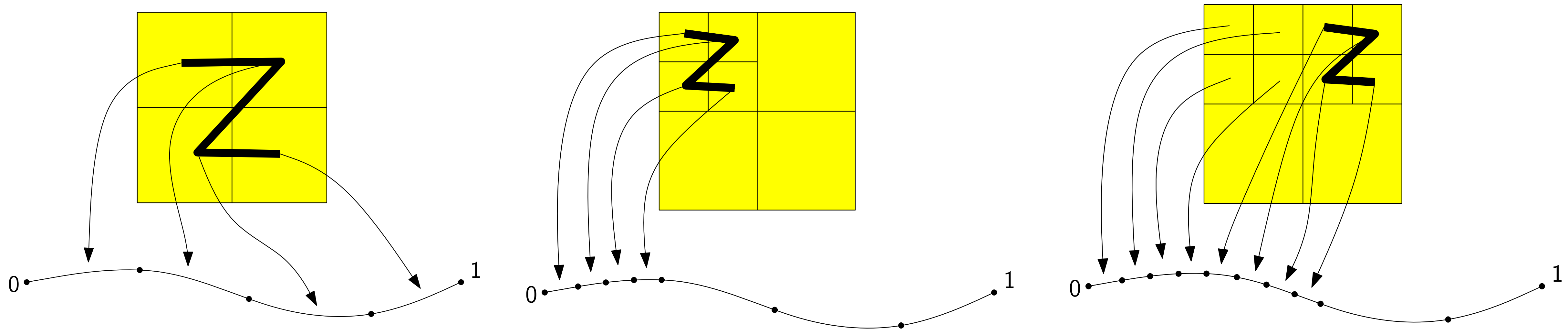


Z-order curve
(a.k.a. Morton curve)



Z-order curve in 3D

- Benefit: a space-filling curve gives a mapping for every point in the unit square onto a point in the unit interval



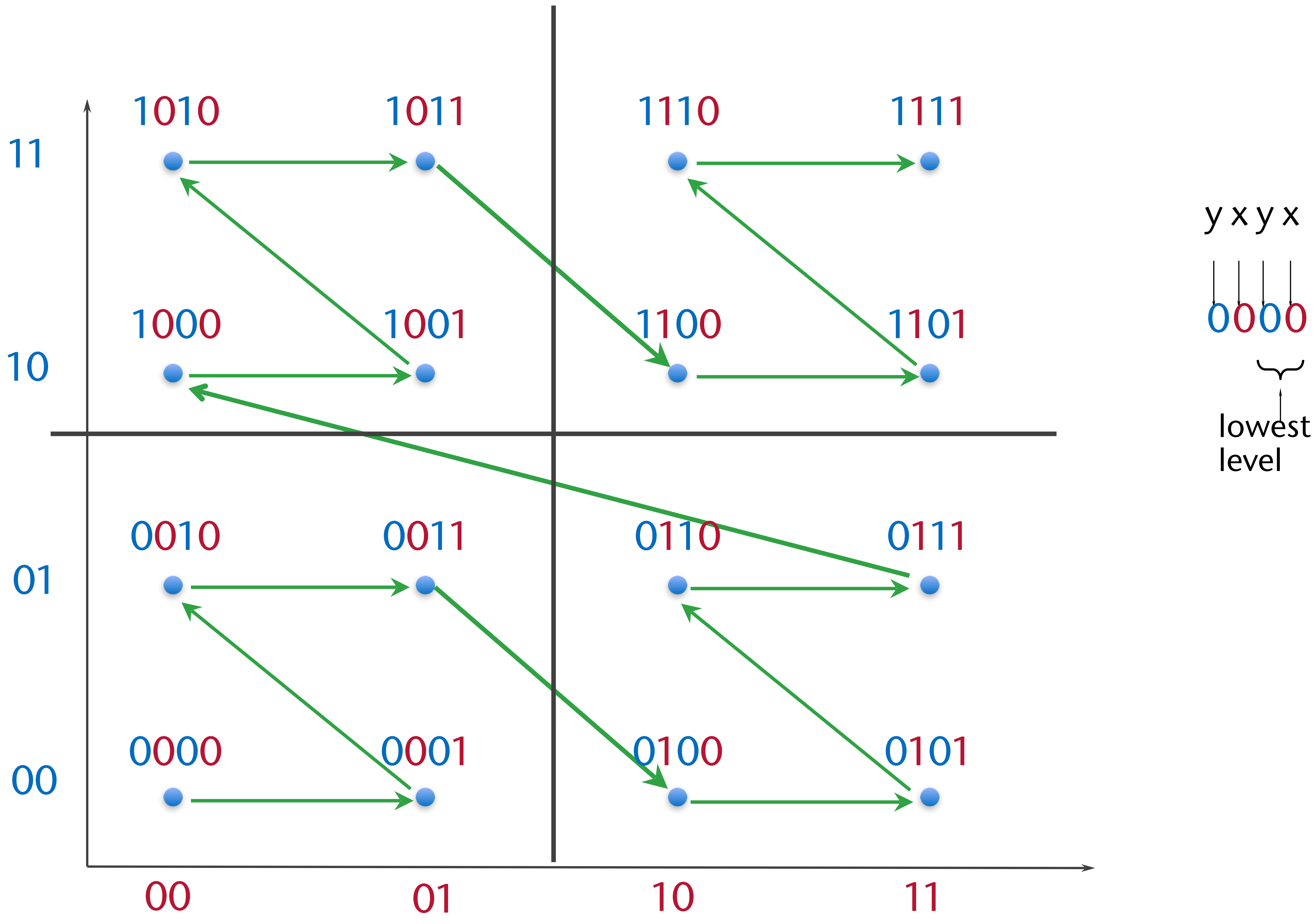
- At least, the limit curve does that …

- In practice, we can construct a "space-filling" curve only up to some specific (recursion) level, i.e., in practice space-filling curves are never really *space-filling*

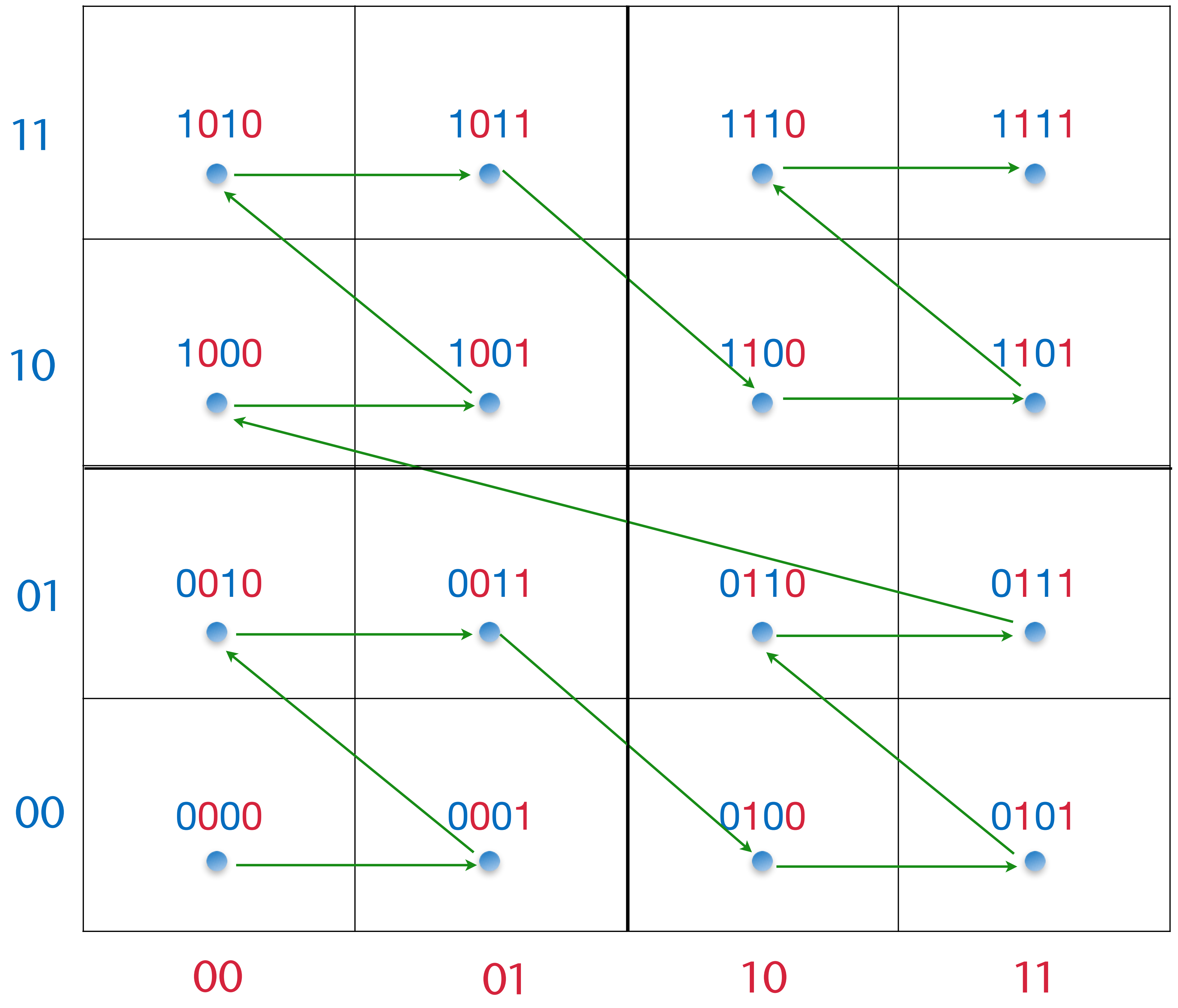
Construction of the Z-Order Curve in 3D

- Choose a level k
- Construct a regular lattice of points in the unit cube, 2^k points along each dimension
- Represent the coordinates of a lattice point p by integer/binary number, i.e., k bits for each coordinate, e.g. $p_x = b_{x,k} \ldots b_{x,1}$
- Define the Morton code of p as the interleaved bits of the coordinates, i.e., $m(p) = b_{z,k} b_{y,k} b_{x,k} \ldots b_{z,1} b_{y,1} b_{x,1}$
- Connect the points in the order of their Morton codes $\longrightarrow$ z-order curve at level k

Example (in 2D)

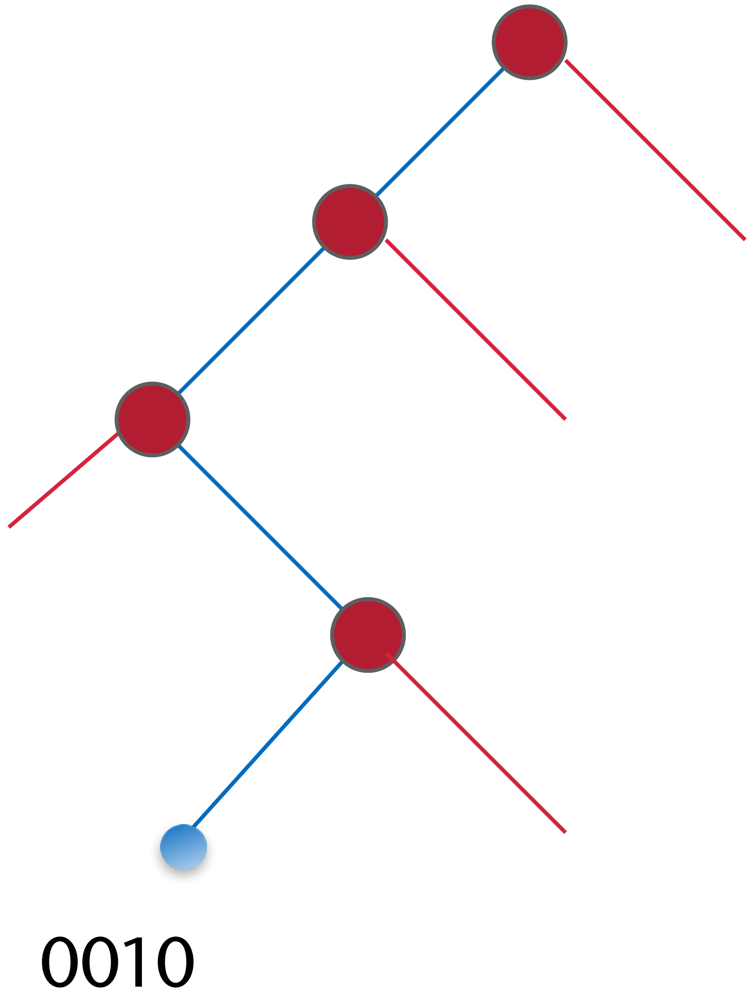


Note: the Z-curve induces a grid (actually, a complete quadtree)



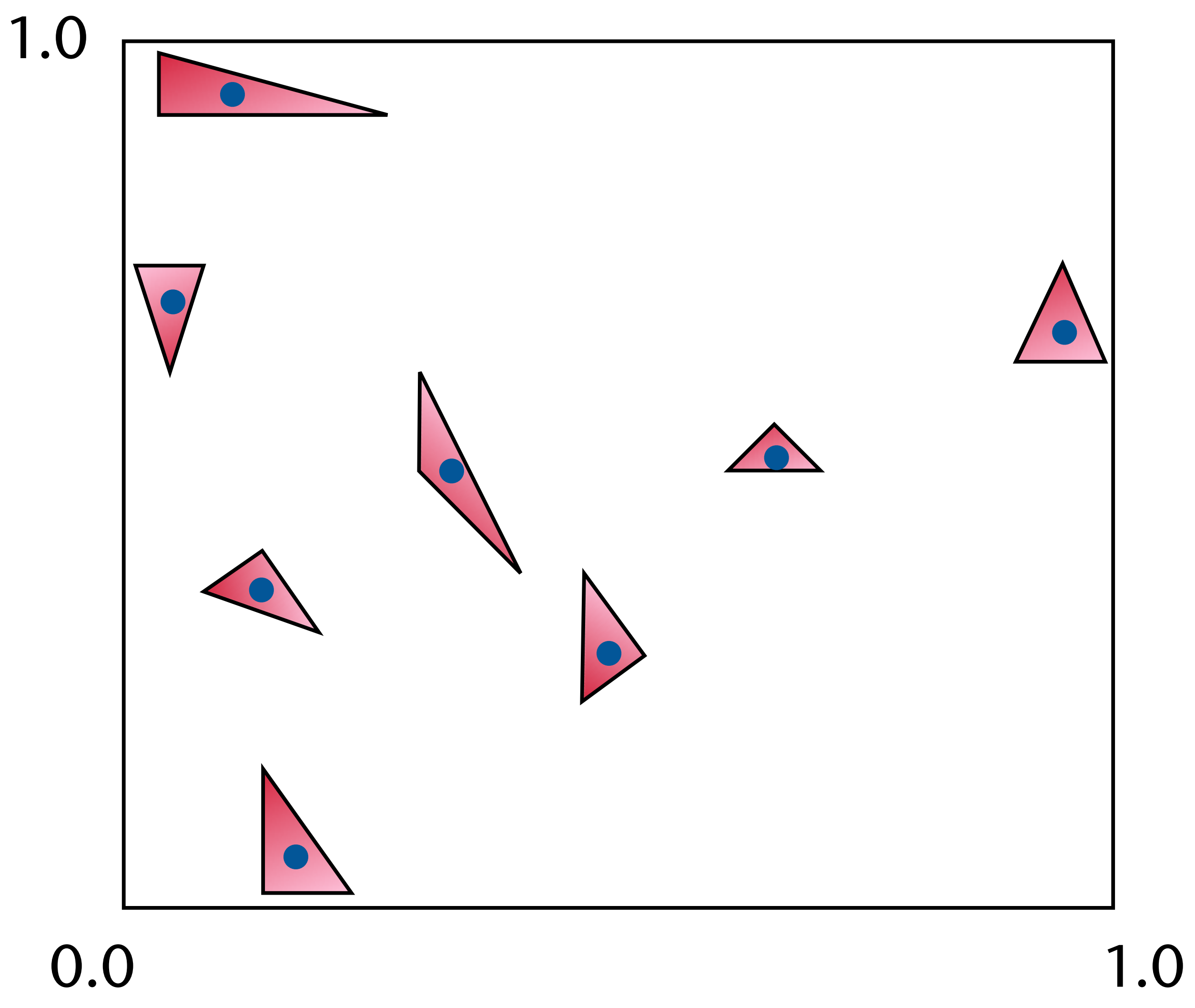
Properties of Morton Codes

- The Morton code of each point is $3k$ bits long (in 3D!)
- All points p with Morton code $m(p)$ = 0xxx lie below the plane z = 0.5
- All points with $m(p)$ = 111xxx lie in the upper right quadrant of the cube
- If we build a quadtree/octree on top of the grid, then the Morton code encodes the *path* of a point, from the root to the leaf that contains the point ("0" = left, "1" = right)
- The Morton codes of two points differ for the first time – when read from left to right – at bit position h $\Leftrightarrow$ the paths in the binary tree split at level h

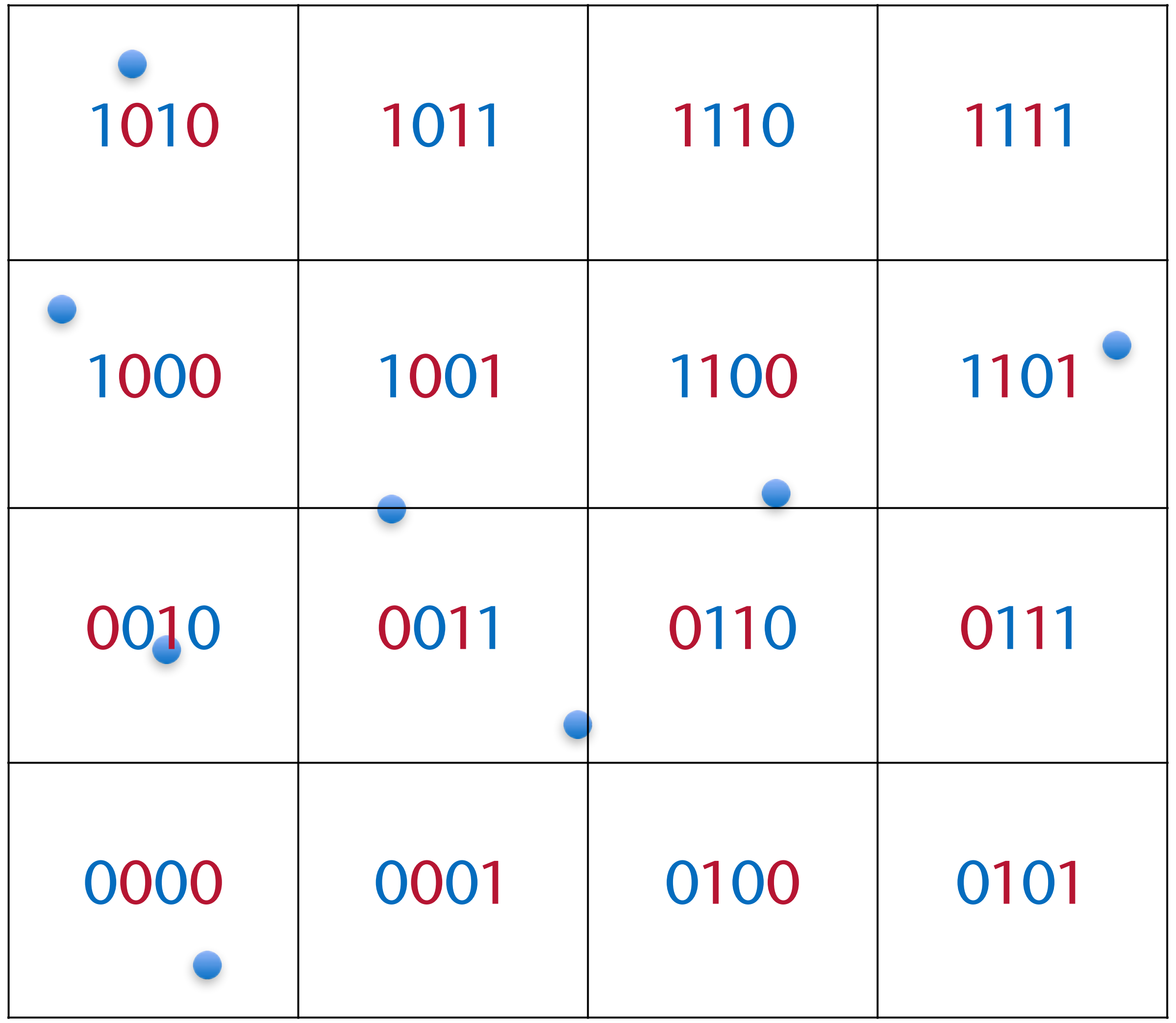


Construction of Linear BVHs

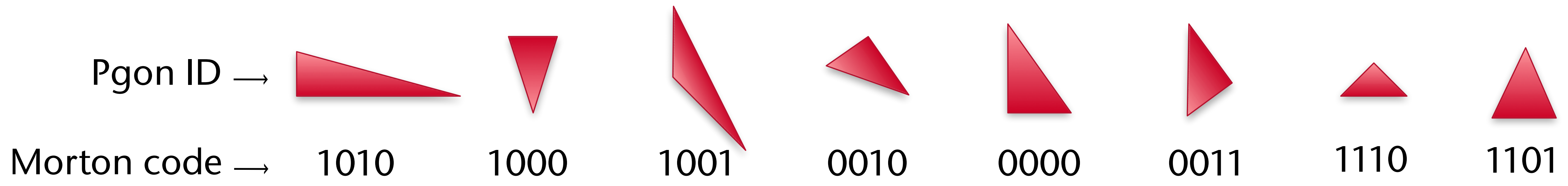
- Scale all polygons such that bbox = unit cube
- Replace polygons by their "center point"
  - E.g., center point = barycenter , or center point = center of bbox of polygon



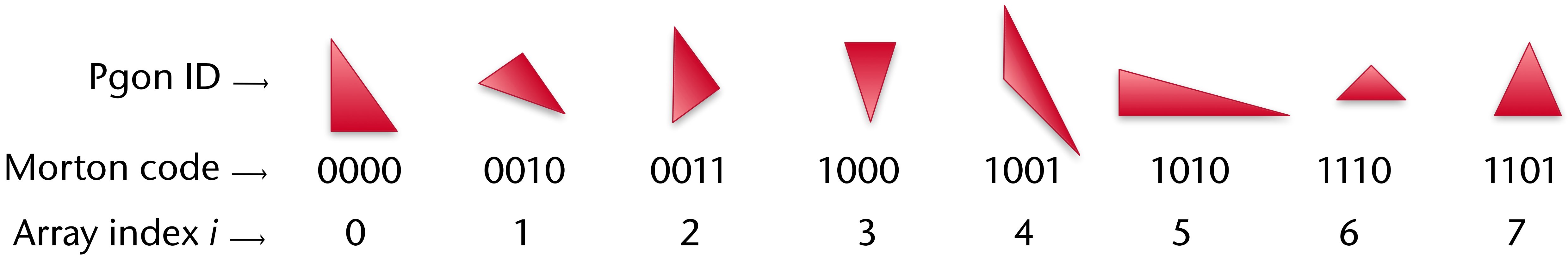
- Assign Morton codes to points according to their enclosing grid cell
- Assign those Morton codes to the original polygons, too



- Now, we've got a list of pairs of ⟨polygon ID, Morton code⟩
- Example:



- Sort list according to Morton code, i.e., along z-curve ⟶ linearization



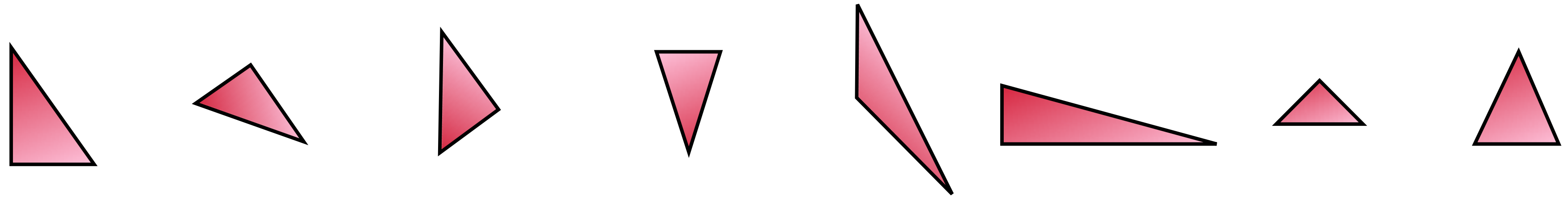
- Next: find index intervals representing BVH nodes at different levels

- Root of BVH = polygons in index range 0,…,N-1
  - All polygons with first bit of Morton code = 0/1 are below/above the plane z = 0.5, resp.
  - In the sorted array, find index i where first bit (MSB) changes from "0" to "1"
  - Left child of root = polygons in index range 0,…,i-1
  - Right child of root = polygons in index range i,…,N-1
- In general (recursive formulation):
  - Given: level h, and index range i,…,j in the sorted array, such that Morton codes are identical for all polygons in that range up to bit h
  - Find index k in $[i,j]$ where the bit at position h' ($h' > h$) in Morton codes changes from "0" to "1" (usually, $h' = h+1$)
- Can be achieved quickly by binary search and CUDA's `__clz()` function (= "count number of leading zeros")

- Consider arbitrary polygons at position i and i+1 in the array
- Condition for "same node":
  Polygons i and i+1 are in the same node of the BVH at level h $\Leftrightarrow$
  Morton codes are the same up to bit h (at least)
- Define a split marker := $\langle$index i, level $h\rangle$
- Parallel computation of all split markers $\longrightarrow$ "split list":
  - Each thread i checks polygons i and i+1
  - Compare their Morton codes from left to right $\longrightarrow$ h = left-most bit position where the two Morton codes differ
    - Can be calculated in one step using XOR and `__clz`
  - Output split markers $\langle i,h\rangle$, ..., $\langle i,3k\rangle$ (seems like a bit of overkill)
  - Can be at most 3k split markers per thread $\longrightarrow$ static memory allocations works

- Example:

Pgon ID ⟶

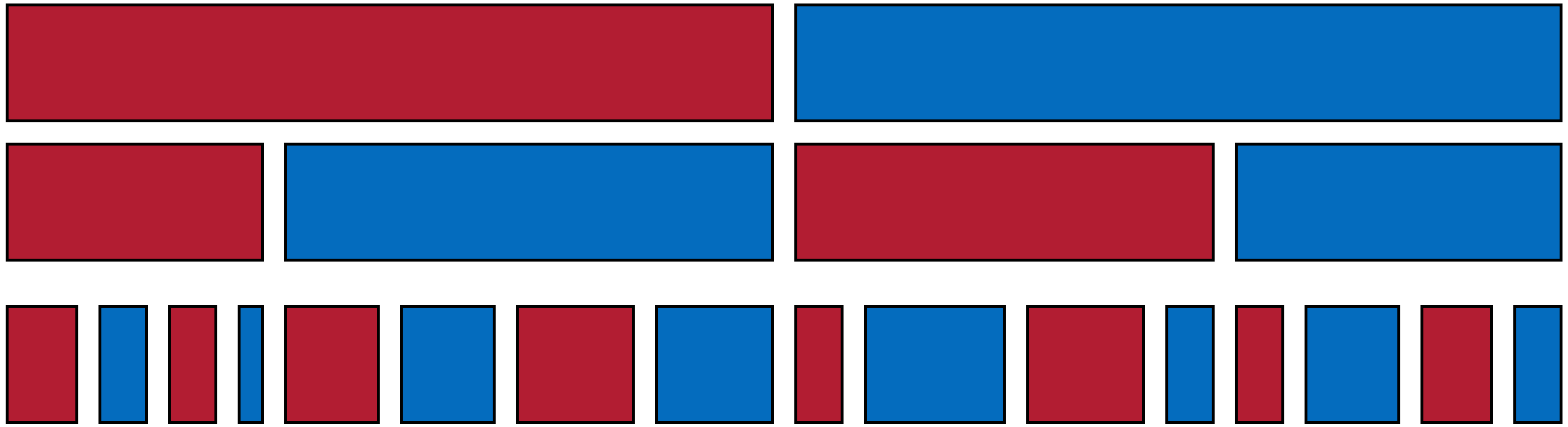


Morton code ⟶	0000	0010	0011	1000	1001	1010	1110	1101
Array index i ⟶	0	1	2	3	4	5	6	7

↓	↓	↓	↓	↓	↓	↓
(0,3)	(1,4)	(2,1)	(3,4)	(4,3)	(5,2)	(6,4)
(0,4)		(2,2)		(4,4)	(5,3)	(6,3)
		(2,3)			(5,4)	
		(2,4)				

Split marker = (i,h) , $i \in [0,N\text{-}1]$, $h \in [1,3k]$

- Last steps:
  1. Compact split list
  2. Sort split list by level *h*
     - Must be a stable sort!
- For each level *h*, we now have ranges of indices into the array of polygons; all primitives within a range are in the same node on that level *h*

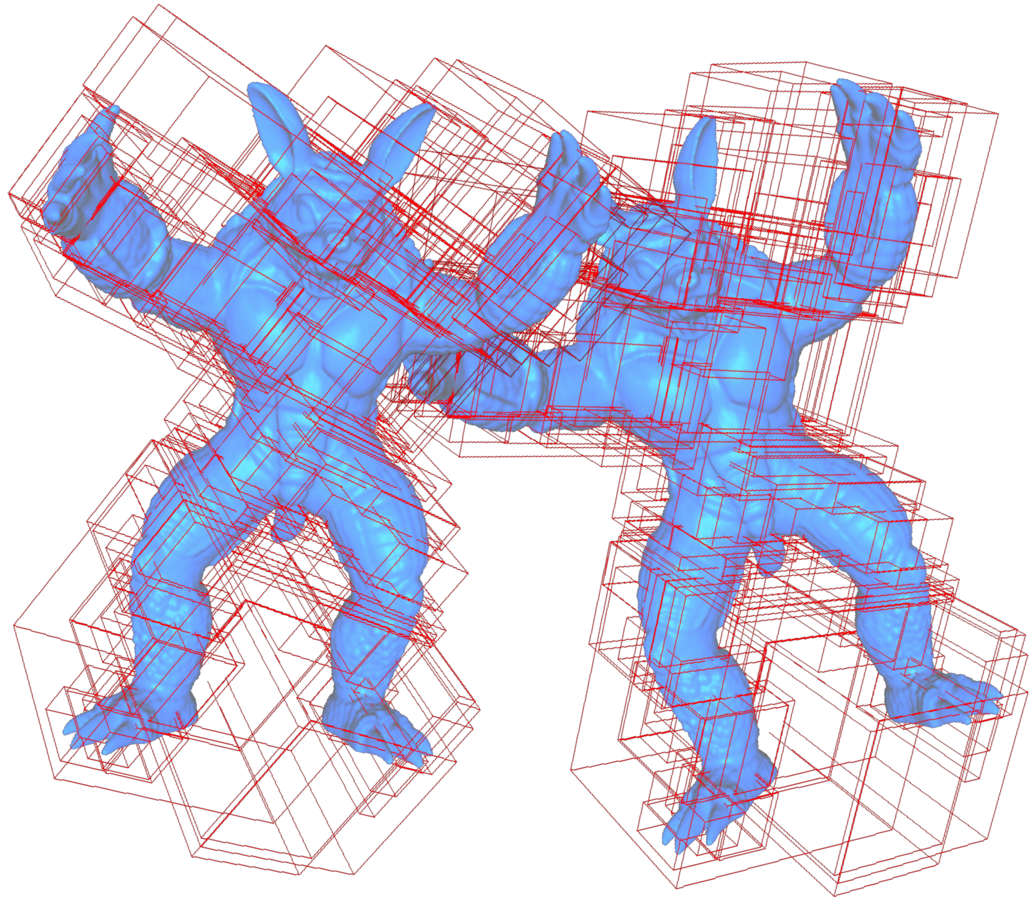


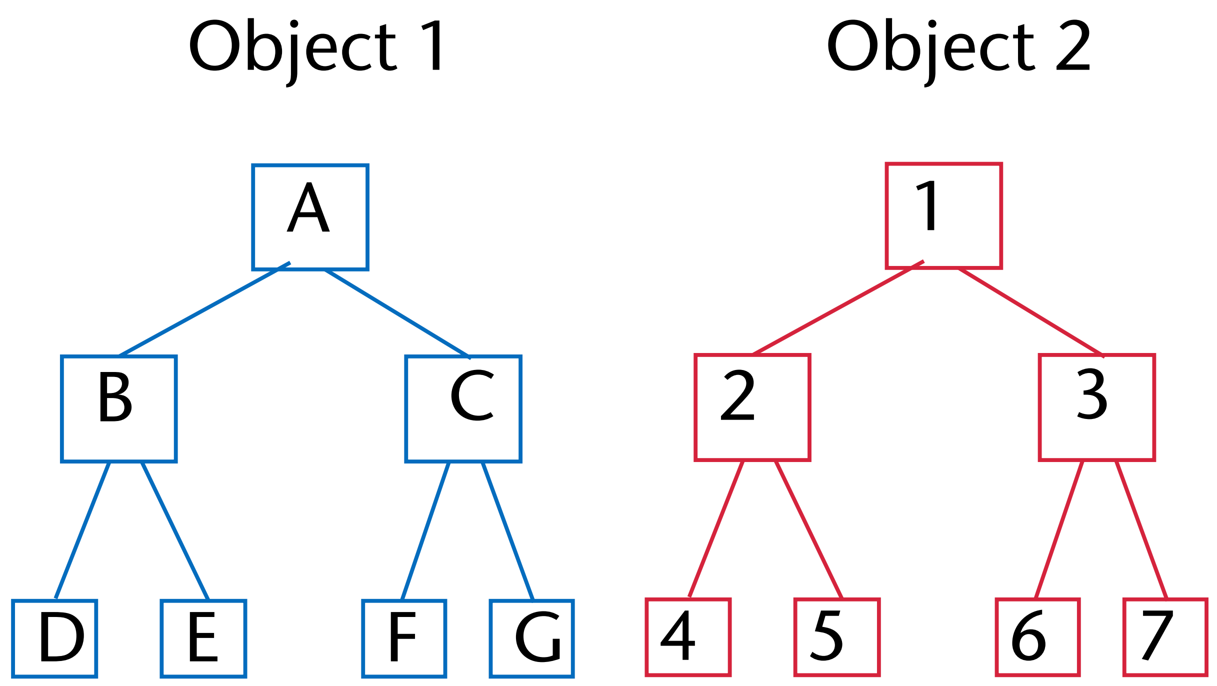
- Final steps:
  - Convert to "regular" BVH with pointers
  - Remove singleton BVH nodes
  - Compute bounding boxes for each node (i.e., interval)
- Challenge: can you use the array of split markers directly
  - Maybe, need to add additional pointers/indices to point to child "nodes"?
  - How would a ray traversal through this kind of BVH work?
  - Could you even use it for collision detection, i.e., simultaneous traversal of 2 BVHs?
- Limitations:
  - Not optimized for ray tracing
  - Morton code only *approximates* locality

Master's Thesis! :-)

Example Application of BVHs: Collision Detection

FYI





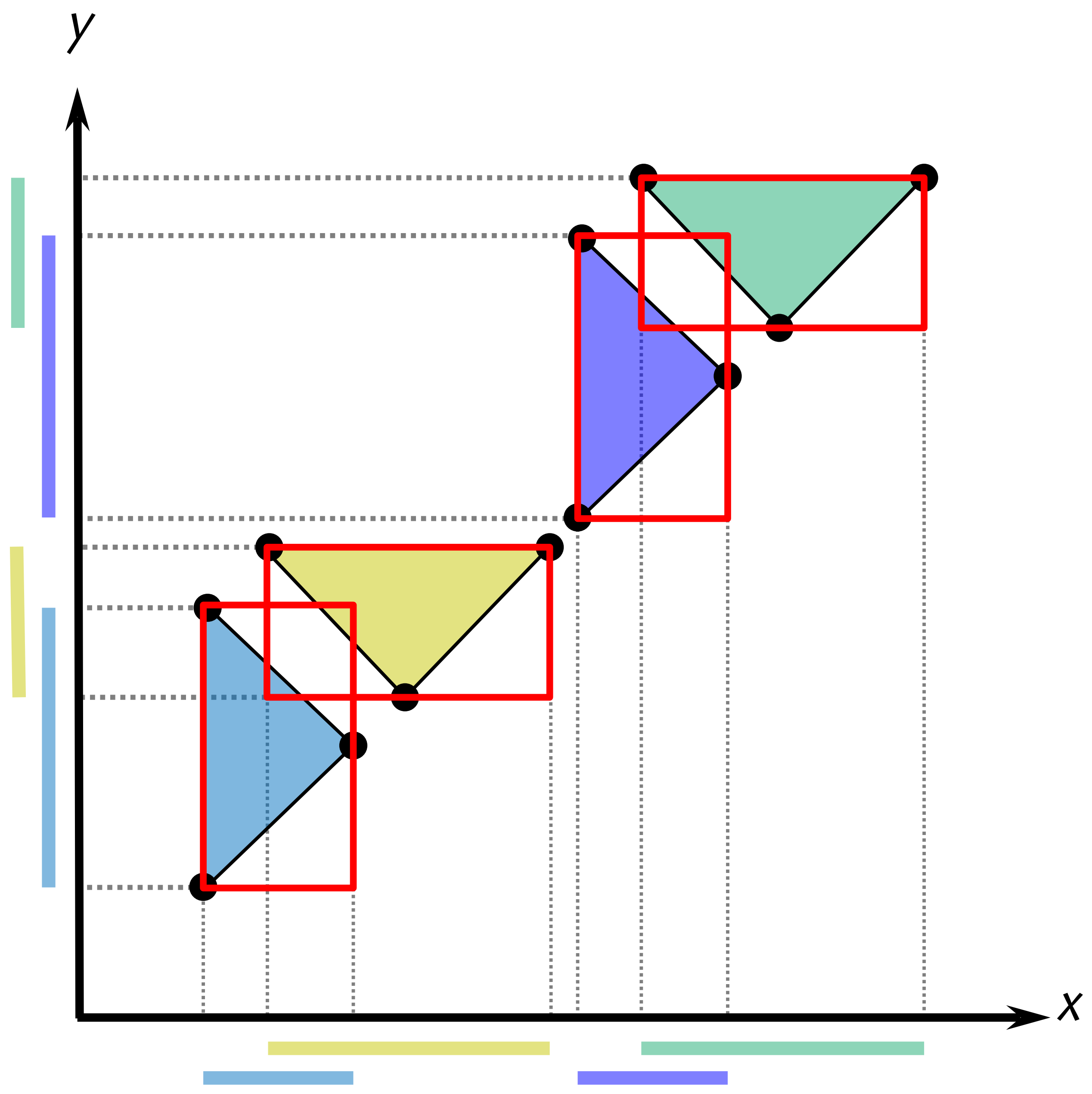

```
traverse( node X, node Y ):
if X,Y do not overlap then:
    return
if X,Y are leaves then:
    check all pairs of polygons
else
    for all children pairs do:
        traverse( Xi, Yj )
```

Application of Collision Detection (Video)

Collision Detection Without Auxiliary Data Structures

- Goal: collision detection of *deformable* objects
- Consequence: auxiliary (acceleration) data structure could potentially slow down the whole method
- Given: a large set of AABB's (each enclosing one polygon)
- Sought: pairs of AABB's that intersect (overlap)
  - Potentially intersecting pairs of polygons
  - Could be boxes of different objects $\longrightarrow$ regular collision detection
  - Could be boxes of same object $\longrightarrow$ self-collision / self-intersection
- Simplification here: ignore problem with pairs of boxes where triangles are adjacent in the same mesh
  - Need to be filtered before doing the actual intersection tests

- General idea: dimension reduction by plane sweep
  - Sweep plane through space along an axis
  - Consider only boxes that intersect that plane
  - Check intersection of those boxes in 2D
- Alternative description:
  - Project all boxes onto the (sweep) axis
    $\longrightarrow$ set of intervals
  - Find pairs of intervals that overlap
- Sweep/projection axis can be chosen arbitrarily



The Algorithm

```
parallel for all triangles:
  compute AABB
sort all end points Sᵢ and Eᵢ of all AABBs
  in one common array
  (key = x-coord., value = triangle ID)
create list C of overlapping intervals                  (x)
parallel for all pairs in C:                            (xx)
  perform complete AABB overlap test
  if no overlap: remove pair from list C
perform stream compaction on C
parallel for all triangle pairs (Ti, Tj) in C:
  if (Ti, Tj) share an edge: remove pair from C
  if (Ti, Tj) do not intersect: remove pair from C
perform stream compaction on C
output C
```

Remark: we can perform steps (x) and (xx) at the same time, shown here as separate steps for clarity

Step (x): create list C of overlapping intervals

- Idea:
  - Consider all starting points S_i
  - Find all intervals $[S_j, E_j]$ with $S_i \in [S_j, E_j]$
  - Do not consider the endpoints E_i , otherwise each overlapping pair is found twice
- Naïve parallelization: one thread per triangle
  - Thread starts at position i of "its" S_i in the sorted array of start/end points
  - Scans array from there to the right
- Goal for parallelization: one thread per overlapping pair
- Problem: number of threads and amount of memory for C is unknown

- Trick: prefix sum over a flags array
  - Similar to "split" in radix sort
- Also, all triangles know their start/end index in the sorted array of endpoints
- Number of potentially overlapping intervals / boxes = $P[E_i] - P[S_i] - 1$ $\longrightarrow$ number of threads per triangle i
- Reduction yields total number of threads = max length of array C

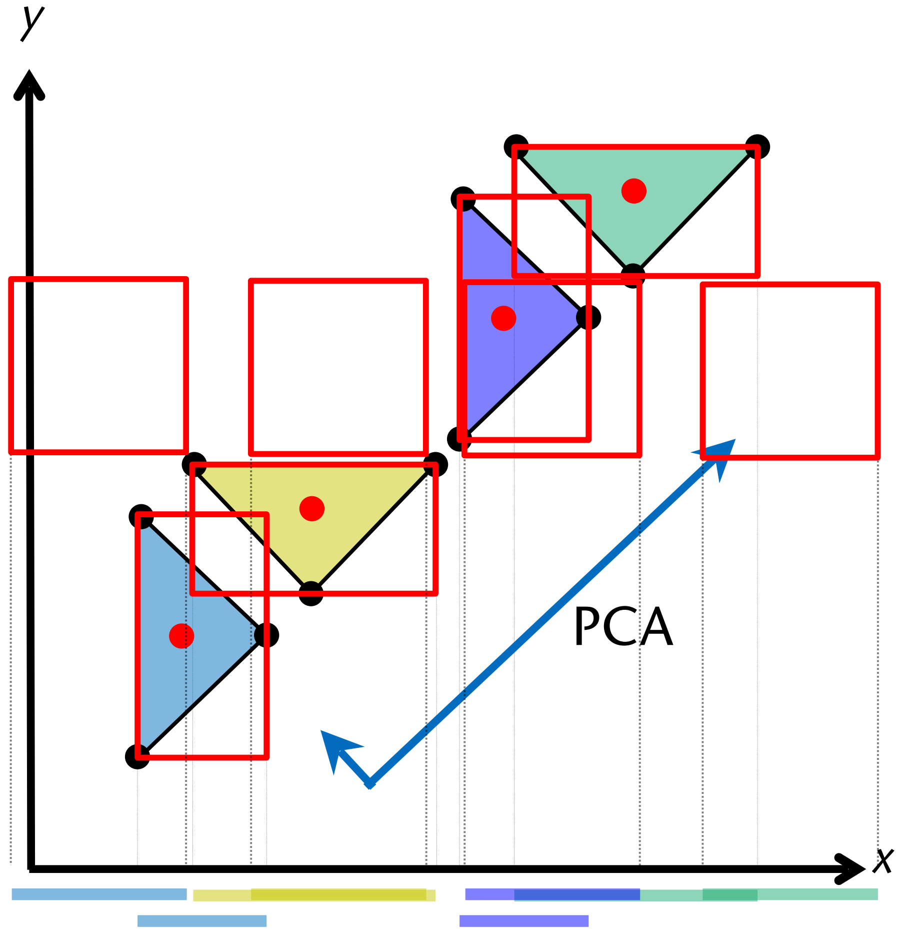
Array indices	0	1	2	3	4	5	6	7	
Sorted array of start/end interval endpoints	S_A	S_C	S_B	E_C	E_A	E_B	S_D	E_D	…
Start/end flags	1	1	1	0	0	0	1	0	…
Prefix sum, P	0	1	2	3	3	3	3	4	…

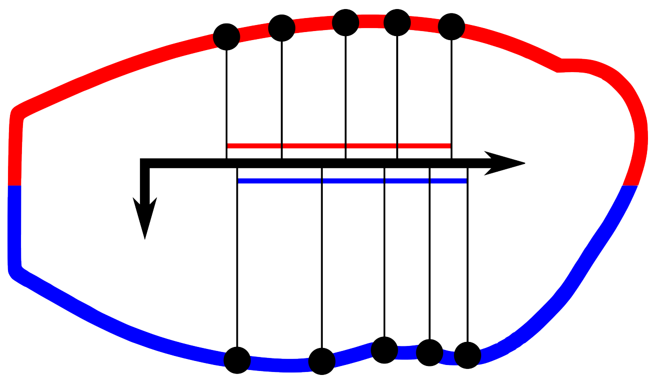
Triangle ID	A	B	C	D
Start index	0	2	1	6
End index	4	5	3	7

A	B	C	D	
$3-0-1$	$3-2-1$	$3-1-1$	$4-3-1$	
2	0	1	0	$= 3$

Extension: the Cluster-PCA-Based Sweep Plane Method

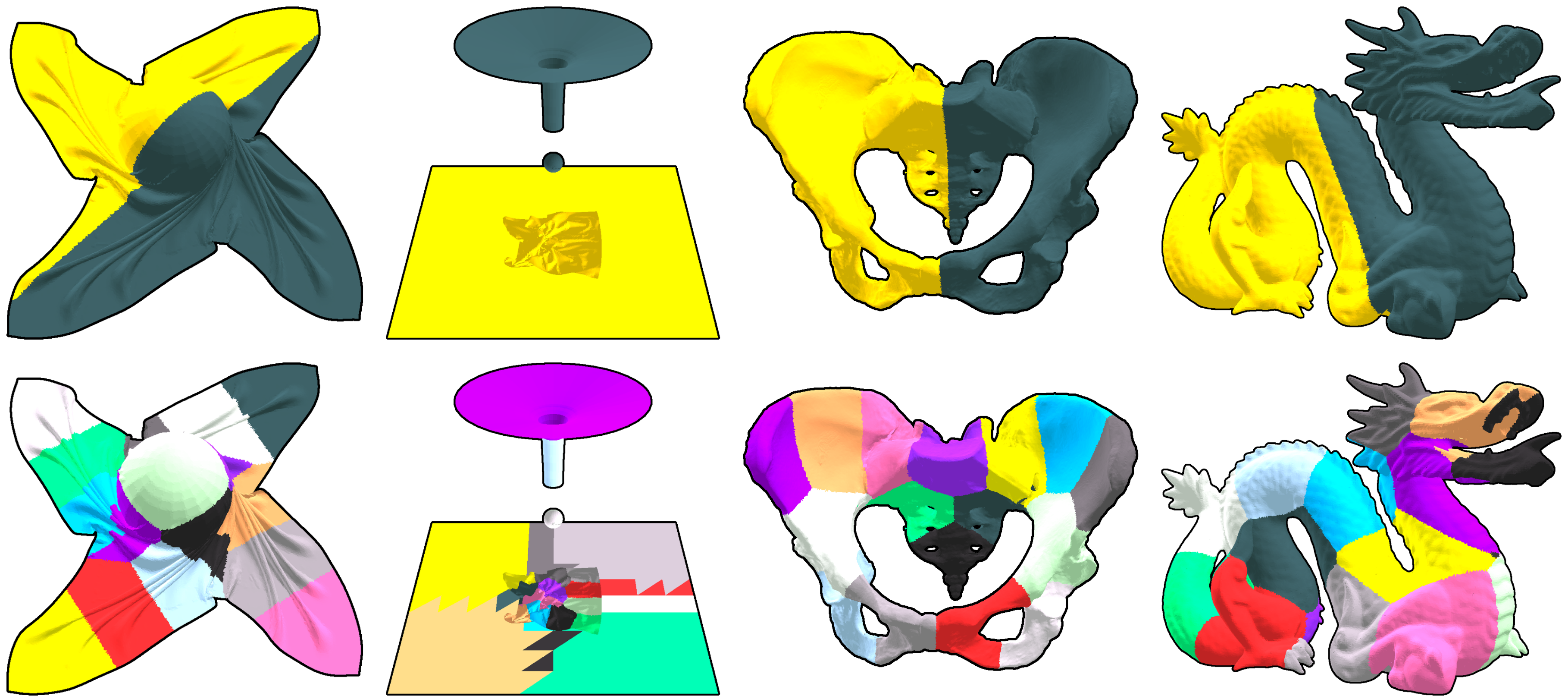
- Problem: sweep plane method ⟶ dimension reduction by projection ⟶ potentially many false positives
- Idea: utilize fact that the sweep/sorting axis can be chosen arbitrarily
- Use axis such that number of overlapping projected AABBs is minimized ⟶ heuristic: longest axis of PCA (= axis of largest variation)
- Further problem: could still produce lots of false positives





FYI

- Second idea to further reduce false positives: partition objects into clusters, perform previous method in parallel for all clusters



2 clusters

16 clusters

- Problem: need to find overlapping AABBs between clusters, too
- Solution: assign polygons along cluster borders to both clusters
- Method: fuzzy c-means (variant of k-means algo)

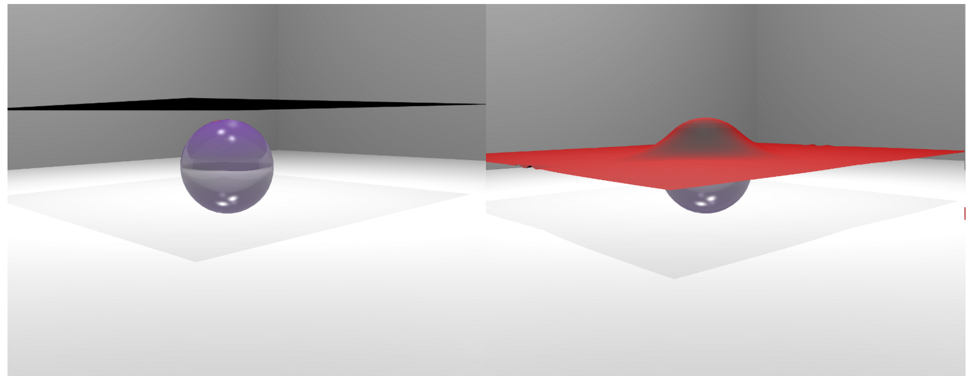
Overall Algorithm

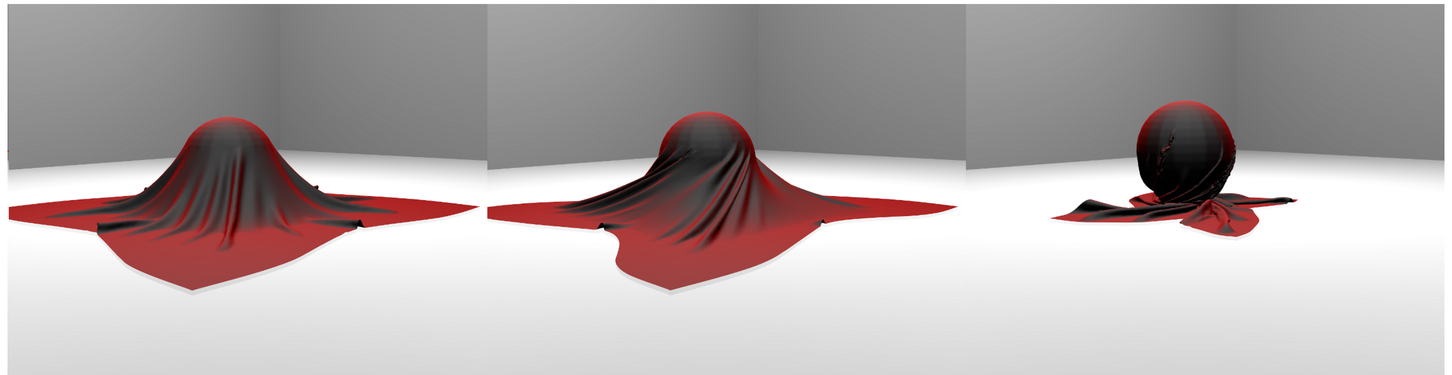
FYI

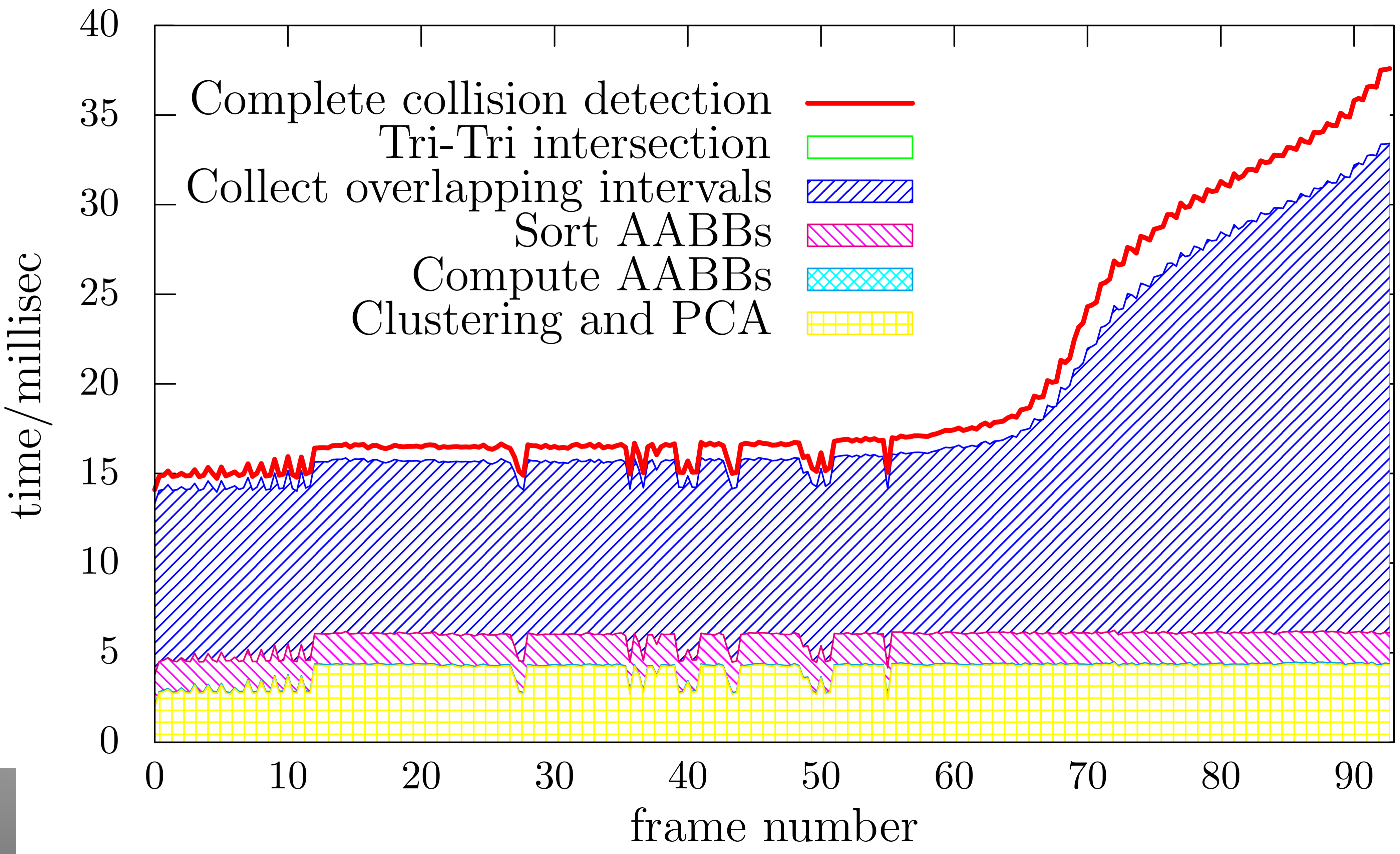
```
parallel for all triangles:
  compute center points
subdivide scene into c (overlapping) clusters
parallel for all clusters:
  compute PCA
  transform all points into PCA coord. system
  perform rest of collision detection as before
```

Results: Cloth on Ball Benchmark

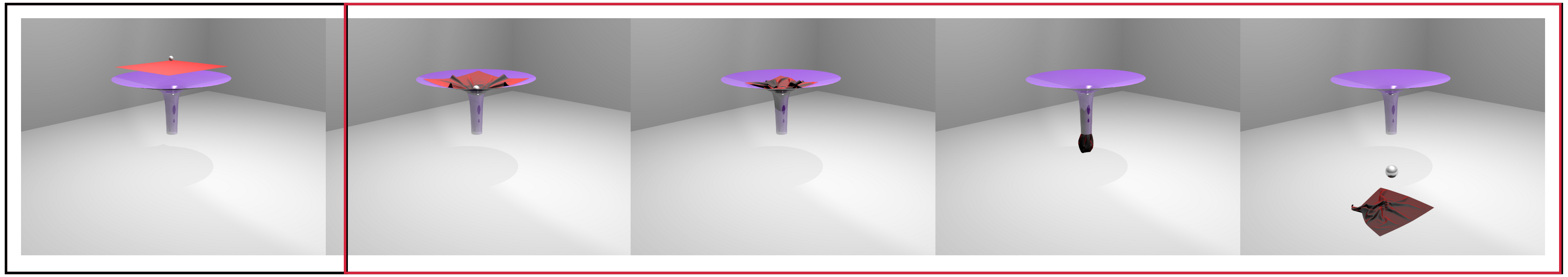
- Cloth (92k triangles) drops down on a rotating ball (760 triangles)



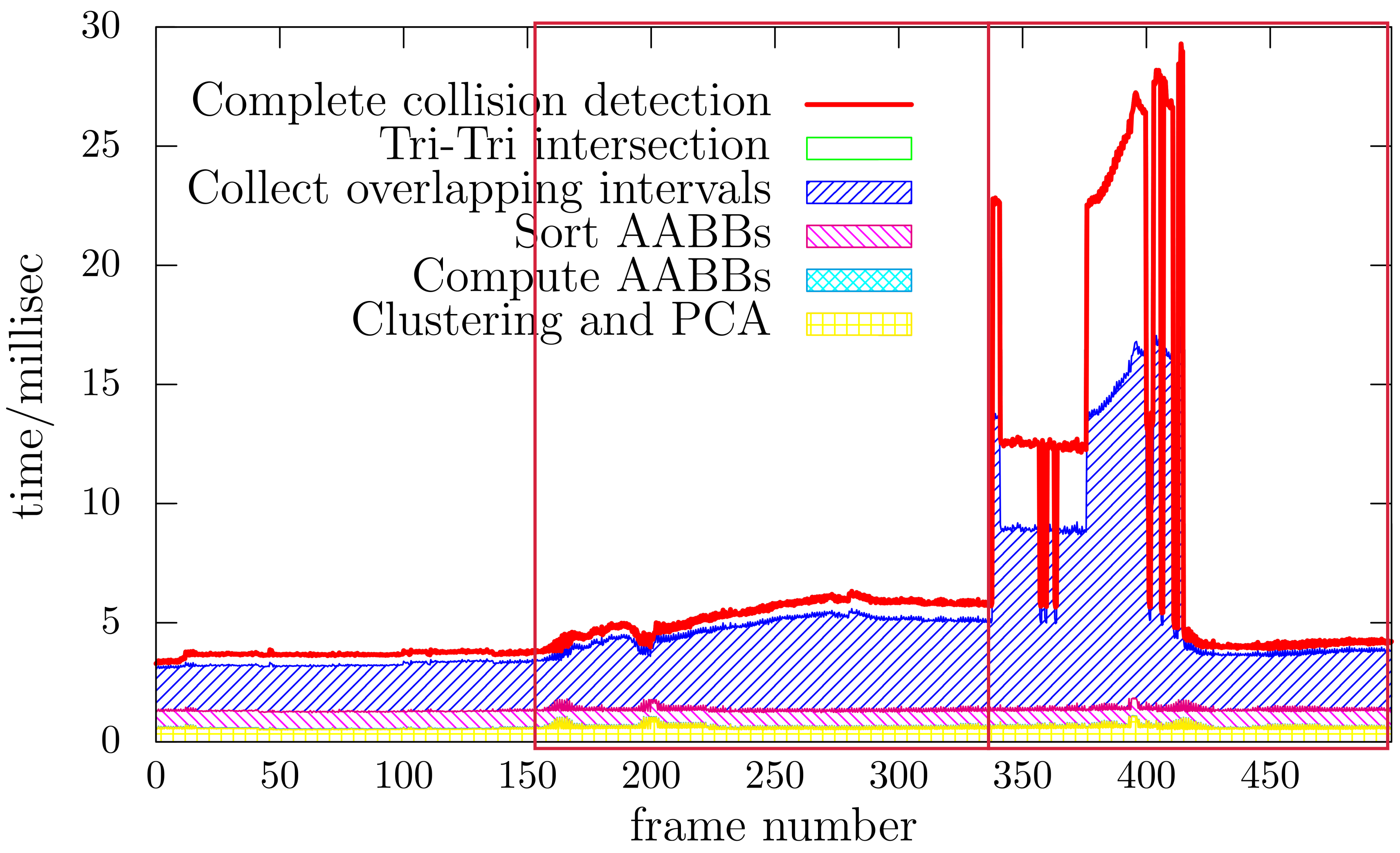




Results: Funnel Benchmark



Ball (1.7k triangles) pushes a cloth (14k triangles) through a funnel (2k triangles)



Demo Movie