

Syllabus

3. Sorting

- Specification
 - Bubble Sort
 - Selection Sort
 - Insertion Sort
 - Merge Sort
 - Quicksort
 - Pivoting
 - Sorting in Linear Time
 - Sorting in Parallel
- Specification
 - Primitives:
semantics and cost
 - Design
 - Analysis
 - Optimality
- of Comparison-
Based Sorting**

3. Sorting

specification

Specification: Fix set X with total order \leq .

Input: $N \in \mathbb{N}$ and finite sequence $x_1, \dots, x_N \in X$ in array $x[1..N]$
and array $\pi[1..N] :=$ identity permutation of $\{1, \dots, N\}$

Output: Permutation π of $\{1, \dots, N\}$ such that $x_{\pi(1)} \leq \dots \leq x_{\pi(N)}$

Primitives: ordered comparison “ $x[n] \leq x[m]?$ ” cost 1
Index integer arithmetic cost 1
swapping $x[n] \leftrightarrow x[m]$ cost 1
swapping $\pi[n] \leftrightarrow \pi[m]$ cost 1

Preprocessing data in order to accelerate queries.

3. Sorting

Bubble Sort

Input: $N \in \mathbb{N}$ and array $x[1 \dots N]$

Output: *Permuted* array $x \circ \pi$
s.t. $x_{\pi(1)} \leq \dots \leq x_{\pi(N)}$

Primitives:

Comparison “ $x[n] \leq x[m]$?”

Swapping $x[n] \leftrightarrow x[m]$

Index integer arithmetic

```

Procedure BubbleSort (  $x[N]$  )
  For  $m := N$  downto 2 do
    For  $k := 1$  to  $m-1$  do ←
      If  $x[k] > x[k+1]$  then
        Swap (  $x[k]$  ,  $x[k+1]$  )
      Endif
    Endfor ←
  Endfor ←

```

6 5 3 1 8 7 2 4

runtime $O(N^2)$

Correctness: $x[1], \dots, x[m] \leq x[m+1] \leq \dots \leq x[N]$

3. Sorting

Select Sort

Procedure **SelectSort** ($x[N]$)

For $m := 1$ to $N-1$ do

$min := m;$

For $k := min+1$ to N do

If $x[k] < x[min]$ then

$min := k$; Endif

Swap($x[m], x[min]$)

Endfor

Endfor

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Input: $N \in \mathbb{N}$ and array $x[1...N]$

Output: *Permuted* array $x \circ \pi$

s.t. $x_{\pi(1)} \leq \dots \leq x_{\pi(N)}$

Primitives:

Comparison “ $x[n] \leq x[m]$?”

Swapping $x[n] \leftrightarrow x[m]$

Index integer arithmetic

runtime $O(N^2)$

Correctness: $x[1] \leq \dots \leq x[m-1] \leq x[m], \dots x[N]$

3. Sorting

6 5 3 1 8 7 2 4

Procedure **InsertSort** ($x[N]$)

For $m := 2$ to N do

$y := x[m];$ $k := m - 1;$

While $k > 0$ and $x[k] > y$

$x[k+1] := x[k]$

$k := k - 1$

Endwhile

$x[k+1] := y$

Endfor

Insert Sort

Input: $N \in \mathbb{N}$ and array $x[1 \dots N]$

Output: *Permuted* array $x \circ \pi$

s.t. $x_{\pi(1)} \leq \dots \leq x_{\pi(N)}$

Primitives:

Comparison “ $x[n] \leq x[m]$?”

Swapping $x[n] \leftrightarrow x[m]$

Index integer arithmetic

runtime $O(N^2)$

Correctness: $x[1] \leq \dots \leq x[m-1] \leq x[m], \dots x[N]$

3. Sorting

Merge Sort

Procedure **MergeSort** ($x[N]$)

If $N \leq 1$ return.

$l := \lfloor N/2 \rfloor$; $r := \lceil N/2 \rceil$; array $y[l]$, $z[r]$;

For $m := 1$ to l do $y[m] := x[m]$;

For $m := 1$ to r do $z[m] := x[l+m]$;

MergeSort(y); MergeSort(z);

While $l > 0$ and $r > 0$ do ; If $z[r] < y[l]$

then $x[l+r] := y[l]$; $l := l - 1$

else $x[l+r] := z[r]$; $r := r - 1$

While $l > 0$ do ; $x[l+r] := y[l]$; $l := l - 1$; Endwhile

While $r > 0$ do ; $x[l+r] := z[r]$; $r := r - 1$; Endwhile

Input: $N \in \mathbb{N}$ and array $x[1 \dots N]$

Output: *Permuted* array $x \circ \pi$

s.t. $x_{\pi(1)} \leq \dots \leq x_{\pi(N)}$

6 5 3 1 8 7 2 4

runtime

$$T(N) = 2 \cdot T(N/2) + O(N) \\ \leq O(N \cdot \log N)$$

memory

$$O(N \cdot \log N)$$

3. Sorting

Quick Sort

Procedure **QuickSort** ($x[]$; l, r)

If $l \geq r$ return. // $x[l] \dots x[r]$ sorted

$s := x[\lfloor (l+r)/2 \rfloor] \in X$ // sample pivot

$a := l$; $b := r$; While $a < b$ do

While $a < b$ and $x[a] < s$ do $a := a+1$ Endwhile;

While $a < b$ and $x[b] \geq s$ do $b := b-1$ Endwhile;

Swap($x[a], x[b]$);

Endwhile ; // $l \leq a = b \leq r$

QuickSort (x , l , $a-1$);

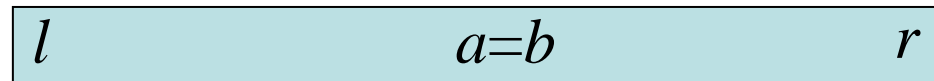
QuickSort (x , $b+1$, r);

Idea: $s := x[p]$ for **random** $p \in [l..r]$

Input: $N \in \mathbb{N}$ and array $x[1 \dots N]$

Output: *Permuted* array $x^{\circ \pi}$
s.t. $x_{\pi(1)} \leq \dots \leq x_{\pi(N)}$

$$T(N) \leq O(N) + T(N-2) + T(2) = O(N^2)$$



QuickSort is in worst-case as bad as BubbleSort

→ *randomized* algorithms

3. Sorting

continued: Quick Sort

Procedure **QuickSort** ($x[] ; l, r$)

Input: $N \in \mathbb{N}$ and array $x[1 \dots N]$

If $l \geq r$ return. // $x[l] \dots x[r]$ sorted

Output: *Permuted* array $x \circ \pi$

$s := x[\lfloor (l+r)/2 \rfloor] \in X$ // sample pivot

s.t. $x_{\pi(1)} \leq \dots \leq x_{\pi(N)}$

$a := l ; b := r ;$ While $a < b$ do

$$T(N) = T(N \cdot \varepsilon) + T(N \cdot (1-\varepsilon)) + O(N)$$

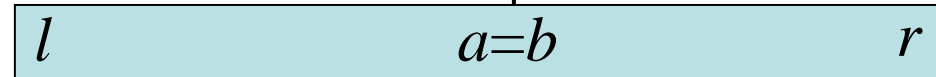
While $a < b$ and $x[a] < y$ do $a := a+1$ Endwhile;

$$\leq O(N \cdot \log N)$$

if $\varepsilon \in (0,1)$ *fixed*

While $a < b$ and $x[b] \geq y$ do $b := b-1$ Endwhile;

Swap($x[a], x[b]$);



Endwhile ; Return a ;

QuickSort ($x , l , a-1$);

$$\varepsilon \cdot (r-l+1) \leq a-l \leq (1-\varepsilon) \cdot (r-l+1)$$

$$\varepsilon \cdot (r-l+1) \leq r-a \leq (1-\varepsilon) \cdot (r-l+1)$$

QuickSort ($x , b+1 , r$);

$$c \cdot N \cdot \log(N) =: T(N) = c \cdot N \cdot \varepsilon \cdot \log(N \cdot \varepsilon) + c \cdot N \cdot (1-\varepsilon) \cdot \log(N \cdot (1-\varepsilon)) + N$$

Ansatz $= c \cdot N \cdot \log(N) + N \cdot c \cdot (\varepsilon \cdot \log(\varepsilon) + (1-\varepsilon) \cdot \log(1-\varepsilon)) + N$

3. Sorting

Approximate Median

Martin

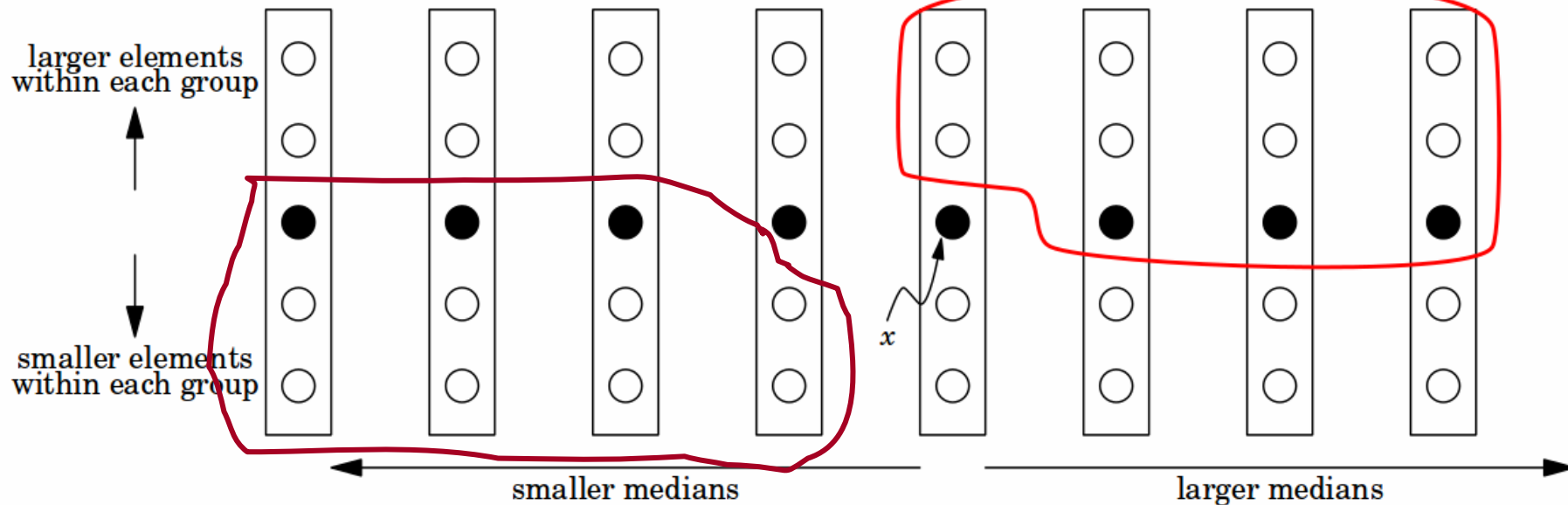
Specification: Fix set X with total order \leq .

Input: $N \in \mathbb{N}$ and finite sequence $x_1, \dots, x_N \in X$

unsorted!

in array $x[1..N]$

Output: m such that $0.3 \cdot N \leq \#\{n : x_n \leq x_m\} \leq 0.7 \cdot N + 1$



Lemma: Median of 5-medians is

“ \geq ” at least $\frac{1}{2} \cdot \frac{3}{5} = 30\%$ of entries, thus “ $<$ ” at most 70%.

3. Sorting

Linear-time Median

Martin

Specification: Fix set X with total order \leq .

unsorted!

Input: $N \in \mathbb{N}$ and finite sequence $x_1, \dots, x_N \in X$

in array $x[1..N]$

Output: m such that $0.3 \cdot N \leq \#\{n : x_n \leq x_m\} \leq 0.7 \cdot N + 1$

Function **ApproxMedian** ($x[], l, r$)

$$T_O(n) = O(n) \quad T_A(n) = O(n)$$

Process $x[l..r]$ in groups of 5, sorting each one to find its median.

Then call **OrderStat** to determine the median of these 5-medians.

Function **OrderStat** ($x[], l, r, K$)

$$T_A(n) = O(n) + T_O(0.2 \cdot n),$$

While $l < r$ do

$$O(n), n := r - l + 1$$

$$T_O(n) = T_A(n) + O(n) + T_O(0.7 \cdot n)$$

Call **ApproxMedian**: determine *approx.* median m of $x[l..r]$.

$m' := \text{Split}(x, l, r, x[m])$ s.t. all elements $< x[m]$ are left of those $> x[m]$

Proceed to the left (=decrease r) /right (=increase l) accordingly.

3. Sorting

Specification: Fix set X with total order \leq .

Input: $N \in \mathbb{N}$ and array $x[1 \dots N]$ with values in X

Output: *Permutation* π s.t. $x_{\pi(1)} \leq \dots \leq x_{\pi(N)}$

Primitives

“ $x[\pi[n]] \leq x[\pi[m]]$?”

Swap $\pi[n] \leftrightarrow \pi[m]$

Index arithmetic

Definition: A *Decision Tree* for sorting N elements is a binary tree

whose nodes are labeled

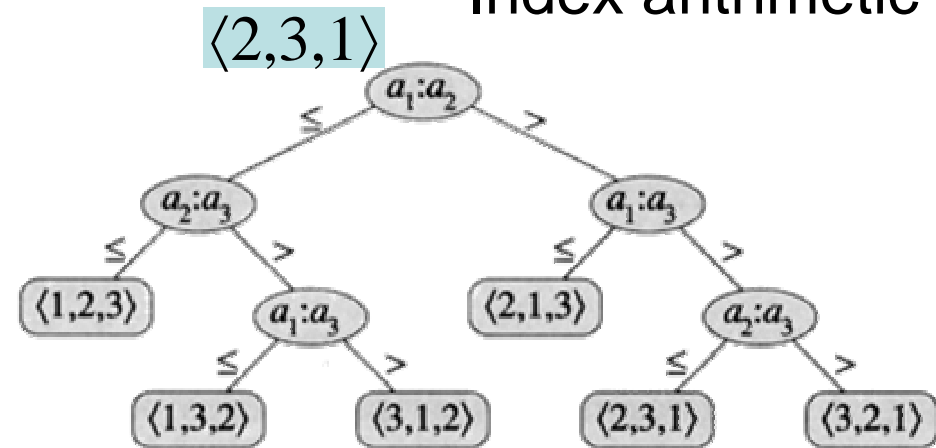
“ $x[i] \leq x[j]$?”, $1 \leq i < j \leq N$.

and whose leaves are labeled with permutations π such that

every input $x = x[1 \dots N]$

ends up in a leaf

whose label π satisfies $x[\pi[1]] \leq \dots \leq x[\pi[N]]$.



3. Sorting

Optimality

Lemma: a) For fixed N , every *comparison-based* sorting algorithm of worst-case runtime $T(N)$ can be “unrolled” into a decision tree for sorting N elements of depth $\leq T(N)$.
 b) Every permutation $x=\pi$ ends up in the unique leaf with label π^{-1} .
 c) A binary tree with $N!$ leaves has depth $\geq \log(N!)=\Theta(N \cdot \log N)$

Definition: A *Decision Tree* for sorting N elements is a binary tree whose internal nodes are labeled “ $x[i] \leq x[j]?$ ”, $1 \leq i < j \leq N$.

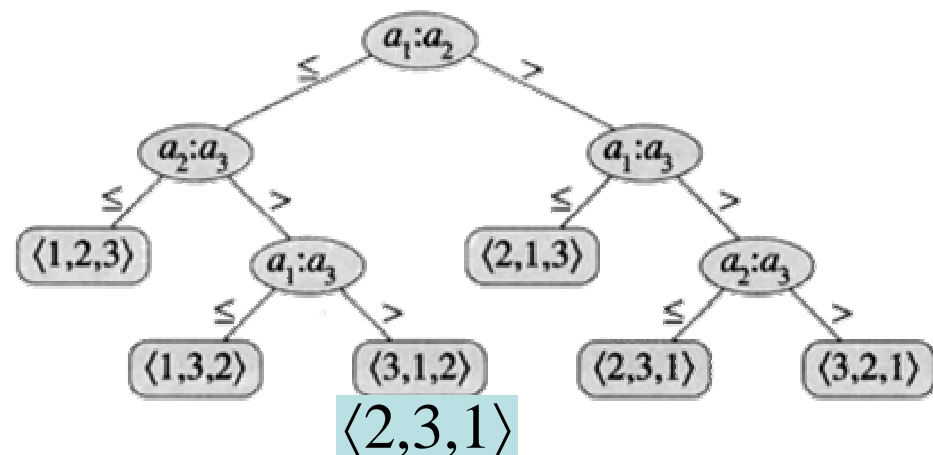
and whose leaves are labeled with permutations π .

Theorem: Any comparison based sorting algorithm requires time at least $\Omega(N \cdot \log N)$.

“ $x[\pi[n]] \leq x[\pi[m]]?$ ”

Swap $\pi[n] \leftrightarrow \pi[m]$

Index arithmetic



3. Sorting

Counting Sort

Specification: Fix set $X=\{1,\dots,M\}$!

Input: $N \in \mathbb{N}$ and array $x[1\dots N]$ with **key** values in X

Output: *Permuted* array $y=x \circ \pi$ s.t. $x.key[\pi[1]] \leq \dots \leq x.key[\pi[N]]$

integer array $count[1\dots M]$;

For $m:=1$ to M do $count[m] := 0$;

For $n:=1$ to N do $count[x.key[n]]++$;

For $m:=2$ to M do

$count[m] += count[m-1]$;

Endfor;

For $n:=1$ to N do

$y[count[x[n].key]--] := x[n]$;

runtime
 $O(N+M)$

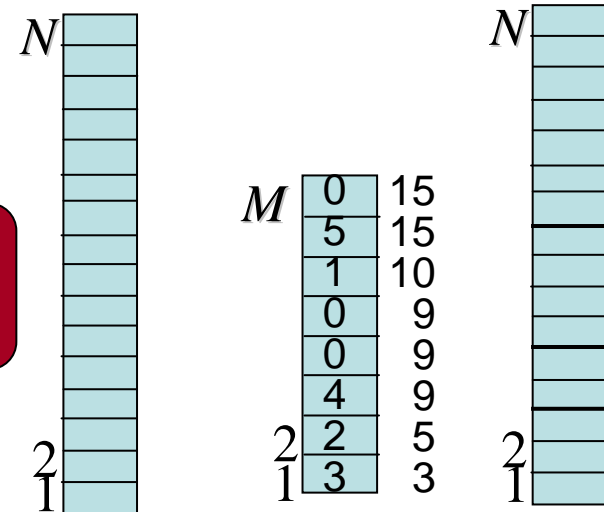
Bit-cost:
 $O(N \cdot \log N + M \cdot \log M)$

memory
 $O(N+M)$

~~“ $x[\pi[n]] \leq x[\pi[m]]$?”~~

~~Swap $\pi[n] \leftrightarrow \pi[m]$~~

~~Index arithmetic~~

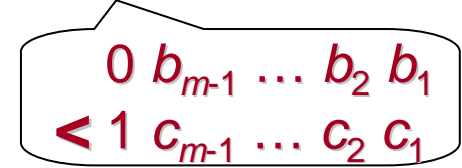


Radix Sort

3. Sorting

Specification: Fix set $X = \{0, 1\}^m$ with **lexicographical** order.

Input: $N \in \mathbb{N}$ and array $x[1 \dots N]$ with values in X



Output: *Permuted* array $y = x \circ \pi$ s.t. $y[1] \leq \dots \leq y[N]$

Quick-/Mergesort
Bit-cost: $O(N \cdot \log N \cdot m)$

~~" $x[n] \leq x[m]$?"~~
~~Swap $x[n] \leftrightarrow x[m]$~~
 Index arithmetic

```

RadixSort( x[], l, r, m );
// Sort x[l...r] w.r.t. bits #m...#1
If l=r or m<1 then return;
// Put all entries with 0 as bit #m before those with 1:

```

```

mid := Split ( x[] , l , r , m);
RadixSort( x[] , l , mid , m-1);
RadixSort( x[] , mid+1 , r , m-1);

```

$O(N \cdot m)$ operations

Bit-cost: $O(N \cdot m^2)$

3. Sorting *in Parallel* **Sorting Networks**

Specification: Fix set X with total order \leq .

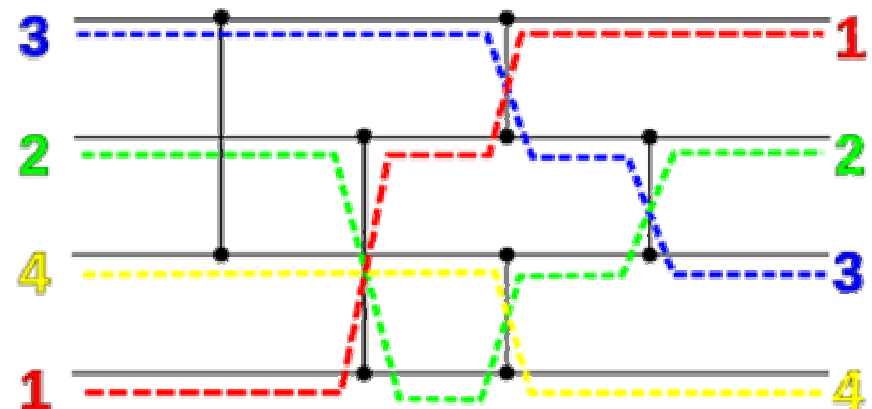
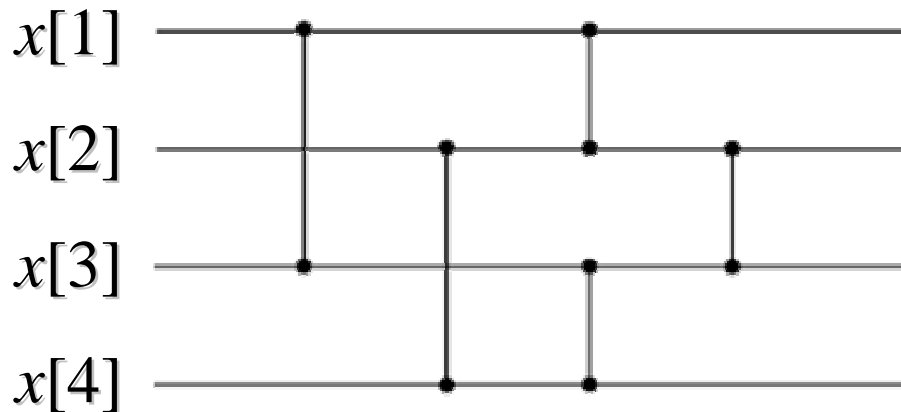
Input: $N \in \mathbb{N}$ and values $x[1 \dots N]$ in X

Output: *Permuted* array $y = x \circ \pi$ s.t. $x[\pi[1]] \leq \dots \leq x[\pi[N]]$



one primitive: gate
 “If $x[n] \leq x[m]$ then
 swap $x[n] \leftrightarrow x[m]$ ”

$N=4$:



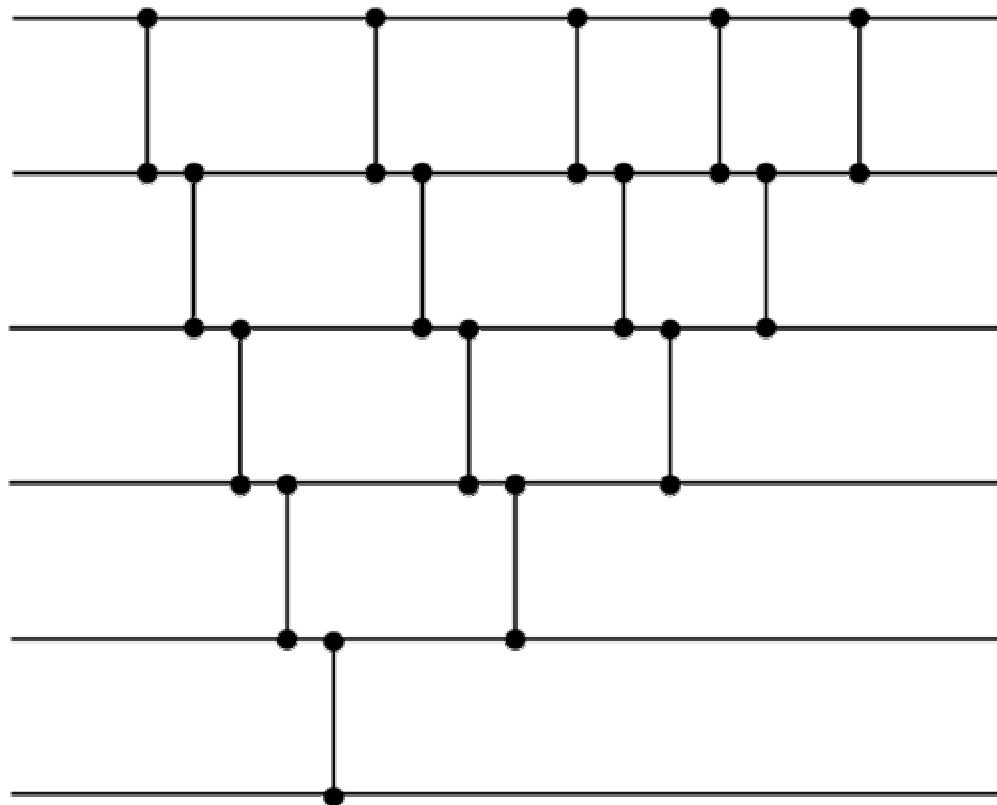
3. Sorting *in Parallel*

Sorting Networks (continued)

Specification: Fix set X with total order \leq .

Input: $N \in \mathbb{N}$ and values $x[1 \dots N]$ in X

Output: *Permuted* array $y = x \circ \pi$ s.t. $x[\pi[1]] \leq \dots \leq x[\pi[N]]$



one primitive: gate

“**If** $x[n] \leq x[m]$ **then**
swap $x[n] \leftrightarrow x[m]$ ”

Bubble Sorting Network:

Size $O(N^2)$,

Depth (=parallel time)
 $O(N)$

3. Sorting *in Parallel*

Sorting Networks (continued)

Specification: Fix set X with total order \leq .

Input: $N \in \mathbb{N}$ and values $x[1..N]$ in X

Output: *Permuted* array $y = x \circ \pi$ s.t. $x[\pi[1]] \leq \dots \leq x[\pi[N]]$

one primitive: gate

Theorem (without proof):

“**If** $x[n] \leq x[m]$ **then**

swap $x[n] \leftrightarrow x[m]$ ”

a) If a sorting network correctly sorts all sequences $x[1..N]$ with **values in $\{0,1\}$** ,

then it correctly sorts all sequences with **values in X** .

b) There exist sorting networks of size $O(N \cdot \log N)$ and depth $O(\log N)$.

Bubble Sorting Network:

Size $O(N^2)$,

Depth (=parallel time)

$O(N)$

This is optimal

Summary

3. Sorting

- Specification
 - Bubble Sort
 - Selection Sort
 - Insertion Sort
 - Merge Sort
 - Quicksort
 - Pivoting
 - Sorting in Linear Time
 - Sorting in Parallel
- Specification
 - Primitives:
semantics and cost
 - Design
 - Analysis
 - Optimality
- of Comparison-
Based Sorting**

Summary of §2 Searching

- 2. Searching
 - Linear Search
 - Neighbor Search
 - Binary Search
 - Uniqueness
 - Hashing
 - Order Statistics/
Median **in linear time!**
 - 1D Range Counting/Reporting
 - 2D Range Counting/Reporting
- Specification
- Primitives:
semantics and cost
- Design
- Analysis
- (Optimality)

Summary of §1 Introduction

(Computational Problems/Algorithmic Solutions)

- Power of Abstraction
- “Virtues” of Computer Science:
 - Specification
 - Algorithm Design (as opposed to Coding)
 - and Analysis (correctness, asymptotic cost)
 - Optimality (Example: *Square and Multiply*)
- Impact of primitive operations, semantics, cost:
 - Four algorithms computing Fibonacci Numbers
- Mathematics: Recurrences and the *Master Theorem*
- Polynomial Multiplication: Long, Karatsuba, Toom, Cook
- Matrix Multiplication: complexity as open problem